

---

# **Milla Documentation**

***Release 0.1.2***

**Dustin C. Hatch**

October 16, 2012



# CONTENTS



Milla is an extremely simple WSGI framework for web applications

Contents:



# RATIONALE

As of early 2011, there is a lot of flux in the Python world with regard to web frameworks. There are a couple of big players, namely [Django](#), [Pylons](#), and [TurboGears](#), as well as several more obscure projects like [CherryPy](#) and [Bottle](#). Having worked with many of these projects, I decided that although each has its strengths, they all also had something about them that just made me feel uncomfortable working with them.

## 1.1 Framework Comparison

### 1.1.1 Django

#### Strengths

- Very popular and actively developed, making it easy to get help and solve problems
- Fully-featured, including an Object-Relational Mapper, URL dispatcher, template engine, and form library. Also includes “goodies” like authentication, an “admin” application, and sessions

#### Discomforts

I am not specifically listing any of these issues as weaknesses or drawbacks, because they aren’t *per-se*. Honestly, there isn’t anything wrong with Django, and many people love it. Personally, I don’t feel comfortable working with it for a few reasons.

- Storing configuration in a Python module is absurd
- All of the components are so tightly-integrated it is nearly impossible to use some pieces without the others.
  - I really don’t like its ORM. [SQLAlchemy](#) is tremendously more powerful, and isn’t nearly as restrictive (naming conventions, etc.)
  - The session handling middleware is very limited in comparison to other projects like [Beaker](#)
  - I am not fond of the template engine and would prefer to use [Genshi](#).

### 1.1.2 Pylons/Pyramid

The original Pylons was a very powerful web framework. It was probably my favorite framework, and I have built a number of applications using it. Unfortunately, development has been discontinued and efforts are now being concentrated on [Pyramid](#) instead.

## Pylons

### Strengths

- While not as popular as Django, there still a significant following
- The code base is very decoupled, allowing developers to swap out components without affecting the overall functionality of the framework.

### Weaknesses

- Overutilization of StackedObject proxies and global variables

## Pyramid

I simply do not like Pyramid at all, and it is really disappointing that the Pylons project has moved in this direction. Essentially everything that I liked about Pylons is gone. The idea of using *traversal* to map URLs to routines is clever, but it is overly complex compared to the familiar URL dispatching in other frameworks.

- Tightly integrated with several Zope components, mostly interfaces (*puke*)
- Template renderers are insanely complex and again, I don't like Zope interfaces. There is no simple way to use Genshi, which I absolutely adore.

### 1.1.3 Other Frameworks

I haven't used the other frameworks as much. In general, I try to avoid having my applications depend on obscure or unmaintained libraries because when I find a bug (and I will), I need some assurance that it will be fixed soon. I do not like having to patch other people's code in production environments, especially if it is an application I am passing along to a client.

I never really looked at TurboGears at all, and with the recent changes to the Pylons project, upon which TurboGears is based, there is a great deal of uncertainty with regard to its future.

CherryPy is very nice, and I did a bit of work with it a while back. I thought it was dead for a long time, though, and I have never really produced a production application built on it. With its most recent release (3.2.0), it is the first web framework to support Python 3, which is exciting. I may revisit it in the near future, as a matter of fact.

## 1.2 The Truth

The truth is, I started *Milla* as an exercise to better understand WSGI. All of the frameworks discussed above are great, and will most likely serve everyone's needs. There really isn't any reason for anyone to use *Milla* over any of them, but I won't stop you.



# API REFERENCE

## 2.1 milla.auth

### 2.1.1 milla.auth.decorators

Convenient decorators for enforcing authorization on controllers

**Created** Mar 3, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

`milla.auth.decorators.auth_required` (*func*)

Simple decorator to enforce authentication for a controller

Example usage:

```
class SomeController(object):

    def __before__(request):
        request.user = find_a_user_somewhat(request)

    @milla.auth_required
    def __call__(request):
        return 'Hello, world!'
```

In this example, the `SomeController` controller class implements an `__before__` method that adds the `user` attribute to the `request` instance. This could be done by extracting user information from the HTTP session, for example. The `__call__` method is decorated with `auth_required`, which will ensure that the user is successfully authenticated. This is handled by a *request validator*.

