mamba Documentation

Release 0.3.5

Oscar Campos

January 05, 2016

Contents

Welcome to the official Mamba project documentation. Mamba is a high-level rapid application development framework (RAD) developed using the Python language and licensed under the terms of the GPL v3 from the Free Software Foundation.

CHAPTER 1

Using Mamba:

1.1 What Mamba is

A part of a genus of *tree snakes*, Mamba is a rapid application development (RAD) framework built on top of Twisted, Storm and Jinja2 templating system.

Mamba is intended to be used for web applications development but can be used to develop any kind of application that we are able to develop using Twisted because Mamba is just Twisted.

1.1.1 The Mamba Project

As Mamba is based on Twisted, it is itself a web server so you don't need additional software like Apache web server to run Mamba applications.

Mamba is available under the GNU General Public License.

1.1.2 Download

Take a look at the Download page in the Mamba web site.

1.1.3 People behind Mamba

Mamba is maintained and developed by a team of volunteers and no company, organization or foundation controls it in any way. You can see a list of people that make mamba possible at the The Mamba Team page.

1.2 Mamba installation guide

Mamba is written in the Python programming language and supports only version 2.7 of the language (some Mamba components do not support Python 3.x yet). Mamba also need a database server (SQLite, MySQL, MariaDB or PostgreSQL are currently supported) in order to create and use schemas through Storm ORM. For HTML rendering, Mamba uses the Jinja2 templating system.

In order to execute Mamba tests suite doublex and PyHamcrest are required.

To build the documentation, Fabric must be installed on the system.

1.2.1 Installation Step

1. ***Dependencies***
    (a) *Mandatory Dependencies*
    (b) *Optional Dependencies*
2. ***Installing Mamba***
    (a) *The easy way and recommended way: PyPI - the Python Package Index*
    (b) *Living on the edge*
3. *Using Mamba*

1.2.2 Dependencies

These are the Mamba framework dependencies

Mandatory Dependencies

The following dependencies must be satisfied to install mamba.

- Python, version >= 2.7 <= 2.7.5 (3.x is not supported)
- Twisted, version >= 10.2.0
- **|mamba-storm|_**, version >= 0.19
- zope.component
- transaction
- Jinja2, version >= 2.4

Is pretty possible that you also need a database manager and the corresponding Python bindings for it. The database can be either SQLite, MySQL, MariaDB (recommended) or PostgreSQL (recommended).

For SQLite database

As you're using Python 2.7, SQLite should be already built in. This may not be true if you compiled Python interpreter yourself, in that case make sure you compile it with –enable-loadable-sqlite-extensions option.

If you are using PyPy, SQLite should be always compiled and present in your installation.

For MySQL and MariaDB databases

The MySQLdb driver should do the work for both database managers.

For PostgreSQL database

The psycopg2 driver is our target for PostgreSQL databases if we are using the CPython interpreter

If you are using PyPy as your interpreter you need to install psycopg2ct instead. Psycopg2ct is a psycopg2 implementation that uses ctypes and is just what we want to do the job in PyPy.

> **Warning:** Versions of psycopg2 (CPython) higher than 2.4.6 doesn't work with Storm so you have to be sure to install a version lower than 2.5 that is the current version as May 2013

Optional Dependencies

The following dependencies must be satisfied if we are planning on running Mamba tests, building the documentation yourself or contributing with the Mamba project

- doublex, version >= 1.5.1
- PyHamcrest
- Sphinx, version >= 1.1.3
- Fabric
- virtualenv
- pyflakes
- PEP-8

1.2.3 Installing Mamba

There are three ways to install mamba in your system.

The first one is install all the Mamba dependencies like any other software: downloading it from sources, precompiled binaries or using your distribution package manager.

The second one is using `pip` or `easy_install` as:

```
$ sudo pip install mamba-framework
```

The easy way and recommended way: PyPI - the Python Package Index

The third one is using virtualenv to create a virtual environment for your Mamba framework installation and then using `pip` on it:

```
$ virtualenv --no-site-packages -p /usr/bin/python --prompt='(mamba-python2.7) ' mamba-python2.7
$ source mamba-python2.7/bin/activate
$ pip install mamba-framework
$ pip install MySQL-Python
```

Or if you prefer to use `virtualenvwrapper`:

```
$ mkvirtualenv --no-site-packages -p /usr/bin/python --prompt='(mamba-python2.7) ' mamba-python2.7
$ pip install mamba-framework
$ pip install MySQL-Python
```

We recommend the use of `virtualenvwrapper` in development environments as it is cleaner and easier to maintain.

Living on the edge

If you like to live in the edge you can clone Mamba's GitHub repository and use the `setup.py` script to install it yourself:

```
$ git clone https://github.com/PyMamba/mamba-framework.git
$ cd mamba-framework
$ mkvirtualenv --no-site-packages -p /usr/bin/pypy --prompt='(mamba-dev-pypy) ' mamba-dev-pypy
$ pip install -r requirements.txt
$ ./tests
$ python setup.py install
```

Warning: The Mamba GitHub repository is under heavy development, we do not guarantee the stability of the Mamba in-development version.

1.2.4 Using Mamba

Once you have Mamba installed in your system, you should be able to generate a new project using the `mamba-admin` command line tool.

Enjoy it!

1.3 Getting Started

Ready to get started? This is a section for the impatient, and give you a very basic introduction about Mamba. If you are looking for detailed information about how to contribute with Mamba go to *Contributing to Mamba* page. If you're looking for Twisted documentation, just click on the link to go their documentation section.

In this section we are going to create a first dummy Mamba application to get familiar with the *mamba-admin* command line tool and the mamba's MVC model.

1.3.1 Generate the application

First of all, we are going to generate our mamba application using the *mamba-admin* command line tool:

```
$ mamba-admin application --description='Dummy Application' --app-version=1.0 --logfile='service.log
Creating dummy directory...                                          [Ok]
Generating application/ directory...                                 [Ok]
Generating application/controller directory...                       [Ok]
Generating application/model directory...                            [Ok]
Generating application/view directory...                             [Ok]
Generating application/view/templates directory...                   [Ok]
Generating application/view/stylesheets directory...                 [Ok]
Generating twisted directory...                                      [Ok]
Generating twisted/plugins directory...                              [Ok]
Generating static directory...                                       [Ok]
```

```
Generating config directory...                                    [Ok]
Writing Twisted plugin...                                         [Ok]
Writing plugin factory...                                         [Ok]
Writing mamba service...                                          [Ok]
Writing configuration file...                                     [Ok]
Writing favicon.ico file...                                       [Ok]
Writting layout.html template file...                             [Ok]
```

This command generates a new Mamba application directory called **dummy** which contains all the necessary files to start working in our new application. The auto-generated application is already startable so we can just run our application using the **start** Mamba admin subcommand inside the recently generated directory:

```
$ cd dummy
$ mamba-admin start
starting application dummy...                                     [Ok]
```

This will start the Twisted web server in the port 8080. If we redirect our browser to this port we should get a blank page like in the image below.

1.3.2 Adding some HTML content

Now, we are going to add some HTML static content to our new Mamba web application, to do that we have to edit the main Jinja2 template layout.html file in the `view/templates` directory.

Note: To get detailed information about Mamba's MVC pattern and directory hierarchy refer to Mamba and the MVC pattern.

When we first open the file we should get a common Jinja2 template file that looks like this:

```
{% extends "root_page.html" %}
{% block head %}
    <!-- Put your head content here, without <head></head> tags -->
{{ super() }}
{% endblock %}
{% block body %}
    <!-- Put your body content here, without <body></body> tags -->
{% endblock %}
{% block scripts %}
    <!-- Put your loadable scripts here -->
{{ super() }}
{% endblock %}
```

The default layout extends `root_page.html` layout that is used internally by Mamba to add scripts and other components in an automatic way into your applications.

Warning: If you want Mamba to include for you all the *mambaerized* CSS and JavaScript files that you added to the `view/scripts` and `view/stylesheets` directory automatically, your layout **must** extend from `root_page.html` template or Mamba will not add any script or CSS file that is present in the directories mentioned above. If you try to use the Jinja2 templating **super()** method in a block that should have been inherited from *root_page.html* you are going to get back an unhandled exception from the Jinja2 templating system.

We are going to add the common HTML elements that all our pages will share in the layout.html template that Mamba generated for us in the previous step. We are going to create an `index.html` template file just for our index page, in this way we can just inherit from our `layout.html` file from whatever other template we add to the site. Add this code to the body block in the `layout.html` file:

```
{% extends "root_page.html" %}
{% block head %}
    <!-- Put your head content here, without <head></head> tags -->
{{ super() }}
{% endblock %}
{% block body %}
    <!-- Put your body content here, without <body></body> tags -->

    {% block navigation %}
    <div class="navigation">
        <ul class="nav">
            <li><a href="/index">Home</a></li>
            <li><a href="/about_us">About us</a></li>
            <li><a href="/contact">Contact</a></li>
        </ul>
    </div>
    {% endblock %}

    {% block content %}
```

```
    {% endblock %}

{% endblock %}
{% block scripts %}
    <!-- Put your loadable scripts here -->
{{ super() }}
{% endblock %}
```

Now we are going to generate our *index* template file using the *mamba-admin* command line tool:

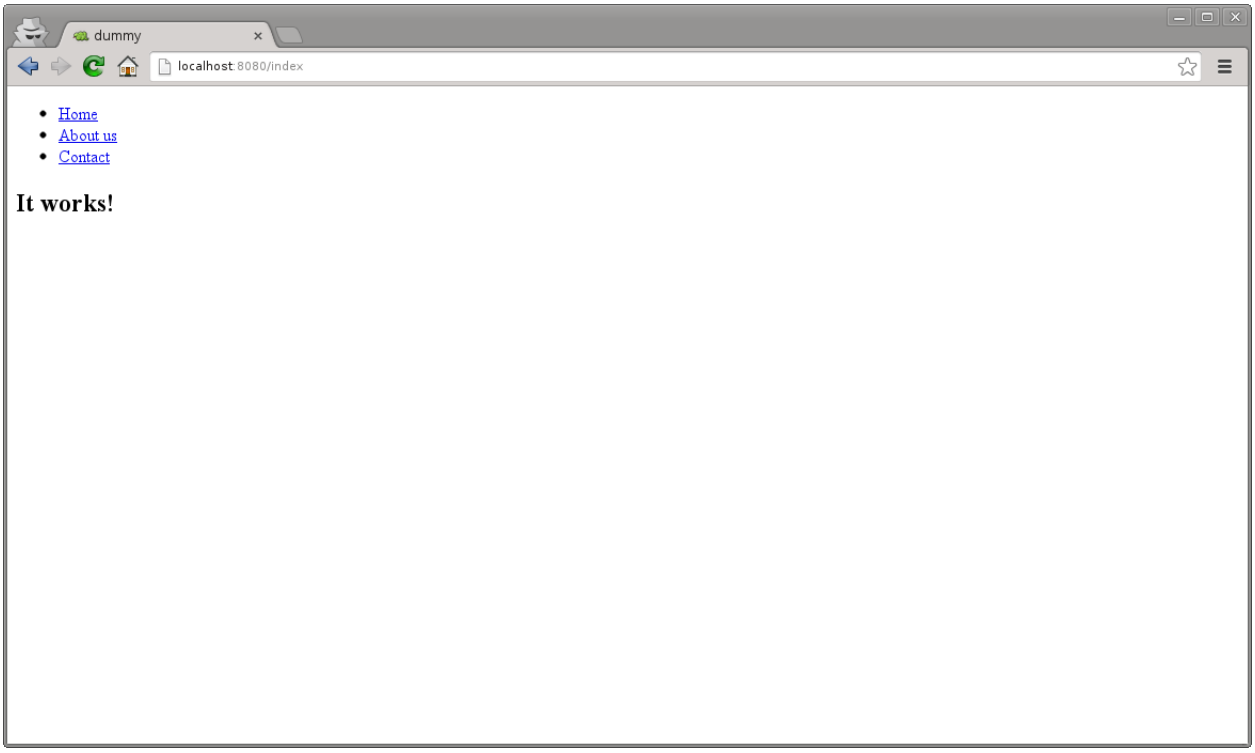
```
$ mamba-admin view --description='Index template for Dummy application' index
```

This will generate a new Jinja2 template file called `index.html` in the `view/templates` directory with the following content:

```
{% extends "layout.html" %}
{% block content %}
{{ super() }}

<!--
    Copyright (c) 2013 - damnwidget <damnwidget@localhost>

    view: Index
        synopsis: Index template for Dummy application

    viewauthor: damnwidget <damnwidget@localhost>
-->

<h2>It works!</h2>

{% endblock %}
```

Note: In your case the copyright and view author information will reflect your environment user configuration, this is pretty OS dependant

If we refresh our browser window we should get the following unstyled HTML on it:



Congratulations, you rendered your first Mamba template sucessfully!. Now we are going to make some changes to the index template and add a CSS file to style a bit our index page:

```
{% extends "layout.html" %}
    {% block content %}
    {{ super() }}

    <!--
        Copyright (c) 2013 - damnwidget <damnwidget@localhost>

        view: Index
            synopsis: Index template for Dummy application

        viewauthor: damnwidget <damnwidget@localhost>
    -->

    <div class="content">
        <h2>Welcome to the Dummy Site!</h2>
        <p>Snakes are so cute aren't it?.</p>
        <img src="http://www.pymamba.com/assets/logo.png" />
    </div>

    {% endblock %}
```

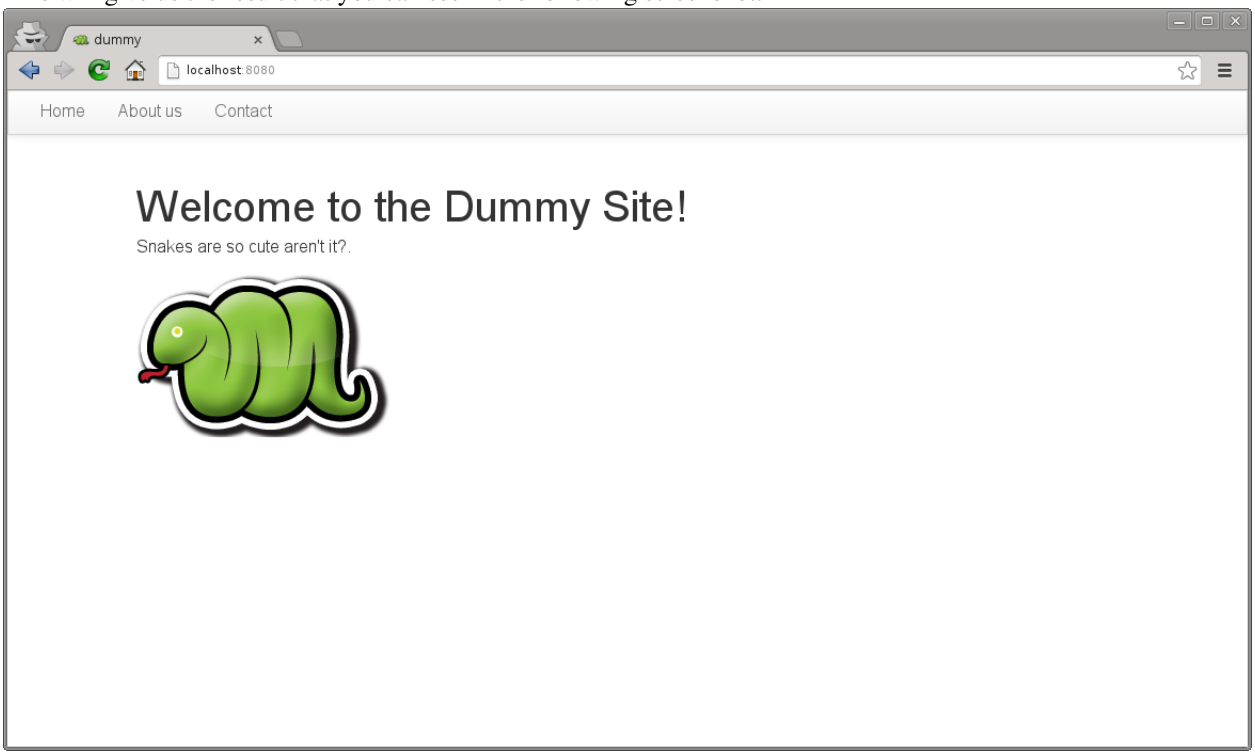
```
/*
 *  -*- mamba-file-type: mamba-css -*-
 */
```

```
body {
    background-color: #fff;
    color: #333;
    display: block;
    font-family: "Helvetica Neue", Helvetica,Arial,sans-serif;
    font-size: 16px;
    line-height: 20px;
    margin: 0;
    padding-top: 40px;
    position: relative;
}

a {
    color: #717171;
}

.navigation {
    content: "";
    background-color: #fafafa;
    background-image: linear-gradient(to bottom, #fff, #f2f2f2);
    background-repeat: repeat x;
    border: 1px solid #d4d4d4;
    box-shadow: 0 1px 10px rgba(0,0,0,0.1);
    line-height: 0;
    left: 0;
    margin-bottom: 0;
    min-height: 40px;
    position: fixed;
    right: 0;
    top: 0;
}

.nav {
    display: block;
    float: left;
    left: 0;
    list-style: none;
    margin: 0 10px 0 0;
    padding: 0;
    position: relative;
}

.nav li {
    display: list-item;
    float: left;
    line-height: 20px;
    margin-left: 30px;
    margin-top: 8px;
}

.nav li a {
    text-decoration: none;
}

.nav li a:hover {
    color: #aab212;
}
```

```
.content {
    margin: 20px auto;
    width: 920px;
}

.content h2 {
    font-size: 40px;
    margin: 60px 0 10px;
    font-weight: 200;
}
```

Note: Mamba CSS files should add the `-*- mamba-file-type: mamba-css -*-` special comment to be automatically loaded by mamba on startup

This will give us the result that you can see in the following screenshot:



Our web site is starting to look like a real one, but if we click in the *About Us* or *Contact* links we will get blank page with an error message saying **No Such Resource**. This is because we didn't add any template or controller to *about_us* or *contact* routes.

Mamba allow us to use views directly without the need of a controller. This way, we can add just static sections into our web site without any controller overhead. We are going to add a new static template for the about us section:

```
$ mamba-admin view --description='About us static template for Dummy application' about_us
```

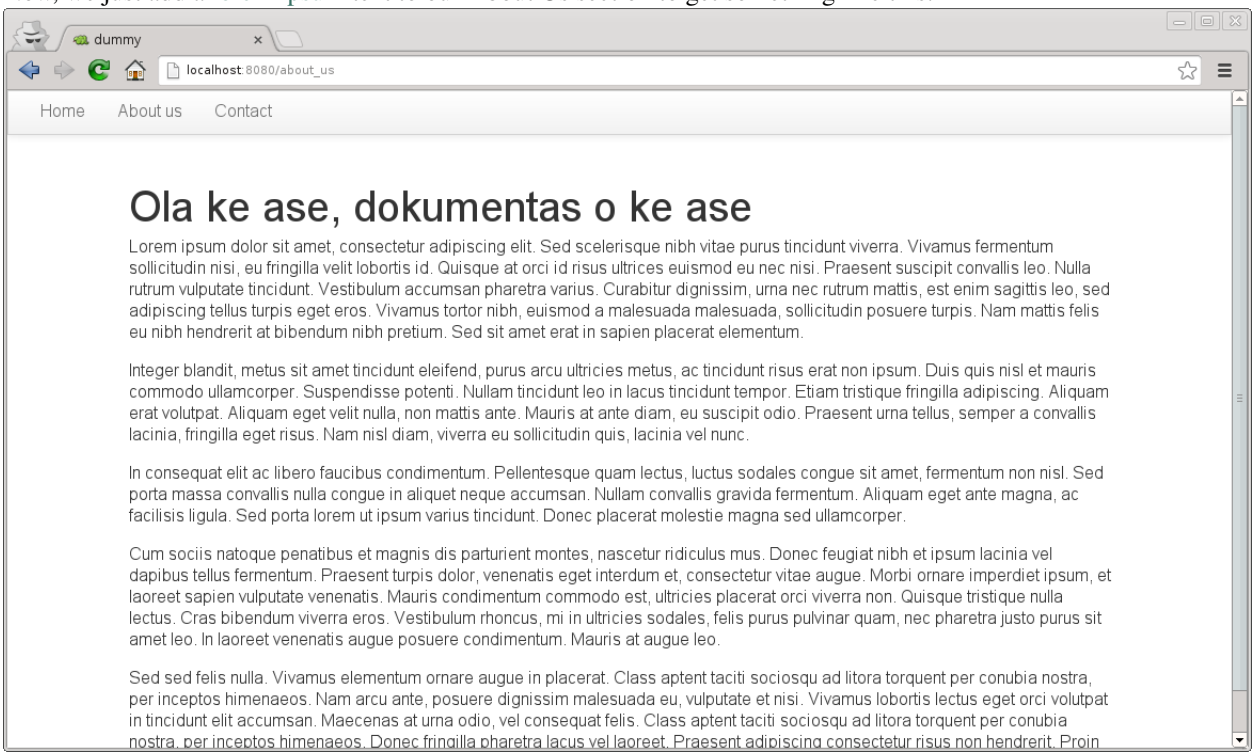
This will create a new file in the `application/view/templates` directory called *about_us.html*. If we click again in the *About Us* link we should get the **It works!** default template message.

At this point maybe you are thinking that the templates directory is kinda `htdocs` directory in a traditional web server like *Apache* but that's not true. Mamba will render any HTML file or Jinja2 template that exists in your *templates* directory but will be unable to find any other media, stylesheet or JavaScript file that is stored in this directory. All the static data that we want to access within our templates must be placed in the *static* directory in the *root* of your application or in *mambaerized* files (files with the right mamba header string) within `view/stylesheets` and `view/scripts` directories.

This is because the HTML rendering is performed internally by the Mamba templating system. It doesn't know anything about files that are not HTML, Jinja2 templates or mamberized cascading stylesheets and JavaScript files. This way, we can mix static HTML data and controllers in the same application sharing the same static resources between them.

