Morepath Documentation

Release 0.18

Morepath developers

Mar 17, 2017

Contents

Part I

Getting Started

If you are new to Morepath, you'll find here a few resources that can help you get up to speed right away.

CHAPTER 1

Morepath: Super Powered Python Web Framework

Morepath is a Python web microframework, with super powers.

Morepath is a Python WSGI microframework. It uses routing, but the routing is to models. Morepath is model-driven and **flexible**, which makes it **expressive**.

- Morepath does not get in your way.
- It lets you express what you want with ease. See *Quickstart*.
- It's extensible, with a simple, coherent and universal extension and override mechanism, supporting reusable code. See *App Reuse*.
- It understands about generating hyperlinks. The web is about hyperlinks and Morepath actually *knows* about them. See *Paths and Linking*.
- Views are simple functions. All views are generic. See *Views*.
- It has all the tools to develop REST web services in the box. See *REST*.
- Documentation is important. Morepath has a lot of *Documentation*.

Sounds interesting?

Walk the Morepath with us!

Video intro

Here is a 25 minute introduction to Morepath, originally given at EuroPython 2014:

Morepath Super Powers

- *Automatic hyperlinks that don't break.*
- *Creating generic UIs is as easy as subclassing.*

- *Simple, flexible, powerful permissions.*
- *Reuse views in views.*
- *Extensible apps. Nestable apps. Override apps, even override Morepath itself!*
- *Extensible framework. Morepath itself can be extended, and your extensions behave exactly like core extensions.*

Curious how Morepath compares with other Python web frameworks? See *Comparison with other Web Frameworks*.

Morepath Knows About Your Models

```
import morepath

class App(morepath.App):
    pass

class Document(object):
    def __init__(self, id):
        self.id = id

@App.path(path='')
class Root(object):
    pass

@App.path(path='documents/{id}', model=Document)
def get_document(id):
    return Document(id)  # query for doc

@App.html(model=Root)
def hello_root(self, request):
    return '<a href="%s">Go to doc</a>' % request.link(Document('foo'))

@App.html(model=Document)
def hello_doc(self, request):
    return '<p>Hello document: %s!</p>' % self.id

if __name__ == '__main__':
    morepath.run(App())
```

Want to know what's going on? Check out the *Quickstart*!

More documentation, please!

- *Read the documentation*

If you have questions, please join the #morepath IRC channel on freenode. Hope to see you there!

I just want to try it!

- Get started using the Morepath cookiecutter template

You will have your own application to fiddle with in no time!

CHAPTER 2

Quickstart

Morepath is a micro-framework, and this makes it small and easy to learn. This quickstart guide should help you get started. We assume you've already installed Morepath; if not, see the *Installation* section.

Hello world

Let's look at a minimal "Hello world!" application in Morepath:

```
import morepath


class App(morepath.App):
    pass


@App.path(path='')
class Root(object):
    pass


@App.view(model=Root)
def hello_world(self, request):
    return "Hello world!"


if __name__ == '__main__':
    morepath.run(App())
```

You can save this as `hello.py` and then run it with Python:

```
$ python hello.py
Running <__main__.App object at 0x10f8398d0>
Listening on http://127.0.0.1:5000
Press Ctrl-C to stop...
```

Making the server externally accessible

The default configuration of *morepath.run()* uses the `127.0.0.1` hostname. This means you can access the web server from your own computer, but not from anywhere else. During development this is often the best way to go about things.

But sometimes do want to make the development server accessible from the outside world. This can be done by passing an explicit `host` argument of `0.0.0.0` to the `morepath.run()` function.

```
morepath.run(App(), host='0.0.0.0')
```

Alternatively, you can specify `0.0.0.0` on the command line:

```
$ python hello.py --host 0.0.0.0
```

Note that the built-in web server is absolutely unsuitable for actual deployment. For those cases don't use `morepath.run()` at all, but instead use an external WSGI server such as waitress, Apache mod_wsgi or nginx mod_wsgi.

If you now go with a web browser to the URL given, you should see "Hello world!" as expected. When you want to stop the server, just press control-C.

Morepath uses port 5000 by default, and it might be the case that another service is already listening on that port. If that happens you can specify a different port on the command line:

```
$ python hello.py --port 6000
```

This application is a bit bigger than you might be used to in other web micro-frameworks. That's for a reason: Morepath is not geared to create the most succinct "Hello world!" application but to be effective for building slightly larger applications, all the way up to huge ones.

Let's go through the hello world app step by step to gain a better understanding.

Code Walkthrough

1. We import `morepath`.
2. We create a subclass of *morepath.App* named `App`. This class contains our application's configuration: what models and views are available. It can also be instantiated into a WSGI application object.
3. We then set up a `Root` class. Morepath is model-driven and in order to create any views, we first need at least one model, in this case the empty `Root` class.

   We set up the model as the root of the website (the empty string `''` indicates the root, but `'/'` works too) using the *morepath.App.path()* decorator.
4. Now we can create the "Hello world" view. It's just a function that takes `self` and `request` as arguments (we don't need to use either in this case), and returns the string `"Hello world!"`. The `self` argument is the instance of the `model` class that is being viewed.

   We then need to hook up this view with the *morepath.App.view()* decorator. We say it's associated with the `Root` model. Since we supply no explicit `name` to the decorator, the function is the default view for the `Root` model on `/`.

5. The `if __name__ == '__main__'` section is a way in Python to make the code only run if the `hello.py` module is started directly with Python as discussed above. In a real-world application you instead use a setuptools entry point so that a startup script for your application is created automatically.
6. We then instantiate the `App` class to create a `WSGI` app using the default web server. Since you create a WSGI app you can also plug it into any other WSGI server.

This example presents a compact way to organize your code in a single module, but for a real project we recommend you read *Organizing your Project*. This supports organizing your project with multiple modules.

Routing

Morepath uses a special routing technique that is different from many other routing frameworks you may be familiar with. Morepath does not route to views, but routes to models instead.

Why route to models?

Why does Morepath route to models? It allows for some nice features. The most concrete feature is automatic hyperlink generation - we'll go into more detail about this later.

A more abstract feature is that Morepath through model-driven design allows for greater code reuse: this is the basis for Morepath's super-powers. We'll show a few of these special things you can do with Morepath later.

Finally Morepath's model-oriented nature makes it a more natural fit for REST applications. This is useful when you need to create a web service or the foundation to a rich client-side application.

Models

A model is any Python object that represents the content of your application: say a document, or a user, an address, and so on. A model may be a plain in-memory Python object or be backed by a database using an ORM such as SQLAlchemy, or some NoSQL database such as the ZODB. This is entirely up to you; Morepath does not put special requirements on models.

Above we've exposed a `Root` model to the root route `/`, which is rather boring. To make things more interesting, let's imagine we have an application to manage users. Here's our `User` class:

```
class User(object):
    def __init__(self, username, fullname, email):
        self.username = username
        self.fullname = fullname
        self.email = email
```

We also create a simple users database:

```
users = {}
def add_user(user):
    users[user.username] = user

faassen = User('faassen', 'Martijn Faassen', 'faassen@startifact.com')
bob = User('bob', 'Bob Bobsled', 'bob@example.com')
add_user(faassen)
add_user(bob)
```

Publishing models

Custom variables function

The default behavior is for Morepath to retrieve the variables by name using `getattr` from the model objects. This only works if those variables exist on the model under that name. If not, you can supply a custom `variables` function that given the model returns a dictionary with all the variables in it. Here's how:

```
@App.path(model=User, path='/users/{username}',
          variables=lambda model: dict(username=model.username))
def get_user(username):
    return users.get(username)
```

Of course this `variables` is not necessary as it has the same behavior as the default, but you can do whatever you want in the variables function in order to get the username.

Getting `variables` right is important for link generation.

We want our application to have URLs that look like this:

```
/users/faassen

/users/bob
```

Here's the code to expose our users database to such a URL:

```
@App.path(model=User, path='/users/{username}')
def get_user(username):
    return users.get(username)
```

The `get_user` function gets a user model from the users database by using the dictionary `get` method. If the user doesn't exist, it returns `None`. We could've fitted a SQLAlchemy query in here instead.

Now let's look at the decorator. The `model` argument has the class of the model that we're putting on the web. The `path` argument has the URL path under which it should appear.

The path can have variables in it which are between curly braces (`{` and `}`). These variables become arguments to the function being decorated. Any arguments the function has that are not in the path are interpreted as URL parameters.

What if the user doesn't exist? We want the end-user to see a 404 error. Morepath does this automatically for you when you return `None` for a model, which is what `get_user` does when the model cannot be found.

Now we've published the model to the web but we can't view it yet.

converters

A common use case is for path variables to be a database id. These are often integers only. If a non-integer is seen in the path we know it doesn't match. You can specify a path variable contains an integer using the integer converter. For instance:

```
@App.path(model=Post, path='posts/{post_id}', converters=dict(post_id=int))
def get_post(post_id):
    return query_post(post_id)
```

You can do this more succinctly too by using a default parameter for `post_id` that is an int, for instance:

```
@App.path(model=Post, path='posts/{post_id}')
def get_post(post_id=0):
    return query_post(post_id)
```

For more on this, see *Paths and Linking*.

Views

In order to actually see a web page for a user model, we need to create a view for it:

```
@App.view(model=User)
def user_info(self, request):
    return "User's full name is: %s" % self.fullname
```

The view is a function decorated by *morepath.App.view()* (or related decorators such as *morepath.App.json()* and *morepath.App.html()*) that gets two arguments: `self`, which is the model that this view is working for, so in this case an instance of `User`, and `request` which is the current request. `request` is a `morepath.request.Request` object (a subclass of `webob.request.BaseRequest`).

Now the URLs listed above such as `/users/faassen` will work.

What if we want to provide an alternative view for the user, such as an `edit` view which allows us to edit it? We need to give it a name:

```
@App.view(model=User, name='edit')
def edit_user(self, request):
    return "An editing UI goes here"
```

Now we have functionality on URLs like `/users/faassen/edit` and `/users/bob/edit`.

For more on this, see *Views*.

Linking to models

Morepath is great at creating links to models: it can do it for you automatically. Previously we've defined an instance of `User` called `bob`. What now if we want to link to the default view of `bob`? We simply do this:

```
>>> request.link(bob)
'http://example.com/users/bob'
```

What if we want to see Bob's edit view? We do this:

```
>>> request.link(bob, 'edit')
'http://example.com/users/bob/edit'
```

Using *morepath.Request.link()* everywhere for link generation is easy. You only need models and remember which view names are available, that's it. If you ever have to change the path of your model, you won't need to adjust any linking code.

For more on this, see *Paths and Linking*.

Link generation compared

If you're familiar with routing frameworks where links are generated to views (such as Flask or Django) link generation is more involved. You need to give each route a name, and then refer back to this route name when you want to generate a link. You also need to supply the variables that go into the route. With Morepath, you don't need

a route name, and if the default way of getting variables from a model is not correct, you only need to explain once how to create the variables for a route, with the `variables` argument to `@App.path`.

In addition, Morepath links are completely generic: you can pass in anything linkable. This means that writing a generic view that uses links becomes easier – there is no dependency on particular named URL paths anymore.

JSON and HTML views

`@App.view` is rather bare-bones. You usually know more about what you want to return than that. If you want to return JSON, you can use the shortcut `@App.json` instead to declare your view:

```
@App.json(model=User, name='info')
def user_json_info(self, request):
    return {'username': self.username,
            'fullname': self.fullname,
            'email': self.email}
```

This automatically serializes what is returned from the function JSON, and sets the content-type header to `application/json`.

If we want to return HTML, we can use `@App.html`:

```
@App.html(model=User)
def user_info(self, request):
    return "<p>User's full name is: %s</p>" % self.fullname
```

This automatically sets the content type to `text/html`. It doesn't do any HTML escaping though, so the use of `%` above is unsafe! We recommend the use of a HTML template language in that case.

Request object

The first argument for a view function is the request object. We'll give a quick overview of what's possible here, but consult the WebOb API documentation for more information.

- `request.GET` contains any URL parameters (`?key=value`). See `webob.request.BaseRequest.GET`.
- `request.POST` contains any HTTP form data that was submitted. See `webob.request.BaseRequest.POST`.
- `request.method` gets the HTTP method (`GET`, `POST`, etc). See `webob.request.BaseRequest.method`.
- `request.cookies` contains the cookies. See `webob.request.BaseRequest.cookies`. `response.set_cookie` can be used to set cookies. See `webob.response.Response.set_cookie()`.

Redirects

To redirect to another URL, use *`morepath.redirect()`*. For example:

```
@App.view(model=User, name='extra')
def redirecting(self, request):
    return morepath.redirect(request.link(self, 'other'))
```

HTTP Errors

To trigger an HTTP error response you can raise various WebOb HTTP exceptions (`webob.exc`). For instance:

```
from webob.exc import HTTPNotAcceptable

@App.view(model=User, name='extra')
def erroring(self, request):
    raise HTTPNotAcceptable()
```

But note that Morepath already raises a lot of these errors for you automatically just by having your structure your code the Morepath way.

CHAPTER 3

Community

Github

Morepath is maintained as a Github project:

https://github.com/morepath/morepath

Feel free to fork it and make pull requests!

We use the Github issue tracker for discussion about bugs and new features:

https://github.com/morepath/morepath/issues

So please report issues there. Feel free to add new issues!

Chat

Want to chat with us? Join us! This uses Discord, with web-based, desktop and mobile clients available.

Mailing list/forum

There's a mailing list/web forum for discussing Morepath. Discussion about use and development of Morepath are both welcome:

https://groups.google.com/forum/#!forum/morepath

Feel free to speak up. Questions are very welcome!

CHAPTER 4

Examples

Sometimes the best way to learn about how something works is to look at an example. The Morepath Morepath organization on GitHub maintains the following example projects:

morepath_batching

Example of a batching UI using server-side templates. Shows how explicit models and link generation makes it easier to implement a batching UI.

morepath_cerebral_todomvc

A React & Cerebral rich frontend using Morepath as a REST backend.

morepath_reactredux

A React & Redux rich frontend using Morepath as REST backend.

morepath_rest_dump_load

A demonstration on how to use the `json_dump` and `json_load` directives to help implement a REST service.

morepath_sqlalchemy

Use SQLAlchemy with Morepath. This uses `more.transaction` to help integrate the two.

morepath_static

Using `more.static` with Morepath to publish static resources such as `.js` and `.css` files.

morepath_wiki

A wiki demo for Morepath, based on the web micro-framework battle by Richard Jones.

CHAPTER 5

Installation

Quick and Dirty Installation

To get started with Morepath right away, first create a Python 3.5 virtualenv:

```
$ virtualenv morepath_env
$ source morepath_env/bin/activate
```

Now install Morepath into it:

```
$ pip install morepath
```

You can now use the virtual env's Python to run any code that uses Morepath:

```
$ python quickstart.py
```

See *Quickstart* for information on how to get started with Morepath itself, including an example of `quickstart.py`.

Creating a Morepath Project Using Cookiecutter

Morepath provides an official cookiecutter template. Cookiecutter is a tool that creates projects through project templates. Morepath's template comes with a very simple application, either in RESTful or traditional HTML flavor.

Follow the instructions on Morepath's cookiecutter template repository to get started:

https://github.com/morepath/morepath-cookiecutter

This is a great way to get started with Morepath as a beginner or to start a new project as a seasoned Morepath user.

Creating a Morepath Project Manually

When you develop a web application it's a good idea to use standard Python project organization practices. *Organizing your Project* has some recommendations on how to do this with Morepath. Relevant in particular is the contents of `setup.py`, which depends on Morepath and also sets up an entry point to start the web server.

Once you have a project you can use tools like pip. We'll briefly describe how to it.

pip

With pip and a virtualenv called `morepath_env`, you can do this in your project's root directory:

```
$ pip install --editable .
```

You can now run the application like this (if you called the console script `myproject-start`):

```
$ myproject-start
```

Depending on Morepath development versions

If you like being on the cutting edge and want to depend on the latest Morepath and Reg development versions always, you can install these using pip (in a virtualenv). Here's how:

```
$ pip install git+git://github.com/morepath/reg.git@master

$ pip install git+git://github.com/morepath/morepath.git@master
```

A more involved method how to install Morepath for development is described in *Developing Morepath*.

CHAPTER 6

Superpowers

We said Morepath has super powers. Are they hard to use, then? No: they're both powerful and also easy to use, which makes them even more super!

Link with Ease

Since Morepath knows about your models, it can generate links to them. If you have a model instance (for example through a database query), you can get a link to it by calling *morepath.Request.link()*:

```
request.link(my_obj)
```

Want a link to its edit view (or whatever named view you want)? Just do:

```
request.link(my_obj, 'edit')
```

If you create links this way everywhere (and why shouldn't you?), you know your application's links will never break.

For much more, see *Paths and Linking*.

Generic UI

Morepath knows about model inheritance. It lets you define views for a base class that automatically become available for all subclasses. This is a powerful mechanism to let you write generic UIs.

For example, if we have this generic base class:

```
class ContainerBase(object):
    def entries(self):
        """All entries in the container returned as a list."""
```

We can easily define a generic default view that works for all subclasses:

```
@App.view(model=ContainerBase)
def overview(self, request):
    return ', '.join([entry.title for entry in self.entries()])
```

But what if you want to do something different for a particular subclass? What if `MySpecialContainer` needs it own custom default view? Easy:

```
@App.view(model=MySpecialContainer)
def special_overview(self, request):
    return "A special overview!"
```

Morepath leverages the power of the flexible Reg generic function library to accomplish this.

For much more, see *Views*.

Model-driven Permissions

Morepath features a very flexible but easy to use permission system. Let's say we have an `Edit` permission; it's just a class:

```
class Edit(object):
    pass
```

And we have a view for some `Document` class that we only want to be accessible if the user has an edit permission:

```
@App.view(model=Document, permission=Edit)
def edit_document(self, request):
    return "Editable"
```

How does Morepath know whether someone has `Edit` permission? We need to tell it using the *morepath.App.permission_rule()* directive. We can implement any rule we want, for instance this one:

```
@App.permission_rule(model=Document, permission=Edit)
def have_edit_permission(identity, model, permission):
    return model.has_permission(identity.userid)
```

Instead of a specific rule that only works for `Document`, we can also give our app a broad rule (use `model=object`).

Composable Views

Let's say you have a JSON view for a `Document` class:

```
@App.json(model=Document)
def document_json(self, request):
    return {'title': self.title}
```

And now we have a view for a container that contains documents. We want to automatically render the JSON views of the documents in a list. We can write this:

```
@App.json(model=DocumentContainer)
def document_container_json(self, request):
    return [document_json(doc, request) for doc in self.entries()]
```

Here we've used `document_json` ourselves. But what now if the container does not only contain `Document` instances? What if one of them is a `SpecialDocument`? Our `document_container_json` function breaks. How to fix it? Easy, we can use *morepath.Request.view()*:

```
@App.json(model=DocumentContainer)
def document_container_json(self, request):
    return [request.view(doc) for doc in self.entries()]
```

Now `document_container_json` works for anything in the container model that has a default view!

Extensible Applications

Somebody else has written an application with Morepath. It contains lots of stuff that does exactly what you want, and one view that *doesn't* do what you want:

```
@App.view(model=Document)
def recalcitrant_view(self, request):
    return "The wrong thing!"
```

Ugh! We can't just change the application as it needs to continue to work in its original form. Besides, it's being maintained by someone else. What do we do now? Monkey-patch? Not at all: Morepath got you covered. You simply create a new application subclass that extends the original:

```
class MyApp(App):
    pass
```

We now have an application that does exactly what `app` does. Now to override that one view to do what we want:

```
@MyApp.view(model=Document)
def whatwewant(self, request):
    return "The right thing!"
```

And we're done!

It's not just the view directive that works this way: *all* Morepath directives work this way.

Morepath also lets you mount one application within another, allowing composition-based reuse. See *App Reuse* for more information. Using these techniques you can build large applications, see *Building Large Applications*.

Extensible Framework

Morepath's directives are implemented using Dectate, the meta-framework for configuring Python frameworks. You can define new directives and registries for Morepath with ease:

```
class Extended(morepath.App):
    pass

@Extended.directive('widget')
class WidgetAction(dectate.Action):
    config = {
        'widget_registry': dict  # use dict as a registry
    }
    def __init__(self, name):
        self.name = name
```

```
    def identifier(self):
        return self.name

    def perform(self, obj, widget_registry):
        widget_registry[self.name] = obj

@Extended.widget('input')
def input_widget():
    ...

@Extended.widget('label')
def label_widget():
    ...
```

CHAPTER 7

Comparison with other Web Frameworks

We hear you ask:

> There are a *million* Python web frameworks out there. How does Morepath compare?

Pyramid Design Choices

This document is a bit like the Design Defense Document of the Pyramid web framework. The Pyramid document makes for a very interesting read if you're interested in web framework design. More web frameworks should do that.

If you're already familiar with another web framework, it's useful to learn how Morepath is the same and how it is different, as that helps you understand it more quickly. So we try to do this a little here.

Our ability to compare Morepath to other web frameworks is limited by our familiarity with them, and also by their aforementioned large quantity. But we'll try. Feel free to pitch in new comparisons, or tell us where we get it wrong!

You may also want to read the *Design Notes* document.

Overview

Morepath aims to be foundational and flexible and is by itself relatively low-level. All web applications are different. Some are simple. Some, like CMSes, are like frameworks themselves. Morepath makes it easy to build other frameworky things on top of Morepath.

Morepath isn't there to be hidden away under another framework – Morepath extensions still look like Morepath, which makes them consistent and easier to approach. This orientation towards being foundational makes Morepath more like Pyramid, or perhaps Flask, than like Django.

Morepath aims to have a small core. It isn't full stack; it's a microframework. It should be easy to pick up. This makes it similar to other microframeworks like Flask or CherryPy, but different from Django and Zope, which offer a lot of features.

Morepath is opinionated. There is only one way to do routing and one way to do configuration. This makes it like a lot of web frameworks, but unlike Pyramid, which takes more of a toolkit approach where a lot of choices are made available.

Morepath is a routing framework, but it's model-centric. Models, that is, any Python objects, have URLs. This makes it like a URL traversal framework like Zope or Grok, and also like Pyramid when traversal is in use. Awareness of models allows Morepath automate linking, generate correct HTTP status codes automatically and lets it have its powerful permission-based security. It makes it unlike other routing frameworks like Django or Flask, which have less awareness of models.

Paradoxically enough one thing Morepath is opinionated about is *flexibility*, as that's part of its mission to be a good foundation. That's what its configuration system (Dectate) and generic function system (Reg) are all about. Want to change behavior? You can override everything. You can introduce new registries and new directives. Even core behavior of Morepath can be changed by overriding its generic functions. This makes Morepath like Zope, and especially like Pyramid, but less like Django or Flask.

Routing

Collect 200 dollars

Do not directly go to the view. Go to the model first. Only *then* go to the view. Do collect 200 dollars. Don't go to jail.

Morepath is a *routing* web framework, like Django and Flask and a lot of others. This is a common way to use Pyramid too (the other is traversal). This is also called URL mapping or dispatching. Morepath is to our knowledge, unique in that the routes don't directly go to *views*, but go through *models* first.

Morepath's route syntax is very similar to Pyramid's, i.e. `/hello/{name}`. Flask is also similar. It's unlike Django's regular expressions. Morepath works at a higher level than that deliberately, as that makes it possible to disambiguate similar routes.

This separation of model and view lookup helps in the following ways:

- Automated HTTP status codes in case things go wrong – no more easy to forget custom error message generation code in all the views.
- Model-based security checks – you can define rules that say exactly what kind of objects get which permissions, and then protect views with those permissions.
- better code organization in application code, as it allows you to separate the code that organizes the URL space from the code that implements your actual views.
- Automated linking.

Linking

Because it routes to models, Morepath allows you to ask for the URL of a model instance, like this:

```
request.link(mymodel)
```

That is an easier and less brittle way to make links than having to name your routes explicitly. Morepath pushes link generation quite far: it can construct links with paths and URL parameters automatically.

Morepath shares the property of model-based links with traversal based web frameworks like Zope and Grok, and also Pyramid in non-routing traversal mode. Uniquely among them Morepath *does* route, not traverse.

For more: *Paths and Linking*.

Permissions

Morepath has a permission framework built-in: it knows about authentication and lets you plug in authenticators, you can protect views with permissions and plug in code that tells Morepath what permissions someone has for which models. It's small but powerful in what it lets you do.

This is unlike most other micro-frameworks like Flask, Bottle, CherryPy or web.py. It's like Zope, Grok and Pyramid, and has learned from them, though Morepath's system is more streamlined.

For more you can check out *Security*.

View lookup

Morepath uses a separate view lookup system. The name of the view is determined from the last step of the path being routed to. With this URL path for instance:

```
/document/edit
```

the `/edit` bit indicates the name of the view to look up for the document model.

If no view step is supplied, the default view is looked up:

```
/document
```

This is like modern Zope works, and like how the Plone CMS works. It's also like Grok. It's like Pyramid if it's used with traversal instead of routing. Overall there's a strong Zope heritage going on, as all these systems are derived from Zope in one way or another. Morepath is unique in that it combines *routing* with view lookup.

This decoupling of views from models helps with expressivity, as it lets you write reusable, generic views, and code organisation as mentioned before.

For more: *Views*.

WSGI

Morepath is a WSGI-based framework, like Flask or Pyramid, and these days Django as well.

A Morepath app is a standard WSGI app. You can plug it into a WSGI compliant web server like Apache or Nginx or gunicorn.

Explicit request

Some frameworks, like Flask and Bottle, have a magic `request` global that you can import. But request isn't really a global, it's a variable, and in Morepath it's a variable that's passed into view functions explicitly. This makes Morepath more similar to Pyramid or Django.

Testability and Global state

Developers that care about writing testable code try to avoid global state, in particular mutable global state, as it can make testing harder. If the framework is required to be in a certain global state before the code under test can be run, it becomes harder to test that code, as you need to know first what global state to manipulate.

Globals can also be a problem when multiple threads try to write the global at the same time. Web frameworks avoid this by using *thread locals*. Confusingly enough these locals are *globals*, but they're isolated from other threads.

Morepath does not require any global state. Of course Morepath's app *are* module globals, but they're not *used* that way once Morepath's configuration is loaded and Morepath starts to handle requests. Morepath's framework code passes the app along as a variable (or attribute of a variable, such as the request) just like everything else.

Morepath is built on the Reg generic function library. Previously Reg had some optional implicit global state, but as of release 0.10 this has been eliminated – state is entirely explicit here as well.

Flask is quite happy to use global state (with thread locals) to have a request that you can import. Pyramid is generally careful to avoid global state, but does allow using thread local state to get access to the current registry in some cases.

No default database

Morepath has no default database integration. This is like Flask and Bottle and Pyramid, but unlike Zope or Django, which have assumptions about the database baked in (ZODB and Django ORM respectively).

You can plug in your own database, or even have no database at all. You could use SQLAlchemy, or the ZODB. Morepath lets you treat anything as models. We have examples and extensions that help you integrate specific databases. Here's morepath_sqlalchemy

Pluggable template languages

Some micro-frameworks like Flask and Bottle and web.py have template languages built-in, some, like CherryPy and the Werkzeug toolkit, don't. Pyramid doesn't have built-in support either, but has standard plugins for the Chameleon, Jijna2 and Mako template languages.

Morepath allows you to plug in server templates. You can plug in Jinja2 through more.jinja2, Chameleon through more.chameleon and Mako through more.mako.

You don't have to use a server template language though: Morepath aims to be a good fit for modern, client-side web applications written in JavaScript. We've made it easy to send anything to the client, especially JSON. If templating is used for such applications, it's done on the client, in the web browser, not on the server.

Code configuration

Most Python web frameworks don't have an explicit code configuration system. With "code configuration" I mean expressing things like "this function handles this route", "this view works for this model", and "this is the authentication system for this app". It also includes extension and overrides, such as "here is an additional route", "use this function to handle this route instead of what the core said".

If a web framework doesn't deal with code configuration explicitly, an implicit code configuration system tends to grow. There is one way to set up routes, another way to declare models, another way to do generic views, yet another way to configure the permission system, and so on. Each system works differently and uses a different API. Config files, metaclasses and import-time side effects may all be involved.

On top of this, if the framework wants to allow reuse, extension and overrides the APIs tends to grow even more distinct with specialised use cases, or yet more new APIs are grown.

Django is an example where configuration gained lots of knobs and buttons; another example is Zope 2.

Microframeworks aim for simplicity so don't suffer from this so much, though probably at the cost of some flexibility. You can still observe this kind of evolution in Flask's pluggable views subsystem, though, for instance.

To deal with this problem in an explicit way the Zope project pioneered a component configuration mechanism. By having a universal mechanism in which code is configured, the configuration API becomes general and allows extension and override in a general manner as well. Zope uses XML files for this.

The Grok project tried to put a friendlier face on the rather verbose configuration system of Zope. Pyramid refined Grok's approach further. It offers a range of options for configuration: explicit calls in Python code, decorators, and an extension that uses Zope-style XML.

In order to do its decorator based configuration, the Pyramid project created the Venusian python library. This is in turn a reimagined version of the Martian python library created by the Grok project. Venusian was used by the Morepath project originally, and even though it is gone it still helped inspire Morepath's configuration system.

Morepath uses a new, general configuration system called Dectate that is based around decorators attached to application objects. These application objects can extend other ones. Dectate supports a range sophisticated extension and override use cases in a general way.

Components and Generic functions

The Zope project made the term "zope component architecture" (ZCA) (in)famous in the Python world. Does it sound impressive, suggesting flexibility and reusability? Or does it sound scary, overengineered, `RequestProcessorFactoryFactory`-like? Are you intimidated by it? We can't blame you.

At its core the ZCA is really a system to add functionality to objects from the outside, without having to change their classes. It helps when you need to build extensible applications and reusable generic functionality. Under the hood, it's just a fancy registry that knows about inheritance. Its a really powerful system to help build more complicated applications and frameworks. It's used by Zope, Grok and Pyramid.

Morepath uses something else: a library called Reg. This is a new, reimagined, streamlined implementation of the idea of the ZCA.

The underlying registration APIs of the ZCA is rather involved, with quite a few special cases. Reg has a much simpler, more general registration API that is flexible enough to fulfill a range of use cases.

Finally what makes the Zope component architecture rather involved to use is its reliance on *interfaces*. An interface is a special kind of object introduced by the Zope component architecture that is used to describe the API of objects. It's like an abstract base class.

If you want to look up things in a ZCA component registry the ZCA requires you to look up an interface. This requires you to *write* interfaces for everything you want to be able to look up. The interface-based way to do lookups also looks rather odd to the average Python developer: it's not considered to be very Pythonic. To mitigate the last problem Pyramid creates simple function-based APIs on top of the underlying interfaces.

Morepath by using Reg does away with interfaces altogether – instead it uses generic functions. The simple function-based APIs *are* what is pluggable; there is no need to deal with interfaces anymore, but the system retains the power. Morepath is simple functions all the way down.

A fancy term you could use for this approach is post object-oriented design.

CHAPTER 8

A Review of the Web

Morepath is a web framework. Here is a quick review of how the web works, how applications can be built with it, and how Morepath fits.

HTTP protocol

HTTP is a protocol by which clients (such as web browsers) and servers can communicate. The client sends a HTTP request, and the server sends back a HTTP response. HTTP is extensible, and can be extended with content types, new headers, and so on.

Version 1.1 of HTTP is most common on the web today. It is defined by a bunch of specifications:

- RFC7230 - HTTP/1.1: Message Syntax and Routing
- RFC7231 - HTTP/1.1: Semantics and Content
- RFC7232 - HTTP/1.1: Conditional Requests
- RFC7233 - HTTP/1.1: Range Requests
- RFC7234 - HTTP/1.1: Caching
- RFC7235 - HTTP/1.1: Authentication

Luckily it's not necessary to understand the full details of these specifications to develop a web application. We'll go into a basic overview of relevant concepts in this document.

Morepath handles the HTTP protocol on the server side: creating a response to incoming HTTP requests.

Web browser

A web browser such as Firefox, Chrome and Internet Explorer uses the HTTP protocol to talk to web servers.

A web browser is a type of *HTTP client*.

Web server

A web server implements the HTTP protocol to respond to requests from HTTP clients such as web browsers.

There are general web servers such as Apache and Nginx. These are programmable in various ways.

There are also more specific web servers that are geared at particular tasks. Examples of these are Waitress and Gunicorn which are geared towards serving web applications written in Python.

A web server is programmable in various ways. Morepath can plug into web servers that implement the Python WSGI protocol.

Web application

A web application is software that presents a user interface by means of a web browser. The web browser is usually a visible piece of software, but may also be embedded in other software, such as in FirefoxOS.

A web application is loaded from a web server. After it is loaded it can still interact with the web server (or other web servers). The web server can implement part of the application logic and maintains the application data.

The dynamic behavior of a web application used to be implemented almost entirely by the server, but it is now also possible to implement a large part of their behavior within the web browser instead, using the JavaScript language.

Morepath code runs entirely on the server, but supports web applications that want to implement a large part of their dynamic behavior within the web browser.

Web service

A web service does not present a user interface to the user. A web service instead presents an application programming interface (API) to custom HTTP client software. The API is to this software what the UI is to the user.

You can layer a full web application on top of a web service. Such layering can result in looser coupling in the implementation, which tends to increase the quality of the implementation.

Morepath helps developers to implement web services.

Custom HTTP client

A web browser is one form of HTTP client, but other HTTP software can be written in a variety of languages to talk to a web server programmatically. This uses it as a web service.

JavaScript code in a web browser can also use the browser's facilities to talk to the web server programmatically (a technique called AJAX), and can thus serve as a custom HTTP client as well.

Framework

A library is reusable code that your code calls, whereas a framework is reusable code that calls your code. "Don't call us, we'll call you".

A framework aims to help you do particular tasks quickly; you only need to fill in the details, and the framework handles the rest.

There is a gray area between library and framework. Morepath is mostly a framework.

Server web framework

A framework that helps you program the behavior of a web server. Morepath is a server web framework written in the Python programming language.

JavaScript

JavaScript is a programming language that is run in the browser. It can use the web browser APIs (such as the DOM) to manipulate the web page, get user input, or access the server programmatically (AJAX).

JavaScript can also be run on the server with Node.JS, but Morepath is a Python web framework and does not make use of server-side JavaScript.

Bower is a tool to help manage client-side JavaScript code.

Bower

A popular way to install client-side JavaScript (and CSS) code is to use the Bower package management tool. By using a package manager installing and updating a collection of JavaScript libraries becomes more easy than doing it by hand.

Morepath offers Bower integration, see: *Static resources with Morepath*.

AJAX

AJAX is a technique to access resources programmatically from a browser application in JavaScript. These resources typically have a JSON representation.

Client web framework

There are also client-side web frameworks that let you program the behavior of a web browser, typically called "JavaScript MVC framework". Examples of such are React, Ember and Angular.

Morepath supports client-side code that uses a client web framework, but does not implement a client web framework itself. You can pick whichever you want.

WSGI

WSGI is a Python protocol by which Python code can be integrated with a web server. WSGI can also be used to implement framework components which are layered between application code and server.

A `morepath.App` instance implements the WSGI protocol and can therefore be integrated with a WSGI-compliant web server and WSGI framework component.

HTTP request

A HTTP request is a message a HTTP client sends to the server. The server then returns a HTTP response.

The HTTP request contains a *URL path*, a *request method*, possibly a *request body*, and various *headers* such as the *content type*.

A HTTP request in Morepath is made accessible programmatically as a Python request object using the WebOb library. It is a *morepath.Request*, which is a subclass of webob.request.BaseRequest.

HTTP response

A HTTP response returns a representation of the resource indicated by the path of the request as the *response body*. The response has a *content type* which determines what representation is being sent. The response also has a *status code* that indicates whether the request could be handled, or the reason why a detailed response could not be generated.

A lot of different representations exist. HTML is a very common one, but for programmatic clients JSON is typically used.

Morepath lets you create a *morepath.Response* object directly, which is a subclass of webob.response.Response, and return it from a view function.

More conveniently you use a specialized view type (*morepath.App.json()* or *morepath.App.html()*) and return the content that should go into the response body, such as a HTML string or a JSON-serializable object. Morepath then automatically creates the response with the right content type for you. Should you wish to set additional information on the response object, you can use *morepath.Request.after()*.

Resource

A resource is anything that can be addressed on the web by a URL (or URI or IRI). Can be a web page presenting a full UI (using HTML + CSS), or can be a piece of information (typically in JSON), or can also be an abstract entity that has no representation at all.

Morepath lets you implement resources of all kinds. Normally Morepath resources have representations, but it is also possible to implement abstract entities that have just a URL and have no representation. Morepath can also help you create links to resources on other web servers.

URL

Here is an example of a URL:

```
http://example.com/documents/3
```

A HTTP client such as a web browser uses URLs to determine:

- What protocol to use to talk to the server (in this case `http`).
- What *host* to talk to (in this case `example.com`). This identifies the web server, though a complex host may be implemented using a combination of web servers.
- What *path* to request from the server (in this case `/documents/3`).

The server determines how it responds to requests for particular paths.

URL parameters

A URL can have additional parameters:

```
http://example.com/documents/3?expand=1&highlight=foo
```

The list of parameters start with `?`. Names are connected with values using =, and name/value pairs are connected with `&`.