If the request is not authorized, the decorated method will never be called. Instead, the response is generated by calling the `NotAuthorized` exception raised inside the `auth_required` decorator.

`class milla.auth.decorators.require_perms` (*\*requirements*)

Decorator that requires the user have certain permissions

Example usage:

```
class SomeController(object):

    def __before__(request):
        request.user = find_a_user_somewhat(request)
```

```
@milla.require_perms('some_permission', 'and_this_permission')
def __call__(request):
    return 'Hello, world!'
```

In this example, the `SomeController` controller class implements an `__before__` method that adds the user attribute to the `request` instance. This could be done by extracting user information from the HTTP session, for example. The `__call__` method is decorated with `require_perms`, which will ensure that the user is successfully authenticated and the the user has the specified permissions. This is handled by a *request validator*.

There are two ways to specify the required permissions:

- By passing the string name of all required permissions as positional arguments. A complex permission requirement will be constructed that requires *all* of the given permissions to be held by the user in order to validate
- By explicitly passing an instance of `Permission` or `PermissionRequirement`

## 2.1.2 milla.auth.permissions

Classes for calculating user permissions

Examples:

```
>>> req = Permission('foo') & Permission('bar')
>>> req.check(PermissionContainer(['foo', 'baz'], ['bar']))
True
```

```
>>> req = Permission('login')
>>> req.check(['login'])
True
```

```
>>> req = Permission('login') | Permission('admin')
>>> req.check(['none'])
False
```

**class** `milla.auth.permissions.BasePermission`

Base class for permissions and requirements

Complex permission requirements can be created using the bitwise `and` and `or` operators:

```
login_and_view = Permission('login') & Permission('view')
admin_or_root = Permission('admin') | Permission('root')

complex = Permission('login') & Permission('view') | Permission('admin')
```

**class** `milla.auth.permissions.Permission(name)`

Simple permission implementation

**Parameters** `name (str)` – Name of the permission

Permissions must implement a `check` method that accepts an iterable and returns `True` if the permission is present or `False` otherwise.

**check** (*perms*)

Check if the permission is held

This method can be overridden to provide more robust support, but this implementation is simple:

```
return self in perms
```

**class** `milla.auth.permissions.PermissionContainer` (*user\_perms*=[], *group\_perms*=[])  
Container object for user and group permissions

#### Parameters

- **user\_perms** (*list*) – List of permissions held by the user itself
- **group\_perms** (*list*) – List of permissions held by the groups to which the user belongs

Iterating over `PermissionContainer` objects results in a flattened representation of all permissions.

**class** `milla.auth.permissions.PermissionRequirement` (*\*requirements*)  
Base class for complex permission requirements

**class** `milla.auth.permissions.PermissionRequirementAll` (*\*requirements*)  
Complex permission requirement needing all given permissions

**class** `milla.auth.permissions.PermissionRequirementAny` (*\*requirements*)  
Complex permission requirement needing any given permissions

Request authorization

**Created** Apr 5, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

**exception** `milla.auth.NotAuthorized`  
Base class for unauthorized exceptions

This class is both an exception and a controller callable. If the request validator raises an instance of this class, it will be called and the resulting value will become the HTTP response. The default implementation simply returns HTTP status 403 and a simple body containing the exception message.

**class** `milla.auth.RequestValidator`  
Base class for request validators

A request validator is a class that exposes a `validate` method, which accepts an instance of `webob.Request` and an optional requirement. The `validate` method should return `None` on successful validation, or raise an instance of `NotAuthorized` on failure. The base implementation will raise an instance of the exception specified by `exc_class`, which defaults to `:py:class`NotAuthorized``.

