

The Ranvier URL mapper

Letting URL structure invoke application work

Skill Level: Intermediate

David Q. Mertz, Ph.D. (mertz@gnosis.cx)

Uniform writer

Gnosis Software

29 Jan 2008

Ranvier is a Python package you can integrate into Web application frameworks to map incoming URL requests to source code. It does this by a mechanism of delegation-and-consumption, which differs from more common regular expression-based URL rewriting. Ranvier also serves as a central registry of all the URLs in a Web application and can itself generate the URLs necessary for cross-linking pages. The registry function allows Ranvier to assure the integrity of links and automate coverage analysis. Ranvier is pure Python code and does not have any third-party dependencies; it should be usable (with a bit of adaptor code) in any Python-based Web application framework.

Prolegomenon

The responsibility of a Uniform Resource Identifier (URI) is to uniquely name a resource in the world. The most familiar subset of URIs is Uniform Resource Locators (URLs), which take on the additional responsibility of providing a description of how to *obtain* the named resource (in other words, a network location and protocol to use in fetching an electronic document or stream). Some URIs are Uniform Resource Names (URNs) without being URLs, that is, they name a resource, but do not provide specific details on how to obtain it.

There is some subtlety to what a URI means, however. Context—especially the time and place at which a resource is obtained—often changes the content of the described resource. For example, the URI (and URL), `file:///etc/hosts`, describes a different content for each reader who might examine that resource on

his or her local machine. Further, if you imagine a Web-based service that incorporates geolocation capabilities, it might utilize a URL like `http://weather.example.com/today/local/` to indicate that a requester wishes to receive content describing the weather for the place and time at which he or she makes the request. You might also utilize a non-URL URN to describe the same content, such as `urn:weather:today:local`; it would presumably be the responsibility of a processing application to determine what protocol or mechanism to use in obtaining or generating the named content.

In principle, a URI might have any form that conforms with RFC 3986. For example, it could utilize a name following the Universally Unique Identifier (UUID) subset defined in RFC 4122. Thus encoded, you might name "today's local weather" as: `urn:uuid:4930fce0-2eda-4f6b-9eaa-fe426fefc655`. Or if a URL form was required, you might use, for example, `tftp://192.0.2.143/'eaecf224-6efc-3499-8cd8-3e50a4f58ec1`. The obvious problem with these latter URIs is that they really are not very memorable or descriptive for humans. (Machines are more indifferent to the descriptive issue.) In practice, developers and users tend to want URIs to embed at least a modicum of semantic information, not only a syntactic form.

Hierarchy and components

The oldest and most popular Web servers have generated URIs whose form directly mirrors the file system structure of the machine's hosting resources. The URI schema itself specifies a hierarchical "path" component of URIs (though a path is potentially empty), but does not *require* any literal mapping between a URI path structure and a file system. Nonetheless, it is commonplace for Web servers to simply take relative directory names as the basis of URIs. This system makes a great deal of sense for organizing (relatively) static documents, where the same organizational principles apply for creating hierarchies of local and network-accessible resources (the original purposes of the WWW, to a first approximation).

In dynamic Web application frameworks, URI path hierarchies are generally mapped out of the structure of the code that serves pages. So, for example, path components in the URIs served by, for example, Rails or Django indicate the relative nesting of classes and methods, often with the addition of a few framework-specific naming conventions for class components.

One thing file system maps and code hierarchy paths have in common is that the URIs they provide to resource consumers emerge out of technical decisions made in configuring a Web server. The code decisions show through in the presentation of URIs. Sometimes this transparency is appropriate and desirable: Under one ideal often associated with OOP, for example, the structure of code objects directly mirrors the semantics of the application domain. However, in other cases the

hierarchies involved in creation of a Web service are quite different from the "natural" hierarchies of the domain itself. In those cases, resource consumers must settle for URLs that merely name the resource they wish to obtain without also *describing* the resource. Or at least these URLs do not describe their resources as well as might be possible if you had started from a consumer's point-of-view rather than a developer's.

Regular expression substitutions

The most common tools for mapping semantically connotative URIs to internally structured ones simply perform replacements based on regular expressions. Apache's `mod_rewrite` is the most widely known example of this approach. Routes for Rails also works similarly. For this approach you must have an internal set of URIs—perhaps ones whose form follows your technical decisions in creating a site—and also an idea of what makes up meaningful URIs. Under regular expression URL rewriting, every rewritten URI has at least one denotative twin. That is, the "public" URIs are mapped by the Web server onto "private" ones. However, the private ones are not necessarily—nor even usually—protected from public access; it is just that the public version of URIs are those normally published in publicizing your resources.