Path

A path is a way for a client to address a particular resource on a server. It is part of the request. The path is also part of URLs, and thus can be used for linking resources.

Morepath connects paths with Python objects using the path directive (*morepath.App.path()*): it can resolve a path to a Python object, and construct a path for a given Python object. This is described in *Paths and Linking*.

Example:

```
@App.path(path='/documents/{id}', model=Document)
def get_document(id):
    return query_document(id)
```

If you declare arguments for `get_document` that do not get listed as variables in the `path` these are interpreted as expected URL parameters.

Link generation

Morepath makes it easy to generate a hyperlink to a Python object. Morepath uses information on the object itself and its class to determine what link to generate.

Given the `path` directive above, we can generate a link to an instance of `Document` using *morepath.Request.link()*:

```
some_document = get_some_document_from_somewhere()
request.link(some_document)
```

This makes it easy to create links within Morepath view functions.

Morepath's link generation can generate links that include URL parameters.

Headers

A HTTP request and a HTTP response have headers. Headers contain information about the message that are not the body: they are about the request or the response, or about the body. For example, the content-type is header named `Content-Type` and has a value that is a MIME type such as `text/html`.

Headers are used for a wide variety of purposes, such as to declare information about how a client may cache a response, or what kind of responses a client accepts from a server, or to pass cookies along. Here is an overview of common headers.

In Morepath, the headers are accessible on a request and response object as the attribute `webob.request.BaseRequest.headers` and `webob.response.Response.headers`. which behaves like a Python dictionary. You could therefore access the request content-type using `request.headers['Content-Type']`. But see below for a more convenient way to access the content type.

To set the headers (or other information) on a response, you can create a `morepath.Response` instance in a view function. You can then pass in the headers, or set them afterward.

Often better is to use the *`morepath.Request.after()`* decorator to declare a function that sets headers the response object once it has been created for you by the framework.

WebOb has APIs that help you deal with many headers at a higher level of abstraction. For example, `webob.request.BaseRequest.content_type` is a more convenient way to access the content type information of a request than to access the header directly, as additional charset information is not there. Before you start to manipulate headers directly it pays off to consult the WebOb documentation for `webob.request.BaseRequest` and `webob.response.Response`: there may well be a better way.

Morepath also has special support for dealing with certain headers. For instance, the Forwarded header can be set by a HTTP proxy. To make Morepath use this header for URL generation, you can use the more.forwarded extension.

Cookies

One special set of headers deals with HTTP cookies. A server can set a cookie on the client by passing back a special header in its response. A cookie is much like a key/value pair in a Python dictionary.

Once the cookie has been set, the client sends back the cookie to the server during each subsequent request, again using a header, until the cookie expires or cookie is explicitly deleted by the server using a response header.

Normally in HTTP requests are independent from each other: assuming the server database is the same, the same request should give the same response, no matter what other requests have gone before it. This makes it easier to reason about HTTP, and it makes it easier to scale it up, for instance by caching responses.

Cookies change this: they can be used to make requests order-dependent. This can be useful, but it can also make it harder to reason about what is going on and scale, so be careful with them. In particular, a REST web service should be able to function without requiring the client to maintain cookies.

Cookies are commonly used to store login session information on the client.

WebOb makes management of cookies more convenient: the `webob.request.BaseRequest.cookies` attribute on the request object contains the list of cookies sent by the client, and the response object has an API incuding `webob.response.Response.set_cookie()` and `webob.response.Response.delete_cookie()` to allow you to manage cookies.

Content types

A resource may present itself in variety of representations. This is indicated by the content type set in the HTTP response, using the `Content-Type` header. There are a lot of content types, including HTML and JSON. The value is a MIME type such as `text/html` for HTML and `application/json` for JSON. The value can also contain additional parameters such as character encoding information.

WebOb makes content-type header information conveniently available with the `webob.request.BaseRequest.content_type`, `webob.response.Response.content_type` and `webob.response.Response.content_type_params` attributes.

A request may also have a content type: the request content type determines what kind of content is sent to the server by the client in the request body.

While you can create any kind of content type with Morepath, it has special support for generating HTML and JSON responses (using *morepath.App.html()* and *morepath.App.json()*), and for processing a JSON request body (see `load function for views` in *JSON and validation*).

View

In Morepath, a view is a Python function that takes a Python object to represent (`self`) and a *morepath.Request* object (`request`) as arguments and returns something that can be turned into a HTTP response, or a HTTP response object directly.

Here is an example of a Morepath view, using the most basic *morepath.App.view()* directive:

```
@App.view(model=MyObject)
def my_object_default(self, request):
    return "some text content"
```

There are also specific *morepath.App.json()* and *morepath.App.html()* directives to support those content types.

See *Views* for much more on how to construct Morepath views.

HTTP request method

A HTTP request has a *method*, also known as *HTTP verb*. The `GET` method is used to retrieve information from the server. The `POST` method is used to add new information to the server (for instance a form submit), and the `PUT` method is used to update existing information. The `DELETE` method is used to delete information from the server.

It is up to the server implementation how to exactly handle the request method. With Morepath, by default a view responds to the `GET` method, but you can also write views to handle the other HTTP methods, by indicating it with a *view predicate*. Here is a view that handles the `POST` method (and returns a representation of what has just been POSTed):

```
@App.view(model=MyCollection, request_method='POST')
def add_to_collection(self, request):
    item = MyItem(request.json)
    self.add(item)
    return request.view(item)
```

You can access the method on the request using `webob.request.BaseRequest.method`, but typically Morepath does this for you when you use the `request_method` predicate.

View predicate

A *view predicate* in Morepath is used to match a view function with details of `self` and `request`.

This view directive:

```
@App.view(model=MyCollection, request_method='POST')
def add_to_collection(self, request):
    ...
```

only matches when `self` is an instance of `MyCollection` (`model` predicate) and when `request.method` is `POST` (`request_method` predicate). Only in this case will `add_to_collection` be called.

You can extend Morepath with additional view predicates. You can also define a *predicate fallback*, which can be used to specify what HTTP status code to set when the view cannot be matched.

See view predicates

HTTP status codes

HTTP status codes such as `200 Ok` and `404 Not Found` are part of the HTTP response. Here is a list of HTTP status codes. The server can use them to indicate to the client whether it was successfully able to create a response, or if not, what the problem was.

Morepath can automatically generate the correct HTTP status codes for you in many cases:

200 Ok: When the path in the request is matched with a path directive, and there is a view for the particular model and request method.

404 Not Found: When the path does not match, or when the path matches but the path function returns `None`.

Also when no view is available for the request in combination with the object returned by the path function. More specifically, the `model` view predicate or the `name` view predicate do not match.

400 Bad Request: When information in the path or request parameters could not be converted to the required types.

405 Method Not Allowed: When no view exists for the given HTTP request method. More specifically, the `request_method` view predicate does not match.

422 Unprocessable Entity: When the request body supplied with a `POST` or `PUT` request can be parsed (as JSON, for instance), but is not the correct type.

500 Internal Server Error: There is a bug in the server that causes an exception to be raised. Morepath does not generate these itself, but a WSGI server automatically catches any exceptions not handled by Morepath and turns them into 500 errors.

Instead of having to write code that sends back the right status codes manually, you declare paths and views with Morepath and Morepath can usually do the right thing for you automatically. This saves you from writing a lot of custom code when you want to implement HTTP properly.

Sometimes it is still useful to set the status code directly. WebOb lets you raise special exceptions for HTTP errors. You can also set the `webob.response.Response.status` attribute on the response.

JSON

A representation of a resource. JSON is a language that represents data, not user interface (like HTML combined with CSS) or logic (like Python or JavaScript). JSON looks like this:

```
{
  "id": "foo_barson",
  "name": "Foo Barson",
  "occupation": "Carpenter",
  "level": 34
  "friends": ["http://example.com/people/qux_quxson",
              "http://example.com/people/one_twonson"]
}
```

JSON is the most common data representation language used in REST web services. The main alternative is XML. While XML does offer more extensive tooling support, it is a lot more verbose and more difficult to process than JSON. JSON is already very close to the data structures of many programming languages, including JavaScript and Python.

In Python, JSON can be constructed by combining Python dictionaries and lists with strings, numbers, booleans and `None`.

With Morepath you can use the *`morepath.App.json()`* directive to generate JSON programmatically:

```
@App.json(model=MyObject)
def my_object_default(self, request):
    return {
        "id": self.id,
        "name": self.name,
        "occuptation": self.get_occupation(),
        "level": self.level,
        "friends": [request.link(friend) for friend in self.friends]
    }
```

This works like the `view` directive, but in addition converts the return value of the function into a JSON response that is sent to the client.

JSON-LD

JSON-LD is an extension of JSON that helps support linked data in JSON. Any JSON-LD structure is valid JSON, but not every JSON structure is valid JSON-LD.

Using a `@context`, it lets a JSON object describe which parts of it contain hyperlinks, and also allows JSON property names themselves to be interpreted as unique hyperlinks. You can also express that particular property values have a particular data type; this can range from basic data types like datetime to custom data types like "person". All of this can help when you want to process JSON coming from different data sources.

Perhaps more important in practice for REST web services is that it also offers a standard way for a JSON object to have a unique id and a type. Both are identified by a hyperlink, as the special `@id` and `@type` properties. `@type` in particular makes it easier to use JSON data as hypermedia: client behavior can be driven by the type of data that is retrieved, instead of what URL it happened to be retrieved from.

Morepath does not mandate the use of JSON-LD, or has any special support for it, but its link generation facilities make it easier to use it.

HTTP API

A HTTP API is a web service that is built on HTTP; it is based on the notion of HTTP resources on URLs and has an understanding of HTTP request methods.

This is to distinguish it from a web service implementation where HTTP is merely a transport mechanism, such as SOAP.

Because the client needs to understand what URLs exist on the server and how to interpret their response, the coupling between client and server code is relatively tight.

This type of web service is commonly called a *REST* web service, but the original definition of REST goes beyond this and adds hypermedia. Many HTTP APIs only reach level 2 on the Richardson Maturity Model, which isn't full REST yet.

A HTTP API is sometimes simply called *API*, which is also confusing, as the word API has a lot of other uses in development outside of HTTP.

Morepath is designed to help you build HTTP APIs, but also to go you a step further to full REST.

REST web service

Morepath helps you to create REST web service, also known as a *hypermedia API*.

This is level 3 on the Richardson Maturity Model.

This means that to interact with the content of the web service you can follow hyperlinks. A client starts at one root URL and to get to other information it follow links in the content.

Different JSON resources can be distinguished from each other by their type; this can based on the `content-type` of the response, or be based on information within the content itself, such as a type property in JSON (`@type` in JSON-LD).

In other words, the web service represents itself to software much like a web site presents itself to a human: as content with links.

A REST web service allows for a looser coupling between server and client than a plain HTTP API allows, as the client does not need to know more than a single entry point URL into the server, and only needs an understanding of the response types and how to navigate links.

HTML and CSS

HTML is a markup language used to represent a resource. Augmented by CSS, a style language, it determines what you see on a web page.

HTML can be loaded from a files on the server; this typically done with a general web server such as Apache and Nginx. For dynamic applications HTML can also be generated on the server, often using a server-side templating language.

HTML may also be manipulated programmatically in the browser using JavaScript through the DOM API.

In Morepath you can use the *`morepath.App.html()`* view directive to generate HTML programmatically:

```
@App.view(model=MyObject)
def my_object_default(self, request):
    return '<html><head></head><body></body></html>'
```

Morepath at this point does not have support for server-side templating.

See *Static resources with Morepath* for information on how you can load static resources such as CSS and JavaScript automatically to augment a HTML page.

Web page

The browser displays a user interface to the user in the form of a *web page*. A web page is usually constructed using HTML and CSS. Other content such as images, video, audio, SVG, canvas, WebGL may also be embedded into it.

JavaScript code is executed in the browser to make the user interface more dynamic, and this dynamism can go very far.

A web page is loaded by putting a URL in the address bar of the browser. The browser then fetches it (and related resources) from the server. You can do this manually, or by clicking a link, or the URL of the browser may be changed programmatically with JavaScript code.

In the past, all web applications were implemented as a multiple web pages that were generated on the server in response to user actions.

It is also possible to change the URL in the address bar without fetching a complete new web page from the server. This technique is used to implement single-page web applications.

Single-page web application

A single-page web application (SPA) is web application that consists of a single web page that is updated within the browser without the need to load a complete need web page. So the web page is loaded from the server only once, when the user first goes there.

When a user interacts with it, JavaScript code is executed that updates the user interface and may also interact with a web server using AJAX.

A single page web application may update the URL in the address bar of the browser, and respond to URL changes, but it is the same web page that implements the behavior for all these URLs. It may need a bit of server-side support to do so.

Morepath supports the creation of single-page web applications. It also lets you create multi-page applications, but at this point in time has no special support for server-side templating.

Part II

User Guide

You'll find in this section a tour of the features of Morepath, and how to use them to develop your web application.

CHAPTER 9

Paths and Linking

Introduction

Morepath lets you publish model classes on paths using Python functions. It also lets you create links to model instances. To be able do so Morepath needs to be told what variables there are in the path in order to find the model object, and how to find these variables again in the model object in order to construct a link to it.

Paths

Overlapping paths

Morepath lets you define multiple overlapping paths:

```
@App.path(model=Item, path='items/{id}')
def get_item(id):
    ...

@App.path(model=ItemDetail, path='items/{id}/details/{detail_id}')
def get_item_detail(id, detail_id):
    ...
```

If you have overlapping paths, you need to name the variable names the same in the overlapping part of the paths, otherwise Morepath reports a configuration conflict. So you can't have this:

```
@App.path(model=Item, path='items/{id}')
def get_item(id):
    ...

@App.path(model=ItemDetail, path='items/{item_id}/details/{detail_id}')
def get_item_detail(item_id, detail_id):
    ...
```

Morepath reports an error in this case, as `{id}` and `{item_id}` overlap but are different variable names.

Let's assume we have a model class `Overview`:

```
class Overview(object):
    pass
```

Here's how we could expose it to the web under the path `overview`:

```
@App.path(model=Overview, path='overview')
def get_overview():
    return Overview()
```

And let's give it a default view so we can see it when we go to its URL:

```
@App.view(model=Overview)
def overview_default(self, request):
    return "Overview"
```

No variables are involved yet: they aren't in the `path` and the `get_overview` function takes no arguments.

Let's try a single variable now. We have a class `Document`:

```
class Document(object):
    def __init__(self, name):
        self.name = name
```

Let's expose it to the web under `documents/{name}`:

```
@App.path(model=Document, path='documents/{name}')
def get_document(name):
    return query_document_by_name(name)

@App.view(model=Document)
def document_default(self, request):
    return "Document: " + self.name
```

Here we declare a variable in the path (`{name}`), and it gets passed into the `get_document` function. The function does some kind of query to look for a `Document` instance by name. We then have a view that knows how to display a `Document` instance.

We can also have multiple variables in a path. We have a `VersionedDocument`:

```
class VersionedDocument(object):
    def __init__(self, name, version):
        self.name = name
        self.version = version
```

We could expose this to the web like this:

```
@App.path(model=VersionedDocument,
          path='versioned_documents/{name}-{version}')
def get_versioned_document(name, version):
    return query_versioned_document(name, version)

@App.view(model=VersionedDocument)
def versioned_document_default(self, request):
    return "Versioned document: %s %s" % (self.name, self.version)
```

The rule is that all variables declared in the path can be used as arguments in the model function.

URL query parameters

What if we want to use URL parameters to expose models? That is possible too. Let's look at the `Document` case first:

```
@App.path(model=Document, path='documents')
def get_document(name):
    return query_document_by_name(name)
```

`get_document` has an argument `name`, but it doesn't appear in the path. This argument is now taken to be a URL parameter. So, this exposes URLs of the type `documents?name=foo`. That's not as nice as `documents/foo`, so we recommend against parameters in this case: you should use paths to identify something.

URL parameters are more useful for queries. Let's imagine we have a collection of documents and we have an API on it that allows us to search in it for some `text`:

```
class DocumentCollection(object):
    def __init__(self, text):
        self.text = text

    def search(self):
        if self.text is None:
            return []
        return fulltext_search(self.text)
```

We now publish this collection, making it searchable:

```
@App.path(model=DocumentCollection, path='search')
def document_search(text):
    return DocumentCollection(text)
```

To be able to see something, we add a view that returns a comma separated string with the names of all matching documents:

```
@App.view(model=DocumentCollection)
def document_collection_default(self, request):
    return ', '.join([document.name for document in self.search()])
```

As you can see it uses the `DocumentCollection.search` method.

Unlike path variables, URL parameters can be omitted, i.e. we can have a plain `search` path without a `text` parameter. In that case `text` has the value `None`. The `search` method has code to handle this special case: it returns the empty list.

Often it's useful to have a default instead. Let's imagine we have a default search query, `all` that should be used if no `text` parameter is supplied (instead of `None`). We make a default available by supplying a default value in the `document_search` function:

```
@App.path(model=DocumentCollection, path='search')
def document_search(text='all'):
    return DocumentCollection(text)
```

Note that defaults have no meaning for path variables, because whenever a path is resolved, all variables in it have been found. They can be used as type hints however; we'll talk more about those soon.

Like with path variables, you can have as many URL parameters as you want.

Extra URL query parameters

URL parameters are matched with function arguments, but it could be you're interested in an arbitrary amount of extra URL parameters. You can specify that you're interested in this by adding an `extra_parameters` argument:

```
@App.path(model=DocumentCollection, path='search')
def document_search(text='all', extra_parameters):
    return DocumentCollection(text, extra_parameters)
```

Now any additional URL parameters are put into the `extra_parameters` dictionary. So, `search?text=blah&a=A&b=B` would match `text` with the `text` parameter, and there would be an `extra_parameters` containing `{'a': 'A', 'b': 'B'}`.

`extra_parameters` can also be useful for the case where the name of the parameter is not a valid Python name (such as `@foo`) – you can still receive such parameters using `extra_parameters`.

Linking

To create a link to a model, we can call *`morepath.Request.link()`* in our view code. At that point the model object is examined to retrieve the variables so that the path can be constructed.

Here is a simple case involving `Document` again:

```
class Document(object):
    def __init__(self, name):
        self.name = name

@App.path(model=Document, path='documents/{name}')
def get_document(name):
    return query_document_by_name(name)
```

We add a named view called `link` that links to the document itself:

```
@App.view(model=Document, name='link')
def document_self_link(self, request):
    return request.link(self)
```

The view at `/documents/foo/link` produces the link `/documents/foo`. That's the right one!

So, it constructs a link to the document itself. This view is not very useful, but the principle is the same everywhere in any view: as long as we have a `Document` instance we can create a link to it using `request.link()`.

You can also give `link` a name to link to a named view. Here's a `link2` view creates a link to the `link` view:

```
@App.view(model=Document, name='link2')
def document_self_link(self, request):
    return request.link(self, name='link')
```

So the view at `/documents/foo/link2` produces the link `/documents/foo/link`.

Linking with path variables

How does the `request.link` code know what the value of the `{name}` variable should be so that the link can be constructed? In this case this happened automatically: the value of the `name` attribute of `Document` is assumed to be the one that goes into the link.

This automatic rule won't work everywhere, however. Perhaps an attribute with a different name is used, or a more complicated method is used to construct the name. For those cases we can take over and supply a custom `variables` function that knows how to construct the variables needed to construct the link from the model.

The variables function gets the model object as a single argument and needs to return a dictionary. The keys should be the variable names used in the path or URL parameters, and the values should be the values as extracted from the model.

As an example, here is the `variables` function for the `Document` case made explicit:

```
@App.path(model=Document, path='documents/{name}',
          variables=lambda obj: dict(name=obj.name))
def get_document(name):
    return query_document_by_name(name)
```

Or to spell it out without the use of `lambda`:

```
def document_variables(obj):
    return dict(name=obj.name)

@App.path(model=Document, path='documents/{name}',
          variables=document_variables)
def get_document(name):
    return query_document_by_name(name)
```

Let's change `Document` so that the name is stored in the `id` attribute:

```
class DifferentDocument(object):
    def __init__(self, name):
        self.id = name
```

Our automatic `variables` won't cut it anymore, so we have to be explicit:: attribute, we can do this:

```
@App.path(model=DifferentDocument, path='documents/{name}',
          variables=lambda obj: dict(name=obj.id))
def get_document(name):
    return query_document_by_name(name)
```

All we've done is adjust the `variables` function to take `model.id`.

Getting variables works for multiple variables too of course. Here's the explicit `variables` for the `VersionedDocument` case that takes multiple variables:

```
@App.path(model=VersionedDocument,
          path='versioned_documents/{name}-{version}',
          variables=lambda obj: dict(name=obj.name,
                                     version=obj.version))
def get_versioned_document(name, version):
    return query_versioned_document(name, version)
```

If you have `extra_parameters`, the default variables expects that `extra_parameters` to exist as an attribute on the object, but you can write a custom `variables` that retrieves this dictionary from the object in some other way:

```
@App.path(model=SearchResults,
          path='search',
          variables=lambda obj: dict(text=obj.search_text,
                                     extra_parameters=obj.get_extra()))
def get_search_results(text, extra_parameters):
    ...
```

Linking with URL query parameters

Linking works the same way for URL parameters as it works for path variables.

Here's a `get_model` that takes the document name as a URL parameter, using an implicit `variables`:

```
@App.path(model=Document, path='documents')
def get_document(name):
    return query_document_by_name(name)
```

Now we add back the same `self_link` view as we had before:

```
@App.view(model=Document, name='link')
def document_self_link(self, request):
    return request.link(self)
```

Here's `get_document` with an explicit `variables`:

```
@App.path(model=Document, path='documents',
          variables=lambda obj: dict(name=obj.name))
def get_document(name):
    return query_document_by_name(name)
```

i.e. exactly the same as for the path variable case.

Let's look at a document exposed on this URL:

```
/documents?name=foo
```

Then the view `documents/link?name=foo` constructs the link:

```
/documents?name=foo
```

The `documents/link?name=foo` is interesting: the `name=foo` parameters are added to the end, but they are used by the `get_document` function, *not* by its views. Here's `link2` again to further demonstrate this behavior:

```
@App.view(model=Document, name='link2')
def document_self_link(self, request):
    return request.link(self, name='link')
```

When we now go to `documents/link2?name=foo` we get the link `/documents/link?name=foo`.

Prefixing links with a base URL

By default, *`morepath.Request.link()`* generates links as fully qualified URLs using the `HOST` header and the given protocol (`http`, `https`), for instance:

```
http://localhost/document
```

You can use the *morepath.App.link_prefix()* decorator to override this behavior. For example, if you *do* not want to add the full hostname (in fact the behavior of Morepath before version 0.9), you can write:

```
@App.link_prefix()
def simple_link_prefix(request):
    return ''
```

The `link_prefix` function is only called once per request per app, during the first call to *morepath.Request.link()* for an app. After this it is cached for the rest of the duration of that request.

Linking to external applications

As a more advanced use case for `link_prefix`, you can use it to represent an application that is completely external, just for the purposes of making it easier to create a link to it.

Let's say we want to be able to link to documents on the external site `http://example.com`, and that these documents live on URLs like `http://example.com/documents/{id}`.

We can create a model for such an external document first:

```
class ExternalDocument(object):
    def __init__(self, id):
        self.id = id
```

And declare the path space of the external site:

```
@ExternalApp.path(model=ExternalDocument, path='/documents/{id}')
def get_external_document(id):
    return ExternalDocument(id)
```

We don't need to declare any views for `ExternalDocument`; `ExternalApp` only exists to let you generate a link to the `example.com` external site more easily.

Now we want `request.link(ExternalDocument('foo'))` to result in the link `http://example.com/documents/foo`. All we need to do is to declare a special `link_prefix` for the external app where we hardcode `http://example.com`:

```
@ExternalApp.link_prefix()
def simple_link_prefix(request):
    return 'http://example.com'
```

Type hints

So far we've only dealt with variables that have string values. But what if we want to use other types for our variables, such as `int` or `datetime`? What if we have a record that you obtain by an `int` id, for instance? Given some `Record` class that has an `int` id like this:

```
class Record(object):
    def __init__(self, id):
        self.id = id
```

We could do this to expose it:

```
@App.path(model=Record, path='records/{id}')
def get_record(id):
    try:
        id = int(id)
    except ValueError:
        return None
    return record_by_id(id)
```

But Morepath offers a better way. We can tell Morepath we expect an int and only an int, and if something else is supplied, the path should not match. Here's how:

```
@App.path(model=Record, path='records/{id}')
def get_record(id=0):
    return record_by_id(id)
```

We've added a default parameter (`id=0`) here that Morepath uses as an indication that only an int is expected. Morepath will now automatically convert `id` to an int before it enters the function. It also gives a `404 Not Found` response for URLs that don't have an int. So it accepts `/records/100` but gives a 404 for `/records/foo`.

Let's examine the same case for an `id` URL parameter:

```
@App.path(model=Record, path='records')
def get_record(id=0):
    return record_by_id(id)
```

This responds to an URL like `/records?id=100`, but rejects `/records/id=foo` as `foo` cannot be converted to an int. It rejects a request with the latter path with a `400 Bad Request` error.

By supplying a default for a URL parameter we've accomplished two in one here, as it's a good idea to supply defaults for URL parameters anyway, as that makes them properly optional.

Conversion

Sometimes simple type hints are not enough. What if multiple possible string representations for something exist in the same application? Let's examine the case of `datetime.date`.

We could represent it as a string in ISO 8601 format as returned by the `datetime.date.isoformat()` method, i.e. `2014-01-15` for the 15th of january 2014. We could also use ISO 8601 compact format, namely `20140115` (and this what Morepath defaults to). But we could also use another representation, say `15/01/2014`.

Let's first see how a string with an ISO compact date can be decoded (deserialized, loaded) into a `date` object:

```
from datetime import date
from time import mktime, strptime

def date_decode(s):
    return date.fromtimestamp(mktime(strptime(s, '%Y%m%d')))
```

We can try it out:

```
>>> date_decode('20140115')
datetime.date(2014, 1, 15)
```

Note that this function raises a `ValueError` if we give it a string that cannot be converted into a date:

```
>>> date_decode('blah')
Traceback (most recent call last):
   ...
ValueError: time data 'blah' does not match format '%Y%m%d'
```

This is a general principle of decode: a decode function can fail and if it does it should raise a `ValueError`.

We also specify how to encode (serialize, dump) a `date` object back into a string:

```
def date_encode(d):
    return d.strftime('%Y%m%d')
```

We can try it out too:

```
>>> date_encode(date(2014, 1, 15))
'20140115'
```

A encode function should never fail, if at least presented with input of the right type, in this case a `date` instance.

Inverse

To help you write these functions, note that they're the inverse each other, so these equality are both True. For any string `s` that can be decoded, this is true:

```
encode(decode(s)) == s
```

And for any object that can be encoded, this is true:

```
decode(encode(o)) == o
```

The output of decode should always be input for encode, and the output of encode should always be input for decode.

Now that we have our `date_decode` and `date_encode` functions, we can wrap them in an *morepath.Converter* object:

```
date_converter = morepath.Converter(decode=date_decode, encode=date_encode)
```

Let's now see how we can use `date_converter`.

We have some kind of `Records` collection that can be parameterized with `start` and `end` to select records in a date range:

```
class Records(object):
   def __init__(self, start, end):
      self.start = start
      self.end = end

   def query(self):
      return query_records_in_date_range(self.start, self.end)
```

We expose it to the web:

```
@App.path(model=Records, path='records',
          converters=dict(start=date_converter, end=date_converter))
def get_records(start, end):
    return Records(start, end)
```

We also add a simple view that gives us comma-separated list of matching record ids:

```
@App.view(model=Records)
def records_view(self, request):
    return ', '.join([str(record.id) for record in self.query()])
```

We can now go to URLs like this:

```
/records?start=20110110&end=20110215
```

The `start` and `end` URL parameters now are decoded into `date` objects, which get passed into `get_records`. And when you generate a link to a `Records` object, the `start` and `end` dates are encoded into strings.

What happens when a decode raises a `ValueError`, i.e. improper dates were passed in? In that case, the URL parameters cannot be decoded properly, and Morepath returns a `400 Bad Request` response.

You can also use encode and decode for arguments used in a path:

```
@App.path(model=Day, path='days/{d}', converters=dict(d=date_converter))
def get_day(d):
    return Day(d)
```

This publishes the model on a URL like this:

```
/days/20110101
```

When you pass in a broken date, like `/days/foo`, a `ValueError` is raised by the date decoder, and a `404 not Found` response is given by the server: the URL does not resolve to a model.

Default converters

Morepath has a number of default converters registered; we already saw examples for int and strings. Morepath also has a default converter for `date` (compact ISO 8601, i.e. `20131231`) and `datetime` (i.e. `20131231T23:59:59`).

You can add new default converters for your own classes, or override existing default behavior, by using the *`morepath.App.converter()`* decorator. Let's change the default behavior for `date` in this example to use ISO 8601 *extended* format, so that dashes are there to separate the year, month and day, i.e. `2013-12-31`:

```
def extended_date_decode(s):
    return date.fromtimestamp(mktime(strptime(s, '%Y-%m-%d')))

def extended_date_encode(d):
    return d.strftime('%Y-%m-%d')

@App.converter(type=date)
def date_converter():
    return Converter(extended_date_decode, extended_date_encode)
```

Now Morepath understand type hints for `date` differently:

```
@App.path(model=Day, path='days/{d}')
def get_day(d=date(2011, 1, 1)):
    return Day(d)
```

has models published on a URL like:

```
/days/2013-12-31
```

Type hints and converters

You may have a situation where you don't want to add a default argument to indicate the type hint, but you know you want to use a default converter for a particular type. For those cases you can pass the type into the `converters` dictionary as a shortcut:

```
@App.path(model=Day, path='days/{d}', converters=dict(d=date))
def get_day(d):
    return Day(d)
```

The variable `d` is now interpreted as a `date`. Morepath uses whatever converter that was registered for that type.

List converters

What if you want to allow a list of parameters instead of just a single one? You can do this by wrapping the converter or type in the `converters` dictionary in a list:

```
@App.path(model=Days, path='days', converters=dict(d=[date]))
def get_days(d):
    return Days(d)
```

Now the `d` parameter will be interpreted as a list. This means URLs like this are accepted:

```
/days?d=2014-01-01

/days?d=2014-01-01&d=2014-01-02

/days
```

For the first case, `d` is a list with one date item, in the second case, `d` has 2 items, and in the third case the list `d` is empty.

get_converters

Sometimes you only know what converters are available at run-time; this particularly relevant if you want to supply converters for the values in `extra_parameters`. You can supply the converters using the special `get_converters` parameter to `@app.path`:

```
def my_get_converters():
    return { 'something': int }

@App.path(path='search', model=SearchResults,
          get_converters=my_get_converters)
...
```

Now if there is a parameter (or extra parameter) called `something`, it is converted to an `int`.

You can combine `converters` and `get_converters`. If you use both, `get_converters` will override any converters also defined in the static `converters`. This can also be useful for dealing with URL parameters that are not valid Python names, such as `@foo` or `foo[]`; these can still be converted using `get_converters`.

Required

Sometimes you may want a URL parameter to be required: when the URL parameter is missing, it's an error and a `400 Bad Request` should be returned. You can do this by passing in a `required` argument to the model decorator:

```
@App.path(model=Record, path='records', required=['id'])
def get_record(id):
    return query_record(id)
```

Normally when the `id` URL parameter is missing, the `None` value is passed into `get_record` (if there is no default). But since we made `id` required, `400 Bad Request` will be issued if `id` is missing now. `required` only has meaning for URL parameters; path variables are always present if the path matches at all.

Absorbing

In some special cases you may want a path to match all sub-paths, absorbing them. This can be useful if you are writing a server backend to a client side application that does routing on the client using the HTML 5 history API – the server needs to handle catch all subpaths in that case and send them back to the client, where they can be handled by the client-side router.

You can do this using the special `absorb` argument to the path decorator, like this:

```
class Model(object):
    def __init__(self, absorb):
        self.absorb = absorb

@App.path(model=Model, path='start', absorb=True)
def get_foo(absorb):
    return Model(absorb)
```

As you can see, if you use `absorb` then a special `absorb` argument is passed into the model factory function.

Now the `start` path matches all of its sub-paths. So for this path:

```
/start/foo/bar/baz
```

`model.absorb` is `foo/bar/baz`.

It also matches if there is no sub-path:

```
/start
```

`model.absorb` is the empty string `''`.

Note that you cannot use view names with a path that absorbs; only a default view with the empty name. View names are absorbed along with the rest of the path.

Note also that you cannot define an explicit path under an absorbed path – this is ignored. This means that the following additional code has no effect:

```
@App.path(model=Foo, path='start/extra')
```

You can still generate a link to a model that is under an absorbed path – it uses the value of the `absorb` variable.

Linking with the model class

Instead of using *morepath.Request.link()* you can also construct links using *morepath.Request.class_link()*. You can use this for optimization purposes when creating an instance to link to would be relatively expensive; if you do have the instance it's generally easier to just link to that instead using *request.link*.

To use *request.class_link* you give the model *class* instead of an instance, and also provide a dictionary of variables to use to construct the link:

```
@App.view(model=Document, name='class_link')
def document_self_link(self, request):
    return request.class_link(Document, variables={'name': 'Document name'})
```

The variables are used in the same way as for *request.link*, so additional parameters listed in the path function are interpreted as URL parameters.

Warning: *request.class_link* does *NOT* obey the *defer_links* directive, as this relies on the instance of what is being linked to in order to determine the application to which it defers.

Proxy support

If you have a Morepath application that sits behind a trusted proxy that sets the Forwarded header, then you want links generated by Morepath take this header into account. To do this, you can make your project depend on the more.forwarded extension. After you have it installed, you can subclass your app from `more.forwarded.ForwardedApp` to make your app proxy-aware. Note that you only need to do this for the root app, not for any apps mounted into it.

You should *only* use this extension if you know you are behind a trusted proxy that indeed sets the `Forwarded` header. This because otherwise you could expose your application to attacks that affect link generation through the Forwarded header.

CHAPTER 10

Views

Introduction

Morepath views are looked up through the URL path, but not through the routing procedure. Routing stops at model objects. Then the last segment of the path is taken to identify the view by name.

Named views

Let's examine a path:

```
/documents/1/edit
```

If there's a model like this:

```
@App.path(model=Document, path='/documents/{id}')
def get_document(id):
    return query_for_document(id)
```

then `/edit` identifies a view named `edit` on the `Document` model (or on one of its base classes). Here's how we define it:

```
@App.view(model=Document, name='edit')
def document_edit(self, request):
    return "edit view on model: %s" % self.id
```

Default views

Let's examine this path:

```
/documents/1
```

If the model is published on the path `/documents/{id}`, then this is a path to the *default* view of the model. Here's how that view is defined:

```
@App.view(model=Document)
def document_default(self, request):
    return "default view on model: %s" % self.id
```

The default view is the view that gets triggered if there is no special path segment in the URL that indicates a specific view. The default view has as its name the empty string `""`, so this registration is the equivalent of the one above:

```
@App.view(model=Document, name="")
def document_default(self, request):
    return "default view on model: %s" % self.id
```

Generic views

Generic views in Morepath are nothing special: the thing that makes them generic is that their model is a base class, and inheritance does the rest. Let's see how that works.

What if we want to have a view that works for any model that implements a certain API? Let's imagine we have a `Collection` model:

```
class Collection(object):
   def __init__(self, offset, limit):
       self.offset = offset
       self.limit = limit

   def query(self):
       raise NotImplementedError
```

A `Collection` represents a collection of objects, which can be ordered somehow. We restrict the objects we actually get by offset and limit. With offset 100 and limit 10, we get objects 100 through 109.

`Collection` is a base class, so we don't actually implement how to do a query. That's up to the subclasses. We do specify that query is supposed to return objects that have an `id` attribute.

We can create a view to this abstract collection that displays the ids of the things in it in a comma separated list:

```
@App.view(model=Collection)
def collection_default(self, request):
    return ", ".join([str(item.id) for item in self.query()])
```

This view is generic: it works for any kind of collection.

We can now create a concrete collection that fulfills the requirements:

```
class Item(object):
   def __init__(self, id):
       self.id = id

class MyCollection(Collection):
   def query(self):
       return [Item(str(i)) for i in
               range(self.offset, self.offset + self.limit)
```

When we now publish the concrete `MyCollection` on some URL:

```
@App.path(model=MyCollection, path='my_collection')
def get_my_collection():
    return MyCollection()
```

it automatically gains a default view for it that represents the ids in it as a comma separated list. So the view `collection_default` is *generic*.

Details

The decorator *morepath.App.view()* (`@App.view`) takes two arguments here, `model`, which is the class of the model the view is representing, and `name`, which is the name of the view in the URL path.

The `@App.view` decorator decorates a function that takes two arguments: a `self` and a `request`.

The `self` object is the model that's being viewed, i.e. the one found by the `get_document` function. It is going to be an instance of the class given by the `model` parameter.

The `request` object is an instance of *morepath.Request*, which in turn is a special kind of `webob.request.BaseRequest`. You can get request information from it like arguments or form data, and it also exposes a few special methods, such as *morepath.Request.link()*.

The `@App.path` and `@App.view` decorators are associated by indirectly their `model` parameters: the view works for a given model path if the `model` parameter is the same, or if the view is associated with a base class of the model exposed by the `@App.path` decorator.

