pyramid_blogr Documentation

Release 1

Marcin Lulek

November 07, 2015

Contents

Pyramid_blogr was originally inspired by Flaskr tutorial and is aimed to introduce readers to basic concepts of the Pyramid Web Framework and web application development.

Note: This tutorial is meant for **Python 2.6+** and **Python 3.3+** versions of the language.

CHAPTER 1

Contents

1.1 What you will know after you finish this tutorial

This tutorial is really simple, but it should give you enough of a head start to writing applications using the Pyramid Web Framework. By the end of it, you should have a basic understanding of templating, working with databases, using URL routes to launch business logic (views), authentication, authorization, using a form library, and usage and pagination of our blog entries.

Pyramid_blogr makes some initial assumptions:

- We will use Pyramid's **alchemy** scaffold with SQLAlchemy as its ORM layer.
- **Mako** templates will be our choice for templating engine.
- **URL dispatch** will be the default way for our view resolution.
- A single user in the database will be created during the setup phase.
- We will perform simple authentication of the user.
- The authenticated user will be authorized to make blog entries.
- The blog entries will be listed from newest to oldest.
- We will use the **webhelpers** package for pagination.
- The **WTForms** form library will provide form validation.

This tutorial was originally created by **Marcin Lulek**, developer, freelancer and founder of app enlight.

Start the tutorial.

1.1.1 Documentation links

- Pyramid
- Mako templates
- SQLAlchemy
- Webhelpers
- WTForms

The complete source code for this application is available on GitHub at:

https://github.com/Pylons/pyramid_blogr

1.2 1. Create your tutorial project structure

Hint: At the time of writing, 1.5.7 was the most recent stable version of Pyramid, you can use newer version of Pyramid but there will be some slight differences in default project templates.

First we need to install Pyramid framework itself:

```
pip install pyramid==1.5.7
```

This will install Pyramid itself with its base dependencies, your Python environment (ideally VirtualEnv), will now contain some helpful commands including:

- **pcreate** used to create fresh project and directory structures from Pyramid scaffolds(project templates) that Pyramid ships with
- **pserve** will be used to start our WSGI server

The next step is to create our project using alchemy scaffold - that will provide SQLAlchemy as our default ORM layer:

```
~/yourVenv/bin/pcreate -s alchemy pyramid_blogr
```

We will end up with pyramid_blogr dir that should have following structure:

```
pyramid_blogr/
-- __init__.py <- main file that will configure and return WSGI application
-- models.py   <- model definitions aka data sources (often RDBMS or noSQL)
-- scripts/    <- util Python scripts
-- static/     <- usually css, js, images
-- templates/  <- template files
-- tests.py    <- tests
-- views.py    <- views aka business logic
```

1.2.1 Adding dependencies to the project

Since Pyramid tries its best to be a non-opinionated solution we will have to decide what libraries we want for form handling and template helpers. For this tutorial we will use great WTForms library and webhelpers packages.

To make them dependencies of our application we need to open setup.py file and extend **requires** with additional packages, in the end it should look like this:

```
requires = [
    'pyramid==1.5.7',
    'pyramid_mako', # replaces default chameleon templates
    'pyramid_debugtoolbar',
    'pyramid_tm',
    'SQLAlchemy==1.0.8',
    'transaction',
    'zope.sqlalchemy',
    'waitress',
    'wtforms==2.0.2',  # form library
    'webhelpers2==2.0', # various web building related helpers
    'paginate==0.5', # pagination helpers
    'paginate_sqlalchemy==0.2.0'
    ]
```

Now we can setup our application for development and add it to our environment path. In the root of our project where setup.py lives execute following line:

```
~/yourVenv/bin/pip install -e .
```

This will install all the requirements for our application and will make it importable in our Python environment.

> **Warning:** Don't forget to add the . after -e switch

Another side effect of this command is that our environment gained another command called **initialize_pyramid_blogr_db**, we will use it to create/populate the database from the models we will create in a moment, this script will also create the default user for our application.

1.2.2 Running our application

To visit our application we need to use a WSGI server that will start serving the content to the browser with following command:

```
~/yourVenv/bin/pserve --reload development.ini
```

This will launch an instance of a WSGI (waitress by default) server that will run both your application code and static files. **development.ini file is used to provide all the configuration details**, the *–reload* parameter tells the server to restart our application every time its code changes, this is a great setting for fast development and testing live changes to our app.

Unfortunately on our first run the application will throw exception:

```
ImportError: No module named 'pyramid_chameleon'
```

This is because we switched from chameleon templating engine to mako.

To fix this you need to open *pyramid_blogr/__init__.py* and change:

```
config.include('pyramid_chameleon')
# to
config.include('pyramid_mako')
```

On your next application restart should see something like this:

```
Starting subprocess with file monitor
Starting server in PID 8517.
serving on http://0.0.0.0:6543
```

You can open your favorite browser and go to http://localhost:6543/ to see how our application looks like.

Unfortunately you will see something like this instead of a webpage ;-)

```
Pyramid is having a problem using your SQL database.  The problem...
```

This is where the **initialize_pyramid_blogr_db** command comes into play, but before we run it we need to create our application models.

Next 2. Create database models

1.3 2. Create database models

At this point we should create our models. In a nutshell models represent data and its underlaying storage mechanisms in an application.

We will use a relational database and sqlalchemy's ORM layer to access our data.

The default pyramid scaffold provides an example model class *MyModel* that we don't need - so we first need to remove whole of it.

In real life applications data models tend to grow over time and contain lots of additional methods. Instead of keeping all of our models in a single file, let's create a new *models* package in our structure that will hold one model per file.

Now we need to move the file *models.py* to our newly created directory. Let's rename it *meta.py* to make a python package from our *models* directory.