To customize the response to unauthorized requests, it is sufficient to subclass `NotAuthorized`, override its `__call__()` method, and specify the class in `exc_class`.

#### **exc\_class**

Exception class to raise if the request is unauthorized

alias of `NotAuthorized`

**validate** (*request*, *requirement=None*)  
Validates a request

#### Parameters

- **request** – The request to validate. Should be an instance of `webob.Request`.
- **requirement** – (Optional) A requirement to check. Should be an instance of `Permission` or `PermissionRequirement`, or some other class with a `check` method that accepts a sequence of permissions.

The base implementation will perform authorization in the following way:

- 1.Does the `request` have a `user` attribute? If not, raise `NotAuthorized`.
- 2.Is the truth value of `request.user` true? If not, raise `NotAuthorized`.
- 3.Does the `request.user` object have a `permissions` attribute? If not, raise `NotAuthorized`.
- 4.Do the user's permissions meet the requirements? If not, raise `NotAuthorized`.

If none of the above steps raised an exception, the method will return `None`, indicating that the validation was successful.

---

**Note:** `WebOb` Request instances do not have a `user` attribute by default. You will need to supply this yourself, i.e. in a WSGI middleware or in the `__before__` method of your controller class.

---

## 2.2 milla.dispatch

### 2.2.1 milla.dispatch.routing

URL router

**Created** Mar 13, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

**class** `milla.dispatch.routing.Generator` (*request*, *path\_only=True*)  
URL generator

Creates URL references based on a *WebOb* request.

Typical usage:

```
>>> generator = Generator(request)
>>> generator.generate('foo', 'bar')
'/foo/bar'
```

A common pattern is to wrap this in a stub function:

```
url = Generator(request).generate
```

**generate** (*\*segments*, *\*\*vars*)

Combines segments and the application's URL into a new URL

**class** `milla.dispatch.routing.Router` (*trailing\_slash=<class 'milla.dispatch.routing.REDIRECT'>*)  
A dispatcher that maps arbitrary paths to controller callables

Typical usage:

```
router = Router()
router.add_route('/foo/{bar}/{baz:\d+}', some_func)
app = milla.Application(dispatcher=router)
```

In many cases, paths with trailing slashes need special handling. The `Router` has two ways of dealing with requests that should have a trailing slash but do not. The default is to send the client an HTTP 301 Moved Permanently response, and the other is to simply treat the request as if it had the necessary trailing slash. A

third option is to disable special handling entirely and simply return HTTP 404 Not Found for requests with missing trailing slashes. To change the behavior, pass a different value to the constructor's `trailing_slash` keyword.

Redirect the client to the proper path (the default):

```
router = Router(trailing_slash=Router.REDIRECT)
router.add_route('/my_collection/', some_func)
```

Pretend the request had a trailing slash, even if it didn't:

```
router = Router(trailing_slash=Router.SILENT)
router.add_route('/my_collection/', some_func)
```

Do nothing, let the client get a 404 error:

```
router = Router(trailing_slash=None)
router.add_route('/my_collection/', some_func)
```

**add\_route** (*template, controller, \*\*vars*)

Add a route to the routing table

#### Parameters

- **template** – Route template string
- **controller** – Controller callable or string Python path

Route template strings are path segments, beginning with `/`. Paths can also contain variable segments, delimited with curly braces.

Example:

```
/some/other/{variable}/{path}
```

By default, variable segments will match any character except a `/`. Alternate expressions can be passed by specifying them alongside the name, separated by a `:`.

Example:

```
/some/other/{alternate:[a-zA-Z]}
```

Variable path segments will be passed as keywords to the controller. In the first example above, assuming `controller` is the name of the callable passed, and the request path was `/some/other/great/place`:

```
controller(request, variable='great', path='place')
```

The `controller` argument itself can be any callable that accepts a *WebOb* request as its first argument, and any keywords that may be passed from variable segments. It can also be a string Python path to such a callable. For example:

```
`some.module:function`
```

This string will resolve to the function `function` in the module `some.module`.