Ambiguity between path and view

Let's examine these simple paths in an application:

```
/folder
/folder/{name}
```

`/folder` shows an overview of the items in it. `/folder/{name}` is a way to get to an individual item.

This means:

```
/folder/some_item
```

is a path if there is an item in the folder with the name `some_item`.

Now what if we also want to have a path that allows you to edit the folder? It'd be natural to spell it like this:

```
/folder/edit
```

i.e. there is a path `/folder` with a view `edit`.

But now we have a problem: how does Morepath know that `edit` is a view and not a named item in the folder? The answer is that it doesn't. You cannot reach the view this way.

Instead we have to make it explicit in the path that we want a view with a + character:

```
/folder/+edit
```

Now Morepath won't try to interpret `+edit` as a named item in the folder, but instead looks up the view.

Any view can be addressed not just by name but also by its name with a + prefix. To generate a link to a name with a + prefix you can use the prefix as well, so you can write:

```
request.link(my_folder, '+edit')
```

render

By default `@App.view` returns either a *morepath.Response* object or a string that gets turned into a response. The `content-type` of the response is not set. For a HTML response you want a view that sets the `content-type` to `text/html`. You can do this by passing a `render` parameter to the `@App.view` decorator:

```
@App.view(model=Document, render=morepath.render_html)
def document_default(self, request):
    return "<p>Some html</p>"
```

morepath.render_html() is a very simple function:

```
def render_html(content, request):
    response = morepath.Response(content)
    response.content_type = 'text/html'
    return response
```

You can define your own `render` functions; they just need to take some content (any object, in this case its a string), and return a `Response` object.

Another render function is *morepath.render_json()*. Here it is:

```
import json

def render_json(content, request):
    response = morepath.Response(json.dumps(content))
    response.content_type = 'application/json'
    return response
```

We'd use it like this:

```
@App.view(model=Document, render=morepath.render_json)
def document_default(self, request):
    return {'my': 'json'}
```

HTML views and JSON views are so common we have special shortcut decorators:

- `@App.html` (*morepath.App.html()*)
- `@App.json` (*morepath.App.json()*)

Here's how you use them:

```
@App.html(model=Document)
def document_default(self, request):
    return "<p>Some html</p>"

@App.json(model=Document)
def document_default(self, request):
    return {'my': 'json'}
```

Templates

You can use a server template with a view by using the `template` argument:

```
@App.html(model=Document, template='document.pt')
def document_default(self, request):
    return { 'title': self.title, 'content': self.content }
```

See *Templates* for more information.

Permissions

We can protect a view using a `permission`. A permission is any Python class:

```
class Edit(object):
    pass
```

The class doesn't do anything; it's just a marker for permission.

You can use such a class with a view:

```
@App.view(model=Document, name='edit', permission=Edit)
def document_edit(self, request):
    return 'edit document'
```

You can define which users have what permission on which models by using the *morepath.App.permission_rule()* decorator. To learn more, read *Security*.

Manipulating the response

Sometimes you want to do things to the response specific to the view, so that you cannot do it in a `render` function. Let's say you want to add a cookie using `webob.Response.set_cookie()`. You don't have access to the response object in the view, as it has not been created yet. It is only created *after* the view has returned. We can register a callback function to be called after the view is done and the response is ready using the *morepath.Request.after()* decorator. Here's how:

```
@App.view(model=Document)
def document_default(self, request):
    @request.after
    def manipulate_response(response):
        response.set_cookie('my_cookie', 'cookie_data')
    return "document default"
```

`after` only applies if the view was successfully resolved into a response. If your view raises an exception for any reason, or if Morepath itself does, any `after` set in the view does not apply to the response for this exception. If the view *returns* a response object directly itself, then `after` is also not run - you have the response object to manipulate directly. Note that this the case when you use *morepath.redirect()*: this returns a redirect response object.

request_method

By default, a view only answers to a `GET` request: it doesn't handle other request methods like `POST` or `PUT` or `DELETE`. To write a view that handles another request method you need to be explicit and pass in the `request_method` parameter:

```
@App.view(model=Document, name='edit', request_method='POST')
def document_edit(self, request):
    return "edit view on model: %s" % self.id
```

Now we have a view that handles `POST`. Normally you cannot have multiple views for the same document with the same name: the Morepath configuration engine rejects that. But you can if you make sure they each have a different request method:

```
@App.view(model=Document, name='edit', request_method='GET')
def document_edit_get(self, request):
    return "get edit view on model: %s" % self.id

@App.view(model=Document, name='edit', request_method='POST')
def document_edit_post(self, request):
    return "post edit view on model: %s" % self.id
```

Grouping views

At some point you may have a lot of view decorators that share a lot of information; multiple views for the same model are the most common example.

Instead of writing this:

```
@App.view(model=Document)
def document_default(self, request):
    return "default"

@App.view(model=Document, name='edit')
def document_edit(self, request):
    return "edit"
```

You can use the `with` statement to write this instead:

```
with App.view(model=Document) as view:
   @view()
   def document_default(self, request):
       return "default"

   @view(name="edit")
   def document_edit(self, request):
       return "edit"
```

This is equivalent to the above, you just don't have to repeat `model=Document`. You can use this for any parameter for `@App.view`.

This use of the `with` statement is in fact general; it can be used like this with any Morepath directive, and with any parameter for such a directive. The `with` statement may even be nested, though we recommend being careful with that, as it introduces a lot of indentation.

Predicates

The `model`, `name`, and `request_method` arguments on the `@App.view` decorator are examples of *view predicates*. You can add new ones by using the *`morepath.App.predicate()`* decorator.

Let's say we have a view that we only want to kick in when a certain request header is set to something:

```
import reg

@App.predicate(generic.view, name='something', default=None,
               index=reg.KeyIndex,
               after=morepath.LAST_VIEW_PREDICATE)
def something_predicate(request):
    return request.headers.get('Something')
```

We can use any information in the request and model to construct the predicate. Now you can use it to make a view that only kicks in when the `Something` header is `special`:

```
@App.view(model=Document, something='special')
def document_default(self, request):
    return "Only if request header Something is set to special."
```

If you have a predicate and you *don't* use it in a `@App.view`, or set it to `None`, the view works for the `default` value for that predicate. The `default` parameter is also used when rendering a view using *`morepath.Request.view()`* and you don't pass in a particular value for that predicate.

Let's look into the predicate directive in a bit more detail.

You can use either `self` or `request` as the argument for the predicate function. Morepath sees this argument and sends in either the object instance or the request.

We use `reg.KeyIndex` as the index for this predicate. You can also have predicate functions that return a Python class. In that case you should use `reg.ClassIndex`.

`morepath.LAST_VIEW_PREDICATE` is the last predicate defined by Morepath itself. Here we want to insert the `something_predicate` after this predicate in the predicate evaluation order.

The `after` parameter for the predicate determines which predicates match more strongly than another; a predicate after another one matches more weakly. If there are two view candidates that both match the predicates, the strongest match is picked.

request.view

It is often useful to be able to compose a view from other views. Let's look at our earlier `Collection` example again. What if we wanted a generic view for our collection that included the views for its content? This is easiest demonstrated using a JSON view:

```
@App.json(model=Collection)
def collection_default(self, request):
    return [request.view(item) for item in self.query()]
```

Here we have a view that for all items returned by query includes its view in the resulting list. Since this view is generic, we cannot refer to a *specific* view function here; we just want to use the view function appropriate to whatever `item` may be. For this we can use *`morepath.Request.view()`*.

We could for instance have a particular item with a view like this:

```
@App.json(model=ParticularItem)
def particular_item_default(self, request):
    return {'id': self.id}
```

And then the result of `collection_default` is something like:

```
[{'id': 1}, {'id': 2}]
```

but if we have a some other item with a view like this:

```
@App.json(model=SomeOtherItem)
def some_other_item_default(self, request):
    return self.name
```

where the name is some string like `alpha` or `beta`, then the output of `collection_default` is something like:

```
['alpha', 'beta']
```

So `request.view` can make it much easier to construct composed JSON results where JSON representations are only loosely coupled.

You can also use `predicates` in `request.view`. Here we get the view with the `name` `"edit"` and the `request_method` `"POST"`:

```
request.view(item, name="edit", request_method="POST")
```

You can also create views that are for internal use only. You can use them with `request.view()` but they won't show up to the web; going to such a view is a 404 error. You can do this by passing the `internal` flag to the directive:

```
@App.json(model=SomeOtherItem, name='extra', internal=True)
def some_other_item_extra(self, request):
    return self.name
```

The `extra` view can be used with `request.view(item, name='extra')`, but it is not available on the web – there is no `/extra` view.

Exception views

WebOb HTTP exceptions

A list of standard WebOb HTTP exceptions

WebOb exceptions are also response objects, so you could return them directly from your view instead of raising them. But not that if you do this exception views won't be used, however – the default WebOb exception response view is used always.

Sometimes your application raises an exception. This can either be a HTTP exception, for instance when the user goes to a URL that does not exist, or an arbitrary exception raised by the application.

HTTP exceptions are by default rendered in the standard WebOb way, which includes some text to describe Not Found, etc. Other exceptions are normally caught by the web server and result in a HTTP 500 error (internal server error).

You may instead want to customize what these exceptions look like. You can do so by declaring a view using the exception class as the model. Here's how you make a custom 404 Not Found:

```
from webob.exc import HTTPNotFound

@App.view(model=HTTPNotFound)
def notfound_custom(self, request):
    def set_status_code(response):
        response.status_code = self.code # pass along 404
    request.after(set_status_code)
    return "My custom not found!"
```

We have to add the `set_status_code` to make sure the response is still a 404; otherwise we change the 404 to a 200 Ok! This shows that `self` is indeed an instance of `HTTPNotFound` and we can access its `code` attribute.

Your application may also define its own custom exceptions that have a meaning particular to the application. You can create custom views for those as well:

```
class MyException(Exception):
    pass

@App.view(model=MyException)
def myexception_default(self, request):
     return "My exception"
```

Without an exception view for `MyException` any view code that raises `MyException` would bubble all the way up to the WSGI server and a 500 Internal Server Error is generated.

But with the view for `MyException` in place, whenever `MyException` is raised you get the special view instead.

CHAPTER 11

Templates

Introduction

When you generate HTML from the server (using HTML views) it can be very handy to have a template language available. A template language provides some high-level constructs for generating HTML, which are handy. It can also help you avoid HTML injection security bugs because it takes care of escaping HTML. It may also be useful to separate HTML presentation from code.

This document discusses template rendering on the server. In some modern web applications template rendering is done in the browser instead of on the server. To do client-side template rendering you need to use a *Client web framework* with Morepath. See also *Static resources with Morepath*.

Morepath does not have a template language built in. The example in this document uses more.chameleon. more.chameleon integrates the Chameleon template engine, which implements the ZPT template language. If you prefer Jinja2, you can use the more.jinja2 extension instead. You can also integrate other template languages.

To use a template you need to use the `template` argument with the *`morepath.App.html()`* view directive.

Example

This example presupposes that more.chameleon and its dependencies have been installed. Here is how we use it:

```
from more.chameleon import ChameleonApp

class App(ChameleonApp):
    pass

@App.template_directory()
def get_template_directory():
    return 'templates'

@App.html(model=Person, template='person.pt')
```

```
def person_default(self, request):
    return { 'name': self.name }
```

Let's examine this code. First we import `ChameleonApp` and subclass from it in our own app. This enables Chameleon templating for the `.pt` file extension.

We then need to specify the directory that contains our templates using the *morepath.App.template_directory()* directive. The directive should return either an absolute or a relative path to this template directory. If a relative path is returned, it is automatically made relative to the directory the Python module is in.

Next we use `template='person.pt'` in the HTML view directive. `person.pt` is a file sitting in the `templates` directory, with this content:

```
<html>
<body>
  <p>Hello ${name}!</p>
</body>
</html>
```

Once we have this set up, given a person with a `name` attribute of `"world"`, the output of the view is the following HTML:

```
<html>
<body>
  <p>Hello world!</p>
</body>
</html>
```

The template is applied on the return value of the view function and the request. This results in a rendered template that is returned as the response.

Overrides

When you subclass an app you may want to override some of the templates it uses, or add new templates. You can do this by using the `template_directory` directive in your subclassed app:

```
class SubApp(App):
    pass

@SubApp.template_directory()
def get_override_template_directory():
   return 'templates_override'
```

Morepath's template integration searches for templates in the template directories in application order, so for `SubApp` here, first `templates_override`, and then `templates` as defined by the base `App`. So for `SubApp`, you can override a template defined in the directory `templates` by placing a file with the same name in the directory `templates_override`. This only affects `SubApp`, not `App` itself.

You can also use the `before` argument with the *morepath.App.template_directory()* directive to specify more exactly how you want template directories to be searched. This can be useful if you want to organize your templates in multiple directories in the same application. If `get_override_template_directory` should come before `get_template_directory` in the directory search path, you should use `before=get_template_directory`:

```
@SubApp.template_directory(before=get_template_directory)
def get_override_template_directory():
    return 'templates_override'
```

but it is usually simpler not to be this explicit and to rely on application inheritance instead.

Details

Templates are loaded during configuration time at startup. The file extension of the extension (such as `.pt`) indicates the template engine to use.

Morepath itself does not support any template language out of the box, but lets you register a template language engine for a file extension. You can reuse a template language integration in the same way you reuse any Morepath code: by subclassing the app class that implements it in your app.

The template language integration works like this:

- During startup time, `person.pt` is loaded from the configured template directories as a template object.
- When the `person_default` view is rendered, its return value is passed into the template, along with the request. The template language integration code then makes this information available for use by the template – the details are up to the integration (and should be documented there).

The `template` argument works not just with `html` but also with `view`, `json`, and any other view functions you may have. It's most useful for `html` views however.

Integrating a new template engine

A template in Morepath is actually just a convenient way to generate a `render` function for a view. That `render` function is then used just like when you write it manually: it's given the return value of the view function along with a request object, and should return a WebOb response.

Here is an example of how you can integrate the Chameleon template engine for `.pt` files (taken from more.chameleon):

```
import chameleon

@App.template_loader(extension='.pt')
def get_template_loader(template_directories, settings):
    settings = settings.chameleon.__dict__.copy()
    # we control the search_path entirely by what we pass here as
    # template_directories, so we never want the template itself
    # to prepend its own path
    settings['prepend_relative_search_path'] = False
    return chameleon.PageTemplateLoader(
        template_directories,
        default_extension='.pt',
        **settings)

@App.template_render(extension='.pt')
def get_chameleon_render(loader, name, original_render):
    template = loader.load(name)

    def render(content, request):
        variables = {'request': request}
```

```
        variables.update(content)
        return original_render(template.render(**variables), request)
    return render

@App.setting_section(section='chameleon')
def get_setting_section():
    return {'auto_reload': False}
```

Some details:

- `extension` is the file extension. When you refer to a template with a particular extension, this template engine is used.
- The function decorated by *`morepath.App.template_loader()`* gets two arguments: directories to look in for templates (earliest in the list first), and Morepath settings from which template engine settings can be extracted.
- The function decorated by *`morepath.App.template_render()`* gets three arguments:
  - `loader`: the loader constructed by the `template_loader` directive.
  - `name`: the name of the template to create a render function for.
  - The `original_render` function as passed into the view decorator, so `render_html` for instance. It takes the content to render and the request and returns a webob response object. then passed along to Chameleon.

  The decorated function needs to return a `render` function which takes the content to render (output from view function) and the request as arguments.

  The implementation of this can use the original `render` function which is passed in as an argument as `original_render` function. It can also create a `morepath.Response` object directly.

CHAPTER 12

Configuration

Introduction

When you use a Morepath directive, for example to define a *view*, a *path*, a *setting* or a *tween*, this is called Morepath *configuration*. Morepath configuration can also be part of third-party code you want to use.

How it works

Avoid top-level

You should not do a commit at the top-level of a module, unless it's guarded by `if __name__ == '__main__'`. Better yet is to use a entry point as described in *Organizing your Project*. Doing a commit at module top-level can cause the commit to happen before you are done importing all required modules that contain Morepath directives, which would leave configuration in a half-baked state.

The same rule applies to starting the WSGI server.

Morepath needs to run the necessary configuration steps before it can serve WSGI requests. You can do this explicitly by running *`morepath.App.commit()`*:

```
if __name__ == '__main__':
    App.commit()

    application = App()
    morepath.run(application)
```

When you import modules, Morepath registers any directive you used in modules that you have imported, directly or indirectly, with the `App` subclass you used it on.

Calling `commit` on the App class then commits that app class and any app classes it mounts. After this, the application can be run. The commit procedure makes sure there are no conflicting pieces of configuration and resolves any

configuration overrides.

You can actually omit `App.commit()` if you want to. In this case the first request served by Morepath also does the commit. This also means any configuration errors are reported during the first request. If you prefer seeing configuration errors immediately during startup, leave the explicit `commit` in place.

Scanning a package

When you depend on a package that contains Morepath code it is convenient to be able to recursively import all of it at once. That way you can't accidentally forget to import a module and thus have its directives not be active. You can scan a whole package with *morepath.scan()*:

```
import my_package

if __name__ == '__main__':
    morepath.scan(my_package)

    App.commit()

    application = App()
    morepath.run(application)
```

All scanning does is recursively import all modules in a package (except for tests directories), nothing more.

Since scanning the current package is common, we have a convenience shortcut that scan the package the code is in automatically. You use it by calling *morepath.scan()* without arguments:

```
if __name__ == '__main__':
    morepath.scan()

    App.commit()

    application = App()
    morepath.run(application)
```

You can also use `scan()` with packages that contain third-party Morepath code, but there is an easier way to do that.

Scanning dependencies

Morepath is a micro-framework at its core, but you can expand it with other packages that add extra functionality. For instance, you can use more.chameleon for templating or more.transaction for SQLAlchemy integration.

These packages contain their own Morepath configuration, so when we use these packages we need to make sure to scan them too.

Manual scan

The most explicit way of scanning your dependencies is a manual scan.

Say you depend on more.jinja2 and you want to extend the the first example.

This is what you do:

```
import more.jinja2

if __name__ == '__main__':
    morepath.scan(more.jinja2) # scan Jinja2 package
    morepath.scan() # scan this package

    App.commit()

    application = App()
    morepath.run(application)
```

As you can see, you need to import your dependency and scan it using `scan()`. If you have more dependencies, just add them in this fashion.

Automatic scan

Scanning versus activation

Automatically configuring all packages that have Morepath configuration in them may seem too aggressive: what if you don't want to use this configuration? This is not a problem as Morepath makes a distinction between scanned configuration and activated configuration.

Configuration is only activated if it's on the *morepath.App* subclass you actually run as a WSGI app, or on any app class that your application class inherits from. App classes that you don't use are not active. It is therefore safe for Morepath to just scan everything that is available.

Manual scanning can get tedious and error-prone as you need to add each and every new dependency that you rely on.

You can use `autoscan()` instead, which scans all packages that have a dependency on Morepath declared. Let's look at a modified example that uses `autoscan`:

```
if __name__ == '__main__':
    morepath.autoscan()
    morepath.scan()

    App.commit()

    application = App()
    morepath.run(application)
```

As you can see, we also don't need to import or scan dependencies anymore. We still need to run `scan()` without parameters however, so our own package or module gets scanned.

If you move your code into a proper Python project that depends on Morepath you can also get rid of the `morepath.scan()` line by itself. The `setup.py` of your project then looks like this:

```
setup(name='myapp',
      packages=find_packages(),
      install_requires=[
          'more.jinja2',
          'morepath'
      ])
```

with the code in a Python package called `myapp` (a directory with an `__init__.py` file in it).

See *Organizing your Project* for a lot more information on how to do this, including tips on how to best organize your Python code.

Once you put your code in a Python project with a `setup.py`, you can simplify the setup code to this:

```
if __name__ == '__main__':
    morepath.autoscan()
    App.commit()
    morepath.run(App())
```

`morepath.autoscan()` makes sure to scan all packages that depend on Morepath directly or indirectly.

Writing scannable packages

A Morepath scannable Python package has to fulfill a few requirements.

1. The package must be made available using a `setup.py` file.

   See *Organizing your Project* and the Setuptool's documentation for more information.

2. The package itself or a dependency of the package must include `morepath` in the `install_requires` list of the `setup.py` file.

   Morepath only scans package that depend directly or indirectly on Morepath. It filters out packages which in no way depend on Morepath. So if your package has any Morepath configuration, you need to add `morepath` to `install_requires`:

   ```
   setup(name='myapp'
     ...
     install_requires=[
       'morepath'
     ])
   ```

   If you set up your dependencies up correctly using `install_requires` this should be there anyway, or be a dependency of another dependency that's in `install_requires`. Morepath just uses this information to do its scan.

3. The Python project name in `setup.py` should have the same name as the Python package name, *or* you use entry points to declare what should be scanned.

   Scan using naming convention:

   > The project name defined by `setup.py` can be imported in Python as well: they have the same name. For example: if the project name is `myapp`, the package that contains your code must be named `myapp` as well. (not `my-app` or `MyApp` or `Elephant`):

   So if you have a `setup.py` like this:

   ```
   setup(
     name='myapp',
     packages=find_packages(),
     ...
   ```

   you should have a project directory structure like this:

   ```
   setup.py
   myapp
     __init__.py
     another_module.py
   ```

In other words, the project name `myapp` can be imported:

```
import myapp
```

If you use a namespace package, you include the full name in the `setup.py`:

```
setup(
  name='my.app'
  packages=find_packages()
  namespace_packages=['my']
  ...
```

This works with a project structure like this:

```
setup.py
my
  __init__.py
  app
    __init__.py
    another_module.py
```

We recommend you use this naming convention as your Python projects get a consistent layout. But you don't have to – you can use entry points too.

Scan entry points:

If for some reason you want a project name that is different from the package name you can still get it scanned automatically by Morepath. In this case you need to explicitly tell Morepath what to scan with an entry point in `setup.py`:

```
setup(name='elephant'
   ...
   entry_points={
       'morepath': [
           'scan = my.package'
       ]
   }
```

Note that you still need to have `morepath` in the `install_requires` list for this to work.

CHAPTER 13

JSON and validation

Introduction

Morepath lets you define a JSON representations for arbitrary Python objects. When you return such an object from a json view, the object is automatically converted to JSON.

When JSON comes in as the POST or PUT body of the request, you can define how it is to be converted to a Python object and how it is to be validated.

This feature lets you plug in external (de)serialization libraries, such as Marshmallow. We've provided Marshmallow integration for Morepath in more.marshmallow

dump_json

The *morepath.App.dump_json()* directive lets you define a function that turns a model of a particular class into JSON. Here we define it for an `Item` class:

```
class Item(object):
   def __init__(self, value):
       self.value = value

@App.dump_json(model=Item)
def dump_item_json(self, request):
    return { 'type': 'Item', 'x': self.value }
```

So for instance, `Item('foo')` is represented in JSON as:

```
{
  'type': 'Item',
  'x': 'foo'
}
```

If we omit the `model` argument from the directive, we define a general dump_json function that applies to all objects.

Now we can write a JSON view that just returns an `Item` instance:

```
@App.json(model=Item)
def item_default(self, request):
    return self
```

The `self` we return in this view is an istance of `Item`. This is now automatically converted to a JSON object.

load function for views

When you specify the `load` function in a view directive you can specify how to turn the request body for a POST or PUT method into a Python object for that view. This Python object comes in as the third argument to your view function:

```
def my_load(request):
    return request.json

@App.json(model=Item, request_method='POST', load=my_load)
def item_post(self, request, obj):
   # the third obj argument contains the result of my_load(request)
```

The `load` function takes the request and must return some Python object (such as a simple `dict`). If the data supplied in the request body is incorrect and cannot be converted into a Python object then you should raise an exception. This can be a webob exception (we suggest `webob.exc.HTTPUnprocessableEntity`), but you could also define your own custom exception and provide a view for it that sets the status to 422. This way conversion and validation errors are reported to the end user.

CHAPTER 14

Security

Introduction

The security infrastructure in Morepath helps you make sure that web resources published by your application are only accessible by those persons that are allowed to do so. If a person is not allowed access, they will get an appropriate HTTP error: HTTP Forbidden 403.

Identity

Using settings in the identity policy

The function decorated by the `@App.identity_policy` decorator takes an optional settings argument, which provides access to the App settings. So if you define some settings for the identity policy you can pass them in like this:

```
@App.setting_section(section="policy")
def get_policy_settings():
    return {'encryption_key': 'secret'}

@App.identity_policy()
def get_identity_policy(settings):
    policy_settings = settings.policy.__dict__.copy()
    return CustomIdentityPolicy(**policy_settings)
```

Before we can determine who is allowed to do what, we need to be able to identify who people are in the first place.

The identity policy in Morepath takes a HTTP request and establishes a claimed identity for it. These are some extensions that provide an identity policy:

more.jwtauth Token based authentication system using JSON Web Token (JWT).

more.itsdangerous Cookie based identity policy using isdangerous.

more.basicauth Identity policy based on the HTTP Basic Authentication.

Choose the one of your choice, install it and follow the instructions in the README. You can also create your own identity policy.

For basic authentication for instance it will extract the username and password. The claimed identity can be accessed by looking at the *morepath.Request.identity* attribute on the request object.

You use the *morepath.App.identity_policy()* directive to install an identity policy into a Morepath app:

```
from more.basicauth import BasicAuthIdentityPolicy

@App.identity_policy()
def get_identity_policy():
    return BasicAuthIdentityPolicy()
```

If you want to create your own identity policy, see the *morepath.IdentityPolicy* API documentation to see what methods you need to implement.

Verify identity

The identity policy only establishes who someone is *claimed* to be. It doesn't verify whether that person is actually who they say they are. For identity policies where the browser repeatedly sends the username/password combination to the server, such as with basic authentication, implemented by more.basicauth and cookie-based authentication like more.itsdangerous, we need to check each time whether the claimed identity is actually a real identity.

By default, Morepath will reject any claimed identities. To let your application verify identities, you need to use *morepath.App.verify_identity()*:

```
@App.verify_identity()
def verify_identity(identity):
    return user_has_password(identity.username, identity.password)
```

The `identity` object received here is as established by the identity policy. What the attributes of the identity object are (besides `username`) is also determined by the specific identity policy you install.

Note that `user_has_password` stands in for whatever method you use to check a user's password; it's not part of Morepath.

Session or token based identity verification

If you use an identity policy based on the session (which you've made secure otherwise), or on a cryptographic token based authentication system such as the one implemented by more.jwtauth, the claimed identity is actually enough.

We know that the claimed identity is actually the one given to the user earlier when they logged in. No database-based identity check is required to establish that this is a legitimate identity. You can therefore implement `verify_identity` like this:

```
@App.verify_identity()
def verify_identity(identity):
    # trust the identity established by the identity policy
    return True
```

Login and logout

So now we know how identity gets established, and how it can be verified. We haven't discussed yet how a user actually logs in to establish an identity in the first place.

For this, we need two things:

- Some kind of login form. Could be taken care of by client-side code or by a server-side view. We leave this as an exercise for the reader.
- The view that the login data is submitted to when the user tries to log in.

How this works in detail is up to your application. What's common to login systems is the action we take when the user logs in, and the action we take when the user logs out. When the user logs in we need to *remember* their identity on the response, and when the user logs out we need to *forget* their identity again.

Here is a sketch of how logging in works. Imagine we're in a Morepath view where we've already retrieved `username` and `password` from the request (coming from a login form):

```
# check whether user has password, using password hash and database
if not user_has_password(username, password):
    return "Sorry, login failed" # or something more fancy

# now that we've established the user, remember it on the response
@request.after
def remember(response):
    identity = morepath.Identity(username)
    request.app.remember_identity(response, request, identity)
```

This is enough for session-based or cryptographic token-based authentication.

For cookie-based authentication where the password is sent as a cookie to the server for each request, we need to make sure to include the password the user used to log in, so that `remember` can then place it in the cookie so that it can be sent back to the server:

```
@request.after
def remember(response):
    identity = morepath.Identity(username, password=password)
    request.app.remember_identity(response, request, identity)
```

When you construct the identity using *morepath.Identity*, you can include any data you want in the identity object by using keyword parameters.

Logging out

Logging out is easy to implement and will work for any kind of authentication except for basic auth. You simply call *morepath.App.forget_identity()* somewhere in the logout view:

```
@request.after
def forget(response):
    request.app.forget_identity(response, request)
```

This will cause the login information (in cookie-form) to be removed from the response.

Permissions

Now that we have a way to establish identity and a way for the user to log in, we can move on to permissions. Permissions are per view. You can define rules for your application that determine when a user has a permission.

Let's say we want two permissions in our application, view and edit. We define those as plain Python classes:

```
class ViewPermission(object):
    pass

class EditPermission(object):
    pass
```

Permission Hierarchy

Since permissions are classes they could inherit from each other and form some kind of permission hierarchy, but we'll keep things simple here. Often a flat permission hierarchy is just fine.

Now we can protect views with those permissions. Let's say we have a `Document` model that we can view and edit:

```
@App.html(model=Document, permission=ViewPermission)
def document_view(request, model):
    return "<p>The title is: %s</p>" % model.title

@App.html(model=Document, name='edit', permission=EditPermission)
def document_edit(request, model):
    return "some kind of edit form"
```

This says:

- Only allow access to `document_view` if the identity has `ViewPermission` on the `Document` model.
- Only allow allow access to `document_edit` if the identity has `EditPermission` on the `Document` model.

Permission rules

Now that we give people a claimed identity and we have guarded our views with permissions, we need to establish who has what permissions where using some rules. We can use the *morepath.App.permission_rule()* directive to do that.

This is very flexible. Let's look at some examples.

Let's give absolutely everybody view permission on `Document`:

```
@App.permission_rule(model=Document, permission=ViewPermission)
def document_view_permission(identity, model, permission)
    return True
```

Let's give only those users that are in a list `allowed_users` on the `Document` the edit permission:

```
@App.permission_rule(model=Document, permission=EditPermission)
def document_edit_permission(identity, model, permission):
    return identity.userid in model.allowed_users
```

This is just is one hypothetical rule. `allowed_users` on `Document` objects is totally made up and not part of Morepath. Your application can have any rule at all, using any data, to determine whether someone has a permission.

Morepath Super Powers Go!

What if we don't want to have to define permissions on a per-model basis? In our application, we may have a *generic* way to check for the edit permission on any kind of model. We can easily do that too, as Morepath knows about inheritance:

```
@App.permission_rule(model=object, permission=EditPermission)
def has_edit_permission(identity, model, permission):
    ... some generic rule ...
```

This permission function is registered for model `object`, so will be valid for *all* models in our application.

What if we want that policy for all models, except `Document` where we want to do something else? We can do that too:

```
@App.permission_rule(model=Document, permission=EditPermission)
def document_edit_permission(identity, model, permission):
    ... some special rule ...
```

You can also register special rules that depend on identity. If you pass `identity=None`, you can can register a permission policy for when the user has not logged in yet and has no claimed identity:

```
@App.permission_rule(model=object, permission=EditPermission, identity=None)
def has_edit_permission_not_logged_in(identity, model, permission):
    return False
```

This permission check works in addition to the ones we specified above.

If you want to defer to a completely generic permission engine, you could define a permission check that works for *any* permission:

```
@App.permission_rule(model=object, permission=object)
def generic_permission_check(identity, model, permission):
     ... generic rule ...
```

CHAPTER 15

Settings

Introduction

A typical application has some settings: if an application logs, a setting is the path to the log file. If an application sends email, there are settings to control how email is sent, such as the email address of the sender.

Applications that serve as frameworks for other applications may have settings as well: the `transaction_app` defined by more.transaction for instance has settings controlling transactional behavior.

Morepath has a powerful settings system that lets you define what settings are available in your application and framework. It allows an app that extends another app to override settings. This lets an app that defines a framework can also define default settings that can be overridden by the extending application if needed.

Defining a setting

You can define a setting using the `App.setting()` directive:

```
@App.setting(section="logging", name="logfile")
def get_logfile():
    return "/path/to/logfile.log"
```

You can also use this directive to override a setting in another app:

```
class Sub(App):
    pass

@Sub.setting(section="logging", name="logfile")
def get_logfile_too():
   return "/a/different/logfile.log"
```

Settings are grouped logically: a setting is in a *section* and has a *name*. This way you can organize all settings that deal with logging under the `logging` section.

Accessing a setting

During runtime, you can access the settings of the current application using the *morepath.App.settings* property:

```
app.settings.logging.logfile
```

Remember that the current application is also accessible from the request object:

```
request.app.settings.logging.logfile
```

Defining multiple settings

It can be convenient to define multiple settings in a section at once. You can do this using the `App.setting_section()` directive:

```
@App.setting_section(section="logging")
def get_setting_section():
    return {
       'logfile': "/path/to/logfile.log",
       'loglevel': logging.WARNING
    }
```

You can mix `setting` and `setting_section` freely, but you cannot define a setting multiple times in the same app, as this will result in a configuration conflict.

Loading settings from a config file

For loading settings from a config file just load the file into a python dictionary and pre-fill the settings with *morepath.App.init_settings()* before committing the app.

A example config file with YAML syntax could look like:

```
# Config file for Morepath in YAML format

chameleon:
  debug: true

jinja2:
  auto_reload: false
  autoescape: true
  extensions:
    - jinja2.ext.autoescape
    - jinja2.ext.i18n

jwtauth:
  algorithm: ES256
  leeway: 20
  public_key:
    "MIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQBWcJwPEAnS/k4kFgUhxNF7J0SQQhZG+nNgy\
    +/mXwhQ5PZIUmId1a1TjkNXiKzv6DpttBqduHbz/V0EtH+QfWy0B4BhZ5MnTyDGjcz1DQqK\
    dexebhzobbhSIZjpYd5aU48o9rXp/OnAnrajddpGsJ0bNf4rtMLBqFYJN6LOslAB7xTBRg="
```

```
sqlalchemy:
  url: 'sqlite:///morepath.db'

transaction:
  attempts: 2
```

You can load it with:

```
import yaml

with open('settings.yml') as config:
     settings_dict = yaml.load(config)
```

Remember to install `pyyaml` before importing `yaml`. For example with:

```
$ pip install pyyaml
```

The same config file with JSON syntax would look like:

```
{
    "chameleon": {
        "debug": true
    },
    "jinja2": {
        "auto_reload": false,
        "autoescape": true,
        "extensions": [
            "jinja2.ext.autoescape",
            "jinja2.ext.i18n"
        ]
    },
    "jwtauth": {
        "algorithm": "ES256",
        "leeway": 20,
        "public_key": "MIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQBWcJwPEAnS/
↪k4kFgUhxNF7J0SQQhZG+nNgy+/mXwhQ5PZIUmId1a1TjkNXiKzv6DpttBqduHbz/
↪V0EtH+QfWy0B4BhZ5MnTyDGjcz1DQqKdexebhzobbhSIZjpYd5aU48o9rXp/
↪OnAnrajddpGsJ0bNf4rtMLBqFYJN6LOslAB7xTBRg="
    },
    "sqlalchemy": {
        "url": "sqlite:///morepath.db"
    },
    "transaction": {
        "attempts": 2
    }
}
```

To load it use:

```
import json

with open('settings.json') as config:
     settings_dict = json.load(config)
```

Now register the settings dictionary in the App settings before starting the App:

```
App.init_settings(settings_dict)
morepath.commit(App)
```

```
app = App()
```

You can access the settings as before:

```
>>> app.settings.jinja2.extensions
['jinja2.ext.autoescape', 'jinja2.ext.i18n']

>>> app.settings.jwtauth.algorithm
'ES256'

>>> app.settings.sqlalchemy.url
'sqlite:///morepath.db'
```

You can also override and extend the settings by loading a config file in an extending app as usual.

CHAPTER 16

Logging

Directive logging

Morepath has support for logging directive execution. This can be helpful when debugging why your Morepath application does not do what was expected. Morepath's directive logging makes use of Python's logging module, which is very flexible.

To get the complete log of directive executions, you can set up the following code in your project:

```
directive_logger = logging.getLogger('morepath.directive')
directive_logger.addHandler(logging.StreamHandler())
directive_logger.setLevel(logging.DEBUG)
```

The `StreamHandler` logs messages to `stderr`. You can reconfigure this or use another handler altogether. You need to change the log level so that `logging.DEBUG` level messages are also shown, as Morepath's directive logging uses this log level.

You can also configure it to just see the output for one particular directive. To see all `path` directive executed in your project you'd change the `getLogger` statement to this:

```
directive_logger = logging.getLogger('morepath.directive.path')
```

The Python logging module has many more options, but this should get you started.

CHAPTER 17

App Reuse

Morepath is a microframework with a difference: it's small and easy to learn like the others, but has special super powers under the hood.

One of those super powers is Reg, which along with Morepath's model/view separation makes it easy to write reusable views. But here we'll talk about another super power: Morepath's application reuse facilities.

We'll talk about how Morepath lets you isolate applications, extend and override applications, and compose applications together. Morepath makes this not only possible, but also *simple*.

Other web frameworks have mechanisms for overriding behavior and reusing code. But these were typically added in an ad-hoc fashion as new needs arose.

Morepath instead has *general* mechanisms for app extension and reuse. Any normal Morepath app is reusable without extra effort. Anything registered in a Morepath app can be overridden.