Warning: Be aware of dragons: Mamba take care of automatically adding CSS and Javascript files that are *mambaerized* in the `view/stylesheets` and `view/scripts` directories into your templates but **will not** do it for the files you place in the static directory

Now, we just add a lorem ipsum text to our About Us section to get something like this:



1.3.3 Adding our first controller

Now we are ready to add our first controller. To do that - yes, you guessed it - we are going to use the *mamba-admin* command line tool. We gotta know a couple of things about controllers before diving into adding one:

- **Registering Routes** Mamba controllers can be attached to static routes using the `--route` parameter (or editing the __route__ property in the controller object) so all the entry points that this controller adds should share the same parent route. For example: if the register route is `api` and we have two methods `login` and `logout` the full URL route will be:

```
        http://localhost/api/login
        http://localhost/api/logout
```

- **Controllers are Twisted Resources** Controllers in Mamba are just special Twisted resources that are *mambaerized* for being loaded (and reloaded on changes if you are running the Mamba server on Linux) automatically on server startup as well as other custom Mamba features. One of those custom features is the Mamba's routing system. In Mamba we don't add childs to Twisted resources that have been already added as childs to the `Site` object. In Mamba we use routes as we do in Flask or Bottle:

```
        ....
        @route('/status', method='GET')
        def status(self, request, **kwargs):
            """Just return a string indicating the status of dummy
            """
            return dummy.get_status()
```

  Mamba is meant to be flexible enough to allow the programmer to use whatever they can already use with `twisted.web` component so the user is allowed to add `twisted.web.resource.Resource` objects as childs on controllers that has configured their `isLeaf` property to `False`, but we recommend using `twisted.web` directly and use Mamba as external library if you need some Mamba functionallity that is not directly related with rendering the web site.

Our first (and unique) controller is going to be the `contact` one. To generate it, we can use the `mamba-admin` command line tool:

```
$ mamba-admin controller --description='Contact form for Dummy' --route='contact' contact
```

This will create a new file called `contact.py` in the `application/controller` directory, that should look like this:

```
# -*- encoding: utf-8 -*-
# -*- mamba-file-type: mamba-controller -*-
# Copyright (c) 2013 - damnwidget <damnwidget@localhost>

"""
.. controller:: Contact
    :platform: Linux
    :synopsis: Contact form for Dummy

.. controllerauthor:: damnwidget <damnwidget@localhost>
"""

from mamba.web.response import Ok
from mamba.application import route
from mamba.application import controller


class Contact(controller.Controller):
    """
```

```
    Contact form for Dummy
    """

    name = 'Contact'
    __route__ = 'contact'

    def __init__(self):
        """
        Put your initialization code here
        """
        super(Contact, self).__init__()

    @route('/')
    def root(self, request, **kwargs):
        return Ok('I am the Contact, hello world!')
```

At this point - if we are on GNU/Linux - the controller has been automatically loaded by the already-running Mamba server and we can show the resulting page clicking in the *Contact* link in our fashion web site, otherwise we have to stop the server and start it again to see the changes:

```
$ mamba-admin stop && mamba-admin start
```

Note: You can also use `mamba-admin restart`.

That's cool but we have to add some HTML to this so we are going to add a new view for this controller using - wait for it - the `mamba-admin` command line interface:

```
$ mamba-admin view --description='Contact view for contact controller on Dummy' root contact
```

As you can see, we've added a new parameter to our `view` subcommand that tells Mamba that this view is using the `contact` controller. In this occasion the `mamba-admin` command has created a new directory called `contact` in `application/view` and inside this one a new file called `root.html` has been generated (as the root method for `/`) route in the controller.

Note: If we have a static template called `contact.html` in the `templates` directory, it will be overwritten by the new controller template.

Note: Controller views are per route so you need a view for every route that need to render HTML directly to the browser.

Now we have to modify our controller a bit in order to make it use the new template file. First, we are going to import the `templating` module from the `mamba.core` package, then we must create a new *`Template`* object and pass the controller to it, we are going to do that in the controller constructor and render the template later as the response from the `root` method:

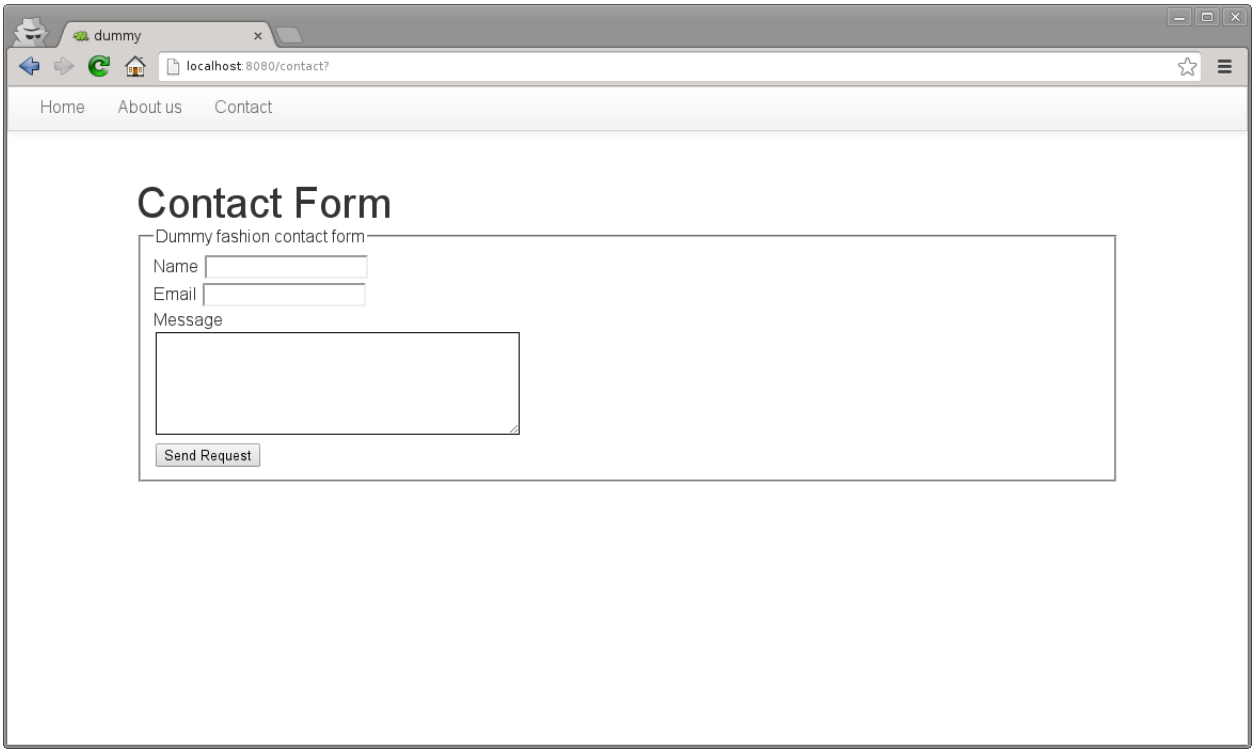
```
# -*- encoding: utf-8 -*-
# -*- mamba-file-type: mamba-controller -*-
# Copyright (c) 2013 - damnwidget <damnwidget@localhost>

"""
.. controller:: Contact
    :platform: Linux
    :synopsis: Contact form for Dummy

.. controllerauthor:: damnwidget <damnwidget@localhost>
"""

from mamba.core import templating
from mamba.web.response import Ok
from mamba.application import route
from mamba.application import controller


class Contact(controller.Controller):
    """
    Contact form for Dummy
    """

    name = 'Contact'
    loaded = False
```

```
    __route__ = 'contact'

    def __init__(self):
        """
        Put your initialization code here
        """
        super(Contact, self).__init__()

        self.template = templating.Template(controller=self)

    @route('/', method='GET')
    def root(self, request, **kwargs):
        return Ok(self.template.render().encode('utf-8'))
```

Let's add some HTML to build our dummy form:

```
{% extends "layout.html" %}
    {% block content %}
    {{ super() }}

    <!--
        Copyright (c) 2013 - damnwidget <damnwidget@localhost>

        view: ContactForm
            synopsis: Contact view for contact controller on Dummy

        viewauthor: damnwidget <damnwidget@localhost>
    -->

    <div class="content">
        <h2>Contact Form</h2>
        <form>
            <fieldset>
                <legend>Dummy fashion contact form</legend>
                <label>Name</label>
                <input id="name" type="text" required><br />
                <label>Email</label>
                <input id="email" type="email" required><br />
                <label>Message</label><br />
                <textarea id="content" rows="6" cols="40">
                </textarea><br />
                <button type="submit">Send Request</button>
            </fieldset>
        </form>
    </div>

    {% endblock %}
```

If we restart (if not on GNU/Linux) the server and go to our `contact` page we should get this fancy form:



1.3.4 The End

And we have reached the end of this basic introduction to Mamba framework, there is a lot more to discover about Mamba's features but we hope you have now an idea of the backbone of the framework.

Of course all the files that we created with the `mamba-admin` command line tool can be created by hand and it should work as expected. If you want to see a real world Mamba application, we suggest you to visit the BlackMamba GitHub repository.

1.4 Mamba development guide

Here you can find any resource you need to learn howto use mamba as your web applications framework.

1.4.1 Mamba and the MVC pattern

According to Wikipedia; *Model View Controller (MVC) is a software architecture, currently considered as an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from input and presentation (GUI), permitting independent development, testing and maintenance of each.*

Mamba implements the MVC pattern using Jinja2 templates as the **view**, then Mamba components act as the **model** and the **controller**. In Mamba, the routing system is integrated into the controller itself.

Is Mamba meant to be a real MVC implementation?

That depends on the interpretation of the paragraph above. Mamba applications are implemented using a MVC like pattern as we separate our business logic from the view (that only knows about render HTML), the controllers receive inputs as HTTP requests and initiates responses by calling methods in model objects that interacts with different data sources, interpret that data and send results back to the view through the controller (or send it through JSON or sockets to third parties).

The last description can be defined as a MVC pattern implementation or not depending on the point of view. Anyway, Mamba isolates the logic of the data from the presentation of the data, that's all that you are going to want to take care of.

1.4.2 Standard Mamba application layout

A Mamba standard application layout is as follows:

```
application                      → Application package
  controller                    → Application controllers
  model                         → Application models
  view                          → Application templates and scripts
     stylesheets                → CSS/LESS files
     scripts                    → JavaScript/Dart files
     templates                  → Jinja2 Templates
  lib                           → General library dependencies that are not part of the MVC and 3rd p
config                           → Configuration files
  application.json              → Application main configuration file
  database.json                 → Application database configuration file (if applicable)
  installed_packages.json       → Application installed shared packagees (if applicable)
docs                             → Application documentation directory
static                           → Application assets durectory
test                             → Application tests directory
twisted                          → Twistd plugin for twisted daemonizer
logs                             → Application logs directory
LICENSE                          → LICENSE file
README.rst                       → README file
app_name.py                      → Application initialization file
mamba_services.py                → File used internally by mamba to perform several tasks on applicat
```

The application directory

The `application` directory contains all our application Python code (with the exception of the shared packages and the `ApplicationFactory`)

The `application` directory contains three directories to implement the MVC pattern and a fourth one where to place all the code that doesn't fit the MVC pattern and 3rd party libraries as well, all of them are Python packages:

- application/controller → Python package
- application/model → Python package
- application/view → Python package
- application/lib → Python package

The `application` directory is a package itself, meaning it contains an `__init__.py` file so you can add whatever other directory/package/module that you need. The application directory and all its contents are exported by default when you *pack* or install your application.

The config directory

The `config` contains the application configuration files. Those files **must** be valid **JSON** formatted files. There are two main configuration files on mamba:

- `application.json`, this is the main configuration file for the application, if you need to add some configurable parameter to your application, this is the file to place it
- `database.json`, this file is used to configure database connections and it parameters

A third file is used to tell Mamba that we want to include some Mamba shared package in our application:

- `installed_packages.json`, this file contains a list of installed Mamba shared *packages* that we want to use in our application

If you need to add some custom configuration file, you shoud place it inside this directory to follow convention.

The docs directory

The `docs` directory is used to store the application documentation. We use Sphinx as documentation system but you can use whatever you want.

The static directory

The `static` directory is used to store mainly images and other static data. You can use it to store and access CSS, JavaScript or HTML static files but is better if you do that using the templating system.

> **Warning:** All the files that you place in the static directory is publicly accessible from internet

The test directory

The `test` directory contains your application unit tests and integration tests

The twisted directory

Twisted directory is used internally by mamba and Twisted to daemonize your mamba applications, you don't have to care about this dIrectory and itS contents.

The logs directory

Mamba writes the log files in the `logs` directory if you don't configure other behaviour by yourself.

1.4.3 Using SQL databases

Mamba provides access to SQL databases through the `Model` class in asynchronous or synchronous way using the same database connection. Currently, Mamba supports **SQLite**, **MySQL/MariaDB** and **PostgreSQL** databases. In order to connect to a database, you have to configure the connection details in your application `config/database.json` file:

```
{
    "max_threads": 20,
    "min_threads": 5,
    "auto_adjust_pool_size": true,
    "drop_table_behaviours": {
        "drop_if_exists": false,
        "restrict": true,
        "cascade": true
    },
    "uri": "backend://user:password@host/dbname",
    "create_table_behaviours": {
        "create_table_if_not_exists": false,
        "drop_table": false
    }
}
```

Ths database configuration file can be created manually or using the `mamba-admin` command line tool using a valid URI or passing explicit parameters:

```
$ mamba-admin sql configure --uri=backend://user:password@host/dbname
```

Or:

```
$ mamba-admin sql configure --username='user' --password='password' --hostname='hostname' --backend=
```

The above two commands are exactly the same, both of them generates a new database config file with default options and connects to the given backend (can be one of: sqlite, mysql or postgres) with the given user and password credentials in the given host and the given database.

The following is a list of all the options that can be passed to the `mamba-admin sql configure` command:

```
Usage: mamba-admin [options] sql [options] command configure [options]
Options:
  -p, --autoadjust-pool       Auto adjust the database thread pool size?
  -c, --create-if-not-exists  If present, when mamba try to create a new table
                              adds an `IF EXISTS` clause to the SQL query
  -d, --drop-table            If present, mamba will drop any table (if exists)
                              before to create it. Note this option is not
                              compatible with `create-if-not-exists
  -e, --drop-if-exists        If present, mamba will add an `IF EXISTS` clause
                              to any intent to DROP a table
  -r, --non-restrict          If present, mamba will NOT use restrict drop
  -a, --cascade               If present, mamba will use CASCADE in drops
  -n, --noquestions           When this option is set, mamba will NOT ask
                              anything to the user that means it will delete any
                              previous database configuration and will accept
                              any options that are passed to it (even default
                              ones).Use with caution
      --uri=                  The database connection URI as is used in Storm.
                              Those are acceptable examples of format:

                              backend:database_name
                              backend://hostname/database_name
                              backend://hostname:port/database_name
                              backend://username:password@hostname/database_name
                              backend://hostname/database_name?option=value
                              backend://username@/database_name

                              Where backend can be one of sqlite, mysql or
```

```
                            postgres. For example:

                            sqlite:app_db slite:/tmp/tmp_database
                            sqlite:db/my_app_db
                            mysql://user:password@hostname/database_name
                            postgres://user:password@hostname/database_name

                            Note that you can also use --hostname --user and
                            --password options instead of the URI syntax
                            [default: sqlite]

      --min-threads=        Minimum number of threads to use by the thread
                            pool [default: 5]
      --max-threads=        Maximum number of thread to use by the thread pool
                            [default: 20]
      --hostname=           Hostname (this is optional)
      --port=               Port (this is optional)
      --username=           Username which connect to (this is optional)
      --password=           Password to connect (this is optional)
      --backend=            SQL backend to use. Should be one of
                            [sqlite|mysql|postgres] (this is optional but
                            should be present if no URI is being to be used)
                            [default: sqlite]
      --database=           database (this is optional but should be suply if
                            not using URI type configuration)
      --path=               database path (only for sqlite)
      --option=             SQLite additional option
      --version             Show version information and exit
      --help                Display this help and exit.
```

The database URI

Mamba uses a valid URI as parameters to connect with the database.

SQLite URIs

The simplest valid URI that we can use for our mamba application is just the SQLite in-memory database:

```
"uri": "sqlite:"
```

Relative (to the web application root directory) or absolute paths can be used for database name/location, the following are all valid possible sqlite configurations:

```
"uri": "sqlite:foo"
"uri": "sqlite:/home/user/foo"
"uri": "sqlite:///foo"
"uri": "sqlite:///home/user/foo"
```

If the database doesn't exists yet, Mamba will create it when we first try to use it. If the path doesn't exists or is not accessible (e.g. permission denied), an exception `OperationalError` will be raised.

SQLite accepts one option in the option part of the URI. We can set the time that SQLite will wait when trying to obtain a lock on the database. The default value for the timeout is five seconds, an example of the above is as follows:

```
"uri": "sqlite:dummy?timeout=0.5"
```

This will create a new SQLite database connection with a timeout of half a second.

MySQL/MariaDB URIs

MySQL and MariaDB share syntax for URI's:

```
"uri": "mysql://username:password@hostname:port/database_name"
```

Note: MySQL/MariaDB support depends on the MySQLdb Python module

PostgreSQL

Syntax for PostgreSQL is exactly the same than MySQL/MariaDB but replacing the `mysql://` with `postgres://` scheme:

```
"uri": "postgres://username:password@hostname:port/database_name"
```

Note: PostgreSQL support depends on the psycopg2 Python module

Warning: If you are planning to use PyPy as your interpreter, you **must** install psycopg2ct that is an implementation of the psycopg2 module using ctypes

Create or dump SQL schema from Mamba models

In Mamba we don't create a schema config file that is then used to generate our model classes, instead of that, we define our model classes and then we generate our SQL schema using our already defined Python code.

To create our database structure in live or dump a SQL file with the schema (for whatever SQL backend we configured) we use the `mamba-admin sql create` subcommand in the command line interface, so for example to dump the schema into a file we should use:

```
$ mamba-admin sql create schema.sql
```

To dump it to the stdout:

```
$ mamba-admin sql create -d
```

And for create it in live in the database (this may delete all your previous data, be careful):

```
$ mamba-admin sql create -l
```

Note: If you don't want Mamba to generate SQL for a specific(s) table(s), you can set the class-level attribute `__mamba_schema__` to `False`. This will also prevent Mamba of dropping this table or truncating its data when you use the `reset` command.

Dump SQL data from the database

If you ever used `mysqldump` you will be familiarized with `mamba-admin sql dump` command. It dumps the actual data into the database to the stdout. Doesn't matter which database backend you are using, it works with SQLite, MySQL and PostgreSQL and you don't need to have installed `mysqldump` command to dump MySQL databases:

```
$ mamba-admin sql dump > database-dump.sql
```

The above command will dump the database into a file in the current directory named `database-dump.sql`

Truncating all data in your database

Sometimes we need to truncate all tables in our database. For that scenario you can use the `reset` command:

```
$ mamba-admin sql reset --noquestions
```

The above command will reset all your data without any questions. So, please, be careful with this command.

Interactive Shell

In case you want to login into the database interactive shell, you can just issue this command and Mamba will take care of authentication details for you:

```
$ mamba-admin sql shell
```

Future plans

For next releases, a live database migration tool is intended to be added to the framework so the developer can just switch from a RDBMS to another one without losing his data.

1.4.4 The Mamba Storm - Twisted integration

Mamba uses a custom modified version of the Storm's Twisted `@transact` integration added in Storm v0.19. It also creates a `ThreadPool` service on initialization time so you don't have to take care of do it yourself. The *transact* system is quite simple:

> Any model method that is decorated with the `@transact` decorator is executed in a separate thread into the Mamba database thread pool and returns a Twisted deferred object. Any method that is not decorated by `@transact` is just using regular storm features and it can't run asynchronous.

Using the *transact* system has advantages and disadvantages:

- The main advantage is it runs asynchronous so it **doesn't block** the Twisted reactor loop.
- The main disadvantage is that any return value from the decorated (with `@transact`) method must **not** contain any reference to Storm objects because they were retrieved in a different thread and of course can't be used outside it. This put limits in some awesome Storm features that can be used within the decorated method only, `Reference` and `ReferenceSets` for example.

The transact mechanism can be dangerous in environments where serialization or synchronization of the data is a requirement because using the `@transact` decorated methods can end up in unexpected race conditions.

The developer should give some thought about what she or the applications needs to implement, in order to know when to use `@transact` or not.

Some Mamba model operations runs with `@transact` decorated method by default (for example, create, find, read, delete or update) if you need to run those methods in synchronous way you can add the named parameter `async=False` to its call, just like `customer.update(async=False)`. This is pretty common workflow if you're calling these methods from places that are already decorated by the `@transact` decorator.

Note: Some notes about the *read* method. Storm uses lazy evaluation to access objects properties, the *read* method can't guarantee that the object that the it returns can be safely used in another thread, if you need safe access to properties of those objects returned with read you can add the named parameter `copy=True` to the *read* call like `kevin = customer.read(1, copy=True)` and mamba will guarantee the lazy evaluation of the object properties is performed before return the object back. You can also run the call with `async=False` to make sure the object that

is returned by read is created in the same thread that the call is made and then is safe to use it (and lazy evaluation is possible)

Note: If you get the exception `ZStormError("Store not registered with ZStorm, or registered with another thread.")` this means that you are trying to use a Storm object that has been created in another thread (mainly using @trasact decorator) or you are trying to access a property that is lazy evaluated.

Warning: *References* and *ReferenceSets* are always lazy evaluated so there is no safe way to access them from outside the thread that the object has been created.

When you use `@transact` (asynchronously or not), mamba takes care of ensure that your stores are connected to the database reconnecting and rerunning your transactions if the database connection gone away because inactivity or any other problem.

How to use Storm in Mamba models?

Mamba's enterprise system take care of any initialization that is needed by the underlying Storm library, we don't have to care about create database connections or stores. We can use it with CPython or PyPy without any type of code modification.

The Store object

Storm (and Mamba by extension) uses *stores* to operate with the underlying database. You can take a look at the Storm API documentation to retrieve a complete list of Store methods and properties.

The mamba's enterprise system initialize a valid Storm Store object for us always that we need it using the `model` model property `store()` method:

```
store = self.store()
```

Note: If you are planning to use stores outside *transacted* methods wil be a good idea to use the named parameter `ensure_connect=True` to make sure storm is connected to your database before try to use the store.

Every model object has a copy of the `database` object that can be used to retrieve stores and other database related information. Normally we don't want to use the *database* store method directly unless we want to use multiple databases from the same model class (more on that later).

Stores are used to retrieve objects from the database, to insert and update objects on it and of course to execute SQL queries directly to the database. Store is like a traditional `cursor` but much more flexible and powerful.

If we need to create and insert a new row into the database we just instantiate the model object and then add it to a valid store:

```
peter = User()
peter.name = u'Peter Griffin'

store = self.store()
store.add(peter)
```

Once an object is added or retrieved from a store, we can verify if it is bound or related to an store easily:

```
>>> Store.of(peter) is store
True
```

```
>>> Store.of(User()) is store
False
```

If we are using the `@transact` decorator in our methods we don't have to care about commit to the database because that is performed in an automatic way by the `@transact` decorator, otherwise we **must** call the `commit` method of the store object:

```
store.commit()
```

If we made a mistake we can just call the `rollback` method in the same way.

