Milla Documentation

Release 0.2

Dustin C. Hatch

October 23, 2013

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

Milla aims to be lightweight and easy to use. As such, it provides only the tools you need to build your application the way you want, without imposing any restrictions on how to do it.

Contents

1.1 Milla's Components

Milla provides a small set of components that help you build your web application in a simple, efficient manner:

- WSGI Application wrapper
- Two types of URL Dispatchers:
  - Traversal (like CherryPy or Pyramid)
  - Routing (like Django or Pylons)
- Authorization framework
- Utility functions

Milla does not provide an HTTP server, so you'll have to use one of the many implementations already available, such as Meinheld or Paste, or another application that understands WSGI, like Apache HTTPD with the mod_wsgi module.

1.2 `Application` Objects

The core class in a *Milla*-based project is its `Application` object. `Application` objects are used to set up the environment for the application and handle incoming requests. `Application` instances are *WSGI* callables, meaning they implement the standard `application(environ, start_response)` signature.

To set up an `Application`, you will need a *URL dispatcher*, which is an object that maps request paths to *controller* callables.

1.3 Choosing a URL Dispatcher

Milla provides two types of URL dispatchers by default, but you can create your own if neither of these suit your needs. The default dispatchers are modeled after the URL dispatchers of other popular web frameworks, but may have small differences.

A *Milla* application can only have one URL dispatcher, so make sure you choose the one that will work for all of your application's needs.

1.3.1 Traversal

Object traversal is the simplest form of URL dispatcher, and is the default for *Milla* applications. Object traversal works by looking for path segments as object attributes, beginning with a *root object* until a *controller* is found.

For example, consider the URL `http://example.org/myapp/hello`. Assuming the *Milla* application is available at `/myapp` (which is controlled by the HTTP server), then the `/hello` portion becomes the request path. It contains only one segment, `hello`. Thus, an attribute called `hello` on the *root object* must be the controller that will produce a response to that request. The following code snippet will produce just such an object.

```
class Root(object):

    def hello(self, request):
        return 'Hello, world!'
```

To use this class as the *root object* for a *Milla* application, pass an instance of it to the `Application` constructor:

```
application = milla.Application(Root())
```

To create URL paths with multiple segments, such as `/hello/world` or `/umbrella/corp/bio`, the root object will need to have other objects corresponding to path segments as its attributes.

This example uses static methods and nested classes:

```
class Root(object):

    class hello(object):

        @staticmethod
        def world(request):
            return 'Hello, world!'

application = milla.Application(Root)
```

This example uses instance methods to create the hierarchy at runtime:

```
class Root(object):

    def __init__(self):
        self.umbrella = Umbrella()

class Umbrella(object):

    def __init__(self):
        self.corp = Corp()
```

```
class Corp(object):

    def bio(self, request):
        return 'T-Virus research facility'

application = milla.Application(Root())
```

If an attribute with the name of the next path segment cannot be found, *Milla* will look for a `default` attribute.

While the object traversal dispatch mechanism is simple, it is not very flexible. Because path segments correspond to Python object names, they must adhere to the same restrictions. This means they can only contain ASCII letters and numbers and the underscore (_) character. If you need more complex names, dynamic segments, or otherwise more control over the path mapping, you may need to use routing.

1.3.2 Routing

Routing offers more control of how URL paths are mapped to *controller* callables, but require more specific configuration.

To use routing, you need to instantiate a `Router` object and then populate its routing table with path-to-controller maps. This is done using the `add_route()` method.

```
def hello(request):
    return 'Hello, world!'

router = milla.dispatch.routing.Router()
router.add_route('/hello', hello)
```

Aft er you've set up a `Router` and populated its routing table, pass it to the `Application` constructor to use it in a *Milla* application:

```
application = milla.Application(router)
```

Using routing allows paths to contain dynamic portions which will be passed to controller callables as keyword arguments.

```
def hello(request, name):
    return 'Hello, {0}'.format(name)

router = milla.dispatch.routing.Router()
router.add_route('/hello/{name}', hello)

application = milla.Application(router)
```

In the above example, the path `/hello/alice` would map to the `hello` function, and would return the response `Hello, alice` when visited.