Application Isolation

Morepath lets you create app classes like this:

```
class App(morepath.App):
    pass
```

When you instantiate the app class, you get a WSGI application. The app class itself serves as a registry for application construction information. You specify this configuration with decorators. Apps consist of paths and views for models:

```
@App.path(model=User, path='users/{username}')
def get_user(username):
    return query_for_user(username)

@App.view(model=User)
def render_user(self, request):
    return "User: %s" % self.username
```

Here we've exposed the `User` model class under the path `/users/{username}`. When you go to such a URL, Morepath looks up the default (unnamed) view. We've implemented that too: it renders "User: {username}".

What now if we have another app where we want to publish `User` in a different way? No problem, we can create one:

```
class OtherApp(morepath.App):
    pass

@OtherApp.path(model=User, path='different_path/{username}')
def get_user(username):
    return different_query_for_user(username)

@OtherApp.view(model=User)
def render_user(self, request):
    return "Differently Displayed User: %s" % self.username
```

Here we expose `User` to the web again, but use a different path and a different view. If you use `OtherApp` (even in the same runtime), it functions independently from `App`.

App isolation is nothing special in Morepath; it's obvious that this is possible. But that's what we wanted. Let's look at some other features next.

Application Extension

Let's look at our first application `App` again. It exposes a single view for users (the default view). What now if we want to add a new functionality to this application so that we can edit users as well?

This is simple; we can add a new `edit` view to `App`:

```
@App.view(model=User, name='edit')
def edit_user(self, request):
    return 'Edit user: %s' % self.username
```

The string we return here is of course useless for a *real* edit view, but you get the idea.

But what if we have a scenario where there is a core application and we want to extend it *without modifying it*?

Why would this ever happen, you may ask? In complex applications and reuse scenarios it does. Imagine you have a common application core and you want to be able to plug into it. Meanwhile, you want that core application to still function as before when used (or tested!) by itself. Perhaps there's somebody else who has created another extension of it.

In software engineering we call this architectural principle the Open/Closed Principle, and Morepath makes it easy to follow it. What you do is create another app that subclasses the original:

```
class ExtendedApp(App):
    pass
```

And then we can add the view to the extended app:

```
@ExtendedApp.view(model=User, name='edit')
def edit_user(self, request):
    return 'Edit user: %s' % self.username
```

Now when we publish `ExtendedApp` using WSGI, the new `edit` view is there, but when we publish `App` it won't be.

Subclassing. Obvious, perhaps. Good! Let's move on.

Application Overrides

Now we get to a more exciting example: overriding applications. What if instead of adding an extension to a core application you want to override part of it? For instance, what if we want to change the default view for `User`?

Here's how we can do that:

```
@ExtendedApp.view(model=User)
def render_user_differently(self, request):
    return 'Different view for user: %s' % self.username
```

We've now overridden the default view for `User` to a new view that renders it differently.

We can also do this for model paths. Here we return a different user object altogether in our overriding app:

```
@ExtendedApp.path(model=OtherUser, path='users/{username}')
def get_user_differently(username):
    return OtherUser(username)
```

To publish `OtherUser` under `/users/{username}` it either needs to be a subclass of `User`. We've already registered a default view for that class. We can also register a new default view for `OtherUser`.

Overriding apps actually doesn't look much different from how you build apps in the first place. Again, it's just subclassing. Hopefully this isn't getting boring, so let's talk about something new.

Nesting Applications

Let's talk about application composition: nesting one app in another.

Imagine our user app allows users to have a wiki associated with them. It has paths like `/users/faassen/wiki/my_wiki_page` and `/users/bob/wiki/page_on_things`.

We could implement this directly in the user app along these lines:

```
def wiki_for_user(username):
    wiki_id = get_wiki_id_for_username(username)
    return get_wiki(wiki_id)

@App.path(model=WikiPage, path='users/{username}/wiki/{page_id}')
def get_wiki_page(username, page_id):
    return wiki_for_user(username).get_page(page_id)

@App.view(model=WikiPage)
def wiki_page_default(self, request):
    return "Wiki Page"
```

To understand this app, we need to describe a hypothetical `Wiki` class first. We can get an instance of it from some database by using `get_wiki` with a wiki id. It has a `get_page` method for getting access to wiki page objects (class `WikiPage`). We also have a way to determine the wiki id for a given username, `get_wiki_id_for_username`.

This application makes available wiki pages on a sub-URL for users, and then supplies a default view for them so we see something when we go to the page.

There are some issues with this implementation, though:

- Why would we implement a wiki as part of our user app? Our wiki application should really be an app by itself, that we can use by itself and also test by itself.

- The `username` appears in the path for the `WikiPage` model. The same would apply to any other wiki related models (like the wiki root). Why should we have to care about the username of a user when we expose a wiki page?
- Related to this, what if we wanted to associate a wiki app with some other object such as a *project*, instead of a user? It would be nice if we can use the wiki app in such other contexts as well, not just for users.

To deal with those issues, we can create a separate app for wikis that is only about wikis. So let's do it. Here's the wiki app by itself:

```
class WikiApp(morepath.App):
    def __init__(self, wiki_id):
        self.wiki_id = wiki_id

@WikiApp.path(path='{page_id}', model=WikiPage)
def get_wiki(page_id, app):
    return get_wiki(app.wiki_id).get_page(page_id)

@WikiApp.view(model=WikiPage)
def wiki_page_default(self, request):
    return "Wiki Page"
```

Here we have a stand-alone wiki app. It needs a `wiki_id` to be instantiated:

```
app = WikiApp(3)
```

We could now use `app` as a WSGI application, but that only works for one wiki id at the time. What if we want to associate the wiki with a user like we had before? We can accomplish this by *mounting* the wiki app into the user app, like this:

```
def variables(app):
    return dict(username=get_username_for_wiki_id(app.wiki_id))

@App.mount(app=WikiApp, path='users/{username}/wiki',
           variables=variables)
def mount_wiki(username):
    return WikiApp(get_wiki_id_for_username(username))
```

Note that in order to be able to link to `WikiApp` we need to supply a special `variables` function that takes the wiki app and returns the username for it. For more details, see the documentation for the *morepath.App.mount()* directive.

Linking to other mounted apps

Reusing views from other applications

Just like *morepath.Request.link()*, *morepath.Request.view()* also takes an `app` parameter. This allows you to reuse a view from another application.

Now that we have applications mounted into each other, we want a way to make links between them.

It is easy to make a link to an object in the same application. We use *morepath.Request.link()*:

```
wiki_page = get_wiki(3).get_page('my_page')

request.link(wiki_page)
```

This works to create links to wiki pages from within the wiki app. But what if we want to link to a wiki page from *outside* the wiki app, for instance from the user app?

To do this, we need not only the wiki page, but also a reference to the specific mounted application the wiki page is in. We can get this by navigating to it from the user app.

If we are in the user application, we can navigate to the mounted wiki app using the *morepath.App.child()* method:

```
wiki_app = request.app.child(WikiApp(3))
```

What if we want to navigate with the `username` under which it was mounted instead? We can do this too. We give `child` the `WikiApp` class and then the `username` as a keyword argument:

```
wiki_app = request.app.child(WikiApp, username='faassen')
```

There is one more alternative. We can also refer to `WikiApp` with the name under which it was mounted (the `path` by default):

```
wiki_app = request.app.child('users/{username}/wiki', username='faassen')
```

We can now use `wiki_app` to make the link from the username app to a wiki page in the wiki app:

```
request.link(wiki_page, app=wiki_app)
```

What if we wanted to create a link from the wiki app into the user app in which it was mounted? We get to the user app from the wiki app with *morepath.App.parent*:

```
request.link(User('faassen'), app=request.app.parent)
```

For a quick navigation to a sibling app, there is also *morepath.App.sibling()*. To quickly get to the root app, use *morepath.App.root*. You can also combine `parent` and `child` together to navigate the application tree.

Deferring links and views

If we have a lot of code that links to objects in another app, it can get cumbersome to have to add the `app` parameter whenever we want to create a view. Instead, we can declare this centrally with the *morepath.App.defer_links()* directive.

We can for instance declare for the `WikiApp` that to link to a `User` object we always use the parent app we were mounted in:

```
@WikiApp.defer_links(model=User)
def defer_user(app, obj):
    return app.parent
```

You can also use it to defer to a child app. If the `WikiPage` model provides a way to obtain the `wiki_id` for it, we can use that information to determine what mounted `WikiApp` we need to link to:

```
@App.defer_links(model=WikiPage)
def defer_wiki_page(app, obj):
    return app.child(WikiApp(obj.wiki_id))
```

You can defer links across multiple applications – a wiki app may defer objects it does not know how to link to to the app it is mounted to, and then this app could defer to another sub-app. When creating a link Morepath follows the defers to the application that knows how to do it.

The *morepath.App.defer_links()* directive also affects the behavior of *morepath.Request.view()* in the same way. It does however *not* affect *morepath.Request.class_link()*, as without the instance, insufficient information is available to defer the link.

Further reading

To see an extended example of how you can structure larger applications to support reuse, see *Building Large Applications*.

CHAPTER 18

Tweens

Introduction

Tweens are a light-weight framework component that sits between the web server and the app. It's very similar to a WSGI middleware, except that a tween has access to the Morepath API and is therefore less low-level.

Tweens can be used to implement transaction handling, logging, error handling and the like.

signature of a handler

Morepath has an internal *publish* function that takes a single *morepath.Request* argument, and returns a *morepath.Response* as a result:

```
def publish(request):
    ...
    return response
```

Tweens have the same signature.

We call such functions *handlers*.

Under and over

Given a handler, we can create a factory that creates a tween that wraps around it:

```
def make_tween(app, handler):
    def my_tween(request):
        print "Enter"
        response = handler(request)
        print "Exit"
```

```
        return response
    return my_tween
```

We say that *my_tween* is *over* the `handler` argument, and conversely that `handler` is *under* `my_tween`.

The application constructs a chain of tween over tween, ultimately reaching the request handler. Requests arrive in the outermost tween and descend down the chain into the underlying tweens, and finally into the Morepath *publish* handler itself.

What can a tween do?

A tween can:

- amend or replace the request before it goes in to the handler under it.
- amend or replace the response before it goes back out to the handler over it.
- inspect the request and completely take over response generation for some requests.
- catch and handle exceptions raised by the handler under it.
- do things before and after the request is handled: this can be logging, or commit or abort a database transaction.

Creating a tween factory

To have a tween, we need to add a tween factory to the app. The tween factory is a function that given a handler constructs a tween. You can register a tween factory using the `App.tween_factory()` directive:

```
@App.tween_factory()
def make_tween(app, handler):
    def my_tween(request):
        print "Enter"
        response = handler(request)
        print "Exit"
        return response
    return my_tween
```

The tween chain is now:

```
my_tween -> publish
```

It can be useful to control the order of the tween chain. You can do this by passing `under` or `over` to *tween_factory*:

```
@App.tween_factory(over=make_tween)
def make_another_tween(app, handler):
    def another_tween(request):
        print "Another"
        return handler(request)
    return another_tween
```

The tween chain is now:

```
another_tween -> my_tween -> publish
```

If instead you used `under`:

```
@App.tween_factory(under=make_tween)
def make_another_tween(app, handler):
    def another_tween(request):
        print "Another"
        return handler(request)
    return another_tween
```

Then the tween chain is:

```
my_tween -> another_tween -> publish
```

Tweens and settings

A tween factory may need access to some application settings in order to construct its tweens. A logging tween for instance needs access to a setting that indicates the path of the logfile.

The tween factory gets two arguments: the app and the handler. You can then access the app's settings using `app.registry.settings`. See also the *Settings* section.

Tweens and apps

You can register different tween factories in different Morepath apps. A tween factory only has an effect when the app under which it is registered is being run directly as a WSGI app. A tween factory has no effect if its app is mounted under another app. Only the tweens of the outer app are in effect at that point, and they are *also* in effect for any apps mounted into it.

This means that if you install a logging tween in an app, and you run this app with a WSGI server, the logging takes place for that app and any other app that may be mounted into it, directly or indirectly.

more.transaction

If you need to integrate SQLAlchemy or the ZODB into Morepath, Morepath offers a special app you can extend that includes a transaction tween that interfaces with the transaction package. The morepath_sqlalchemy demo project gives an example of what that looks like with SQLAlchemy.

CHAPTER 19

Static resources with Morepath

Introduction

A modern client-side web application is built around JavaScript and CSS. A web server is responsible for serving these and other types of static content such as images to the client.

Morepath does not include in itself a way to serve these static resources. Instead it leaves the task to other WSGI components you can integrate with the Morepath WSGI component. Examples of such systems that can be integrated through WSGI are BowerStatic, Fanstatic, Webassets, and webob.static.

Examples will focus on BowerStatic integration to demonstrate a method for serving JavaScript and CSS. To demonstrate a method for serving other static resources such as an image we will use webob.static.

We recommend you read the BowerStatic documentation, but we provide a small example of how to integrate it here that should help you get started. You can find all the example code in the github repo.

Application layout

To integrate BowerStatic with Morepath we can use the more.static extension.

First we need to include `more.static` as a dependency of our code in `setup.py`. Once it is installed, we can create a Morepath application that subclasses from `more.static.StaticApp` to get its functionality:

```
from more.static import StaticApp

class App(StaticApp):
    pass
```

We give it a simple HTML page on the root HTML that contains a `<head>` section in its HTML:

```
@App.path(path='/')
class Root(object):
    pass
```

```
@App.html(model=Root)
def root_default(self, request):
    return ("<!DOCTYPE html><html><head></head><body>"
            "jquery is inserted in the HTML source</body></html>")
```

It's important to use `@App.html` as opposed to `@App.view`, as that sets the content-header to `text/html`, something that BowerStatic checks before it inserts any `<link>` or `<script>` tags. It's also important to include a `<head>` section, as that's where BowerStatic includes the static resources by default.

The app configuration code we store in the `app.py` module of the Python package.

In the `run.py` module of the Python package we set up a `run()` function that when run serves the WSGI application to the web:

```
from .app import App

def run():
   morepath.autoscan()
   App.commit()
   wsgi = App()
   morepath.run(wsgi)
```

Manual scan

We recommend you use `morepath.autoscan` to make sure that all code that uses Morepath is automatically scanned. If you *do not* use `autoscan` but use manual `morepath.scan()` instead, you need to scan `more.static` explicitly, like this:

```
import more.static

def run():
   morepath.scan(more.static)
   App.commit()
   wsgi = App()
   morepath.run(wsgi)
```

Bower

BowerStatic integrates the Bower JavaScript package manager with a Python WSGI application such as Morepath.

Once you have `bower` installed, go to your Python package directory (where the `app.py` lives), and install a Bower component. Let's take `jquery`:

```
bower install jquery
```

You should now see a `bower_components` subdirectory in your Python package. We placed it here so that when we distribute the Python package that contains our application, the needed bower components are automatically included in the package archive. You could place `bower_components` elsewhere however and manage its contents separately.

Registering bower_components

BowerStatic needs a single global bower object that you can register multiple bower_components directories against. Let's create it first:

```
bower = bowerstatic.Bower()
```

We now tell that bower object about our bower_component directory:

```
components = bower.components(
  'app', os.path.join(os.path.dirname(__file__), 'bower_components'))
```

The first argument to bower.components is the name under which we want to publish them. We just pick app. The second argument specifies the path to the bower.components directory. The os.path business here is a way to make sure that we get the bower_components next to this module (app.py) in this Python package.

BowerStatic now lets you refer to files in the packages in bower_components to include them on the web, and also makes sure they are available.

Saying which components to use

We now need to tell our application to use the components object. This causes it to look for static resources only in the components installed there. We do this using the @App.static_components directive, like this:

```
@App.static_components()
def get_static_components():
    return components
```

You could have another application that use another components object, or share this components with the other application. Each app can only have a single components registered to it, though.

The static_components directive is not part of standard Morepath. Instead it is part of the more.static extension, which we enabled before by subclassing from StaticApp.

Including stuff

Now we are ready to include static resources from bower_components into our application. We can do this using the include() method on request. We modify our view to add an include() call:

```
@App.html(model=Root)
def root_default(self, request):
    request.include('jquery')
    return ("<!DOCTYPE html><html><head></head><body>"
            "jquery is inserted in the HTML source</body></html>")
```

When we now open the view in our web browser and check its source, we can see it includes the jquery we installed in bower_components.

Note that just like the static_components directive, the include() method is not part of standard Morepath, but has been installed by the more.static.StaticApp base class as well.

Local components

In many projects we want to develop our *own* client-side JS or CSS code, not just rely on other people's code. We can do this by using local components. First we need to wrap the existing `components` in an object that allows us to add local ones:

```
local = bower.local_components('local', components)
```

We can now add our own local components. A local component is a directory that needs a `bower.json` in it. You can create a `bower.json` file most easily by going into the directory and using `bower init` command:

```
$ mkdir my_component
$ cd my_component
$ bower init
```

You can edit the generated `bower.json` further, for instance to specify dependencies. You now have a bower component. You can add any static files you are developing into this directory.

Now you need to tell the local components object about it:

```
local.component('/path/to/my_component', version=None)
```

See the BowerStatic local component documentation for more of what you can do with `version` – it's clever about automatically busting the cache when you change things.

You need to tell your application that instead of plain `components` you want to use `local` instead, so we modify our `static_components` directive:

```
@App.static_components()
def get_static_components():
    return local
```

When you now use `request.include()`, you can include local components by their name (as in `bower.json`) as well:

```
request.include('my_component')
```

It automatically pulls in any dependencies declared in `bower.json` too.

As mentioned before, check the `morepath_static` github repo for the complete example.

A note about mounted applications

`more.static` uses a tween to inject scripts into the response (see *Tweens*). If you use `more.static` in a view in a mounted application, you need to make sure that the root application also derives from `more.static.StaticApp`, otherwise the resources aren't inserted correctly:

```
from more.static import StaticApp

class App(StaticApp):  # this needs to subclass StaticApp too
    pass

class Mounted(StaticApp):
    pass

 @App.mount(app=Mounted, path='mounted')
```

```
def mount():
    return Mounted()
```

Other static content

In essence, Morepath doesn't enforce any particular method for serving static content to the client as long as the content eventually ends up in the response object returned. Therefore, there are different approaches to serving static content.

Since a Morepath view returns a WebOb response object, that object can be loaded with any type of binary content in the body along with the necessary HTTP headers to describe the content type and size.

In this example, we use a WebOb helper class webob.static.FileApp to serve a PNG image:

```
from webob import static

@App.path(path='')
class Image(object):
    path = 'image.png'

@App.view(model=Image)
def view_image(self, request):
    return request.get_response(static.FileApp(self.path))
```

In the above example FileApp does the heavy lifting by opening the file, guessing the MIME type, updating the headers, and returning the response object which is in-turn returned by the Morepath view. Note that the same helper class can be used to to serve most types of `MIME` content.

This example is one way to serve an image, but it is not the only way. In cases that require a more elaborate method for serving the content this WebOb File-Serving Example may be helpful.

Part III

Advanced Topics

A selection of special topics to get the best out of your Morepath project.

CHAPTER 20

Organizing your Project

Introduction

Morepath does not put any requirements on how your Python code is organized. You can organize your Python project as you see fit and put app classes, paths, views, etc, anywhere you like. A single Python package (or even module) may define a single Morepath app, but could also define multiple apps. In this Morepath is like Python itself; the Python language does not restrict you in how you organize functions and classes.

While this leaves you free to organize your code as you see fit, that doesn't mean that your code shouldn't be organized. Here are some guidelines on how you may want to organize things in your own project. But remember: these are guidelines to break when you see the need.

Sounds Like a Lot of Work

You're in luck. If you want to skip this chapter and just get started, you can use the Morepath cookiecutter template, which follows the guidelines layed out in this chapter:

https://github.com/morepath/morepath-cookiecutter

If you want to find out more about the why and the how, you can always keep on reading of course.

Python project

It is recommended you organize your code in a Python project with a `setup.py` where you declare the dependency on Morepath. If you're unfamiliar with how this works, you can check out this tutorial.

Doing this is good Python practice and makes it easy for you to install and distribute your project using common tools like pip, buildout and PyPI. In addition Morepath itself can also load its code more easily.

Project layout

Here's a quick overview of the files and directories of Morepath project that follows the guidelines in this document:

```
myproject
    setup.py
    myproject
         __init__.py
        app.py
        model.py
        [collection.py]
        path.py
        run.py
        view.py
```

Project setup

Here is an example of your project's `setup.py` with only those things relevant to Morepath shown and everything else cut out:

```
from setuptools import setup, find_packages

setup(name='myproject',
      packages=find_packages(),
      install_requires=[
          'morepath'
      ],
      entry_points={
          'console_scripts': [
           'myproject-start = myproject.run:run'
           ]
      })
```

This `setup.py` assumes you also have a `myproject` subdirectory in your project directory that is a Python package, i.e. it contains an `__init__.py`. This is the directory where you put your code. The `find_packages()` call finds it for you.

The `install_requires` section declares the dependency on Morepath. Doing this makes everybody who installs your project automatically also pull in a release of Morepath and its own dependencies. In addition, it lets this package be found and configured when you use *`morepath.autoscan()`*.

Finally there is an `entry_points` section that declares a console script (something you can run on the command-prompt of your operating system). When you install this project, a `myproject-start` script is automatically generated that you can use to start up the web server. It calls the `run()` function in the `myproject.run` module. Let's create this next.

You now need to install this project. If you want to install this project for development purposes you can use `python setup.py develop`, or `pip install -e .` from within a virtualenv.

See also the setuptools documentation.

Project naming

Its possible to name your project differently than you name your Python package; you could for instance have the name `ThisProject` in `setup.py`, and then have your Python package be still called `myproject`. We recommend naming the project the same as the Python package to avoid confusion.

Namespace packages

Sometimes you have projects that are grouped in some way: they are all created by the same organization or they are part of the same larger project. In that case you can use Python namespace packages to make this relationship clear. Let's say you have a larger project called `myproject`. The namespace package itself may not contain any code, so unlike the example everywhere else in this document the `myproject` directory is always empty but for a `__init__.py`.

Different sub-projects could then be called `myproject.core`, `myproject.wiki`, etc. Let's examine the files and directories of `myproject.core`:

```
myproject.core
    setup.py
    myproject
        __init__.py
        core
            __init__.py
            app.py
            model.py
            [collection.py]
            path.py
            run.py
            view.py
```

The change is the namespace package directory `myproject` that contains a single file, `__init__.py`, that contains the following code to declare it is a namespace package:

```
__import__('pkg_resources').declare_namespace(__name__)
```

Inside is the normal package called `core`.

`setup.py` is modified too to include a declaration in `namespace_packages`, and we've changed the entry point:

```
setup(name='myproject.core',
      packages=find_packages(),
      namespace_packages=['myproject'],
      install_requires=[
          'morepath'
      ],
      entry_points={
          'console_scripts': [
           'myproject.core-start = myproject.core.run:run'
           ]
      })
```

See also the namespace packages documentation.

App Module

The `app.py` module is where we define our Morepath app. Here's a sketch of `app.py`:

```
import morepath

class App(morepath.App):
    pass
```

Run Module

Why we keep app.py and run.py separate

Morepath attaches a configuration registry to each application class. This can happen twice if we run the run function directly from python (through use of `__main__`). By keeping the application from the run code we can be sure that this never happens.

In the `run.py` module we define how our application should be served. We take the `App` class defined in `app.py`, then have a `run()` function that is going to be called by the `myproject-start` entry point we defined in `setup.py`:

```
from .app import App

def run():
    morepath.autoscan()
    App.commit()
    morepath.run(App())
```

This run function does the following:

- Use *`morepath.autoscan()`* to recursively import your own package plus any dependencies that are installed.
- Commit the `App` class so that its configuration is ready. You can omit this step and in this case the configuration is committed when Morepath processes the first request. But if you want to see configuration errors at startup, use an explicit `commit`.
- start a WSGI server for the `App` instance on port localhost, port 5000. This uses the standard library wsgiref WSGI server. Note that this should only used for testing purposes, not production! For production, use an external WSGI server.

The run module is also a good place to do other general configuration for the application, such as setting up a database connection.

Upgrading your project to a newer version of Morepath

See *Upgrading to a new Morepath version*.

Debugging scanning problems

If you for some reason get `404 Not Found` errors where you expect some content, something may have gone wrong with scanning the configuration of your project. Here's a checklist:

- Check whether your project has a `setup.py` with an `install_requires` that depends on `morepath` (possibly indirectly through another dependency). You need to declare your code as a project so that `autoscan` can find it.
- Check whether your project is installed in a virtualenv using `pip install -e .` or in a buildout. Morepath needs to be able to find your project in order to scan it.
- Be sure that you have your modules in an actual sub-directory to the project with its own `__init__.py`. Modules in the top-level of a project won't be scanned as a package
- Try manually scanning a package and see whether it works then:

```
import mysterious_package

morepath.scan(mysterious_package)
```

  If this fixes things, the package is somehow not being picked up for automatic scanning. Check the package's `setup.py`.

- Try manually importing the modules before doing a *`morepath.autoscan()`* and see whether it works then:

```
import mysterious_module

morepath.autoscan()
```

  If this fixes things, then your own package is not being picked up as a Morepath package for some reason.

- Try moving Morepath directives into the module that also runs the application. If this works, your own package is not recognized as a proper Morepath package.

Variation: automatic restart

During development it can be very helpful to have the WSGI server restart the Morepath app whenever a file is changed.

Morepath's built in development server does not offer this feature, but you can accomplish it with Werkzeug's server.

First install the Werkzeug package into your project. Then modify your run module to look like this:

```
import morepath
from werkzeug.serving import run_simple
from .app import App

def run():
    morepath.autoscan()
    App.commit()
    run_simple('localhost', 8080, App(), use_reloader=True)
```

Using this runner changes to Python code in your package trigger a restart of the WSGI server.

Variation: no or multiple entry points

Not all packages have an entry point to start it up: a framework app that isn't intended to be run directly may not define one. Some packages may define multiple apps and multiple entry points.

Variation: waitress

Instead of using Morepath's simple built-in WSGI server you can use another WSGI server. The built-in WSGI server is only meant for testing, so we strongly recommend doing so in production. Here's how you'd use Waitress. First we adjust `setup.py` so we also require waitress:

```
...
      install_requires=[
         'morepath',
         'waitress'
      ],
...
```

Then we modify `run.py` to use waitress:

```
import waitress

...

def run():
   ...
   waitress.serve(App())
```

Variation: command-line WSGI servers

You could also do away with the entry point and instead use `waitress-serve` on the command line directly. For this we need to first create a factory function that returns the fully configured WSGI app:

```
def wsgi_factory():
   morepath.autoscan()
   App.commit()
   return App()

$ waitress-serve --call myproject.run:wsgi_factory
```

This uses waitress's `--call` functionality to invoke a WSGI factory instead of a WSGI function. If you want to use a WSGI function directly we have to create one using the `wsgi_factory` function we just defined. To avoid circular dependencies you should do it in a separate module that is only used for this purpose, say `wsgi.py`:

```
prepared_app = wsgi_factory()
```

You can then do:

```
$ waitress-serve myproject.wsgi:prepared_app
```

You can also use gunicorn this way:

```
$ gunicorn -w 4 myproject.wsgi:prepared_app
```

Model module

The `model.py` module is where we define the models relevant to the web application. They may integrate with some kind of database system, for instance the SQLAlchemy ORM. Note that your model code is completely independent from Morepath and there is no reason to import anything Morepath related into this module. Here is an example `model.py` that just uses plain Python classes:

```
class Document(object):
    def __init__(self, id, title, content):
        self.id = id
        self.title = title
        self.content = content
```

Variation: models elsewhere

Sometimes you don't want to include model definitions in the same codebase that also implements a web application, as you would like to reuse them outside of the web context without any dependencies on Morepath. Your model classes are independent from Morepath, so this is easy to do: just put them in a separate project and depend on it from your web project.

You can also have a project that reuses models defined by another Morepath project. Each Morepath app is isolated from the others by default, so you could remix its models into a whole new web application.

Variation: collection module

An application tends to contain two kinds of models:

- content object models, i.e. a Document. If you use an ORM like SQLAlchemy these would typically be backed by a table.
- collection models, i.e. a collection of documents. This typically let you browse content models, search/filter for them, and let you add or remove them.

Since collection models tend to not be backed by a database directly but are often application-specific classes, it can make sense to maintain them in a separate `collection.py` module. This module, like `model.py` also does not have any dependencies on Morepath.

Path module

Now that we have models, we need to publish them on the web. First we need to define their paths. We do this in a `path.py` module:

```
from .app import App
from . import model

@App.path(model=model.Document, path='documents/{id}')
def get_document(id):
   if id != 'foo':
      return None # not found
   return Document('foo', 'Foo document', 'FOO!')
```

In the functions decorated by `App.path()` we do whatever query is necessary to retrieve the model instance from a database, or return `None` if the model cannot be found.

Morepath allows you to scatter `@App.path` decorators throughout your codebase, but by putting them all together in a single module it becomes really easy to inspect and adjust the URL structure of your application, and to see exactly what is done to query or construct the model instances. Once it becomes really big you can always split a single path module into multiple ones, though at that point you may want to consider splitting off a separate project with its own application instead.

View module

We have models and they're published on a path. Now we need to represent them as actual web resources. We do this in the `view.py` module:

```
from .app import App
from . import model

@App.json(model=model.Document)
def document_default(self, request):
    return {'id': self.id, 'title': self.title, 'content': self.content }
```

Here we use `App.view()`, `App.json()` and `App.html()` directives to declare views.

By putting them all in a view module it becomes easy to inspect and adjust how models are represented, but of course if this becomes large it's easy to split it into multiple modules.

Directive debugging

Morepath's directive issue log messages that can help you debug your application: see *Logging* for more information.

CHAPTER 21

Building Large Applications

Introduction

A small web application is relatively easy to understand. It does less stuff. That makes the application easier to understand: the UI (or REST web service) is smaller, and the codebase too.

But sometimes we need larger web applications. Morepath offers a number of facilities to help you manage the complexity of larger web applications:

- Morepath lets you build larger applications from multiple smaller ones. A CMS may for instance be composed of a document management application and a user management application. This is much like how you manage complexity in a codebase by decomposing it into smaller functions and classes.
- Morepath lets you factor out common, reusable functionality. In other words, Morepath helps you build *frameworks*, not just end-user applications. For instance, you may have multiple places in an application where you need to represent a large result-set in smaller batches (with previous/next), and they should share common code.

There is also the case of reusable *applications*. Larger applications are often deployed multiple times. An open source CMS is a good example: different organizations each have their own installation. Or imagine a company with an application that it sells to its customers: each customer can have its own special deployment.

Different deployments of an application have real differences as every organization has different requirements. This means that you need to be able to customize and extend the application to fit the purposes of each particular deployment. As a result the application has to take on framework-like properties. Morepath recognizes that there is a large gray area between application and framework, and offers support to build framework-like applications and application-like frameworks.

The document *App Reuse* describes the basic facilities Morepath offers for application reuse. The document *Organizing your Project* describes how a single application project can be organized, and we will follow its guidelines in this document.

This document sketches out an example of a larger application that consists of multiple sub-projects and sub-apps, and that needs customization.

A Code Hosting Site

Our example large application is a code hosting site along the lines of Github or Bitbucket. This example is a sketch, not a complete working application. We focus on the structure of the application as opposed to the details of the UI.

Let's examine the URL structure of a code hosting site. Our hypothetical code hosting site lives on `example.com`:

```
example.com
```

A user (or organization) has a URL directly under the root with the user name or organization name included:

```
example.com/faassen
```

Under this URL we can find repositories, using the project name in the URL:

```
example.com/faassen/myproject
```

We can interact with repository settings on this URL:

```
example.com/faassen/myproject/settings
```

We also have a per-repository issue tracker:

```
example.com/faassen/myproject/issues
```

And a per-repository wiki:

```
example.com/faassen/myproject/wiki
```

Simplest approach

The simplest approach to make this URL structure work is to implement all paths in a single application, like this:

```
from .model import Root, User, Repository, Settings, Issues, Wiki

class App(morepath.App):
    pass

@App.path(path='', model=Root)
def get_root():
    ...

@App.path(path='{user_name}', model=User)
def get_user(user_name):
    ...

@App.path(path='{user_name}/{repository_name}', model=Repository)
def get_repository(user_name, repository_name):
    ...
```

We could try to implement settings, issues and wiki as views on repository, but these are complicated pieces of functionality that benefit from having sub-URLs (i.e. `issues/12` or `...wiki/mypage`), so we model them using paths as well:

```
@App.path(path='{user_name}/{repository_name}/settings', model=Settings)
def get_settings(user_name, repository_name):
    ...

@App.path(path='{user_name}/{repository_name}/issues', model=Issues)
def get_issues(user_name, repository_name):
    ...

@App.path(path='{user_name}/{repository_name}/wiki', model=Wiki)
def get_wiki(user_name, repository_name):
    ...
```

Let's also make a path to an individual issue, i.e. `example.com/faassen/myproject/issues/12`:

```
from .model import Issue

@App.path(path='{user_name}/{repository_name}/issues/{issue_id}', model=Issue)
def get_issue(user, repository, issue_id):
    ...
```

Problems

This approach works perfectly well, and it's often the right way to start, but there are some problems with it:

- The URL patterns in the path are repetitive; for each sub-model under the repository we keep having to repeat `{user_name}/{repository_name}`.
- We may want to be able to test the wiki or issue tracker during development without having to worry about setting up the whole outer application.
- We may want to reuse the wiki application elsewhere, or in multiple places in the same larger application. But `user_name` and `repository_name` are now hardcoded in the way to get any sub-path into the wiki.
- We could have different teams developing the core app and the wiki (and issue tracker, etc). It would be nice to partition the code so that the wiki developers don't need to look at the core app code and vice versa.
- You may want the abilitity to swap in new implementations of a issue tracker or a wiki under the same paths, without having to change a lot of code.

We're going to show how Morepath can solve these problems by partitioning a larger app into smaller ones, and mounting them.

The code to accomplish this is more involved than simply declaring all paths under a single core app as we did before. If you feel more comfortable doing that, by all means do so; you don't have these problems. But if your application is successful and grows larger you may encounter these problems, and these features are then there to help.

Multiple sub-apps

Let's split up the larger app into multiple sub apps. How many sub-apps do we need? We could go and partition things up into many sub-applications, but that risks getting lost in another kind of complexity. So let's start with three application:

- core app, everything up to repository, and including settings.
- issue tracker app.

- wiki sub app.

In code:

```
class CoreApp(morepath.App):
    pass

class IssuesApp(morepath.App):
    def __init__(self, issues_id):
        self.issues_id = issues_id

class WikiApp(morepath.App):
    def __init__(self, wiki_id):
        self.wiki_id = wiki_id
```

Note that `IssuesApp` and `WikiApp` expect arguments to be initialized; we'll learn more about this later.

We now can group our paths into three. First we have the core app, which includes the repository and its settings:

```
@CoreApp.path(path='', model=Root)
def get_root():
    ...

@CoreApp.path(path='{user_name}', model=User)
def get_user(user_name):
    ...

@CoreApp.path(path='{user_name}/{repository_name}', model=Repository)
def get_repository(user_name, repository_name):
    ...

@CoreApp.path(path='{user_name}/{repository_name}/settings', model=Settings)
def get_settings(user_name, repository_name):
    ...
```

Then we have the paths for our issue tracker:

```
@IssuesApp.path(path='', model=Issues)
def get_issues():
    ...

@IssuesApp.path(path='{issue_id}', model=Issue)
def get_issue(issue_id):
    ...
```

And the paths for our wiki:

```
@WikiApp.path(path='', model=Wiki)
def get_wiki():
    ...
```

We have drastically simplified the paths in `IssuesApp` and `WikiApp`; we don't deal with `user_name` and `repository_name` anymore.

Mounting apps

Now that we have an independent `IssuesApp` and `WikiApp`, we want to be able to mount these under the right URLs under `CoreApp`. We do this using the mount directive:

```
def variables(app):
    repository = get_repository_for_wiki_id(app.wiki_id)
    return dict(
          repository_name=repository.name,
          user_name=repository.user.name)

@CoreApp.mount(path='{user_name}/{repository_name}/issues',
               app=IssuesApp, variables=variables)
def mount_issues(user_name, repository_name):
    return IssuesApp(issues_id=get_issues_id(user_name, repository_name))
```

Let's look at what this does:

- `@CoreApp.mount`: We mount something onto `CoreApp`.
- `path='{user_name}/{repository_name}/issues'`: We are mounting it on that path. All sub-paths in the issue tracker app will fall under it.
- `app=IssuesApp`: We are mounting `IssuesApp`.
- The `mount_issues` function takes the path variables `user_name` and `repository_name` as arguments. It then returns an instance of the `IssuesApp`. To create one we need to convert the `user_name` and `repository_name` into an issues id. We do this by looking it up in some kind of database.
- The `variables` function needs to do the inverse: given a `WikiApp` instance it needs to translate this back into a `repository_name` and `user_name`. This allows Morepath to link to a mounted `WikiApp`.

Mounting the wiki is very similar:

```
def variables(app):
    return dict(user_name=get_username_for_wiki_id(app.id))

@CoreApp.mount(path='{user_name}/{repository_name}/wiki',
                app=WikiApp, variables=variables)
def mount_wiki(user_name, repository_name):
    return WikiApp(get_wiki_id(user_name, repository_name))
```

No more path repetition

We have solved the repetition of paths issue now; the issue tracker and wiki handle many paths, but there is no more need to repeat '{user_name}/{repository_name}' everywhere.