Of course we can use the store object to find rows already inserted on the database. The following is an example of how to use a store to find an user in the underlying database:

```
store = self.store()
user = store.find(User, User.name == u'Peter Griffin').one()
```

We can also retrieve the object using its primary key:

```
pk_user = store.get(Person, 1)
```

Stores caches objects as default behaviour so we can check that `user` and `pk_user` are efectively the same object:

```
>>> pk_user is user
True
```

Each store has an object cache. When an object is linked to a store, it is cached by the store for as long there is a reference to the object somewhere, or when the object becomes dirty (has changes on it). In this way Storm make sure that we don't access to the database when is not necessary to retrieve the same objects again.

Modifiying objects with the Store

We don't have to retrieve an object from the database and then modify and save it, we can just use the Store to do the work for us using expressions:

```
store.find(User, User.name == u'Peter Griffin').set(name='Peter Sellers')
```

How do I use stores in an asynchronous way?

Just decorate your model methods with the `@transact` decorator and make sure to don't return any Storm object from that method:

```
from mamba.application import model
from mamba.enteprise import Int, Unicode, transact


class Dummy(model.Model):

    __storm_table__ = 'dummy'

    id = Int(primary=True)
    name = Unicode()

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

    @transact
```

```
    def get_last(self):
        """
        Get the last inserted row from the database.
        This is not thread safe
        """
        store = self.store()
        return store.find(Dummy).order_by(Dummy.id).last()
```

The `get_last` method above will retrieve the last inserted row in the database. As we are using the `@transact` decorator we can't use `Reference` or `ReferenceSet` in the returned object because those are lazy evaluated and the object was created in a different thread. If we ever try to do that we will get an exception from Storm ZStore module.

If we don't want to use an asynchronous operation we can just remove the `@transact` line and it will work perfectly synchronous, of course the limitations about using references with the returned object does not apply on this scenario.

How do I use a store from outside the model method?

Even if Mamba allows us to use Store objects everywhere, they are not suposed to be used outside the model but nothing stop you to use it in the controller or whatever other part of your application.

If you think that you need to use a store object from outside your model class,then you can do it in several ways:

1. Don't decorate a method in your model with `@transact` and then return the store from it. As this store has been created in the same thread that the rest of the application you can use it anywhere.
2. **Just retrieve a store object executing the `database.store()` object directly from your model at class level:**

```
        from application.model.dummy import Dummy

        store = Dummy.store()
        dummy = store.get(Dummy, 1)
```

> **Warning:** Please, be careful, we recommend energically to don't use stores outside the model. It doesn't follow the MVC pattern and violates the encapsulation principle.

Should I share stores between threads?

No. Everytime that you call the `database.store()` method in the model object, Mamba gives you a ready to use Store for the thread that you are calling the method from. Don't even try to share stores between threads. This means that you are not able to share stores between methods if they are decorated with the `@transact` decorator.

Connecting to more than one database

Mamba allow our models to connect more than one database at the same time, to do that, we have to convert the *uri* database config setting into a dictionary where each key is a connection/database name and the value is the connection *URI*, so for example, if we want to use a PostgreSQL database for our operations but a MySQL to store logs we can do it as follows:

```
{
    "max_threads": 20,
    "min_threads": 5,
    "auto_adjust_pool_size": true,
```

```
    "drop_table_behaviours": {
        "drop_if_exists": false,
        "restrict": true,
        "cascade": true
    },
    "uri": {
      "operations": "postgres://user:password@host/dbname",
      "reports": "mysql://user:password@host/dbname"
    },
    "create_table_behaviours": {
        "create_table_if_not_exists": false,
        "drop_table": false
    }
}
```

Then we have to define which database is our models going to use, we can do it setting the *__mamba_database__* that by default

```
    from mamba.application import model
    from mamba.enteprise import Int, Unicode, transact


    class Dummy(model.Model):

        __storm_table__ = 'dummy'
        __mamba_database__ = 'operations'

        id = Int(primary=True)
        name = Unicode()
        ...
```

The former model will read and write from the *operations* database that has been configured to use our PostgreSQL backend, w

```
    from mamba.application import model
    from mamba.enteprise import Int, Unicode, transact


    class DummyLogs(model.Model):

        __storm_table__ = 'dummy_logs'
        __mamba_database__ = 'reports'

        id = Int(primary=True)
        name = Unicode()
        ...
```

This last model will read and write into our MySQL backend.

Note: If you don't want to define the *__mamba_database__* property in each of your models you can define the key of your *default* database as *mamba* as the models *store* method try to connect to the *mamba* named database by default.

How to connect to different databases from the same model?

Even mamba doesn't provide any out of the box mechanism to connect to multiple databases from the same model, this can be done relatively easy.

Every model has a *database* property that is the object that create the model stores in a lower level layer. As the models, the *data*

```
    ...
    def create_and_log(self, log_name):
        """Create a new Dummy and write a new DummyLog
        """

        self.create(async=False)
        log_store = self.database.store('reports')
        dummy_log = DummyLogs()
        dummy_log.name = u'This is a new log that uses MySQL from Dummy'
        log_store.add(dummy_log)
        log_store.commit()
```

1.4.5 The Mamba model guide

Mamba doesn't use any type of configuration file to generate database schemas, in Mamba the schema is defined in Python classes directly extending the `mamba.application.model.Model` class.

Creating a new model

We can generate new model classes using the `mamba-admin` command line tool or adding a new file with a class that inherits from `mamba.application.model.Model` directly. If you use the command line, you have pass two arguments at least, the model name and the real database table that is related to:

```
$ mamba-admin model dummy dummy_table
```

The command above will create a new `dummy.py` file in the `application/model` directory with the following content:

```
# -*- encoding: utf-8 -*-
# -*- mamba-file-type: mamba-model -*-
# Copyright (c) 2013 - user <user@localhost>

"""
.. model:: Dummy
    :plarform: Linux
    :synopsis: None
.. modelauthor:: user <user@localhost>
"""

from mamba.enterprise import Int
from mamba.application import model


class Dummy(model.Model):
    """
    None
    """

    __storm_table__ = 'dummy_table'
```

```
    id = Int(primary=True, unsigned=True)
```

You can pass many parameters to the `mamba-admin model` subcommand. You can set the description of the model, the author name and email, the platforms that this model is compatible with and the classname you want for your model.

As you can see in the code that `mamba-admin` tool generated for us, we define a new class `Dummy` that inherits from `mamba.application.model.Model` class and define a static property named `__storm_table__` that points to the real name of our database table, `dummy_table` in this case. The class define another static property `id` that is an instance of the Storm class `storm.properties.Int` with the parameters `primary` and `unsigned` as `True`.

Its possible to create subpackages to maintain our models under more order and control using a *doted* name for the model, for example:

```
$ mamba-admin model community.users users
```

The previous command will add a python package *community* in *application/model* so finally we can import it in our application with:

```
.. sourcecode:: python
```

    from application.model.community import users

Mamba's Storm properties

In mamba we define our database schema just creating new Python classes like the one in the example above. The following is a table of the available Storm types and their SQLite, MySQL/MariaDB and PostgreSQL equivalents:

Property	Python	SQLite	MySQL/MariaDB	PostgreSQL
Bool	bool	INT	TINYINT	BOOL
Int	int, long	INT	TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT	SERIAL, BIGSERIAL, SMALLSERIAL, INT, BIGINT, SMALLINT
Float	float	REAL, FLOAT, DOUBLE	FLOAT, REAL, DOUBLE PRECISSION	FLOAT, REAL, DOUBLE PRECISION
Decimal	Decimal	VARCHAR, TEXT	DECIMAL, NUMERIC	DECIMAL, NUMERIC, MONEY
UUID	uuid.UUID	TEXT	BLOB	UUID
Unicode	unicode	VARCHAR, TEXT	VARCHAR, CHAR, TEXT	VARCHAR, CHAR, TEXT
RawStr	str	BLOB	BLOB, BINARY, VARBINARY	BYTEA
Pickle	any	BLOB	BLOB, BINARY, VARBINARY	BYTEA
Json	dict	BLOB	BLOB	JSON
DateTime	datetime	VARCHAR, TEXT	DATETIME, TIMESTAMP	TIMESTAMP
Date	date	VARCHAR, TEXT	DATE	DATE
Time	time	VARCHAR, TEXT	TIME	TIME
TimeDelta	timedelta	VARCHAR, TEXT	TEXT	INTERVAL
List	list	VARCHAR, TEXT	TEXT	ARRAY[]
Enum	str	INT	INT	INT
NativeEnum	str	VARCHAR	ENUM	ENUM

All these properties except **NativeEnum** are common Storm properties. The **NativeEnum** property class is just a convenience class that we created to support legacy databases that uses native Enum types in the scenario where we can't change this because the database is used by other applications that we can't modify to switch to Int type.

Warning: The use of the native enum type in MySQL is considered by some developers bad practice and something really evil http://kcy.me/nit3

Properties in deeper detail

A property is a Storm object that *maps* our classes properties with a related field in the database and perform several other operations as cache values among others.

All property classes define a class level property called `variable_class` that is an object that represents the value stored in the database as Python and is the part of the library that effectively *map* the Python representation of the value with the value itself as is stored in the database.

Variables are responsible for setting and getting values on and from the underlying database backend and perform any special operation that is needed to convert the native database types into Python ones.

Property constructor parameters

The parameters that are accepted depends on two factors:

1. The type of the property
2. The selected underlying database backend

All the options that we can pass to the constructor are optional and some of them has no effects at all in some database backends. Mamba defines the following common parameters:

- **name**: The column name in the database. If you set this parameter, the database field and the class attribute can differ. So for example you can have a class attribute called `customer_id` while in the database the field is called `id`.
- **primary**: If you set this parameter as `True`, this attribute is considered to map the primary key of the database table. You can create compound keys by using the class level definition `__storm_primary__` attribute instead.
- **default**: The default value for the property.
- **default_factory**: A factory which returns default values for the property. Mainly used when the default value is a mutable one.
- **validator: A callable object that takes three arguments. The validator has to return the value that the property should be**

    1. the object that the property is attached to
    2. the attribute name as a string
    3. the value that is being set

- **size** (*special behaviour*): The behaviour of this attribute differs depending on the database backend and the type of the property we are setting but mainly it sets the size of the field we are defining in the database.
- **allow_none**: If set to False, Mamba will not allow None (NULL) values to be inserted.
- **index** (*special behaviour*): If set to True, Mamba will create an index for that field.
- **unique** (*special behaviour*): If set to True, Mamba will create an unique index for that field.
- **unsigned** (*special behaviour*): The `unsigned` parameter has different behaviours depending in the database engine and the type as well. Basically, it sets a numeric field as unsigned, this is mainly used with *MySQL/MariaDB* database engines.
- **auto_increment** (*special behaviour*): As his friends above, this parameters has special meanings depending on database engine and field type. It's used to set a column as an auto incremental field (mainly primary keys id's).
- **array** (*postgres only*): This parameter is used to define an array type for PostgreSQL databases. PostgreSQL allows table columnns to be defined as variable-length multidimensional arrays.

The **size**, **index**, **unique**, **unsigned**, **auto_increment** and **array** attributes are not present on Storm, they are implemented only in Mamba and its utility is closely related to the ability of Mamba to generate SQL schemas using Python classes definitions.

Defining a default behaviour

Mamba allows you to run queries synchronous or asynchronous by passing the parameter **async** on functions like *read*, *find*, *update*, *create* and so on. However, sometimes, you want that a specific model to have a default behaviour. Mamba's default is always asynchronous, but if you want to make queries on a specific model always synchronous, you can just set `__mamba_async__` property.

```
from mamba.enterprise import Int
from mamba.application import model

class Dummy(model.Model):

    __storm_table__ = 'dummy'
    __mamba_async__ = False

    id = Int()
    status = Int()
```

Queries on this model will always run synchronous.

Note: You can always override the default behaviour for a single operation. If you issue Dummy().read(1, async=True), this single query will be executed asynchronously.

Defining compound keys

To define a compound key we have to use the `__storm_primary__` class-level attribute and set it as a tuple with the names of the properties that composes the primary key:

```
from mamba.enterprise import Int
from mamba.application import model

class Dummy(model.Model):

    __storm_table__ = 'dummy'
    __storm_primary__ = 'id', 'status'

    id = Int()
    status = Int()
```

Defining an unique for multiple columns

To define a compound unique we have to use the `__mamba_unique__` class-level attribute and set it as a tuple of tuples with the names of the properties that composes the compound unique index:

```
from mamba.enterprise import Int
from mamba.application import model

class Dummy(model.Model):

    __storm_table__ = 'dummy'
    __mamba_unique__ = (('id', 'status'), )

    id = Int()
    status = Int()
```

Defining an index for multiple columns

To define a compound index we have to use the `__mamba_index__` class-level attribute and set it as a tuple of tuples with the names of the properties that composes the compound index:

```
from mamba.enterprise import Int
from mamba.application import model

class Dummy(model.Model):

    __storm_table__ = 'dummy'
    __mamba_index__ = (('id', 'status'), )

    id = Int()
    status = Int()
```

Understanding size

If we set the `size` parameter in an `Unicode` property, Mamba will use it to specify the length of the varchar in the SQL representation. For example:

```
name = Unicode(size=64)
```

will be mapped to

```
name VARCHAR(64)
```

in the resulting SQL schema. It works with any type of database backend that we set.

If the property type that we use is `Decimal` it will work on MySQL/MariaDB **only** and should be completely ignored by PostgreSQL and SQLite backends. In the case of MySQL and `Decimal` the `size` attribute has special meaning depending on the type that you use to define it. That is in this way because you can define a size and a precision in the decimal part of the value:

```
some_field = Decimal(size=(10, 2))   # using a tuple
some_field = Decimal(size=[10, 2])   # using a list
some_field = Decimal(size=10.2)      # using a float
some_field = Decimal(size='10,2')    # using a string
some_field = Decimal(size=10)        # using an int (precission is set to 2)
```

In the above examples, the size is set to 10 and the precision to 2. When passing a single `int`, the precision is set to 2 by default.

If the property type is `Int`, Mamba should ignore it for PostgreSQL and SQLite. If the configured backend is MySQL, Mamba will use the given parameter as the size of the int:

```
age = Int(size=2)
```

should be mapped to

```
age INT(2)
```

Some notes about unsigned

Unsigned is completely ignored by PostgreSQL and SQLite backends so it has no effect at all if you are using any of them.

The auto_increment attribute

This attribute sets a colums or field as `AUTO INCREMENT` in MySQL and MariaDB backends if it's present and `True` in a `Int` property, otherwise is ignored. This is normally used with `primary` attribute also set as `True`.

If you define `auto_increment` as `True` in a `Int` type property using a PostgreSQL backend, then it will be automaticaly transformed to a `serial` type.

This attribute is ignored when using SQLite backend.

Model operations

Create and insert a new object into the database is pretty straightforward, we just have to create a new instance of our model and call the `create` method on it:

```
>>> dummy = Dummy()
>>> dummy.name = u'The Dummy'
>>> dummy.create(async=False)
```

Read a model instance (or row) from the database is as easy as using the `read` method of the `Model` class with the id of the row we want to get from the database:

```
>>> dummy = Dummy.read(1, async=False)
```

Update is performed in the same easy way, we just modify our object and call the `update` method on it:

```
>>> dummy.name = u'Modified Dummy'
>>> dummy.update(async=False)
```

The delete operation is no different, we just call the `delete` method from our object (note that this doesn't delete the object reference itself, only the databse row):

```
>>> dummy.delete()
```

Note: In Mamba, by default, **CRUD** operations are executed as Twisted transactions in the model object if we don't override the methods to have a different behaviour or add the `async=False` named param to the call.

Other model operations for your convenience

Mamba supports the `find` and `all` for your convenience.

```
>>> [c.name for c in Customer.all(async=False)]
>>> [c.name for c in Customer.find(Customer.age >= 30)]
```

Note: The find method accepts the same arguments and options than the regular Storm `store.find` but you don't have to define the model to look for as is automatically added for you.

References

We can define references between models (and between tables by extension) instantiating `Reference` and `ReferenceSet` objects in our model definition:

```
from mamba.application import model
from mamba.enterprise import Int, Reference

from application.model.dojo import Dojo

class Fighter(model.Model):
```

```
    __storm_table__ = 'fighter'
    __mamba_async__ = False  # we don't want go asynchronous as we are on terminal

    id = Int(primary=True, auto_increment=True, unsigned=True)
    dojo_id = Int(unsigned=True)
    dojo = Reference(dojo_id, Dojo.id)
```

In the previous example we defined a `Fighter` class that defines a many-to-one reference with the `Dojo` class imported from the dojo model. As this reference has been set we can use the following code to refer to the fighter's dojo in our application:

```
>>> fighter = Fighter().read(1)
>>> print(fighter.dojo.id)
1
>>> print(fighter.dojo.name)
u'SuperDojo'
```

Many-to-many relationships are a bit more complex than the last example. Let's continue with our previous fighter example to draw this operation:

```
from mamba.application import model
from mamba.enterprise import Join, Select, Not
from mamba.enterprise import Int, Unicode, Reference, ReferenceSet

from application.model.dojo import Dojo
from application.model.tournament import Tournament


class Fighter(model.Model):

    __storm_table__ = 'fighter'
    __mamba_async__ = False  # we don't want go asynchronous as we are on terminal

    id = Int(primary=True, auto_increment=True, unsigned=True)
    name = Unicode(size=128)
    dojo_id = Int(unsigned=true)
    dojo = Reference(dojo_id, Dojo.id)

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


class TournamentFighter(model.Model):

    __storm_table__ = 'tournament_fighter'
    __storm_primary__ = 'tournament_id', 'fighter_id'
    __mamba_async__ = False  # we don't want go asynchronous as we are on terminal

    tournament_id = Int(unsigned=True)
    fighter_id = Int(unsigned=True)
```

Note: Tournament and Dojo definition classes has been avoided to maintain the simplicity of the example

In the above example we defined a `TournamentFighter` class to reference tournaments and fighters in a many-to-many relationship. We defined a compound primary key with the `tournament_id` and `fighter_id` fields. To make the relationship real we have to add a `ReferenceSet`:

```
Tournament.fighters = ReferenceSet(
    Tournament.id, TournamentFighter.tournament_id,
    TournamentFighter.fighter_id, Fighter.id
)
```

We can also add the definition to the `Tournament` class definition directly but in that case we have to use special inheritance that we didn't see yet, we will cover it later in this chapter. Since the reference set is created we can use it as follows:

```
>>> chuck = Fighter(u'Chuck Norris')
>>> bruce = Fighter(u'Bruce Lee')

>>> kung_fu_masters = Tournament(u'Kung Fu Masters Tournament')
>>> kung_fu_masters.fighters.add(chuck)
>>> kung_fu_masters.fighters.add(bruce)

>>> kung_fu_masters.fighters.count()
2

>>> store.get(TournamentFighters, (kung_fu_masters.id, chuck.id))
<TournamentFighter object at 0x...>

>>> [fighter.name for fighter in kung_fu_masters.fighters]
[u'Chuck Norris', u'Bruce Lee']
```

We can also create a reverse relationship between fighters and tournaments to know in which tournaments a fighter is figthing on:

```
>>> Fighter.tournaments = ReferenceSet(
    Fighter.id, TournamentFighter.fighter_id,
    TournamentFighter.tournament_id, Tournament.id
)

>>> [tournament.name for tournament in chuck.tournaments]
[u'Kung Fu Masters Tournament']
```

SQL Subselects

Sometimes we need to use subselects to retrieve some data, for example we may want to get all the fighters that are not actually figthing in any tournament:

```
>>> yip_man = Fighter(u'Yip Man')
>>> store.add(yip_man)

>>> [fighter.name for fighter in store.find(
        Fighter, Not(Fighter.id.is_in(Select(
            TournamentFighter.fighter_id, distinct=True))
        )
)]
[u'Yip Man']
```

You can split this operation in two steps for improved readability:

```
>>> subselect = Select(TournamentFighter.fighter_id, distinct=True)
>>> result = store.find(Fighter, Not(Fighter.id.is_in(subselect)))
```

SQL Joins

We can perform implicit or explicit joins. An implicit join that use the data in our previous examples may be:

```
>>> resutl = store.find(
    Dojo, Fighter.dojo_id == Dojo.id, Fighter.name.like(u'%Lee')
)

>>> [dojo.name for dojo in result]
[u'Bruce Lee awesome Dojo']
```

The same query using explicit joins should look like:

```
>>> join = [Dojo, Join(Fighter, Fighter.dojo_id == Dojo.id)]
>>> result = store.using(*join).find(Dojo, Fighter.name.like(u'%Lee'))

>>> [dojo.name for dojo in result]
[u'Bruce Lee awesome Dojo']
```

Or more compact syntax as:

```
>>> [dojo.name for dojo in store.using(
        Dojo, Join(Fighter, Fighter.dojo_id == Dojo.id)
    ).find(Dojo, Fighter.name.like(u'%Lee'))]
[u'Bruce Lee awesome Dojo']
```

Common SQL operations

Two common operations with SQL are just ordering and limiting results, you can also perform those operations using the underlying Storm ORM when you are using Mamba models. Order our results is really simple as you can see in the following example:

```
>>> result = store.find(Fighter)
>>> [fighter.name for fighter in result.order_by(Fighter.name)]
[u'Bruce Lee', u'Chuck Norris', u'Yip Man']

>>> [fighter.name for fighter in result.order_by(Desc(Fighter.name))]
[u'Yip Man', u'Chuck Norris', u'Bruce Lee']
```

In the example above we get all the records from the fighters database and order them by name and then order them by name in descendent way. As you can see Chuck Norris is always inmutable but that is just because he is Chuck Norris.

To limit the given results we just slice the result:

```
>>> [fighter.name for fighter in result.order_by(Fighter.name)[:2]]
[u'Bruce Lee', u'Chuck Norris']
```

Those slices are translated to OFFSET and LIMIT in the underlaying database SQL query by Storm so this last operation is translated to something like this for MySQL/MariaDB:

    SELECT fighter.name FROM fighters LIMIT 2 OFFSET 0

Storm adds the possibility to use SQL expressions in an agnostic database backend way to perform those operations as well:

```
>>> result = store.execute(Select(Fighter.name, order_by=Desc(Fighter.name), limit=2))
>>> result.get_all()
[(u'Yip Man',), (u'Chuck Norris',)]
```

Multiple results

Sometimes is useful that we get more than one object from the underlying database using only one query:

```
>>> result = store.find(
    (Dojo, Fighter),
    Fighter.dojo_id == Dojo.id, Fighter.name.like(u'Bruce %')
)

>>> [(dojo.name, fighter.name) for dojo, fighter in result]
[(u'Bruce Lee awesome Dojo', u'Bruce Lee')]
```

Value auto reload and expression values

Storm offers a way to make values to be auto reloaded from the database when touched. This is performed assigning the `AutoReload` attribute to it

```
>>> from storm.locals import AutoReload

>>> steven = store.add(Person(u'Steven'))
>>> print(steven.id)
None

>>> steven.id = AutoReload
print(steven.id)
```

Steven has been autoflushed into the database. This is useful to making objects automatically flushed if necessary.