`Router` instances can have any number of routes in their routing table. To add more routes, simply call `add_route` for each path and controller combination you want to expose.

```
def hello(request):
    return 'Hello, world!'

def tvirus(request):
    return 'Beware of zombies'

router = milla.dispatch.routing.Router()
router.add_route('/hello', hello)
```

```
router.add_route('/hello-world', hello)
router.add_route('/umbrellacorp/tvirus', tvirus)
```

1.4 Controller Callables

Controller callables are where most of your application's logic will take place. Based on the MVC (Model, View, Controller) pattern, controllers handle the logic of interaction between the user interface (the *view*) and the data (the *model*). In the context of a *Milla*-based web application, controllers take input (the HTTP request, represented by a `Request` object) and deliver output (the HTTP response, represented by a `Response` object).

Once you've decided which URL dispatcher you will use, it's time to write controller callables. These can be any type of Python callable, including functions, instance methods, classmethods, or partials. *Milla* will automatically determine the callable type and call it appropriately for each controller callable mapped to a request path.

This example shows a controller callable as a function (using routing):

```
def index(request):
    return 'this is the index page'

def hello(request):
    return 'hello, world'

router = milla.dispatch.routing.Router()
router.add_route('/', index)
router.add_route('/hello', hello)
application = milla.Application(router)
```

This example is equivalent to the first, but shows a controller callable as a class instance (using traversal):

```
class Controller(object):

    def __call__(self, request):
        return 'this is the index page'

    def hello(self, request):
        return 'hello, world'

application = milla.Application(Controller())
```

Controller callables must take at least one argument, which will be an instance of `Request` representing the HTTP request that was made by the user. The `Request` instance wraps the *WSGI* environment and exposes all of the available information from the HTTP headers, including path, method name, query string variables, POST data, etc.

If you are using Routing and have routes with dynamic path segments, these segments will be passed by name as keyword arguments, so make sure your controller callables accept the same keywords.

1.4.1 Before and After Hooks

You can instruct *Milla* to perform additional operations before and after the controller callable is run. This could, for example, create a SQLAlchemy session before the controller is called and roll back any outstanding transactions after it completes.

To define the before and after hooks, create an `__before__` and/or an `__after__` attribute on your controller callable. These attributes should be methods that take exactly one argument: the request. For example:

```
def setup(request):
     request.user = 'Alice'

def teardown(request):
     del request.user

def controller(request):
    return 'Hello, {user}!'.format(user=request.user)
controller.__before__ = setup
controller.__after__ = teardown
```

To simplify this, *Milla* handles instance methods specially, by looking for the `__before__` and `__after__` methods on the controller callable's class as well as itself.

```
class Controller(object):

    def __before__(self, request):
        request.user = 'Alice'

    def __after__(self, request):
        del request.user

    def __call__(self, request):
        return 'Hello, {user}'.format(user=request.user)
```

1.5 Returing a Response

Up until now, the examples have shown *controller* callables returning a string. This is the simplest way to return a plain HTML response; *Milla* will automatically send the appropriate HTTP headers for you in this case. If, however, you need to send special headers, change the content type, or stream data instead of sending a single response, you will need to return a `Response` object. This object contains all the properties necessary to instruct *Milla* on what headers to send, etc. for your response.

To create a `Response` instance, use the `ResponseClass` attribute from the request:

```
def controller(request):
    response = request.ResponseClass()
    response.content_type = 'text/plain'
    response.text = 'Hello, world!'
    return response
```

CHAPTER

TWO

ADVANCED FEATURES

Milla contains several powerful tools that allow web developers complete control over how their applications behave.

Contents

2.1 Propagating Configuration

While one possible way for *controller* callables to obtain configuration information would be for them to read it each time a request is made, it would be extremely inefficient. To help with this, *Milla* provides a simple configuration dictionary that can be populated when the `Application` is created and will be available to controllers as the `config` attribute of the request.

```
def controller(request):
    if request.config['t_virus'] == 'escaped':
        return 'Zombies!'
    else:
        return 'Raccoon City is safe, for now'

router = milla.dispatch.routing.Router()
router.add_route('/', controller)
application = milla.Application(router)
application.config['t_virus'] = 'contained'
```

Milla provides a simple utility called `read_config()` that can produce a flat dictionary from a standard configuration file:

```
; umbrella.ini
[t_virus]
status = escaped

# app.py
class Root(object):
```

```
    def __call__(self, request):
        if request.config['t_virus.status'] == 'escaped':
            return 'Zombies!'
        else:
            return 'Raccoon City is safe, for now'

application = milla.Application(Root())
application.config.update(read_config('umbrella.ini'))
```

Notice that the section name appears in the dictionary key as well as the option name, separated by a dot (`.`). This allows you to specify have multiple options with the same name, as long as they are in different sections.

2.2 Allowing Various HTTP Methods

By default, *Milla* will reject HTTP requests using methods other than `GET`, `HEAD`, or `OPTIONS` by returning an HTTP 405 response. If you need a controller callable to accept these requests, you need to explicitly specify which methods are allowed.

