Morepath Documentation

Release 0.6

Morepath developers

September 08, 2014

Contents

1 Table of Contents **1**

1.1 Morepath: Super Powered Python Web Framework . . . 1
1.2 Quickstart . . . 2
1.3 Community . . . 8
1.4 Installation . . . 9
1.5 Superpowers . . . 12
1.6 Paths and Linking . . . 14
1.7 Views . . . 24
1.8 Security . . . 31
1.9 REST . . . 35
1.10 Settings . . . 40
1.11 Organizing your Project . . . 41
1.12 App Reuse . . . 46
1.13 Building Large Applications . . . 51
1.14 Tweens . . . 58
1.15 Static resources with Morepath . . . 60
1.16 Morepath API . . . 63
1.17 Comparison with other Web Frameworks . . . 76
1.18 Design Notes . . . 81
1.19 Developing Morepath . . . 83
1.20 CHANGES . . . 84
1.21 History of Morepath . . . 88

2 Indices and tables **91**

Python Module Index **93**

CHAPTER 1

Table of Contents

1.1 Morepath: Super Powered Python Web Framework

Morepath is a Python web microframework, with super powers.

Morepath is an 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, easily. 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. Generic views are just views too. 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!

1.1.1 Video intro

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

1.1.2 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!*

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

1.1.3 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__':
    config = morepath.setup()
    config.scan()
    config.commit()
    morepath.run(app())
```

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

1.1.4 More documentation, please!

- *Read the documentation*

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

1.2 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.

1.2.1 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__':
    config = morepath.setup()
    config.scan()
    config.commit()
    morepath.run(app())
```

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

```
$ python hello.py
Running <morepath.App 'Hello'> with wsgiref.simple_server on http://127.0.0.1:5000
```

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')

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.

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.

1.2.2 Code Walkthrough

1. We import `morepath`.
2. We create a subclass of `morepath.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. func:*morepath.setup* sets up Morepath's default behavior, and returns a Morepath config object. If your app is in a Python package and you've set up the right `install_requires` in `setup.py`, consider using `morepath.autosetup()` to be done in one step.
7. We then `scan()` this module (or package) for configuration decorators (such as `morepath.App.path()` and `morepath.App.view()`) and cause the registration to be registered using `morepath.Config.commit()`.

   This step ensures your configuration (model routes, views, etc) is loaded exactly once in a way that's reusable and extensible.
8. 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, but for a real project we recommend you read *Organizing your Project*.

1.2.3 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)
```

which generates the path `/users/bob` for us.

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

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

And we get `/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.

1.2.4 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()`.

1.2.5 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'))
```

1.2.6 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()
```

1.3 Community

1.3.1 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!

1.3.2 #morepath on Freenode IRC

Want to chat with us? Join us on the #morepath channel on Freenode IRC.

If you don't have an IRC client, an easy way to use Freenode is through its webchat feature.

1.3.3 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!

1.4 Installation

1.4.1 Quick and Dirty Installation

To get started with Morepath right away, first create a Python 2.7 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`.

1.4.2 Creating a Morepath Project

When you develop a web application it's a good idae to use standard Python project organization practices. *Organizing your Project* describes 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 or buildout. We'll briefly describe how to use both.

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
```

buildout

Buildout is more involved than pip, but can also do a lot more for you automatically and repeatedly.

Create a `buildout.cfg` file containing this:

```
[buildout]
develop = .
parts = scripts devpython
versions = versions

[versions]
venusian = 1.0a8
morepath = 0.1
reg = 0.6

[scripts]
recipe = zc.recipe.egg:scripts
```

```
eggs = myproject
       pytest

[devpython]
recipe = zc.recipe.egg
interpreter = devpython
eggs = myproject
       flake8
```

This describes how to install our project for development. Change `myproject` to the name your project has in `setup.py`.

Place a buildout bootstrap.py in your project's root directory.

The first time you create or check out a project you need to bootstrap the buildout. You can do this using the `bootstrap.py` script. Do this from a virtualenv:

```
$ /path/to/morepath_env/bin/python bootstrap.py
```

You only need to do this once. After that you can run:

```
$ bin/buildout
```

each time you want to redo the installation after you change the buildout config. It's safe to run this when nothing has really changed too.

Once you've run buildout, you can start your application. If it's named `myproject-start` in the entry point in `setup.py`, you can run it like this:

```
$ bin/myproject-start
```

`bin` directory

Everything in `bin` will run in the virtualenv you've used to bootstrap your project automatically (or in a subset thereof).

What's going on with buildout?

What's going on? What else did that `buildout.cfg` do for us?

The `develop` line tells which directories to look in for Python projects (with a `setup.py`). In this case only the local project directory `.` is one. But if you also have the checkout of another project that you depend on (maybe a development version of Morepath itself), you can add that directory to the `develop` section.

mr.developer

If you are going to develop such a multi-project codebase you should consider the buildout extension mr.developer which can help you automate this.

`parts` tells buildout what to configure; they are described in the `[scripts]` and `[devpython]` sections later.

The line `versions=versions` tells buildout to lock down version numbers according to the `[versions]` section.

show-picked-versions

You can add a line `show-picked-versions = true` to the `[buildout]` section. When you now run `bin/buildout` this dumps all versions of libraries you use directly or indirectly that you haven't locked down to an explicit version to the console. You can then lock them down in the `[versions]` section.
Locking down versions is useful if you want to make sure everybody has the same versions of the libraries in development.

The `[scripts]` section installs your web application as a script in the `bin` subdirectory of your project, according to the `console_scripts` entry point in your project's `setup.py`. If it's called `myproject-start`, then you can start it like this:

```
bin/myproject-start
```

This will start a HTTP server for your project.

The buildout also has installed pytest so you can run your project's tests automatically:

```
bin/py.test myproject
```

(if your Python package is in `myproject`)

Test dependencies

If you want to add some extra dependencies just for testing, you can do this in your project's `setup.py` by adding:

```
extras_require = dict(
  test=['pytest >= 2.5',
        'pytest-cov'],
),
```

This makes sure we have a `pytest` version 2.5 or later, and we install the `pytest-cov` code coverage extension.
You can then modify the `[scripts]` section in `buildout.cfg` to use the extra `test` requirements:

```
[scripts]
recipe = zc.recipe.egg:scripts
eggs = myproject [test]
       pytest
```

Now as to some optional extras. The `[devpython]` section installs a Python interpreter which can import exactly what your project can import. It assumes your project is called `myproject` in its `setup.py`; change the name to match your project. You can start it using:

```
$ bin/devpython
```

You'll get the usual Python console `>>>`. This is useful for testing your project's imports and API manually.

It also installs the flake8 tool which runs pep 8 checks and pyflakes automatically. You can run it against your project by writing:

```
$ bin/flake8 myproject
```

where `myproject` is your project's source code directory.

1.4.3 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, we recommend you use buildout with the mr.developer extension for your project. You can see how in this buildout.cfg.