Our directory structure should look like this after this operation:

```
pyramid_blogr/
-- __init__.py <- main file that will configure and return WSGI application
-- models      <- model definitions aka data sources (often RDBMS or noSQL)
|     -- __init__.py
|     -- meta.py <- former models.py
-- scripts/    <- util python scripts
-- static/     <- usually css, js, images
-- templates/  <- template files
-- tests.py    <- tests
-- views.py    <- views aka business logic
```

Our application will consist of two tables:

- **users** - stores all users for our application
- **entries** - stores our blog entries

We should assume that our users might use some non-english characters, so we need to import the Unicode datatype from sqlalchemy, we will also need a DateTime field to timestamp our blog entries.

Let's first create *models/user.py*. Let's remove the now unused import code from *models/meta.py* and paste it into our newly created user file. While we are doing this, let's add a few more imports to our *user.py* file.

```
import datetime #<- will be used to set default dates on models
from pyramid_blogr.models.meta import Base  #<- we need to import our sqlalchemy metadata for model
from sqlalchemy import (
    Column,
    Integer,
    Unicode,      #<- will provide unicode field,
    UnicodeText, #<- will provide unicode text field,
    DateTime      #<- time abstraction field,
    )
```

Now repeat the step and insert the same code into *models/blog_record.py*

After all operations our *models/meta.py* should only contain:

```
from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import (
    scoped_session,
    sessionmaker,
)

from zope.sqlalchemy import ZopeTransactionExtension

DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
Base = declarative_base()
```

And our project structure should look like this:

```
pyramid_blogr/
-- __init__.py <- main file that will configure and return WSGI application
-- models      <- model definitions aka data sources (often RDBMS or noSQL)
|     -- __init__.py
|     -- meta.py <- former models.py
|     -- blog_record.py
|     -- user.py
-- scripts/    <- util python scripts
-- static/     <- usually css, js, images
-- templates/  <- template files
-- tests.py    <- tests
-- views.py    <- views aka business logic
```

1.3.1 Database session management

Hint: To learn how to use sqlalchemy please consult its Object Relational Tutorial: http://docs.sqlalchemy.org/en/latest/orm/tutorial.html

If you are new to sqlalchemy or ORM's you are probably wondering what this code does:

```
DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
Base = declarative_base()
```

The first line initializes sqlalchemy's threaded **session maker** - we will use it to interact with the database and persist our changes to the database. It is thread-safe meaning that it will handle multiple requests at same time in a safe way, and our code from different views will not impact other requests. It will also open and close database connections for us transparently when needed.

The *extension=ZopeTransactionExtension()* we pass as a parameter to sessionmaker in order to use the registered zope transaction extension. This will work with pyramid's transaction manager (pyramid_tm).

1.3.2 What does transaction manager do?

WHOA THIS SOUNDS LIKE SCARY MAGIC!!

Hint: It's not.

Ok, so while it might sound complicated - in practice it's very simple and saves a developer a lot of headaches managing transactions inside application.

How it works:

- A transaction is started when a browser request invokes our view code
- Some operations take place; for example database rows are inserted/updated in our favorite datastore
  - if everything went fine - we don't need to commit our transaction explictly, transaction manager will do this for us
  - if some unhandled exception occured - at this point we usually want to roll back all the changes/queries that were sent to our datastore - transaction manager will handle this for us

What are the implications of this?

Imagine you have an application that sends a confirmation email every time a user registers. A user, John, inputs the data to register, we send John a nice welcome email and maybe an activation link, but during registration flow something unexpected happens and the code errored out.

It is very common in this situation that the user would get a welcome email, but in reality his profile was never persisted in the database. With packages like **pyramid_mailer** it is perfectly possible to delay email sending until **after** the user's information is successfully saved in the database.

Nice, huh?

But this is a more advanced topic not covered in this tutorial, the most simple explanation is that transaction manager will make sure our data gets correctly saved if everything went smoothly and if an error occurs - our datastore modifications are rolled back.

1.3.3 Adding model definitions

Hint: This will make the app error out and prevent it from starting till we reach the last point of current step and fix imports in other files. It's perfectly normal, so don't worry about this.

We will need two declarations of models that will replace the *MyModel* class that was created when we scaffolded our project:

After the import part of *models/user.py* add the following:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode(255), unique=True, nullable=False)
    password = Column(Unicode(255), nullable=False)
    last_logged = Column(DateTime, default=datetime.datetime.utcnow)
```

After the import part of *models/blog_record.py* add the following:

```
class BlogRecord(Base):
    __tablename__ = 'entries'
    id = Column(Integer, primary_key=True)
    title = Column(Unicode(255), unique=True, nullable=False)
    body = Column(UnicodeText, default=u'')
    created = Column(DateTime, default=datetime.datetime.utcnow)
    edited = Column(DateTime, default=datetime.datetime.utcnow)
```

Now its time to update our *models/__init__.py* to include our models - this is especially handy because it ensures that sqlalchemy mappers will pick up all our model classes and functions like *create_all* do what you expect them to do.

Add these imports to the end of the file:

```
from .user import User
from .blog_record import BlogRecord
```

1.3.4 Update initialization script

It's time to update our database initialization script to mirror the changes in models.py.

For this we need to open */pyramid_blogr/scripts/initializedb.py* - this is the file that actually gets executed when we run *initialize_pyramid_blogr_db*.

First remove *MyModel* import from that file and fix imports from modules package, also import *User* model:

```
from ..models.meta import DBSession, Base
from ..models import User
```

Since MyModel model is now gone we want to replace:

```
with transaction.manager:
    model = MyModel(name='one', value=1)
    DBSession.add(model)
```

with:

```
with transaction.manager:
    admin = User(name=u'admin', password=u'admin')
    DBSession.add(admin)
```

When you initialize a fresh database this will populate it with a single user, with both login and unencrypted password equal to admin.

> **Warning:** This is just a tutorial example and **production code should utilize passwords hashed with a strong one-way encryption function**. You can use a package like **passlib** or **cryptacular** for this purpose.

The last step is to fix the imports from MyModel to User model and meta package in __init__.py.

in *pyramid_blogr/__init__.py*:

```
from .models import (
    DBSession,
    Base,
    )
```

becomes:

```
from .models.meta import (
    DBSession,
    Base,
    )
```

> **Warning:** Remember to replace the imports of MyModel, DBSession classes in */pyramid_blogr/scripts/initializedb.py* **and** */pyramid_blogr/tests.py*, otherwise your app will not start because of failed imports.

Same as with models, when your application grows over time you will want to organize views into logical sections based on their functionality. Fow now remove the *views.py* completely.

Our application should start again if we try running the server. In case you have problems starting the application, you can see complete source code of the files we modifed below.

If our application starts correctly, you should run the *initialize_pyramid_blogr_db*, command from your environment, it may look like this:

```
~/yourVenv/bin/initialize_pyramid_blogr_db development.ini
```

Next 3. Application routes

1.4 3. Application routes

This is the point where we want to define our routes that will be used to map view callables to request paths.

URL dispatch provides a simple way to map URLs to view code using a simple pattern matching language.

Our application will consist of few sections:

- index page that will list all of our sorted blog entries
- a sign in/sign out section that will be used for authentication
- a way to create and edit our blog posts

Our urls could look like this:

To sign in users:

```
/sign/in
```

So when user visits http://somedomain.foo/sign/in - the view callable responsible for signing in the user based on POST vars will be executed.

To sign out users:

```
/sign/out
```

Index page (it is already defined via default scaffold under name "home"):

```
/
```

Creation of new blog entries:

```
/blog/{action}
```

You probably noted that this url looks somewhat different, the {action} part in the pattern determines that this part is dynamic, so our URL could look like:

```
/blog/create
/blog/edit
/blog/foobar
```

This single route could map to different views.

Finally a route used to for our blog entries:

```
/blog/{id:\d+}/{slug}
```

This route constists of two dynamic parts, {id:\d+} and {slug}.

The **:d+** pattern means that the route will only match integers, so url like:

```
/blog/156/Some-blog-entry
```

would work, but this one will not be matched:

```
/blog/something/Some-blog-entry
```

1.4.1 Basics of pyramid configuration

Now that we know what routes we want we should add them to our application.

Pyramid's config object will store them for us. To access it we will need to open the file __init__.py in root of our project. This is the central point that will perform initial application configuration on runtime.

The main function will accept parsed ini file that we passed to our pserve command.

Lets quickly go over what this file does by default.

```
engine = engine_from_config(settings, 'sqlalchemy.')
DBSession.configure(bind=engine)
```

Those lines read the settings for SQLAlchemy and configure connection engine and session maker objects.

```
config = Configurator(settings=settings)
```

This creates the configurator itself, when needed we will be able to access it in our views via request object as *request.registry.settings*.

```
config.add_static_view('static', 'static', cache_max_age=3600)
config.add_route('home', '/')
```

Now two routes are added:

- **static route** that starts with */static* - that will serve all our static files like javascript, css, images. When a browser makes a request to */static/some/resource.foo*, our application will check if /some/resource.foo resource is present in our static dir, if it's there it will get served to browser.
- **view route** called *"home"* that maps to path */*.

```
config.scan()
```

This runs the scan process that will scan all our whole project and load all decorators and includes to add them to our config object.

```
return config.make_wsgi_app()
```

Instance of WSGI app is returned to the server.

1.4.2 Adding routes to application configuration

Lets add our routes to configurator after "home" route:

```
config.add_route('blog', '/blog/{id:\d+}/{slug}')
config.add_route('blog_action', '/blog/{action}')
config.add_route('auth', '/sign/{action}')
```

Now we are ready to develop actual views

Next 4. Initial views

1.5 4. Initial views

Now it's time to create our views files and add our view callables.

Every view will be decorated with **@view_config** decorator.

@view_config will configure our pyramid application by telling it how to corellate our view callables with routes, also setting some restrictions on specific view resolution mechanisms. It's is being picked up when config.scan() gets run from our __init__.py, all of our views are registerd with our app.

Hint: You could do it explictly with **config.add_view()** method but this approach is often more convenient.

First lets create a new package called views and create 3 files, *views/__init__.py*, *views/default.py* and *views/blog.py*.

Your project structure should look like this at this point:

```
pyramid_blogr/
-- __init__.py <- main file that will configure and return WSGI application
-- models      <- model definitions aka data sources (often RDBMS or noSQL)
|     -- __init__.py
|     -- meta.py <- former models.py
|     -- blog_record.py
|     -- user.py
-- scripts/    <- util python scripts
-- static/     <- usually css, js, images
-- templates/  <- template files
-- tests.py    <- tests
-- views    <- views aka business logic
      -- __init__.py <- empty
      -- blog.py
      -- default.py
```

Lets make some stubs for our views, we will populate them with actual code in next chapters.

In *views/default.py* add:

```
from pyramid.view import view_config

@view_config(route_name='home', renderer='pyramid_blogr:templates/index.mako')
def index_page(request):
    return {}
```

Here @view_config takes 2 params that will register our index_page callable within pyramid's registry, specifying the route that should be used to match this view, we also specified renderer that will be used to transform the data view returns into response suitable for the client.

The template location is specified using *asset location* format which is in form of *package_name:path_to_template*.

Hint: It also easy to add your own custom renderer, or use a drop in package like *pyramid_jinja2*.

The renderer is picked up automaticly by specifying file extension like: *asset.mako/asset.jinja2* or when your provide name for string/json renderer.

Pyramid by provides few renderers including:

- mako templates
- chameleon templates
- string output
- json encoder

In *views/blog.py* add:

```
from pyramid.view import view_config

@view_config(route_name='blog', renderer='pyramid_blogr:templates/view_blog.mako')
def blog_view(request):
    return {}
```

Registers blog_view with a route named "blog" using view_blog.mako template as response.

The next views we should create are views that will handle creation and updates to our blog entries.

