Routes Documentation

Release 1.12

Ben Bangert, Mike Orr

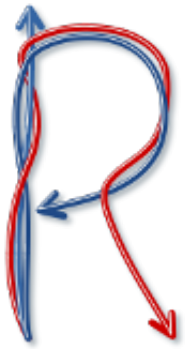
March 01, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION



Routes tackles an interesting problem that comes up frequently in web development, *how do you map URLs to your application's actions*? That is, how do you say that *this* should be accessed as "/blog/2008/01/08", and "/login" should do *that*? Many web frameworks have a fixed dispatching system; e.g., "/A/B/C" means to read file "C" in directory "B", or to call method "C" of class "B" in module "A.B". These work fine until you need to refactor your code and realize that moving a method changes its public URL and invalidates users' bookmarks. Likewise, if you want to reorganize your URLs and make a section into a subsection, you have to change your carefully-tested logic code.

Routes takes a different approach. You determine your URL hierarchy and and actions separately, and then link them together in whichever ways you decide. If you change your mind about a particular URL, just change one line in your route map and never touch your action logic. You can even have multiple URLs pointing to the same action; e.g., to support legacy bookmarks. Routes was originally inspired by the dispatcher in Ruby on Rails but has since diverged.

Routes is the primary dispatching system in the Pylons web framework, and an optional choice in CherryPy. It can be added to any framework without much fuss, and used for an entire site or a URL subtree. It can also forward subtrees to other dispatching systems, which is how TurboGears 2 is implemented on top of Pylons.

Current features:

- Sophisticated route lookup and URL generation
- Named routes
- Redirect routes
- Wildcard paths before and after static parts
- Sub-domain support built-in
- Conditional matching based on domain, cookies, HTTP method (RESTful), and more

- Easily extensible utilizing custom condition functions and route generation functions
- Extensive unit tests

Buzzword compliance: REST, DRY.

If you're new to Routes or have not read the Routes 1.11 manual before, we recommend reading the Glossary before continuing.

This manual is written from the user's perspective: how to use Routes in a framework that already supports it. The Porting manual describes how to add Routes support to a new framework.

You may have heard about a development version called "Routes 2". Routes 2 is now called "Routes-experimental". It was originally intended to be a refactoring with a new API. Instead its features are being incorporated into Routes 1 in a compatible manner. There may be another Routes 2 in the future that drops deprecated features, but it's too early to say when/if that might happen.

CHAPTER

TWO

SETTING UP ROUTES

It is assumed that you are using a framework that has preconfigured Routes for you. In Pylons, you define your routes in the `make_map` function in your *myapp/config/routing.py* module. Here is a typical configuration:

```
1   from routes import Mapper
2   map = Mapper()
3   map.connect(None, "/error/{action}/{id}, controller="error")
4   map.connect("home", "/", controller="main", action="index")
5   # ADD CUSTOM ROUTES HERE
6   map.connect(None, "/{controller}/{action}")
7   map.connect(None, "/{controller}/{action}/{id}")
```

Lines 1 and 2 create a mapper.

Line 3 matches any three-component route that starts with "/error", and sets the "controller" variable to a constant, so that a URL "/error/images/arrow.jpg" would produce:

```
{"controller": "error", "action": "images", "id": "arrow.jpg"}
```

Line 4 matches the single URL "/", and sets both the controller and action to constants. It also has a route name "home", which can be used in generation. (The other routes have `None` instead of a name, so they don't have names. It's recommended to name all routes that may be used in generation, but it's not necessary to name other routes.)

Line 6 matches any two-component URL, and line 7 matches any 3-component URL. These are used as catchall routes if we're too lazy to define a separate route for every action. If you *have* defined a route for every action, you can delete these two routes.

Note that a URL "/error/images/arrow.jpg" could match both line 3 and line 7. The mapper resolves this by trying routes in the order defined, so this URL would match line 3.

If no routes match the URL, the mapper returns a "match failed" condition, which is seen in Pylons as HTTP 404 "Not Found".

Here are some more examples of valid routes:

```
m.connect("/feeds/{category}/atom.xml", controller="feeds", action="atom")
m.connect("history", "/archives/by_eon/{century}", controller="archives",
          action="aggregate")
m.connect("article", "/article/{section}/{slug}/{page}.html",
          controller="article", action="view")
```

Extra variables may be any Python type, not just strings. However, if the route is used in generation, `str()` will be called on the value unless the generation call specifies an overriding value.

Other argument syntaxes are allowed for compatibility with earlier versions of Routes. These are described in the `Backward Compatibility` section.

Route paths should always begin with a slash ("/"). Earlier versions of Routes allowed slashless paths, but their behavior now is undefined.

2.1 Requirements

It's possible to restrict a path variable to a regular expression; e.g., to match only a numeric component or a restricted choice of words. There are two syntaxes for this: inline and the `requirements` argument. An inline requirement looks like this:

```
map.connect(R"/blog/{id:\d+}")
map.connect(R"/download/{platform:windows|mac}/{filename}")
```

This matches "/blog/123" but not "/blog/12A". The equivalent `requirements` syntax is:

```
map.connect("/blog/{id}", requirements={"id": R"\d+"}
map.connect("/download/{platform}/{filename}",
    requirements={"platform": R"windows|mac"})
```

Note the use of raw string syntax (`R""`) for regexes which might contain backslashes. Without the R you'd have to double every backslash.

Another example:

```
m.connect("archives/{year}/{month}/{day}", controller="archives",
          action="view", year=2004,
          requirements=dict(year=R"\d{2,4}", month=R"\d{1,2}"))
```

The inline syntax was added in Routes (XXX 1.10?? not in changelog). Previous versions had only the `requirements` argument. Two advantages of the `requirements` argument are that if you have several variables with identical requirements, you can set one variable or even the entire argument to a global:

```
NUMERIC = R"\d+"
map.connect(..., requirements={"id": NUMERIC})

ARTICLE_REQS = {"year": R"\d\d\d\d", "month": R"\d\d", "day": R"\d\d"}
map.connect(..., requirements=ARTICLE_REQS)
```

Because the argument `requirements` is reserved, you can't define a routing variable by that name.

2.2 Magic path_info

If the "path_info" variable is used at the end of the URL, Routes moves everything preceding it into the "SCRIPT_NAME" environment variable. This is useful when delegating to another WSGI application that does its own routing: the subapplication will route on the remainder of the URL rather than the entire URL. You still need the ":.*" requirement to capture the following URL components into the variable.

```
map.connect(None, "/cards/{path_info:.*}",
    controller="main", action="cards")
# Incoming URL "/cards/diamonds/4.png"
=> {"controller": "main", action: "cards", "path_info": "/diamonds/4.png"}
# Second WSGI application sees:
# SCRIPT_NAME="/cards"   PATH_INFO="/diamonds/4.png"
```

This route does not match "/cards" because it requires a following slash. Add another route to get around this:

```
map.connect("cards", "/cards", controller="main", action="cards",
    path_info="/")
```

Tip: You may think you can combine the two with the following route:

```
map.connect("cards", "/cards{path_info:.*}",
    controller="main", action="cards")
```

There are two problems with this, however. One, it would also match "/cardshark". Two, Routes 1.10 has a bug: it forgets to take the suffix off the SCRIPT_NAME.