You can also 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
```

1.5 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!

1.5.1 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_model)
```

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

```
request.link(my_model, '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*.

1.5.2 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*.

1.5.3 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()` directive. We can implement any rule we want, for instance this one:

```
@app.permission(model=Document, permission=Edit)
def have_edit_permission(model, identity):
    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`).

1.5.4 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(request, doc) 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!

1.5.5 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 my_app(app):
    pass
```

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

```
@my_app.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. Using the `morepath.App.function()` decorator you could even override the core functionality of Morepath itself!

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*.

1.6 Paths and Linking

1.6.1 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, and how to find these variables again in the model in order to construct a link to it.

1.6.2 Paths

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.

1.6.3 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.

1.6.4 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`.

1.6.5 Linking

To create a link to a model, we can call `morepath.Request.link()` in our view code. At that point the model 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 `documents/foo/link2` produces the link `documents/foo/link`.

1.6.6 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 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 model: dict(name=model.name))
def get_document(name):
    return query_document_by_name(name)
```

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

```
def document_variables(model):
    return dict(name=model.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 model: dict(name=model.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 model: dict(name=model.name,
                                       version=model.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 model: dict(text=model.search_text,
                                       extra_parameters=model.get_extra()))
def get_search_results(text, extra_parameters):
    ...
```

1.6.7 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 model: dict(name=model.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`.

1.6.8 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.

1.6.9 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.

1.6.10 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
```

1.6.11 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.

1.6.12 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.

1.6.13 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 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`.

1.6.14 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.

1.6.15 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.

1.7 Views

1.7.1 Introduction

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

1.7.2 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
```

1.7.3 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
```

1.7.4 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*.

1.7.5 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.

1.7.6 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')
```

1.7.7 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(class=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):
    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):
    response = morepath.Response(json.dumps(content))
    response.content_type = 'application/json'
    return response
```

We'd use it like this:

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

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

1.7.8 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()` decorator. To learn more, read *Security*.

1.7.9 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"
```

1.7.10 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
```

1.7.11 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.

1.7.12 Predicates

The `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:

```
@app.predicate(name='something', order=100, default=None)
def get_something_header(self, 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. If you don't care what the predicate is and want the view to match for any value, you can pass in the special sentinel `morepath.ANY`. 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.

The `order` parameter for the predicate determines which predicates match more strongly than another; lower order matches more strongly. If there are two view candidates that both match the predicates for a request and model, the strongest match is picked.

1.7.13 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.

1.7.14 Exception views

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.

1.8 Security

1.8.1 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.

1.8.2 Identity

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. 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.

This is how you install an identity policy into a Morepath app:

```
from morepath.security import BasicAuthIdentityPolicy

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

1.8.3 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 and cookie-based authentication, 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.

1.8.4 Session or ticket identity verification

If you use an identity policy based on the session (which you've made secure otherwise), or on a cryptographic ticket based authentication system such as the one implemented by mod_auth_tkt, 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
```

a ticket based identity policy implementation?

There is no implementation yet of a ticket based identity policy in Morepath. Will you implement one? You could port it from Pyramid.

1.8.5 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 in. When the user logs in we need to *remember* their identity on the response, and when the user logs in 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)
    morepath.remember_identity(response, identity)
```

This is enough for session-based or cryptographic ticket-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 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)
    morepath.remember_identity(response, identity)
```

When you construct the identity using `morepath.Identity`, you can 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 (see later). You simply call `morepath.forget_identity` somewhere in the logout view:

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

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

Basic authentication

Basic authentication is special in a number of ways:

- The HTTP response status that triggers basic auth is Unauthorized (401), not the default Forbidden (403). This needs to be sent back to the browser each time login fails, so that the browser asks the user for a username and a password.
- The username and password combination is sent to the server by the browser automatically; there is no need to set some type of cookie on the response. Therefore `remember_identity` does nothing.
- With basic auth, there is no universal way for a web application to trigger a log out. Therefore `forget_identity` does nothing either.

To trigger a `401` status when time Morepath raises a `403` status, we can use an exception view, something like this:

```
from webob.exc import HTTPForbidden

@app.view(model=HTTPForbidden)
def make_unauthorized(self, request):
    @request.after
    def set_status_code(response):
        response.status_code = 401
    return "Unauthorized"
```

The core of the login code can remain the same as `remember_identity` is a no-op, but you could reduce it to this:

```
# 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
```

1.8.6 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.

1.8.7 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.

1.8.8 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 ...
```

1.9 REST

1.9.1 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.

1.9.2 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.

1.9.3 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, id, title, author, content):
        self.id = id
        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):
   return document_by_id(id)
```

We assume here that a `document_by_id()` function exists that returns a `Document` instance by id from some database, or `None` if the document cannot be found. Any way to get your model instance is fine.

Now we want a `metadata` resource that exposes its metadata as JSON:

```
@app.json(model=Document, name='metadata')
def document_metadata(self, request):
    return {
      'id': self.id,
      'title': self.title,
      'author': self.author
    }
```

1.9.4 Modeling as resources

Modeling a web service as multiple resources comes pretty naturally to Morepath, as it's model-oriented in the first place. You can think carefully about how to place models in the URL space and expose them using `morepath.App.path()`. In Morepath 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 = []

    def add(self, doc):
        self.documents.append(doc)
```

We now want to expose this collection to a URL path `/documents`. We want:

- a resource `/documents` to GET the ids of all documents in the collection.
- a resource `/documents/add` that lets you POST an `id` to it so that this document is added to the collection.

Here is how we could make `documents` available on a URL:

```
documents = DocumentCollection()

@app.path(model=DocumentCollection, path='documents')
def documents_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:

```
@app.json(model=DocumentCollection)
def collection_default(self, request):
    return {
       'type': 'document_collection',
       'ids': [doc.id for doc in self.documents]
    }
```

Then we want to allow people to POST the document id (as a URL parameter) to the `/documents/add` resource:

```
@app.json(model=DocumentCollection, name='add', request_method='POST')
def collection_add_document(self, request):
    doc = document_by_id(request.args['id'])
    self.add(doc)
    return {}
```

We again use the `document_by_id` function. We also return an empty JSON object in the response; not very useful, but in this simple view we don't have anything more interesting to report when the POST succeeds.

Note the use of `request_method`, which we'll talk about more next.

Note also that there are some things still missing: giving back a proper response with status codes, and error handling when things go wrong.

1.9.5 HTTP methods

As you saw above, we've used `request_method` to make sure that `/documents/add` only works for `POST` requests.

By default, `request_method` is `GET`, meaning that `/documents` only responds to a `GET` request, which is what we want. Let's make it explicit:

```
@app.json(model=DocumentCollection, request_method='GET')
def collection_default(self, request):
    ...
```

What if we had defined our web service differently, and instead of having a `/documents/add` we wanted to allow the POSTing of document ids on `/documents` directly? Here's how you rewrite `collection_add_document` to be the view directly on `/documents`':