To change the request methods that a controller callable accepts, use the `allow()` decorator.

```
@milla.allow('GET', 'HEAD', 'POST')
def controller(request):
    response = request.ResponseClass()
    if request.method == 'POST':
        release_t_virus()
        response.text = 'The T Virus has been released. Beware of Zombies'
        return response
    else:
        status = check_t_virus()
        response.text = 'The T Virus is {0}'.format(status)
        return response
```

Note: You do not need to explicitly allow the `OPTIONS` method; it is always allowed. If an `OPTIONS` request is made, *Milla* will automatically create a valid response informing the user of the allowed HTTP request methods for the given request path. Your controller will not be called in this case.

2.3 Controlling Access

Milla provides a powerful and extensible authorization framework that can be used to restrict access to different parts of a web application based on properties of the request. This framework has two major components—request validators and permission requirements. To use the framework, you must implement a *request validator* and then apply a *permission requirement* decorator to your *controller* callables as needed.

2.3.1 Request Validators

The default request validator (`milla.auth.RequestValidator`) is likely sufficient for most needs, as it assumes that a user is associated with a request (via the `user` attribute on the `Request` object) and that the user has a `permissions` attribute that contains a list of permissions the user holds.

Note: *Milla* does not automatically add a `user` attribute to `Request` instances, nor does it provide any way of determining what permissions the user has. As such, you will need to handle both of these on your own by utilizing the *Before and After Hooks*.

Request validators are classes that have a `validate` method that takes a request and optionally a permission requirement. The `validate` method should return `None` if the request meets the requirements or raise `NotAuthorized` (or a subclass thereof) if it does not. This exception will be called as the controller instead of the actual controller if the request is not valid.

If you'd like to customize the response to invalid requests or the default request validator is otherwise insufficient for your needs, you can create your own request validator. To do this, you need to do the following:

1. Create a subclass of `RequestValidator` that overrides `validate()` method (taking care to return `None` for valid requests and raise a subclass of `NotAuthorized` for invalid requests)
2. Register the new request validator in the `milla.request_validator` entry point group in your `setup.py`

   For example:

   ```
   setup(name='UmbrellaCorpWeb',
         ...
         entry_points={
             'milla.request_validator': [
                 'html_login = umbrellacorpweb.lib:RequestValidatorLogin'
             ],
         },
   )
   ```

3. Set the `request_validator` application config key to the entry point name of the new request validator

   For example:

   ```
   application = milla.Application(Root())
   application.config['request_validator'] = 'html_login'
   ```

2.3.2 Permission Requirements

Permission requirements are used by request validators to check whether or not a request is authorized for a particular controller. Permission requirements are applied to controller callables by using the `require_perms()` decorator.

```
class Root(object):

    def __call__(self, request):
        return 'This controller requires no permission'

    @milla.require_perms('priority1')
    def special(self, request):
        return 'This controller requires Priority 1 permission'
```

You can specify advanced permission requirements by using `Permission` objects:

```
class Root(object):

    def __call__(self, request):
        return 'This controller requires no permission'

    @milla.require_perms(Permission('priority1') | Permission('alpha2'))
```

```
    def special(self, request):
        return 'This controller requires Priority 1 or Alpha 2 permission'
```

2.3.3 Example

The following example will demonstrate how to define a custom request validator that presents an HTML form to the user for failed requests, allowing them to log in:

setup.py:

```
from setuptools import setup

setup(name='MyMillaApp',
      version='1.0',
      install_requires='Milla',
      py_modules=['mymillaapp'],
      entry_points={
          'milla.request_validator': [
              'html_login = mymillaapp:RequestValidatorLogin',
          ],
      },
)
```

mymillaapp.py:

```
import milla
import milla.auth

class NotAuthorizedLogin(milla.auth.NotAuthorized):

    def __call__(self, request):
        response = request.ResponseClass()
        response.text = '''\
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Please Log In</title>
  <meta charset="UTF-8">
</head>
<body>
<h1>Please Log In</h1>
<div style="color: #ff0000;">{error}</div>
<form action="login" method="post">
<div>Username:</div>
<div><input type="text" name="username"></div>
<div>Password:</div>
<div><input type="password" name="password"></div>
<div><button type="submit">Submit</button></div>
</form>
</body>
</html>'''.format(error=self)
        response.status_int = 401
        response.headers['WWW-Authenticate'] = 'HTML-Form'
        return response

class RequestValidatorLogin(milla.auth.RequestValidator):
```

```
    exc_class = NotAuthorizedLogin

class Root(object):

    def __before__(self, request):
        # Actually determining the user from the request is beyond the
        # scope of this example. You'll probably want to use a cookie-
        # based session and a database for this.
        request.user = get_user_from_request(request)

    @milla.require_perms('kill_zombies')
    def kill_zombies(self, request):
        response = request.ResponseClass()
        response.text = 'You can kill zombies'
        return response

    def __call__(self, request):
        response = request.ResponseClass()
        response.text = "Nothing to see here. No zombies, that's for sure"
        return response

application = milla.Application(Root())
```