A simple and common example of URL rewriting is in mapping path-based resource descriptions onto FastCGI (or just CGI) calls. That is, a path may not exist on the literal file system of a Web server; in order to simulate a "real" path, a rewrite rule can delegate to a script that returns a "virtual" path (virtual relative to the server file system, that is—every URI path component is morally equal as a resource component). An example in *The Django Book* shows how to let Apache return any file that exists in a file system path, but delegate where it does not exist. The CGI-based demo of Ranvier on its Web page uses a similar rule, as do I on my test system:

Apache `mod_rewrite`: Delegate path if no file exists

```
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
```

How Ranvier stands apart

Instead of using regular expressions to rewrite public URIs onto private ones, Ranvier implements a chain-of-responsibility design pattern to delegate resource determination to relevant code. The goal here is largely to enable meaningful code reuse: Each path component causes dispatch to a relevant code handler. In turn, a handler consumes zero or more path components, performs any required processing, and may modify a context object that carries processing state. Since the

same handler might be passed through while processing various URIs, any common responsibility is easy to code in a single handler.

An example helps illustrate this chain. In the demo accompanying Ranvier, an example for processing usernames within a path is given. For example, the URI `http://furius.ca/ranvier/demo/users/mertz/name` returns a description of my "user account" (even though my name was not part of the design of the demo). All this example does is capitalize my last name and return a message about its action—but it well illustrates how a URI might be used to fetch an arbitrary user's account information from a database, or otherwise perform some computation as part of the URI unpacking.

At the heart of a Ranvier application is a `create_application()` function that defines a `root` object. The `root` object—often and in the demo—is an instance of the `Folder` class, initialized with a collection of keyword arguments pointing to URI path-nested components. Some code makes this more clear. The below is an abridged version of what you can find in `demoapp.py` in the Ranvier distribution:

Defining a root URI tree in Ranvier

```
root = Folder(
    _default='home',
    home=Home(),
    resources=EnumResource(mapper),
    users=UsernameRoot(Folder(username=PrintUsername(),
                              name=PrintName(),
                              data=UserData())),
)
```

The relative URI `/home` is defined by a class, `Home(LeafResource)`, defined in `demoapp.py`. In short summary, this is simply a way of writing an expanded template to the requestor. Part of what gets included in this template are computed URIs like `mapurl("@@Home")`. As Ranvier sorts things out for you, the URI provided this way is just `http://mysite.com/home/` itself. Other URIs are computed in a like manner. The class `EnumResources` is a Ranvier built-in, and becomes accessible in the above definition at `/resources`.

Of greater interest is the definition of the path component `/users`. This is defined by the class `UsernameRoot(VarDelegatorResource)`. Take a look at its code:

A VarDelegatorResource handler class

```
class UsernameRoot(VarDelegatorResource):
    def __init__(self, next, **kwargs):
        VarDelegatorResource.__init__(self, 'username', next, **kwargs)

    def handle(self, ctxt):
        if not re.match('[a-z]+$', ctxt.username):
            return ctxt.response.errorNotFound()
```

```
ctxt.name = 'Mr or Mrs %s' % ctxt.username.capitalize()
```

The attribute `.username` is assigned to the context object in the `.__init__()` method of this class, which consumes the next path component as well. The attribute `ctxt.username` is utilized in the `.handle()` method of this class, mostly to calculate another context attribute `ctxt.name` (a trivial computation here, but imagine a database access instead).

Notice that the work of `UsernameRoot` is largely to contain another `Folder`. The nested folder contains path components `username`, `name`, and `data`. This enables provision of URI paths like `/users/mertz/username`, `/users/blais/data/keyname`, and `/users/smith/name`. Take a look in turn at the class `PrintName`, which is delegated to after the `/users/username` components are utilized by `UsernameRoot`:

A LeafResource handler class

```
class PrintName(LeafResource):
    def handle(self, ctxt):
        ctxt.page.render_header(ctxt)
        ctxt.response.write(''<p>The user\'s name is %(name)s.</p>'' %
                           {'name': ctxt.name})
        ctxt.page.render_footer(ctxt)
```

The only notable thing `PrintName` does is utilize the context attribute `ctxt.name`. Presumably, if integrated with a full Web application framework, this value might be used in a general template language rather than in the simplistic template used in the demo.

This brief excerpt from the demo shows only a part of what `Ranvier` can do. Only passingly mentioned is any discussion of mapping backwards from handlers to URIs.

Conceptual issues

While I like the elegance of code written by my colleague Martin Blais, creator of `Ranvier`, I find there is a certain conceptual gap in the overall design of `Ranvier` mapped URIs. The problem is specifically in a mismatch between the RFC 3986 form of URIs and the more "popular" attachment to the path component of URIs.