```
@app.json(model=DocumentCollection, request_method='POST')
def collection_add_document(self, request):
    ...
```

It's just a matter of removing the `name` parameter so that it becomes the default view on `DocumentCollection`.

1.9.6 HTTP response status codes

When a view did its thing with success, Morepath automatically returns the HTTP status code `200`. When you try to access a URL that cannot be routed to a model or a view, a `404` error is raised.

But what if the view did not manage to do something successfully? Let's get back to this view:

```
@app.json(model=DocumentCollection, name='add', request_method='POST')
def collection_add_document(self, request):
    doc = document_by_id(request.args['id'])
    self.add(doc)
    return {}
```

What if there is no `id` parameter in the request? That's something our application cannot handle: a bad request, status code 400.

What status code is right?

There is some debate over what status code to pick for particular errors. Sometimes the HTTP specification is pretty clear, but in the case of a missing parameter, it's not. Status code 400 (Bad Request) while according to the HTTP specd more about the syntax of a request than its content, is still chosen by many implementers in case of errors like this.
But no matter what kind of HTTP error you pick, how you cause them to happen is the same: just raise the appropriate exception.

WebOb, the request/response library upon which Morepath is built, defines a set of HTTP exception classes `webob.exc` that we can use. In this case we need `webob.exc.HTTPBadRequest`. We modify our view so it is raised if there was no id:

```
from webob.exc import HTTPBadRequest

@app.json(model=DocumentCollection, name='add', request_method='POST')
def collection_add_document(self, request):
    id = request.args.get('id')
    if id is None:
        raise HTTPBadRequest()
    doc = document_by_id(id)
    self.add(doc)
    return {}
```

We also want to deal with the situation where an id was given, but no document with that id exists. Let's handle that with 400 Bad Request too:

```
@app.json(model=DocumentCollection, name='add', request_method='POST')
def collection_add_document(self, request):
    id = request.args.get('id')
    if id is None:
        raise BadRequest()
    doc = document_by_id(id)
    if doc is None:
        raise BadRequest()
    self.add(doc)
    return {}
```

1.9.7 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 very easy to create hyperlinks, so we won't have to do much. Let's first modify our default `GET` view for the collection so it also has a link to the `add` resource:

```
@app.json(model=DocumentCollection)
def collection_default(self, request):
    return {
       'type': 'document_collection',
       'ids': [doc.id for doc in self.documents],
       'add': request.link(documents, 'add')
    }
```

`documents`, if you can remember, is the instance of `DocumentCollection` we were working with, and we want to link to its `add` view.

Let's make things more interesting though. Before we had the default view for the collection return a list of document ids. We can change this so we return a list of document URLs instead:

```
@app.json(model=DocumentCollection)
def collection_default(self, request):
    return {
```

```
        'type': 'document_collection',
        'documents': [request.link(doc) for doc in self.documents],
        'add': request.link(documents, 'add')
    }
```

Or perhaps better, include the id *and* the URL:

```
@app.json(model=DocumentCollection)
def collection_default(self, request):
    return {
        'type': 'document_collection',
        'documents': [dict(id=doc.id, link=request.link(doc))
                      for doc in self.documents],
        'add': request.link(documents, 'add')
    }
```

Now we've got HATEOAS: the collection links to the documents it contains, and also to the `add` URL that can be used to add a new document. The developers looking at the responses your web service sends get a few clues about where to go next. Coupling is looser.

We got 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!

1.9.8 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.

1.10 Settings

1.10.1 Introduction

A typical application has some settings: if an application logs, a setting is the path to the log file. If an aplication 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.

1.10.2 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.

1.10.3 Accessing a setting

During runtime, you can access the settings of the current application using the `morepath.settings()` function, like this:

```
settings().logging.logfile
```

In a tween factory (see :doc:tweens) or a directive implementation, you can access a setting through the `app` object like this:

```
app.settings.logging.logfile
```

1.10.4 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.

1.11 Organizing your Project

1.11.1 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.

1.11.2 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.

1.11.3 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
        main.py
        model.py
        [collection.py]
        path.py
        view.py
```

1.11.4 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.main:main'
          ]
      })
```

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.autosetup()`.

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 `main()` function in the `myproject.main` module. Let's create this next.

See also the setuptools documentation.

1.11.5 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.

1.11.6 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
            main.py
            model.py
            [collection.py]
            path.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.main:main'
          ]
      })
```

See also the namespace packages documentation.

1.11.7 Main Module

The `main.py` module is where we define our Morepath app and allow a way to start it up as a web server. Here's a sketch of `main.py`:

```
import morepath

class app(morepath.App):
    pass

def main():
   morepath.autosetup()
   morepath.run(app())
```

We create an `app` class, then have a `main()` function that is going to be called by the `myproject-start` entry point we defined in `setup.py`. This main function does two things:

- Use `morepath.autosetup()` to set up Morepath, including any of your code.
- 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 main module is also a good place to do other general configuration for the application, such as setting up a database connection.

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 `main.py` to use waitress:

```
import waitress

...

def main():
   ...
   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.autosetup()
   return app()
```

```
$ waitress-serve --call myproject.main: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
```

1.11.8 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.

1.11.9 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 myproject.main import app
from myproject 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.

1.11.10 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 myproject.main import app
from myproject 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.

1.12 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 tries to make these things simple.

1.12.1 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. This configuration is specify used 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, the default (unnamed) view is found. We've provided that too: it just renders "User: {username}".

What now if we have another app where we want to publish `User` in a different way? No problem, we can just create one:

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

@other_app.path(model=User, path='different_path/{username}')
def get_user(username):
    return different_query_for_user(username)

@other_app.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 `other_app` (even in the same runtime), it functions independently from `app`.

This app isolation is nothing really special; it's kind of obvious that this is possible. But that's what we wanted. Let's look at a few more involved possibilities next.

1.12.2 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? Well, it can, especially in more complex applications and reuse scenarios. Often 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.

This architectural principle is called the Open/Closed Principle in software engineering, and Morepath makes it really easy to follow it. What you do is create another app that subclasses the original:

```
class extended_app(app):
    pass
```

And then we can add the view to the extended app:

```
@extended_app.view(model=User, name='edit')
def edit_user(self, request):
    return 'Edit user: %s' % self.username
```

Now when we publish `extended_app` using WSGI, the new `edit` view is there, but when we publish `app` it won't be.

Just subclassing. Kind of obvious, perhaps. Good. Let's move on.