A future version of Routes may delegate directly to WSGI applications, but for now this must be done in the framework. In Pylons, you can do this in a controller action as follows:

```
from paste.fileapp import DirectoryApp
def cards(self, environ, start_response):
    app = DirectoryApp("/cards-directory")
    return app(environ, start_response)
```

Or create a fake controller module with a `__controller__` variable set to the WSGI application:

```
from paste.fileapp import DirectoryApp
__controller__ = DirectoryApp("/cards-directory")
```

2.3 Conditions

Conditions impose additional constraints on what kinds of requests can match. The `conditions` argument is a dict with up to three keys:

method

A list of uppercase HTTP methods. The request must be one of the listed methods.

sub_domain

Can be a list of subdomains, `True`, `False`, or `None`. If a list, the request must be for one of the specified subdomains. If `True`, the request must contain a subdomain but it can be anything. If `False` or `None`, do not match if there's a subdomain.

New in Routes 1.10: ''False'' and ''None'' values.

function

A function that evaluates the request. Its signature must be `func(environ, match_dict) => bool`. It should return true if the match is successful or false otherwise. The first arg is the WSGI environment; the second is the routing variables

that would be returned if the match succeeds. The function can modify `match_dict` in place to affect which variables are returned. This allows a wide range of transformations.

Examples:

```
# Match only if the HTTP method is "GET" or "HEAD".
m.connect("/user/list", controller="user", action="list",
          conditions=dict(method=["GET", "HEAD"]))

# A sub-domain should be present.
m.connect("/", controller="user", action="home",
          conditions=dict(sub_domain=True))

# Sub-domain should be either "fred" or "george".
m.connect("/", controller="user", action="home",
          conditions=dict(sub_domain=["fred", "george"]))

# Put the referrer into the resulting match dictionary.
# This function always returns true, so it never prevents the match
# from succeeding.
def referals(environ, result):
    result["referer"] = environ.get("HTTP_REFERER")
    return True
m.connect("/{controller}/{action}/{id}",
    conditions=dict(function=referals))
```

2.4 Wildcard routes

By default, path variables do not match a slash. This ensures that each variable will match exactly one component. You can use requirements to override this:

```
map.connect("/static/{filename:.*?}")
```

This matches "/static/foo.jpg", "/static/bar/foo.jpg", etc.

Beware that careless regexes may eat the entire rest of the URL and cause components to the right of it not to match:

```
# OK because the following component is static and the regex has a "?".
map.connect("/static/{filename:.*?}/download")
```

The lesson is to always test wildcard patterns.

2.5 Format extensions

A path component of `{.format}` will match an optional format extension (e.g. ".html" or ".json"), setting the format variable to the part after the "." (e.g. "html" or "json") if there is one, or to `None` otherwise. For example:

```
map.connect('/entries/{id}{.format}')
```

will match "/entries/1" and "/entries/1.mp3". You can use requirements to limit which extensions will match, for example:

```
map.connect('/entries/{id:\d+}{.format:json}')
```

will match "/entries/1" and "/entries/1.json" but not "/entries/1.mp3".

As with wildcard routes, it's important to understand and test this. Without the `\d+` requirement on the `id` variable above, "/entries/1.mp3" would match successfully, with the `id` variable capturing "1.mp3".

New in Routes 1.12.

2.6 Submappers

A submapper lets you add several similar routes without having to repeat identical keyword arguments. There are two syntaxes, one using a Python `with` block, and the other avoiding it.

```
# Using 'with'
with map.submapper(controller="home") as m:
    m.connect("home", "/", action="splash")
    m.connect("index", "/index", action="index")

# Not using 'with'
m = map.submapper(controller="home")
m.connect("home", "/", action="splash")
m.connect("index", "/index", action="index")

# Both of these syntaxes create the following routes::
# "/"      => {"controller": "home", action="splash"}
# "/index" => {"controller": "home", action="index"}
```

You can also specify a common path prefix for your routes:

```
with map.submapper(path_prefix="/admin", controller="admin") as m:
    m.connect("admin_users", "/users", action="users")
    m.connect("admin_databases", "/databases", action="databases")

# /admin/users     => {"controller": "admin", "action": "users"}
# /admin/databases => {"controller": "admin", "action": "databases"}
```

All arguments to `.submapper` must be keyword arguments.

The submapper is *not* a complete mapper. It's just a temporary object with a `.connect` method that adds routes to the mapper it was spawned from.

New in Routes 1.11.

2.7 Submapper helpers

Submappers contain a number of helpers that further simplify routing configuration. This:

```
with map.submapper(controller="home") as m:
    m.connect("home", "/", action="splash")
    m.connect("index", "/index", action="index")
```

can be written:

```
with map.submapper(controller="home", path_prefix="/") as m:
    m.action("home", action="splash")
    m.link("index")
```

The `action` helper generates a route for one or more HTTP methods ('GET' is assumed) at the submapper's path ('/' in the example above). The `link` helper generates a route at a relative path.

There are specific helpers corresponding to the standard `index`, `new`, `create`, `show`, `edit`, `update` and `delete` actions. You can use these directly:

```
with map.submapper(controller="entries", path_prefix="/entries") as entries:
    entries.index()
    with entries.submapper(path_prefix="/{id}") as entry:
        entry.show()
```

or indirectly:

```
with map.submapper(controller="entries", path_prefix="/entries",
                   actions=["index"]) as entries:
    entries.submapper(path_prefix="/{id}", actions=["show"])
```

Collection/member submappers nested in this way are common enough that there is helper for this too:

```
map.collection(collection_name="entries", member_name="entry",
               controller="entries",
               collection_actions=["index"], member_actions["show"])
```

This returns a submapper instance to which further routes may be added; it has a `member` property (a nested submapper) to which which member-specific routes can be added. When `collection_actions` or `member_actions` are omitted, the full set of actions is generated (see the example under "Printing" below).

See "RESTful services" below for `map.resource`, a precursor to `map.collection` that does not use submappers.

New in Routes 1.12.

2.8 Adding routes from a nested application

New in Routes 1.11. Sometimes in nested applications, the child application gives the parent a list of routes to add to its mapper. These can be added with the `.extend` method, optionally providing a path prefix:

```
routes = [
    Route("index", "/index.html", controller="home", action="index"),
    ]

map.extend(routes)
# /index.html => {"controller": "home", "action": "index"}

map.extend(routes, "/subapp")
# /subapp/index.html => {"controller": "home", "action": "index"}
```

This does not exactly add the route objects to the mapper. It creates identical new route objects and adds those to the mapper.

New in Routes 1.11.

CHAPTER

THREE

GENERATION

To generate URLs, use the `url` or `url_for` object provided by your framework. `url` is an instance of Routes `URLGenerator`, while `url_for` is the older `routes.url_for()` function. `url_for` is being phased out, so new applications should use `url`.

To generate a named route, specify the route name as a positional argument:

```
url("home")   =>  "/"
```

If the route contains path variables, you must specify values for them using keyword arguments:

```
url("blog", year=2008, month=10, day=2)
```

Non-string values are automatically converted to strings using `str()`. (This may break with Unicode values containing non-ASCII characters.)

However, if the route defines an extra variable with the same name as a path variable, the extra variable is used as the default if that keyword is not specified. Example:

```
m.connect("archives", "/archives/{id}",
    controller="archives", action="view", id=1)
url("blog", id=123)  =>  "/blog/123"
url("blog")  =>  "/blog/1"
```

(The extra variable is *not* used for matching unless minimization is enabled.)