CHAPTER

THREE

CHANGE LOG

3.1 0.2

- Python 3 support
- Added new utility functions:
  - `http_date()`
  - `read_config()`
- Added `static_resource()`
- Corrected default handling of HTTP `OPTIONS` requests (Issue #5)
- Deprecated `milla.cli`
- Deprecated `Generator` in favor of `create_href()`

3.2 0.1.2

- Improvements to `FaviconController` (Issue #1)

3.3 0.1.1

- Fixed a bug when generating application-relative URLs with `URLGenerator`:

3.4 0.1

Initial release

CHAPTER

FOUR

API REFERENCE

4.1 milla.auth

4.1.1 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_somehow(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_somehow(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`

4.1.2 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.

4.2 milla.dispatch

4.2.1 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
```

Deprecated since version 0.2: Use `milla.Request.create_href()` instead.

`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 0x2bb4520>
Compiled regular expression for variable segments

4.2.2 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

4.3 milla.app

Module milla.app

Please give me a docstring!

Created Mar 26, 2011

Author dustin

Updated $Date$

Updater $Author$

class milla.app.**Application**(*obj*)
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 **obj** – An object implementing the dispatcher protocol, or an object to be used as the root for a Traverser

Application instances are WSGI applications.

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

4.4 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)

EXPIRY_DAYS = 365
Number of days in the future to set the cache expiration for the icon

4.5 milla

Milla is an extremely simple WSGI framework for web applications

class milla.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
WebOb Request with minor tweaks

GET
Return a MultiDict containing all the variables from the QUERY_STRING.

POST
Return a MultiDict containing all the variables from a form request. Returns an empty dict-like object for non-form requests.

Form requests are typically POST requests, however PUT requests with an appropriate Content-Type are also supported.

ResponseClass
alias of `Response`

accept
Gets and sets the `Accept` header (HTTP spec section 14.1).

accept_charset
Gets and sets the `Accept-Charset` header (HTTP spec section 14.2).

accept_encoding
Gets and sets the `Accept-Encoding` header (HTTP spec section 14.3).

accept_language
Gets and sets the `Accept-Language` header (HTTP spec section 14.4).

application_url
The URL including SCRIPT_NAME (no PATH_INFO or query string)

as_bytes(*skip_body=False*)
Return HTTP bytes representing this request. If skip_body is True, exclude the body. If skip_body is an integer larger than one, skip body only if its length is bigger than that number.

authorization
Gets and sets the `Authorization` header (HTTP spec section 14.8). Converts it using `parse_auth` and `serialize_auth`.

classmethod blank(*path*, **args*, ***kwargs*)
Create a simple request for the specified path

See `webob.Request.blank` for information on other arguments and keywords

body
Return the content of the request body.

body_file
Input stream of the request (wsgi.input). Setting this property resets the content_length and seekable flag (unlike setting req.body_file_raw).

body_file_raw
Gets and sets the `wsgi.input` key in the environment.

body_file_seekable
Get the body of the request (wsgi.input) as a seekable file-like object. Middleware and routing applications should use this attribute over .body_file.

If you access this value, CONTENT_LENGTH will also be updated.

cache_control
Get/set/modify the Cache-Control header (HTTP spec section 14.9)

call_application(*application*, *catch_exc_info=False*)
Call the given WSGI application, returning `(status_string, headerlist, app_iter)`

Be sure to call `app_iter.close()` if it's there.

If catch_exc_info is true, then returns `(status_string, headerlist, app_iter, exc_info)`, where the fourth item may be None, but won't be if there was an exception. If you don't do this and there was an exception, the exception will be raised directly.

client_addr
The effective client IP address as a string. If the `HTTP_X_FORWARDED_FOR` header exists in the WSGI environ, this attribute returns the client IP address present in that header (e.g. if the header value is `192.168.1.1, 192.168.1.2`, the value will be `192.168.1.1`). If no `HTTP_X_FORWARDED_FOR` header is present in the environ at all, this attribute will return the value of the `REMOTE_ADDR` header. If the `REMOTE_ADDR` header is unset, this attribute will return the value `None`.

> **Warning:** It is possible for user agents to put someone else's IP or just any string in `HTTP_X_FORWARDED_FOR` as it is a normal HTTP header. Forward proxies can also provide incorrect values (private IP addresses etc). You cannot "blindly" trust the result of this method to provide you with valid data unless you're certain that `HTTP_X_FORWARDED_FOR` has the correct values. The WSGI server must be behind a trusted proxy for this to be true.

content_length
Gets and sets the `Content-Length` header (HTTP spec section 14.13). Converts it using int.

content_type
Return the content type, but leaving off any parameters (like charset, but also things like the type in `application/atom+xml; type=entry`)

If you set this property, you can include parameters, or if you don't include any parameters in the value then existing parameters will be preserved.

cookies
Return a dictionary of cookies as found in the request.

copy()
Copy the request and environment object.

This only does a shallow copy, except of wsgi.input

copy_body()
Copies the body, in cases where it might be shared with another request object and that is not desired.

This copies the body in-place, either into a BytesIO object or a temporary file.

copy_get()
Copies the request and environment object, but turning this request into a GET along the way. If this was a POST request (or any other verb) then it becomes GET, and the request body is thrown away.

create_href(*path*, ***keywords*)
Combine the application's path with a path to form an HREF

> **Parameters** **path** – relative path to join with the request URL

Any other keyword arguments will be encoded and appended to the URL as querystring arguments.

The HREF returned will will be the absolute path on the same host and protocol as the request. To get the full URL including scheme and host information, use `create_href_full()` instead.

create_href_full(*path*, ***keywords*)