1.12.3 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:

```
@extended_app.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.

You can also do this for what is returned for model paths. We might for instance want to return a different user object altogether in our overriding app:

```
@extended_app.path(model=OtherUser, path='users/{username}')
def get_user_differently(username):
    return OtherUser(username)
```

To make `OtherUser` actually be published on the web under `/users/{username}` it either needs to be a subclass of `User`, for which we've already registered a default view, or we need to 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 like subclassing. Hopefully not so obvious that it's boring. Let's talk about something new.

1.12.4 Nesting Applications

Let's talk about application composition: nesting one app in another.

Imagine our user app allows users to have wikis associated with them. It has paths like `/users/faassen/wiki` and `/users/bob/wiki`.

One approach might be to implement a wiki application within the user application we already have, along these lines:

```
@app.path(model=Wiki, path='users/{username}/wiki')
def get_wiki(username):
    return wiki_for_user(username)

@app.view(model=Wiki)
def wiki_default_view(request, model):
    return "Default view for wiki"
```

(this is massively simplified of course. we'd also have a `Page` model that's exposed on a sub-path under the wiki, with its own views, etc)

But this feels bad. Why?

- 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 byitself and also test by itself.
- There's the issue of the username: it appears in all paths that go to wiki-related models (the wiki itself, any wiki pages). But why should we have to care about the username of a user when we are thinking about wikis?
- It would also be nice if we can use the wiki app in other contexts as well, instead of only letting it be associated with users. What about associating a wiki app with a project instead, like you can do in github?

A separate app for wikis seems obvious. So let's do it. Here's the wiki app by itself:

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

@wiki_app.path(model=Wiki, path='{wiki_id}')
def get_wiki(wiki_id):
    return query_wiki(wiki_id)

@wiki_app.view(model=Wiki)
def wiki_default_view(self, request):
    return "Default view for wiki"
```

This is an app that exposes wikis on URLs using `wiki_id`, like `/my_wiki`, `/another_wiki`.

But that won't work if we want to associate wikis with users. What if we want the paths we had before, like `/users/faassen/wiki`?

Morepath has a solution. We can *mount* the wiki app in the user app, like this:

```
@app.mount(app=wiki_app, path='users/{username}/wiki')
def mount_wiki(username):
    return {
       'wiki_id': get_wiki_id_for_username(username)
    }
```

We do need to adjust the wiki app a bit as right now it expects `wiki_id` to be in its paths, and the wiki id won't show up when mounted. We need to do two things: tell the wiki app that we expect the `wiki_id` variable:

```
class wiki_app(morepath.App):
    variables = ['wiki_id']
```

And we need to register the model so that its path is empty:

```
@wiki_app.path(model=Wiki, path='')
def get_wiki(wiki_id):
    return query_wiki(wiki_id)
```

But where does `wiki_id` come from now if not from the path? We already have it: it was determined when the app was mounted, and comes from the dictionary that we return from `mount_wiki()`.

What if we want to use `wiki_app` by itself, as a WSGI app? That can be useful, also for testing purposes. It needs this `wiki_id` parameter now. We simply pass it the `wiki_id` parameter when we instantiate it:

```
wsgi_app = wiki_app(wiki_id=5)
```

This is a WSGI app that we can run by itself that uses `wiki_id`.

1.12.5 Linking to other mounted apps

Using mount names

Instead of using the application class as the first argument to `morepath.Request.child()` and `morepath.Request.sibling()`, you can instead use the name under which it was mounted. The `name` can be explicitly passed in the `mount` directive. If the mount name is ommitted it defaults to what was given as the `path`.

When we have one app mounted inside another, we want a way to make links between them.

You can use `morepath.Request.parent` to link to an object in an app's parent app:

```
request.parent.link(obj)
```

If there is no parent application, this raises a `morepath.error.LinkError`.

Besides using `.link` you can also use `.view` this way.

You can use `morepath.Request.child()` to link to an object in a mounted child application:

```
request.child(child_app).link(obj)
```

If the `child_app` is not mounted here, this will also raise a `morepath.error.LinkError`.

This won't work though in the case of `wiki_app` of the previous example, as it mounted inside `app` using the `username`. Here's how we supply it to get the appropriate `wiki_app`:

```
request.child(wiki_app, username='foo').link(obj)
```

You can compose `parent` and `child` together in order to get to anywhere in the mounted app graph; getting to a sibling app for instance looks like this:

```
app.parent.child(sibling_app)
```

There is a convenience shortcut for this, `morepath.Request.sibling()`:

```
app.sibling(sibling_app)
```

1.12.6 Application Reuse

Many web frameworks have mechanisms for overriding specific behavior and to support reusable applications. These tend to have been developed in an ad-hoc fashion as new needs arose.

Morepath instead has a *general* mechanism for supporting app extension and reuse. You use the same principles and APIs you already use to create new applications. Any normal Morepath app can without extra effort be reused. Anything registered in a Morepath app can be overridden. This is because Morepath builds on a powerful general configuration system.

1.12.7 Further reading

To see an extended example of how you can structure larger applications to support reuse, see *Building Large Applications*.

1.13 Building Large Applications

1.13.1 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 doc:*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.

1.13.2 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`

1.13.3 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 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):
    ...
```

1.13.4 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 Morepath is there to help.

We'll now show what changes you would make.

1.13.5 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 core_app(morepath.App):
    pass

class issues_app(morepath.App):
    variables = ['issues_id']

class wiki_app(morepath.App):
    variables = ['wiki_id']
```

Note that `issues_app` and `wiki_app` expect variables; 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:

```
@core_app.path(path='', model=Root)
def get_root():
    ...

@core_app.path(path='{user_name}', model=User)
def get_user(user_name):
    ...

@core_app.path(path='{user_name}/{repository_name}', model=Repository)
def get_repository(user_name, repository_name):
    ...

@core_app.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:

```
@issues_app.path(path='', model=Issues)
def get_issues(issues_id):
    ...

@issues_app.path(path='{issue_id}', model=Issue)
def get_issue(issues_id, issue_id):
    ...
```

And the paths for our wiki:

```
@wiki_app.path(path='', model=Wiki)
def get_wiki(wiki_id):
    ...
```

We have drastically simplified the paths in `issues_app` and `wiki_app`; we don't deal with `user_name` and `repository_name` anymore. Instead we get a `issues_id` and `wiki_id`, but not from the path. Where does they come from? They are specified by the `variables` argument for `morepath.App` that we saw earlier. Next we need to explore the `App.mount()` directive to see how they are actually obtained.

1.13.6 Mounting apps

Now that we have an independent `issues_app` and `wiki_app`, we want to be able to mount these under the right URLs under `core_app`. We do this using the mount directive:

```
@core_app.mount(path='{user_name}/{repository_name}/issues',
                app=issues_app)
def mount_issues(user_name, repository_name):
    return { 'issues_id': get_issues_id(user_name, repository_name) }
```

Let's look at what this does:

- `@core_app.mount`: We mount something onto `core_app`.
- `app=issues_app`: We are mounting `issues_app`.
- `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.
- The `mount_issues` function takes the path variables `user_name` and `repository_name` as arguments. It then returns a dictionary with the mount variables expected by `issues_app`, in this case `issues_id`. It does this by using `get_issues_id`, which does some kind of database access in order to determine `issues_id` for `user_name` and `repository_name`.

Mounting the wiki is very similar:

```
@core_app.mount(path='{user_name}/{repository_name}/wiki',
                app=wiki_app)
def mount_wiki(user_name, repository_name):
    return { 'wiki_id': get_wiki_id(user_name, repository_name) }
```

1.13.7 No more path repetition

We have solved the repetition of paths issue now; the issue tracker and wiki can consist of many paths, but there is no more need to repeat '{user_name}/{repository_name}' everywhere.

1.13.8 Testing in isolation

To test the issue tracker by itself, we can run it as a separate WSGI app. To do this we first need to mount it by passing an `issues_id` to it:

```
def run_issue_tracker():
    mounted = issues_app(issues_id=4)
    morepath.run(mounted)
```

Here we mount and run the `issues_app` with issue tracker id `4`. We 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*.

1.13.9 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`. But what if we want to mount the issue tracker app in a separate project altogether? To use it we would need to import it from our project that also contains the core app and the wiki app, meaning that the new project would need to depend on all of this code. That can hinder reuse.

To make it more reusable across projects we can instead maintain the code for the issue tracker app in a separate project, and the same for the wiki app. The core app can then depend on the issue tracker and wiki projects. Another app that also wants to have an issue tracker can depend on the issue tracker project too.

To do this we'd 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` would have an `install_requires` in its `setup.py` that depends on `myproject.issues` and `myproject.wiki`. To get `issues_app` and `wiki_app` in order to mount them in the core, we would simply import them (for instance in `myproject.core.main`):

```
from myproject.issues.main import issues_app
from myproject.wiki.main import wiki_app
```

1.13.10 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, letting you free up your brain.

1.13.11 Swapping in a new sub-app

Perhaps a different, better wiki implementation is developed. Let's call it `shiny_new_wiki_app`. Swapping in the new sub application is easy: it's just a matter of changing the mount directive:

```
@core_app.mount(path='{user_name}/{repository_name}/wiki',
                app=shiny_new_wiki_app)
def mount_wiki(user_name, repository_name):
    return { 'wiki_id': get_wiki_id(user_name, repository_name) }
```

1.13.12 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, but then ... wait for it ... they actually 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 imagine is called `customerdatabase`).

Now we can import `core_app` from it in `important_customer.core`'s `main.py` module, and extend from it:

```
from myproject.core.main import core_app

class customer_app(core_app):
    pass
```

At this point `customer_app` behaves identically to `core_app`. Now let's 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

@customer_app.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 `customer_app` 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 now made exactly the tweak necessary without having to modify our original project. The original project continues to work the same way it always did.

1.13.13 Swapping in, for one customer

Morepath lets you add any directive, not just views. It also lets you *override* things in the applications you extend. What if we had a new wiki like before, but we only want to upgrade one particular to it, and leave the others with the

original? Perhaps our important customer needs *exactly* the wiki app mounted in core app, really, it's perfect... but they actually need a minor tweak to the wiki too.

We'd tweak the wiki just as we would tweak the core app. We end up with a `tweaked_wiki_app`:

```
from myproject.wiki.main import wiki_app

class tweaked_wiki_app(wiki_app):
     pass

# some kind of tweak
@tweaked_wiki_app.json(model=WikiPage, name='extra_info')
def page_extra_info(self, request):
    ...
```

We now want a new version of `core_app` just for this customer that mounts `tweaked_wiki_app` instead of `wiki_app`:

```
class important_customer_app(core_app):
    pass

@important_customer_app.mount(path='{user_name}/{repository_name}/wiki',
                               app=tweaked_wiki_app)
def mount_wiki(user_name, repository_name):
    return { 'wiki_id': get_wiki_id(user_name, repository_name) }
```

The `mount` directive above overrides the one in the `core_app` that we're extending, because it uses the same `path` but mounts `tweaked_wiki_app` instead.

You can override any other directive (path, view, etc) the same way.

1.13.14 Framework apps

A `morepath.App` subclass does not need to be a full working web application. Instead it can be a framework consisting of just a few with only those paths, subpaths and views that we intend to be reusable.

For views this works together well with Morepath's understanding of inheritance. We could for instance have a base class `Metadata`. Whenever any model subclasses from it, we want that model to gain a `metadata` view that returns this metadata as JSON data. Let's write some code for that:

```
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 (say, how links are generated) 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.

1.14 Tweens

1.14.1 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.

1.14.2 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*.

1.14.3 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
```

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. Request come in in the outermost tween and descend down the chain into the underlying tweens, and finally into the Morepath *publish* handler itself.

1.14.4 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.

1.14.5 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
```

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)
```

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)
```

Then the tween chain is:

```
my_tween -> another_tween -> publish
```

1.14.6 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.settings`. See also the *Settings* section.

1.14.7 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.

1.14.8 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.

1.15 Static resources with Morepath

1.15.1 Introduction

A modern client-side web application is built around JavaScript and CSS. The code is in files that is served by the web server too.

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 and Webassets.

We will focus on BowerStatic integration here. We recommend you read its 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.

1.15.2 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.

We also set up a `main()` function that when run serves the WSGI application to the web:

```
def main():
   morepath.autosetup()
   wsgi = app()
   morepath.run(wsgi)
```

All this code lives in the `main.py` module of a Python package.

1.15.3 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 `main.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.

1.15.4 Registering `bower_components`

BowerStatic needs a single global `bower` object that you can register multiple `bower_copmonents` 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 (`main.py`) in this Python package.

Finally we need to adjust our `main()` function so that we plug in the BowerStatic WSGI middleware:

```
def main():
    morepath.autosetup()
    wsgi = bower.wrap(app())
    morepath.run(wsgi)
```

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.

1.15.5 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`.

1.15.6 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.

1.15.7 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.

1.16 Morepath API

class `morepath.`**App**(***context*)

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.

`@`**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()`.

`@`**function**(*target*, **sources*)

Register function as implementation of generic function

The decorated function is an implementation of the generic function supplied to the decorator. This way you can override parts of the Morepath framework, or create new hookable functions of your own. This is a layer over `reg.IRegistry.register()`.

The `target` argument is a generic function, so a Python function marked with either `reg.generic()` or with `reg.classgeneric()`.

Parameters

- **target** (*function object*) – the generic function to register an implementation for.
- **sources** – classes of parameters to register for.

@html (*model*, *render=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.
- **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.

@identity_policy
Register identity policy.

The decorated function should return an instance of an identity policy, which should have `identify`, `remember` and `forget` methods.

@json (*model*, *render=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.
- **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.

@mount(*path*, *app*, *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 dictionary with the required variables for the mounted app. The variables are declared in the `morepath.App` constructor.

Parameters

- **path** – the path to mount the application on.
- **app** – the `morepath.App` subclass to mount.
- **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.

@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 that given a model object can construct the variables used in the path (including any URL parameters). If 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 `remaining` variable.

@permission_rule(*model*, *permission*, *identity=<class 'morepath.security.Identity'>*)

Declare whether a model has a permission.

The decorated function receives `model`, *permission'* (instance of any permission object) and `identity` (`morepath.security.Identity`) parameters. 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.security.NO_IDENTITY`.

@predicate(*name*, *order*, *default*, *index=<class 'reg.predicate.KeyIndex'>*)

Register custom view predicate.

The decorated function gets `model` and `request` (a `morepath.Request` object) parameters.

From this information it should calculate a predicate value and return it. You can then pass these extra predicate arguments to `morepath.App.view()` and this view is only found if the predicate matches.