Testing in isolation

To test the issue tracker by itself, we can run it as a separate WSGI app:

```
def run_issue_tracker():
    mounted = IssuesApp(4)
    morepath.run(mounted)
```

Here we mount and run the `issues_app` with issue tracker id `4`.

You can hook the `run_issue_tracker` function up to a script by using an entry point in `setup.py` as we've seen in *Organizing your Project*.

You can also mount applications this way in automated tests and then use WebTest or some other WSGI testing library, as explained in *Writing automated tests*.

Reusing an app

We can now reuse the issue tracker app in the sense that we can mount it in different apps; all we need is a way to get `issues_id`. What then if we have another Python project and we wanted to reuse the issue tracker in it as well? In that case it may start sense to start maintaining the issue tracker it in a separate Python project of its own.

We could for instance split our code into three separate Python projects, for instance:

- `myproject.core`
- `myproject.issues`
- `myproject.wiki`

Each would be organized as described in *Organizing your Project*.

`myproject.core` could have an `install_requires` in its `setup.py` that depends on `myproject.issues` and `myproject.wiki`. To get `IssuesApp` and `WikiApp` in order to mount them in the core, we would simply import them (for instance in `myproject.core.app`):

```
from myproject.issues.app import IssuesApp
from myproject.wiki.app import WikiApp
```

In some scenarios you may want to turn this around: the `IssuesApp` and `WikiApp` know they should be mounted in `CoreApp`, but the `CoreApp` wants to remain innocent of this. In that case, you would have `myproject.issues` and `myproject.wiki` both depend on `myproject.core`, whereas `myproject.core` depends on nothing. The wiki and issues projects then mount themselves into the core app.

Different teams

Now that we have separate projects for the core, issue tracker and wiki, it becomes possible for a team to focus on the wiki without having to worry about core or the issue tracker and vice versa.

This may in fact be of benefit even when you alone are working on all three projects! When developing software it is important to free up your brain so you only have to worry about one detail at the time: this an important reason why we decomposition logic into functions and classes. By decomposing the project into three independent ones, you can temporarily forget about the core when you're working on the issue tracker, allowing you to focus on the problems at hand.

Swapping in a new sub-app

Perhaps a different, better wiki implementation is developed. Let's call it `ShinyNewWikiApp`. Swapping in the new sub application is easy: it's just a matter of changing the mount directive:

```
@CoreApp.mount(path='{user_name}/{repository_name}/wiki',
               app=ShinyNewWikiApp, variables=variables)
def mount_wiki(user_name, repository_name):
    return ShinyNewWikiApp(get_wiki_id(user_name, repository_name))
```

Customizing an app

Let's change gears and talk about customization now.

Imagine a scenario where a particular customer wants *exactly* core app. Really, it's perfect, exactly what they need, no change needed, but then ... wait for it ... they actually do need a minor tweak.

Let's say they want an extra view on `Repository` that shows some important customer-specific metadata. This metadata is retrieved from a customer-specific extra database, so we cannot just add it to core app. Besides, this new view isn't useful to other customers.

What we need to do is create a new customer specific core app in a separate project that is exactly like the original core app by extending it, but with the one extra view added. Let's call the project `important_customer.core`. `important_customer.core` has an `install_requires` in its `setup.py` that depends on `myproject.core` and also the customer database (which we call `customerdatabase` in this example).

Now we can import `CoreApp` in `important_customer.core`'s `app.py` module, and extend it:

```
from myproject.core.app import CoreApp

class CustomerApp(CoreApp):
    pass
```

At this point `CustomerApp` and `CoreApp` have identical behavior. We can now make our customization and add a new JSON view to `Repository`:

```
from myproject.core.model import Repository
# customer specific database
from customerdatabase import query_metadata

@CustomerApp.json(model=Repository, name='customer_metadata')
def repository_customer_metadata(self, request):
    metadata = query_metadata(self.id) # use repository id to find it
    return {
      'special_marketing_info': medata.marketing_info,
      'internal_description': metadata.description
    }
```

You can now run `CustomerApp` and get the core app with exactly the one tweak the customer wanted: a view with the extra metadata. The `important_customer.core` project depends on `customerdatabase`, but `myproject.core` remains unchanged.

We've made exactly the tweak necessary without having to modify our original project. The original project continues to work the same way it always did.

Swapping in, for one customer

Morepath lets you extend *any* directive, not just the `view` directive. It also lets you *override* things in the applications you extend. Let's say the important customer wants *exactly* the original wiki, with just one tiny teeny little tweak.

Other customers should still continue to use the original wiki.

We'd tweak the wiki just as we would tweak the core app. We end up with a `TweakedWikiApp`:

```
from myproject.wiki.app import WikiApp

class TweakedWikiApp(WikiApp):
     pass

# some kind of tweak
@TweakedWikiApp.json(model=WikiPage, name='extra_info')
def page_extra_info(self, request):
    ...
```

We want a new version of `CoreApp` just for this customer that mounts `TweakedWikiApp` instead of `WikiApp`:

```
class ImportantCustomerApp(CoreApp):
    pass

@ImportantCustomerApp.mount(path='{user_name}/{repository_name}/wiki',
                            app=TweakedWikiApp, variables=variables)
def mount_wiki(user_name, repository_name):
    return TweakedWikiApp(get_wiki_id(user_name, repository_name))
```

The `mount` directive above overrides the one in the `CoreApp` that we're extending, because it uses the same `path` but mounts `TweakedWikiApp` instead.

Framework apps

A `morepath.App` subclass does not need to be a full working web application. Instead it can be a framework with only those paths and views that we intend to be reusable.

We could for instance have a base class `Metadata` and define some views for it in the framework app. If we then have an application that inherits from the framework app, any `Metadata` model we expose to the web using the `path` directive automatically gets its views supplied by the framework.

For instance:

```
class Framework(morepath.App):
    pass

class Metadata(object):
    def __init__(self, d):
        self.d = d # metadata dictionary

    def get_metadata(self):
        return self.d

@Framework.json(model=Metadata, name='metadata')
def metadata_view(self, request):
    return self.get_metadata()
```

We want to use this framework in our own application:

```
class App(Framework):
    pass
```

Let's have a model that subclasses from `Metadata`:

```
class Document(Metadata):
    ...
```

Let's put the model on a path:

```
@App.path(path='documents/{id}', model=Document)
def get_document(id):
    ...
```

Since `App` extends `Framework`, all documents published this way have a `metadata` view automatically. Apps that don't extend `Framework` won't have this behavior, of course.

As we mentioned before, there is a gray area between application and framework; applications tend to gain attributes of a framework, and larger frameworks start to look more like applications. Don't worry too much about which is which, but enjoy the creative possibilities!

Note that Morepath itself is designed as an application (*morepath.App*) that your apps extend. This means you can override parts of it just like you would override a framework app! We did our best to make Morepath do the right thing already, but if not, you *can* customize it.

CHAPTER 22

REST

Introduction

How to think RESTful thoughts

So what does it mean for a web service to be RESTful? It might help to remember this when thinking about REST:

client :: RESTful web service

is like:

human with browser :: well-designed multi-page web application

So if you have experience with developing good multi-page web applications, then you can apply this experience to REST web service design and you're off to a good start.

In this section we'll look at how you could go about implementing a RESTful web service with Morepath.

REST stands for Representational State Transfer, and is a particular way to design web services. We won't try to explain here *why* this can be a good thing for you to do, just explain what is involved.

REST is not only useful for pure web services, but is also highly relevant for web application development, especially when you are building a single-page rich client application in JavaScript in the web browser. It can be beneficial to organize the server-side application as a RESTful web service.

Elements of REST

That's all rather abstract. Let's get more concrete. It's useful to refer to the Richardson Maturity Model for REST in this context. In REST we do the following:

- We uses HTTP as a transport system. What you use to communicate is typically JSON or XML, but it could be anything.

- We don't just use HTTP to tunnel method calls to a single URL. Instead, we model our web service as resources, each with their own URL, that we can interact with.
- We use HTTP methods meaningfully. Most importantly we use `GET` to retrieve information, and `POST` when we want to change information. Along with this we also use HTTP response status codes meaningfully.
- We have links between the resources. So, one resource points to another. A container resource could point to a link that you can `POST` to create a new sub resource in it, for instance, and may have a list of links to the resources in the container. See also HATEOAS.

Morepath has features that help you create RESTful applications.

HTTP as a transport system

We don't really need to say much here, as Morepath is of course all about HTTP in the end. Morepath lets you write a bare-bones view using *morepath.App.view()*. This also lets you pass in a `render` function that lets you specify how to render the return value of the view function as a *morepath.Response*. If you use JSON, for convenience you can use *morepath.App.json()* has a JSON render function baked in.

We could for instance have a `Document` model in our application:

```
class Document(object):
    def __init__(self, title, author, content):
        self.title = title
        self.author = author
        self.content = content
```

We can expose it on a URL:

```
@App.path(model=Document, path='documents/{id}')
def get_document(id=0):
   return document_by_id(id)
```

We assume here that a `document_by_id()` function exists that returns a `Document` instance by integer id from some database, or `None` if the document cannot be found. Any way to get your model instance is fine. We use `id=0` to tell Morepath that ids should be converted to integers, and to with a `BadRequest` if that is not possible.

Now we need a view that exposes the resource to JSON:

```
@App.json(model=Document)
def document_default(self, request):
    return {
      'type': 'document',
      'id': self.id,
      'title': self.title,
      'author': self.author,
      'content': self.content
    }
```

Modeling as resources

Modeling a web service as multiple resources comes pretty naturally to Morepath. You think carefully about how to place models in the URL space and then expose them using *morepath.App.path()*. Each model class can only be exposed on a single URL (per app), which gives them a canonical URL automatically.

A collection resource could be modelled like this:

```
class DocumentCollection(object):
    def __init__(self):
        self.documents = []
        self.id_counter = 0

    def add(self, doc):
        doc.id = self.id_counter
        self.id_counter += 1
        self.documents.append(doc)
        return doc
```

We now want to expose this collection to a URL path `/documents`. We want:

- when you `GET /documents` we want to get the ids documents in the collection.
- when you `POST` to `/documents` with a JSON body we want to add it to the collection.

Here is how we can make `documents` available on a URL:

```
documents = DocumentCollection()

@App.path(model=DocumentCollection, path='documents')
def get_document_collection():
    return documents
```

When someone accesses `/documents` they should get a JSON structure which includes ids of all documents in the collection. Here's how to do that (for `GET`, the default):

```
@App.json(model=DocumentCollection)
def document_collection_default(self, request):
    return {
       'type': 'document_collection',
       'ids': [doc.id for doc in self.documents]
    }
```

We also want to allow people to `POST` new documents (as a JSON POST body):

```
@App.json(model=DocumentCollection, request_method='POST')
def document_collection_post(self, request):
    json = request.json
    result = self.add(Document(title=json['title'],
                               author=json['author'],
                               content=json['content']))
    return request.view(result)
```

We use `Request.view()` to return the JSON structure for the added document again. This is handy as it includes the `id` field.

HTTP response status codes

When a view function returns normally, Morepath automatically sets the response HTTP status code to `200 Ok`.

When you try to access a URL that cannot be routed to a model because no path exists, or because the function involved returns `None`, or because the view cannot be found, a `404 Not Found` error is raised.

If you access a URL that does exist but with a request method that is not supported, a `405 Method Not Allowed` error is raised.

What if the user sends the wrong information to a view? Let's consider the `POST` view again:

```
@App.json(model=DocumentCollection, request_method='POST')
def document_collection_post(self, request):
    json = request.json
    result = self.add(Document(title=json['title'],
                               author=json['author'],
                               content=json['content']))
    return request.view(result)
```

What if the structure of the JSON submitted is not a valid document but contains some other information, or misses essential information? We should reject it if so. We can do this by raising a HTTP error ourselves. WebOb, the request/response library upon which Morepath is built, defines a set of HTTP exception classes `webob.exc` that we can use:

```
@App.json(model=DocumentCollection, request_method='POST')
def document_collection_post(self, request):
    json = request.json
    if not is_valid_document_json(json):
        raise webob.exc.HTTPUnprocessableEntity()
    result = self.add(Document(title=json['title'],
                               author=json['author'],
                               content=json['content']))
    return request.view(result)
```

What status code is right?

There is some debate over what status code to pick for content that is submitted that can be parsed but is incorrect. Some REST implementations use `400 Bad Request`, others use `422 Unprocessable Entity`. Morepath uses the latter by default, as we'll see in a bit.

Now we raise `422 Unprocessable Entity` when the submitted JSON body is invalid, using a function `is_valid_document_json` that does the checking. `is_valid_document` could look this:

```
def is_valid_document_json(json):
   if json['type'] != 'document':
      return False
   for name in ['title', 'author', 'content']:
      if name not in json:
         return False
   return True
```

load

The code that checks the validity of the POST or PUT body in the view can be moved out into a `load` function that you can use in multiple views:

```
def load(request):
   if not is_valid_document_json(json):
       raise webob.exc.HTTPUnprocessableEntity()
   return request.json
```

```
@App.json(model=DocumentCollection, request_method='POST', load=load)
def document_collection_post(self, request, json):
    result = self.add(Document(title=json['title'],
                               author=json['author'],
                               content=json['content']))
    return request.view(result)
```

The return value of the `load` function is passed in as a third argument into the view function. This means that you can also do conversion of input in the `load` function and reuse it between views. And if the load fails to work you get a 422 status code.

Linking: HATEOAS

We've now reached the point where many would say that this is a RESTful web service. But in fact a vital ingredient is still missing: hyperlinks. That ugly acronym HATEOAS thing.

Hyperlinks!

Since hyperlinks are so commonly missing from web services that claim to be RESTful, we'll break our promise here not to motivate why REST is good, and have a brief discussion on why hyperlinking is a good idea.

Without hyperlinks, a client is coupled to the server in two ways:

- URLs: it needs to know what URLs the server exposes.
- Data: it needs to know how to interpret the data coming from the server, and what data to send to the server.

Now add HATEOAS and get true REST. Now the client is coupled to the server in only one way: data. It gets the URLs it needs from the data. We gain looser coupling between server and client: the server can change all its URLs and the client will continue to work.

You may quibble and say the client still needs to know the original URL of the server to get started, and dig up all the other URLs from the data afterward. That's true – but that's all that's needed. It's normal. Think again like how a human interacts with the web through the browser: you may use a search engine or bookmarks to get the initial URL of a site, and then you go to pages in that site by clicking links.

Morepath makes it easy to create hyperlinks, so we won't have to do much. Before we had this for the collection view:

```
@App.json(model=DocumentCollection)
def document_collection_default(self, request):
    return {
        'type': 'document_collection',
        'ids': [doc.id for doc in self.documents]
    }
```

We can change this so instead of ids, we return a list of document URLs instead:

```
@App.json(model=DocumentCollection)
def document_collection_default(self, request):
    return {
        'type': 'document_collection',
        'documents': [request.link(doc) for doc in self.documents],
    }
```

Now we've got HATEOAS: the collection links to the documents it contains. The developers looking at the responses your web service sends get a few clues about where to go next. Coupling is looser.

We have HATEOAS, so at last we got true REST. Why is hyperlinking so often ignored? Why don't more systems implement HATEOAS? Perhaps because they make linking to things too hard or too brittle. Morepath instead makes it easy. Link away!

Compose from reusable apps

If you're going to create a larger RESTful web service, you should start thinking about composing them from smaller applications. See *App Reuse* for more information.

CHAPTER 23

Writing automated tests

This an introductory guide to writing automated tests for your Morepath project. We assume you've already installed Morepath; if not, see the *Installation* section.

In order to carry out the test we'll use WebTest, which you'll need to have installed. You also need a test automation tool; we recommend pytest. The *cookiecutter template* installs both for you, alternatively you can install them with pip:

```
$ pip install webtest pytest
```

Testing "Hello world!"

Let's look at a minimal test of the "Hello world!" application from the *Quickstart*:

```
from hello import App
from webtest import TestApp as Client

def test_hello():
    c = Client(App())

    response = c.get('/')

    assert response.body == b'Hello world!'
```

You can save this function into a file, say `test_hello.py` and use a test automation tool like pytest to run it:

```
$ py.test -q test_hello.py
.
1 passed in 0.13 seconds
```

If you invoke it as a regular Python function, a silent completion signifies success:

```
>>> test_hello()
```

Let's now go through the test, line by line.

1. We import the application that we want to test. In this case we assume that you have saved the "Hello world!" application from the *Quickstart* in `hello.py`:

   ```
   >>> from hello import App
   ```

   You can additionally use *`morepath.scan()`* if you are not sure whether importing the app imports all the modules that are required. In this particular instance, we know that importing `hello` is sufficient and *`morepath.scan()`* is not needed.

2. WebTest provides a class called `webtest.app.TestApp` that emulates a client for WSGI apps. We don't want to confuse it with the app under test, so we as a convention we import it as `Client`. This also stops pytest from scanning it for tests as it has the `Test` prefix:

   ```
   >>> from webtest import TestApp as Client
   ```

3. We instantiate the app under test and the client:

   ```
   >>> c = Client(App())
   ```

4. At this point we can use the client to query the app:

   ```
   >>> response = c.get('/')
   ```

   The returned response is an instance of `webtest.response.TestResponse`:

   ```
   >>> response
   <200 OK text/plain body=b'Hello world!'>
   ```

5. We can now verify that the response satisfies our expectations. In this case we test the response body in its entirety:

   ```
   >>> assert response.body == b'The view for model: foo'
   ```

CHAPTER 24

Directive tricks

Why not inside the class?

This in fact works:

```
class A(object):
    @classmethod
    @App.json(model=Foo)
    def foo_default(self, request):
        ...
```

But it is equivalent to using `@staticmethod`, so there is no point to do this.

This is **broken code**:

```
class A(object):
    @classmethod
    @App.json(model=Foo)
    def foo_default(cls, self, request):
        ...
```

This is broken because at the point `foo_default` is registered with `App.json` it isn't a `classmethod` yet, but a plain function, and it has the wrong signature to work with Morepath.

This is also **broken code**:

```
class A(object):
    @App.json(model=Foo)
    @classmethod
    def foo_default(cls, self, request):
        ...
```

This is broken because what gets registered with Morepath is an unbound class method, which is not callable.

But if you do:

```
class A(object):
    @classmethod
    def foo_default(cls, self, request):
        ...

App.json(model=Foo)(A.foo_default)
```

it works as `A.foo_default` binds the `cls` argument first.

You usually use Morepath directives like decorators on functions:

```
@App.json(model=Foo)
def foo_default(self, request):
    ...
```

You can also use directives with `@staticmethod`:

```
class A(object):
    @staticmethod
    @App.json(model=Foo)
    def foo_default(self, request):
        ...
```

It is important to apply `@staticmethod` directive after the Morepath directive is applied; it won't work the other away around.

With `@classmethod` the situation is slightly more involved. This is the correct way to do it:

```
class A(object):
  @classmethod
  def foo_default(cls, self, request):
      ...

App.json(model=Foo)(A.foo_default)
```

So, you apply the directive as a function to `A.foo_default` outside of the class.

This points to a general principle: we can use any Morepath directive as a plain function, not just as a decorator. This means you can combine a directive with a lambda, which sometimes leads to shorter code:

```
App.template_directory()(lambda: 'templates')
```

This means you can also register functions programmatically:

```
for i, func in enumerate(functions):
   App.json(model=Foo, name='view_%s' % i)(func)
```

We recommend caution here though – stick with the normal decorator based approach as much as you can as it is more declarative. This tends to lead to more maintainable code.

CHAPTER 25

Querying configuration

Creating a tool

A Morepath-based application may over time grow big, have multiple authors and spread over many modules. In this case it is helpful to have a tool that helps you explore Morepath configuration and quickly find what directives are defined where. The Dectate library details how to create such a tool, but we repeat it here for Morepath:

```
import dectate
from mybigapp import App

def query_tool():
    dectate.query_tool(App.commit())
```

You save it in a module called `query.py` in the `mybigapp` package. Then you hook it up in `setup.py` so that a query script gets generated:

```
entry_points={
    'console_scripts': [
        'morepathq = mybigapp.query:query_tool',
    ]
},
```

Now when you re-install your project, you get a command-line query tool called `morepathq` that lets you issue queries.

What just happened?

- In order to be able to query an app's configuration you need to commit it first. `App.commit()` also commits any other application you may have mounted into it. You get an iterable of apps that got committed.
- You pass this iterable into the `query_tool` function. This lets the query tool search through the configuration of the apps you committed only.
- **You hook it up so that a command-line script gets generated using** setuptool's `console_scripts` mechanism.

Usage

So now that you have a `morepathq` query tool, let's use it:

```
$ morepathq view
App: <class 'mybigapp.App'>
  File ".../somemodule.py", line 4
  @App.html(model=Foo)

  File ".../anothermodule.py", line 8
  @App.json(model=Bar)
```

Here we query for the `view` directive; since the `view` directive is grouped with `json` and `html` we get those back too. We get the module and line number where the directive was used.

You can also filter:

```
$ morepathq view model=mybigapp.model.Foo
App: <class 'mybigapp.App'>
  File ".../somemodule.py", line 4
  @App.html(model=Foo)
```

Here we query all views that have the `model` value set to `Foo` or one of its subclasses. Note that in able to refer to `Foo` in the query we use the dotted name to that class in the module it was defined.

You can query any Morepath directive this way:

```
$ morepathq path model=mybigapp.model.Foo
App: <class 'mybigapp.App'>
  File ".../path.py", line 8
  @App.path(model=Foo, path="/foo")
```

Part IV

Reference

In this section you can look up a specific function, class, or method.

CHAPTER 26

API

morepath

This is the main public API of Morepath.

Additional public APIs can be imported from the *morepath.error* and *morepath.pdbsupport* modules. For custom directive implementations that interact with core directives for grouping or subclassing purposes, or that need to use one of the Morepath registries, you may need to import from *morepath.directive*.

The other submodules are considered private. If you find yourself needing to import from them in application or extension code, please report an issue about it on the Morepath issue tracker.

class morepath.**App**

A Morepath-based application object.

You subclass App to create a morepath application class. You can then configure this class using Morepath decorator directives.

An application can extend one or more other applications, if desired, by subclassing them. By subclassing App itself, you get the base configuration of the Morepath framework itself.

Conflicting configuration within an app is automatically rejected. An subclass app cannot conflict with the apps it is subclassing however; instead configuration is overridden.

You can turn your app class into a WSGI application by instantiating it. You can then call it with the `environ` and `start_response` arguments.

Subclasses from `dectate.App`, which provides the `dectate.App.directive()` decorator that lets you register new directives.

request_class

The class of the Request to create. Must be a subclass of *morepath.Request*.

By default the request class is *morepath.Request*

alias of *Request*

classmethod _path(*path*, *model=None*, *variables=None*, *converters=None*, *required=None*, *get_converters=None*, *absorb=False*)

classmethod converter(*type*)
Register custom converter for type.

Parameters **type** – the Python type for which to register the converter. Morepath uses converters when converting path variables and URL parameters when decoding or encoding URLs. Morepath looks up the converter using the type. The type is either given explicitly as the value in the `converters` dictionary in the *morepath.App.path()* directive, or is deduced from the value of the default argument of the decorated model function or class using `type()`.

classmethod defer_class_links(*model*, *variables*)
Defer class link generation for model class to mounted app.

With `defer_class_links` you can specify that link generation for model classes is to be handled by a returned mounted app if it cannot be handled by the given app itself. *Request.class_link()*, *Request.link()* and *Request.view()* are affected by this directive.

The decorated function gets an instance of the application, the model class and a variables dict. It should return another application that it knows can create links for this class. The function uses navigation methods on *App* to do so like *App.parent()* and *App.child()*.

You also have to supply a `variables` argument to describe how to get the variables from an instance – this should be return the same variables as needed by the `path` directive in the app you are deferring to. This allows `defer_class_links` to function as `defer_links` for model objects as well.

Parameters

- **model** – the class for which we want to defer linking.
- **variables** – a function that given a model object can construct the variables used in the path (including any URL parameters).

classmethod defer_links(*model*)
Defer link generation for model to mounted app.

With `defer_links` you can specify that link generation for instances of `model` is to be handled by a returned mounted app if it cannot be handled by the given app itself. *Request.link()* and *Request.view()* are affected by this directive. Note that *Request.class_link()* is **not** affected by this directive, but you can use *morepath.App.defer_class_links()* instead.

The decorated function gets an instance of the application and object to link to. It should return another application that it knows can create links for this object. The function uses navigation methods on *App* to do so like *App.parent()* and *App.child()*.

Parameters **model** – the class for which we want to defer linking.

classmethod dump_json(*model=<type 'object'>*)
Register a function that converts model to JSON.

The decorated function gets `app` (app instance), `obj` (model instance) and `request` (*morepath.Request*) arguments. The `app` argument is optional. The function should return an JSON object. That is, a Python object that can be dumped to a JSON string using `json.dump`.

Parameters **model** – the class of the model for which this function is registered. The `self` passed into the function is an instance of the model (or of a subclass). By default the model is `object`, meaning we register a function for all model classes.

classmethod html(*model*, *render=None*, *template=None*, *load=None*, *permission=None*, *internal=False*, ***predicates*)
Register HTML view.

This is like *morepath.App.view()*, but with *morepath.render_html()* as default for the *render* function.

Sets the content type to `text/html`.

Parameters

- **model** – the class of the model for which this view is registered.
- **name** – the name of the view as it appears in the URL. If omitted, it is the empty string, meaning the default view for the model.
- **render** – an optional function that can render the output of the view function to a response, and possibly set headers such as `Content-Type`, etc. Renders as HTML by default. This function takes `self` and `request` parameters as input.
- **template** – a path to a template file. The path is relative to the directory this module is in. The template is applied to the content returned from the decorated view function.

  Use the `morepath.App.template_engine()` directive to define support for new template engines.
- **load** – a load function that turns the request into an object. If load is in use, this object will be the third argument to the view function
- **permission** – a permission class. The model should have this permission, otherwise access to this view is forbidden. If omitted, the view function is public.
- **internal** – Whether this view is internal only. If `True`, the view is only useful programmatically using *morepath.Request.view()*, but will not be published on the web. It will be as if the view is not there. By default a view is `False`, so not internal.
- **name** – the name of the view as it appears in the URL. If omitted, it is the empty string, meaning the default view for the model. This is a predicate.
- **request_method** – the request method to which this view should answer, i.e. GET, POST, etc. If omitted, this view will respond to GET requests only. This is a predicate.
- **predicates** – predicates to match this view on. See the documentation of *App.view()* for more information.

classmethod identity_policy()
Register identity policy.

The decorated function should return an instance of *morepath.IdentityPolicy*. Either use an identity policy provided by a library or implement your own.

It gets one optional argument: the settings of the app for which this identity policy is in use. So you can pass some settings directly to the IdentityPolicy class.

classmethod json(*model*, *render=None*, *template=None*, *load=None*, *permission=None*, *internal=False*, ***predicates*)
Register JSON view.

This is like *morepath.App.view()*, but with *morepath.render_json()* as default for the *render* function.

Transforms the view output to JSON and sets the content type to `application/json`.

Parameters

- **model** – the class of the model for which this view is registered.
- **name** – the name of the view as it appears in the URL. If omitted, it is the empty string, meaning the default view for the model.

- **render** – an optional function that can render the output of the view function to a response, and possibly set headers such as `Content-Type`, etc. Renders as JSON by default. This function takes `self` and `request` parameters as input.
- **template** – a path to a template file. The path is relative to the directory this module is in. The template is applied to the content returned from the decorated view function.

  Use the `morepath.App.template_engine()` directive to define support for new template engines.
- **load** – a load function that turns the request into an object. If load is in use, this object will be the third argument to the view function.
- **permission** – a permission class. The model should have this permission, otherwise access to this view is forbidden. If omitted, the view function is public.
- **internal** – Whether this view is internal only. If `True`, the view is only useful programmatically using *morepath.Request.view()*, but will not be published on the web. It will be as if the view is not there. By default a view is `False`, so not internal.
- **name** – the name of the view as it appears in the URL. If omitted, it is the empty string, meaning the default view for the model. This is a predicate.
- **request_method** – the request method to which this view should answer, i.e. GET, POST, etc. If omitted, this view will respond to GET requests only. This is a predicate.
- **predicates** – predicates to match this view on. See the documentation of *App.view()* for more information.

classmethod link_prefix()

Register a function that returns the prefix added to every link generated by the request.

By default the link generated is based on `webob.Request.application_url()`.

The decorated function gets `app` and `request` (*morepath.Request*) arguments. The `app` argument is optional. The function should return a string.

classmethod method(*dispatch_method*, ***kw*)

Register function as implementation of dispatch method.

This way you can create new hookable functions of your own, or override parts of the Morepath framework itself.

The `dispatch_method` argument is a dispatch method, so a method on a *morepath.App* class marked with `reg.dispatch_method()`, so for instance `App.foo`. The registered function gets the instance of this app class as its first argument. The registered function must have the same arguments as the arguments of the dispatch function.

The reason to use this form of registration instead of `reg.Dispatch.register()` directly is so that they are overridable just like any other Morepath directive.

Parameters

- **dispatch_method** – the dispatch method to register an implementation for.
- **kw** – keyword parameters with the predicate keys to register for. Argument names are predicate names, values are the predicate values to match on. These are like the predicate arguments for `reg.Dispatch.register()`.

classmethod mount(*path*, *app*, *variables=None*, *converters=None*, *required=None*, *get_converters=None*, *name=None*)

Mount sub application on path.

The decorated function gets the variables specified in path as parameters. It should return a new instance of an application class.

Parameters

- **path** – the path to mount the application on.
- **app** – the *morepath.App* subclass to mount.
- **variables** – a function that given an app instance can construct the variables used in the path (including any URL parameters). If omitted, variables are retrieved from the app by using the arguments of the decorated function.
- **converters** – converters as for the *morepath.App.path()* directive.
- **required** – list or set of names of those URL parameters which should be required, i.e. if missing a 400 Bad Request response is given. Any default value is ignored. Has no effect on path variables. Optional.
- **get_converters** – a function that returns a converter dictionary. This function is called once during configuration time. It can be used to programmatically supply converters. It is merged with the `converters` dictionary, if supplied. Optional.
- **name** – name of the mount. This name can be used with `Request.child()` to allow loose coupling between mounting application and mounted application. Optional, and if not supplied the `path` argument is taken as the name.

classmethod path(*path*, *model=None*, *variables=None*, *converters=None*, *required=None*, *get_converters=None*, *absorb=False*)

Register a model for a path.

Decorate a function or a class (constructor). The function should return an instance of the model class, for instance by querying it from the database, or `None` if the model does not exist.

The decorated function gets as arguments any variables specified in the path as well as URL parameters.

If you declare a `request` parameter the function is able to use that information too.

Parameters

- **path** – the route for which the model is registered.
- **model** – the class of the model that the decorated function should return. If the directive is used on a class instead of a function, the model should not be provided.
- **variables** – a function takes `app` and `model object` arguments. The `app` argument is optional. It can construct the variables used in the path (including any URL parameters). If `variables` is omitted, variables are retrieved from the model by using the arguments of the decorated function.
- **converters** – a dictionary containing converters for variables. The key is the variable name, the value is a *morepath.Converter* instance.
- **required** – list or set of names of those URL parameters which should be required, i.e. if missing a 400 Bad Request response is given. Any default value is ignored. Has no effect on path variables. Optional.
- **get_converters** – a function that returns a converter dictionary. This function is called once during configuration time. It can be used to programmatically supply converters. It is merged with the `converters` dictionary, if supplied. Optional.
- **absorb** – If set to `True`, matches any subpath that matches this path as well. This is passed into the decorated function as the `absorb` argument.

classmethod permission_rule(*model*, *permission*, *identity=<class 'morepath.authentication.Identity'>*)

Declare whether a model has a permission.

The decorated function receives `app`, `model`, `permission` (instance of any permission object) and `identity` (*morepath.Identity*) parameters. The `app` argument is optional. The decorated function should return `True` only if the given identity exists and has that permission on the model.

Parameters

- **model** – the model class
- **permission** – permission class
- **identity** – identity class to check permission for. If `None`, the identity to check for is the special *morepath.NO_IDENTITY*.

classmethod predicate(*dispatch*, *name*, *default*, *index*, *before=None*, *after=None*)

Register a custom predicate for a dispatch method.

The function to be registered should have the same arguments as the dispatch method and return a value that is used when registering an implementation for the dispatch method.

The predicates are ordered by their before and after arguments.

Parameters

- **dispatch** – the dispatch method this predicate is for. You can use the *App.method()* directive to add a dispatch method to an app.
- **name** – the name used to identify the predicate when registering the implementation of the dispatch method.
- **default** – the expected value of the predicate, to be used when registering an implementation if the expected value for the predicate is not given explicitly.
- **index** – the index to use. Typically `reg.KeyIndex` or `reg.ClassIndex`.
- **before** – predicate function this function wants to have priority over.
- **after** – predicate function we want to have priority over this one.

classmethod predicate_fallback(*dispatch*, *func*)

For a given dispatch and function dispatched to, register fallback.

The fallback is called with the same arguments as the dispatch function. It should return a response (or raise an exception that can be turned into a response).

Parameters

- **dispatch** – the dispatch function
- **func** – the registered function we are the fallback for

classmethod setting(*section*, *name*)

Register application setting.

An application setting is registered under the `.config.settings_registry` class attribute of *morepath.App* subclasses. It will be executed early in configuration so other configuration directives can depend on the settings being there.

The decorated function returns the setting value when executed.

Parameters

- **section** – the name of the section the setting should go under.

- **name** – the name of the setting in its section.

classmethod setting_section(*section*)
Register application setting in a section.

An application settings are registered under the `settings` attribute of `morepath.app.Registry`. It will be executed early in configuration so other configuration directives can depend on the settings being there.

The decorated function returns a dictionary with as keys the setting names and as values the settings.

Parameters **section** – the name of the section the setting should go under.

classmethod template_directory(*after=None*, *before=None*, *name=None*)
Register template directory.

The decorated function gets no argument and should return a relative or absolute path to a directory containing templates that can be loaded by this app. If a relative path, it is made absolute from the directory this module is in.

Template directories can be ordered: templates in a directory `before` another one are found before templates in a directory `after` it. But you can leave both `before` and `after` out: template directories defined in sub-applications automatically have a higher priority than those defined in base applications.

Parameters

- **after** – Template directory function this template directory function to be under. The other template directory has a higher priority. You usually want to use `over`. Optional.
- **before** – Template directory function function this function should have priority over. Optional.
- **name** – The name under which to register this template directory, so that it can be overridden by applications that extend this one. If no name is supplied a default name is generated.

classmethod template_loader(*extension*)
Create a template loader.

The decorated function gets a `template_directories` argument, which is a list of absolute paths to directories that contain templates. It also gets a `settings` argument, which is application settings that can be used to configure the loader.

It should return an object that can load the template given the list of template directories.

classmethod template_render(*extension*)
Register a template engine.

Parameters **extension** – the template file extension (`.pt`, etc) we want this template engine to handle.

The decorated function gets `loader`, `name` and `original_render` arguments. It should return a `callable` that is a view `render` function: take a `content` and `request` object and return a *morepath.Response* instance. This render callable should render the return value of the view with the template supplied through its `template` argument.

classmethod tween_factory(*under=None*, *over=None*, *name=None*)
Register tween factory.

The tween system allows the creation of lightweight middleware for Morepath that is aware of the request and the application.

The decorated function is a tween factory. It should return a tween. It gets two arguments: the app for which this tween is in use, and another tween that this tween can wrap.

A tween is a function that takes a request and a mounted application as arguments.

Tween factories can be set to be over or under each other to control the order in which the produced tweens are wrapped.

Parameters

- **under** – This tween factory produces a tween that wants to be wrapped by the tween produced by the `under` tween factory. Optional.
- **over** – This tween factory produces a tween that wants to wrap the tween produced by the over `tween` factory. Optional.
- **name** – The name under which to register this tween factory, so that it can be overridden by applications that extend this one. If no name is supplied a default name is generated.

classmethod verify_identity(*identity=<type 'object'>*)

Verify claimed identity.

The decorated function takes an `app` argument and an `identity` argument which contains the claimed identity. The `app` argument is optional. It should return `True` only if the identity can be verified with the system.

This is particularly useful with identity policies such as basic authentication and cookie-based authentication where the identity information (username/password) is repeatedly sent to the the server and needs to be verified.

For some identity policies (auth tkt, session) this can always return `True` as the act of establishing the identity means the identity is verified.

The default behavior is to always return `False`.

Parameters **identity** – identity class to verify. Optional.

classmethod view(*model*, *render=None*, *template=None*, *load=None*, *permission=None*, *internal=False*, ***predicates*)

Register a view for a model.

The decorated function gets `self` (model instance) and `request` (*morepath.Request*) parameters. The function should return either a (unicode) string that is the response body, or a *morepath.Response* object.

If a specific `render` function is given the output of the function is passed to this first, and the function could return whatever the `render` parameter expects as input. This function should take the object to render and the request. func:*morepath.render_json* for instance expects as its first argument a Python object such as a dict that can be serialized to JSON.

See also *morepath.App.json()* and *morepath.App.html()*.