**resolve** (*path\_info*)

Find a controller for a given path

**Parameters** `path_info` – Path for which to locate a controller

**Returns** A `functools.partial` instance that sets the values collected from variable segments as keyword arguments to the callable

This method walks through the routing table created with calls to `add_route()` and finds the first whose template matches the given path. Variable segments are added as keywords to the controller function.

**template\_re** = `<_sre.SRE_Pattern object at 0x3bcf0a0>`  
Compiled regular expression for variable segments

## 2.2.2 milla.dispatch.traversal

URL Dispatching

**Created** Mar 26, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

**class** `milla.dispatch.traversal.Traverser` (*root*)  
Default URL dispatcher

**Parameters** *root* – The root object at which lookup will begin

The default URL dispatcher uses object attribute traversal to locate a handler for a given path. For example, consider the following class:

```
class Root(object):  
  
    def foo(self):  
        return 'Hello, world!'
```

The path `/foo` would resolve to the `foo` method of the `Root` class.

If a path cannot be resolved, `UnresolvedPath` will be raised.

**resolve** (*path\_info*)  
Find a handler given a path

**Parameters** *path\_info* – Path for which to find a handler

**Returns** A handler callable

**exception** `milla.dispatch.UnresolvedPath`  
Raised when a path cannot be resolved into a handler

## 2.3 milla.app

Module `milla.app`

Please give me a docstring!

**Created** Mar 26, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

**class** `milla.app.Application(dispatcher)`

Represents a Milla web application

Constructing an `Application` instance needs a dispatcher, or alternatively, a root object that will be passed to a new `milla.dispatch.traversal.Traverser`.

#### Parameters

- **root** – A root object, passed to a traverser, which is automatically created if a root is given
- **dispatcher** – An object implementing the dispatcher protocol

`Application` instances are WSGI applications.

#### config

A mapping of configuration settings. For each request, the configuration is copied and assigned to `request.config`.

## 2.4 milla.controllers

Stub controller classes

These classes can be used as base classes for controllers. While any callable can technically be a controller, using a class that inherits from one or more of these classes can make things significantly easier.

**Created** Mar 27, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

**class** `milla.controllers.Controller`

The base controller class

This class simply provides empty `__before__` and `__after__` methods to facilitate cooperative multiple inheritance.

**class** `milla.controllers.FaviconController(icon=None, content_type='image/x-icon')`

A controller for the “favicon”

This controller is specifically suited to serve a site “favicon” or bookmark icon. By default, it will serve the *Milla* icon, but you can pass an alternate filename to the constructor.

#### Parameters

- **icon** – Path to an icon to serve
- **content\_type** – Internet media type describing the type of image used as the favicon, defaults to ‘image/x-icon’ (Windows ICO format)

## 2.5 milla.util

Module `milla.util`

Please give me a docstring!

**Created** Mar 30, 2011

**Author** dustin

**Updated** \$Date\$

**Updater** \$Author\$

`milla.util.asbool` (*val*)

Test a value for truth

Returns `False` values evaluating as false, such as the integer `0` or `None`, and for the following strings, irrespective of letter case:

- `false`
- `no`
- `f`
- `n`
- `off`
- `0`

Returns `True` for all other values.

*Milla* is released under the terms of the [Apache License, version 2.0](#).



# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*



# PYTHON MODULE INDEX

## m

- milla, ??
- milla.app, ??
- milla.auth, ??
- milla.auth.decorators, ??
- milla.auth.permissions, ??
- milla.controllers, ??
- milla.dispatch, ??
- milla.dispatch.routing, ??
- milla.dispatch.traversal, ??
- milla.util, ??