Parameters

- **name** – the name of the view predicate.
- **order** (*int*) – when this custom view predicate should be checked compared to the others. A lower order means a higher importance.
- **default** – the default value for this view predicate. This is used when the predicate is omitted or `None` when supplied to the `morepath.App.view()` directive. This is also used when using `Request.view()` to render a view.
- **index** – the predicate index to use. Default is `reg.KeyIndex` which matches by name.

@predicate_fallback(*name*)

For a given predicate name, register fallback view.

The decorated function gets `self` and `request` parameters.

The fallback view is a view that gets called when the named predicate does not match and no view has been registered that can handle that case.

Parameters **name** – the name of the predicate.

@**setting**(*section*, *name*)
Register application setting.

An application setting is 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 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.

@**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.

@**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.

@**verify_identity**(*identity=<type 'object'>*)
Verify claimed identity.

The decorated function gives a single `identity` argument which contains the claimed identity. 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.

@view(*model*, *render=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. `morepath.render_json()` for instance expects 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.
- **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** – predicates to match this view on. Use `morepath.ANY` for a predicate if you don't care what the value is. If you don't specify a predicate, the default value is used. Standard predicate values are `name` and `request_method`, but you can install your own using the `morepath.App.predicate()` directive.

classmethod directive(*name*)
Decorator to register a new directive with this application class.

You use this as a class decorator for a `morepath.Directive` subclass:

```
@app.directive('my_directive')
class FooDirective(morepath.Directive):
    ...
```

This needs to be executed *before* the directive is being used and thus might introduce import dependency issues unlike normal Morepath configuration, so beware! An easy way to make sure that all directives are installed before you use them is to make sure you define them in the same module as where you define the application class that has them.

request(*environ*)
Create a `Request` given WSGI environment.

Parameters **environ** – WSGI environment

Returns `morepath.Request` instance

lookup
Get the `reg.Lookup` for this application.

Returns a `reg.Lookup` instance.

`morepath.`**`autoconfig`**(*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 loads all needed Morepath configuration from all packages automatically. These packages do need to be made available using a `setup.py` file including currect `install_requires` information so that they can be found using setuptools.

Creates a `Config` object as with `setup()`, but before returning it scans all packages, looking for those that depend on Morepath directly or indirectly. This includes the package that calls this function. Those packages are then scanned for configuration as with `Config.scan()`.

You can add manual `Config.scan()` calls yourself on the returned `Config` object. Finally you need to call `Config.commit()` on the returned `Config` object so the configuration is committed.

Typically called immediately after startup just before the application starts serving using WSGI.

See also `autosetup()`.

Parameters **ignore** – Venusian style ignore to ignore some modules during scanning. Optional.

Returns `Config` object.

`morepath.`**`autosetup`**(*ignore=None*)

Automatically commit Morepath configuration from packages.

As with `autoconfig()`, but also commits configuration. This can be your one-stop function to load all Morepath configuration automatically.

Typically called immediately after startup just before the application starts serving using WSGI.

Parameters **ignore** – Venusian style ignore to ignore some modules during scanning. Optional.

`morepath.`**`setup`**()

Set up core Morepath framework configuration.

Returns a `Config` object; you can then `Config.scan()` the configuration of other packages you want to load and then `Config.commit()` it.

See also `autoconfig()` and `autosetup()`.

Returns `Config` object.

`morepath.`**`run`**(*wsgi*, *host=None*, *port=None*)

Uses wsgiref.simple_server to run application for debugging purposes.

Don't use this in production; use an external WSGI server instead, for instance Apache mod_wsgi, Nginx wsgi, Waitress, Gunicorn.

Parameters

- **wsgi** – WSGI app.
- **host** – hostname.
- **port** – port.

`morepath.`**`settings`**(**args*, ***kw*)

Return current settings object.

In it are sections, and inside of the sections are the setting values. If there is a `logging` section and a `loglevel` setting in it, this is how you would access it:

```
settings().logging.loglevel
```

class morepath.**Request**(*environ*)

Request.

Extends webob.request.BaseRequest

after(*func*)

Call function with response after this request is done.

Can be used explicitly:

```
def myfunc(response):
    response.headers.add('blah', 'something')
request.after(my_func)
```

or as a decorator:

```
@request.after
def myfunc(response):
    response.headers.add('blah', 'something')
```

Parameters **func** – callable that is called with response

Returns func argument, not wrapped

child(*app*, ***variables*)

Obj to call Request.link() or Request.view() on child.

Get an object that represents the application mounted in this app. You can call link and view on it.

Parameters

- **app** – either subclass of morepath.App that you want to link to, or a string. This string represents the name of the mount (by default it's the path under which the mount happened).
- ****variables** – Keyword parameters. These are the mount variables under which the app was mounted.

link(*obj*, *name=''*, *default=None*)

Create a link (URL) to a view on a model instance.

If no link can be constructed for the model instance, a :exc:morepath.LinkError is raised. None is treated specially: if None is passed in the default value is returned.

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.

sibling(*app*, ***variables*)

Obj to call Request.link() or Request.view() on sibling.

Get an object that represents the application mounted as a sibling to this app, so the child of the parent. You can call link and view on it.

Parameters

- **app** – either subclass of `morepath.App` that you want to link to, or a string. This string represents the name of the mount (by default it's the path under which the mount happened).
- ****variables** – Keyword parameters. These are the mount variables under which the app was mounted.

view(*obj*, *default=None*, ***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.
- **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.

identity
Self-proclaimed identity of the user.

The identity is established using the identity policy. Normally this would be an instance of `morepath.security.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.security.NO_IDENTITY`.

The identity can be used for authentication/authorization of the user, using Morepath permission directives.

parent
Obj to call `Request.link()` or `Request.view()` on parent.

Get an object that represents the parent app that this app is mounted inside. You can call `link` and `view` on it.

class morepath.**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Response.

Extends `webob.response.Response`.

morepath.**render_html**(*content*)
Take string and return text/html response.

morepath.**render_json**(*content*)
Take dict/list/string/number content and return json response.

morepath.**ANY = <ANY>**

class morepath.security.**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.

class morepath.security.**BasicAuthIdentityPolicy**(*realm='Realm'*)
Identity policy that uses HTTP Basic Authentication.

Note that this policy does **not** do any password validation. You're expected to do so using permission directives.

forget(*response*, *request*)
Forget identity on response.

This causes the browser to issue a basic authentication dialog. Warning: for basic auth, the browser in fact does not forget the information even if `forget` is called.

Parameters

- **response** (morepath.Response) – response object on which to forget identity.
- **request** (morepath.Request) – request object.

identify(*request*)
Establish claimed identity using request.

Parameters **request** (morepath.Request.) – Request to extract identity information from.

Returns morepath.security.Identity instance.

remember(*response*, *request*, *identity*)
Remember identity on response.

This is a no-op for basic auth, as the browser re-identifies upon each request in that case.

Parameters

- **response** (morepath.Response) – response object on which to store identity.
- **request** (morepath.Request) – request object.
- **identity** (morepath.security.Identity) – identity to remember.

morepath.security.**NO_IDENTITY = <morepath.security.NoIdentity object at 0x7feae3408410>**

class morepath.**Converter**(*decode*, *encode=None*)
How to decode from strings to objects and back.