Specifically, let me repeat an example given in the `Ranvier` documentation. Martin mentions a "naive" URI similar to:

```
http://example.com/get_balance?user=blais&acc=64
```

This URI might be implemented by a function like:

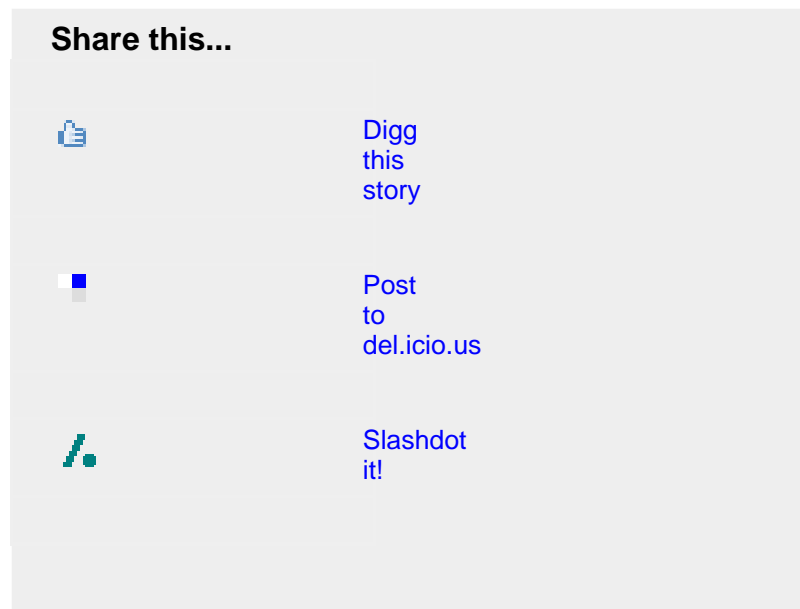
```
def get_balance(user, accno):  
    "Lookup balance based on user and account number"
```

Using Ranvier, you might easily create a different form of this URI in which all the path components might serve as arguments, such as:

http://example.com/user/blais/acc/64/get_balance/

Martin comments in the documentation of the alternative URI:

This tends to produce nicer URLs and bends the user into conceptually organizing the resources he makes available on his server in a hierarchy. Query arguments are best reserved for optional arguments.



What the transformation of URI form does, however, in my mind, is *not* a conceptual separation of optional and mandatory arguments. After all, a handler in Ranvier is perfectly well capable of consuming a variable number of (optional) path components, too. The real distinction between path components and query parameters is that between hierarchical versus unordered arguments. For the most part, this type of URI mapping by Ranvier simply lets non-hierarchical arguments be embedded into a hierarchical path. All this really does is express a syntax preference for slashes over ampersands, equals signs, and question marks. A stylistic desire for certain characters is fairly neutral, until you realize that RFC 3986 has already reserved the desired hierarchical/unordered distinction in the path versus query URI parts. Why shoehorn one specified part into the other; the mere

presence of popular misunderstanding of the distinction seems a poor reason to me.

That criticism stated, there are certainly many cases where domain resources are semantically hierarchical, and not merely in a way that mirrors peculiarities of the development framework and tools used in implementation. Ranvier provides a flexible way of organizing dispatch of functional aspects of URI processing into multiple reusable blocks of code.

Resources

Learn

- The home page for [Ranvier](#) is your number one resource for all things Ranvier.
- The IETF [formally defines the generic syntax for URIs](#), as well as gives general guidance to their usage and meaning, in RFC 3986. Meanwhile, RFC 4122 defines [a Uniform Resource Name namespace for UUIDs](#).
- Wikipedia has [a nice general introduction to URL rewriting](#). At the time of this writing, it focuses on server-level tools rather than application-level ones.
- Apache's `mod_rewrite` module provides sophisticated, regular expression-based rewriting of URL requests. See [the module reference](#) for more information.
- A gentler introduction is the Apache [URL Rewriting Guide](#).
- Ruby on Rails includes [a native URL rewriting engine called Routes](#): I use a `mod_rewrite` example taken from [Chapter 21 of *The Django Book*](#) (copyright 2006 Adrian Holovaty and Jacob Kaplan-Moss).
- David Mertz has written [a widely read tutorial on regular expressions](#). You might want to take a look at it if you are using URL rewriting through Apache's `mod_rewrite`, or a similar regular expression URL rewriting engine.
- Find more articles from the [Web development zone technical library](#).
- [Subscribe](#) to the developerWorks Web development newsletter.

Get products and technologies

- Download [IBM product evaluation versions](#).

About the author

David Q. Mertz, Ph.D.

David Mertz's life is a text, constantly undergoing revision; he hopes to resolve his resources one day. David may be reached at mertz@gnosis.cx; his life pored over at <http://gnosis.cx/publish/>. Check out David's book *Text Processing in Python* (<http://gnosis.cx/TPiP/>).

Trademarks