You can also assign what in the Storm project they call a "*lazy expression*" to any attribute. The expressions are flushed to the database when the attribute is accessed or when the object is flushed to the database.

```
>>> from storm.locals import SQL
>>> steven.name = SQL('(SELECT name || ? FROM fighter WHERE id=4)', (' Seagal',))
>>> steven.name
u'Steven Seagal'
```

Serialize

You can serialize Model objects either to a dictionary or to JSON. All references will be serialized correctly and you can specify, if you like, specify only the fields you'd like in a `fields` parameter or fields to exclude in a `exclude` parameter.

The json property is very simple, it returns a JSON representation of the model instance.

```
>>> customer = Customer.find(Customer.id == 2, async=False)
>>> customer.json
'{"name": "Austin Powers", "id": 2, "adresses": [{"street": "Memory Lane", "postcode": 60}]}'
```

The dict function supports more options, however.

```
>>> customer = Customer.find(Customer.id == 2, async=False)
>>> customer.dict(json=True)  # json property uses dict method internally
'{"name": "Austin Powers", "id": 2, "adresses": [{"street": "Memory Lane", "postcode": 60}]}'
>>> customer.dict(traverse=False, json=True)  # We don't want references
'{"name": "Austin Powers", "id": 2}'
>>> customer.dict(traverse=False, json=True, exclude=['name'])
'{"id": 2}'
```

```
>>> customer.dict(traverse=False, json=True, fields=['id', 'adresses.street'])  # Specify a single fi
'"id": 2, "adresses": [{"street": "Memory Lane"}]}'
```

Queries debug

Sometimes is really useful to see which statement Storm is being executed behind the curtains. We can use a debug tracer that comes integrated in Storm itself. Using it is really easy:

```
>>> import sys
>>> from storm.tracer import debug

>>> debug(True, stream=sys.stdout)
>>> result = store.find(Fighter, Fighter.id == 1)
>>> list(result)
EXECUTE: 'SELECT fighter.id, fighter.name, fighter.dojo_id FROM fighters WHERE fighter.id = 1', ()
[<Fighter object at 0x...>]

>>> debug(False)
>>> list(result)
[<Fighter object at 0x...>]
```

Real SQL Queries

We can use real **database dependant** queries if we want to:

```
>>> result = store.execute('SELECT * FROM fighters')
>>> result.get_all()
[u'Bruce Lee', u'Chuck Norris', u'Yip Man', u'Steven Seagal']
```

The Storm base class

In some situations we are not going to be able to import every other model class into our local scope (mainly because circular import issues) to define `Reference` and `ReferenceSet` properties. In that scenario we can define these references using a stringfield version of the class and property names involved in the relationship.

To do that, we have to inherit our classes from the `Storm` base class as well as from `Model` class. There is another inconvenience related with this. When we inherit from both classes we have to define that the metaclass of the class that we are defining is *`MambaStorm`* to avoid metaclass collisions between mamba model and storm itself.

```
from mamba.enterprise import Int, Unicode, Reference, ReferenceSet

class Fighter(model.Model, Storm):

    __metaclass__ = model.MambaStorm
    __storm_table__ = 'fighters'

    id = Int(primary=True)
    name = Unicode()
    dojo_id = Int()
    dojo = Reference(dojo_id, 'Dojo.id')
```

1.4.6 The Mamba controller guide

The controller is what connects your model and business logic with the views in the frontend part of the application. In Mamba we don't have routing system like Django does, in Mamba we use a request routing system similar to the one found in Bottle or Flask.

How it works?

Every controller in a Mamba application inherits directly from the `mamba.application.controller.Controller` class. A controller can define **action methods** and regular methods.

- An action method is a method that is invoked by the underlying request routing system when our application receives a request. Each action in our controller has to fit in a route, we define routes using the `route` decorator and decorating the method to fit a route.
- A regular method is just a normal method that is not decorated and is usually some kind of helper method that helps our controllers to make its work.

A controller can define as many action and regular emthods as is needed and every controller action represents a single URL segment of the application.

```
@route('/')
def root(self, request. **kwargs):
    """Just an action method for the root of the controller
    """
    return Ok(sef.salutation())

def salutation(self):
    """Just a regular method used as helper for our controller class
    """

    return 'Hello World!'
```

Its possible to create subpackages to maintain our controllers under more order and control using a *doted* name for the controller, for example:

```
$ mamba-admin controller community.user_controller
```

The previous command will add a python package *community* in *application/controller* so finally we can import it in our application with:

```
.. sourcecode:: python
```

    from application.controller.community.user_controller import UserController

See also:

The Mamba routing guide

Mamba create and handles an internal controller manager object that load all the controllers that are found in our application on initialization time and register all its routes with the routing system.

When a HTTP request is made from a client, it's captured by the underlying Twisted routing mechanism that delegate its process to the main or root Mamba's controller (that is defined internally into the framework) that pass it to the Mamba's routing and templating system to locate, retrieve and render a response for the given request.

The response can be a HTML rendered page, text, a JSON structure, a XML Object or arbitrary binary data, that depends completely on the application and what the client is looking for. If Mamba can't locate a valid action or resource for the given request, a 404 response will be returned back.

Controllers route

Every Mamba controller has to define a route for themselves, this route is their position in the web site path hierarchy and it **must** be a single path segment of the URL. A controller that doesn't define a controller route is attached to the main Twisted Site object and becomes the root of the application. That means no Mamba's default special behaviour is going to be available for the site if we don't define it mannually in the controller that becomes the root.

Note: Don't worry if you don't totally understand that, it will be clear later in this chapter.

The controller's route can be defined when we create the controller using the `mamba-admin controller` subcommand or just editing the resulting file with our text editor. The argument for the `mamba-admin` command line tool is `--route=<our route>` so for example, the command:

```
$ mamba-admin controller --route=api webservice
```

Will create a new controller python script on `application/controller/webservice.py` with the following content:

```
# -*- encoding: utf-8 -*-
# -*- mamba-file-type: mamba-controller -*-
# Copyright (c) 2013 - damnwidget <damnwidget@localhost>

"""
.. controller:: Webservice
    :platform: Linux
    :synopsis: None

.. controllerauthor:: damnwidget <damnwidget@localhost>
"""

from mamba.web.response import Ok
from mamba.application import route
from mamba.application import controller


class Webservice(controller.Controller):
    """
    None
    """

    name = 'Webservice'
    __route__ = 'api'

    def __init__(self):
        """
        Put your initializarion code here
        """
        super(Webservice, self).__init__()
```

As you can see, the generated file already defines the controller's route as `'api'` but we can just modify that value to whatever other route that we want. If we use more than one single URL path segment the route is totally ignored and our controller is not registered in the system making it unavailable.

```
...
__route__ = 'api/socket'
...
```

The above example should end in the behaviour described above.

Controllers actions

Controllers can define arbitrary routes with the `@route` decorator that finally callbacks the decorated method. Those routes can be static routes (that only defines a path) or dynamic routes (that defines a path and wildcards for parameters).

```
# static route example
@route('/comments')
def comments(self, request, **kwargs):
    ...

# dynamic route example
@route('/comments/<int:comment_id>')
def read_comment(self, request, comment_id, **kwargs):
    ...
```

Controller actions can define more extensive route paths so we can for example register the following route for our `Webservice` example controller (defined in the last section):

```
...
@route('/contacts/add/<email>/<password>')
def add_contact(self, request, email, password, **kwargs):
    contact = new Contact(email, password)
    contact.create()
```

In the above example our final route path (as will be invoked from the web client) is `http://localhost/api/contacts/add/john_doe@gmail.com/ultrasecret`. This is:

Controller route	Action route	Match
/api	/contacts/add	{'email': 'john_doe@gmail.com', 'password': 'ultrasecret'}

See also:

The Mamba routing guide

Mamba's default root

Mamba defines internally a default root route that points always to the `index.html` template view. Sometimes we need a controller to become the root of our application because we want to develop a full backend REST service or for whatever other reasson. When we do that, we are going to override all the Mamba's auto insertion of **mambaerized** resources like CSS, LESS or JavaScript files.

If you are not going to use a frontend at all then you are just done, all is ok and you don't have to care about but if you are planning to use Mamba's templating system then you have to create a new index to recover the default root functionality.

First of all we have to create a new view for the controller using the `mamba-admin view` subcommand. Let's imagine we defined a controller that becomes the root resource in our application and we call it `Main` and we use the default `root` action method as the `/` or index route:

```
class Main(controller.Controller):

    name = 'Main'
    __route__ = ''

    def __init__(self):
        """
        Put your initialization code here
        """
```

```
        super(Main, self).__init__()

    @route('/')
    def root(self, request, **kwargs):
        Ok('I am the Main, hello world!')
```

Then we generate a new view for the root action using the `mamba-admin` command line tool:

```
$ mamba-admin view root Main
```

This will generate a new file `application/view/Main/root.html` that will be our new index template for the whole application that inherits from the `layout.html` template. This view will know how to insert the **mambaerized** resources into our templates in automatic way.

Our last step is to just make a small change in the `root` action in the controller to make it render our new index:

```
from mamba.core import templating

class Main(controller.Controller):

    name = 'Main'
    __route__ = ''

    def __init__(self):
        """
        Put your initialization code here
        """
        super(Main, self).__init__()
        self.template = templating.Template(controller=self)

    @route('/')
    def root(self, request, **kwargs):
        return Ok(self.template.render().encode('utf-8'))
```

Note: If you don't know what a *mambaerized resource file* is, we recommend you to read the Getting Started document and come back here when you read it

Nested Controllers

Sometimes we want to group different controllers under the same path (user, cart and actions under the *api* path for example), controllers can be attached to other controllers setting the *__parent__* property to the route of the controller that we want to arrach it.

A container can (and should) define a controller's *__route__* just like any other controller. If the *__route__* property is not set in an attached controller, this can lead to totally unpredictable and hard to debug weird behaviour in your routes dispatching.

You can add as much nested level as you wish.

How to use it?

We can set the container of any controllers (including other containers) just defining the class property *__parent__* with the correct name of the route of the container that we want to add our controller to. We can have for example a container called *api* and attach the controllers *users* and *wallet* to it.

The controller's container code should look like the code below:

```
from mamba.application import route
from mamba.application import controller
from mamba.web.response import BadRequest


class Api(controller.Controller):

    name = 'Api'
    isLeaf = False
    __route__ = 'api'

    def __init__(self):
        """Put your initialization code here
        """
        super(Api, self).__init__()

    @route('/')
    def root(self, request, **kwargs):
        return BadRequest()
```

While the attached controllers should look like:

```
from mamba.web.response import Ok
from mamba.application import route
from mamba.application import controller


class UserController(controller.Controller):
    """User Controller
    """

    name = 'User'
    __route__ = 'user'
    __parent__ = 'api'

    def __init__(self):
        """Put your initialization code here
        """
        super(UserController, self).__init__()

    @route('/')
    def root(self, request, **kwargs):
        return Ok('Give me users please!')

    @route('/test')
    def test(self, request, **kwargs):
        return Ok('TEST OK!')
```

```
from mamba.web.response import Ok
from mamba.application import route
from mamba.application import controller


class WalletController(controller.Controller):
    """Wallet Controller
    """

    name = 'Wallet'
```

```
    __route__ = 'wallet'
    __parent__ = 'api'

    def __init__(self):
        """Put your initialization code here
        """
        super(WalletController, self).__init__()

    @route('/<int:wallet_id>')
    def test(self, request, wallet_id, **kwargs):
        return Ok('Fake Wallet')
```

With the configuration above, we should end with the following routes:

Path	Result
/api	BadRequest
/api/user	Give me users please
/api/user/test	TEST OK!
/api/wallet/1	Fake Wallet

Going asynchronous

Mamba is just Twisted and Twisted is an asynchronous network framework. We can run operations asynchronous and return back callbacks from Twisted deferreds as we do in any normal Twisted application. We can do it always that we decorate a model method with the `@transact` decorator in our models.

```
from twisted.internet import defer

from mamba.application import route
from mamba.application.controller import Controller

from application import controller
from application.model.post import Post


class Blog(Controller):
    """
    Blog controller
    """

    name = 'Blog'
    __route__ = 'blog'

    def __init__(self):
        """
        Put your initialization code here
        """
        super(Blog, self).__init__()

    @route('/<int:post_id>/comments', method=['GET', 'POST'])
    @defer.inlineCallbacks
    def root(self, request, post_id, **kwargs):
        """Return back the comments for the given post
        """

        comments = yield Post().comments
        defer.returnValue(comments)
```

We just used the Twisted's `@defer.inlineCallbacks` decorator to yield results from asynchronous operations and then we returned back the value using `defer.returnValue`.

See also:

Twisted: Introduction to Deferreds, Twisted: Deferred Reference, Twisted: Generating Deferreds

Returning values from controller actions

Surely the reader already noticed that we use an `Ok` object as return from our controller actions. The `Ok` class is one of the multiple built-in response objects that you can return from your application controllers.

Mamba defines 15 predefined types of response objects that set the content-type and other parameters of the HTTP response that our applications can return back to the web clients.

- *`Response`* dummy base response object, we can use this object to create ad-hoc responses on demand. All the rest of responses inherits from this class
- *`Ok`* - Ok 200 HTTP Response
- *`Created`* - Ok 201 HTTP Response
- `Unknown` - Unknown 209 HTTP Response (this HTTP code is not defined, mamba returns that when a route just returns None)
- *`MovedPermanently`* - Ok 301 HTTP Response
- *`Found`* - Ok 302 HTTP Response
- *`SeeOther`* - Ok 303 HTTP Response
- *`BadRequest`* - Error 400 HTTP Response
- *`Unauthorized`* - Error 401 HTTP Response
- `Forbidden` - Error 403 HTTP Response
- *`NotFound`* - Error 404 HTTP Response
- *`Conflict`* - Error 409 HTTP Response
- *`AlreadyExists`* - Error 409 HTTP Response (Conflict found in POST)
- *`InternalServerError`* - Internal Error 500 HTTP Response
- *`NotImplemented`* - Error 501 HTTP Response

Mamba return back some of those codes by itself in some situations, for example, if we try to use a route that exists but in a different HTTP method, we get a *`NotImplemented`* response object.

You can return whatever of these objects from your controller. Mamba take care of rendering it correctly to the web client. You can also return dictionaries and other objects. Mamba will try to convert whatever object you return from a controller into a serializable JSON structure with a default 200 OK HTTP response code and an 'application/json' encoding.

1.4.7 The Mamba routing guide

Mamba uses a routing engine similar to the ones found in Flask and Bottle. Of course you can register the same route path for those seven different HTTP methods

- GET
- POST

- PUT
- DELETE
- OPTIONS
- PATCH
- HEAD

Note: The default HTTP method is 'GET'.

Routes are defined in controllers using the `@route` decorator:

```
@route('/test', method='GET')
def test(self, request, **kwargs):
    return Ok('I am a test!')
```

The decorated function has to accept both positional arguments `self` and `request`. The request argument is a low-level Twisted argument that is used internally by the routing system.

We can pass whatever argument we want to the route and access them using the `kwargs` dictionary. There is another way to register routes with arguments (that should be validated before fire the route callback) that we will review later in this chapter.

We can define that the route should be available for all the HTTP methods that we want just passing a tuple or a list of HTTP methods in the decorator.

```
@route('/test', method=('GET', 'POST'))
def test(self, request, **kwargs):
    return Ok('I am a test!')
```

Controllers and routes

In Mamba there are two types of routes:

1. Static routes
2. Controller routes

The first one is used for Mamba to locate and render templates that do not depends on a controller, like an static HTML page or a main HTML page that loads JavaScript to create single page applications using AngularJS or whatever other library or framework.

The second one is used to create relationships between routes and actions in our application controllers. Those actions can end in some template rendering, some acces to the database, some communication with external services or whatever other action that we can think about.

Static routes

In Mamba there is always a main static route tied to the web site root page that will point always to the index template.

Note: This default behaviour can be overriden using controllers.

To create new static routes to other templates you only have to place a hyperlink in your HTML pointing to the new route and create a new template with the same name of the route in the `application.view.templates` directory. If for example your new route is `http://localhost/about_us` the template should be `application/view/templates/about_us.html`.

Static routes has one limitation, they can't be nested in path hierarchies. So you can't define a static route that links to `http://localhost/some_path/some_static_template`

That's because mamba doesn't know how to follow route to the `some_path` without a controller that define routes to do it.

Note: The exposed above is an implementation decision. Mamba uses Twisted web as primary component in order to run and render our web sites. Twisted use `twisted.web.resource.Resource` class instances in to execute any logic that our web site needs, we always need an instance of this class to render our pages. Mamba take care of abstract this but it maintains itself flexible enough to make the developer able to override any Mamba's specific behaviour.

With this in mind, Mamba does not allow the user code to define actions without a valid Twisted resource, controllers are an abstraction of those Twisted resources and they can be interchanged if needed. That means, an experienced Twisted developer can bypass Mamba's routing system at all and use controllers as pure Twisted resources or even use Twisted resources as first class citizens.

See also:

The Mamba view guide

Controller routes

We have to differentiate between the controller `route` and the controller actions:

1. The controller `route` is the base path where the controller lives. If this route is a void string, then this controller override the default static route to the index page that we discuss about in the Controllers and routes section. If the controller route is for example *api* then all the routes to controller actions that the controller defines should really pont to `http://localhost/api/<action>`
2. The controller actions are regular methods that are decorated with the `@route` decorator and become unique URL entry points into our application.

A controller can have only one route but it can have as many actions as is needed. Only one controller can override the page index, if you define more than one controller with an empty string as it's route, then only one of them (in random way really) should be known by the routing system hiding the others completely.

See also:

The Mamba controller guide

Controller actions

Mamba distinguishes between two types of routes, static and dynamic routes. Static routes are all those routes that defines only a path part, for example:

```
@route('/blog')
```

Dynamic routes in the other hand, contains one or more *wildcards*:

```
@route('/blog/<post_id>')
```

Wildcards

There are three type of wildcards on mamba:

1. **Int** wildcard, that matches digits and cast them to integers

2. **Float** wildcard, that macthes a decimal number and converts it to float
3. **untyped** wildcards, that matches whatever other type of argument as strings

The wildcard consist in a name enclosed in angle brackets for untype wildcars or a type followed by a colon and a name enclosed in angle brackets if we are going to define numeric arguments.

```
# untyped wildcard
@route('/run/<action>')
def run(self, request, action, **kwargs):
    ...

# int wildcard
@route('/run/<action>/<int:post_id>')
def run(self, request, action, post_id, **kwargs):
    ...

# float wildcard
@route('/sum/<float:amount>')
def run(self, request, amount, **kwargs):
    ...
```

Wildcards names should be unique for a given route and must be a valid python identifier because they are going to be used as keyword arguments in the request callback when the routing system dispatch them.

Following the latest examples, the route `/run/<action>` matches:

Path	Result
/run/close	{'action': 'close'}
/run/close/	{'action': 'close'}
/run/close/bar	Doesn't match
/run/	Doesn't match
/run	Doesn't match

If the decorated method does not define a positional argument that match the wildcard name, we can always get it using the **kwargs** dictionary:

```
@route('/run')
def run(self, request, **kwargs):
    return Ok(kwargs.get('action'))
```

Although this is totally valid, we should use explicit argument definition in our methods to be more semantic.

Of course you can decide to don't use wildcards at all and just pass arguments to your actions in a traditional way with form encoding for POST and URIs for GET and you will be totally able to access all those arguments through the **kwargs** dictionary. Mamba give you the tool, but you decide how to use it.

1.4.8 The Mamba view guide

Mamba uses Jinja2 as template engine. Jinja2 is a modern and designer friendly templating language for Python, modelled after Django's templating system. It is fast, widely used and secure with optional sandboxed template execution environment.

Jinja2 Documentation

Jinja2 has its own (and extensive) documentation, Mamba take cares of the initialization of the common contexts and loaders so you don't have to take care yourself.

The Jinja2 documentation is available in it's project web site, we recommend you read their documentation.

Mamba templating

Mamba implements its own wrapper around Jinja2 templating engine to integrate it with Mamba routing and controllers systems.

In Mamba, the only thing that you need to create and render a new template is create a Jinja2 template file into `application/view/templates` directory and point to it with your browsers or link it in another of your application pages.

Mamba define a default root template internally called `root_page.html` that is needed for the framework to insert the web resources that we add to the `stylesheets` and `scripts` directories into `application/view`. The layout that all the views in your application share inherit directly from this internal template.

Files inside the `stylesheets` and `scripts` directory must define an special file header for make it able to being automatic loaded by the styles and scripts managers and inserted in all your templates. Those special file headers are:

```
/*
 * -*- mamba-file-type: mamba-css -*-
 */
```

for css files and:

```
/*
 * -*- mamba-file-type: mamba-javascript -*-
 */
```

for JavaScript ones.

When we generate a new Mamba application with the `mamba-admin` command line tool, it creates a default `layout.html` file that is located inside the `application/view` directory. This file can be used as common layout for all your application pages and it **must** inherit directly from `root_page.html`.

The default content of the basic layout is as follows:

```
{% extends "root_page.html" %}
{% block head %}
    <!-- Put your head content here, without <head></head> tags -->
{{ super() }}
{% endblock %}
{% block body %}
    <!-- Put your body content here, without <body></body> tags -->
{% endblock %}
{% block scripts %}
    <!-- Put your loadable scripts here -->
{{ super() }}
{% endblock %}
```

We want to define all the common elements in our web site or web application in this file and then create new templates that inherit from this file. To create a new template for our application index we can do it with the `mamba-admin` command tool:

```
$ mamba-admin view index
```

The command above will generate a new `index.html` template file inside `application/view/templates` directory with the following content:

```
{% extends "layout.html" %}
    {% block content %}
    {{ super() }}

    <!--
        Copyright (c) 2013 - damnwidget <damnwidget@localhost>

        view: Index
            synopsis: None

        viewauthor: damnwidget <damnwidget@localhost>
    -->

    <h2>It works!</h2>

    {% endblock %}
```

As you can see, this new template file inherits from `layout.html`, as we already defined our web site layout we have only to take care of the *content* of this especific template or page, in this way we don't need to rewrite unnecessary HTML code in every template in our web site.

We can create as many template files as we want for our application but those templates are just static templates their content is really inmutable. Sometimes this us just what you want because you are creating a single page application for example and what you need is just load some JavaScript files that really takes care of all the frontend related actions.

To generate dynamic content for our templates we need controllers and routes that call actions in those controllers and then render a view based in our template files.

The Mamba Template class

The way that Mamba controllers have to render our templates is just using the *`mamba.core.templating.Template`* class. We can render whatever template that we need instantiating those classes and calling their `render` method with the arguments that we need.

```
...
from mamba.core import interfaces, templating
...

class DummyController(Controller):
    """Just a dummy example controller
    """

    name = 'Dummy'
    __route__ = 'dummy'

    def __init__(self):
        super(DummyController, self).__init__()
        self.template = templating.Template(controller=self)
```

When we pass `self` as the `controller` argument to the constructor we are telling Mamba to look for templates also in the controller templates directory. Every template that is inside this directory hides whatever other template that is located in the general templates directory (`application/view/templates`) that has the same name.