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.

class morepath.**Config**
Contains and executes configuration actions.

Morepath configuration actions consist of decorator calls on App instances, i.e. `@app.view()` and `@app.path()`. The Config object can scan these configuration actions in a package. Once all required configuration is scanned, the configuration can be committed. The configuration is then processed, associated with

morepath.config.Configurable objects (i.e. App objects), conflicts are detected, overrides applied, and the configuration becomes final.

Once the configuration is committed all configured Morepath App objects are ready to be served using WSGI.

See setup(), which creates an instance with standard Morepath framework configuration. See also autoconfig() and autosetup() which help automatically load configuration from dependencies.

action(*action*, *obj*)
Register an action and obj with this config.

This is normally not invoked directly, instead is called indirectly by scan().

A Morepath directive decorator is an action, and obj is the function that was decorated.

Param The Action to register.

Obj The object to perform action on.

commit()
Commit all configuration.

- Clears any previous configuration from all registered morepath.config.Configurable objects.
- Prepares actions using prepared().
- Actions are grouped by type of action (action class).
- The action groups are executed in order of depends between their action classes.
- Per action group, configuration conflicts are detected.
- Per action group, extending configuration is merged.
- Finally all configuration actions are performed, completing the configuration process.

This method should be called only once during the lifetime of a process, before the configuration is first used. After this the configuration is considered to be fixed and cannot be further modified. In tests this method can be executed multiple times as it automatically clears the configuration of its configurables first.

configurable(*configurable*)
Register a configurable with this config.

This is normally not invoked directly, instead is called indirectly by scan().

A App object is a configurable.

Param The morepath.config.Configurable to register.

prepared()
Get prepared actions before they are performed.

The preparation phase happens as the first stage of a commit. This allows configuration actions to complete their configuration, do error checking, or transform themselves into different configuration actions.

This calls Action.prepare() on all registered configuration actions.

Returns An iterable of prepared action, obj combinations.

scan(*package=None*, *ignore=None*, *recursive=True*)
Scan package for configuration actions (decorators).

Register any found configuration actions with this object. This also includes finding any morepath.config.Configurable objects.

If given a package, it scans any modules and sub-packages as well recursively.

Parameters **package** – The Python module or package to scan. Optional; if left empty case the calling package is scanned.

Ignore A Venusian style ignore to ignore some modules during scanning. Optional.

Recursive Scan packages recursively. By default this is `True`. If set to `False`, only the `__init__.py` of a package is scanned.

class `morepath.config.`**`Configurable`**(*extends=None*, *testing_config=None*)
Object to which configuration actions apply.

Actions can be added to a configurable.

Once all actions are added, the configurable is executed. This checks for any conflicts between configurations and the configurable is expanded with any configurations from its extends list. Then the configurable is performed, meaning all its actions are performed (to it).

Parameters

- **extends** (*list of configurables, single configurable.*) – the configurables that this configurable extends. Optional.
- **testing_config** – We can pass a config object used during testing. This causes the actions to be issued against the configurable directly instead of waiting for Venusian scanning. This allows the use of directive decorators in tests where scanning is not an option. Optional, default no testing config.

`action`(*action*, *obj*)
Register an action with configurable.

This is normally not invoked directly, instead is called indirectly by `Config.commit()`.

Parameters

- **action** – The action to register with the configurable.
- **obj** – The object that this action is performed on.

`action_classes`()
Get action classes sorted in dependency order.

`action_extends`(*action_class*)
Get actions for action class in extends.

`actions`()
Actions the configurable wants to register as it is scanned.

A configurable may want to register some actions as it is registered with the config system.

Should return a sequence of action, obj tuples.

`clear`()
Clear any previously registered actions.

This is normally not invoked directly, instead is called indirectly by `Config.commit()`.

`execute`()
Execute actions for configurable.

`group_actions`()
Group actions into `Actions` by class.

class `morepath.config.`**`Action`**(*configurable*)
A configuration action.

A configuration action is performed on an object. Actions can conflict with each other based on their identifier and discriminators. Actions can override each other based on their identifier.

Can be subclassed to implement concrete configuration actions.

Action classes can have a `depends` attribute, which is a list of other action classes that need to be executed before this one is. Actions which depend on another will be executed after those actions are executed.

Initialize action.

Parameters **configurable** – `morepath.config.Configurable` object for which this action was configured.

clone(***kw*)
Make a clone of this action.

Keyword parameters can be used to override attributes in clone.

Used during preparation to create new fully prepared actions.

codeinfo()
Info about where in the source code the action was invoked.

By default there is no code info.

discriminators(*configurable*)
Returns a list of immutables to detect conflicts.

Parameters **configurable** – `morepath.config.Configurable` object for which this action is being executed.

Used for additional configuration conflict detection.

group_key()
By default we group directives by their class.

Override this to group a directive with another directive, by returning that Directive class. It will create conflicts between those directives. Typically you'd do this when you are already subclassing from that directive too.

identifier(*configurable*)
Returns an immutable that uniquely identifies this config.

Parameters **configurable** – `morepath.config.Configurable` object for which this action is being executed.

Used for overrides and conflict detection.

perform(*configurable*, *obj*)
Register whatever is being configured with configurable.

Parameters

- **configurable** – the `morepath.config.Configurable` being configured.
- **obj** – the object that the action should be performed on.

prepare(*obj*)
Prepare action for configuration.

Parameters **obj** – The object that the action should be performed on.

Returns an iterable of prepared action, obj tuples.

class morepath.converter.**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.

argument_and_explicit_converters(*arguments*, *converters*)
Use explict converters unless none supplied, then use default args.

converter_for_explicit_or_type(*c*)
Given a converter or a type, turn it into an explicit one.

converter_for_explicit_or_type_or_list(*c*)
Given a converter or type or list, turn it into an explicit one.

Parameters **c** – can either be a converter, or a type for which a converter can be looked up, or a list with a converter or a type in it.

Returns a Converter instance.

converter_for_type(*type*)
Get converter for type.

Is aware of inheritance; if nothing is registered for given type it returns the converter registered for its base class.

Parameters **type** – The type for which to look up the converter.

Returns a morepath.Converter instance.

converter_for_value(*v*)
Get converter for value.

Is aware of inheritance; if nothing is registered for type of given value it returns the converter registered for its base class.

Parameters **value** – The value for which to look up the converter.

Returns a morepath.Converter instance.

explicit_converters(*converters*)
Given converter dictionary, make everything in it explicit.

This means types have converters looked up for them, and lists are turned into ListConverter.

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**(*app*)

morepath.**directive**
alias of morepath.directive

1.17 Comparison with other Web Frameworks

We hear you ask:

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

Pyramid Design Chocies

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 go into some of this 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.