```
@view_config(route_name='blog_action', match_param='action=create',
             renderer='pyramid_blogr:templates/edit_blog.mako')
```

```
def blog_create(request):
    return {}
```

Notice that there is a new keyword introduced to @view_config decorator.

match_params purpose is to tell pyramid which view callable to use when our dynamic part of route {action} is matched, so this view will be launched for following URL: */blog/create*.

And then we have the view for */blog/edit* URL.

```
@view_config(route_name='blog_action', match_param='action=edit',
             renderer='pyramid_blogr:templates/edit_blog.mako')
def blog_update(request):
    return {}
```

Hint: Every view can be decorated unlimited times with different parameters passed to @view_config, .

Now back in *views/default.py* add:

```
@view_config(route_name='auth', match_param='action=in', renderer='string',
             request_method='POST')
@view_config(route_name='auth', match_param='action=out', renderer='string')
def sign_in_out(request):
    return {}
```

These routes will handle user authentication and logout. They are not using any template because they will just perform HTTP redirects.

Note that this view is decorated more than once, also it introduces one new parameter.

request_method just restricts view resolution to specific request method, this route will not be reachable with GET requests.

Hint: if you navigate your browser directly to /sign/in - you will get a 404 page, because this view is not matched for GET requests.

At this point we can start implementing our view code.

Next 5. Blog models and views

1.6 5. Blog models and views

1.6.1 Models

Since our stubs are in place we can start developing blog related code.

First lets start with models, now that we have them we can create some service classes and implement some methods that we will use in our views and templates.

Lets create a new module **models/services/__init__.py** and then open **models/services/blog_record.py** and import some helper modules to generate our slugs, add pagination, and print nice dates - they will all come from excellent webhelpers package - so the top of blog_record.py should have following imports added:

```
import sqlalchemy as sa
from paginate_sqlalchemy import SqlalchemyOrmPage #<- provides pagination
from ..meta import DBSession
from ..blog_record import BlogRecord
```

Now we need to create our BlogRecordService with following methods:

```
class BlogRecordService(object):

    @classmethod
    def all(cls):
        return DBSession.query(BlogRecord).order_by(sa.desc(BlogRecord.created))
```

This method will return query object that can return whole dataset to us when needed.

The query object will be sorting the rows by date in descending order.

```
@classmethod
def by_id(cls, id):
    return DBSession.query(BlogRecord).filter(BlogRecord.id == id).first()
```

This method will return a single blog entry by id, or None object if nothig is found.

```
@classmethod
def get_paginator(cls, request, page=1):
    query = DBSession.query(BlogRecord).order_by(sa.desc(BlogRecord.created))
    query_params = request.GET.mixed()

    def url_maker(link_page):
        # replace page param with values generated by paginator
        query_params['page'] = link_page
        return request.current_route_url(_query=query_params)
    return SqlalchemyOrmPage(query, page, items_per_page=5,
                             url_maker=url_maker)
```

get_paginator method will return an excellent paginator that is able to return us only the entries from specific "page" of database resulteset. It will add LIMIT/OFFSET to our query based on items_per_page and current page number.

Paginator uses SqlalchemyOrmPage wrapper that will attempt to generate a paginator with links, link urls will be constructed using *url_maker* function that uses request object to generate new url from current one replacing page query param with new value.

Your project structure should look like this at this point:

```
pyramid_blogr/
-- __init__.py <- main file that will configure and return WSGI application
-- models      <- model definitions aka data sources (often RDBMS or noSQL)
|     -- services      <- they query the models for data
|     |     -- __init__.py
|     |     -- blog_record.py
|     -- __init__.py
|     -- meta.py <- former models.py
|     -- blog_record.py
|     -- user.py
-- scripts/    <- util python scripts
-- static/     <- usually css, js, images
-- templates/  <- template files
-- tests.py    <- tests
-- views    <- views aka business logic
      -- __init__.py <- empty
      -- blog.py
      -- default.py
```

Now it is time to add imports and properties to **models/blog_record.py**:

```
from webhelpers2.text import urlify #<- will generate slugs
from webhelpers2.date import distance_of_time_in_words #<- human friendly dates
```

And to model itself:

```
@property
def slug(self):
    return urlify(self.title)
```

This property of entry instance will return nice slugs for us to use in urls, title of "Foo Bar Baz" will become "Foo-Bar-Baz". Also non-latin characters will be approximated to their closest counterparts.