Any keyword args that do not correspond to path variables will be put in the query string. Append a "_" if the variable name collides with a Python keyword:

```
map.connect("archive", "/archive/{year}")
url("archive", year=2009, font=large)  =>  "/archive/2009?font=large"
url("archive", year=2009, print_=1)  =>  "/archive/2009?print=1"
```

If the application is mounted at a subdirectory of the URL space, all generated URLs will have the application prefix. The application prefix is the "SCRIPT_NAME" variable in the request's WSGI environment.

If the positional argument corresponds to no named route, it is assumed to be a literal URL. The application's mount point is prefixed to it, and keyword args are converted to query parameters:

```
url("/search", q="My question")  =>  "/search?q=My+question"
```

If there is no positional argument, Routes will use the keyword args to choose a route. The first route that has all path variables specified by keyword args and the fewest number of extra variables not overridden by keyword args will be chosen. This was common in older versions of Routes but can cause application bugs if an unexpected route is chosen, so using route names is much preferable because that guarantees only the named route will be chosen. The most common use for unnamed generation is when you have a seldom-used controller with a lot of ad hoc methods; e.g., `url(controller="admin", action="session")`.

An exception is raised if no route corresponds to the arguments. The exception is `routes.util.GenerationException`. (Prior to Routes 1.9, `None` was returned instead. It was changed to an exception to prevent invalid blank URLs from being insered into templates.)

You'll also get this exception if Python produces a Unicode URL (which could happen if the route path or a variable value is Unicode). Routes generates only `str` URLs.

The following keyword args are special:

anchor

Specifies the URL anchor (the part to the right of "#").

```
url("home", "summary")  =>  "/#summary"
```

host

Make the URL fully qualified and override the host (domain).

protocol

Make the URL fully qualified and override the protocol (e.g., "ftp").

qualified

Make the URL fully qualified (i.e., add "protocol://host:port" prefix).

sub_domain

See "Generating URLs with subdomains" below.

The syntax in this section is the same for both `url` and `url_for`.

New in Routes 1.10: ''url'' and the ''URLGenerator'' class behind it.

3.1 Generating routes based on the current URL

`url.current()` returns the URL of the current request, without the query string. This is called "route memory", and works only if the RoutesMiddleware is in the middleware stack. Keyword arguments override path variables or are put on the query string.

`url_for` combines the behavior of `url` and `url_current`. This is deprecated because nameless routes and route memory have the same syntax, which can lead to the wrong route being chosen in some cases.

Here's an example of route memory:

```
m.connect("/archives/{year}/{month}/{day}", year=2004)

# Current URL is "/archives/2005/10/4".
# Routing variables are {"controller": "archives", "action": "view",
  "year": "2005", "month": "10", "day": "4"}

url.current(day=6)    =>  "/archives/2005/10/6"
```

```
url.current(month=4)   =>  "/archives/2005/4/4"
url.current()          =>  "/archives/2005/10/4"
```

Route memory can be disabled globally with `map.explicit = True`.

3.2 Generation-only routes (aka. static routes)

A static route is used only for generation – not matching – and it must be named. To define a static route, use the argument `_static=True`.

This example provides a convenient way to link to a search:

```
map.connect("google", "http://google.com/", _static=True)
url("google", q="search term")  =>  "/http://google.com/?q=search+term")
```

This example generates a URL to a static image in a Pylons public directory. Pylons serves the public directory in a way that bypasses Routes, so there's no reason to match URLs under it.

```
map.connect("attachment", "/images/attachments/{category}/{id}.jpg",
    _static=True)
url("attachment", category="dogs", id="Mastiff") =>
    "/images/attachments/dogs/Mastiff.jpg"
```

Starting in Routes 1.10, static routes are exactly the same as regular routes except they're not added to the internal match table. In previous versions of Routes they could not contain path variables and they had to point to external URLs.

3.3 Filter functions

A filter function modifies how a named route is generated. Don't confuse it with a function condition, which is used in matching. A filter function is its opposite counterpart.

One use case is when you have a `story` object with attributes for year, month, and day. You don't want to hardcode these attributes in every `url` call because the interface may change someday. Instead you pass the story as a pseudo-argument, and the filter produces the actual generation args. Here's an example:

```
class Story(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @staticmethod
    def expand(kw):
        try:
            story = kw["story"]
        except KeyError:
            pass   # Don't modify dict if ``story`` key not present.
        else:
            # Set the actual generation args from the story.
            kw["year"] = story.year
            kw["month"] = story.month
            kw["day"] = story.day
```

```
        return kw

m.connect("archives", "/archives/{year}/{month}/{day}",
    controller="archives", action="view", _filter=Story.expand)

my_story = Story(2009, 1, 2)
url("archives", story=my_story)  =>  "/archives/2009/1/2"
```

The `_filter` argument can be any function that takes a dict and returns a dict. In the example we've used a static method of the `Story` class to keep everything story-related together, but you may prefer to use a standalone function to keep Routes-related code away from your model.

3.4 Generating URLs with subdomains

If subdomain support is enabled and the `sub_domain` arg is passed to `url_for`, Routes ensures the generated route points to that subdomain.

```
# Enable subdomain support.
map.sub_domains = True

# Ignore the www subdomain.
map.sub_domains_ignore = "www"

map.connect("/users/{action}")

# Add a subdomain.
url_for(action="update", sub_domain="fred")  =>  "http://fred.example.com/users/update"

# Delete a subdomain.  Assume current URL is fred.example.com.
url_for(action="new", sub_domain=None)  =>  "http://example.com/users/new"
```

CHAPTER

FOUR

RESTFUL SERVICES

Routes makes it easy to configure RESTful web services. `map.resource` creates a set of add/modify/delete routes conforming to the Atom publishing protocol.

A resource route addresses *members* in a *collection*, and the collection itself. Normally a collection is a plural word, and a member is the corresponding singular word. For instance, consider a collection of messages:

```
map.resource("message", "messages")

# The above command sets up several routes as if you had typed the
# following commands:
map.connect("messages", "/messages",
    controller="messages", action="create",
    conditions=dict(method=["POST"]))
map.connect("messages", "/messages",
    controller="messages", action="index",
    conditions=dict(method=["GET"]))
map.connect("formatted_messages", "/messages.{format}",
    controller="messages", action="index",
    conditions=dict(method=["GET"]))
map.connect("new_message", "/messages/new",
    controller="messages", action="new",
    conditions=dict(method=["GET"]))
map.connect("formatted_new_message", "/messages/new.{format}",
    controller="messages", action="new",
    conditions=dict(method=["GET"]))
map.connect("/messages/{id}",
    controller="messages", action="update",
    conditions=dict(method=["PUT"]))
map.connect("/messages/{id}",
    controller="messages", action="delete",
    conditions=dict(method=["DELETE"]))
map.connect("edit_message", "/messages/{id}/edit",
    controller="messages", action="edit",
    conditions=dict(method=["GET"]))
map.connect("formatted_edit_message", "/messages/{id}.{format}/edit",
    controller="messages", action="edit",
    conditions=dict(method=["GET"]))
map.connect("message", "/messages/{id}",
    controller="messages", action="show",
    conditions=dict(method=["GET"]))
map.connect("formatted_message", "/messages/{id}.{format}",
    controller="messages", action="show",
    conditions=dict(method=["GET"]))
```

This establishes the following convention:

```
GET    /messages        => messages.index()    => url("messages")
POST   /messages        => messages.create()   => url("messages")
GET    /messages/new    => messages.new()      => url("new_message")
PUT    /messages/1      => messages.update(id) => url("message", id=1)
DELETE /messages/1      => messages.delete(id) => url("message", id=1)
GET    /messages/1      => messages.show(id)   => url("message", id=1)
GET    /messages/1/edit => messages.edit(id)   => url("edit_message", id=1)
```

Thus, you GET the collection to see an index of links to members ("index" method). You GET a member to see it ("show"). You GET "COLLECTION/new" to obtain a new message form ("new"), which you POST to the collection ("create"). You GET "MEMBER/edit" to obtain an edit for ("edit"), which you PUT to the member ("update"). You DELETE the member to delete it. Note that there are only four route names because multiple actions are doubled up on the same URLs.