Parameters

- **model** – the class of the model for which this view is registered. The `self` passed into the view function is an instance of the model (or of a subclass).
- **render** – an optional function that can render the output of the view function to a response, and possibly set headers such as `Content-Type`, etc. This function takes `self` and `request` parameters as input.
- **template** – a path to a template file. The path is relative to the directory this module is in. The template is applied to the content returned from the decorated view function.

  Use the *morepath.App.template_loader()* and *morepath.App.template_render()* directives to define support for new template engines.

- **load** – a load function that turns the request into an object. If load is in use, this object will be the third argument to the view function.
- **permission** – a permission class. The model should have this permission, otherwise access to this view is forbidden. If omitted, the view function is public.
- **internal** – Whether this view is internal only. If `True`, the view is only useful programmatically using *morepath.Request.view()*, but will not be published on the web. It will be as if the view is not there. By default a view is `False`, so not internal.
- **name** – the name of the view as it appears in the URL. If omitted, it is the empty string, meaning the default view for the model. This is a predicate.
- **request_method** – the request method to which this view should answer, i.e. GET, POST, etc. If omitted, this view responds to GET requests only. This is a predicate.
- **predicates** – additional predicates to match this view on. You can install your own using the *morepath.App.predicate()* directive.

__call__(*environ*, *start_response*)
This app as a WSGI application.

See the WSGI spec for more information.

Uses *App.request()* to generate a *morepath.Request* instance, then uses meth:*App.publish* get the *morepath.Response* instance.

Parameters

- **environ** – WSGI environment
- **start_response** – WSGI start_response

Returns WSGI iterable.

ancestors()
Return iterable of all ancestors of this app.

Includes this app itself as the first ancestor, all the way up to the root app in the mount chain.

child(*app*, ***variables*)
Get app mounted in this app.

Either give it an instance of the app class as the first parameter, or the app class itself (or name under which it was mounted) as the first parameter and as `variables` the parameters that go to its `mount` function.

Returns the mounted application object, with its `parent` attribute set to this app object, or `None` if this application cannot be mounted in this one.

classmethod commit()
Commit the app, and recursively, the apps mounted under it.

Mounted apps are discovered in breadth-first order.

Returns the set of discovered app clasess.

forget_identity(*response*, *request*)
Modify response so that identity is forgotten by client.

Parameters

- **response** – *morepath.Response* to forget identity on.
- **request** – *morepath.Request*

get_view(*obj*, *request*)
Get the view that represents the obj in the context of a request.

This view is a representation of the obj that can be rendered to a response. It may also return a *morepath.Response* directly.

Predicates are installed in *morepath.core* that inspect both `obj` and `request` to see whether a matching view can be found.

You can also install additional predicates using the *morepath.App.predicate()* and `morepath.App.precicate_fallback()` directives.

Parameters

- **obj** – model object to represent with view.
- **request** – *morepath.Request* instance.

Returns *morepath.Response* object, or `webob.exc.HTTPNotFound` if view cannot be found.

classmethod init_settings(*settings*)
Pre-fill the settings before the app is started.

Add settings to App, which can act as normal, can be overridden, etc.

Parameters **settings** – a dictionary of setting sections which contain dictionaries of settings.

classmethod mounted_app_classes(*callback=None*)
Returns a set of this app class and any mounted under it.

This assumes all app classes involved have already been committed previously, for instance by *morepath.App.commit()*.

Mounted apps are discovered in breadth-first order.

The optional `callback` argument is used to implement *morepath.App.commit()*.

Parameters **callback** – a function that is called with app classes as its arguments. This can be used to do something with the app classes when they are first discovered, like commit them. Optional.

Returns the set of app classes.

remember_identity(*response*, *request*, *identity*)
Modify response so that identity is remembered by client.

Parameters

- **response** – *morepath.Response* to remember identity on.
- **request** – *morepath.Request*
- **identity** – *morepath.Identity*

request(*environ*)
Create a *Request* given WSGI environment for this app.

Parameters **environ** – WSGI environment

Returns *morepath.Request* instance

sibling(*app*, ***variables*)
Get app mounted next to this app.

Either give it an instance of the app class as the first parameter, or the app class itself (or name under which it was mounted) as the first parameter and as `variables` the parameters that go to its `mount` function.

Returns the mounted application object, with its `parent` attribute set to the same parent as this one, or `None` if such a sibling application does not exist.

logger_name = 'morepath.directive'
Prefix used by dectate to log configuration actions.

parent = None
The parent in which this app was mounted.

publish
Publish functionality wrapped in tweens.

You can use middleware (*Tweens*) that can hooks in before a request is passed into the application and just after the response comes out of the application. Here we use `morepath.tween.TweenRegistry.wrap()` to wrap the *`morepath.publish.publish()`* function into the configured tweens.

This property uses *`morepath.reify.reify()`* so that the tween wrapping only happens once when the first request is handled and is cached afterwards.

Returns a function that a *`morepath.Request`* instance and returns a *`morepath.Response`* instance.

root
The root application.

settings
Returns the settings bound to this app.

`morepath.`**`scan`**(*package=None*, *ignore=None*, *handle_error=None*)
Scan package for configuration actions (decorators).

It scans by recursively importing the package and any modules in it, including any sub-packages.

Register any found directives with their app classes.

Parameters

- **package** – The Python module or package to scan. Optional; if left empty case the calling package is scanned.
- **ignore** – A list of packages to ignore. Optional. Defaults to `['.test', '.tests']`. See `importscan.scan()` for details.
- **handle_error** – Optional error handling function. See `importscan.scan()` for details.

`morepath.`**`autoscan`**(*ignore=None*)
Automatically load Morepath configuration from packages.

Morepath configuration consists of decorator calls on *`App`* instances, i.e. `@App.view()` and `@App.path()`.

This function tries to load needed Morepath configuration from all packages automatically. This only works if:

- The package is made available using a `setup.py` file.
- The package or a dependency of the package includes `morepath` in the `install_requires` list of the `setup.py` file.
- The `setup.py` name is the same as the name of the distributed package or module. For example: if the module inside the package is named `myapp` the package must be named `myapp` as well (not `my-app` or `MyApp`).

If the `setup.py` name differs from the package name, it's possible to specify the module morepath should scan using entry points:

```
setup(name='some-package',
  ...
  install_requires=[
      'setuptools',
      'morepath'
  ],
  entry_points={
      'morepath': [
          'scan = somepackage',
      ]
  })
```

This function simply recursively imports everything in those packages, except for test directories.

In addition to calling this function you can also import modules that use Morepath directives manually, and you can use *scan()* to automatically import everything in a single package.

Typically called immediately after startup just before the application starts serving using WSGI.

`autoscan` always ignores `.test` and `.tests` sub-packages – these are assumed never to contain useful Morepath configuration and are not scanned.

`autoscan` can fail with an `ImportError` when it tries to scan code that imports an optional dependency that is not installed. This happens most commonly in test code, which often rely on test-only dependencies such as `pytest` or `nose`. If those tests are in a `.test` or `.tests` sub-package they are automatically ignored, however.

If you have a special package with such expected import errors, you can exclude them from `autoscan` using the `ignore` argument, for instance using `['special_package']`. You then can use *scan()* for that package, with a custom `ignore` argument that excludes the modules that generate import errors.

See also *scan()*.

Parameters **ignore** – ignore to ignore some modules during scanning. Optional. If ommitted, ignore `.test` and `.tests` packages by default. See `importscan.scan()` for more details.

morepath.**commit**(**apps*)
Commit one or more app classes

A commit causes the configuration actions to be performed. The resulting configuration information is stored under the `.config` class attribute of each *App* subclass supplied.

This function may safely be invoked multiple times – each time the known configuration is recommitted.

Parameters ***apps** – one or more *App* subclasses to perform configuration actions on.

morepath.**run**(*wsgi*, *host='127.0.0.1'*, *port=5000*, *prog=None*, *ignore_cli=False*, *callback=None*)
Uses wsgiref.simple_server to run an application for debugging purposes.

By default, this function looks at the command line for arguments specified with the `--host` or `--port` options. These override the actual arguments passed to this function. Use `ignore_cli=True` to disable this behavior.

Under non-exceptional circumstances this function never returns.

Don't use this in production; use an external WSGI server instead, for instance Apache mod_wsgi, Nginx wsgi, Waitress, Gunicorn.

Parameters

- **wsgi** (*callable*) – WSGI app.
- **host** (*str*) – hostname or IP address on which to listen.

- **port** (*int*) – TCP port on which to listen.
- **prog** (*str or None*) – the name of the program displayed by diagnostics and help.
- **ignore_cli** (*bool*) – whether to ignore `sys.argv`.
- **callback** (*function(server) or None*) – function invoked after the creation of the server.

Returns never.

Note: Unless `ignore_cli` is true, this function provides a full-featured command-line parser. Its help message describes how to use it:

```
usage: <script name> [-h] [-p PORT] [-H HOST]

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  TCP port on which to listen (default: 5000)
  -H HOST, --host HOST  hostname or IP address on which to listen (default:
                        127.0.0.1)
```

The default values for the `--port` and `--host` options are takend from the value of the arguments passed to *morepath.run()*.

---

class `morepath.`**Request**(*environ*, *app*, ***kw*)

Request.

Extends `webob.request.BaseRequest`

after(*func*)

Call a function with the response after a successful request.

A request is considered *successful* if the HTTP status is a 2XX or a 3XX code (e.g. 200 OK, 204 No Content, 302 Found). In this case `after` *is* called.

A request is considered *unsuccessful* if the HTTP status lies outside the 2XX-3XX range (e.g. 403 Forbidden, 404 Not Found, 500 Internal Server Error). Usually this happens if an exception occurs. In this case `after` is *not* called.

Some exceptions indicate a successful request however and their occurrence still leads to a call to `after`. These exceptions inherit from either `webob.exc.HTTPOk` or `webob.exc.HTTPRedirection`.

You use *request.after* inside a view function definition.

It can be used explicitly:

```
@App.view(model=SomeModel)
def some_model_default(self, request):
    def myfunc(response):
        response.headers.add('blah', 'something')
    request.after(my_func)
```

or as a decorator:

```
@App.view(model=SomeModel)
def some_model_default(self, request):
    @request.after
    def myfunc(response):
        response.headers.add('blah', 'something')
```

Parameters **func** – callable that is called with response

Returns func argument, not wrapped

class_link(*model*, *variables=None*, *name=''*, *app=<SAME_APP>*)
Create a link (URL) to a view on a class.

Given a model class and a variables dictionary, create a link based on the path registered for the class and interpolate the variables.

If you have an instance of the model available you'd link to the model instead, but in some cases it is expensive to instantiate the model just to create a link. In this case *class_link* can be used as an optimization.

The *morepath.App.defer_class_links()* directive can be used to defer link generation for a particular class (if this app doesn't handle them) to another app.

Note that the *morepath.App.defer_links()* directive has **no** effect on `class_link`, as it needs an instance of the model to work, which is not available.

If no link can be constructed for the model class, a *morepath.error.LinkError* is raised. This error is also raised if you don't supply enough variables. Additional variables not used in the path are interpreted as URL parameters.

Parameters

- **model** – the model class to link to.
- **variables** – a dictionary with as keys the variable names, and as values the variable values. These are used to construct the link URL. If omitted, the dictionary is treated as containing no variables.
- **name** – the name of the view to link to. If omitted, the the default view is looked up.
- **app** – If set, change the application to which the link is made. By default the link is made to an object in the current application.

link(*obj*, *name=''*, *default=None*, *app=<SAME_APP>*)
Create a link (URL) to a view on a model instance.

The resulting link is prefixed by the link prefix. By default this is the full URL based on the Host header.

You can configure the link prefix for an application using the *morepath.App.link_prefix()* directive.

If no link can be constructed for the model instance, a *morepath.error.LinkError* is raised. `None` is treated specially: if `None` is passed in the default value is returned.

The *morepath.App.defer_links()* or *morepath.App.defer_class_links()* directives can be used to defer link generation for all instances of a particular class (if this app doesn't handle them) to another app.

Parameters

- **obj** – the model instance to link to, or `None`.
- **name** – the name of the view to link to. If omitted, the the default view is looked up.
- **default** – if `None` is passed in, the default value is returned. By default this is `None`.
- **app** – If set, change the application to which the link is made. By default the link is made to an object in the current application.

link_prefix(*app=None*)
Prefix to all links created by this request.

Parameters **app** – Optionally use the given app to create the link. This leads to use of the link prefix configured for the given app. This parameter is mainly used internally for link creation.

reset()
Reset request.

This resets the request back to the state it had when request processing started. This is used by `more.transaction` when it retries a transaction.

resolve_path(*path*, *app=<SAME_APP>*)
Resolve a path to a model instance.

The resulting object is a model instance, or `None` if the path could not be resolved.

Parameters

- **path** – URL path to resolve.
- **app** – If set, change the application in which the path is resolved. By default the path is resolved in the current application.

Returns instance or `None` if no path could be resolved.

view(*obj*, *default=None*, *app=<SAME_APP>*, ***predicates*)
Call view for model instance.

This does not render the view, but calls the appropriate view function and returns its result.

Parameters

- **obj** – the model instance to call the view on.
- **default** – default value if view is not found.
- **app** – If set, change the application in which to look up the view. By default the view is looked up for the current application. The `defer_links` directive can be used to change the default app for all instances of a particular class.
- **predicates** – extra predicates to modify view lookup, such as `name` and `request_method`. The default `name` is empty, so the default view is looked up, and the default `request_method` is `GET`. If you introduce your own predicates you can specify your own default.

app = None
morepath.App instance currently handling request.

identity
Self-proclaimed identity of the user.

The identity is established using the identity policy. Normally this would be an instance of *morepath.Identity*.

If no identity is claimed or established, or if the identity is not verified by the application, the identity is the the special value *morepath.NO_IDENTITY*.

The identity can be used for authentication/authorization of the user, using Morepath permission directives.

unconsumed = None
Stack of path segments that have not yet been consumed.

See *morepath.publish*.

class morepath.**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, *charset=<object object>*, ***kw*)

Response.

Extends webob.response.Response.

morepath.**render_html**(*content*, *request*)

Take string and return text/html response.

Parameters

- **content** – contnet as returned from view function.
- **request** – a *morepath.Request* instance.

Returns a *morepath.Response* instance with content as the body.

morepath.**render_json**(*content*, *request*)

Take dict/list/string/number content and return json response.

This respects the *morepath.App.dump_json()* directive that can be used to serialize any object to JSON. By default this serializes Python objects like dicts, strings to JSON.

Parameters

- **content** – content as returned from view function.
- **request** – a *morepath.Request* instance.

Returns a *morepath.Response* instance with a serialized JSON body.

morepath.**redirect**(*location*)

Return a response object that redirects to location.

Parameters **location** – a URL to redirect to.

Returns a webob.exc.HTTPFound response object. You can return this from a view to redirect.

class morepath.**Identity**(*userid*, ***kw*)

Claimed identity of a user.

Note that this identity is just a claim; to authenticate the user and authorize them you need to implement Morepath permission directives.

Parameters

- **userid** – The userid of this identity
- **kw** – Extra information to store in identity.

as_dict()

Export identity as dictionary.

This includes the userid and the extra keyword parameters used when the identity was created.

Returns dict with identity info.

userid = None

The user ID of the identity.

May be None if no particular identity was established.

class morepath.**IdentityPolicy**

Identity policy API.

Implement this API if you want to have a custom way to establish identities for users in your application.

forget(*response*, *request*)
Forget identity on response.

Implements `morepath.App.forget_identity`, which is called from user logout code.

Remove identifying information from the response. This could delete a cookie or issue a basic auth re-authentication.

Parameters

- **response** (*morepath.Response*) – response object on which to forget identity.
- **request** (*morepath.Request*) – request object.

identify(*request*)
Establish what identity this user claims to have from request.

Parameters **request** (*morepath.Request*.) – Request to extract identity information from.

Returns *morepath.Identity* instance or *morepath.NO_IDENTITY* if identity cannot be established.

remember(*response*, *request*, *identity*)
Remember identity on response.

Implements `morepath.App.remember_identity`, which is called from user login code.

Given an identity object, store it on the response, for instance as a cookie. Some policies may not do any storing but instead retransmit authentication information each time in the request.

Parameters

- **response** (*morepath.Response*) – response object on which to store identity.
- **request** (*morepath.Request*) – request object.
- **identity** (*morepath.Identity*) – identity to remember.

morepath.**NO_IDENTITY = <morepath.authentication.NoIdentity object>**
The identity if the request is anonymous.

The user has not yet logged in.

morepath.**EXCVIEW = <function excview_tween_factory>**
Exception views.

If an exception is raised by application code and a view is declared for that exception class, use it.

If no view can be found, raise it all the way up – this will be a 500 internal server error and an exception logged.

morepath.**HOST_HEADER_PROTECTION = <function poisoned_host_header_protection_tween_factory>**
Protect Morepath applications against the most basic host header poisoning attacts.

The regex approach has been copied from the Django project. To find more about this particular kind of attack have a look at the following references:

•http://skeletonscribe.net/2013/05/practical-http-host-header-attacks

•https://www.djangoproject.com/weblog/2012/dec/10/security/

•https://github.com/django/django/commit/77b06e41516d8136b56c040cba7e235b

class morepath.**Converter**(*decode*, *encode=None*)
Decode from strings to objects and back.

Used internally by the *morepath.App.converter()* directive.

Only used for decoding for a list with a single value, will error if more or less than one value is entered.

Used for decoding/encoding URL parameters and path parameters.

Create new converter.

Parameters

- **decode** – function that given string can decode them into objects.
- **encode** – function that given objects can encode them into strings.

decode(*strings*)
Decode list of strings into Python value.

String must have only a single entry.

Parameters **strings** – list of strings.

Returns Python value

encode(*value*)
Encode Python value into list of strings.

Parameters **value** – Python value

Returns List of strings with only a single entry

is_missing(*value*)
True is a given value is the missing value.

morepath.**dispatch_method**(**predicates*, ***kw*)
Decorator to make a method on a context class dispatch.

This takes the predicates to dispatch on as zero or more parameters.

Parameters

- **predicates** – sequence of `Predicate` instances to do the dispatch on. You create predicates using `reg.match_instance()`, `reg.match_key()`, `reg.match_class()`, or with a custom predicate class.

  You can also pass in plain string argument, which is turned into a `reg.match_instance()` predicate.
- **get_key_lookup** – a function that gets a `PredicateRegistry` instance and returns a key lookup. A `PredicateRegistry` instance is itself a key lookup, but you can return a caching key lookup (such as `reg.DictCachingKeyLookup` or `reg.LruCachingKeyLookup`) to make it more efficient.
- **first_invocation_hook** – a callable that accepts an instance of the class in which this decorator is used. It is invoked the first time the method is invoked.

morepath.error – exception classes

The exception classes used by Morepath.

Morepath republishes some configuration related errors from Dectate:

- `dectate.ConfigError`
- `dectate.ConflictError`
- `dectate.DirectiveReportError`

- dectate.DirectiveError
- dectate.TopologicalSortError

Morepath specific errors:

exception morepath.error.**AutoImportError**(*module_name*)
Raised when Morepath fails to import a module during autoscan.

exception morepath.error.**TrajectError**
Raised when path supplied to traject is not allowed.

exception morepath.error.**LinkError**
Raised when a link cannot be made.

exception morepath.error.**TopologicalSortError**
Raised if dependencies cannot be sorted topologically.

This is due to circular dependencies.

morepath.pdbsupport – debugging support

morepath.pdbsupport.**set_trace**(**args*, ***kw*)
Set pdb trace as in `import pdb; pdb.set_trace`, ignores `reg`.

Use `from morepath import pdbsupport; pdbsupport.set_trace()` to use.

The debugger won't step into `reg`, `inspect` or `repoze.lru`.

CHAPTER 27

morepath.directive – Extension API

This module contains the extension API for Morepath. It is useful when you want to define new directives in a Morepath extension. An example an extension that does this is more.static.

If you just use Morepath you should not have to import from *morepath.directive* in your code. Instead you use the directives defined in here through *morepath.App*.

Morepath uses the Dectate library to implement its directives. The directives are installed on *morepath.App* using the dectate.App.directive() decorator.

We won't repeat the directive documentation here. If you are interested in creating a custom directive in a Morepath extension it pays off to look at the source code of this module. If your custom directive needs to interact with a core directive you can inherit from them, and/or refer to them with group_class.

When configuration is committed it is written into various configuration registries which are attached to the dectate.App.config class attribute. If you implement your own directive dectate.Action that declares one of these registries in dectate.Action.config you can import their class from *morepath.directive*.

Registry classes

class morepath.directive.**ConverterRegistry**

A registry for converters.

Used to decode/encode URL parameters and path variables used by the *morepath.App.path()* directive.

Is aware of inheritance.

actual_converter(*spec*)

Return an actual converter for a given spec.

Parameters **spec** – if a type, return the registered converter for that; if a list use its first element as a spec for a converter; else, assume it is a converter and return it.

Returns a *morepath.Converter* instance.

argument_and_explicit_converters(*arguments*, *converters*)
Use explict converters unless none supplied, then use default args.

register_converter(*type*, *converter*)
Register a converter for type.

Parameters

- **type** – the Python type for which to register the converter.
- **converter** – a *morepath.Converter* instance.

class morepath.directive.**PathRegistry**(*app_class*, *converter_registry*)
A registry for routes.

Subclasses *morepath.traject.TrajectRegistry*.

Used by *morepath.App.path()* and *morepath.App.mount()* directives to register routes. Also used by the *morepath.App.defer_links()* and *morepath.App.defer_class_links()* directives.

Parameters **converter_registry** – a *morepath.directive.ConverterRegistry* instance

register_defer_class_links(*model*, *get_variables*, *app_factory*)
Register factory for app to defer class links to.

See *morepath.App.defer_class_links()* for more information.

Parameters

- **model** – model class to defer links for.
- **get_variables** – get variables dict for obj.
- **app_factory** – function that model class, app instance and variables dict as arguments and should return another app instance that does the link generation.

register_defer_links(*model*, *app_factory*)
Register factory for app to defer links to.

See *morepath.App.defer_links()* for more information.

Parameters

- **model** – model class to defer links for.
- **app_factory** – function that takes app instance and model object as arguments and should return another app instance that does the link generation.

register_inverse_path(*model*, *path*, *factory_args*, *converters=None*, *absorb=False*)
Register information for link generation.

Parameters

- **model** – model class
- **path** – the route
- **factory_args** – a list of the arguments of the factory function for this path.
- **converters** – a converters dict.
- **absorb** – bool, if true this is an absorbing path.

register_mount(*app*, *path*, *variables*, *converters*, *required*, *get_converters*, *mount_name*, *code_info*, *app_factory*)
Register a mounted app.

See *morepath.App.mount()* for more information.

Parameters

- **app** – *morepath.App* subclass.
- **path** – route
- **variables** – function that given model instance extracts dictionary with variables used in path and URL parameters.
- **converters** – converters structure
- **required** – required URL parameters
- **get_converters** – get a converter dynamically.
- **mount_name** – explicit name of this mount
- **code_info** – a dectate.CodeInfo instance used to register this directive.
- **app_factory** – function that constructs app instance given variables extracted from path and URL parameters.

register_path(*model*, *path*, *variables*, *converters*, *required*, *get_converters*, *absorb*, *code_info*, *model_factory*)

Register a route.

See *morepath.App.path()* for more information.

Parameters

- **model** – model class
- **path** – route
- **variables** – function that given model instance extracts dictionary with variables used in path and URL parameters.
- **converters** – converters structure
- **required** – required URL parameters
- **get_converters** – get a converter dynamically.
- **absorb** – absorb path
- **code_info** – the dectate.CodeInfo object describing the line of code used to register the path.
- **model_factory** – function that constructs model object given variables extracted from path and URL parameters.

register_path_variables(*model*, *func*)

Register variables function for a model class.

Parameters

- **model** – model class
- **func** – function that gets a model instance argument and returns a variables dict.

class morepath.directive.**PredicateRegistry**(*app_class*)

A registry of what predicates are registered for which functions.

It also keeps track of how predicates are to be ordered.

get_predicates(*dispatch*)
Create Reg predicates.

This creates `reg.Predicate` objects for a particular dispatch function.

Uses *`PredicateRegistry.sorted_predicate_infos()`* to sort the predicate infos.

Parameters **dispatch** – the dispatch function to create the predicates for.

Returns a list of `reg.Predicate` instances in the correct order.

install_predicates()
Install the predicates with reg.

This should be called during configuration once all predicates and fallbacks are known. Uses *`PredicateRegistry.get_predicates()`* to get out the predicates in the correct order.

register_predicate(*func*, *dispatch*, *name*, *default*, *index*, *before*, *after*)
Register a predicate for installation into the reg registry.

See *`morepath.App.predicate()`* for details.

Parameters

- **func** – the function that implements the predicate.
- **dispatch** – the dispatch function to register the predicate on.
- **name** – name of the predicate.
- **default** – default value.
- **index** – index to use.
- **before** – predicate function to have priority over.
- **after** – predicate function that has priority over this one.

register_predicate_fallback(*dispatch*, *func*, *fallback_func*)
Register a predicate fallback for installation into reg registry.

See *`morepath.App.predicate_fallback()`* for details.

Parameters

- **dispatch** – the dispatch function to register fallback on.
- **func** – the predicate function to register fallback for.
- **fallback_func** – the fallback function.

sorted_predicate_infos(*dispatch*)
Topologically sort predicate infos for a dispatch function.

Parameters **dispatch** – the dispatch function to sort for.

Returns a list of sorted `PredicateInfo` instances.

class `morepath.directive.`**SettingRegistry**
Registry of settings.

Used by the *`morepath.App.setting`* directive and *`morepath.App.setting_section`* directives.

Stores sections as attributes, which then have the settings as attributes.

This settings registry is exposed through *`morepath.App.settings`*.

register_setting(*section_name*, *setting_name*, *func*)
Register a setting.

Parameters

- **section_name** – name of section to register in
- **setting_name** – name of setting
- **func** – function that when called without arguments creates the setting value.

class morepath.directive.**TemplateEngineRegistry**(*setting_registry*)

A registry of template engines.

Is used by the *morepath.App.view()*, *morepath.App.json()* and *morepath.App.html()* directives for template-based rendering.

Parameters **setting_registry** – a *morepath.directive.SettingRegistry* instance.

get_template_render(*name*, *original_render*)

Get a template render function.

Parameters

- **name** – filename of the template (with extension, without path), such as `foo.pt`.
- **original_render** – `render` function supplied with the view directive.

Returns a `render` function that uses the template to render the result of a view function.

initialize_template_loader(*extension*, *func*)

Initialize a template loader for an extension.

Used by the *morepath.App.template_loader()* directive.

Parameters

- **extension** – template extension like `.p.t`
- **func** – function that given a list of template directories returns a load object that be used to load the template for use.

register_template_directory_info(*key*, *directory*, *before*, *after*, *configurable*)

Register a directory to look for templates.

Used by the *morepath.App.template_directory()* directive.

Parameters

- **key** – unique key identifying this directory
- **directory** – absolute path to template directory
- **before** – key to before in template lookup
- **after** – key to sort after in template lookup
- **configurable** – `dectate.Configurable` used that registered this template directory. Used for implicit sorting by app inheritance.

register_template_render(*extension*, *func*)

Register way to get a view render function for a file extension.

Used by the *morepath.App.template_render()* directive. See there for more information about parameters.

Parameters

- **extension** – template extension like `.pt`

- **func** – function that given loader, name and original_renderer constructs a view `render` function.

sorted_template_directories()
Get sorted template directories.

Use explicit `before` and `after` information but also App inheritance to sort template directories in order of template lookup.

Returns a list of template directory paths in the right order

class `morepath.directive.`**TweenRegistry**
Registry for tweens.

register_tween_factory(*tween_factory*, *over*, *under*)
Register a tween factory.

Parameters **tween_factory** – a function that constructs a tween given a *morepath.App* instance and a function that takes a *morepath.Request* argument and returns a *morepath.Response* (or a `webob.response.Response`).

Over this tween factory wraps the tween created by the `over` factory (possibly indirectly).

Under the `under` factory wraps the tween created by this one (possibly indirectly).

sorted_tween_factories()
Sort tween factories topologically by over and under.

Returns a sorted list of tween infos.

wrap(*app*)
Wrap app with tweens.

This wraps *morepath.publish.publish()* with tweens.

Parameters **app** – an instance of *morepath.App*.

Returns the application wrapped with tweens. This is a function that takes request and returns a a response.

Action classes

To instantiate an action you need to give it the same arguments as the directive it implements. Reading the source of existing actions is helpful when you want to implement your own actions. See `morepath/directive.py`.

class `morepath.directive.`**SettingAction**
morepath.App.setting()

class `morepath.directive.`**SettingSectionAction**
morepath.App.setting_section()

class `morepath.directive.`**PredicateFallbackAction**
morepath.App.predicate_fallback()

class `morepath.directive.`**PredicateAction**
morepath.App.predicate()

class `morepath.directive.`**FunctionAction**
`morepath.App.function()`

class `morepath.directive.`**ConverterAction**
morepath.App.converter()

class morepath.directive.**PathAction**
Helps to implement *morepath.App.path()*

class morepath.directive.**PathCompositeAction**
morepath.App.path()

class morepath.directive.**PermissionRuleAction**
morepath.App.permission_rule()

class morepath.directive.**TemplateDirectoryAction**
morepath.App.template_directory()

class morepath.directive.**TemplateLoaderAction**
morepath.App.template_loader()

class morepath.directive.**TemplateRenderAction**
morepath.App.template_render()

class morepath.directive.**ViewAction**
morepath.App.view()

class morepath.directive.**JsonAction**
morepath.App.json()

class morepath.directive.**HtmlAction**
morepath.App.html()

class morepath.directive.**MountAction**
morepath.App.mount()

class morepath.directive.**DeferLinksAction**
morepath.App.defer_links()

class morepath.directive.**DeferClassLinksAction**
morepath.App.defer_class_links()

class morepath.directive.**TweenFactoryAction**
morepath.App.tween_factory()

class morepath.directive.**IdentityPolicyFunctionAction**
Helps to implement *morepath.App.identity_policy()*

class morepath.directive.**IdentityPolicyAction**
morepath.App.identity_policy()

class morepath.directive.**VerifyIdentityAction**
morepath.App.verify_identity()

class morepath.directive.**DumpJsonAction**
morepath.App.dump_json()

class morepath.directive.**LinkPrefixAction**
morepath.App.link_prefix()

Part V

Contributor Guide

If you want to contribute to the project, this part of the documentation is for you.

CHAPTER 28

Developing Morepath

Community

Communication is important, so see *Community* for information on how to get in touch!

Install Morepath for development

Clone Morepath from github:

```
$ git clone git@github.com:morepath/morepath.git
```

If this doesn't work and you get an error 'Permission denied (publickey)', you need to upload your ssh public key to github.

Then go to the morepath directory:

```
$ cd morepath
```

Make sure you have virtualenv installed.

Create a new virtualenv for Python 3 inside the morepath directory:

```
$ virtualenv -p python3 env/py3
```

Activate the virtualenv:

```
$ source env/py3/bin/activate
```

Make sure you have recent setuptools and pip installed:

```
$ pip install -U setuptools pip
```

Install the various dependencies and development tools from requirements/develop.txt:

```
$ pip install -Ur requirements/develop.txt --src src
```

This needs your ssh key installed in github to work.

The `--src src` option makes sure that the dependent `reg`, `dectate` and `importscan` projects are checked out in the `src` directory. You can make changes to them during development too.

For upgrading the sources and requirements just run the command again.

If you want to test Morepath with Python 2.7 as well you can create a second virtualenv for it:

```
$ virtualenv -p python2.7 env/py27
```

You can then activate it:

```
$ source env/py27/bin/activate
```

Then uprade setuptools and pip and install the develop requirements as described above.

Note: The following commands work only if you have the virtualenv activated.

Running the tests

You can run the tests using py.test:

```
$ py.test
```

To generate test coverage information as HTML do:

```
$ py.test --cov --cov-report html
```

You can then point your web browser to the `htmlcov/index.html` file in the project directory and click on modules to see detailed coverage information.

flake8

flake8 is a tool that can do various checks for common Python mistakes using pyflakes and checks for PEP8 style compliance. We want a codebase where there are no flake8 messages.

To do pyflakes and pep8 checking do:

```
$ flake8 morepath
```

radon

radon is a tool that can check various measures of code complexity.

To check for cyclomatic complexity (excluding the tests):

```
$ radon cc morepath -e "morepath/tests*"
```

To filter for anything not ranked A:

```
$ radon cc morepath --min B -e "morepath/tests*"
```

And to see the maintainability index:

```
$ radon mi morepath -e "morepath/tests*"
```

Running the documentation tests

The documentation contains code. To check these code snippets, you can run this code using this command:

```
(py3) $ sphinx-build -b doctest doc doc/build/doctest
```

Or alternatively if you have Make installed:

```
(py3) $ cd doc
(py3) $ make doctest
```

Or from the Morepath project directory:

```
(py3) $ make -C doc doctest
```

Since the sample code in the documentation is maintained in Python 3 syntax, we do not support running the doctests with Python 2.7.

Building the HTML documentation

To build the HTML documentation (output in doc/build/html), run:

```
$ sphinx-build doc doc/build/html
```

Or alternatively if you have Make installed:

```
$ cd doc
$ make html
```

Or from the Morepath project directory:

```
$ make -C doc html
```

Developing Reg, Dectate or Importscan

If you need to adjust the sources of Reg, Dectate or Importscan and test them together with Morepath, they're available in the src directory. You can edit them and test changes in the Morepath project directly.

If you want to run the tests for one of them, let's say Reg, do:

```
$ cd src/reg
$ py.test
```

Tox

With tox you can test Morepath under different Python environments.

We have Travis continuous integration installed on Morepath's github repository and it runs the same tox tests after each checkin.

First you should install all Python versions which you want to test. The versions which are not installed will be skipped. You should at least install Python 3.5 which is required by flake8, coverage and doctests and Python 2.7 for testing Morepath with Python 2.

One tool you can use to install multiple versions of Python is pyenv.

To find out which test environments are defined for Morepath in tox.ini run:

```
$ tox -l
```

You can run all tox tests with:

```
$ tox
```

You can also specify a test environment to run e.g.:

```
$ tox -e py35
$ tox -e pep8
$ tox -e docs
```

To find out which dependencies and which versions tox installs in the testenv, you can use:

```
$ tox -e freeze
```

Deprecation

In some cases we have to make changes that break compatibility and break user code. We mark these in `CHANGES.txt` (*CHANGES*) using **breaking change**, **deprecated** or **removed**.

These entries should explain the change, and also tell the user what to do to upgrade their code. Do include an before/after code example as that makes it much easier, even if it's a simple import change.

We like to keep things moving and reserve the right to introduce breaking changes. When we do make a breaking change it should be marked clearly in `CHANGES.txt` (*CHANGES*) with a **Breaking change** marker.

If it is not a great burden we use deprecations. Morepath in this case retains the old APIs but issues a deprecation warning. See *Upgrading to a new Morepath version* for the notes for end-users concerning this. Here is the deprecation procedure for developers:

- Add a **Deprecated** entry in `CHANGES.txt` that describes what to do, as in a **breaking change**.
- Issue a deprecation warning in the code that is deprecated.
- Put a `**Deprecated**` entry in the docstring of whatever got deprecated with a brief comment on what to do.

- Put an issue labeled `remove deprecation` in the tracker for one release milestone after the upcoming release that states we should remove the deprecation. Create the milestone if needed.

  This way we don't maintain deprecated code and their warnings indefinitely – one release later we remove the backwards compatibility code and deprecation warnings.

- Once we go and remove code, we repeat the information on what to do in a new *Removed** entry in `CHANGES.txt`; treat it just like **Breaking change** and recycle the text written for the previous **Deprecated** entry for the stuff we're now removing.

CHAPTER 29

Design Notes

Some of the use cases that influenced Morepath's design are documented here.

Publish any model

It should be possible to publish any model object to the web on a readable URL. This includes model objects that are retrieved from a relational database and were created with a ORM.

Allowing individual models to be published on separate URLs avoids the god object antipattern where all web operations are routed through a single object. Instead each model, through view objects, can handle model-specific requests and operations. This encourages a more modular and reusable application design.

Routing

It should be easy to declare explicit routes to model. A route consists of a routing pattern with zero or more variables. The variables are used to identify the model, for instance using a relational database query.

Having routes makes it easier to reason about the URL structure of an application. Routes also make it easier to expose models that are retrieved using a query or are constructed on the fly, without imposing a specific structure on the models.

Traversal

It should be possible to associate routes with specific models in the application, not just to the root. This way models with sub-paths to sub-components can be made available as reusable components; an example of this could be a container. If the model is published, its sub-components are then exposed as well.

This allows for increased reuse of not just models but relationships between models, and lets the developer publish nested structures that cannot be specified using routing alone.

Linking

If a model is published, it should be possible to automatically generate a link to a model instance in the form of a URL.

This way there is no need to construct URLs manually, and there is no need to have to refer to routes explicitly in order to construct URLs. The system knows which route to use and how to construct the parameters that go into the route itself, given the model.

This is useful when creating RESTful web services (where hypermedia is the engine of application state), or to construct rich client-side applications that get all their URLs from the server from a REST-style web service.

Model is web-agnostic

Model classes should not have to have any web knowledge; no particular base classes are required, and no methods or attributes need to be implemented in order to publish instances of that model to the web. In case of an ORM, the ORM does not need to be reconfigured in order to publish ORM-mapped classes to the web. Models do not receive any request object and do not have to generate a response object.