The controller templates directory

The controller's templates directory is a directory inside `application/view` with the same name than the controller, so following our previous example, the `DummyController` templates directory will be `application/view/DummyController`.

If for example we want to render the index (or root) of a given controller route we only have to create a new Jinja2 template file with the same name than the action function. So if our `DummyController` index action is called `root` we have to create a `root.html` file inside `application/view/DummyController`, we can do it of course using the `mamba-admin` command line tool:

```
$ mamba-admin view root DummyController
```

The command is exactly the same than before but we add a second argument that is the name of the controller that we want to create this template for. Let's imagine that we generate that `root.html` file inside the `DummyController` templates directory with the following content:

```
{% extends "layout.html" %}
    {% block content %}
    {{ super() }}

    <h2>Hello {{ name }}!</h2>

    {% endblock %}
```

The python action function in the controller that renders this view should look like:

```
@route('/')
def root(self, request, **kwargs):
    """Renders the DummyController main page
    """

    template_args = {'name': 'Mamba'}
    return Ok(self.template.render(**template_args).encode('utf-8'))
```

Our web site will render *Hello Mamba!*

Shared template arguments

Sometimes we are really going to need to share some global data between different controllers and templates, for example on which section of the web site we are to correctly draw a navigation menu.

In those cases we need somewhere to place global common data that we can share between controllers to correctly render our views, we can usually place this code in two common places:

- The `application/__init__.py`
- The `application/controller/__init__.py`

This is the case of the mamba's main page navigation bar for example. Mamba main site define a tipical bootstrap nav bar like:

```
<ul class="nav">
  {% if menu %}
    {% macro selected_li(path, label, active, available, caller) -%}
    <li {% if active %} class="active" {% endif %}><a {% if not available %} data-toggle="modal" data
    {%- endmacro %}
    {% for link in menu_options %}
      {% call selected_li(link.path, link.label, link.active, link.available) %}
```

```
      {% endcall %}
    {% endfor %}
  {% else %}
    <li class="active"><a href="/index">Home</a></li>
    <li><a href="#gettingstart">Get started</a></li>
    <li><a href="#docs">Documentation</a></li>
    <li><a href="download">Download</a></li>
    <li><a href="#blog">Blog</a></li>
    <li><a href="contact">Contact</a></li>
  {% endif %}
</ul>
```

To correctly render the web site we need to know in which page we are now and which pages are available from where we are. To do that we added a global structure in the `application/controller/__init__.py` file that we can use from whatever controller:

```
# Controllers should be placed here

"""
Some helper functions and stuff here to make our life easier
"""

HOME, GET_STARTED, DOCUMENTATION, DOWNLOAD, BLOG, CONTACT = range(6)

template_args = {
    'menu': True,
    'menu_options': [
        {'path': '/', 'label': 'Home', 'active': False, 'available': True},
        {
            'path': 'gettingstart', 'label': 'Get started',
            'active': False, 'available': True
        },
        {
            'path': 'docs', 'label': 'Documentation',
            'active': False, 'available': True
        },
        {
            'path': 'download', 'label': 'Download',
            'active': False, 'available': True
        },
        {'path': 'blog', 'label': 'Blog', 'active': False, 'available': True},
        {
            'path': 'contact', 'label': 'Contact',
            'active': True, 'available': True
        },
    ]
}


def toggle_menu(menu_entry):
    """
    Toggle all the active state menus that are not the given menu entry to
    inactive and set the given one as active

    :param menu_entry: the menu entry to active
    :type menu_entry: dict
    """
```

```
    menu = template_args['menu_options'][menu_entry]
    if menu['available'] is False:
        return

    for item in template_args['menu_options']:
        if item['active']:
            if item == menu:
                continue

            item['active'] = False

    menu['active'] = True
```

In this way in the downloads page controller we can set this structure as needed:

```
from application import controller

class Downloads(Controller):
...
    @route('/', method=['GET', 'POST'])
    @defer.inlineCallbacks
    def root(self, request, **kwargs):
        """Renders downloads main page
        """

        controller.toggle_menu(controller.DOWNLOAD)
        template_args = controller.template_args

        template_args['releases'] = yield Release().last_release_files()
        template_args['old_releases'] = yield Release().old_release_files()

        defer.returnValue(
            Ok(self.template.render(**template_args).encode('utf-8')))
...
```

We fill the `controller.template_args` using the function `toggle_menu` with the right location before pass it as part of the arguments to the template `render` method.

Note: The full code of the mamba web site can be found under the GPLv3 License at https://github.com/PyMamba/BlackMamba

Rendering global templates from controller actions

We can pass a template name as the first argument (or as template keyword argument) to the template `render` method. If we passed `self` as value of the `controller` argument when you instantiate the `Template` object, then Mamba should try to load it from the controller's templates directory and if can't find it then it will look in the global templates directory.

If Mamba can't find any template with that name then it raises a `core.templating.NotConfigured` exception.

Auto compiling LESS scripts

Mamba can auto-compile LESS scripts if the *lessc* tool has been installed on the system and it's available to the user that is running the Mamba application. In case the *lessc* tool is not installed on the system, the raw contents of the less file are returned as fallback.

To add a LESS resource to our application we should just place the LESS file into the *application/view/stylesheets* directory with the following header:

```
/*
 * -*- mamba-file-type: mamba-less -*-
 */
 ...
```

Mamba will try to compile the LESS scripts that are placed in that directory and return it back as already ccompiled CSS contents to the browser in total transparent way.

LESS auto-compilation when lessc is not available

In environments where *lessc* is not available, Heroku for example, we can set the configuration option *lessjs* in *config/application.json* with the exact file name of the *less.js* script that we want to user. Mamba will insert the LESS JavaScript version in the main layout for us.

Note: The full list of available *less.js* script versions can be found in GitHub: https://github.com/less/less.js/tree/master/dist

---

1.5 Mamba reusability

Mamba applications, controllers, templates, models and components can be reused all packed in an importable application using the Mamba reusability system.

The motivation for supporting this system in mamba is pretty straightforward, as Python developers, sometimes we need a common feature like being able to send emails to our users using some type of SMTP service in our applications. Mamba itself is not able to send emails because it doesn't know how to do it, this is an implementation design decision as we think sending emails is not part of the responsabilities of the framework but it's a pretty standard need for several web related projects, but not all.

The need to reuse the same type of features in several of our projects show us the importance to add some way of easy-code-reusability system in mamba, and now here we are.

1.5.1 How it works?

Mamba is able to include and reuse applications and components from and to your applications using the Mamba packaging system.

Although we are allowed to reuse any Python application just adding a package directly inside our *application* directory (and it works and has nothing wrong), Mamba uses the *mamba-admin* command line tool to package Mamba-based applications and create reusable Mamba packages. These packages can then be installed globally or per-user using *mamba-admin package* subcommand.

Installing an application

When we are happy with our reusable application code we can then make it shareable between Mamba applications using the *package* system but we have to fit some requirements first:

- Your application root **must** contain a *docs* directory (with documentation about your shared package)
- Your application root **must** contain a *LICENSE* and *README.rst* files on it
- Your application **must** be error-free as it is going to be compiled by Mamba packaging script

When we fit all those requirements we can just use the *mamba-admin package* command line to *pack* or *install* our application:

```
$ mamba-admin package install -g
```

The above command will install the mamba application directly to the global *site-package* of our Python installation. If we can't install our application directly in the global *site-package* maybe because we don't have root access and we are not running into a *virtualenv* environment we can install the package in the *user site-package* using the *-u* option instead of *-g*. Obviously we can't use both options at the same time.

Packing an application

The goal of packing a Mamba application is *not* to distribute it as a standalone Mamba full application. Mamba packing system doesn't intend to be a replacement for distribute/setuptools/distutils/distutils2 in any way. If what you want to do is create a full Mamba installable package ready to distribute with others using *pip* or similar you better go to this (unofficial) Guide to Packaging using distribute or Distributing Python Modules from the official documentation.

Sometimes what we want to do is just pack our application to share with others or to perform a *fine tuning* on it. The *mamba-admin package* command line tool adds a subcommand to pack our application in both *.tar.gz* and *.egg* format so we can just share it with other or decompress it and perform all the changes we need to the *setup.py* file and the application structure just using it as a *valid mamba package* template before to share it.

The *pack* subcommand is used to this task and it accepts exactly the same parameters than install does except those that are specific for define the installation location (*-g* and *-u*) and adds the *-e* and *-c* one as well in order to generate an *.egg* distribution package and adds the *config* directory into the packed file if neccesary.

To pack an application we just run *mamba-admin* command line tool inside the directory of the appliction we want to pack (note that the same requirements than with the *install* command are applied here as well):

```
$ mamba-admin package pack
```

The above command will create a *mamba-<app_name>-<version>.tar.gz* file in the root directory of our application.

Installing packed applications

Install a previously packed application use the same syntax and command that install the application directly but adding the file path as the last argument:

```
$ mamnba-admin package install -g mamba-<app_name>-<version>.tar.gz
```

This will install the given file in the global *site-packages* of our Python installation.

1.5.2 How I use it?

After install a mamba application we can just add it totally or parcially into another application using the configuration file *installed_packages.json*:

```
{
    "packages": {
        "dummy_example": {
            "autoimport": true,
            "use_scripts": true
        }
    }
}
```

If *"autoimport"* is set as *true*, Mamba will add and register controllers from the installed package into our local application. That means if the installed package have a controller called *contact* our application is going to import and register it and all it's routes into our application. So if for example our imported package has a controller registered on route */shop* we can just navigate to *http://localhost:1936/shop* in our new application and we will get the *shop* controller rendered page (maybe we should add some database configuration if the shared package needs it, just read the documentation on the shared package to discover what we need to set in our application to make the package work).

If we create a new controller called *contact* in our *application/controller* directory, then our new controller will hide and replace the shared one but already registered routes will be available and this can raise an exception in the best of the cases, in the worst the application may seems to be working fine but strange behaviours are expected so be careful with this. You **have to** extend your controller from the shared one in order to don't have problems with that.

If you really need to extend the imported packages controllers, is maybe better if the *"autoimport"* option is just set as *false* and you extend whatever controller that you need. An example of a controller *contact* from an application called *dummy* extension is as follow:

```
# -*- encoding: utf-8 -*-
# -*- mamba-file-type: mamba-controller -*-
# Copyright (c) 2013 - Oscar Campos <oscar.campos@member.fsf.org>

"""
.. controller:: Shared
    :platform: Unix, Windows
    :synopsis: Shared Controller

.. controllerauthor:: Oscar Campos <oscar.campos@member.fsf.org>
"""

from mamba.application import route

from dummy.controller.contact import Contact

class Shared(Contact):
    """
    Shared Controller
    """

    name = 'Shared'
    __route__ = 'contact'

    def __init__(self):
        """
        Put your initialization code here
        """
        super(Shared, self).__init__()

    @route('/')
    def root(self, request, **kwargs):
```

```
        return super(Shared, self).root()(request, **kwargs)
```

Note how in the previous code, we are calling the function returned by *super(Shared, self).root()* that means, we are calling the wrapped function *root* from the shared controller *Contact* bypassing its own *@route* decorator, if we dont do that, we should get a 500 error because recursion depth exceeded as we are calling the *Contact* *@route* decorator instead of the real *root* method.

If *use_scripts* is set as *true*, Mamba will include all the scripts from the shared package in the *scripts* and *stylesheets* **mambaerized** resources so they are totally available into your application. You can override them by creating you own scripts with the same name in your *application/view/scripts* and *application/view/stylesheets* directories.

The same is applicable for shared templates and scripts in controller sub-directories.

In the other hand, shared templates are always included in the *Jinja2* search path so them are always available in our application. If we need to override a shared template we just have to create a template in our *application/view/templates* or *application/view/<controller>* directories and Mamba will use those instead of the shared ones.

Assets included in the *static* directory of the shared package are always available in our application *assets/* route as well. If we need to override one of them, just create a new file in our application *static* directory with the same name as the asset that we want to override.

Models

Models are always available, does not matters if autoimport is set as true or false, we can use them like:

```
from dummy.model.some_model import SomeModel

some1 = SomeModel()
```

Is always better if you subclass the shared models into your own application models, in this way you can generate the SQL schema and perform other mainteance tasks like database dump:

```
from dummy.model.some_model import SomeModel as SharedSomeModel


class SomeModel(SharedSomeMode):
    """Just a subclass of the shared model to make it available
    """
```

1.6 API Documentation

This is the Mamba API documentation.

1.6.1 Mamba Application

Objects for build Mamba Applications

class mamba.**Mamba**(*options=None*)

This object is just a global configuration for mamba applications that act as the central object on them and is

able to act as a central registry. It inherits from the :class: *~mamba.utils.borg.Borg* so you can just instantiate a new object of this class and it will share all the information between instances.

You create an instance of the *Mamba* class in your main module or in your *Twisted tac* file:

```
from mamba import Mamba
app = Mamba({'name': 'MyApp', 'description': 'My App', ...})
```

Parameters **options** (dict) – options to initialize the application with

class mamba.**ApplicationError**
ApplicationError raises when an error occurs

AppStyles

class mamba.**AppStyles**(*config_file_name='config/installed_packages.json'*)
Manager for Application Stylesheets

seealso: *Stylesheet*

get_styles()
Return the mamba.Stylesheet pool

setup()
Setup the managers

AppScripts

class mamba.application.scripts.**AppScripts**(*config_file_name='config/installed_packages.json'*)
Manager for Application Scripts

seealso: *Script*

get_scripts()
Return the mamba.Scripts pool

setup()
Setup the managers

Controllers

class mamba.application.controller.**ControllerError**
Raised on controller errors

class mamba.**Controller**(**args*, ***kwargs*)
Mamba Controller Class define a web accesible resource and its actions.

A controller can (and should) be attached to a twisted.web.resource.Resource as a child or to others *Controller*.

Unlike twisted.web.resource.Resource, *Controller* don't use the Twisted URL dispatching mechanism. The *Controller* uses *Router* for URL dispatching through the route() decorator:

```
@route('/hello_world', method='GET')
def helloworld(self, request, **kwargs):
    return 'Hello World'
```

See also:

Router, *Route*

getChild(*name*, *request*)
This method is not supposed to be called because we are overriden the full route dispatching mechanism already built in Twisted.

Class variable level `isLeaf` is supposed to be always `True` but any users can override it in their *Controller* implementation so we make sure that the native twisted behavior is never executed.

If you need Twisted native url dispatching in your site you should use `Resource` class directly in your code.

Parameters

- **name** (*string*) – ignored
- **request** (`Request`) – a `twisted.web.server.Request` specifying meta-information about the request that is being made for this child that is ignored at all.

Return type *Controller*

get_register_path()
Return the controller register path for URL Rewritting

prepare_headers(*request*, *code*, *headers*)
Prepare the back response headers

Parameters

- **request** (`Request`) – the HTTP request
- **code** (*int*) – the HTTP response code
- **headers** (dict) – the HTTP headers

render(*request*)
Render a given resource. see: `twisted.web.resource.IResource`'s render method.

I try to render a router response from the routing mechanism.

Parameters **request** (`twisted.web.server.Request`) – the HTTP request

route_dispatch(*request*)
Dispatch a route if any through the routing dispatcher.

Parameters **request** (`Request`) – the HTTP request

Returns `twisted.internet.defer.Deferred` or `mamba.web.response.WebResponse`

run(*port=8080*)
This method is used as a helper for testing purposes while you are developing your controllers.

You should never use this in production.

sendback(*result*, *request*)
Send back a result to the browser

Parameters

- **request** (`Request`) – the HTTP request
- **result** (dict) – the result for send back to the browser

class mamba.application.controller.**ControllerProvider**
Mount point for plugins which refer to Controllers for our applications.

Controllers implementing this reference should implements the IController interface

class mamba.**ControllerManager**(*store=None*, *package=None*)
Uses a ControllerProvider to load, store and reload Mamba Controllers.

_model_store A private attribute that sets the prefix path for the controllers store :param store: if is not None it sets the _module_store attr

build_controller_tree()
Build the controller's tree

get_controllers()
Return the controllers pool

is_valid_file(*file_path*)
Check if a file is a Mamba controller file

Parameters **file_path** (*str*) – the file path of the file to check

length()
Returns the controller pool length

load(*filename*)
Loads a Mamba module

Parameters **filename** (*str*) – the module filname

lookup(*module*)
Find and return a controller from the pool

Parameters **module** (*str*) – the module to lookup

lookup_path(*path*)
Lookup for a controller using its path

Parameters **path** (*str*) – the path to lookup for

reload(*module*)
Reload a controller module

Parameters **module** (*str*) – the module to reload

setup()
Setup the loder and load the Mamba plugins

Models

class mamba.application.model.**MambaStorm**(*name*, *bases*, *dict*)
Metaclass for solve conflicts when using Storm base classes

If you need to inherit your models from storm.base.Storm class in order to use references before the referenced object had been created in the local scope, you need to set your class __metaclass__ as model.MambaStorm to prevent metaclasses inheritance problems. For example:

```
class Foo(model.Model, Storm):

    __metaclass__ = model.MambaStorm
```

warning:

```
    Mamba support for database dump and SQL Schema generation through
    Storm classes is possible because a monkeypatching and hack of
    regular Storm behaviour, if you are not using Storm base classes
    for your Reference's and ReferenceSet's you may experience weird
    behaviours like not all the object columns being displayed in your
    generated schema.

    You should use mamba.application.model.MambaStorm metaclass
    and storm.base.Storm classes in order to fix it
```

class mamba.application.model.**ModelError**
Base class for Model Exceptions

class mamba.**Model**
All the models in the application should inherit from this class.

We use the new `storm.twisted.transact.Transactor` present in the 0.19 version of Storm to run transactions that will block the reactor using a `twsited.python.threadpool.ThreadPool` to execute them in different threads so our reactor will not be blocked.

You must take care of don't return any **Storm** object from the methods that interacts with the `storm.Store` underlying API because those ones are created in a different thread and cannot be used outside.

If you don't want any of the methods in your model to run asynchronous inside the transactor you can set the class property *__mamba_async__* as *False* and them will run synchronous in the main thread (blocking the reactor in the process).

We don't care about the instantiation of the **Storm** `Store` because we use `zope.transaction` through `storm.zope.zstorm.ZStorm` that will take care of create different instances of `Store` per thread for us.

classmethod all (*klass*, *order_by=None*, *desc=False*, **args*, ***kwargs*)
Return back all the rows in the database for this model

Parameters

- **order_by** (*model property*) – order the resultset by the given field/property
- **desc** (*bool*) – if True, order the resultset by descending order

New in version 0.3.6.

copy (*orig*)
Copy this object properties and return it

create (**args*, ***kwargs*)
Create a new register in the database

create_table (**args*, ***kwargs*)
Create the table for this model in the underlying database system

delete (**args*, ***kwargs*)
Delete a register from the database

dict (*traverse=True*, *json=False*, **parent*, ***kwargs*)
Returns the object as a dictionary

Parameters

- **traverse** (*bool*) – if True traverse over references
- **json** (*bool*) – if True we convert datetime to string and Decimal to float
- **fields** – If set we filter only the fields specified,

mutually exclusive with exclude, having precedence if both are set. :type fields: list :param exclude: If set we exclude the fields specified, mutually exclusive with fields, not working if you also set fields. :type exclude: list

drop_table(**args*, ***kwargs*)
Delete the table for this model in the underlying database system

dump_data(*scheme=None*)
Dumps the SQL data

dump_indexes()
Dump SQL indexes (used by PostgreSQL and SQLite)

dump_references()
Dump SQL references (used by PostgreSQL)

dump_table()
Dumps the SQL command used for create a table with this model

Parameters

- **schema** (*str*) – the SQL schema, SQLite by default
- **force_drop** (*bool*) – when True drop the tables always

classmethod find(*klass*, **args*, ***kwargs*)
Find an object in the underlying database

Some examples:

model.find(Customer.name == u"John") model.find(name=u"John") model.find((Customer, City), Customer.city_id == City.id)

New in version 0.3.6.

get_adapter()
Get a valid adapter for this model

get_primary_key()
Return back the model primary key (or keys if it's compound)

get_uri()
Return an URI instance using the uri config for this model

json
Returns a JSON representation of the object (if possible)

classmethod mamba_database()
Return back the configured underlying mamba database (if any)

on_schema()
Checks if Mamba should take care of this model.

pickle
Returns a Pyhon Pickle representation of the object (if possible)

classmethod read(**args*, ***kwargs*)
Read a register from the database. The give key (usually ID) should be a primary key.

Parameters **id** (*int*) – the ID to get from the database

store(*database=None*)
Return a valid Storm store for this model

update (**args*, ***kwargs*)
Update a register in the database

uri
Returns the database URI for this model

class mamba.application.model.**ModelProvider**
Mount point for plugins which refer to Models for our applications

class mamba.**ModelManager** (*store=None*, *package=None*)
Uses a ModelProvider to load, store and reload Mamba Models.

_model_store A private attribute that sets the prefix path for the models store :param store: if is not None it sets the _module_store attr

get_models()
Return the models pool

is_valid_file (*file_path*)
Check if a file is a Mamba model file

Parameters **file_path** (*str*) – the file path of the file to check

length()
Returns the controller pool length

load (*filename*)
Loads a Mamba module

Parameters **filename** (*str*) – the module filname

lookup (*module*)
Find and return a controller from the pool

Parameters **module** (*str*) – the module to lookup

reload (*module*)
Reload a controller module

Parameters **module** (*str*) – the module to reload

setup()
Setup the loder and load the Mamba plugins

1.6.2 Mamba Core

Core components of the Mamba framework itself.

Interfaces

Interfaces that are part of the Mamba Core, you are supossed to never use those ones unless you are developing new features for the Mamba framework itself.

class mamba.core.interfaces.**INotifier**
Every Inotifier class will implement this interface

_notify (*ignore*, *file_path*, *mask*)

Parameters

- **ignore** – ignored parameter

- **file_path** – twisted.python.filepath.Filepath on which the event happened
- **mask** (*int*) – inotify event as hexadecimal mask

notifier
A twisted.internet.inotify.INotify instance where to watch a FilePath

class mamba.core.interfaces.**IController**
Manba Controllers interface.

Every controller will implement this interface

Parameters

- **name** (*str*) – Controller's name
- **desc** (*str*) – Controller's description
- **loaded** (*bool*) – true if the controller has been loaded, otherwise returns False

class mamba.core.interfaces.**IDeployer**
Mamba Deployers interface.

Every deployer must implement this interface

class mamba.core.interfaces.**IResponse**
Mamba Web Response interface.

Every web response must implement this interface.