This URL structure may look strange if you're not used to the Atom protocol. REST is a vague term, and some people think it means proper URL syntax (every component contains the one on its right), others think it means not putting IDs in query parameters, and others think it means using HTTP methods beyond GET and POST. `map.resource` does all three, but it may be overkill for applications that don't need Atom compliance or prefer to stick with GET and POST. `map.resource` has the advantage that many automated tools and non-browser agents will be able to list and modify your resources without any programming on your part. But you don't have to use it if you prefer a simpler add/modify/delete structure.

HTML forms can produce only GET and POST requests. As a workaround, if a POST request contains a `_method` parameter, the Routes middleware changes the HTTP method to whatever the parameter specifies, as if it had been requested that way in the first place. This convention is becoming increasingly common in other frameworks. If you're using WebHelpers, the The WebHelpers `form` function has a `method` argument which automatically sets the HTTP method and "_method" parameter.

Several routes are paired with an identical route containing the `format` variable. The intention is to allow users to obtain different formats by means of filename suffixes; e.g., "/messages/1.xml". This produces a routing variable "xml", which in Pylons will be passed to the controller action if it defines a formal argument for it. In generation you can pass the `format` argument to produce a URL with that suffix:

```
url("message", id=1, format="xml")  =>  "/messages/1.xml"
```

Routes does not recognize any particular formats or know which ones are valid for your application. It merely passes the `format` attribute through if it appears.

New in Routes 1.7.3: changed URL suffix from ";edit" to "/edit". Semicolons are not allowed in the path portion of a URL except to delimit path parameters, which nobody uses.

4.1 Resource options

The `map.resource` method recognizes a number of keyword args which modifies its behavior:

controller

Use the specified controller rather than deducing it from the collection name.

collection

Additional URLs to allow for the collection. Example:

```
map.resource("message", "messages", collection={"rss": "GET"})
# "GET /message/rss"  =>  ``Messages.rss()``.
# Defines a named route "rss_messages".
```

member

Additional URLs to allow for a member. Example:

```
map.resource('message', 'messages', member={'mark':'POST'})
# "POST /message/1/mark"  =>  ``Messages.mark(1)``
# also adds named route "mark_message"
```

This can be used to display a delete confirmation form:

```
map.resource("message", "messages", member={"ask_delete": "GET"}
# "GET /message/1/ask_delete"   =>   ``Messages.ask_delete(1)``.
# Also adds a named route "ask_delete_message".
```

new

Additional URLs to allow for new-member functionality.

```
map.resource("message", "messages", new={"preview": "POST"})
# "POST /messages/new/preview"
```

path_prefix

Prepend the specified prefix to all URL patterns. The prefix may include path variables. This is mainly used to nest resources within resources.

name_prefix

Prefix the specified string to all route names. This is most often combined with `path_prefix` to nest resources:

```
map.resource("message", "messages", controller="categories",
    path_prefix="/category/{category_id}",
    name_prefix="category_")
# GET /category/7/message/1
# Adds named route "category_message"
```

parent_resource

A dict containing information about the parent resource, for creating a nested resource. It should contain the member_name and collection_name of the parent resource. This dict will be available via the associated Route object which can be accessed during a request via `request.environ["routes.route"]`.

If parent_resource is supplied and path_prefix isn't, path_prefix will be generated from parent_resource as "<parent collection name>/:<parent member name>_id".

If parent_resource is supplied and name_prefix isn't, name_prefix will be generated from parent_resource as "<parent member name>_".

Example:

```
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'))
>>> # path_prefix is "regions/:region_id"
>>> # name prefix is "region_"
>>> url('region_locations', region_id=13)
'/regions/13/locations'
>>> url('region_new_location', region_id=13)
'/regions/13/locations/new'
>>> url('region_location', region_id=13, id=60)
'/regions/13/locations/60'
>>> url('region_edit_location', region_id=13, id=60)
'/regions/13/locations/60/edit'

Overriding generated path_prefix:

>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            path_prefix='areas/:area_id')
>>> # name prefix is "region_"
>>> url('region_locations', area_id=51)
'/areas/51/locations'

Overriding generated name_prefix:

>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            name_prefix='')
>>> # path_prefix is "regions/:region_id"
>>> url('locations', region_id=51)
'/regions/51/locations'
```

CHAPTER

FIVE

UNICODE, REDIRECTS, AND MORE

5.1 Unicode

Routes assumes UTF-8 encoding on incoming URLs, and `url` and `url_for` also generate UTF-8. You can change the encoding with the `map.charset` attribute:

```
map.charset = "latin-1"
```

New in Routes 1.10: several bugfixes.

5.2 Redirect Routes

Redirect routes allow you to specify redirects in the route map, similar to RewriteRule in an Apache configuration. This avoids the need to define dummy controller actions just to handle redirects. It's especially useful when the URL structure changes and you want to redirect legacy URLs to their new equivalents. The redirection is done by the Routes middleware, and the WSGI application is not called.

`map.redirect` takes two positional arguments: the route path and the destination URL. Redirect routes do not have a name. Both paths can contain variables, and the route path can take inline requirements. Keyword arguments are the same as `map.connect`, both in regards to extra variables and to route options.

```
map.redirect("/legacyapp/archives/{url:.*}", "/archives/{url}")

map.redirect("/legacyapp/archives/{url:.*}", "/archives/{url}")
```

By default a "302 Found" HTTP status is issued. You can override this with the `_redirect_code` keyword argument. The value must be an entire status string.

```
map.redirect("/home/index", "/", _redirect_code="301 Moved Permanently")
```

New in Routes 1.10.

5.3 Printing

Mappers now have a formatted string representation. In your python shell, simply print your application's mapper:

```
>>> map.collection("entries", "entry")
>>> print map
Route name   Methods Path
entries      GET     /entries{.format}
create_entry POST    /entries{.format}
new_entry    GET     /entries/new{.format}
entry        GET     /entries/{id}{.format}
update_entry PUT     /entries/{id}{.format}
delete_entry DELETE  /entries/{id}{.format}
edit_entry   GET     /entries/{id}/edit{.format}
```

New in Routes 1.12.

5.4 Introspection

The mapper attribute `.matchlist` contains the list of routes to be matched against incoming URLs. You can iterate this list to see what routes are defined. This can be useful when debugging route configurations.

5.5 Other

If your application is behind an HTTP proxy such a load balancer on another host, the WSGI environment will refer to the internal server rather than to the proxy, which will mess up generated URLs. Use the ProxyMiddleware in PasteDeploy to fix the WSGI environment to what it would have been without the proxy.

To debug routes, turn on debug logging for the "routes.middleware" logger. (See Python's `logging` module to set up your logging configuration.)

5.6 Backward compatibility

The following syntaxes are allowed for compatibility with previous versions of Routes. They may be removed in the future.