```
@property
def created_in_words(self):
    return distance_of_time_in_words(self.created, datetime.datetime.utcnow())
```

This property will return information when specific entry was created in a friendly form like "2 days ago".

1.6.2 Index view

First lets add our BlogRecord service to imports in views/default.py:

```
from ..models.services.blog_record import BlogRecordService
```

Now it's time to implement our actual index view:

```
@view_config(route_name='home', renderer='pyramid_blogr:templates/index.mako')
def index_page(request):
    page = int(request.params.get('page', 1))
    paginator = BlogRecordService.get_paginator(request, page)
    return {'paginator': paginator}
```

We first retrieve from url the page number we want to present to the user, if not present it defaults to 1.

The paginator object returned by *BlogRecord.get_paginator* will then be used in template to build nice list of entries.

Hint: Everything we return from our views in dictionaries will be available in templates as variables. So if we return {'foo':1, 'bar':2} we will be able to access the variables inside the template directly as *foo* and *bar*.

1.6.3 Index view template

For the purpose of this tutorial please use ready-made mako templates + some minimal page styling.

First delete everything in /templates folder.

We will now create layout.mako template file that will store a "master" template that other view templates will inherit from. This template will contain page header and footer shared by all pages.

In /templates please create "layout.mako" with following contents:

```
<!DOCTYPE html>
<html lang="${request.locale_name}">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="pyramid web application">
```

```
    <meta name="author" content="Pylons Project">
    <link rel="shortcut icon" href="${request.static_url('pyramid_blogr:static/pyramid-16x16.png')}">

    <title>Alchemy Scaffold for The Pyramid Web Framework</title>

    <!-- Bootstrap core CSS -->
    <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css" rel="stylesheet">

    <!-- Custom styles for this scaffold -->
    <link href="${request.static_url('pyramid_blogr:static/theme.css')}" rel="stylesheet">
    <style type="text/css">
    a, a:link, a:visited{
        color: #ffcc00;
        font-weight: bold;
    }
    a:hover{
        color: #ffff00
    }

    </style>
    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
      <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
      <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>
    <![endif]-->
  </head>

  <body>

    <div class="starter-template">
      <div class="container">
        <div class="row">
          <div class="col-md-2">
            <img class="logo img-responsive" src="${request.static_url('pyramid_blogr:static/pyramid
          </div>
          <div class="col-md-10">
            <div class="content">
              <h1><span class="font-semi-bold">Pyramid</span> <span class="smaller">Alchemy scaffold<
              <p class="lead">Welcome to <span class="font-normal">Pyramid Blogr</span>, an appl

                <div>
                    <!-- this is where contents of template inheriting from this layout will be inser
                  ${next.body()}
                    <!-- this is where contents of template inheriting from this layout will be inser
                </div>

            </div>
          </div>
        </div>
        <div class="row">
          <div class="links">
            <ul>
              <li class="current-version">Generated by v1.5.7</li>
              <li><i class="glyphicon glyphicon-bookmark icon-muted"></i><a href="http://docs.pylonsp
              <li><i class="glyphicon glyphicon-cog icon-muted"></i><a href="https://github.com/Pylon
              <li><i class="glyphicon glyphicon-globe icon-muted"></i><a href="irc://irc.freenode.net
              <li><i class="glyphicon glyphicon-home icon-muted"></i><a href="http://pylonsproject.or
          </div>
```

```
        </div>
        <div class="row">
          <div class="copyright">
            Copyright &copy; Pylons Project
          </div>
        </div>
      </div>
    </div>


    <!-- Bootstrap core JavaScript
    ================================================== -->
    <!-- Placed at the end of the document so the pages load faster -->
    <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></script>
    <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/bootstrap.min.js"></script>
  </body>
</html>
```

Hint: request object is always available inside your templates namespace

Inside your template you will notice that we used request.static_url method, that will generate correct links to your static assets, this is handy when building apps using URL prefixes.

In the middle of template you will also notice **${next.body()}** tag - after we render a template that inherits from our layout file - this is the place where our index template (or another for other view) will appear.

Now lets create another template called index.mako with following contents:

```
<%inherit file="pyramid_blogr:templates/layout.mako"/>
<% link_attr={"class": "btn btn-default btn-xs"} %>
<% curpage_attr={"class": "btn btn-default btn-xs disabled"} %>
<% dotdot_attr={"class": "btn btn-default btn-xs disabled"} %>


% if paginator.items:

    <h2>Blog entries</h2>

    <ul>
        % for entry in paginator.items:
            <li>
                <a href="${request.route_url('blog', id=entry.id, slug=entry.slug)}">
                    ${entry.title}</a>
            </li>
        % endfor
    </ul>

    ${paginator.pager(link_attr=link_attr, curpage_attr=curpage_attr, dotdot_attr=dotdot_attr) |n}

% else:

    <p>No blog entries found.</p>

%endif

<p><a href="${request.route_url('blog_action',action='create')}">
    Create a new blog entry</a></p>
```

This template inherits from layout.mako which means that it's contents will be wrapped by layout provided by parent template.

${paginator.pager()} - will print nice paginator links (it will only show up, if you have more than 5 blog entries in database)

${request.route_url - is used to generate links based on routes defined in our project. For example:

```
${request.route_url('blog_action',action='create')} -> /blog/create
```

1.6.4 Blog view

Time to update our blog view.

At the top of views/blog.py lets add following imports:

```
from pyramid.httpexceptions import HTTPNotFound, HTTPFound
from ..models.meta import DBSession
from ..models.blog_record import BlogRecord
from ..models.services.blog_record import BlogRecordService
```

Those exceptions will be used to perform redirects inside our apps.

- **HTTPFound** will return a 302 HTTP code response, it can accept *location* argument that will add a Location: header for the browser - we will perform redirects to other pages this way.
- **HTTPNotFound** on other hand will just make the server serve a standard 404 response.