Parameters

- **code** (*int*) – the HTTP response code
- **body** (*string*) – the HTTP response body
- **headers** (dict) – the HTTP response headers

class mamba.core.interfaces.**IMambaSQL**
Mamba SQL interface.

I'm usefull to create common SQL queries for create or alter tables for all the SQL backends supported by Mamba.

Parameters **original** – the original underlying SQL backend type

class mamba.core.interfaces.**ISession**
Mamba Session interface. I'm just a Session interface used to store objects in Twisted Sessions

Parameters **session** – A mamba session object

Decorators

mamba.core.decorators.**cache**(*size=16*)
Cache the results of the function if the same positional arguments are provided.

We only store the size provided (if any) in MB, after that we should perform FIFO queries until the size of the cache is lower than the provided one.

If the size is 0 then an unlimited cache is provided

Notice

The memory size of the int_cache is just an approximation

mamba.core.decorators.**unlimited_cache**(*func*)
Just a wrapper over cache decorator to alias `@cache(size=0)()`

Core Services

Core Services used by mamba applications

class mamba.core.services.threadpool.**ThreadPoolService**(*pool*)
Service to being started by twistd

This service handles and serve the ThreadPool that is used by Storm when we use the @transact decorator in out queries to use the Twisted integration

Packages

New in version 0.3.6.

Mamba packages are used by the Mamba reusability system to make posible the reuse of mamba applications and components between mamba applications.

class mamba.core.packages.**PackagesManager**(*config_file='config/installed_packages.json'*)
The PackagesManager is used to register the shared packages that our mamba applications are going to import and use. The manager is instanced once by the `Mamba` borg and create and load controllers and models managers for every package that we want to use in our application.

Parameters **config_file** (*str*) – the path to config file that we want to use

New in version 0.3.6.

register_packages()
Register packages found in the config file.

Styles and Scripts are a bit special and we register the main application ones here so we can refer all of them from the *Resource* subclasses and they will be auto imported in our templates

Resource

Mamba Resource object extends Twisted web Resources mainly to integrate the Jinja2 templating system

class mamba.core.resource.**Resource**
Mamba resources base class. A web accessible resource that add common properties for scripts in Mamba applications

Session

class mamba.core.session.**MambaSession**(*session*)
An authed session store object

class mamba.core.session.**Session**(*site*, *uid*, *reactor=None*)
Mamba session wrapper

authenticate()
Set authed as True

is_authed()
Return the authed value

set_lifetime(*lifetime*)
Set the sessionTimeout to lifetime value

Parameters **lifetime** (*int*) – the value to set as session timeout in seconds

Templating

Mamba integrates the Jinja2 templating system as a core component of the framework.

class mamba.core.templating.**MambaTemplate**(*env=None*, *template=None*)
This class loads templates from the Mamba package and is used internally by Mamba. You are not supossed to use this class in your code

Parameters

- **env** (jinja2.Environment) – jinja2 environment
- **template** (*str*) – the template to render

render(***kwargs*)
Renders a template

class mamba.core.templating.**Template**(*env=None*, *controller=None*, *size=50*, *template=None*)
This class loads and render templates from Mamba Applications view and controller directories.

If controller is not None, then we use the controller directory templates instead of global view ones.

Parameters

- **env** – the Jinja2 environment
- **controller** (Controller) – mamba controller that uses the templates
- **cache_size** – size of the Jinja2 templating environment

Raises TemplateNotFound if no template is found in the filesystem

Raises NotConfigured it no parameters are provided

render(*template=None*, ***kwargs*)
Renders a template and get the result back

Parameters **template** (*string*) – the template to render

class mamba.core.templating.**TemplateError**
Base class for Mamba Template related exceptions

class mamba.core.templating.**NotConfigured**
Raised when we are missing configuration

1.6.3 Deployment

Mamba integrates the Fabric deployment library and it's used by Mamba itself to release new versions and deploy the framework to the live mamba web site. To see an example of usage you can check the mamba devel package on GitHub.

class mamba.deployment.deployer.**DeployerImporter**(*filename*)
Imports DeployerConfig files in a very similar way as __import__ does

mamba.deployment.deployer.**deployer_import**(*name*, *path*)
Import a deployer configuration file as Python code and initializes it

class mamba.deployment.fabric_deployer.**FabricDeployer**
Fabric Deployment System compatible with Mamba Plugabble Interface

deploy(*config_file=None*)
Deploys the system following the configuration in the config file

identify()
Returns the deployer identity

load(*config_file*)
Load the workflow rules from a Mamba .dc Python file

Parameters **config_file** (*str*) – The file where to load the configuration from

operation_mode()
Return the operation mode for this deployer

class mamba.deployment.fabric_deployer.**FabricMissingConfigFile**
Fired when missing config file is detected

class mamba.deployment.fabric_deployer.**FabricConfigFileDontExists**
Fired when the config file does not exists

class mamba.deployment.fabric_deployer.**FabricNotValidConfigFile**
Fired when the config file is not valid

1.6.4 Http

Mamba Http Headers

class mamba.http.headers.**Headers**
An object that build the Application page header and returns it as a well formated *XHTML/HTML* string.

get_description_content()
Returns the Headers description

get_doctype()
Get the configured or default mamba doctype (html by default)

get_favicon_content(*media='/media'*)
Returns the favicon

Parameters **media** (*str*) – a media directory to add, defaults to */media*

get_generator_content()
Returns the meta generator content

get_language_content()
Returns the Headers language

get_mamba_content()
Returns mamba specific meta content

1.6.5 Utils

Subpackage containing the modules that implement common utilities

Borg

class mamba.utils.borg.**Borg**
The Mamba Borg Class.

Every object created using the Borg pattern will share their information, as long as they refer to the same state information. This is a more elegant type of singleton, but, in other hand, Borg objects doesn't have the same ID, every object have his own ID

Borg objects doesn't share data between inherited classes. If a class A inherits from Borg and a class B inherits from A then A and B doesn't share the same namespace

Example:

```
class LockerManager(borg.Borg):

    def __init__(self):
        super(LockerManager, self).__init__()
```

Used as:

```
>>> from managers import LockerManager
>>> manager1 = LockerManager()
>>> manager1.name = 'Locker One'
>>> manager2 = LockerManager()
>>> print(manager2.name)
Locker One
>>>
```

CamelCase

class mamba.utils.camelcase.**CamelCase**(*camelize*)
Stupid class that just offers stupid CamelCase funcionallity

Parameters **camelize** (*str*) – the string to camelize

camelize(*union=False*)
Camelize and return camelized string

Parameters **union** (*bool*) – if true is will use a space between words

Converter

class mamba.utils.converter.**Converter**
Object Converter class

static fix_common(*value*)
Fix commons uncommons

static serialize(*obj*)
Serialize an object and returns it back

config.Database

This class is used to load configuration files in JSON format from the file system. If no config is provided a basic configuration based on SQLite is automatically created for us.

class mamba.utils.config.**Database**(*config_file='config/database.json'*)

Database configuration object

This object load and parses the configuration details for the database access using a JSON configuration file with this format:

```
{
    'uri': 'sqlite:',
    'min_threads': 5,
    'max_threads': 20,
    'auto_adjust_pool_size': false,
    'create_table_behaviours': {
        'create_table_if_not_exists': true,
        'drop_table': false
    },
    'drop_table_behaviours': {
        'drop_if_exists': true,
        'restrict': true,
        'cascade': false
    }
}
```

Where uri is the Storm URI format for create ZStores and min, max threads are the minimum and maximum threads in the thread pool for operate with the database. If auto_adjust_pool_size is True, the size of the thread pool should be adjust dynamically.

For *create_table_bevaviour* possible values are:

create_if_not_exists this is the default behaviour and it should add 'IF DONT EXISTS' to the table creation scripts

drop_table this behaviour will always drop a table before try to create it. Be carefull with this behaviour because you can lose all your data if you dont take care of it

If no config_file or invalid config_file is given at first load attempt, then a fallback default settings with a SQLite in memory table are returned back.

If you loaded a valid config file and you instance the database config again with an invalid JSON format file, then a fallback default settings with SQLite in memory URI is returned back in order to preserve your data (if we don't fallback to a default configuration you can overwrite important data in your previous well configured environment).

If you loaded a valid config file and pass a non existent or a void file in the constructor you get back your previous config so you can use it like a singleton instance:

```
config.Database('path/to/my_valid/config.json')
...
cfg = config.Database()
```

If you want to clear your config and return it back to a default state you should pass 'clean' or 'default' as parameter:

```
config.Database('default')
```

If the URI property is a dictionary, mamba will then create configuration and conections for each element in the dictionary using the key as ZStorm name and the value as the URI. Then you can get speific stores for specific URI's using the store method in the database object.

Parameters **config_file** (*str*) – the file to load the configuration from, it can (and should) be empty to get back the previous configured data

static write (*options*)
Write options to the configuration file

Parameters **options** (dict) – the options from the mamba-admin commands tool

config.Application

As with the previous one, this class is used to load configuration related with the application from files in JSON format in the file system. If no configuration file is provided, a basic configuration is automatically created for us.

class mamba.utils.config.**Application**(*config_file=''*)
Application configuration object

This object loads and parses the Mamba application configuration options using a JSON file with the following format:

```
{
    "name": "Mamba Application",
    "description": "Mamba application description",
    "version": 1.0
    "port": 8080,
    "doctype": "html",
    "content_type": "text/html",
    "description": "My cool application",
    "language": "en",
    "description": "This is my cool application",
    "favicon": "favicon.ico",
    "platform_debug": false,
    "development": true,
    "debug": false
}
```

If we want to force the mamba application to run under some specific twisted reactor, we can add the *"reactor"* option to the configuration file to enforce mamba to use the configured reactor.

Parameters **config_file** (*str*) – the JSON file to load

config.InstalledPackages

New in version 0.3.6.

This is used to load configuration realated to the Mamba reusability system. If no configuration file is provided (or it doesn't exists), a basic configuration is automatically created for us.

class mamba.utils.config.**InstalledPackages**(*config_file=''*)
Instaleld Packages configuration object

This object loads and parses configuration that indicates to the Mamba framework that this application have to import and register the indicated shared packages.

The JSON file must follow this format:

```
{
    "packages": {
        "package_one": {"autoimport": true, "use_scripts": true},
        "package_two": {"autoimport": true, "use_scripts": false},
        "package_three": {"autoimport": false, "use_scripts": true}
    }
}
```

The packages *must* be installed already in the system

Parameters **config_file** (*str*) – the JSON file to load

Less

class mamba.utils.less.**LessCompiler**(*style*, *exe='lessc'*, *path=None*)
Compile LESS scripts if LESS NodeJS compiler is present. Otherwise adds the less.js JavaScript compiler to the page.

compile()
Compile a LESS script

class mamba.utils.less.**LessResource**(*path=None*)
Mamba LessResource class define a web accesible LESS script

entityType = <InterfaceClass twisted.web.resource.IResource>

getChild(*path*, *request*)
Retrieve a 'child' resource from me.

Implement this to create dynamic resource generation – resources which are always available may be registered with self.putChild().

This will not be called if the class-level variable 'isLeaf' is set in your subclass; instead, the 'postpath' attribute of the request will be left as a list of the remaining path elements.

For example, the URL /foo/bar/baz will normally be:

```
| site.resource.getChild('foo').getChild('bar').getChild('baz').
```

However, if the resource returned by 'bar' has isLeaf set to true, then the getChild call will never be made on it.

Parameters and return value have the same meaning and requirements as those defined by L{IResource.getChildWithDefault}.

getChildWithDefault(*path*, *request*)
Retrieve a static or dynamically generated child resource from me.

First checks if a resource was added manually by putChild, and then call getChild to check for dynamic resources. Only override if you want to affect behaviour of all child lookups, rather than just dynamic ones.

This will check to see if I have a pre-registered child resource of the given name, and call getChild if I do not.

@see: L{IResource.getChildWithDefault}

putChild(*path*, *child*)
Register a static child.

You almost certainly don't want '/' in your path. If you intended to have the root of a folder, e.g. /foo/, you want path to be ''.

@see: L{IResource.putChild}

render(*request*)
Render a given resource. See L{IResource}'s render method.

I delegate to methods of self with the form 'render_METHOD' where METHOD is the HTTP that was used to make the request. Examples: render_GET, render_HEAD, render_POST, and so on. Generally you should implement those methods instead of overriding this one.

render_METHOD methods are expected to return a byte string which will be the rendered page, unless the return value is C{server.NOT_DONE_YET}, in which case it is this class's responsibility to write the results using C{request.write(data)} and then call C{request.finish()}.

Old code that overrides render() directly is likewise expected to return a byte string or NOT_DONE_YET.

@see: L{IResource.render}

render_GET(*request*)
Try to compile a LESS file and then serve it as CSS

render_HEAD(*request*)
Default handling of HEAD method.

I just return self.render_GET(request). When method is HEAD, the framework will handle this correctly.

Output

Output printing functionallity based and partially taken from Gentoo portage system

1.6.6 Web

Subpackage containing the modules that implement web stuff for projects

AsyncJSON

class mamba.web.asyncjson.**AsyncJSON**(*value*)
Asynchronous JSON response.

I use a cooperate Twisted task in order to create a producer that send huge amounts of JSON data in an asynchronous way.

If the data being serialized into JSON is huge, the serialization process can take longest than the browser waiting response and can block itself the web server, preventing other requests from being serviced. This class prevents that type of inconveniences.

This class is based on Jean Paul Calderone post at: http://jcalderone.livejournal.com/55680.html

Page

class mamba.web.**Page**(*app*, *template_paths=None*, *cache_size=50*, *loader=None*)
This represents a full web page in mamba applications. It's usually the root page of your web site/application.

The controllers for the routing system are eregistered here. We first register any package shared controller because we want to overwrite them if our application defines the same routes.

Parameters

- **app** (Application) – The Mamba Application that implements this page
- **template_paths** – additional template paths for resources
- **cache_size** – the cache size for Jinja2 Templating system
- **loader** – Jinja2 custom templating loader

add_script(*script*)
Adds a script to the page

add_template_paths(*paths*)
Add template paths to the underlying Jinja2 templating system

entityType = <InterfaceClass twisted.web.resource.IResource>

generate_dispatches()
Generate singledispatches

getChild(*path*, *request*)
If path is an empty string or index, render_GET should be called, if not, we just look at the templates loaded from the view templates directory. If we find a template with the same name than the path then we render that template.

> **Caution:** If there is a controller with the same path than the path parameter then it will be hidden and the template in templates path should be rendered instead

Parameters

- **path** (*str*) – the path
- **request** – the Twisted request object

getChildWithDefault(*path*, *request*)
Retrieve a static or dynamically generated child resource from me.

First checks if a resource was added manually by putChild, and then call getChild to check for dynamic resources. Only override if you want to affect behaviour of all child lookups, rather than just dynamic ones.

This will check to see if I have a pre-registered child resource of the given name, and call getChild if I do not.

@see: L{IResource.getChildWithDefault}

initialize_templating_system(*template_paths*, *cache_size*, *loader*)
Initialize the Jinja2 templating system for static HTML resources

insert_scripts()
Insert scripts to the HTML

insert_stylesheets()
Insert stylesheets into the HTML

putChild(*path*, *child*)
Register a static child.

You almost certainly don't want '/' in your path. If you intended to have the root of a folder, e.g. /foo/, you want path to be ''.

@see: L{IResource.putChild}

register_controllers()
Add a child for each controller in the ControllerManager

register_shared_controllers()
Add a child for each shared package controller. If the package includes a static files directory we add an asset for it

New in version 0.3.6.

render(*request*)
Render a given resource. See L{IResource}'s render method.

I delegate to methods of self with the form 'render_METHOD' where METHOD is the HTTP that was used to make the request. Examples: render_GET, render_HEAD, render_POST, and so on. Generally you should implement those methods instead of overriding this one.

render_METHOD methods are expected to return a byte string which will be the rendered page, unless the return value is C{server.NOT_DONE_YET}, in which case it is this class's responsibility to write the results using C{request.write(data)} and then call C{request.finish()}.

Old code that overrides render() directly is likewise expected to return a byte string or NOT_DONE_YET.

@see: L{IResource.render}

render_GET (*request*)
Renders the index page or other templates of templates directory

render_HEAD (*request*)
Default handling of HEAD method.

I just return self.render_GET(request). When method is HEAD, the framework will handle this correctly.

run (*port=8080*)
Method to run the application within Twisted reactor

This method exists for testing purposes only and fast controller test-development-test workflow. In production you should use twistd

Parameters **port** (*number*) – the port to listen

Response

class mamba.web.response.**Response** (*code*, *subject*, *headers*)
Mamba web request response base dummy object

Parameters

- **code** (*int*) – the HTML code for the response
- **subject** (Response or dict or str) – the subject body of he response
- **headers** (*dict or a list of dicts*) – the HTTP headers to return back in the response to the browser

class mamba.web.response.**Ok** (*subject=''*, *headers={}*)
Ok 200 HTTP Response

Parameters

- **subject** (Response or dict or str) – the subject body of the response
- **headers** (*dict or a list of dicts*) – the HTTP headers to return back in the response to the browser

class mamba.web.response.**Created** (*subject=''*, *headers={}*)
Ok Created 201 HTTP Response

Parameters

- **subject** (Response or dict or str) – the subject body of the response
- **headers** (*dict or a list of dicts*) – the HTTP headers to return back in the response to the browser

class mamba.web.response.**MovedPermanently** (*url*)
Ok 301 Moved Permanently HTTP Response

Parameters **url** (*str*) – the url where the resource has been moved

See also:

https://tools.ietf.org/html/rfc2616#page-62

class mamba.web.response.**Found**(*url*)
Ok 302 Found HTTP Response

Parameters **url** (*str*) – the url where we want to redirect the browser

class mamba.web.response.**SeeOther**(*url*)
Ok 303 See Other HTTP Response

Parameters **url** (*str*) – the url where to find the information via GET

class mamba.web.response.**NotFound**(*subject*, *headers={}*)
Error 404 Not Found HTTP Response

Parameters

- **subject** (Response or dict or str) – the subject body of he response
- **headers** (*dict or a list of dicts*) – the HTTP headers to return back in the response to the browser

class mamba.web.response.**BadRequest**(*subject=''*, *headers={}*)
BadRequest 400 HTTP Response

Parameters

- **subject** (Response or dict or str) – the subject body of he response
- **headers** (*dict or a list of dicts*) – the HTTP headers to return back in the response to the browser

class mamba.web.response.**Unauthorized**(*subject='Unauthorized'*, *headers={}*)
Unauthorized 401 HTTP Response

class mamba.web.response.**Conflict**(*subject*, *value*, *message=''*)
Error 409 Conflict found

Parameters

- **subject** (Response or dict or str) – the subject body of he response
- **value** – the value of the conflicted operatio
- **message** (*str*) – a customer user messahe for the response

class mamba.web.response.**AlreadyExists**(*subject*, *value*, *message=''*)
Error 409 Conflict found in POST

Parameters

- **subject** (Response or dict or str) – the subject body of he response
- **value** – the value of the conflicted operatio
- **message** (*str*) – a customer user messahe for the response

class mamba.web.response.**NotImplemented**(*url*, *message=''*)
Error 501 Not Implemented

Parameters

- **url** (*str*) – the URL that is not implemented

- **message** (*str*) – a user custom message describing the problem

class mamba.web.response.**InternalServerError**(*message*)
Error 500 Internal Server Error

Parameters **message** (*str*) – a user custom message with a description of the nature of the error

Routing

class mamba.web.**Route**(*method*, *url*, *callback*)
I am a Route in the Mamba routing system.

compile()
Compiles the regex matches using the complete URL

validate(*dispatcher*)
Validate a given path against stored URLs. Returns None if nothing matched itself otherwise

Parameters **dispatcher** (*RouteDispatcher*) – the dispatcher object that containing the information to validate

class mamba.web.**Router**
I store, lookup, cache and dispatch routes for Mamba

A route is stores as: [methods][route][Controller.__class__.__name__]

dispatch(*controller*, *request*)
Dispatch a route and return back the appropiate response.

Parameters

- **controller** (*Controller*) – the mamba controller
- **request** (twisted.web.server.Request) – the HTTP request

install_routes(*controller*)
Install all the routes in a controller.

Parameters **controller** (*Controller*) – the controller where to fid routes

register_route(*controller*, *route*, *func_name*)
Method that register a route for the given controller

Parameters

- **controller** (*Controller*) – the controller where to register the route
- **route** (*Route*) – the Route to register
- **func_name** (*str*) – the callable object name

route(*url*, *method='GET'*)
Register routes for controllers or full REST resources.

class mamba.web.**RouteDispatcher**(*router*, *controller*, *request*, *url=True*)
Look for a route, compile/process if neccesary and return it

lookup()
I traverse the URLs at router picking up the ones that match the controller name and then process it to validate which ones match by path/arguments to a particular Route

If nothing match just returns None

Scripts

class mamba.web.**Script**(*path=''*, *prefix='scripts'*)
Object that represents an script

Parameters

- **path** (*str*) – the path of the script
- **prefix** (*str*) – the prefix where the script reside

class mamba.web.**ScriptManager**(*scripts_store=None*)
Manager for Scripts

load(*filename*)
Load a new script file

lookup(*key*)
Find and return a script from the pool

setup()
Setup the loader and load the scripts

class mamba.web.**ScriptError**
Generic class for Script exceptions

class mamba.web.script.**InvalidFile**
Fired if a file is lacking the mamba css or less headers

class mamba.web.script.**InvalidFileExtension**
Fired if the file has not a valid extension (.css or .less)

class mamba.web.script.**FileDontExists**
Raises if the file does not exists

Stylesheets

class mamba.web.**Stylesheet**(*path=''*, *prefix='styles'*)
Object that represents an stylesheet or a less script

Parameters

- **path** (*str*) – the path of the stylesheet
- **prefix** (*str*) – the prefix where the stylesheets reside

class mamba.web.**StylesheetError**
Generic class for Stylesheet exceptions

class mamba.web.**InvalidFile**
Fired if a file is lacking the mamba css or less headers

class mamba.web.**InvalidFileExtension**
Fired if the file has not a valid extension (.css or .less)

class mamba.web.**FileDontExists**
Raises if the file does not exists

Url Sanitizer

class mamba.web.url_sanitizer.**UrlSanitizer**
Sanitize URLs for a correct use

WebSockets

class mamba.web.websocket.**WebSocketProtocol**(**args*, ***kwargs*)

Wrapped protocol to handle Websockaet transport layer. Thw websocket protocol just wrap another protocol to provide a WebSocket transport.

warning:: This protocol is not HTTP

How to use it?:

```
from twisted.internet import protocol

from mamba.web import websocket

class EchoFactory(protocol.Protocol):

    def dataReceived(self, data):
        # this is the websocket data frame
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    protocol = EchoProtocol


reactor.listenTCP(6543, websocket.WebSocketFactory(EchoFactory()))
```

New in version 0.3.6.

close(*reason=''*)

Close the connection.