5.6.1 Omitting the name arg

In the tutorial we said that nameless routes can be defined by passing `None` as the first argument. You can also omit the first argument entirely:

```
map.connect(None, "/{controller}/{action}")
map.connect("/{controller}/{action}")
```

The syntax with `None` is preferred to be forward-compatible with future versions of Routes. It avoids the path argument changing position between the first and second arguments, which is unpythonic.

5.6.2 :varname

Path variables were defined in the format `:varname` and `:(varname)` prior to Routes 1.9. The form with parentheses was called "grouping", used to delimit the variable name from a following letter or number. Thus the old syntax "/:controller/:(id)abc" corresponds to the new syntax "/{controller}/{id}abc".

The older wildcard syntax is `*varname` or `*(varname)`:

```
# OK because the following component is static.
map.connect("/static/*filename/download")

# Deprecated syntax.  WRONG because the wildcard will eat the rest of the
# URL, leaving nothing for the following variable, which will cause the
# match to fail.
map.connect("/static/*filename/:action")
```

5.6.3 Minimization

Minimization was a misfeature which was intended to save typing, but which often resulted in the wrong route being chosen. Old applications that still depend on it must now enable it by putting `map.minimization = True` in their route definitions.

Without minimization, the URL must contain values for all path variables in the route:

```
map.connect("basic", "/{controller}/{action}",
    controller="mycontroller", action="myaction", weather="sunny")
```

This route matches any two-component URL, for instance "/help/about". The resulting routing variables would be:

```
{"controller": "help", "action": "about", "weather": "sunny"}
```

The path variables are taken from the URL, and any extra variables are added as constants. The extra variables for "controller" and "action" are *never used* in matching, but are available as default values for generation:

```
url("basic", controller="help") => "/help/about?weather=sunny"
```

With minimization, the same route path would also match shorter URLs such as "/help", "/foo", and "/". Missing values on the right of the URL would be taken from the extra variables. This was intended to lessen the number of routes you had to write. In practice it led to obscure application bugs because sometimes an unexpected route would be matched. Thus Routes 1.9 introduced non-minimization and recommended "map.minimization = False" for all new applications.

A corollary problem was generating the wrong route. Routes 1.9 tightened up the rule for generating named routes. If a route name is specified in `url()` or `url_for()`, *only* that named route will be chosen. In previous versions, it might choose another route based on the keyword args.

5.6.4 Implicit defaults and route memory

Implicit defaults worked with minimization to provide automatic default values for the "action" and "id" variables. If a route was defined as `map.connect("/{controller}/{action}/{id}")` and the URL `"/archives"` was requested, Routes would implicitly add `action="index", id=None` to the routing variables.

To enable implicit defaults, set `map.minimization = True; map.explicit = False`. You can also enable implicit defaults on a per-route basis by setting `map.explicit = True` and defining each route with a keyword argument `explicit=False`.

Previous versions also had implicit default values for "controller", "action", and "id". These are now disabled by default, but can be enabled via `map.explicit = True`. This also enables route memory

5.6.5 url_for()

`url_for` was a route generation function which was replaced by the `url` object. Usage is the same except that `url_for` uses route memory in some cases and `url` never does. Route memory is where variables from the current URL (the current request) are injected into the generated URL. To use route memory with `url`, call `url.current()` passing the variables you want to override. Any other variables needed by the route will be taken from the current routing variables.

In other words, `url_for` combines `url` and `url.current()` into one function. The location of `url_for` is also different. `url_for` is properly imported from `routes`:

```
from routes import url_for
```

`url_for` was traditionally imported into WebHelpers, and it's still used in some tests and in `webhelpers.paginate`. Many old Pylons applications contain `h.url_for()` based on its traditional importation to helpers.py. However, its use in new applications is discouraged both because of its ambiguous syntax and because its implementation depends on an ugly singleton.

The `url` object is created by the RoutesMiddleware and inserted into the WSGI environment. Pylons makes it available as `pylons.url`, and in templates as `url`.

5.6.6 redirect_to()

This combined `url_for` with a redirect. Instead, please use your framework's redirect mechanism with a `url` call. For instance in Pylons:

```
from pylons.controllers.util import redirect
redirect(url("login"))
```

CHAPTER

SIX

GLOSSARY

component A part of a URL delimited by slashes. The URL "/help/about" contains two components: "help" and "about".

generation The act of creating a URL based on a route name and/or variable values. This is the opposite of matching. Finding a route by name is called *named generation*. Finding a route without specifying a name is called *nameless generation*.

mapper A container for routes. There is normally one mapper per application, although nested subapplications might have their own mappers. A mapper knows how to match routes and generate them.

matching The act of matching a given URL against a list of routes, and returning the routing variables. See the *route* entry for an example.

minimization A deprecated feature which allowed short URLs to match long paths. Details are in the `Backward Compatibility` section in the manual.

route A rule mapping a URL pattern to a dict of routing variables. For instance, if the pattern is "/{controller}/{action}" and the requested URL is "/help/about", the resulting dict would be:

```
{"controller": "help", "action": "about"}
```

Routes does not know what these variables mean; it simply returns them to the application. Pylons would look for a `controllers/help.py` module containing a `HelpController` class, and call its `about` method. Other frameworks may do something different.

A route may have a name, used to identify the route.

route path The URL pattern in a route.

routing variables A dict of key-value pairs returned by matching. Variables defined in the route path are called *path variables*; their values will be taken from the URL. Variables defined outside the route path are called *default variables*; their values are not affected by the URL.

The WSGI.org environment key for routing variables is "wsgiorg.routing_args". This manual does not use that term because it can be confused with function arguments.

CHAPTER

SEVEN

PORTING ROUTES TO A WSGI WEB FRAMEWORK

7.1 RoutesMiddleware

An application can create a raw mapper object and call its `.match` and `.generate` methods. However, WSGI applications probably want to use the `RoutesMiddleware` as Pylons does:

```
# In myapp/config/middleware.py
from routes.middleware import RoutesMiddleware
app = RoutesMiddleware(app, map)      # ''map'' is a routes.Mapper.
```

The middleware matches the requested URL and sets the following WSGI variables:

```
environ['wsgiorg.routing_args'] = ((url, match))
environ['routes.route'] = route
environ['routes.url'] = url
```

where `match` is the routing variables dict, `route` is the matched route, and `url` is a `URLGenerator` object. In Pylons, `match` is used by the dispatcher, and `url` is accessible as `pylons.url`.

The middleware handles redirect routes itself, issuing the appropriate redirect. The application is not called in this case.

To debug routes, turn on debug logging for the "routes.middleware" logger.

See the Routes source code for other features which may have been added.

7.2 URL Resolution

When the URL is looked up, it should be matched against the Mapper. When matching an incoming URL, it is assumed that the URL path is the only string being matched. All query args should be stripped before matching:

```
m.connect('articles/{year}/{month}', controller='blog', action='view', year=None)

m.match('/articles/2003/10')
# {'controller':'blog', 'action':'view', 'year':'2003', 'month':'10'}
```

Matching a URL will return a dict of the match results, if you'd like to differentiate between where the argument came from you can use routematch which will return the Route object that has all these details:

```
m.connect('articles/{year}/{month}', controller='blog', action='view', year=None)

result = m.routematch('/articles/2003/10')
# result is a tuple of the match dict and the Route object

# result[0] - {'controller':'blog', 'action':'view', 'year':'2003', 'month':'10'}
# result[1] - Route object
# result[1].defaults - {'controller':'blog', 'action':'view', 'year':None}
# result[1].hardcoded - ['controller', 'action']
```

Your integration code is then expected to dispatch to a controller and action in the dict. How it does this is entirely up to the framework integrator. Your integration should also typically provide the web developer a mechanism to access the additional dict values.