```
@view_config(route_name='blog', renderer='pyramid_blogr:templates/view_blog.mako')
def blog_view(request):
    blog_id = int(request.matchdict.get('id', -1))
    entry = BlogRecordService.by_id(blog_id)
    if not entry:
        return HTTPNotFound()
    return {'entry': entry}
```

This view is also very simple, first we get the id variable from our route. It will be present in **matchdict** property of request object - all of our defined route arguments will end up there.

After we get entry id, that will be passed to BlogRecord classmethod **by_id()** to fetch specific blog entry, if it's found - we return the db row for the template to use, otherwise we present user with standard 404 response.

1.6.5 Blog view template

The template used for blog article presentation is named view_blog.mako:

```
<%inherit file="pyramid_blogr:templates/layout.mako"/>

<h1>${entry.title}</h1>
<hr/>
<p>${entry.body}</p>
<hr/>
<p>Created <strong title="${entry.created}">
    ${entry.created_in_words}</strong> ago</p>

<p><a href="${request.route_url('home')}">Go Back</a> ::
    <a href="${request.route_url('blog_action', action='edit',
    _query={'id':entry.id})}">Edit entry</a>
```

```
</p>
```

The **_query** argument introduced here to url generator is a list of k,v tuples , that will be used to append GET(query) parameters, in our case it will be ?id=X.

If you start the application now you will get an empty welcome page stating that "No blog entries are found"

Next 6. Adding and editing blog entries

1.7 6. Adding and editing blog entries

1.7.1 Form handling with WTForms library

For form validation and creation we will use a very friendly and easy to use form library called WTForms. First, we need to define our form schemas, that will be used to generate form HTML and validation of form fields.

First in root of our application lets create file *forms.py* with following contents:

```
from wtforms import Form, StringField, TextAreaField, validators
from wtforms import HiddenField

strip_filter = lambda x: x.strip() if x else None

class BlogCreateForm(Form):
    title = StringField('Title', [validators.Length(min=1, max=255)],
                        filters=[strip_filter])
    body = TextAreaField('Contents', [validators.Length(min=1)],
                         filters=[strip_filter])

class BlogUpdateForm(BlogCreateForm):
    id = HiddenField()
```

We create a simple filter that will be used to remove all the whitespace from the beginning and end of our input.

Then we create a BlogCreateForm class that defines 2 fields:

- **title** - it has a label of "Title" and single validator that will check the length of our trimmed data - the title length needs to be between 1-255 characters.
- **body** has a label of "Contents", also has a validator that requires its length to be at least 1 character.

Next is BlogUpdateForm class that inherits all the fields from BlogCreateForm, and adds a new hidden field called id - it will be used to determine which entry we want to update.

1.7.2 Create blog entry view

Now that our simple form definition is ready we can actually write our view code.

Lets start by importing our freshly created form schemas to views/blog.py:

```
from ..forms import BlogCreateForm, BlogUpdateForm
```

Next we implement actual view callable that will handle new entries for us:

```
@view_config(route_name='blog_action', match_param='action=create',
             renderer='pyramid_blogr:templates/edit_blog.mako')
def blog_create(request):
```

```
    entry = BlogRecord()
    form = BlogCreateForm(request.POST)
    if request.method == 'POST' and form.validate():
        form.populate_obj(entry)
        DBSession.add(entry)
        return HTTPFound(location=request.route_url('home'))
    return {'form': form, 'action': request.matchdict.get('action')}
```

What it does step by step:

- we create a new fresh entry row and form object from BlogCreateForm
- the form will be populated via POST if present
- if request method is POST the form gets validated
- if the form is valid - our form sets its values to the model instance, and adds it to the database session
- redirect to index page is performed

If the form doesn't validate correctly, view result is returned and standard HTML response is returned instead - form markup will have error messages included.

1.7.3 Create update entry view

The following view will handle updates to existing blog entries:

```
@view_config(route_name='blog_action', match_param='action=edit',
             renderer='pyramid_blogr:templates/edit_blog.mako')
def blog_update(request):
    blog_id = int(request.params.get('id', -1))
    entry = BlogRecordService.by_id(blog_id)
    if not entry:
        return HTTPNotFound()
    form = BlogUpdateForm(request.POST, entry)
    if request.method == 'POST' and form.validate():
        form.populate_obj(entry)
        return HTTPFound(location=request.route_url('blog', id=entry.id,
                                                    slug=entry.slug))
    return {'form': form, 'action': request.matchdict.get('action')}
```

What it does step by step:

- we fetch blog entry from the database based on the "id" query parameter
- we show 404 page if its hot present
- the form object is created, it gets populated from POST, and actual entry if we haven't POST-ed any values yet.

Hint: This approach ensures our form is always populated with latest data from database, OR if it's not validated - with values we posted in our last request.

- if the form is valid - our form sets its values to the model instance
- redirect to blog page is performed

The final step is to add a view that will present users with form to create and edit entries, lets call it *edit_blog.mako*

```
<%inherit file="pyramid_blogr:templates/layout.mako"/>

<form action="${request.route_url('blog_action',action=action)}" method="post" class="form">
```

```
    %if action =='edit':
        ${form.id()}
    %endif

    % for error in form.title.errors:
        <div class="error">${ error }</div>
    % endfor

    <div class="form-group">
        <label for="title">${form.title.label}</label>
            ${form.title(class_='form-control')}
    </div>

    % for error in form.body.errors:
        <div class="error">${error}</div>
    % endfor

    <div class="form-group">
        <label for="body">${form.body.label}</label>
        ${form.body(class_='form-control')}
    </div>
    <div class="form-group">
        <label></label>
        <button type="submit" class="btn btn-default">Submit</button>
    </div>