Combine the application's full URL with a path to form a new URL

> **Parameters** **path** – relative path to join with the request URL

Any other keyword arguments will be encoded and appended to the URL as querystring arguments/

The HREF returned will be the full URL, including scheme and host information. To get the path only, use `create_href()` instead.

date
Gets and sets the `Date` header (HTTP spec section 14.8). Converts it using HTTP date.

classmethod from_bytes(*b*)
Create a request from HTTP bytes data. If the bytes contain extra data after the request, raise a ValueError.

classmethod from_file(*fp*)
Read a request from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the request, not the end of the file (unless the request is a POST or PUT and has no Content-Length, in that case, the entire file is read).

This reads the request as represented by `str(req)`; it may not read every valid HTTP request properly.

get_response(*application=None*, *catch_exc_info=False*)
Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

headers
All the request headers as a case-insensitive dictionary-like object.

host
Host name provided in HTTP_HOST, with fall-back to SERVER_NAME

host_port
The effective server port number as a string. If the `HTTP_HOST` header exists in the WSGI environ, this attribute returns the port number present in that header. If the `HTTP_HOST` header exists but contains no explicit port number: if the WSGI url scheme is "https" , this attribute returns "443", if the WSGI url scheme is "http", this attribute returns "80" . If no `HTTP_HOST` header is present in the environ at all, this attribute will return the value of the `SERVER_PORT` header (which is guaranteed to be present).

host_url
The URL through the host (no path)

http_version
Gets and sets the `SERVER_PROTOCOL` key in the environment.

if_match
Gets and sets the `If-Match` header (HTTP spec section 14.24). Converts it as a Etag.

if_modified_since
Gets and sets the `If-Modified-Since` header (HTTP spec section 14.25). Converts it using HTTP date.

if_none_match
Gets and sets the `If-None-Match` header (HTTP spec section 14.26). Converts it as a Etag.

if_range
Gets and sets the `If-Range` header (HTTP spec section 14.27). Converts it using IfRange object.

if_unmodified_since
Gets and sets the `If-Unmodified-Since` header (HTTP spec section 14.28). Converts it using HTTP date.

is_body_readable
webob.is_body_readable is a flag that tells us that we can read the input stream even though CONTENT_LENGTH is missing. This allows FakeCGIBody to work and can be used by servers to support chunked encoding in requests. For background see https://bitbucket.org/ianb/webob/issue/6

is_body_seekable
Gets and sets the `webob.is_body_seekable` key in the environment.

is_xhr
Is X-Requested-With header present and equal to `XMLHttpRequest`?

Note: this isn't set by every XMLHttpRequest request, it is only set if you are using a Javascript library that sets it (or you set the header yourself manually). Currently Prototype and jQuery are known to set this header.

json
Access the body of the request as JSON

json_body
Access the body of the request as JSON

make_body_seekable()
This forces `environ['wsgi.input']` to be seekable. That means that, the content is copied into a BytesIO or temporary file and flagged as seekable, so that it will not be unnecessarily copied again.

After calling this method the .body_file is always seeked to the start of file and .content_length is not None.

The choice to copy to BytesIO is made from `self.request_body_tempfile_limit`

make_tempfile()
Create a tempfile to store big request body. This API is not stable yet. A 'size' argument might be added.

max_forwards
Gets and sets the `Max-Forwards` header (HTTP spec section 14.31). Converts it using int.

method
Gets and sets the `REQUEST_METHOD` key in the environment.

params
A dictionary-like object containing both the parameters from the query string and request body.

path
The path of the request, without host or query string

path_info
Gets and sets the `PATH_INFO` key in the environment.

path_info_peek()
Returns the next segment on PATH_INFO, or None if there is no next segment. Doesn't modify the environment.

path_info_pop(*pattern=None*)
'Pops' off the next segment of PATH_INFO, pushing it onto SCRIPT_NAME, and returning the popped segment. Returns None if there is nothing left on PATH_INFO.

Does not return '' when there's an empty segment (like `/path//path`); these segments are just ignored.

Optional `pattern` argument is a regexp to match the return value before returning. If there is no match, no changes are made to the request and None is returned.

path_qs
The path of the request, without host but with query string

path_url
The URL including SCRIPT_NAME and PATH_INFO, but not QUERY_STRING

pragma
Gets and sets the `Pragma` header (HTTP spec section 14.32).

query_string
Gets and sets the `QUERY_STRING` key in the environment.

range
Gets and sets the `Range` header (HTTP spec section 14.35). Converts it using Range object.

referer
Gets and sets the `Referer` header (HTTP spec section 14.36).

referrer
Gets and sets the `Referer` header (HTTP spec section 14.36).

relative_url(*other_url*, *to_application=False*)
Resolve other_url relative to the request URL.

If `to_application` is True, then resolve it relative to the URL with only SCRIPT_NAME

remote_addr
Gets and sets the `REMOTE_ADDR` key in the environment.

remote_user
Gets and sets the `REMOTE_USER` key in the environment.

remove_conditional_headers(*remove_encoding=True*, *remove_range=True*, *remove_match=True*, *remove_modified=True*)
Remove headers that make the request conditional.

These headers can cause the response to be 304 Not Modified, which in some cases you may not want to be possible.

This does not remove headers like If-Match, which are used for conflict detection.

scheme
Gets and sets the `wsgi.url_scheme` key in the environment.

script_name
Gets and sets the `SCRIPT_NAME` key in the environment.

send(*application=None*, *catch_exc_info=False*)
Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

server_name
Gets and sets the `SERVER_NAME` key in the environment.

server_port
Gets and sets the `SERVER_PORT` key in the environment. Converts it using int.

static_resource(*path*)
Return a URL to the given static resource

This method combines the defined static resource root URL with the given path to construct a complete URL to the given resource. The resource root should be defined in the application configuration dictionary, under the name `milla.static_root`, for example:

```
app = milla.Application(dispatcher)
app.config.update({
    'milla.static_root': '/static/'
})
```

Then, calling `static_resource` on a `Request` object (i.e. inside a controller callable) would combine the given path with `/static/`, like this:

```
request.static_resource('/images/foo.png')
```

would return `/static/images/foo.png`.

If no `milla.static_root` key is found in the configuration dictionary, the path will be returned unaltered.

Parameters **path** – Path to the resource, relative to the defined root

str_GET
<Deprecated attribute None>

str_POST
<Deprecated attribute None>

str_cookies
<Deprecated attribute None>

str_params
<Deprecated attribute None>

text
Get/set the text value of the body

upath_info
Gets and sets the `PATH_INFO` key in the environment.

url
The full request URL, including QUERY_STRING

url_encoding
Gets and sets the `webob.url_encoding` key in the environment.

urlargs
Return any *positional* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like `routes` set this value.

urlvars
Return any *named* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like `routes` set this value.

uscript_name
Gets and sets the `SCRIPT_NAME` key in the environment.

user_agent
Gets and sets the `User-Agent` header (HTTP spec section 14.43).

class `milla.`**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
`WebOb Response` with minor tweaks

accept_ranges
Gets and sets the `Accept-Ranges` header (HTTP spec section 14.5).

age
Gets and sets the `Age` header (HTTP spec section 14.6). Converts it using int.

allow
Gets and sets the `Allow` header (HTTP spec section 14.7). Converts it using list.

app_iter
Returns the app_iter of the response.

If body was set, this will create an app_iter from that body (a single-item list)

app_iter_range(*start*, *stop*)
Return a new app_iter built from the response app_iter, that serves up only the given `start:stop` range.

body
The body of the response, as a `str`. This will read in the entire app_iter if necessary.

body_file
A file-like object that can be used to write to the body. If you passed in a list app_iter, that app_iter will be modified by writes.

cache_control
Get/set/modify the Cache-Control header (HTTP spec section 14.9)

charset
Get/set the charset (in the Content-Type)

conditional_response_app(*environ*, *start_response*)
Like the normal __call__ interface, but checks conditional headers:

- If-Modified-Since (304 Not Modified; only on GET, HEAD)
- If-None-Match (304 Not Modified; only on GET, HEAD)
- Range (406 Partial Content; only on GET, HEAD)

content_disposition
Gets and sets the `Content-Disposition` header (HTTP spec section 19.5.1).

content_encoding
Gets and sets the `Content-Encoding` header (HTTP spec section 14.11).

content_language
Gets and sets the `Content-Language` header (HTTP spec section 14.12). Converts it using list.

content_length
Gets and sets the `Content-Length` header (HTTP spec section 14.17). Converts it using int.

content_location
Gets and sets the `Content-Location` header (HTTP spec section 14.14).

content_md5
Gets and sets the `Content-MD5` header (HTTP spec section 14.14).

content_range
Gets and sets the `Content-Range` header (HTTP spec section 14.16). Converts it using ContentRange object.

content_type
Get/set the Content-Type header (or None), *without* the charset or any parameters.

If you include parameters (or `;` at all) when setting the content_type, any existing parameters will be deleted; otherwise they will be preserved.

content_type_params
A dictionary of all the parameters in the content type.

(This is not a view, set to change, modifications of the dict would not be applied otherwise)

copy()
Makes a copy of the response

date
Gets and sets the `Date` header (HTTP spec section 14.18). Converts it using HTTP date.

delete_cookie(*key*, *path='/'*, *domain=None*)
Delete a cookie from the client. Note that path and domain must match how the cookie was originally set.

This sets the cookie to the empty string, and max_age=0 so that it should expire immediately.

encode_content(*encoding='gzip'*, *lazy=False*)
Encode the content with the given encoding (only gzip and identity are supported).

etag
Gets and sets the `ETag` header (HTTP spec section 14.19). Converts it using Entity tag.