7.3 Request Configuration

If you intend to support `url_for()` and `redirect_to()`, they depend on a singleton object which requires additional configuration. You're better off not supporting them at all because they will be deprecated soon. `URLGenerator` is the forward-compatible successor to `url_for()`. `redirect_to()` is better done in the web framework'as in `pylons.controllers.util.redirect_to()`.

`url_for()` and `redirect_to()` need information on the current request, and since they can be called from anywhere they don't have direct access to the WSGI environment. To remedy this, Routes provides a thread-safe singleton class called "request_config", which holds the request information for the current thread. You should update this after matching the incoming URL but before executing any code that might call the two functions. Here is an example:

```
from routes import request_config

config = request_config()

config.mapper = m                   # Your mapper object
config.mapper_dict = result         # The dict from m.match for this URL request
config.host = hostname              # The server hostname
config.protocol = port              # Protocol used, http, https, etc.
config.redirect = redir_func        # A redirect function used by your framework, that is
                                    # expected to take as the first non-keyword arg a single
                                    # full or relative URL
```

See the docstring for `request_config` in routes/__init__.py to make sure you've initialized everything necessary.

CHAPTER

EIGHT

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*
- *Glossary*

8.1 Module Listing

8.1.1 Routes Modules

`routes` – Routes Common Classes and Functions

Provides common classes and functions most users will want access to.

Module Contents

`request_config`(*original=False*)

Returns the Routes RequestConfig object.

To get the Routes RequestConfig:

```
>>> from routes import *
>>> config = request_config()
```

The following attributes must be set on the config object every request:

mapper mapper should be a Mapper instance thats ready for use

host host is the hostname of the webapp

protocol protocol is the protocol of the current request

mapper_dict mapper_dict should be the dict returned by mapper.match()

redirect redirect should be a function that issues a redirect, and takes a url as the sole argument

prefix (optional) Set if the application is moved under a URL prefix. Prefix will be stripped before matching, and prepended on generation

environ (optional) Set to the WSGI environ for automatic prefix support if the webapp is underneath a 'SCRIPT_NAME'

Setting the environ will use information in environ to try and populate the host/protocol/mapper_dict options if you've already set a mapper.

Using your own requst local

If you have your own request local object that you'd like to use instead of the default thread local provided by Routes, you can configure Routes to use it:

```
from routes import request_config()
config = request_config()
if hasattr(config, 'using_request_local'):
    config.request_local = YourLocalCallable
    config = request_config()
```

Once you have configured request_config, its advisable you retrieve it again to get the object you wanted. The variable you assign to request_local is assumed to be a callable that will get the local config object you wish.

This example tests for the presence of the 'using_request_local' attribute which will be present if you haven't assigned it yet. This way you can avoid repeat assignments of the request specific callable.

Should you want the original object, perhaps to change the callable its using or stop this behavior, call request_config(original=True).

class _RequestConfig()
RequestConfig thread-local singleton

The Routes RequestConfig object is a thread-local singleton that should be initialized by the web framework that is utilizing Routes.

load_wsgi_environ(*environ*)
Load the protocol/server info from the environ and store it. Also, match the incoming URL if there's already a mapper, and store the resulting match dict in mapper_dict.

routes.mapper – Mapper and Sub-Mapper

Mapper and Sub-Mapper

Module Contents

class SubMapperParent()
Base class for Mapper and SubMapper, both of which may be the parent of SubMapper objects

collection(*collection_name, resource_name, path_prefix=None, member_prefix='/{id}', controller=None, collection_actions=, ['index', 'create', 'new'], member_actions=, ['show', 'update', 'delete', 'edit'], member_options=None, **kwargs*)
Create a submapper that represents a collection.

This results in a `routes.mapper.SubMapper` object, with a `member` property of the same type that represents the collection's member resources.

Its interface is the same as the `submapper` together with `member_prefix`, `member_actions` and `member_options` which are passed to the `member` submatter as ``path_prefix``, `actions` and keyword arguments respectively.

Example:

```
>>> from routes.util import url_for
>>> map = Mapper(controller_scan=None)
>>> c = map.collection('entries', 'entry')
>>> c.member.link('ping', method='POST')
>>> url_for('entries') == '/entries'
True
>>> url_for('edit_entry', id=1) == '/entries/1/edit'
True
>>> url_for('ping_entry', id=1) == '/entries/1/ping'
True
```

submapper(***kargs*)

Create a partial version of the Mapper with the designated options set

This results in a `routes.mapper.SubMapper` object.

If keyword arguments provided to this method also exist in the keyword arguments provided to the submapper, their values will be merged with the saved options going first.

In addition to `routes.route.Route` arguments, submapper can also take a `path_prefix` argument which will be prepended to the path of all routes that are connected.

Example:

```
>>> map = Mapper(controller_scan=None)
>>> map.connect('home', '/', controller='home', action='splash')
>>> map.matchlist[0].name == 'home'
True
>>> m = map.submapper(controller='home')
>>> m.connect('index', '/index', action='index')
>>> map.matchlist[1].name == 'index'
True
>>> map.matchlist[1].defaults['controller'] == 'home'
True
```

Optional `collection_name` and `resource_name` arguments are used in the generation of route names by the `action` and `link` methods. These in turn are used by the `index`, `new`, `create`, `show`, `edit`, `update` and `delete` methods which may be invoked indirectly by listing them in the `actions` argument. If the `formatted` argument is set to `True` (the default), generated paths are given the suffix '{.format}' which matches or generates an optional format extension.

Example:

```
>>> from routes.util import url_for
>>> map = Mapper(controller_scan=None)
>>> m = map.submapper(path_prefix='/entries', collection_name='entries', resource_name='entr
>>> url_for('entries') == '/entries'
True
>>> url_for('new_entry', format='xml') == '/entries/new.xml'
True
```

class **SubMapper**(*obj*, *resource_name=None*, *collection_name=None*, *actions=None*, *formatted=None*, ***kwargs*)

Partial mapper for use with_options

action(*name=None*, *action=None*, *method='GET'*, *formatted=None*, ***kwargs*)

Generates a named route at the base path of a submapper.

Example:

```
>>> from routes import url_for
>>> map = Mapper(controller_scan=None)
>>> c = map.submapper(path_prefix='/entries', controller='entry')
>>> c.action(action='index', name='entries', formatted=True)
>>> c.action(action='create', method='POST')
>>> url_for(controller='entry', action='index', method='GET') == '/entries'
True
>>> url_for(controller='entry', action='index', method='GET', format='xml') == '/entries.xml
True
>>> url_for(controller='entry', action='create', method='POST') == '/entries'
True
```

add_actions(*actions*)

connect(**args*, ***kwargs*)

create(***kwargs*)
Generates the "create" action for a collection submapper.

delete(***kwargs*)
Generates the "delete" action for a collection member submapper.

edit(***kwargs*)
Generates the "edit" link for a collection member submapper.

index(*name=None*, ***kwargs*)
Generates the "index" action for a collection submapper.

link(*rel=None*, *name=None*, *action=None*, *method='GET'*, *formatted=None*, ***kwargs*)
Generates a named route for a subresource.