Instead this knowledge is external to the models. Models should be optimized for programmatic use in general.

View/model separation

View objects are responsible for translating the model to the web and web operations to operations on the model. Views receive the request object and generate the response object. This is again to avoid giving the models knowledge about the web. This is a kind of model/view separation where the view is the intermediary between the model and the web.

Isolation between applications

The system allows multiple applications to be published at the same time. Applications work in isolation from each other by default. For instance, publishing a model on a URL does not affect another application, and publishing a view for a model does not make that view available in the other application.

Sharing between applications

In order to support reusable components, it should be possible to explicitly break application isolation and make routes to models and views globally available. Each application will share this information.

[Morepath in fact now allows more controlled sharing; only Morepath itself is globally shared]

Models can be published once per application

Per application a model can be exposed on a single URL pattern. So, the same instance could be published once per application, in a URL structure optimal for each application.

Again this supports applications working in isolation - they may treat database models differently than other applications do.

Linking to another application

It should be possible to construct URLs to models in the context of another application, if this application is given explicitly during link time.

Reusable components

It should be possible to define a base class (or interface) for a model that automatically pulls in (globally registered) views and sub-paths when you subclass from it. This lets a framework developer define APIs that an application developer can implement. By doing so, the application developer automatically gets a whole set of views for their models.

Declarative

It should be possible to register the components in a declarative way. This avoids spaghetti registration code, and also makes it possible to more easily reason about registrations (for instance to do overrides or detect conflicts).

Conflicts

If you try to do the same registration multiple times, the system should fail explicitly, as otherwise this would lead to subtle errors.

Overrides

It should be possible to override one registration with another one. This should either be an explicit operation, or the result of overriding in a different registry that has precedence over the defaults.

CHAPTER 30

Implementation Overview

Introduction

This documentation contains an overview of how the implementation of Morepath works. This includes a description of the different pieces of Morepath as well as an overview of the internal APIs.

How it all works

import-time

When a Morepath-based application starts, the first thing that happens is that all directives in modules that are imported are registered with Morepath's configuration engine. This configuration engine is implemented using the Dectate library. Configuration is associated with *morepath.App* subclasses.

Only the minimum code is executed to register the directives with their App classes; the directive actions are not performed yet.

Besides normal Python imports, *morepath.scan()* and *morepath.autoscan()* can be used to automatically import modules so that their directives are registered.

commit

Configuration is then committed using *morepath.App.commit()*, or the more low-level *morepath.commit()*.

This causes `dectate.commit()` to be called for a particular App class. This takes all the configuration as recorded during import-time and resolves it. This involves:

- detect any conflicts between documentation and reporting it.
- detect any `morepath.error.DirectiveError` errors raised by individual directive actions.

- resolve subclassing so that apps inherit from base apps and can override as well.
- performing the resulting configuration actions in the correct order.

All this is implemented by Dectate.

Morepath specific directives are defined in *morepath.directive*. Each directly or indirectly cause dectate.Action objects to be created. When the action is performed various configuration registry objects are affected. These registries are the end result of configuration.

morepath.directive.RegRegistry is the most advanced of registries used in Morepath and is based on reg.Registry. In this registry generic dispatch functions as defined in morepath.generic get individual implementations registered for them. Reg dispatches to specific implementations based on the function arguments used to call the generic function. Much of the functionality in Morepath ultimately causes a registration into the Reg registry and during runtime uses the API in morepath.generic.

A special registry contains the settings; setting functions as declared by *morepath.App.setting()* and *morepath.App.setting_section()* are executed and stored in a *morepath.directive.SettingRegistry* which is accessible through *morepath.App.settings*.

instantiate the app

Once configuration has successfully completed, the app can be instantiated. Apps are also instantiated during run-time when they are mounted and a path resolves into a mounted app.

The app is now also a WSGI function so can be run with any WSGI server.

run-time

When a request comes in through WSGI into *morepath.App.__call__()*, a *morepath.Request* object is created.

morepath.publish.publish() defines the core Morepath publication procedure, which turns requests into responses. This is done by first resolving the model and then rendering the resulting model instance as a response.

During the first request, tweens (as declared using *morepath.App.tween_factory()*) are created and wrapped around *morepath.publish.publish()*. Tweens are request/response based middleware functions. A standard Morepath tween implemented by *morepath.core.excview_tween_factory()*, renders exceptions raised by application code as views. The default Morepath tween factories are declared in *morepath.core* and tween factories get registered into *morepath.directive.TweenRegistry*.

resolve the model

morepath.publish.resolve_model() looks up a model object as created by the factory functions the user declared with the *morepath.App.path()* directive and the *morepath.App.mount()* directives.

morepath.path contains the *morepath.directive.PathRegistry* that has the API to register paths.

The route registration and resolution system is implemented by *morepath.traject*.

Default converters used during this step are declared in *morepath.core*. Converters are declared with the *morepath.App.converter()* directive and are registered in the *morepath.directive.ConverterRegistry*.

render the model object

morepath.publish.resolve_response() then renders the model object to a response using a view function as declared by user with the *morepath.App.view()* directive (and *morepath.App.json()* and *morepath.App.html()*).

This behavior is implemented using the `morepath.directive.ViewRegistry`. This builds on Reg and uses special predicates declared in *morepath.core* and registered into the Reg registry using *morepath.directive.PredicateRegistry*. The views are registered in the Reg registry too.

Views can use templates as declared with the *morepath.App.template_directory*, *morepath.App.template_loader* and *morepath.App.template_render* directives. These are registered into the *morepath.directive.TemplateEngineRegistry*.

Before a view is rendered a permission check is done for a model object and an identity. This uses the rules defined by *morepath.App.permission_rule()* are used. These are registered into the Reg registry.

Permission checks use *morepath.Request.identity*. When this is accessed the first time during a request the user's identity is constructed from information in the request according to the *morepath.App.identity_policy()*, as implemented by *morepath.authentication*.

creating links

During the rendering of the model object to a response a link can be generated for a model object by user code that invokes *morepath.Request.link*. *morepath.App* has a bunch of private functions (`morepath.App._get_path` etc) that implement constructing paths. This uses inverse path information (*morepath.path.Path*) stored into the Reg registry using *morepath.directive.PathRegistry.register_inverse_path()*.

Dependencies

Morepath uses the following dependencies:

- Webob: a request and response implementation based on WSGI.
- Reg: a generic dispatch library. This is used for all functions you can register that are aware of subclassing, in particular views.
- Dectate: the configuration engine. This is used to implement directives and how configuration actions are executed during commit.
- importscan: automatically recursively import all modules in a package.

Internal APIs

These are the internal modules used by Morepath. For more information click on the module headings to see the internal APIs. See also *morepath* for the public API and *morepath.directive* for the extension API.

`morepath.app` – App class

Here we define the Morepath application class: *morepath.App*. The application class makes available the directives to the developer. When instantiated it is a WSGI application that can be hooked into WSGI servers.

Because it is a dectate.App subclass, the class object has two special class attributes: dectate.App.dectate, which contains Dectate internals, and dectate.App.config which contains the actual configurations.

To actually serve requests it uses *morepath.publish.publish()*.

Entirely documented in *morepath.App* in the public API.

morepath.authentication – Authentication

This module defines the authentication system of Morepath.

Authentication is done by establishing an identity for a request using an identity policy registered by the *morepath.App.identity_policy()* directive.

morepath.NO_IDENTITY, *morepath.Identity*, *morepath.IdentityPolicy* are part of the public API.

See also morepath.directive.IdentityPolicyRegistry

class morepath.authentication.**NoIdentity**
The user is not yet logged in.

The request is anonymous.

morepath.autosetup – Configuration Automation

This module defines functionality to automatically configure Morepath.

morepath.scan(), *morepath.autoscan()* are part of the public API.

morepath.autosetup.**import_package**(*distribution*)
Takes a pkg_resources distribution and loads the module contained in it, if it matches the rules layed out in *morepath.autoscan()*.

morepath.autosetup.**import_package**(*distribution*)
Takes a pkg_resources distribution and loads the module contained in it, if it matches the rules layed out in *morepath.autoscan()*.

class morepath.autosetup.**DependencyMap**
A registry that tracks dependencies between distributions.

Used by morepath_packages() to find installed Python distributions that depend on Morepath, directly or indirectly.

depends(*project_name*, *on_project_name*)
Check whether project transitively depends on another.

A project depends on another project if it directly or indirectly requires the other project.

Parameters

- **project_name** – Python distribution name.
- **on_project_name** – Python distribution name it depends on.

Returns True if project_name depends on on_project_name.

load()
Fill the registry with dependency information.

relevant_dists(*on_project_name*)
Iterable of distributions that depend on project.

Dependency is transitive.

Parameters **on_project_name** – Python distribution name

Returns iterable of Python distribution objects that depend on project

morepath.autosetup.**get_module_name**(*distribution*)
Determines the module name to import from the given distribution.

If an entry point named `scan` is found in the group `morepath`, it's value is used. If not, the project_name is used.

See *morepath.autoscan()* for details and an example.

morepath.autosetup.**caller_module**(*level=2*)
Give module where calling function is defined.

Level levels deep to look up the stack frame

Returns a Python module

morepath.autosetup.**caller_package**(*level=2*)
Give package where calling function is defined.

Level levels deep to look up the stack frame

Returns a Python module (representing the `__init__.py` of a package)

morepath.compat – Python 2/3 Compatibility

Infrastructure to help make Morepath work with both Python 2 and Python 3.

It used throughout the code to make Morepath portable across Python versions.

morepath.compat.**PY3 = False**
True if we are running on Python 3

morepath.compat.**text_type**
The type used for non-bytes text.

morepath.compat.**string_types**
Can be used with `isinstance` to determine whether an object is considered to be a string, i.e. `isinstance(s, string_types)`.

morepath.compat.**bytes_**(*s*, *encoding='latin-1'*, *errors='strict'*)
Encode string if needed

If `s` is an instance of `text_type`, return `s.encode(encoding, errors)`, otherwise return `s`

morepath.converter – Convert URL variables

Convert path variables and URL parameters to Python objects.

This module contains functionality that can convert traject and URL parameters (`?foo=3`) into Python values (ints, date, etc) that are passed into model factories that are configured using the *morepath.App.path()* and *morepath.App.mount()* directives. The inverse conversion back from Python value to string also needs to be provided to support link generation.

morepath.Converter is exported to the public API.

See also *morepath.directive.ConverterRegistry*

class morepath.converter.**ListConverter**(*converter*)
How to decode from list of strings to list of objects and back.

Used morepath.converter.ConverterRegistry to handle lists of repeated names in parameters.

Used for decoding/encoding URL parameters and path variables.

Create new converter.

Parameters **converter** – the converter to use for list entries

decode(*strings*)
Decode list of strings into list of Python values.

Parameters **strings** – list of strings

Returns list of Python values

encode(*values*)
Encode list of Python values into list of strings

Parameters **values** – list of Python values.

Returns List of strings.

is_missing(*value*)
True is a given value is the missing value.

morepath.converter.**IDENTITY_CONVERTER = <morepath.converter.Converter object>**
Converter that has no effect.

String becomes string.

morepath.core – Default Morepath Configuration

This module contains default Morepath configuration shared by all Morepath applications. It is the only part of the Morepath implementation that uses directives like user of Morepath does.

It uses Morepath directives to configure:

- view predicates (for model, request method, etc), including what HTTP errors should be returned when a view cannot be matched.
- converters for common Python values (int, date, etc)
- a tween that catches exceptions raised by application code and looks up an exception view for it.
- a default exception view for HTTP exceptions defined by webob.exc, i.e. subclasses of webob.exc.HTTPException.

Should you wish to do so you could even override these directives in a subclass of *morepath.App*. We do not guarantee we won't break your code with future version of Morepath if you do that, though.

morepath.core.**date_converter**()
Converter for date.

morepath.core.**datetime_converter**()
Converter for datetime.

morepath.core.**excview_tween_factory**(*app*, *handler*)
Exception views.

If an exception is raised by application code and a view is declared for that exception class, use it.

If no view can be found, raise it all the way up – this will be a 500 internal server error and an exception logged.

morepath.core.**int_converter**()
Converter for int.

morepath.core.**method_not_allowed**(*self*, *obj*, *request*)
if request predicate not matched, method not allowed.

Fallback for *morepath.App.view()*.

morepath.core.**model_not_found**(*self*, *obj*, *request*)
if model not matched, HTTP 404.

Fallback for *morepath.App.view()*.

morepath.core.**model_predicate**(*self*, *obj*, *request*)
match model argument by class.

Predicate for *morepath.App.view()*.

morepath.core.**name_not_found**(*self*, *obj*, *request*)
if name not matched, HTTP 404.

Fallback for *morepath.App.view()*.

morepath.core.**name_predicate**(*self*, *obj*, *request*)
match name argument with request.view_name.

Predicate for *morepath.App.view()*.

morepath.core.**poisoned_host_header_protection_tween_factory**(*app*, *handler*)
Protect Morepath applications against the most basic host header poisoning attacts.

The regex approach has been copied from the Django project. To find more about this particular kind of attack have a look at the following references:

- http://skeletonscribe.net/2013/05/practical-http-host-header-attacks
- https://www.djangoproject.com/weblog/2012/dec/10/security/
- https://github.com/django/django/commit/77b06e41516d8136b56c040cba7e235b

morepath.core.**request_method_predicate**(*self*, *obj*, *request*)
match request method.

Predicate for *morepath.App.view()*.

morepath.core.**standard_exception_view**(*self*, *request*)
We want the webob standard responses for any webob-based HTTP exception.

Applies to subclasses of `webob.HTTPException`.

morepath.core.**str_converter**()
Converter for non-text str.

morepath.core.**unicode_converter**()
Converter for text.

morepath.path – Path registry

Registration of routes.

This builds on *morepath.traject*.

See also *morepath.directive.PathRegistry*

class morepath.path.**PathInfo**(*path*, *parameters*)
Abstract representation of a path.

Parameters

- **path** – a str representing a path
- **parameters** – a dict representing URL parameters.

url(*prefix*, *name*)
Turn a path into a URL.

Parameters

- **prefix** – the URL prefix to put in front of the path. This should contain something like `http://localhost`, so the URL without the path or parameter information.
- **name** – additional view name to postfix to the path.

Returns a URL with the prefix, the name and URL encoded parameters.

class morepath.path.**Path**(*path*, *factory_args*, *converters*, *absorb*)
Registered path for linking purposes.

Parameters

- **path** – the route.
- **factory_args** – the arguments for the factory function used to construct this path. This is used to determine the URL parameters for the path.
- **converters** – converters dictionary that is used to represent variables in the path.
- **absorb** – bool indicating this is an absorbing path.

__call__(*app*, *model*, *variables*)
Get path info given model and variables.

Parameters

- **app** – the app instance. Not actually used in the implementation but passed if this is registered as a method.
- **model** – model class. Not actually used in the implementation but used for dispatch in `GenericApp._class_path()`.
- **variables** – dict with the variables used in the path. each argument to the factory function should be represented.

Returns *PathInfo* instance representing the path.

get_variables_and_parameters(*variables*, *extra_parameters*)
Get converted variables and parameters.

Parameters

- **variables** – dict of variables to use in the path.
- **extra_parameters** – dict of additional parameters to use.

Returns `variables, parameters` tuple with dicts of converted path variables and converted URL parameters.

morepath.path.**get_arguments**(*callable*, *exclude*)
Introspect callable to get callable arguments and their defaults.

Parameters

- **callable** – callable object such as a function.
- **exclude** – a set of names not to extract.

Returns a dict with as keys the argument names and as values the default values (or `None` if no default value was defined).

`morepath.path.`**`filter_arguments`**(*arguments*, *exclude*)

Filter arguments.

Given a dictionary with arguments and defaults, filter out arguments in `exclude`.

Parameters

- **arguments** – arguments dict
- **exclude** – set of argument names to exclude.

Returns filtered arguments dict

`morepath.path.`**`fixed_urlencode`**(*s*, *doseq=0*)

`urllib.urlencode` fixed for ~

Workaround for Python bug:

https://bugs.python.org/issue16285

tilde should not be encoded according to RFC3986

`morepath.predicate` – Predicate registry

The *`morepath.App.predicate()`* directive lets you install predicates for function that use *`reg.dispatch_method()`*. This is used by *`morepath.core`* to install the view predicates, and you can also use it for your own functions.

This implements the functionality that drives Reg to install these predicates.

See also *`morepath.directive.PredicateRegistry`*

class `morepath.predicate.`**`PredicateInfo`**(*func*, *name*, *default*, *index*, *before*, *after*)

Used by `PredicateRegistry` internally.

Is used to store registration information on a predicate before it is registered with Reg.

`morepath.publish` – Web publisher

Functionality to turn a *`morepath.Request`* into a *`morepath.Response`* using Morepath configuration. It looks up a model instance for the request path and parameters, then looks up a view for that model object to create the response.

The publish module:

- resolves the request into a model object.
- resolves the model object and the request into a view.
- the view then generates a response.

It all starts at *`publish()`*.

`morepath.publish.`**`get_view_name`**(*stack*)

Determine view name from leftover stack of path segments

Parameters **stack** – a list of path segments left over after consuming the path.

Returns view name string or `None` if no view name can be determined.

`morepath.publish.`**`publish`**(*request*)
Handle request and return response.

It uses *`resolve_model()`* to use the information in `request` (path, request method, etc) to resolve to a model object. *`resolve_response()`* then creates a view for the request and the object.

Parameters

- **`request`** – *`morepath.Request`* instance.
- **`return`** – *`morepath.Response`* instance.

`morepath.publish.`**`resolve_model`**(*request*)
Resolve request to a model object.

This takes the path information as a stack of path segments in *`morepath.Request.unconsumed`* and consumes it step by step using `morepath.TrajectRegistry.consume()` to find the model object as declared by *`morepath.App.path()`* directive. It can traverse through mounted applications as indicated by the *`morepath.App.mount()`* directive.

Param *`morepath.Request`* instance.

Returns model object or `None` if not found.

`morepath.publish.`**`resolve_response`**(*obj*, *request*)
Given model object and request, create response.

This uses *`get_view_name()`* to set up the view name on the request object.

If no view name exist it raises `webob.exc.HTTPNotFound`.

It then uses *`morepath.App.get_view()`* to resolve the view for the model object and the request by doing dynamic dispatch.

Parameters

- **`obj`** – model object to get response for.
- **`request`** – *`morepath.Request`* instance.

Returns *`morepath.Response`* instance

`morepath.reify` – Caching property

class `morepath.reify.`**`reify`**(*wrapped*)
Cache a property.

Use as a method decorator. It operates almost exactly like the Python `@property` decorator, but it puts the result of the method it decorates into the instance dict after the first call, effectively replacing the function it decorates with an instance variable. It is, in Python parlance, a non-data descriptor. An example:

```
from morepath import reify

class Foo(object):
    @reify
    def jammy(self):
        print('jammy called')
        return 1
```

And usage of Foo:

```
>>> f = Foo()
>>> v = f.jammy
jammy called
>>> print(v)
1
>>> print(f.jammy)
1
>>> # jammy func not called the second time; it replaced itself with 1
```

morepath.request – Request and Response

Morepath request implementation.

Entirely documented in *morepath.Request* and *morepath.Response* in the public API.

morepath.settings – Settings

This module defines a registry of settings.

See *morepath.directive.SettingRegistry*

class morepath.settings.**SettingSection**
A setting section that contains setting.

morepath.template – Template Engines

This module lets you register template engines.

See *morepath.directive.TemplateEngineRegistry*

class morepath.template.**TemplateDirectoryInfo**(*key*, *directory*, *before*, *after*, *configurable*)
Used by TemplateEngineRegistry internally.

morepath.toposort – Topological sorting

Topological sort functionality.

class morepath.toposort.**Info**(*key*, *before*, *after*)
Toposorted info helper.

Base class that helps with toposorted. before and after can be lists of keys, or a single key, or None.

morepath.toposort.**toposorted**(*infos*)
Sort infos topologically.

Info object must have a key attribute, and before and after attributes that returns a list of keys. You can use *Info*.

morepath.traject – Routing

Implementation of routing.

The idea is to turn the routes into a tree, so that the routes:

```
a/b
a/b/c
a/d
```

become a tree like this:

```
a
  b
  b
    c
  d
```

Nodes in the tree can have a value attached that can be found through routing; in Morepath the value is a model instance factory.

When presented with a path, Traject traverses this internal tree.

For a description of a similar algorithm also read: http://littledev.nl/?p=99

class `morepath.traject.`**`Node`**

A node in the traject tree.

`add`(*step*)

Add a step into the tree as a child node of this node.

`add_name_node`(*step*)

Add a step into the tree as a node that doesn't match variables.

`add_variable_node`(*step*)

Add a step into the tree as a node that matches variables.

`resolve`(*segment*, *variables*)

Match a path segment, traversing this node.

Matches non-variable nodes before nodes with variables in them.

Updates the `variables` argument.

Segment a path segment

Variables variables dictionary to update.

Returns matched node, or `None` if node didn't match.

class `morepath.traject.`**`ParameterFactory`**(*parameters*, *converters*, *required*, *extra=False*)

Convert URL parameters.

Given expected URL parameters, converters for them and required parameters, create a dictionary of converted URL parameters with Python values.

Parameters

- **`parameters`** – dictionary of parameter names -> default values.
- **`converters`** – dictionary of parameter names -> converters.
- **`required`** – dictionary of parameter names -> required booleans.
- **`extra`** – should extra unknown parameters be included?

`__call__`(*request*)

Convert URL parameters to Python dictionary with values.

class morepath.traject.**Path**(*path*)
Helper when registering paths.

Used by *morepath.App.path()* to register inverse paths used for link generation.

Also used by *morepath.App.path()* for creating discriminators.

Parameters **path** – the route.

discriminator()
Creates a unique discriminator for the path.

interpolation_str()
Create a string for interpolating variables.

Used for link generation (inverse).

variables()
Get the variables used by the path.

Returns a list of variable names

class morepath.traject.**Step**(*s*, *converters=None*)
A single step in the tree.

Parameters

- **s** – the path segment, such as `'foo'` or `'{variable}'` or `'foo{variable}bar'`.
- **converters** – dict of converters for variables.

__eq__(*other*)
True if this step is the same as another.

__ge__(*other*)
x.__ge__(y) <==> x>=y

__gt__(*other*)
x.__gt__(y) <==> x>y

__le__(*other*)
x.__le__(y) <==> x<=y

__lt__(*other*)
Used for inserting steps in correct place in the tree.

The order in which a step is inserted into the tree compared to its siblings affects which step preferentially matches first.

In Traject, steps that contain no variables match before steps that do contain variables. Steps with more specific variables sort before steps with more general ones, i.e. `prefix{foo}` sorts before `{foo}` as `prefix{foo}` is more specific.

__ne__(*other*)
True if this step is not equal to another.

discriminator_info()
Information needed to construct path discriminator.

has_variables()
True if there are any variables in this step.

match(*s*, *variables*)
Match this step with actual path segment.

Parameters

- **s** – path segment to match with
- **variables** – variables dictionary to update with new converted variables that are found in this segment.

Returns bool. The bool indicates whether `s` matched with the step or not.

validate()
Validate whether step makes sense.

Raises *morepath.error.TrajectError* if there is a problem with the segment.

validate_parts()
Check whether all non-variable parts of the segment are valid.

Raises *morepath.error.TrajectError* if there is a problem with the segment.

validate_variables()
Check whether all variables of the segment are valid.

Raises *morepath.error.TrajectError* if there is a problem with the variables.

class morepath.traject.**StepNode**(*step*)
A node that is also a step in that it can match.

Parameters **step** – the step

match(*segment*, *variables*)
Match a segment with the step.

class morepath.traject.**TrajectRegistry**
Tree of route steps.

add_pattern(*path*, *model_factory*, *defaults=None*, *converters=None*, *absorb=False*, *required=None*, *extra=None*, *code_info=None*)
Add a route to the tree.

Parameters

- **path** – route to add.
- **model_factory** – the factory used to construct the model instance
- **defaults** – mapping of URL parameters to default value for parameter
- **converters** – converters to store with the end step of the route
- **absorb** – does this path absorb all segments
- **required** – list or set of required URL parameters
- **extra** – bool indicating whether extra parameters are expected
- **code_info** – `dectate.CodeInfo` instance describing the code line that registered this path.

consume(*request*)
Consume a stack given route, returning object.

Removes the successfully consumed path segments from *morepath.Request.unconsumed*.

Extracts variables from the path and URL parameters from the request.

Then constructs the model instance given this information. (or *morepath.App* instance in case of mounted apps).

Parameters **request** – the request to consume segments from and to retrieve URL parameters from.

Returns the model instance that can be found, or `None` if no model instance exists for this sequence of segments.

`morepath.traject.`**`create_path`**(*segments*)
Builds a path from a list of segments.

Parameters **`stack`** – a list of segments

Returns a path

`morepath.traject.`**`create_variables_re`**(*s*)
Create regular expression that matches variables from route segment.

Parameters **`s`** – a route segment with variables in it.

Returns a regular expression that matches with variables for the route.

`morepath.traject.`**`generalize_variables`**(*s*)
Generalize a route segment.

Parameters **`s`** – a route segment.

Returns a generalized route where all variables are empty ({}).

`morepath.traject.`**`interpolation_str`**(*s*)
Create a Python string with interpolation variables for a route segment.

Given `a{foo}b`, creates `a%sb`.

`morepath.traject.`**`is_identifier`**(*s*)
Check whether a variable name is a proper identifier.

Parameters **`s`** – variable

Returns True if variable is an identifier.

`morepath.traject.`**`normalize_path`**(*path*)
Converts path into normalized path.

Rules:

•Collapse dots:

```
>>> normalize_path('/../blog')
'/blog'
```

•Ensure absolute paths:

```
>>> normalize_path('./site')
'/site'
```

•Remove double-slashes:

```
>>> normalize_path('//index')
'/index'
```

For example:

```
>>> normalize_path('../static//../app.py')
'/app.py'
```

Parameters **`path`** – path string to parse

Returns normalized path.

morepath.traject.**parse_path**(*path*)
Parses path, creates normalized segment list.

Dots are collapsed:

```
>>> parse_path('../static//../app.py')
['app.py']
```

Parameters **path** – path string to parse

Returns normalized list of path segments.

morepath.traject.**parse_variables**(*s*)
Parse variables out of a segment.

Raised a *morepath.error.TrajectError* if a variable is not a valid identifier.

Parameters **s** – a path segment

Returns a list of variables.

morepath.traject.**IDENTIFIER = <_sre.SRE_Pattern object>**
regex for a valid variable name in a route.

same rule as for Python identifiers.

morepath.traject.**PATH_VARIABLE = <_sre.SRE_Pattern object>**
regex to find curly-brace marked variables `{foo}` in routes.

morepath.tween – Tweens

Tweens are lightweight middleware using webob.

A tween is a function that takes a *morepath.Request* and returns a *morepath.Response*. A tween factory is a function that given an application instance and tween constructs another tween that wraps it.

Used by *morepath.App.tween_factory()*.

See also *morepath.directive.TweenRegistry*

morepath.view – View registry

Rendering views.

A view is a function that returns something. This can be a *morepath.Response*, but it can also be a structure (such a dict) that should be rendered *to* a response. If the view is a JSON view this dumps this structure as JSON. If the view is a HTML view this structure can be converted to HTML using a template.

morepath.render_json(), *morepath.render_html()* and *morepath.redirect()* are members of the public API.

class morepath.view.**View**(*func*, *render=None*, *load=None*, *permission=None*, *internal=False*, *code_info=None*)
A view as registered with *morepath.App.get_view()*.

Parameters

- **func** – view function. Given a model instance and a request argument, this function must return either a structure that can be turned into a response or a response.
- **render** – a function used to render view function return value as a response.

- **load** – a function used to load the body data into a third argument to the view.
- **permission** – permission class that the identity must have according to permission rules. If the view doesn't have the permission access is forbidden.
- **internal** – bool to indicate whether this view is internal. If the view is internal you can use it with *morepath.Request.view()* but it doesn't have a URL and will be 404 Not Found.

__call__(*app*, *obj*, *request*)
Render a model instance.

If view is internal it cannot be rendered.

If the identity does not have the permission for this object according to the permission rules then webob.exc.HTTPForbidden is raised.

Any functions specified using *morepath.Request.after()* are run against the response once it is created, if that response is not an error.

Parameters

- **obj** – the model instance
- **request** – the request

Returns A webob.response.Response instance.

morepath.view.**render_view**(*content*, *request*)
Default render function for view if none was supplied.

This just assumes the content is a string and renders it into a response.

Parameters

- **content** – content as returned by view function.
- **request** – request object

Returns a response instance with the content.

morepath.error and *morepath.pdbsupport* are documented as part of the public API.

Part VI

History

The change log and how to use it.

CHAPTER 31

History of Morepath

For more recent changes, see *CHANGES*.

Morepath was originally created by Martijn Faassen, but now has a team of developers (see `CREDITS.txt` in the project).

The genesis of Morepath is complex and involves a number of projects.

Web Framework Inspirations

Morepath was inspired by Zope, in particular its component architecture; a reimagined version of this is available in Reg, a core dependency of Morepath.

An additional inspiration was the Grok web framework Martijn conceived of and helped to create, which was based on Zope 3 technologies, and Pyramid, a reimagined version of Zope 3, created by Chris McDonough.

Pyramid in particular has been the source of a lot of ideas, including bits of implementation.

Once the core of Morepath had been created Martijn found there had been quite a bit of parallel evolution with Flask. Flask served as a later inspiration in its capabilities and documentation. Morepath also used Werkzeug (basis for Flask) for a while to implement its request and response objects, but eventually we found WebOb the better fit for Morepath and switched to that.

Configuration system

In 2006 Martijn co-founded the Grok web framework. The fundamental configuration mechanism this project uses was distilled into the Martian library:

https://pypi.python.org/pypi/martian

Martian was reformulated by Chris McDonough (founder of the Pyramid project) into Venusian, a simpler, decorator based approach:

https://pypi.python.org/pypi/venusian

Morepath originally used Venusian as a foundation to its configuration system. It is good that Venusian defers execution of decorators until after import-time, but Venusian also makes setup more difficult to reason about for users than simply registering the decorator with the configuration system during import-time.

Morepath's configuration system had grown over time and had grown a few hacks here and there. Removing Venusian was not simple as a result. The configuration system of Morepath was also underdocumented, a long standing issue in Morepath.

So in 2016 we extracted Morepath's configuration system into its own reusable project, called dectate. Martijn also extensively refactored it and removed the Venusian dependency. Morepath now uses Dectate as a clean and well-documented configuration system.

Routing system

In 2009 Martijn wrote a library called Traject:

https://pypi.python.org/pypi/traject

Martijn was familiar with Zope traversal. Zope traversal matches a URL with an object by parsing the URL and going through an object graph step by step to find the matching object. This works well for objects stored in an object database, as they're already in such a graph. Martijn tried to make this work properly with a relational database exposed through an ORM, but noticed that he had to adjust the object mapping too much just to please the traversal system.

This led Martijn to a routing system, so expose the relational database objects to a URL. But he didn't want to give up some nice properties of traversal, in particular that for any object that you can traverse to you can also generate a URL. He also wanted to maintain a separation between models and views. This led to the creation of Traject.

Martijn used Traject successfully in a number of projects (based on Grok), and also ported Traject to JavaScript as part of the Obviel client-side framework. While Traject is fairly web-framework independent, to Martijn's knowledge Traject hasn't found much adoption elsewhere.

Morepath contains a further evolution of the Traject concept (though not the Traject library directly).

Reg

In early 2010 Martijn started the iface project with Thomas Lotze. In 2012 Martijn started the Crom project. Finally he combined them into the Comparch project in 2013. He then renamed Comparch to Reg, and finally converted Reg to a generic function implementation.

In late 2014 there was another major change in Reg, when Martijn generalized it into a predicate dispatch implementation. In the summer of 2016 Stefano Taschini had a bunch of good ideas and we managed to simplify Reg's implementation and got rid of its implicitness by introducing dispatch methods.

See Reg's history section for more information on its history. The Reg project provides the fundamental registries that Morepath builds on.

Publisher

In 2010 Martijn wrote a system called Dawnlight:

https://bitbucket.org/faassen/dawnlight

It was the core of an object publishing system with a system to find a model and a view for that model, based on a path. It used some concepts Martijn had learned while implementing Traject (a URL path can be seen as a stack that's being consumed), and it was intended to be easy to plug in Traject. Martijn didn't use Dawnlight himself, but it was adopted by the developers of the Cromlech web framework (Souheil Chelfouh and Alex Garel):

http://pypi.dolmen-project.org/pypi/cromlech.dawnlight

Morepath contains a reformulation of the Dawnlight system, particularly in its publisher module.

Combining it all

In 2013 Martijn started to work with CONTACT Software. They encouraged me to rethink these various topics. This led Martijn to combine these lines of development into Morepath: Reg registries, decorator-based configuration, and traject-style publication of models and resources.

Spinning a Web Framework

In the fall of 2013 Martijn gave a keynote speech at PyCon DE about the creative processes behind Morepath, called "Spinning a Web Framework":

CHAPTER 32

CHANGES

0.18 (2017-03-17)

- **New**: The *load‘* API, which allows you to define how incoming JSON (through a POST, PUT or PATH request) will be converted to a Python object and how it will be validated. This feature lets you plug in external serialization and validation libraries, such as Marshmallow, Colander, Cerberus, Jsonschema or Voluptuous.
- **Removed**: `morepath.body_model_predicate` is removed from the Morepath API together with the `morepath.App.load_json` directive and the `morepath.request.body_obj` property. If you use the `load_json` directive, this functionality has been moved to a separate more.body_model package. Use this package instead by subclassing your App from *more.body_model.BodyModelApp*.
- Uploading huge files lead to excessive memory consumption as the whole body was consumed for no good reason. This is now fixed.

  See #504
- Fixes link prefix not applying to mounted applications.

  See ‘#516’

0.17 (2016-12-23)

- **Removed**: The class `morepath.ViewRegistry` is gone.
- Upload universal wheels to pypi during release.
- Refactored and simplified implementation of `ConverterRegistry`.
- Bugfix: exception views in mounted apps weren’t looked up correctly anymore.
- Adds compatibility with WebOb 1.7.
- Removed extra spaces after the colon in json. For example: {“foo”: “bar”} is now {“foo”:”bar”}.

- Morepath now keeps track of what code was used to resolve a path and a view. You use more.whytool to get a command line tool that provides insight in what code was used for a request.

0.16.1 (2016-10-04)

- Adjust `setup.py` to require Reg 0.10 and Dectate 0.12, otherwise Morepath won't work properly.

0.16 (2016-10-04)

Release highlights

- A new, cleaner and faster implementation of Reg underlies this version of Morepath. It turns generic functions into methods on the `App` class, and removes implicit behavior entirely.

  This has some impact if you used the low-level `function` directive or if you defined your own predicates with the `predicate` and `predicate_fallback` directives, see details below.
- A new build environment based around virtualenv and pip. We've removed the old buildout-based build environment. `doc/developing.rst` has much more.
- Performance work boosts performance of Morepath significantly.

Removals & Deprecations

- **Removed**: `morepath.remember_identity` is removed from the Morepath API.

  Use

  ```
  request.app.remember_identity(response, request, identity)
  ```

  Instead of

  ```
  remember_identity(response, request, identity, lookup=request.lookup)
  ```

- **Removed**: `morepath.forget_identity` is removed from the Morepath API.

  Use

  ```
  request.app.forget_identity(response, request)
  ```

  Instead of

  ```
  morepath.forget_identity(response, request, lookup=request.lookup)
  ```

- **Removed** `morepath.settings` is removed from the Morepath API.

  Use the `morepath.App.settings` property instead. You can access this through `app.settings`. You can access this through `request.app.settings` if you have the request. The following directives now get an additional optional first argument called `app`: `permission_rule`, `verify_identity`, `dump_json`, `load_json`, `link_prefix` and the `variables` function passed to the path directive.

- **Removed** `morepath.enable_implicit` and `morepath.disable_implicit` are both removed from the Morepath API.

  Morepath now uses generic *methods* on the application class. The application class determines the context used.

- **Removed** We previously used buildout to install a development environment for Morepath. We now use pip. See `doc/developing.rst` for details, and also below.

Features

- **Breaking change** Dectate used to support the `directive` pseudo-directive to let you define directives. But this could lead to import problems if you forgot to import the module where the pseudo-directives are defined before using them. In this release we define the directives directly on the `App` class using the new `dectate.directive` mechanism, avoiding this problem.

  If you have code that defines new directives, you have to adjust your code accordingly; see the Dectate changelog for more details.

- **Breaking change** Previously Morepath used Reg's dispatch functions directly, with a mechanism to pass in a `lookup` argument to a dispatch function to control the application context. The lookup was maintained on `App.lookup`. Tests were to pass the lookup explicitly. Reg also maintained this lookup in a thread-local variable, and any dispatch call that did not have a explicit lookup argument passed in used this implicit lookup directly.

  Reg has undergone a major refactoring which affects Morepath. As a result, Morepath is faster and dispatch code becomes more Pythonic. The concept of lookup is gone: no more lookup argument, `app.lookup` or implicit lookup. Instead, Morepath now makes use of dispatch *methods* on the application. The application itself provides the explicit dispatch context. See #448 for the discussion leading up to this change.