</form>
<p><a href="${request.route_url('home')}">Go Back</a></p>
```

Our template knows if we are creating new row or updating existing one based on action variable value, if we are editing existing row - it will add a hidden field "id" that holds the id of entry that is being updated.

If the form doesn't validate field errors properties contain lists of errors for us to present to user.

If you visit http://localhost:6543/ you will notice that you can already create and edit blog entries. Now it is time to work towards securing them.

Hint: Because WTForms form instances are iterable you can easly write a template, function that will iterate over their fields and auto generate dynamic html for each of them.

Next 7. Authorization

1.8 7. Authorization

At this point we have a fully working application but you have probably noticed everyone can alter our entries. We should change that by introducing user authentication and permission checks.

For the sake of simplicity of this tutorial we will assume that every user can edit every blog entry as long as he/she is signed in to our application.

Pyramid provides some ready-made policies for this and mechanisms for writing custom ones aswell.

We will use the ones provided by the framework:

- **AuthTktAuthenticationPolicy**

  Obtains user data from a Pyramid "auth ticket" cookie.

- **ACLAuthorizationPolicy**

  An authorization policy which consults an ACL object attached to a context to determine authorization information about a principal or multiple principals.

OK, so **ACLAuthorizationPolicy** explanation has a lots of scary words in it, but in practice it's a simple concept that allows for great flexibility when defining permission systems.

The policy basicly checks if user has a permission to specific context of a view based on Access Control Lists.

What does this mean, what is a context?

A context could be anything, imagine you are building a forum application, and you want to add a functionality where only moderators will be able to edit specific topic of a specific forum. - in this case our context would be forum object - it would have attached info about who has specific permissions to this resource.

Or something simplier, who can access admin page? In this case a context would be an arbitrary object that has information attached about who is administrator of the site.

How does this relate to our application?

Since our application does not track who owns blog entries, we will also assume the latter scenario. We will make the most trivial context factory object - as its name implies factory will return the context object (in our class an arbitrary class).

It will say that *everyone logged* to our application can create and edit, blog entries.

In root of our application package lets create a new file called *security.py* with following contents

```
from pyramid.security import Allow, Everyone, Authenticated

class BlogRecordFactory(object):
    __acl__ = [(Allow, Everyone, 'view'),
               (Allow, Authenticated, 'create'),
               (Allow, Authenticated, 'edit'), ]

    def __init__(self, request):
        pass
```

This is the object that was mentioned a moment ago (It's called context factory), it's **not** tied to any specific entity in a database, and returns __acl__ property that says that everyone has a *'view'* permission and users that are logged in also have *create* and *edit* permissions.

Now it's time to tell pyramid about what policies we want to register with our app.

Let's open our configuration related *__init__.py* and add following imports:

```
from pyramid.authentication import AuthTktAuthenticationPolicy
from pyramid.authorization import ACLAuthorizationPolicy
```

Now it's time to update our configuration, we need to create our policies, and pass them to configurator:

```
authentication_policy = AuthTktAuthenticationPolicy('somesecret', hashalg='sha512')
authorization_policy = ACLAuthorizationPolicy()
config = Configurator(settings=settings,
                      authentication_policy=authentication_policy,
                      authorization_policy=authorization_policy
                      )
```

"somesecret" passed to policy will be a secret string used for cookie signing, so our auth cookie is secure.

The last thing we need to add is to assign our context factory to our routes, we want this to be the route responsible for entry creation/updates:

```
config.add_route('blog_action', '/blog/{action}',
                 factory='pyramid_blogr.security.BlogRecordFactory')
```

Now the finishing touch, we set "create" and "edit" permissions on our views.

For this we need to change our view_config decorators like this:

```
@view_config(route_name='blog_action', match_param='action=create',
             renderer='pyramid_blogr:templates/edit_blog.mako',
             permission='create')
             ...

@view_config(route_name='blog_action', match_param='action=edit',
             renderer='pyramid_blogr:templates/edit_blog.mako',
             permission='edit')
             ...
```

Now if you try to visit the links to create/update entries you will see that they actually respond with 403 HTTP status because pyramid detects that there there is no user object that has *edit* or *create* permissions.

Our views are secured!

Next 8. Authentication

1.9 8. Authentication

Great, we secured our views, but now no one can add new entries to our application, so the finishing touch is to implement our authentication views.

First we need to add a login form to our existing **index.mako** template:

```
<%inherit file="pyramid_blogr:templates/layout.mako"/>

...

% if request.authenticated_userid:
    Welcome <strong>${request.authenticated_userid}</strong> ::
    <a href="${request.route_url('auth',action='out')}">Sign Out</a>
%else:
    <form action="${request.route_url('auth',action='in')}" method="post" class="form-inline">
        <div class="form-group">
            <label>User</label> <input type="text" name="username" class="form-control">
        </div>
        <div class="form-group">
        <label>Password</label> <input type="password" name="password" class="form-control">
        <input type="submit" value="Sign in" class="btn btn-default">
        </div>
    </form>
%endif

% if paginator.items:
    ...
```

Now the template first check if we are logged in, if we are it greets the user, and presents sign-out link. Otherwise we are presented with sign-in form.

Now it's time to update our views and User model.

Lets update our model with two methods, “verify_password” to check user input with password associated with user instance and “by_name” that will fetch our user from database based on login.

We add following method to our User class in models.py:

```
def verify_password(self, password):
    return self.password == password
```

We also need to create UserService class in models/services/user.py:

```
from ..meta import DBSession
from ..user import User

class UserService(object):

    @classmethod
    def by_name(cls, name):
        return DBSession.query(User).filter(User.name == name).first()
```

Warning: In a real application verify_password should be using some strong way one-way hashing algorithm like bcrypt or pbkdf2. Use a package like **cryptacular** to provide strong hashing.

The final step is to update the view that handles authentication.

First we need to add following import to views/default.py:

```
from pyramid.httpexceptions import HTTPFound
from pyramid.security import remember, forget
from ..models.services.user import UserService
```

Those functions will return headers used to set our AuthTkt cookie (from AuthTktAuthenticationPolicy) for users browser, “remember” is used to set the current user, “forget” is used to sign out our users.

Now we have everything ready to implement our actual view:

```
@view_config(route_name='auth', match_param='action=in', renderer='string',
             request_method='POST')
@view_config(route_name='auth', match_param='action=out', renderer='string')
def sign_in_out(request):
    username = request.POST.get('username')
    if username:
        user = UserService.by_name(username)
        if user and user.verify_password(request.POST.get('password')):
            headers = remember(request, user.name)
        else:
            headers = forget(request)
    else:
        headers = forget(request)
    return HTTPFound(location=request.route_url('home'),
                     headers=headers)
```

This is a very simple view that checks if database row with name supplied by user is present is in database, if it is a password check is performed. If password check was successful a new set of headers used to set the cookie is generated and passed back to the client on redirect. If user is not found or password doesnt match a set of headers meant to remove the cookie (if any) is issued.

Voilà!!!

Congratulations, this tutorial is now complete, you can now sign in and out to add/edit blog entries using login *admin* with password *admin* (this user was added to database during *initialize_db step*).

Now is the time to go back to documentation to read on the details of functions/packages used in this example. I've barely scratched the surface of what is possible with Pyramid.

Next 9. Registration

1.10 9. Registration

Now we have a basic functioning application, but we have only one user hardcoded administrator that can add blogs. We can provide a registration page for our users to sign in to.

Then we need to to provide a quality hashing solution so we can store secure password hashes instead of cleartextm this functionality will be provided by **passlib**.

We should create a form to handle registration requests, lets open *forms.py* and add a new form class:

```
# add missing PasswordField at the top of the file

class RegistrationForm(Form):
    username = StringField('Username', [validators.Length(min=1, max=255)],
                           filters=[strip_filter])
    password = PasswordField('Password', [validators.Length(min=3)])
```

Our second step will be adding a new route that handles user registration in our main *__init__.py* file:

```
...
config.add_route('auth', '/sign/{action}')
config.add_route('register', '/register')
config.scan()
...
```

We should add link to the registration page in our *index.mako* template so we can easly navigate to it.

% if request.authenticated_userid: Welcome ${request.authenticated_userid} :: Sign Out

%else:

<form action="${request.route_url('auth',action='in')}" method="post" class="form-inline"> ...

</form> Register here

%endif

So at this point we have the form object and routing set up, we are missing related view, model and template code. Let us move forward with the view code in *views/default.py*.

First we need to import our form definition user model at the top of the file:

```
...
from ..forms import RegistrationForm
from ..models.meta import DBSession
from ..models.user import User
...
```

And we can start implementing our view logic:

```
@view_config(route_name='register', renderer='pyramid_blogr:templates/register.mako')
def register(request):
    form = RegistrationForm(request.POST)
    if request.method == 'POST' and form.validate():
        new_user = User()
        new_user.username = form.username.data
```

```
        new_user.password = form.password.data
        DBSession.add(new_user)
        return HTTPFound(location=request.route_url('home'))
    return {'form': form}
```

Next, let us start by creating a new registration template called *register.mako* with following contents:

```
<%inherit file="pyramid_blogr:templates/layout.mako"/>

<h1>Register</h1>

<form action="${request.route_url('register')}" method="post" class="form">

    % for error in form.username.errors:
        <div class="error">${ error }</div>
    % endfor

    <div class="form-group">
        <label for="title">${form.username.label}</label>
        ${form.username(class_='form-control')}
    </div>

    % for error in form.password.errors:
        <div class="error">${error}</div>
    % endfor

    <div class="form-group">
        <label for="body">${form.password.label}</label>
        ${form.password(class_='form-control')}
    </div>
    <div class="form-group">
        <label></label>
        <button type="submit" class="btn btn-default">Submit</button>
    </div>


</form>
<p><a href="${request.route_url('home')}">Go Back</a></p>
```

Our users can now register themselves and are stored within database using unencrypted passwords (which is a really bad idea).

This is exactly where **passlib** comes into play, so we should add it to our projects requirements in *setup.py*:

```
requires = [
    ...
    'paginate==0.5', # pagination helpers
    'paginate_sqlalchemy==0.2.0',
    'passlib'
]
```

Now we can run *pip install passlib* or run *python setup.py develop* to pull in new dependency to our project - password hashing will be implemented in our *User* model class.

We need to import the hash context object from passlib and alter *User* class to contain new versions of methods *verify_password* and *set_password*, our file should look like this:

```
from passlib.apps import custom_app_context as blogger_pwd_context

class User(Base):
```

```
    __tablename__ = 'users'

    ...

    def verify_password(self, password):
        return blogger_pwd_context.verify(password, self.password)

    def set_password(self, password):
        password_hash = blogger_pwd_context.encrypt(password)
        self.password = password_hash
```

The last step is to alter our *views/default.py* to set password like this:

```
...
new_user.name = form.username.data
new_user.set_password(form.password.data.encode('utf8'))
DBSession.add(new_user)
...
```

Now our passwords are properly hashed and can be securely stored.

If you tried to log in with *admin/admin* credentials you may notice that the application threw exception *ValueError: hash could not be identified* because our old clear text passwords are not identified, so we should allow our application to migrate to secure hashes (usually strong sha512_crypt if we are using the quickstart class).

We can easly fix this by altering our *verify_password* method:

```
def verify_password(self, password):
    # is it cleartext?
    if password == self.password:
        self.set_password(password)

    return blogger_pwd_context.verify(password, self.password)
```

Keep in mind that for proper migration of valid hash schemes passlib provides mechanism you can use to quickly upgrade from one scheme to another.

The complete source code for this application is available at:

https://github.com/Pylons/pyramid_blogr

CHAPTER 2

Indices and tables

- genindex
- modindex
- search