This includes telling the other side we are closing the connection.

If the other ending didn't signal that the connection is beign closed, then we might not see their last message, but since their last message should, according to the specification, be a simpel acknowledgement, it shouldn't be a problem.

Refer to [RFC6455][Page 35][Page 41] for more details

codecs

Return the list of available codecs (WS protocols)

complete_hybi00(*challenge*)

Generate the response for a HyBi-00 challenge.

dataReceived(*data*)

Protocol dataReceived

If our state is HANDSHAKE then we capture the request path fo echo it back to those browsers that require it (Chrome mainly) and set our state to NEGOTIATION

If our state is NEGOTIATION we look at the headers to check if we have one complete set. If we can't validate headers we just lose connection

If out state is CHALLENGE we have to call authentication related code for protocol versions HyBi-00/Hixie-76. For more information refer to http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76#page-5

If our state is FRAMES then we parse it.

We always kick any pending frames after each call to *dataReceived*. This is neccesary because frames might have started being sended early we can get write()s from our protocol above when they makeCon-

nection() inmediately before the browser actually send any data. In those cases, we need to manually kick pending frames.

handle_challenge()
Handle challenge. This is exclusive to HyBi-00/Hixie-76

handle_frames()
Use the correct frame parser and send parsed data to the underlying protocol

handle_handshake()
Handle initial request. These look very much like HTTP requests but aren't. We have to capture the request path to echo it back to the browsers that care about.

These lines looks like:

```
GET /some/path/to/a/websocket/resource HTTP/1.1
```

handle_negotiation()
Handle negotiations here. We perform some basic checks here, for example we check if the protocol being used is WebSocket and if we support the version used by the browser.

In the case that we don't support the broser version wew send back an HTTP/1.1 404 Bad Request message with a list of our supported protocol versions as is defined in the [RFC6455][Section 4.4]

is_hybi00
Determine whether a given set of headers are HyBi-00 compilant

parse_headers(*headers*)
Parse raw HTTP headers.

Parameters **headers** (*str*) – the headers to parse

secure
Borrowed technique for determining whether this connection is over SSL/TLS

write(*data*)
Write to the transport.

Parameters **data** – the buffer data to write

writeSequence(*data*)
Write a sequence of data to the transport

Parameters **data** – the sequence to be written

class mamba.web.websocket.**WebSocketFactory**(*wrappedFactory*)
Factory that wraps another factory to provide WebSockets transports for all of its protocols

protocol
alias of WebSocketProtocol

class mamba.web.websocket.**HyBi07Frame**(*buf*)
A WebSocket HiBi-07+ frame object representation

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
```

```
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

For more information about WebSocket framing read [RFC6455][Section 5.2]

Parameters **buf** – the buffer data

generate(*opcode=1*)
Generate a HyBi-07+ frame.

This function always creates unmasked frames, and attemps to use the smallest possible lengths.

Parameters **opcode** (*int*) – the opcode to use: 0x1 -> Text 0x2 -> Binary

mask(*buf*, *key*)
Mask or unmask a buffer of bytes with a masking key

The masking key is a 32-bit value chosen by the browser client. The key shouldn't affect the length of the *Payload data*. The used algorithm is the following. Octet i of the transformed data is the XOR of octet i of the original data with octet at index i modulo 4 of the masking key:

```
j = i % 4
octet[i] = octet[i] ^ key[j]
```

This means that if a third party have access to the key used by our connection we are exposed. For more information about this please, refer to [RFC6455][Page31]

Parameters

- **buf** – the buffer to mask or unmask
- **key** – the masking key, it shoudl be exactly four bytes long

parse()
Parse HyBi-07+ frame.

class mamba.web.websocket.**HyBi00Frame**(*buf*)
A WebSocket HyBi-00 frame object representation

```
Frame type byte <-------------------------------------.
    |      |                                          |
    |      `--> (0x00 to 0x7F) --> Data... --> 0xFF -->-+
    |                                                  |
    `--> (0x80 to 0xFE) --> Length --> Data... ------->-'
```

The implementation of this protocol is really simple, in HyBi-00/Hixie-76 only 0x00-0xFF is valid so we use 0x00 always as opcode and then put the buffer between the opcode and the last 0xFF.

Only 0x00 to 0xFE values are allowed in the buffer (because the 0xFF is used mark for the end of the buffer).

For more information about this read [HyBi-00/Hixie-76][Page 6]

Parameters **buf** – the buffer data

generate(*opcode=0*)
Generate a HyBi-00/Hixie-76 frame.

is_valid
Check if the buffer is valid (no xff characters on it)

parse()
Parse a HyBi-00/Hixie-76 frame.

class mamba.web.websocket.**InvalidProtocolVersionPreamble**
Send invalid protocol version response

class mamba.web.websocket.**HandshakePreamble**
Common HandShake preamble class for all protocols

write_to_transport(*transport*)
Write data to the wrapped transport

Parameters **transport** – the underlying transport

class mamba.web.websocket.**HyBi07HandshakePreamble**(*protocol*)
HyBi-07 preamble

generate_accept_opening(*protocol*)
Generates the accept response for a given key.

Concatenates the *Sec-WebSocket-Key* given by the client with the string 258EAFA5-E914-47DA-95CA-C5AB0DC85B11 to form a response that proves the handshake was received by the server by taking the SHA-1 hash of this encoding it to base64 and echoing back to the client.

Refer to [RFC6455][Page 7] for more details.

class mamba.web.websocket.**HyBi00HandshakePreamble**(*protocol*)
HyBi-00 preamble

write_to_transport(*transport*, *response*)
Write data to the wrapped transport and add a hibi00 auth response

Parameters

- **transport** – the underlying transport
- **response** – the generated HyBi-00/Hixie-76 response

class mamba.web.websocket.**WebSocketError**
Fired when something went wrong

class mamba.web.websocket.**NoWebSocketCodec**
Fired when we can't handle the codec (WS protocol)

class mamba.web.websocket.**InvalidProtocolVersion**
Fired when the protocol version is not supported

class mamba.web.websocket.**InvalidCharacterInHyBi00Frame**
Fired when a xff charecter is found in HyBi-00/Hixie-76 frame buffer

class mamba.web.websocket.**ReservedFlagsInFrame**
Fired when any of the three reserved bits of HyBi-07 are set on

class mamba.web.websocket.**UnknownFrameOpcode**
Fired when we get an unused RFC6455 frame opcode

1.6.7 Enterprise

This is the package that give you access to Database layers. You can use traditional Open Source SQL solutions as PostgreSQL PostgreSQL, MySQL or SQLite as well as No-SQL Open Source solutions as MongoDB (work in

progress).

The SQL database access is performed through Storm with some monkey patching to make possible database creation from the Python defined model.

Mamba is supossed to work fine with Storm since revision 223 of the bazaar repository in Storm-0.12 but we only tested it with version 0.19.

SQL through Storm

class `mamba.`**`Database`**(*pool=None*, *testing=False*)

Storm ORM database provider for Mamba.

Parameters **`pool`** (`twisted.python.threadpool.ThreadPool`) – the thrad pool for this database

`adjust_poolsize`(*min_threads=None*, *max_threads=None*)

Adjusts the underlying threadpool size

Parameters

- **`min`** (*int*) – minimum number of threads
- **`max`** (*int*) – maximum number of threads

`backend`

Return the type or types of backends this databse is using

`database`

Return the database name or names we are using

`dump`(*model_manager*, *scheme=None*, *full=False*)

Dumps the full database

Parameters

- **`model_manager`** (`ModelManager`) – the model manager from mamba application
- **`scheme`** (*str*) – dump which scheme? if None just everything
- **`full`** (*bool*) – should be dumped full?

`host`

Return the hostname or hostnames this database is using

`reset`(*model_manager*, *scheme=None*)

Delete all the data in the database and return it to primitive state

Parameters

- **`model_manager`** (`ModelManager`) – the model manager from mamba application
- **`scheme`** (*str*) – the specific scheme to reset (if any)

`start`()

Starts the Database (and the threadpool)

`stop`()

Stops the Database (and the threadpool)

`store`(*database='mamba'*, *ensure_connect=False*)

Returns a Store per-thread through `storm.zope.zstorm.ZStorm`

class mamba.**Model**

All the models in the application should inherit from this class.

We use the new `storm.twisted.transact.Transactor` present in the 0.19 version of Storm to run transactions that will block the reactor using a `twsited.python.threadpool.ThreadPool` to execute them in different threads so our reactor will not be blocked.

You must take care of don't return any **Storm** object from the methods that interacts with the `storm.Store` underlying API because those ones are created in a different thread and cannot be used outside.

If you don't want any of the methods in your model to run asynchronous inside the transactor you can set the class property *__mamba_async__* as *False* and them will run synchronous in the main thread (blocking the reactor in the process).

We don't care about the instantiation of the **Storm** `Store` because we use `zope.transaction` through `storm.zope.zstorm.ZStorm` that will take care of create different instances of `Store` per thread for us.

classmethod all (*klass*, *order_by=None*, *desc=False*, **args*, ***kwargs*)

Return back all the rows in the database for this model

Parameters

- **order_by** (*model property*) – order the resultset by the given field/property
- **desc** (*bool*) – if True, order the resultset by descending order

New in version 0.3.6.

copy (*orig*)

Copy this object properties and return it

create (**args*, ***kwargs*)

Create a new register in the database

create_table (**args*, ***kwargs*)

Create the table for this model in the underlying database system

delete (**args*, ***kwargs*)

Delete a register from the database

dict (*traverse=True*, *json=False*, **parent*, ***kwargs*)

Returns the object as a dictionary

Parameters

- **traverse** (*bool*) – if True traverse over references
- **json** (*bool*) – if True we convert datetime to string and Decimal to float
- **fields** – If set we filter only the fields specified,

mutually exclusive with exclude, having precedence if both are set. :type fields: list :param exclude: If set we exclude the fields specified, mutually exclusive with fields, not working if you also set fields. :type exclude: list

drop_table (**args*, ***kwargs*)

Delete the table for this model in the underlying database system

dump_data (*scheme=None*)

Dumps the SQL data

dump_indexes ()

Dump SQL indexes (used by PostgreSQL and SQLite)

dump_references()
Dump SQL references (used by PostgreSQL)

dump_table()
Dumps the SQL command used for create a table with this model

Parameters

- **schema** (*str*) – the SQL schema, SQLite by default
- **force_drop** (*bool*) – when True drop the tables always

classmethod find(*klass*, **args*, ***kwargs*)
Find an object in the underlying database

Some examples:

model.find(Customer.name == u"John") model.find(name=u"John") model.find((Customer, City), Customer.city_id == City.id)

New in version 0.3.6.

get_adapter()
Get a valid adapter for this model

get_primary_key()
Return back the model primary key (or keys if it's compound)

get_uri()
Return an URI instance using the uri config for this model

json
Returns a JSON representation of the object (if possible)

classmethod mamba_database()
Return back the configured underlying mamba database (if any)

on_schema()
Checks if Mamba should take care of this model.

pickle
Returns a Pyhon Pickle representation of the object (if possible)

classmethod read(**args*, ***kwargs*)
Read a register from the database. The give key (usually ID) should be a primary key.

Parameters **id** (*int*) – the ID to get from the database

store(*database=None*)
Return a valid Storm store for this model

update(**args*, ***kwargs*)
Update a register in the database

uri
Returns the database URI for this model

Specific SQL Backends and Adaptors (used internally by Mamba)

It's probable that you never use those ones by yourself

class mamba.enterprise.sqlite.**SQLite**(*model*)
This class implements the SQLite syntax layer for mamba

Parameters **module** (*Model*) – the module to generate SQLite syntax for

create_table()
Return the SQLite syntax for create a table with this model

detect_primary_key()
Detect the primary key for the model and return it back with the correct SQLite syntax

Returns a string with the correct SQLite syntax

Return type str

Raises SQLiteMissingPrimaryKey on missing primary key

detect_uniques()
Checks if the model has an __mamba_unique__ property. If so, we create a compound unique with the fields specified inside __mamba_unique__. This variable must be a tuple of tuples.

Example: (('field1', 'field2'), ('field3', 'field4', 'field5')

)

drop_table()
Return SQLite syntax for drop this model table

get_compound_indexes()
Checks if the model has an __mamba_index__ property. If so, we create a compound index with the fields specified inside __mamba_index__. This variable must be a tuple of tuples.

Example: (('field1', 'field2'), ('field3', 'field4', 'field5')

)

get_primary_key_columns()
Return one or more primary key column(s)

get_primary_key_names()
Return one or more primary key name(s)

insert_data(*scheme*)
Return the SQL syntax needed to insert the data already present in the table.

is_compound_key(*name*)
Detects if given name is part of a compound primary key

Returns *bool*

parse(*column*)
This funciton is just a fallback to text (tears are comming)

parse_column(*column*)
Parse a Storm column to the correct SQLite value type. For example, if we pass a column of type `storm.variable.IntVariable` with name *amount* we get back:

amount integer

Parameters **column** (`storm.properties`) – the Storm properties column to parse

parse_references()
Get all the `storm.references.Reference` and create foreign keys for the SQL creation script if the SQLite version is equal or better than 3.6.19

If we are using references we should define our classes in a correct way. If we have a model that have a relation of many to one, we should define a many-to-one Storm relationship in that object but we must

create a one-to-many relation in the related model. That means if for example we have a *Customer* model and an *Adress* model and we need to relate them as one Customer may have several addresses (in a real application address may have a relation many-to-many with customer) we should define a relation with *Reference* from Address to Customer using a property like *Address.customer_id* and a *ReferenceSet* from *Customer* to *Address* like:

Customer.addresses = ReferenceSet(Customer.id, Address.id)

In the case of many-to-many relationships, mamba create the relation tables by itself so you dont need to take care of yourself.

static register()
Register this component

class mamba.enterprise.mysql.**MySQL**(*model*)
This class implements the MySQL syntax layer for mamba

Parameters **module** (*Model*) – the module to generate MySQL syntax for

create_table()
Return the MySQL syntax for create a table with this model

detect_indexes()
Go through all the fields defined in the model and create a index constraint if the index property is set on the field.

detect_primary_key()
Detect the primary key for the model and return it back with the correct MySQL syntax, Example:

PRIMARY KEY(*id*)

Returns a string with the correct MySQL syntax

Return type str

Raises MySQLMissingPrimaryKey on missing primary key

detect_uniques()
Go through all the fields defined in the model and create a unique key if the unique property is set on the field.

drop_table()
Return MySQL syntax for drop this model table

engine
Return back the type of engine defined for this MySQL table, if no engnine has been configured use InnoDB as default

get_compound_indexes()
Checks if the model has an __mamba_index__ property. If so, we create a compound index with the fields specified inside __mamba_index__. This variable must be a tuple of tuples.

Example: **(** ('field1', 'field2'), ('field3', 'field4', 'field5')

)

get_compound_uniques()
Checks if the model has an __mamba_unique__ property. If so, we create a compound unique with the fields specified inside __mamba_unique__. This variable must be a tuple of tuples.

Example: **(** ('field1', 'field2'), ('field3', 'field4', 'field5')

)

get_primary_key_columns()
Return one or more primary key column(s)

get_primary_key_names()
Return one or more primary key name(s)

get_single_indexes()
Goes through every field looking for an index parameter.

get_single_uniques()
Goes through every field looking for an unique parameter.

insert_data(*scheme*)
Return the SQL syntax needed to insert the data already present in the table.

is_compound_key(*name*)
Detects if given name is part of a compound primary key

Returns *bool*

parse(*column*)
This function is just a fallback to text (tears are comming)

parse_column(*column*)
Parse a Storm column to the correct MySQL value type. For example, if we pass a column of type `SmallIntVariable` with name *amount* we get back:

amount smallint

Parameters **column** (`storm.properties`) – the Storm properties column to parse

parse_decimal(*column*)
Parse decimal sizes for MySQL, for example:

decimal(10,2)

Parameters **column** (`storm.properties.Decimal`) – the Storm properties column to parse

parse_enum(*column*)
Parse an enum column

Parameters **column** (`storm.properties`) – the Storm properties column to parse

parse_int(*column*)
Parse an specific integer type for MySQL, for example:

smallint UNSIGNED

Parameters **column** (`storm.properties.Int`) – the Storm properties column to parse

parse_references()
Get all the `storm.references.Reference` and create foreign keys for the SQL creation script

If we are using references we should define our classes in a correct way. If we have a model that have a relation of many to one, we should define a many-to-one Storm relationship in that object but we must create a one-to-many relation in the related model. That means if for example we have a *Customer* model and an *Adress* model and we need to relate them as one Customer may have several addresses (in a real application address may have a relation many-to-many with customer) we should define a relation with *Reference* from Address to Customer using a property like *Address.customer_id* and a *ReferenceSet* from *Customer* to *Address* like:

Customer.addresses = ReferenceSet(Customer.id, Address.id)

In the case of many-to-many relationships, mamba create the relation tables by itself so you dont need to take care of yourself.

static register()
Register this component

class mamba.enterprise.postgres.**PostgreSQL**(*model*)
This class implements the PostgreSQL syntax layer for mamba

Parameters **module** (*Model*) – the module to generate PostgreSQL syntax for

create_table()
Return the PostgreSQL syntax for create a table with this model

detect_primary_key()
Detect the primary key for the model and return it back with the correct PostgreSQL syntax, Example:

PRIMARY KEY('id')

Returns a string with the correct PostgreSQL syntax

Return type str

Raises PostgreSQLMissingPrimaryKey on missing primary key

detect_uniques()
Checks if the model has an __mamba_unique__ property. If so, we create a compound unique with the fields specified inside __mamba_unique__. This variable must be a tuple of tuples.

Example: (('field1', 'field2'), ('field3', 'field4', 'field5')

)

drop_table()
Return PostgreSQL syntax for drop this model table

get_compound_indexes()
Checks if the model has an __mamba_index__ property. If so, we create a compound index with the fields specified inside __mamba_index__. This variable must be a tuple of tuples.

Example: (('field1', 'field2'), ('field3', 'field4', 'field5')

)

get_primary_key_columns()
Return one or more primary key column(s)

get_primary_key_names()
Return one or more primary key name(s)

insert_data(*scheme*)
Return the SQL syntax needed to insert the data already present in the table.

is_compound_key(*name*)
Detects if given name is part of a compound primary key

Returns *bool*

parse(*column*)
This function is just a fallback to text (tears are comming)

parse_column(*column*)
Parse a Storm column to the correct PostgreSQL value type. For example, if we pass a column of type `storm.variable.IntVariable` with name *amount* we get back:

'amount' integer

Parameters **column** (`storm.properties`) – the Storm properties column to parse

parse_enum(*column*)
Parses an enumeration column type. In PostgreSQL enumerations are created using `CREATE TYPE <name> AS ENUM (<values>);` format so we need to parse it separeted from regular column parsing.

We have to add the table name to the enum name as we can't define more than one enum with the same name.

Parameters **column** (`storm.properties`) – the Storm properties column to parse

parse_references()
Get all the `storm.references.Reference` and create foreign keys for the SQL creation script

If we are using references we should define our classes in a correct way. If we have a model that have a relation of many to one, we should define a many-to-one Storm relationship in that object but we must create a one-to-many relation in the related model. That means if for example we have a *Customer* model and an *Adress* model and we need to relate them as one Customer may have several addresses (in a real application address may have a relation many-to-many with customer) we should define a relation with *Reference* from Address to Customer using a property like *Address.customer_id* and a *ReferenceSet* from *Customer* to *Address* like:

Customer.addresses = ReferenceSet(Customer.id, Address.id)

In the case of many-to-many relationships, mamba create the relation tables by itself so you dont need to take care of yourself.

static register()
Register this component

class `mamba.enterprise.database.`**AdapterFactory**(*scheme*, *model*)
This is a factory which produces SQL Adapters.

Parameters

- **scheme** (*str*) – the database scheme (one of PostgreSQL, MySQL, SQLite)
- **model** (*Model*) – the model to use with this adapter

1.6.8 Extension Point

class `mamba.`**ExtensionPoint**(*mcs*, *name*, *bases*, *attrs*)
Extensions mount point

This class serves 3 purposes:

1. A way to declare a mount point for plugins or extensions point.
2. A way to register a plugin in a particular mount point.
3. A way to retrieve the plugins that have been registered.

The system works by declaring classes that serves as mount points. Since I subclass `type` I can be used as a metaclass, for example:

```
class ShareProvider:
    '''
    Mount point for plugins which refer to share services that can
    be used.

    Plugins implementing this reference should provide the
    following attributes:

    ========== ======================================================
    name        Share service name
    url         The URL to connect with the service
    username    The username which connect to the service
    password    The user password
    API_key     The API Key to use with the service
    ========== ======================================================

    '''
    __metaclass__ = ExtensionPoint
```

Then we can subclass those mount points in order to define plugins. As the plugin will also inherits from the metaclass, they will be auto registered just for subclassing the provider:

```
class Twitter(ShareProvider):
    name = 'Twitter'
    url = 'http://api.twitter.com/'
    ...
```

We can register any Share provider plugin just subclassing from the ShareProvider class and then use it to share our contents:

```
for share_service in ShareProvider.plugins:
    share_service().share(SharedObject)
```

CHAPTER 2

Getting Involved:

2.1 Contributing to Mamba

There are many ways that you can contribute and get involved in the Mamba project. Even if you are not a programmer, there are ways that you can help to make Mamba more useful for yourself and others.

The following is a list of possible ways to contribute and get involved with the mamba project.

2.1.1 User Feedback

In order to improve Mamba, we need feedback from both users and developers with bug reports, ideas and suggestions. For that we recommend to filling a new issue in the GitHub project issues tracker for the project.

You can also send an email to any of the *mamba mailing lists <http://www.pymamba.com/contact*.

2.1.2 Documentation contributions

If you want to contribute with some documentation for the project, thank you very much, we really appreciate it. Mamba uses the Sphinx documentation system so you should be familiarized with it if you want to contribute Mamba with some documentation.

2.1.3 Code contributions

Of course code contributions are more than welcome, please, take your time to review all the related documentation with the process.

Useful documentation for contributors

- Mamba guide to code contributions is a basic overview about how to contribute code to the Mamba project

2.1.4 Others way to contribute with mamba

As you may have already noticed, none of the Mamba developers are native english speakers so corrections to the full documentation will be really welcomed.

2.2 Mamba guide to code contributions

This is just a basic guide to Mamba development. If you came here looking for a guide about how to use Mamba to develop web applications, please, refer to the Mamba development guide.

2.2.1 Just a few considerations

We assume you have already a GitHub account and you know how to use the git version control system. If you aren't familiar with git, please refer to the git documentation where you can find all the documentation that you need to start working with git.

Mamba follows the workflow described in this Git Workflow Document wrote by Benjamin Sandofsky.

You should be familiarized already with Mamba concepts. In particular you should read and understand Mamba's 12 steps workflow, Howto submit code and Mamba coding style guide.