expires
Gets and sets the `Expires` header (HTTP spec section 14.21). Converts it using HTTP date.

classmethod from_file(*fp*)
Reads a response from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the response, not the end of the file.

This reads the response as represented by `str(resp)`; it may not read every valid HTTP response properly. Responses must have a `Content-Length`

headerlist
The list of response headers

headers
The headers in a dictionary-like object

json
Access the body of the response as JSON

json_body
Access the body of the response as JSON

last_modified
Gets and sets the `Last-Modified` header (HTTP spec section 14.29). Converts it using HTTP date.

location
Gets and sets the `Location` header (HTTP spec section 14.30).

md5_etag(*body=None*, *set_content_md5=False*)
Generate an etag for the response object using an MD5 hash of the body (the body parameter, or `self.body` if not given)

Sets `self.etag` If `set_content_md5` is True sets `self.content_md5` as well

merge_cookies(*resp*)
Merge the cookies that were set on this response with the given *resp* object (which can be any WSGI application).

If the *resp* is a `webob.Response` object, then the other object will be modified in-place.

pragma
Gets and sets the `Pragma` header (HTTP spec section 14.32).

retry_after
Gets and sets the `Retry-After` header (HTTP spec section 14.37). Converts it using HTTP date or delta seconds.

server
Gets and sets the `Server` header (HTTP spec section 14.38).

set_cookie(*key*, *value=''*, *max_age=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *comment=None*, *expires=None*, *overwrite=False*)
Set (add) a cookie for the response.

Arguments are:

`key`

The cookie name.

`value`

The cookie value, which should be a string or `None`. If `value` is `None`, it's equivalent to calling the `webob.response.Response.unset_cookie()` method for this cookie key (it effectively deletes the cookie on the client).

`max_age`

An integer representing a number of seconds or `None`. If this value is an integer, it is used as the `Max-Age` of the generated cookie. If `expires` is not passed and this value is an integer, the `max_age` value will also influence the `Expires` value of the cookie (`Expires` will be set to now + max_age). If this value is `None`, the cookie will not have a `Max-Age` value (unless `expires` is also sent).

`path`

A string representing the cookie `Path` value. It defaults to `/`.

`domain`

A string representing the cookie `Domain`, or `None`. If domain is `None`, no `Domain` value will be sent in the cookie.

`secure`

A boolean. If it's `True`, the `secure` flag will be sent in the cookie, if it's `False`, the `secure` flag will not be sent in the cookie.

`httponly`

A boolean. If it's `True`, the `HttpOnly` flag will be sent in the cookie, if it's `False`, the `HttpOnly` flag will not be sent in the cookie.

`comment`

A string representing the cookie `Comment` value, or `None`. If `comment` is `None`, no `Comment` value will be sent in the cookie.

`expires`

A `datetime.timedelta` object representing an amount of time or the value `None`. A non-`None` value is used to generate the `Expires` value of the generated cookie. If `max_age` is not passed, but this value is not `None`, it will influence the `Max-Age` header (`Max-Age` will be 'expires_value - datetime.utcnow()'). If this value is `None`, the `Expires` cookie value will be unset (unless `max_age` is also passed).

`overwrite`

If this key is `True`, before setting the cookie, unset any existing cookie.

status
The status string

status_code
The status as an integer

status_int
The status as an integer

text
Get/set the text value of the body (using the charset of the Content-Type)

ubody
Deprecated alias for .text

unicode_body
Deprecated alias for .text

unset_cookie(*key*, *strict=True*)
Unset a cookie with the given name (remove it from the response).

vary
Gets and sets the `Vary` header (HTTP spec section 14.44). Converts it using list.

www_authenticate
Gets and sets the `WWW-Authenticate` header (HTTP spec section 14.47). Converts it using `parse_auth` and `serialize_auth`.

milla.**allow**(**methods*)
Specify the allowed HTTP verbs for a controller callable

Example:

```
@milla.allow('GET', 'POST')
def controller(request):
    return 'Hello, world!'
```

4.6 milla.util

Convenience utility functions

Created Mar 30, 2011

Author dustin

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.util.`**`http_date`**(*date*)
Format a datetime object as a string in RFC 1123 format

This function returns a string representing the date according to RFC 1123. The string returned will always be in English, as required by the specification.

Parameters **date** – A `datetime.datetime` object

Returns RFC 1123-formatted string

`milla.util.`**`read_config`**(*filename*, *defaults=None*)
Parse an ini file into a nested dictionary

Parameters

- **filename** (*string*) – Path to the ini file to read
- **defaults** (*dict*) – (Optional) A mapping of default values that can be used for interpolation when reading the configuration file

Returns A dictionary whose keys correspond to the section and option, joined with a dot character (.)

For example, consider the following ini file:

```
[xmen]
storm = Ororo Monroe
cyclops = Scott Summers

[avengers]
hulk = Bruce Banner
iron_man = Tony Stark
```

The resulting dictionary would look like this:

```
{
    'xmen.storm': 'Ororo Monroe',
    'xmen.cyclops': 'Scott Summers',
    'avengers.hulk': 'Bruce Banner',
    'avengers.iron_man': 'Tony Stark',
}
```

Thus, the option values for any section can be obtained as follows:

```
config['xmen.storm']
```

This dictionary can be used to configure an `Application` instance by using the `update` method:

```
config = milla.util.read_config('superheros.ini')
app = milla.Application(router)
app.config.update(config)
```

CHAPTER

FIVE

GLOSSARY

controller, controller callable A callable that accepts a `Request` instance and any optional parameters and returns a response

permission requirement A set of permissions required to access a particular URL path. Permission requirements are specified by using the `require_perm()` decorator on a restricted *controller callable*

request validator A function that checks a request to ensure it meets the specified *permission requirement* before calling a *controller callable*

root object The starting object in the object traversal URL dispatch mechanism from which all path lookups are performed

URL dispatcher An object that maps request paths to *controller* callables

Milla is a simple and lightweight web framework for Python. It built on top of WebOb and thus implements the WSGI standard. It aims to be easy to use while imposing no restrictions, allowing web developers to write code the way they want, using the tools, platform, and extensions they choose.

CHAPTER

SIX

EXAMPLE

```
from wsgiref import simple_server
from milla.dispatch import routing
import milla


def hello(request):
    return 'Hello, world!'

router = routing.Router()
router.add_route('/', hello)
app = milla.Application(router)

httpd = simple_server.make_server('', 8080, app)
httpd.serve_forever()
```

Milla is released under the terms of the Apache License, version 2.0.

CHAPTER

SEVEN

INDICES AND TABLES

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

PYTHON MODULE INDEX

m