Example:

```
>>> from routes.util import url_for
>>> map = Mapper(controller_scan=None)
>>> c = map.collection('entries', 'entry')
>>> c.link('recent', name='recent_entries')
>>> c.member.link('ping', method='POST', formatted=True)
>>> url_for('entries') == '/entries'
True
>>> url_for('recent_entries') == '/entries/recent'
True
>>> url_for('ping_entry', id=1) == '/entries/1/ping'
True
>>> url_for('ping_entry', id=1, format='xml') == '/entries/1/ping.xml'
True
```

new(***kwargs*)
Generates the "new" link for a collection submapper.

show(*name=None*, ***kwargs*)
Generates the "show" action for a collection member submapper.

update(***kwargs*)
Generates the "update" action for a collection member submapper.

class Mapper(*controller_scan=<function controller_scan at 0x1055ec488>*, *directory=None*, *always_scan=False*, *register=True*, *explicit=True*)
Mapper handles URL generation and URL recognition in a web application.

Mapper is built handling dictionary's. It is assumed that the web application will handle the dictionary returned by URL recognition to dispatch appropriately.

URL generation is done by passing keyword parameters into the generate function, a URL is then returned.

Create a new Mapper instance

All keyword arguments are optional.

controller_scan Function reference that will be used to return a list of valid controllers used during URL matching. If `directory` keyword arg is present, it will be passed into the function during its call. This option defaults to a function that will scan a directory for controllers.

Alternatively, a list of controllers or None can be passed in which are assumed to be the definitive list of controller names valid when matching 'controller'.

directory Passed into controller_scan for the directory to scan. It should be an absolute path if using the default `controller_scan` function.

always_scan Whether or not the `controller_scan` function should be run during every URL match. This is typically a good idea during development so the server won't need to be restarted anytime a controller is added.

register Boolean used to determine if the Mapper should use `request_config` to register itself as the mapper. Since it's done on a thread-local basis, this is typically best used during testing though it won't hurt in other cases.

explicit Boolean used to determine if routes should be connected with implicit defaults of:

```
{'controller':'content','action':'index','id':None}
```

When set to True, these defaults will not be added to route connections and `url_for` will not use Route memory.

Additional attributes that may be set after mapper initialization (ie, map.ATTRIBUTE = 'something'):

encoding Used to indicate alternative encoding/decoding systems to use with both incoming URL's, and during Route generation when passed a Unicode string. Defaults to 'utf-8'.

decode_errors How to handle errors in the encoding, generally ignoring any chars that don't convert should be sufficient. Defaults to 'ignore'.

minimization Boolean used to indicate whether or not Routes should minimize URL's and the generated URL's, or require every part where it appears in the path. Defaults to True.

hardcode_names Whether or not Named Routes result in the default options for the route being used *or* if they actually force url generation to use the route. Defaults to False.

connect(**args*, ***kargs*)

Create and connect a new Route to the Mapper.

Usage:

```
m = Mapper()
m.connect(':controller/:action/:id')
m.connect('date/:year/:month/:day', controller="blog", action="view")
m.connect('archives/:page', controller="blog", action="by_page",
requirements = { 'page':'\d{1,2}' })
m.connect('category_list', 'archives/category/:section', controller='blog', action='category
section='home', type='list')
m.connect('home', '', controller='blog', action='view', section='home')
```

create_regs(**args, **kwargs*)
Atomically creates regular expressions for all connected routes

extend(*routes, path_prefix=''*)
Extends the mapper routes with a list of Route objects

If a path_prefix is provided, all the routes will have their path prepended with the path_prefix.

Example:

```
>>> map = Mapper(controller_scan=None)
>>> map.connect('home', '/', controller='home', action='splash')
>>> map.matchlist[0].name == 'home'
True
>>> routes = [Route('index', '/index.htm', controller='home',
...                 action='index')]
>>> map.extend(routes)
>>> len(map.matchlist) == 2
True
>>> map.extend(routes, path_prefix='/subapp')
>>> len(map.matchlist) == 3
True
>>> map.matchlist[2].routepath == '/subapp/index.htm'
True
```

Note: This function does not merely extend the mapper with the given list of routes, it actually creates new routes with identical calling arguments.

generate(**args, **kargs*)
Generate a route from a set of keywords

Returns the url text, or None if no URL could be generated.

```
m.generate(controller='content',action='view',id=10)
```

match(*url=None, environ=None*)
Match a URL against against one of the routes contained.

Will return None if no valid match is found.

```
resultdict = m.match('/joe/sixpack')
```

redirect(*match_path, destination_path, *args, **kwargs*)
Add a redirect route to the mapper

Redirect routes bypass the wrapped WSGI application and instead result in a redirect being issued by the RoutesMiddleware. As such, this method is only meaningful when using RoutesMiddleware.

By default, a 302 Found status code is used, this can be changed by providing a `_redirect_code` keyword argument which will then be used instead. Note that the entire status code string needs to be present.

When using keyword arguments, all arguments that apply to matching will be used for the match, while generation specific options will be used during generation. Thus all options normally available to connected Routes may be used with redirect routes as well.

Example:

```
map = Mapper()
map.redirect('/legacyapp/archives/{url:.*}, '/archives/{url})
map.redirect('/home/index', '/', _redirect_code='301 Moved Permanently')
```

resource(*member_name, collection_name, **kwargs*)
Generate routes for a controller resource

The member_name name should be the appropriate singular version of the resource given your locale and used with members of the collection. The collection_name name will be used to refer to the resource collection methods and should be a plural version of the member_name argument. By default, the member_name name will also be assumed to map to a controller you create.

The concept of a web resource maps somewhat directly to 'CRUD' operations. The overlying things to keep in mind is that mapping a resource is about handling creating, viewing, and editing that resource.

All keyword arguments are optional.

controller If specified in the keyword args, the controller will be the actual controller used, but the rest of the naming conventions used for the route names and URL paths are unchanged.

collection Additional action mappings used to manipulate/view the entire set of resources provided by the controller.

Example:

```
map.resource('message', 'messages', collection={'rss':'GET'})
# GET /message/rss (maps to the rss action)
# also adds named route "rss_message"
```

member Additional action mappings used to access an individual 'member' of this controllers resources.

Example:

```
map.resource('message', 'messages', member={'mark':'POST'})
# POST /message/1/mark (maps to the mark action)
# also adds named route "mark_message"
```

new Action mappings that involve dealing with a new member in the controller resources.

Example:

```
map.resource('message', 'messages', new={'preview':'POST'})
# POST /message/new/preview (maps to the preview action)
# also adds a url named "preview_new_message"
```

path_prefix Prepends the URL path for the Route with the path_prefix given. This is most useful for cases where you want to mix resources or relations between resources.

name_prefix Perpends the route names that are generated with the name_prefix given. Combined with the path_prefix option, it's easy to generate route names and paths that represent resources that are in relations.

Example:

```
map.resource('message', 'messages', controller='categories',
    path_prefix='/category/:category_id',
    name_prefix="category_")
# GET /category/7/message/1
# has named route "category_message"
```

parent_resource A `dict` containing information about the parent resource, for creating a nested resource. It should contain the `member_name` and `collection_name` of the parent resource. This `dict` will be available via the associated `Route` object which can be accessed during a request via `request.environ['routes.route']`

If `parent_resource` is supplied and `path_prefix` isn't, `path_prefix` will be generated from `parent_resource` as "<parent collection name>/:<parent member name>_id".

If `parent_resource` is supplied and `name_prefix` isn't, `name_prefix` will be generated from `parent_resource` as "<parent member name>_".