1.17.1 Overview

Morepath aims to be foundational. All web applications are different. Some are simple. Some, like CMSes, are like frameworks themselves. It's likely that some of you will need to build your own frameworky things on top of Morepath. Morepath doesn't get in your way. Morepath isn't there to be hidden away under another framework though - these extensions still look like Morepath. The 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 have URLs. This makes it like a URL traversal framework like Zope or Grok, and also like Pyramid when traversal is in use. 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 and generic function systems are all about. Want to change behavior? You can override everything. 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.

1.17.2 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 code organisation, as it allows you to separate the code that organises the URL space from the code that implements your actual views.

1.17.3 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*.

1.17.4 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*.

1.17.5 WSGI

Morepath is a WSGI-based framework, like Flask or Pyramid. It's natively WSGI, unlike Django, which while WSGI is supported also has its own way of doing middleware.

A Morepath app is a standard WSGI app. You can plug it into a WSGI compliant web server like Apache or Nginx or gunicorn. You can also combine Morepath with WSGI components, such as for instance the Fanstatic static resource framework.

1.17.6 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 this blog entry. (It will be integrated in this documentation later).

1.17.7 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.

1.17.8 Testability and Global state

Developers that care about writing 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 the framework 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. Implementations of generic functions can be plugged in separately per Morepath app: each app is a registry. When you call a generic function Reg needs to know what registry to use to look it up. You can make this completely explicit by using a special `lookup` argument:

```
some_generic_function(doc, 3, lookup=app.lookup())
```

That's all right in framework code, but doing that all the time is not very pretty in application code. For convenience, Morepath therefore sets up the current lookup implicitly as thread local state. Then you can simply write this:

```
some_generic_function(doc, 3)
```

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.

Summary: Morepath does not require any global state, but allows the current lookup to be set up as such for convenience.

1.17.9 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're not against writing examples or extensions that help you do this, though we haven't done so yet. Contribute!

1.17.10 No template language

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 and Mako template languages.

Morepath aims to be a good fit for modern, client-side web applications written in JavaScript. So we've focused on making 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.

We're planning on letting you plug in server-side template languages as they're sometimes useful, but we haven't done so yet. Feel free to contribute!

For now, you can plug in something yourself. CherryPy has a good document on how to do that with CherryPy, and it'd look very similar with Morepath.

1.17.11 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 current authentication system". 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 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 the original Zope.

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.

Morepath has a new configuration system that is based around decorators (using Venusian) attached to application objects. These application objects can extend other ones. This way it supports a range sophisticated extension and override use cases in a general way.

1.17.12 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 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.

1.18 Design Notes

Some of the use cases that influenced Morepath's design are documented here.

1.18.1 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.

1.18.2 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.

1.18.3 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 reuable 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.

1.18.4 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.

1.18.5 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.

1.18.6 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.

1.18.7 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.

1.18.8 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]

1.18.9 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.

1.18.10 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.

1.18.11 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.

1.18.12 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).

1.18.13 Conflicts

If you try to do the same registration multiple times, the system should fail explicitly, as otherwise this would lead to subtle errors.

1.18.14 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.

1.19 Developing Morepath

1.19.1 Community

Communication is important, so see *Community* for information on how to get in touch!

1.19.2 Install Morepath for development

First make sure you have virtualenv installed for Python 2.7.

Now create a new virtualenv somewhere for Morepath development:

```
$ virtualenv /path/to/ve_morepath
```

You should also be able to recycle an existing virtualenv, but this guarantees a clean one. Note that we skip activating the environment here, as this is just needed to initially bootstrap the Morepath buildout.

Clone Morepath from github and go to the morepath directory:

```
$ git clone git@github.com:morepath/morepath.git
$ cd morepath
```

Now we need to run bootstrap.py to set up buildout, using the Python from the virtualenv we've created before:

```
$ python /path/to/ve_morepath/bin/python/bootstrap.py
```

This installs buildout, which can now set up the rest of the development environment:

```
$ bin/buildout
```

This downloads and installs various dependencies and tools. The commands you run in `bin` are all restricted to the virtualenv you set up before. There is therefore no need to refer to the virtualenv once you have the development environment going.

1.19.3 Running the tests

You can run the tests using py.test. Buildout has installed it for you in the `bin` subdirectory of your project:

```
$ bin/py.test morepath
```

To generate test coverage information as HTML do:

```
$ bin/py.test morepath --cov morepath --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.

1.19.4 flake8

The buildout also installs flake8, which is a tool that can do various checks for common Python mistakes using pyflakes and checks for PEP8 style compliance.

To do pyflakes and pep8 checking do:

```
$ bin/flake8 morepath
```

1.19.5 radon

The buildout installs radon. This is a tool that can check various measures of code complexity.

To check for cyclomatic complexity (excluding the tests):

```
$ bin/radon cc morepath -e "morepath/tests*"
```

To filter for anything not ranked `A`:

```
$ bin/radon cc morepath --min B -e "morepath/tests*"
```

And to see the maintainability index:

```
$ bin/radon mi morepath -e "morepath/tests*"
```

1.20 CHANGES

1.20.1 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.

1.20.2 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.

1.20.3 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.

1.20.4 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%.

1.20.5 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.

1.20.6 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.

1.20.7 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)

1.20.8 0.1 (2014-04-08)

- Initial public release.

1.21 History of Morepath

For more recent changes, see *CHANGES*.

Morepath was written by Martijn Faassen (me writing this document).

The genesis of Morepath is complex and involves a number of projects.

1.21.1 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 I helped 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 I 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 I found WebOb the better fit for Morepath and switched to that.

1.21.2 Configuration system

In 2006 I 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

Now Morepath uses Venusian as a foundation to its configuration system.

1.21.3 Routing system

In 2009 I wrote a library called Traject:

https://pypi.python.org/pypi/traject

I 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. I tried to make this work properly with a relational database exposed through an ORM, but noticed that I had to adjust the object mapping too much just to please the traversal system.

This led me to a routing system, so expose the relational database objects to a URL. But I 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. I also wanted to maintain a separation between models and views. This led to the creation of Traject.

I used Traject successfully in a number of projects (based on Grok). I also ported Traject to JavaScript as part of the Obviel client-side framework. While Traject is fairly web-framework independent, to my knowledge Traject hasn't found much adoption elsewhere.

Morepath contains a further evolution of the Traject concept (though not the Traject library directly).

1.21.4 Reg

In early 2010 I started the iface project with Thomas Lotze. In 2012 I started the Crom project. Finally I combined them into the Comparch project in 2013. I then renamed Comparch to Reg, and finally converted Reg to a generic function implementation.

See Reg's history section for more information on its history. The Reg project provides the fundamental registries that Morepath builds on.

1.21.5 Publisher

In 2010 I 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 I'd 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. I didn't use Dawnlight myself, 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.

1.21.6 Combining it all

In 2013 I started to work with CONTACT Software. They encouraged me to rethink these various topics. This led me to combine these lines of development into Morepath: Reg registries, decorator-based configuration using Venusian, and traject-style publication of models and resources.

1.21.7 Spinning a Web Framework

In the fall of 2013 I gave a keynote speech at PyCon DE about the creative processes behind Morepath, called "Spinning a Web Framework":

CHAPTER 2

Indices and tables

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

m

morepath, 63