  Most Morepath application and library projects should continue to work unchanged, but some changes are necessary if you used some advanced features:

  - If in your code you call a generic function from `morepath.generic` directly it won't work anymore. Call the equivalent method on the app instance instead.
  - If you pass through the `lookup` argument explicitly, remove this. Calling the dispatch method on the app instance is enough to indicate context.
  - If you defined a generic function in your code, you should move it to a `morepath.App` subclass instead and use `morepath.dispatch_method` instead of `reg.dispatch`. Using `reg.dispatch_method` directly is possible but not recommended: `morepath.dispatch_method` includes caching behavior that speeds up applications. For example:

    ```
    class MyApp(morepath.App):
        @morepath.dispatch_method('obj')
        def my_dispatch(self, obj):
            pass
    ```

  - The `function` directive has been replaced by the `method` directive, where you indicate the dispatch method on the first argument. For example:

    ```
    @App.method(MyApp.my_dispatch, obj=Foo)
    def my_dispatch_impl(app, obj):
        return "Implementation for Foo"
    ```

  - The `predicate` directive can be used to install new predicates for dispatch methods. The first argument should be a reference to the dispatch method, for instance:

    ```
    @App.predicate(App.get_view, name='model', default=None,
                   index=ClassIndex)
    def model_predicate(obj):
      return obj.__class__
    ```

There is a new public method called `App.get_view` that you can install view predicates on.

– The `predicate_fallback` directive gets a reference to the method too. The decorated function needs to take the same arguments as the dispatch method; previously it could be a subset. So for example:

```
@App.predicate_fallback(App.get_view, model_predicate)
def model_not_found(self, obj, request):
    raise HTTPNotFound()
```

Where `self` refers to the app instance.

Bug fixes

- Fix code_examples path for doctests with tox.

Build environment

- We now use virtualenv and pip instead of buildout to set up the development environment. The development documentation has been updated accordingly. Also see issues #473 and #484.
- Have the manifest file for source distribution include all files under VCS.
- As we reached 100% code coverage for pytest, coveralls integration was replaced by the `--fail-under=100` argument of `coverage report` in the tox coverage test.

Other

- Refactored traject routing code with an eye on performance.
- Use abstract base classes from the standard library for `morepath.IdentityPolicy`.
- Reorganize the table of contents of the documentation into a hierarchy (#468).
- Expand the test suite to cover `morepath.Request.reset`, loop detection for deferred class links, dispatching of `@App.verify_identity`-decorated functions on the `identity` argument (#464). Coverage ratio is now 100%.

0.15 (2016-07-18)

Removals & Deprecations

- **Removed**: `morepath.autosetup` and `morepath.autocommit` are both removed from the Morepath API.

  Use `autoscan`. Also use new explicit `App.commit` method, or rely on Morepath automatically committing during the first request. So instead of:

```
morepath.autosetup()
morepath.run(App())
```

  you do:

```
morepath.autoscan()
App.commit()  # optional
morepath.run(App())
```

- **Removed**: the `morepath.security` module is removed, and you cannot import from it anymore. Change imports from it to the public API, so go from:

```
from morepath.security import NO_IDENTITY
```

to:

```
from morepath import NO_IDENTITY
```

- **Deprecated** `morepath.remember_identity` and `morepath.forget_identity` are both deprecated.

  Use the `morepath.App.remember_identity` and `morepath.App.forget_identity` methods, respectively.

  Instead of

```
remember_identity(response, request, identity, lookup=request.lookup)
...
morepath.forget_identity(response, request, lookup=request.lookup)
```

  you do:

```
request.app.remember_identity(response, request, identity)
...
request.app.forget_identity(response, request)
```

- **Deprecated** `morepath.settings` is deprecated.

  Use the `morepath.App.settings` property instead.

- **Deprecated** `morepath.enable_implicit` and `morepath.disable_implicit` are both deprecated.

  You no longer need to choose between implicit or explicit lookup for generic functions, as the generic functions that are part of the API have all been deprecated.

Features

- Factored out new `App.mounted_app_classes()` class method which can be used to determine the mounted app classes after a commit. This can used to get the argument to `dectate.query_tool` if the commit is known to have already been done earlier.
- The `morepath.run` function now takes command-line arguments to set the host and port, and is friendlier in general.
- Add `App.init_settings` for pre-filling the settings registry with a python dictionary. This can be used to load the settings from a config file.
- Add a `reset` method to the `Request` class that resets it to the state it had when request processing started. This is used by `more.transaction` to reset request processing when it retries a transaction.

Bug fixes

- Fix a bug where a double slash at the start of a path was not normalized.

Cleanups

- Cleanups and testing of `reify` functionality.
- More doctests in the narrative documentation.
- A few small performance tweaks.
- Remove unused imports and fix pep8 in core.py.

Other

- Add support for Python 3.5 and make it the default Python environment.

0.14 (2016-04-26)

- **New** We have a new chat channel available. You can join us by clicking this link:

  https://discord.gg/0xRQrJnOPiRsEANa

  Please join and hang out! We are retiring the (empty) Freenode #morepath channel.

- **Breaking change**: Move the basic auth policy to `more.basicauth` extension extension. Basic auth is just one of the authentication choices you have and not the default. To update code, make your project depend on `more.basicauth` and import `BasicAuthIdentityPolicy` from `more.basicauth`.

- **Breaking change**: Remove some exception classes that weren't used: `morepath.error.ViewError`, `morepath.error.ResolveError`. If you try to catch them in your code, just remove the whole `except` statement as they were never raised.

- **Deprecated** Importing from `morepath.security` directly. We moved a few things from it into the public API: `enable_implicit`, `disable_implicit`, `remember_identity`, `forget_identity`, `Identity`, `IdentityPolicy`, `NO_IDENTITY`. Some of these were already documented as importable from `morepath.security`. Although importing from `morepath.security` won't break yet, you should stop importing from it and import directly from `morepath` instead.

- **Deprecated** `morepath.autosetup` and `morepath.autocommit` are both deprecated.

  Use `autoscan`. Also use new explicit `App.commit` method, or rely on Morepath automatically committing during the first request. So instead of:

  ```
  morepath.autosetup()
  morepath.run(App())
  ```

  you do:

  ```
  morepath.autoscan()
  App.commit()  # optional
  morepath.run(App())
  ```

- **Breaking change** Extensions that imported `RegRegistry` directly from `morepath.app` are going to be broken. This kind of import:

  ```
  from morepath.app import RegRegistry
  ```

  needs to become:

```
from morepath.directive import RegRegistry
```

This change was made to avoid circular imports in Morepath, and because `App` did not directly depend on `RegRegistry` anymore.

- **Breaking change**: the `variables` function for the `path` directive *has* to be defined taking a first `obj` argument. In the past it was possible to define a `variables` function that took no arguments. This is now an error.
- Introduce a new `commit` method on `App` that commits the App and also recursively commits all mounted apps. This is more explicit than `autocommit` and less verbose than using the lower-level `dectate.commit`.
- Automatic commit of the app is done during the first request if the app wasn't committed previously. See issue #392.
- Introduce a deprecation warnings (for `morepath.security`, `morepath.autosetup`) and document how a user can deal with such warnings.
- Adds host header validation to protect against header poisoning attacks.

  See https://github.com/morepath/morepath/issues/271

  You can use `morepath.HOST_HEADER_PROTECTION` in your own tween factory to wrap before or under it.
- Refactor internals of publishing/view engine. Reg is used more effectively for view lookup, order of some parameters is reversed for consistency with public APIs.
- Document the internals of Morepath, see implementation document. This includes docstrings for all the internal APIs.
- The framehack module was merged into `autosetup`. Increased the coverage to this module to 100%.
- New cookiecutter template for Morepath, and added references in the documentation for it.

  See https://github.com/morepath/morepath-cookiecutter
- Test cleanup; scan in many tests turns out to be superfluous. Issue #379
- Add a test that verifies we can instantiate an app before configuration is done. See issue #378 for discussion.
- Started doctesting some of the docs.
- Renamed `RegRegistry.lookup` to `RegRegistry.caching_lookup` as the `lookup` property was shadowing a lookup property on `reg.Registry`. This wasn't causing bugs but made debugging harder.
- Refactored link generation. Introduce a new `defer_class_links` directive that lets you defer link generation using `Request.class_link()` in addition to `Request.link()`. This is an alternative to `defer_links`, which cannot support `Request.class_link`.
- Morepath now has extension API docs that are useful when you want to create your own directive and build on one of Morepath's registries or directives.
- A friendlier `morepath.run` that tells you how to quit it with `ctrl-C`.
- A new document describing how to write a test for Morepath-based applications.
- Document how to create a Dectate-based command-line query tool that lets you query Morepath directives.
- Uses the topological sort implementation in Dectate. Sort out a mess where there were too many `TopologicalSortError` classes.

0.13.2 (2016-04-13)

- Undid change in 0.13.1 where `App` could not be instantiated if not committed, as ran into real-world code where this assumption was broken.

0.13.1 (2016-04-13)

- Enable queries by the Dectate query tool.
- Document `scan` function in API docs.
- Work around an issue in Python where ~ (tilde) is quoted by `urllib.quote` & `urllib.encode`, even though it should not be according to the RFC, as ~ is considered unreserved.

  https://www.ietf.org/rfc/rfc3986.txt
- Document some tricks you can do with directives in a new "Directive tricks" document.
- Refactor creation of tweens into function on TweenRegistry.
- Update the REST document; it was rather old and made no mention of `body_model`.
- Bail out with an error if an App is instantiated without being committed.

0.13 (2016-04-06)

- **Breaking change**. Morepath has a new, extensively refactored configuration system based on dectate and importscan. Dectate is an extracted, and heavily refactored version of Morepath's configuration system that used to be in `morepath.config` module. It's finally documented too!

  Dectate and thus Morepath does not use Venusian (or Venusifork) anymore so that dependency is gone.

  Code that uses `morepath.autosetup` should still work.

  Code that uses `morepath.setup` and scans and commits manually needs to change. Change this:

```
from morepath import setup

config = morepath.setup()
config.scan(package)
config.commit()
```

  into this:

```
import morepath

morepath.scan(package)
morepath.autocommit()
```

  Similarly `config.scan()` without arguments to scan its own package needs to be rewritten to use `morepath.scan()` without arguments.

  Anything you import directly now does not need to be scanned anymore; the act of importing a module directly registers the directives with Morepath, though as before they won't be active until you commit. But scanning something you've imported before won't do any harm.

The signature for `morepath.scan` is somewhat different than that of the old `config.scan`. There is no third argument `recursive=True` anymore. The `onerror` argument has been renamed to `handle_error` and has different behavior; the importscan documentation describes the details.

If you were writing tests that involve Morepath, the old structure of the test was:

```
import morepath

def test_foo():
    config = morepath.setup()

    class App(morepath.App):
        testing_config = config

    ... use directives on App ...

    config.commit()

    ... do asserts ...
```

This now needs to change to:

```
import morepath

def test_foo():
    class App(morepath.App):
        pass

    ... use directives on App ...

    morepath.commit([App])

    ... do asserts ...
```

So, you need to use the `morepath.commit()` function and give it a list of the application objects you want to commit, explicitly. `morepath.autocommit()` won't work in the context of a test.

If you used a test that scanned code you need to adjust it too, from:

```
import morepath
import some_package

def test_foo():
    config = morepath.setup()

    config.scan(some_package)

    config.commit()

    ... do asserts ...
```

to this:

```
import morepath
import some_package

def test_foo():
    morepath.scan(some_package)
    morepath.commit([some_package.App])
```

```
    ... do asserts ...
```

Again you need to be explicit and use `morepath.commit` to commit those apps you want to test.

If you had a low-level reference to `app.registry` in your code it will break; the registry has been split up and is now under `app.config`. If you want access to `lookup` you can use `app.lookup`.

If you created custom directives, the way to create directives is now documented as part of the dectate project. The main updates you need to do are:

- subclass from *dectate.Action* instead of *morepath.Directive*.
- no more `app` first argument.
- no `super` call is needed anymore in `__init__`.
- add a `config` class variable to declare the registries you want to affect. Until we break up the main registry this is:

```
from morepath.app import Registry

...
config = { 'registry': Registry }
```

- reverse the arguments to `perform`, so that the object being registered comes first. So change:

```
def perform(self, registry, obj):
    ...
```

  into:

```
def perform(self, obj, registry):
    ...
```

  But instead of `registry` use the registry you set up in your action's `config`.

- no more `prepare`. Do error checking inside the `perform` method and raise a `DirectiveError` if something is wrong.

  If you created sub-actions from `prepare`, subclass from *dectate.Composite* instead and implement an `actions` method.

- `group_key` method has changed to `group_class` class variable.

If you were using `morepath.sphinxext` to document directives using Sphinx autodoc, use `dectate.sphinxext` instead.

- **Breaking change** If you want to use Morepath directives on `@staticmethod`, you need to change the order in which these are applied. In the past:

```
@App.path(model=Foo, path='bar')
@staticmethod
def get_foo():
    ....
```

But now you need to write:

```
@staticmethod
@App.path(model=Foo, path='bar')
```

```
def get_foo():
    ....
```

- **Breaking change** You cannot use a Morepath `path` directive on a `@classmethod` directly anymore. Instead you can do this:

```
class Foo(object):
    @classmethod
    def get_something():
        pass

@App.path('/', model=Something)(Foo.get_something)
```

- **Breaking change**. Brought *app.settings* back, a shortcut to the settings registry. If you use settings, you need to replace any references to `app.registry.settings` to `app.settings`.
- Add *request.class_link*. This lets you link using classes instead of instances as an optimization. In some cases instantiating an object just so you can generate a link to it is relatively expensive. In that case you can use *request.class_link* instead. This lets you link to a model class and supply a *variables* dictionary manually.
- **Breaking change**. In Morepath versions before this there was an class attribute on `App` subclasses called `registry`. This was a giant mixed registry which subclassed a lot of different registries used by Morepath (reg registry, converter registry, traject registry, etc). The Dectate configuration system allows us to break this registry into a lot of smaller interdependent registries that are configured in the `config` of the directives.

  While normally you shouldn't be, if you were somehow relying on `App.registry` in your code you should now rewrite it to use `App.config.reg_registry`, `App.config.setting_registry`, `App.config.path_registry` etc.

0.12 (2016-01-27)

- **Breaking change**. The `request.after` function is now called even if the response was directly created by the view (as opposed to the view returning a value to be rendered by morepath). Basically, `request.after` is now guaranteed to be called if the response's HTTP status code lies within the 2XX-3XX range.

  See https://github.com/morepath/morepath/issues/346
- Fixed a typo in the *defer_link* documentation.
- Morepath's link generation wasn't properly quoting paths and parameters in all circumstances where non-ascii characters or URL-quoted characters were used. See issue #337.
- Morepath could not handle varargs or keyword arguments properly in path functions. Now bails out with an error early during configuration time. To fix existing code, get rid of any `*args` or `**kw`.
- Morepath could not properly generate links if a path directive defines a path variable for the path but does not actually use it in the path function. Now we complain during configuration time. To fix existing code, add all variables that are defined in the path (i.e. `{id}`) to the function signature.
- Certain errors (`ConfigError`) were not reporting directive line number information. They now do.
- Better `ConfigError` reporting when `setting_section` is in use.
- Removed the unused `request` parameter from the `link` method in `morepath.request`. See issue #351.
- Require venusifork 2.0a3. This is a hacked version which works around some unusual compatibility issues with `six`.

0.11.1 (2015-06-29)

- setuptools has the nasty habit to change underscores in project names to minus characters. This broke the new autoscan machinery for packages with an underscore in their name (such as *morepath_sqlalchemy*). This was fixed.

0.11 (2015-06-29)

- **Breaking change**. The `morepath.autoconfig` and `morepath.autosetup` methods had to be rewritten. Before, Morepath was unable to autoload packages installed using `pip`.

  As a result, Morepath won't be able to autoload packages if the setup.py name differs from the name of the distributed package or module.

  For example: A package named `my-app` containing a module named `myapp` won't be automatically loaded anymore.

  Packages like this need to be loaded manually now:

  ```
  import morepath
  import myapp

  config = morepath.setup()
  config.scan(myapp)
  config.commit()
  ```

  See https://github.com/morepath/morepath/issues/319

- The `config.scan` method now excludes 'test' and 'tests' directories by default.

  See https://github.com/morepath/morepath/issues/326

- The `template_directory` directive will no longer inspect the current module if the template directory refers to an absolute path. This makes it easier to write tests where the current module might not be available.

  Fixes https://github.com/morepath/morepath/issues/299

- The `identity_policy` passes `settings` to the function if it defines such an argument. This way an identity policy can be created that takes settings into account.

  See https://github.com/morepath/morepath/issues/309

- Dots in the request path are now always normalized away. Before, Morepath basically relied on the client to do this, which was a potential security issue.

  See https://github.com/morepath/morepath/issues/329

- Additional documentation on the Morepath config system: http://morepath.readthedocs.org/en/latest/configuration.html

- Additional documentation on how to serve static images in https://morepath.readthedocs.org/en/latest/more.static.html

- Move undocumented `pdb` out of `__init__.py` as it could sometimes trip up things. Instead documented it in the API docs in the special *morepath.pdbsupport* module.

  https://github.com/morepath/morepath/issues/328

0.10 (2015-04-09)

- Server-side templating language support: there is now a `template` argument for the `html` directive (and `view` and `json`). You need to use a plugin to add particular template languages to your project, such as `more.chameleon` and `more.jinja2`, but you can also add your own.

  See http://morepath.readthedocs.org/en/latest/templates.html

- Add a new "A Review of the Web" document to the docs to show how Morepath fits within the web.

  http://morepath.readthedocs.org/en/latest/web.html

- The publisher does not respond to a `None` render function anymore. Instead, the `view` directive now uses a default `render_view` if `None` is configured. This simplifies the publisher guaranteeing a `render` function always exists.

  Fixes https://github.com/morepath/morepath/issues/283

- Introduce a `request.resolve_path` method that allows you to resolve paths to objects programmatically.

- Modify `setup.py` to use `io.open` instead of `open` to include the README and the CHANGELOG and hardcode UTF-8 so it works on all versions of Python with all default encodings.

- Various documentation fixes.

0.9 (2014-11-25)

- **Breaking change**. In previous releases of Morepath, Morepath did not include the full hostname in generated links (so `/a` instead of `http://example.com/a`). Morepath 0.9 does include the full hostname in generated links by default. This to support the non-browser client use case better. In the previous system without fully qualified URLs, client code needs to manually add the base of links itself in order to be able to access them. That makes client code more complicated than it should be. To make writing such client code as easy as possible Morepath now generates complete URLs.

  This should not break any code, though it can break tests that rely on the previous behavior. To fix `webtest` style tests, prefix the expected links with `http://localhost/`.

  If for some reason you want the old behavior back in an application, you can use the `link_prefix` directive:

```
@App.link_prefix()
def my_link_prefix(request):
    return '' # prefix nothing again
```

- Directives are now logged to the `morepath.directive` log, which is using the standard Python `logging` infrastructure. See http://morepath.readthedocs.org/en/latest/logging.html

- Document `more.forwarded` proxy support in http://morepath.readthedocs.org/en/latest/paths_and_linking.html

- Document behavior of `request.after` in combination with directly returning a response object from a view.

- Expose `body_model_predicate` to the public Morepath API. You can now say your predicate comes after it.

- Expose `LAST_VIEW_PREDICATE` to the Morepath API. This is the last predicate defined by the Morepath core.

- Update the predicate documentation.

- Updated the more.static doc to reflect changes in it.

- Fix doc for grouping views with the `with` statement.
- Suggest a few things to try when your code doesn't appear to be scanned properly.
- A new view predicate without a fallback resulted in an internal server error if the predicate did not match. Now it results in a 404 Not Found by default. To override this default, define a predicate fallback.

0.8 (2014-11-13)

- **Breaking change**. Reg 0.9 introduces a new, more powerful way to create dispatch functions, and this has resulted in a new, incompatible Reg API.

  Morepath has been adjusted to make use of the new Reg. This won't affect many Morepath applications, and they should be able to continue unchanged. But some Morepath extensions and advanced applications may break, so you should be aware of the changes.

  - The `@App.function` directive has changed from this:

```
class A(object):
    pass

class B(object):
    pass

@reg.generic
def dispatch_function(a, b):
    pass

@App.function(dispatch_function, A, B)
def dispatched_to(a, b):
    return 'dispatched to A and B'
```

    to this:

```
class A(object):
    pass

class B(object):
    pass

@reg.dispatch('a', 'b')
def dispatch_function(a, b):
    pass

@App.function(dispatch_function, a=A, b=B)
def dispatched_to(a, b):
    return 'dispatched to A and B'
```

    The new system in Reg (see its docs) is a lot more flexible than what we had before. When you use `function` you don't need to know about the order of the predicates anymore – this is determined by the arguments to `@reg.dispatch()`. You can now also have function arguments that Reg ignores for dispatch.

  - The `@App.predicate` and `@App.predicate_fallback` directive have changed. You can now install custom predicates and fallbacks for *any* generic function that's marked with `@reg.dispatch_external_predicates()`. The Morepath view code has been simplified to be based

on this, and is also more powerful as it can now be extended with new predicates that use predicate-style dispatch.

- Introduce the `body_model` predicate for views. You can give it the class of the `request.body_obj` you want to handle with this view. In combination with the `load_json` directive this allows you to write views that respond only to the POSTing or PUTing of a certain type of object.
- Internals refactoring: we had a few potentially overridable dispatch functions in `morepath.generic` that actually were never overridden in any directives. Simplify this by moving their implementation into `morepath.publish` and `morepath.request`. `generic.link`, `generic.consume` and `generic.response` are now gone.
- Introduce a `link_prefix` directive that allows you to set the URL prefix used by every link generated by the request.
- A bug fix in `request.view()`; the `lookup` on the `request` was not properly updated.
- Another bug fix in `request.view()`; if `deferred_link_app` app is used, `request.app` should be adjusted to the app currently being deferred to.
- `request.after` behavior is clarified: it does not run for any exceptions raised during the handling of the request, only for the "proper" response. Fix a bug where it *did* sometimes run.
- Previously if you returned `None` for a path in a `variables` function for a path, you would get a path with `None` in it. Now it is a `LinkError`.
- If you return a non-dict for `variables` for a path, you get a proper `LinkError` now.
- One test related to defer_links did not work correctly in Python 3. Fixed.
- Add API doc for `body_obj`. Also fix JSON and objects doc to talk about `request.body_obj` instead of `request.obj`.
- Extend API docs for security: detail the API an identity policy needs to implement and fix a few bugs.
- Fix ReST error in API docs for `autoconfig` and `autosetup`.
- Fix a few ReST links to the API docs in the app reuse document.

0.7 (2014-11-03)

- **Breaking change**. There has been a change in the way the mount directive works. There has also been a change in the way linking between application works. The changes result in a simpler, more powerful API and implementation.

  The relevant changes are:

  – You can now define your own custom `__init__` for `morepath.App` subclasses. Here you can specify the arguments with which your application object should be mounted. The previous `variables` class attribute is now ignored.

  It's not necessary to use `super()` when you subclass from `morepath.App` directly.

  So, instead of this:

  ```
  class MyApp(morepath.App):
      variables = ['mount_id']
  ```

  You should now write this:

```
class MyApp(morepath.App):
    def __init__(self, mount_id):
        self.mount_id = mount_id
```

– The `mount` directive should now return an *instance* of the application being mounted, not a dictionary with mount parameters. The application is specified using the `app` argument to the directive. So instead of this:

```
@RootApp.mount(app=MyApp, path='sub/{id}')
def mount_sub(id):
    return {
       'mount_id': id
    }
```

You should now use this:

```
@RootApp.mount(app=MyApp, path='sub/{id}')
def mount_sub(id):
    return MyApp(mount_id=id)
```

– The `mount` directive now takes a `variables` argument. This works like the `variables` argument to the `path` directive and is used to construct links.

It is given an instance of the app being mounted, and it should reconstruct those variables needed in its path as a dictionary. If omitted, Morepath tries to get them as attributes from the application instance, just like it tries to get attributes of any model instance.

`MyApp` above is a good example of where this is required: it does store the correct information, but as the `mount_id` attribute, not the `id` attribute. You should add a `variables` argument to the `mount` directive to explain to Morepath how to obtain `id`:

```
@RootApp.mount(app=MyApp, path='sub/{id}',
               variables=lambda app: dict(id=app.mount_id))
def mount_sub(id):
    return MyApp(mount_id=id)
```

The simplest way to avoid having to do this is to name the attributes the same way as the variables in the paths, just like you can do for model classes.

– In the past you'd get additional mount context variables as extra variables in the function decorated by the `path` decorator. This does not happen anymore. Instead you can add a special `app` parameter to this function. This gives you access to the current application object, and you can extract its attributes there.

So instead of this:

```
@MyApp.path(path='models/{id}', model=Model)
def get_root(mount_id, id):
    return Model(mount_id, id)
```

where `mount_id` is magically retrieved from the way `MyApp` was mounted, you now write this:

```
@MyApp.path(path='models/{id}', model=Model)
def get_root(app, id):
    return Model(app.mount_id, id)
```

– There was an `request.mounted` attribute. This was a special an instance of a special `Mounted` class. This `Mounted` class is now gone – instead mounted applications are simply instances of their class. To access the currently mounted application, use `request.app`.

- The `Request` object had `child` and `sibling` methods as well as a `parent` attribute to navigate to different "link makers". You'd navigate to the link maker of an application in order to create links to objects in that application. These are now gone. Instead you can do this navigation from the application object directly, and instead of link makers, you get application instances. You can pass an application instance as a special `app` argument to `request.link` and `request.view`.

  So instead of this:

  ```
  request.child(foo).link(obj)
  ```

  You now write this:

  ```
  request.link(obj, app=request.app.child(foo))
  ```

  And instead of this:

  ```
  request.parent.link(obj)
  ```

  You now write this:

  ```
  request.link(obj, app=request.app.parent)
  ```

  Note that the new `defer_links` directive can be used to automate this behavior for particular models.

- The `.child` method on `App` can the app class as well as the parameters for the function decorated by the `mount` directive:

  ```
  app.child(MyApp, id='foo')
  ```

  This can also be done by name. So, assuming `MyApp` was mounted under `my_app`:

  ```
  app.child('my_app', id='foo')
  ```

  This is how `request.child` worked already.

  As an alternative you can now instead pass an app *instance*:

  ```
  app.child(MyApp(mount_id='foo'))
  ```

  Unlike the other ways to get the child, this takes the parameters need to create the app instance, as opposed to taking the parameters under which the app was mounted.

Motivation behind these changes:

Morepath used to have a `Mount` class separate from the `App` classes you define. Since Morepath 0.4 application objects became classes, and it made sense to make their instances the same as the mounted application. This unification has now taken place.

It then also made sense to use its navigation methods (`child` and friend) to navigate the mount tree, instead of using the rather complicated "link maker" infrastructure we had before.

This change simplifies the implementation of mounting considerably, without taking away features and actually making the APIs involved more clear. This simplification in turn made it easier to implement the new `defer_links` directive.

- **Breaking change**. The arguments to the `render` function have changed. This is a function you can pass to a view directive. The render function now takes a second argument, the request. You need to update your render functions to take this into account. This only affects code that supplies an explicit `render` function to the `view`, `json` and `html` directives, and since not a lot of those functions exist, the impact is expected to be minimal.

- **Breaking change**. In certain circumstances it was useful to access the settings through an application instance using `app.settings`. This does not work anymore; access the settings through `app.registry.settings` instead.
- `dump_json` and `load_json` directives. This lets you automatically convert an object going to a response to JSON, and converts JSON coming in as a request body from JSON to an object. See http://morepath.readthedocs.org/en/latest/json.html for more information.
- `defer_links` directive. This directive can be used to declare that a particular mounted application takes care of linking to instances of a class. Besides deferring `request.link()` it will also defer `request.view`. This lets you combine applications with more ease. By returning `None` from it you can also defer links to this app's parent app.
- `app.ancestors()` method and `app.root` attribute. These can be used for convenient access to the ancestor apps of a mounted application. To access from the request, use `request.app.root` and `request.app.ancestors()`.
- The `App` class now has a `request_class` class attribute. This determines the class of the request that is created and can be overridden by subclasses. `more.static` now makes use of this.
- Several generic functions that weren't really pulling their weight are now gone as part of the mount simplification: `generic.context` and `generic.traject` are not needed anymore, along with `generic.link_maker`.
- Change documentation to use uppercase class names for App classes everywhere. This reflects a change in 0.4 and should help clarity.
- Added documentation about auto-reloading Morepath during development.
- No longer silently suppress ImportError during scanning: this can hide genuine `ImportError` in the underlying code.

  We were suppressing `ImportError` before as it can be triggered by packages that rely on optional dependencies.

  This is a common case in the `.tests` subdirectory of a package which may import a test runner like `pytest`. `pytest` is only a test dependency of the package and not a mainline dependencies, and this can break scanning. To avoid this problem, Morepath's autosetup and autoconfig now automatically ignore `.tests` and `.test` sub-packages.

  Enhanced the API docs for `autosetup` and `autoconfig` to describe scenarios which can generate legitimate `ImportError` exceptions and how to handle them.
- Fix of examples in tween documentation.
- Minor improvement in docstrings.

0.6 (2014-09-08)

- Fix documentation on the `with` statement; it was not using the local `view` variable correctly.
- Add #morepath IRC channel to the community docs.
- Named mounts. Instead of referring to the app class when constructing a link to an object in an application mounted elsewhere, you can put in the name of the mount. The name of the mount can be given explicitly in the mount directive but defaults to the mount path.

  This helps when an application is mounted several times and needs to generate different links depending on where it's mounted; by referring to the application by name this is loosely coupled and will work no matter what application is mounted under that name.

This also helps when linking to an application that may or may not be present; instead of doing an import while looking for `ImportError`, you can try to construct the link and you'll get a `LinkError` exception if the application is not there. Though this still assumes you can import the model class of what you're linking to.

(see issue #197)

- Introduce a `sibling` method on Request. This combines the `.parent.child` step in one for convenience when you want to link to a sibling app.

0.5.1 (2014-08-28)

- Drop usage of sphinxcontrib.youtube in favor of raw HTML embedding, as otherwise too many things broke on readthedocs.

0.5 (2014-08-28)

- Add `more.static` documentation on local components.
- Add links to youtube videos on Morepath: the keynote at PyCon DE 2013, and the talk on Morepath at EuroPython 2014.
- Add a whole bunch of extra code quality tools to buildout.
- `verify_identity` would be called even if no identity could be established. Now skip calling `verify_identity` when we already have `NO_IDENTITY`. See issue #175.
- Fix issue #186: mounting an app that is absorbing paths could sometimes generate the wrong link. Thanks to Ying Zhong for the bug report and test case.
- Upgraded to a newer version of Reg (0.8) for `@reg.classgeneric` support as well as performance improvements.
- Add a note in the documentation on how to deal with URL parameters that are not Python names (such as `foo@`, or `blah[]`). You can use a combination of `extra_parameters` and `get_converters` to handle them.
- Document the use of the `with` statement for directive abbreviation (see the Views document).
- Created a mailing list:

  https://groups.google.com/forum/#!forum/morepath

  Please join!

  Add a new page on community to document this.

0.4.1 (2014-07-08)

- Compatibility for Python 3. I introduced a meta class in Morepath 0.4 and Python 3 did not like this. Now the tests pass again in Python 3.
- remove `generic.lookup`, unused since Morepath 0.4.
- Increase test coverage back to 100%.

0.4 (2014-07-07)

- **BREAKING CHANGE** Move to class-based application registries. This breaks old code and it needs to be updated. The update is not difficult and amounts to:
  - subclass `morepath.App` instead of instantiating it to create a new app. Use subclasses for extension too.
  - To get a WSGI object you can plug into a WSGI server, you need to instantiate the app class first.

  Old way:

```
app = morepath.App()
```

  So, the `app` object that you use directives on is an instance. New way:

```
class app(morepath.App):
    pass
```

  So, now it's a class. The directives look the same as before, so this hasn't changed:

```
@app.view(model=Foo)
def foo_default(self, request):
    ...
```

  To extend an application with another one, you used to have to pass the `extends` arguments. Old way:

```
sub_app = morepath.App(extends=[core_app])
```

  This has now turned into subclassing. New way:

```
class sub_app(core_app):
    pass
```

  There was also a `variables` argument to specify an application that can be mounted. Old way:

```
app = morepath.App(variables=['foo'])
```

  This is now a class attribute. New way:

```
class app(morepath.App):
    variables = ['foo']
```

  The `name` argument to help debugging is gone; we can look at the class name now. The `testing_config` argument used internally in the Morepath tests has also become a class attribute.

  In the old system, the application object was both configuration point and WSGI object. Old way:

```
app = morepath.App()

# configuration
@app.path(...)
...

# wsgi
morepath.run(app)
```

  In the Morepath 0.4 this has been split. As we've already seen, the application *class* serves. To get a WSGI object, you need to first *instantiate* it. New way:

```
class app(morepath.App):
  pass

# configuration
@app.path(...)
...

# wsgi
morepath.run(app())
```

To mount an application manually with variables, we used to need the special `mount()` method. Old way:

```
mounted_wiki_app = wiki_app.mount(wiki_id=3)
```

In the new system, mounting is done during instantiation of the app:

```
mounted_wiki_app = wiki_app(wiki_id=3)
```

Class names in Python are usually spelled with an upper case. In the Morepath docs the application object has been spelled with a lower case. We've used lower-case class names for application objects even in the updated docs for example code, but feel free to make them upper-case in your own code if you wish.

Why this change? There are some major benefits to this change:

- both extending and mounting app now use natural Python mechanisms: subclassing and instantation.
- it allows us to expose the facility to create new directives to the API. You can create application-specific directives.

- You can define your own directives on your applications using the `directive` directive:

```
@my_app.directive('my_directive')
```

This exposes details of the configuration system which is underdocumented for now; study the `morepath.directive` module source code for examples.

- Document how to use more.static to include static resources into your application.
- Add a `recursive=False` option to the config.scan method. This allows the non-recursive scanning of a package. Only its `__init__.py` will be scanned.
- To support scanning a single module non-recursively we need a feature that hasn't landed in mainline Venusian yet, so depend on Venusifork for now.
- A small optimization in the publishing machinery. Less work is done to update the generic function lookup context during routing.

0.3 (2014-06-23)

- Ability to absorb paths entirely in path directive, as per issue #132.
- Refactor of config engine to make Venusian and immediate config more clear.
- Typo fix in docs (Remco Wendt).
- Get version number in docs from setuptools.
- Fix changelog so that PyPI page generates HTML correctly.
- Fix PDF generation so that the full content is generated.

- Ability to mark a view as internal. It will be available to `request.view()` but will give 404 on the web. This is useful for structuring JSON views for reusability where you don't want them to actually show up on the web.
- A `request.child(something).view()` that had this view in turn call a `request.view()` from the context of the `something` application would fail – it would not be able to look up the view as lookups still occurred in the context of the mounting application. This is now fixed. (thanks Ying Zhong for reporting it)

  Along with this fix refactored the request object so it keeps a simple `mounted` attribute instead of a stack of `mounts`; the stack-like nature was not in use anymore as mounts themselves have parents anyway. The new code is simpler.

0.2 (2014-04-24)

- Python 3 support, in particular Python 3.4 (Alec Munro - fudomunro on github).
- Link generation now takes `SCRIPT_NAME` into account.
- Morepath 0.1 had a security system, but it was undocumented. Now it's documented (docs now in Morepath Security), and some of its behavior was slightly tweaked:
  - new `verify_identity` directive.
  - `permission` directive was renamed to `permission_rule`.
  - default unauthorized error is 403 Forbidden, not 401 Unauthorized.
  - `morepath.remember` and `morepath.forbet` renamed to `morepath.remember_identity` and `morepath.forget_identity`.
- Installation documentation tweaks. (Auke Willem Oosterhoff)
- `.gitignore` tweaks (Auke Willem Oosterhoff)

0.1 (2014-04-08)

- Initial public release.

CHAPTER 33

Upgrading to a new Morepath version

Morepath keeps a detailed changelog (*CHANGES*) that describes what has changed in each release of Morepath. You can learn about new features in Morepath this way, but also about things in your code that might possibly break. Pay particular attention to entries marked **Breaking change**, **Deprecated** and **Removed**.

Breaking change means that you have to update your code as described if you use this feature of Morepath.

Deprecated means that your code won't break yet but you get a deprecation warning instead. You can then upgrade your code to use the newer APIs. You can show deprecation warnings by passing the following flag to the Python interpreter when you run your code:

```
$ python -W error::DeprecationWarning
```

If you use an entry point to create a command-line tool you will have to supply your Python interpreter manually:

```
$ python -W error::DeprecationWarning the_tool
```

You can also turn these on in your code:

```
import warnings
warnings.simplefilter('always', DeprecationWarning)
```

It's also possible to turn deprecation warnings into an error:

```
import warnings
warnings.simplefilter('error', DeprecationWarning)
```

A **Deprecated** entry in the changelog changes into a **Removed** in a future release; we are not maintaining deprecation warnings forever. If you see a **Removed** entry, it pays off to run your code with deprecation warnings turned on before you upgrade to this version.

Part VII

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

P

R

S

T

U

V

W