Example:

```
>>> from routes.util import url_for
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'))
>>> # path_prefix is "regions/:region_id"
>>> # name prefix is "region_"
>>> url_for('region_locations', region_id=13)
'/regions/13/locations'
>>> url_for('region_new_location', region_id=13)
'/regions/13/locations/new'
>>> url_for('region_location', region_id=13, id=60)
'/regions/13/locations/60'
>>> url_for('region_edit_location', region_id=13, id=60)
'/regions/13/locations/60/edit'
```

Overriding generated `path_prefix`:

```
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            path_prefix='areas/:area_id')
>>> # name prefix is "region_"
>>> url_for('region_locations', area_id=51)
'/areas/51/locations'
```

Overriding generated `name_prefix`:

```
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            name_prefix='')
>>> # path_prefix is "regions/:region_id"
>>> url_for('locations', region_id=51)
'/regions/51/locations'
```

routematch(*url=None, environ=None*)
Match a URL against against one of the routes contained.

Will return None if no valid match is found, otherwise a result dict and a route object is returned.

```
resultdict, route_obj = m.match('/joe/sixpack')
```

routes.route – Route

Module Contents

class Route(*name, routepath, **kargs*)
The Route object holds a route recognition and generation routine.

See Route.__init__ docs for usage.

Initialize a route, with a given routepath for matching/generation

The set of keyword args will be used as defaults.

Usage:

```
>>> from routes.base import Route
>>> newroute = Route(None, ':controller/:action/:id')
>>> sorted(newroute.defaults.items())
[('action', 'index'), ('id', None)]
>>> newroute = Route(None, 'date/:year/:month/:day',
...     controller="blog", action="view")
>>> newroute = Route(None, 'archives/:page', controller="blog",
...     action="by_page", requirements = { 'page':'\d{1,2}' })
>>> newroute.reqs
{'page': '\\d{1,2}'}
```

Note: Route is generally not called directly, a Mapper instance connect method should be used to add routes.

buildfullreg(*clist, include_names=True*)
Build the regexp by iterating through the routelist and replacing dicts with the appropriate regexp match

buildnextreg(*path, clist, include_names=True*)
Recursively build our regexp given a path, and a controller list.

Returns the regular expression string, and two booleans that can be ignored as they're only used internally by buildnextreg.

generate(*_ignore_req_list=False, _append_slash=False, **kargs*)
Generate a URL from ourself given a set of keyword arguments

Toss an exception if this set of keywords would cause a gap in the url.

generate_minimized(*kargs*)
Generate a minimized version of the URL

generate_non_minimized(*kargs*)
Generate a non-minimal version of the URL

make_full_route()
Make a full routelist string for use with non-minimized generation

make_unicode(*s*)
Transform the given argument into a unicode string.

makeregexp(*clist*, *include_names=True*)
Create a regular expression for matching purposes

Note: This MUST be called before match can function properly.

clist should be a list of valid controller strings that can be matched, for this reason makeregexp should be called by the web framework after it knows all available controllers that can be utilized.

include_names indicates whether this should be a match regexp assigned to itself using regexp grouping names, or if names should be excluded for use in a single larger regexp to determine if any routes match

match(*url*, *environ=None*, *sub_domains=False*, *sub_domains_ignore=None*, *domain_match=''*)
Match a url to our regexp.

While the regexp might match, this operation isn't guaranteed as there's other factors that can cause a match to fail even though the regexp succeeds (Default that was relied on wasn't given, requirement regexp doesn't pass, etc.).

Therefore the calling function shouldn't assume this will return a valid dict, the other possible return is False if a match doesn't work out.

routes.middleware – Routes WSGI Middleware

Routes WSGI Middleware

Module Contents

class **RoutesMiddleware**(*wsgi_app*, *mapper*, *use_method_override=True*, *path_info=True*, *singleton=True*)
Routing middleware that handles resolving the PATH_INFO in addition to optionally recognizing method overriding.

Create a Route middleware object

Using the use_method_override keyword will require Paste to be installed, and your application should use Paste's WSGIRequest object as it will properly handle POST issues with wsgi.input should Routes check it.

If path_info is True, then should a route var contain path_info, the SCRIPT_NAME and PATH_INFO will be altered accordingly. This should be used with routes like:

```
map.connect('blog/*path_info', controller='blog', path_info='')
```

is_form_post(*environ*)
Determine whether the request is a POSTed html form

routes.lru – LRU caching class and decorator

LRU caching class and decorator

Module Contents

class LRUCache (*size*)
Implements a psueudo-LRU algorithm (CLOCK)

routes.util – URL Generator and utility functions

Utility functions for use in templates / controllers

PLEASE NOTE: Many of these functions expect an initialized RequestConfig object. This is expected to have been initialized for EACH REQUEST by the web framework.

Module Contents

exception RoutesException
Tossed during Route exceptions

exception MatchException
Tossed during URL matching exceptions

exception GenerationException
Tossed during URL generation exceptions

class URLGenerator (*mapper, environ*)
The URL Generator generates URL's

It is automatically instantiated by the RoutesMiddleware and put into the `wsgiorg.routing_args` tuple accessible as:

```
url = environ['wsgiorg.routing_args'][0][0]
```

Or via the `routes.url` key:

```
url = environ['routes.url']
```

The url object may be instantiated outside of a web context for use in testing, however sub_domain support and fully qualified URL's cannot be generated without supplying a dict that must contain the key `HTTP_HOST`.

Instantiate the URLGenerator

mapper The mapper object to use when generating routes.

environ The environment dict used in WSGI, alternately, any dict that contains at least an `HTTP_HOST` value.

current (**args, **kwargs*)
Generate a route that includes params used on the current request

The arguments for this method are identical to `__call__` except that arguments set to None will remove existing route matches of the same name from the set of arguments used to construct a URL.

url_for (**args, **kargs*)
Generates a URL

All keys given to url_for are sent to the Routes Mapper instance for generation except for:

```
anchor          specified the anchor name to be appened to the path
host            overrides the default (current) host if provided
protocol        overrides the default (current) protocol if provided
qualified       creates the URL with the host/port information as
                needed
```

The URL is generated based on the rest of the keys. When generating a new URL, values will be used from the current request's parameters (if present). The following rules are used to determine when and how to keep the current requests parameters:

- If the controller is present and begins with '/', no defaults are used
- If the controller is changed, action is set to 'index' unless otherwise specified

For example, if the current request yielded a dict of {'controller': 'blog', 'action': 'view', 'id': 2}, with the standard ':controller/:action/:id' route, you'd get the following results:

```
url_for(id=4)                    =>  '/blog/view/4',
url_for(controller='/admin')     =>  '/admin',
url_for(controller='admin')      =>  '/admin/view/2'
url_for(action='edit')           =>  '/blog/edit/2',
url_for(action='list', id=None)  =>  '/blog/list'
```

Static and Named Routes

If there is a string present as the first argument, a lookup is done against the named routes table to see if there's any matching routes. The keyword defaults used with static routes will be sent in as GET query arg's if a route matches.

If no route by that name is found, the string is assumed to be a raw URL. Should the raw URL begin with / then appropriate SCRIPT_NAME data will be added if present, otherwise the string will be used as the url with keyword args becoming GET query args.

_url_quote(*string, encoding*)
A Unicode handling version of urllib.quote.

_str_encode(*string, encoding*)

_screenargs(*kargs, mapper, environ, force_explicit=False*)
Private function that takes a dict, and screens it against the current request dict to determine what the dict should look like that is used. This is responsible for the requests "memory" of the current.

_subdomain_check(*kargs, mapper, environ*)
Screen the kargs for a subdomain and alter it appropriately depending on the current subdomain or lack therof.