2.2.2 Contributors documentation

- Mamba's 12 steps workflow
- Commits guidelines
- Howto submit code
- Mamba coding style guide
- Unit testing

2.3 Mamba's 12 steps workflow

This is the general process for commiting code into the master branch of Mamba project. There are exceptions to this general workflow for example for security or critical bug fixes, minor changes to documentation and typo fixes or fix a broken build.

All the development workflow is performed in the Mamba's GitHub project site.

1. **When you are going to start to work in a fix or a feature you have to make sure of**
   (a) Make sure there is already an open issue about the code you are going to submit. If not, create it yourself.
   (b) The issue has been discussed and approved by the mamba's development team.
   (c) If the fix or feature affects the mamba core, make sure a mamba core developer is working (or is going to) with you in the integration of the code into mamba.
2. You should perform your work in your own fork.
3. You have to do it into a new branch.
4. Make sure your fork is synced with the last in-development version and your branch is rebased if needed.
5. Make sure you add an entry about the new fix/feature that you introduced on Mamba into the relnotes index file.

6. Make sure you add whatever documentation that is needed following the documentation for the project, thank you very much, we really appreciate it. Mamba uses the Sphinx documentation syntax.
7. Make sure you reference the issue or issues that your changes affects in your commit message so we can just track the code from the issues panel.
8. When your work is complete you should send a Pull Request on GitHub
9. Your pull request will be then reviewed by Mamba developers and other contributors.
10. If there are issues with your code in the review, you should fix them and pushing your changes for new review (make sure you add a new comment in the pull request issue thread as GitHub doesn't alert anyone about new commits added into a pull request).
11. When the review finishes, a core developer will merge your contribution into the master branch.
12. Goto step 1

2.4 Commits guidelines

Before to send your pull request you have to follow the following general rules.

1. Generally you are not going to do a single big commit with all your changes while you are working in your code. Normally you are going to have safe points commits, merges/rebases and other information that is useful for your development process but doesn't add any useful information to the project if you send your pull request all those commits on it. First you have to squash the commits into a single commit with a really good commit message. The general rule to follow is that every commit should be able to be used as an isolated cherry pick.
2. You message has to be descriptive and useful. First line should be a descriptive sentence about what the commit is doing. It has to be clear and fully understable what the full commit is going to do just reading that single line.
3. You have to include the issue number prefixed with # at the end of the first line
4. The next line should be a blank line followed by an enumerated list with the commit details.
5. Add the right keywords for the commit as described in Closing issues via commit messages in the GitHub documentation.

Finally you have to get a commit message like this one:

```
Adds <feature description> fo mamba web. Resolves #1936 and Fixes #456

    * Addition 1
    * Fix 1
    * Change 1
    * Deprecates 1
    * Deprecates 2
    ...
```

For a detailed information refer to the Git Workflow guidelines that mamba project follow since version 0.3.6

2.5 Howto submit code

Code submition in Mamba is done through GitHub pull requests system so you should be familiarized with it before trying to contribute your code with Mamba.

2.5.1 Contributing with a bug fix

You found a bug and you want to fix it and share the fix with the rest of the community?. First of all, thank you very much, contributions like that make Mamba better and maintains it alive.

Before start working in your fix, you have to go to the mamba issues and look for the bug that you've found because maybe some other person is working on fix it already.

If you don't find any open issue related to the bug that you found, make sure that the bug is not already fixed in a more up-to-date version of mamba. You can do that just checking the code in the last commit of the GitHub repo or just looking for a closed issue related with the bug.

Another useful place were to find information related with the bug is the relnotes index where the developers add information about bug fixes and other changes for the incomming new release version.

Of course your bugfix **must** follow the Mamba coding style guide and Unit testing guidelines.

My bug is listed as a closed issue

That means that the bug that you found is already fixed by another developer and is added already to a more up-to-date version of Mamba or is fixed in the in-development version and it's waiting for the release of the new version of the framework.

What I can do then?

You can pull the in-development version of the framework (that is always the last commit in the master branch) and test if your bug is gone. If is not, just re-open the issue and explain what the problem is and why the fix doesn't work. You may also want to work with the person that wrote the first fix.

My bug is listed as an open issue

Then this is a known issue, just read the issue discussion thread to discover if someone else is already working on a fix for that, if someone is already working to fix this problem, you can just contact him (or her) to work together in solving the issue.

If no other person is working on fixing the issue, just write a new comment to the issue discussion thread informing that you are going to work on solve the issue actively and ask other developers about guidance or ideas.

My bug is not listed anywhere

Then you found an unknown Mamba issue. Create a new issue in the GitHub repo and tag it as **bug**, explain that you are starting to work on fix it and ask for any help or guidance if needed.

Depending on the importance of the bug that you found, a mamba core developer may like to assist you or even solve himself the issue as fast as possible with your help.

2.5.2 Contributing with a new feature

You added a *killer* feature to your own mamba fork and you decided to share it with the community? That's really great, thank you.

New features **may or may not** been always welcome, that depends of the nature of the new feature and the impact in the framework and how developers use it. If you are planning to develop and share a new feature for mamba, or you already developed it and what you want is just include it as part of the project, create a new issue in the GitHub project and tag it as **enhancement** or just send a **pull request** explaining about yor new feature and why it should be added into the framework.

Note: Remember that all the code that you submit to the project **must** include unit tests.

2.5.3 Licensing the code

All the code that get into the Mamba framework **must** be licensed under the **GPLv3** (or a later version on your choice) license.

2.6 Mamba coding style guide

Like other Python projects, Mamba try to follow the PEP-8 and Sphinx as docstrings conventions. Make sure to read those documents if you intent to contrbute to Mamba.

2.6.1 Naming conventions

- Package and module names **must** be all lower-case, words may be separated with underscores. No exceptions.
- Class names are *CamelCase* e.g. *ControllerManager*
- The function, variables and class member names **must** be all lower-case, with words separated by underscores. No exceptions.
- Internal (private) methods and members are prefixed with a single underscore e.g. *_templates_search_path*

2.6.2 Style rules

- **Lines shouldn't exceed 79 characters length.**
- Tabs **must** be replaced by four blank spaces, **never merge** tabs and blank spaces.
- **Never** use multiple statements in the same line, e.g. `if check is True:  a = 0` with the only one exception for conditional expressions
- Comprehensions are preferred to the built-in functions `filter()` and `map()` when appropiate.
- Only use a `global` declaration in a function if that function actually modifies a global variable.
- Use the `format()` function to format strings instead of the semi-deprecated old '%s' % (data,) way.
- **Never** use a print statement, we always use a `print()` function instead.

- You **never** use `print` for logging purposes, for that you use `twisted.python.log` methods.

2.6.3 Inconsistences

Mamba uses Twisted as its main component and Twisted doesn't follow PEP-8 coding style and is never going to follow it. Because of that, be aware of some inconsistences where we use twisted objects and methods.

We want to be clear on that: we **never** use Twisted coding name conventions in exclusive Mamba code, **no exceptions** to this rule.

2.6.4 Principles

1. Never violates DRY.
2. Any code change **must** pass unit tests and shouldn't break any other test in the system.
3. No commit should break the master build.
4. No change should break user code silently, deprecations and incompatibilities must be always a known (and a well documented) matter.

2.7 Unit testing

Each unit test will test a single functionallity of the system. Unit tests are automated and should run with no human intervention. The output of an unit test should be **pass** or **fail**. All the tests on mamba are gathered in a single test suite and can run as a single batch.

2.7.1 How to run tests

Mamba uses Twisted's Trial tool to run unit tests. To run the test suite you have to cd into the Mamba root directory and run the trial tool:

```
$ trial mamba
```

You can also execute the *tests* bash script that runs *trial mamba* in the background:

```
$ ./tests
```

2.7.2 Adding new tests

You never add a new module to Mamba without adding a batch of unit tests for the module as well. You never commit any code **until you make sure** that all the tests pass, so that means all the tests should be green when you run *trial mamba* in the root of the project.

If you commit code that breaks the unit test suite you can be *hunted down* by other developers so make sure your code follow the guidelines and pass the tests.

You have to add unit tests for your module and code because in this way other developers can check if their own changes to other parts of the code affects in any way your modules just running the test suite.

Tests go into dedicated test packages, normally *mamba/test/* for framework tests and *mamba/scripts/test* for the *mamba-admin* command line tool. Tests should be named as *test_module.py*. If the module is part of a core system like *web* you can add the tests directly into the *mamba/test/test_web.py* file.

Mamba tests cases should inherit from *twisted.trial.unittest.TestCase* instead of the standard library *unittest.TestCase*.

2.7.3 Twisted test implementation guidelines

The next section is exact to twisted unit testing documentation, Mamba is based on twisted itself so the same guidelines are applied to the implementation of unit tests. Those guidelines has been copied (and modified for convenience in some cases) as is from the current twisted version documentation.

Real I/O

Most unit tests should avoid performing real, platform-implemented I/O operations. Real I/O is slow, unreliable, and unwieldy. When implementing a protocol, twisted.test.proto_helpers.StringTransport can be used instead of a real TCP transport. StringTransport is fast, deterministic, and can easily be used to exercise all possible network behaviors.

Note: Mamba is a web applications framework but can be used to create any type of applications that you can create using twisted itself, it's very common in mamba to implement custom protocols and implement mashup solutions that integrate servers and clients using different protocols.

Real Time

Most unit tests should also avoid waiting for real time to pass. Unit tests which construct and advance a twisted.internet.task.Clock are fast and deterministic.

Skipping tests

Trial implements some custom features in top of the stadard *unittest* library that are designed to encougare developers to add new tests. One common situation is that a test exercises some optional functionality: maybe it depends upon certain external libraries being available, maybe it only works on certain operating systems. The important common factor is that nobody considers these limitations to be a bug.

To make it easy to test as much as possible, some tests may be skipped in certain situations. Individual test cases can raise the SkipTest exception to indicate that they should be skipped, and the remainder of the test is not run. In the summary (the very last thing printed, at the bottom of the test output) the test is counted as a "*skip*" instead of a "*success*" or "*fail*". This should be used inside a conditional which looks for the necessary prerequisites:

```
# code extracted from 'mamba/test/test_controller.py'
class ControllerManagerTest(unittest.TestCase):
    ...
    def test_inotifier_provided_by_controller_manager(self):
        if not GNU_LINUX:
            raise unittest.SkipTest('File monitoring only available on Linux')
        self.assertTrue(INotifier.providedBy(self.mgr))
```

You can also set the *.skip* attribute on the method, with a string to indicate why the test is being skipped. This is convenient for temporarily turning off a test case, but it can also be set conditionally (by manipulating the class attributes after they've been defined):

```
class SomeThingTests(unittest.TestCase):
    def test_thing(self):
        dotest()
    test_thing.skip = "disabled locally"
```

```
class MyTestCase(unittest.TestCase):
    def test_one(self):
        ...
    def test_thing(self):
        dotest()

if not haveThing:
    MyTestCase.test_thing.im_func.skip = "cannot test without Thing"
    # but test_one() will still run
```

Finally, you can turn off an entire TestCase at once by setting the .skip attribute on the class. If you organize your tests by the functionality they depend upon, this is a convenient way to disable just the tests which cannot be run.

```
class TCPTestCase(unittest.TestCase):
    ...
class SSLTestCase(unittest.TestCase):
    if not haveSSL:
        skip = "cannot test without SSL support"
    # but TCPTestCase will still run
```

Associating test cases with source files

Please add a test-case-name tag to the source file that is covered by your new test. This is a comment at the beginning of the file which looks like one of the following:

```
# -*- test-case-name: mamba.test.test_web -*-
```

or

```
#!/usr/bin/env python
# -*- test-case-name: mamba.test.test_web -*-
```

This format is understood by emacs to mark "*File Variables*". The intention is to accept *test-case-name* anywhere emacs would on the first or second line of the file (but not in the File Variables: block that emacs accepts at the end of the file). If you need to define other emacs file variables, you can either put them in the File Variables: block or use a semicolon-separated list of variable definitions:

```
# -*- test-case-name: mamba.test.test_web; fill-column: 75; -*-
```

If the code is exercised by multiple test cases, those may be marked by using a comma-separated list of tests, as follows: (NOTE: not all tools can handle this yet.. *trial –testmodule* does, though)

```
# -*- test-case-name: mamba.test.test_web,mamba.test.test_resource -*-
```

The test-case-name tag will allow *trial –testmodule twisted/dir/myfile.py* to determine which test cases need to be run to exercise the code in *myfile.py*. Several tools (as well as twisted-dev.el's F9 command) use this to automatically run the right tests.

Sublime Text

A plugin is being developed for integrate this on Sublime Text versions 2 and 3

2.7.4 Must-read links

Tips for writing tests for Twisted code

2.8 The Mamba Team

The developers currently working on mamba are:

- Oscar Campos
- Nicholas Amorim

2.9 Oscar Campos

- Email: <oscar dot campos at member dot fsf dor org>
- Twitter: @damnwidget

2.9.1 Role

Project Lead

2.9.2 Development Areas

- Mamba Core (Enterprise, Web, Deployment ...)
- Command Line Interface tool (mamba-admin)
- Storm Integration
- BlackMamba (mamba main website)
- Mamba Git repository maintenance and integration
- Mamba devel tools

2.9.3 System Administration Areas

- Full mamba infraestructure (Databases, Servers, Network, etc)

2.9.4 Additional Information

Oscar was born in Spain but as many others spanish engineers he left his country to go abroad and currently is living and working in Dublin, Ireland.

Oscar works for Dedsert, an online gambling startup company

2.10 Release Notes for Mamba v0.3.4

Those are the release notes for Mamba v0.3.4 released on May 05, 2013.

2.10.1 Features

This is the first Mamba release.

Routing System

Mamba implements a custom routing system that don't uses the traditional Twisted routing based on childs.

For more information refer to the routing system documentation

Database ORM Layer

Mamba implements an ORM Database access layer in top of Storm and it's fully integrated with Twisted Deferred system.

The supported database backends are:

- Postgres
- MySQL/MariaDB
- SQLite

Mamba adds several features to Storm making possible for example generate database schemes using programatic configuration Python code.

For more information refer to the Mamba Storm integration documentation

Jinja2 Templating System

Mamba integrates the Jinja2 Templating System in the routing and rendering process but you are allowed to use whatever other templating system you want (including Twisted Templating System of course).

For detailed information about Mamba Jinja2 templating integration refer to its documentation.

Mamba Administration Command Line Interface Tools

Mamba include a complete CLI toolset to manage and configure Mamba web applications, you can use the *mamba-admin* script in your terminal to access the complete toolset.

To get more information about *mamba-admin* refer to it's documentation.

2.10.2 Bug Fixes

2.10.3 Deprecations

2.10.4 Removals

2.10.5 Uncompatible Changes

2.10.6 Details

If you need a more detailed description of the changes made in this release you can use git itself using:

```
git log 0.3.3..0.3.4
```

2.11 Release Notes for Mamba 0.3.5

Those are the release notes for Mamba 0.3.5 released on May 06, 2013.

2.11.1 Features

- Added 302 (Found) HTTP response to prebuilt responses.

2.11.2 Bug Fixes

- Fix for ticket #4. Now mamba-admin view create the Controller templates directory if it doesn't exists yet.
- Solved problem with Storm stores in Model object related with the switch from ZStorm to Twisted transactions.

2.11.3 Deprecations

2.11.4 Removals

2.11.5 Uncompatible Changes

2.11.6 Details

If you need a more detailed description of the changes made in this release you can use git itself using:

```
git log 0.3.4..0.3.5
```

2.12 Release Notes for Mamba ${version}

Those are the release notes for Mamba ${version} released on ${release_date}.

2.12.1 Features

- Added Created (201 HTTP) Response to predefined responses
- Added Unknown (209 HTTP) Response to predefined responses (209 is unassigned)
- Added Unauthorized (401 HTTP) Response to predefined responses
- Added MovedPermanently (301 HTTP) Response to predefined responses
- Added SeeOther (303 HTTP) Response to predefined responses
- Added Forbidden (403 HTTP) Response to predefined responses
- Added decimal size and precission using size property for MySQL decimal fields definitions:

```
some_field = Decimal(size=(10, 2))   # using a tuple
some_field = Decimal(size=[10, 2])   # using a list
some_field = Decimal(size=10.2)      # using a float
some_field = Decimal(size='10,2')    # using a string
some_field = Decimal(size=10)        # using an int (precission is set to 2)
```

- If user define *development* as *true* in the *application.json* config file, the twistd server will be started in **no** daemon mode, logging will be printed to standard output and no log file should be produced at all. If the user press Ctrl+C the mamba application will terminate immediately, this mode is quite useful to development stages
- We redirect all the regular twistd process logging messages to *syslog* if we don't define development mode as *true* in the application.json config file
- Added *package* subcommand to *mamba-admin* command line tool that allow us to pack, install and uninstall mamba based applications in order to reuse them (that doesn't replace setuptools and is not the way meant to package a full mamba application in order to distribute it, this is only meant to reuse code already present in other projects)
- Added shared controllers, models and views for mamba installed packages. We can now add a configuration file *installed_packages.json* with the following syntax so mamba can install and reuse scrips (JavaScript, CSS) controllers, models and views in automatic way (nota that views search paths are always added to Jinja2 templating search path even if *autoimport* is *false*):

```
{
    "packages": {
        "package_name": {
            "autoimport": true,
            "use_scripts": true
        }
    }
}
```

- Added *restart* command to *mamba-admin* command line tool
- Added *checkers* package to *mamba.utils* for common checks on forms and applications
- Added auto LESS resources compiling for mambaerized styleseets
- Added *find* method that can be used with or without a class instance:

customer = yield Customer.find(name=u'John') customer = yield Customer().find(name=u'John') customer = yield Customer.find(Customer.name == u'john')

- Added *all* method that can be used with or without a class instance and return all the rows from a given model:

  customers = yield Customer.all() customers = yield Customer().all()

- **Added *async* magic named boolean argument to *Transactor.run* so we can now run a @*transact* decorated method synchro**
  customer = Customer.find(name=u'Pam', async=False) customer = Customer().read(1, async=False)

- Added global class *__mamba_async__* property to *mamba.application.model* so we can just make all the methods from a given class synchronous by default

- Added *auto_commit* msgic boolean argument to *Transactor.run* so we can now run a @*transact* decorated method that is not commit in automatic way by the transact mechanism (autocommit is True by default).

- Added several unit tests

- Added support for FOREIGN KEYS in SQLite version >= 3.6.19

- Added UNIQUE and INDEX for single and compound SQLite, MySQL/MariaDB and PostgreSQL fields

- Added support to don't add some tables to the generated schema using `__mamba_schema__ = False` option

- Added `--noschema` option in `mamba-admin sql` command line options

- Added shell to `mamba-admin sql` command line tool

- Now controllers can be attached to other controllers controllers to form a path route tree

- Added TestableDatabase and prepare_model_for_test function to make easier the task of test mamba applications models

- Added fixtures class (extends Storm's Schema)

- Added modules and controllers sub pacages automatic pre-load (for ex: application/model/sub_package/model.py)

- Added dict, json and pickle method to serialize Models

2.12.2 Bug Fixes

- We were not using properly ZStorm/transaction and Twisted Transactor integration in Storm, now is fixed and a new *copy* helper method has been added to `mamba.application.model.Model` class to allow simple object copy on user code because we can't use a store that has been created in a thread into the `twisted.python.threadpool.ThreadPool` using the @*transact* decorator. If you need to pass initialized Storm objects directly to a view for whatever reasson you shouldn't use the @*transact* decorator at all (so you shouldn't use asynchronous call to the database for that).

- Now unhandled errors in Deferreds on routing module are displayed nicely in the logs file

- Model read method now returns a copy of the Storm object that can be used in other threads if the optional parameter copy is True (it's False by default)

- Fixed a bug in create SQL mamba-admin command when used with live (-l) option

- Fixed a bug related with PyPy and it's lack of **set_debug** method in **gc** object

- Now mamba-admin start and stop subcommands can be used inside valid mamba application directories only

- Adding dependency to fabric package as docs will not build without it

- Added mandatory option parameter *development* to the application.json template.

- Fixed memory leak in the routing system cache
- Fixed bug that hides log_file being null in options
- Fixed bug in package pack when using alternative names
- Fixed bug in package pack when version string has more than two levels
- Fixed bug related with routed methods that does not return anything
- Now mamba does not print a bogus and unrelated error message when there is some problem with the JSON config files
- Fixes paths in scrips and stylesheets that were preventing those ones to be added into the HTML generated by the templating engine
- `Converter` wasn't serializing properties that were other objects properly, now is fixed
- decimal.Decimal values are now corretly serialized on `Converter`
- Fixed some model tests that weren't working
- When *mamba-admin sql configure* ran in a validmamba app directory that does not contains a *config* directory, it crashed, fixed
- Fixed bug in PostgreSQL schema generation for FOREIGN KEYS
- Fixed wrong response being displayed when installing mamba reusabiility package from file
- Fixed bug where updates made to an installed mamba package was not updated.
- Fixed bug where mamba packages in egg format were not being installed. Added two extra unit tests in test_mamaba_admin.py for installing from egg and tar.
- Fixed exception being raised when POST, PUT and PATCH requests were send with no body

2.12.3 Changes

- Now we can add a custom Jinja2 templates loader to our controller templates in two different ways:
  - **Method One**: Just pass the named param *loader=<your customer loader class>* to the *Template.render* call and it will overwrite any previous loader configuration
  - **Method Two**: When you first instanciate your template object (commonly with *self.template = templating.Template()*) add just your custom loader class as a property of the new template instance:

```
self.template.loader = CustomLoader
```

    Note that is a class and not an instance what you have to use in both methods. The class **must** expect a list of strings (paths) as first and unique argument.

- The mamba-admin application subcommand generates now a `logs` directory and logs files are created inside it
- The mamba-admin application subcommand generates now a `lib` directory into the `application` directory in oreder to place code that doesn't fit the MVC pattern and 3rd party libraries
- The `@route` decorator now accepts lists and tuples defining more than one HTTP method where to register the given action
- The `NativeEnum` type has been reimplemented as a `set`. Implementation provided by Patrick O'Loughlin @paddyoloughlin on GitHub
- Added new find method to model object to find ojects into the database

- Storm.locals imports moved to `mamba.entreprise` package
- Now is possible to create subpakages for modules and controllers using 'subpackage.module_name' as the name of the controller or model, for ex: mamba-admin controller community.users
- If we return a Model object from a controller method, the routing system try to convert it into JSON instead of silently fail

2.12.4 Documentation

- Added contributors documentation
- Added developers documentation

2.12.5 Deprecations

None

2.12.6 Removals

- Removed unused cleanups in controller tests

2.12.7 Uncompatible Changes

None

2.12.8 Details

If you need a more detailed description of the changes made in this release you can use git itself using:

```
git log ${current_version}..${version}
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

m

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W