molly Documentation

Release 1.4.1

University of Oxford

Sep 27, 2017

Contents

The Molly Project is a framework written in Django and Python for developing a location-aware mobile portal with particular emphasis on Higher Education.

It is released under the *Academic Public License*, and is available from mollyproject.org.

CHAPTER 1

Getting Started

Installing Molly

Preparing your system

Warning: CentOS 5 users should be aware that some of these dependencies are not in the default repositories. These dependencies live in the the EPEL and RPMforge third-party repositories, which you may need to install to satisfy these dependencies.

In order to install Molly, there are several non-Python dependencies which are needed for Molly and its dependencies first.

Fedora

```
su -c 'yum install python-pip libxml-devel libxslt-devel python-devel postgresql-
↪devel openldap-devel openssl-devel gdal-python proj geos libjpeg-devel imagemagick␣
↪gcc make git libyaml'
```

CentOS

```
su -c 'yum install python-pip python26 python-setuptools python26-devel binutils␣
↪libxslt-devel cyrus-sasl-devel openldap-devel ImageMagick proj proj-devel␣
↪postgresql-devel postgresql-contrib geos-3.1.0 geos-devel-3.1.0 gdal libjpeg-devel␣
↪make gcc openssl-devel libyaml-devel'
```

Ubuntu/Debian

Note: Different versions of Ubuntu and Debian may include different versions of libgeos, libgdal and postgresql which changes the package name. The version below are for Ubuntu 10.04.

```
sudo apt-get install python-pip build-essential python-gdal proj libgeos-3.1.0␣
↪binutils libgdal1-1.6.0 postgresql-server-dev-8.4 python-setuptools python-dev␣
↪libxslt-dev libldap2-dev libsasl2-dev libjpeg-dev imagemagick libyaml
```

Note: Versions below are for Ubuntu 10.11

```
sudo apt-get install python-pip build-essential python-gdal proj libgeos-3.2.2␣
↪binutils libgdal1-1.7.0 postgresql-server-dev-8.4 python-setuptools python-dev␣
↪libxslt1-dev libldap2-dev libsasl2-dev libjpeg62-dev imagemagick python-yaml
```

If your system isn't one of those listed above, then you will need to ensure the following packages, or their equivalent on your platform, are available:

- libxml-devel
- libxslt-devel
- python-devel
- postgresql-devel
- openldap-devel
- openssl-devel
- gdal-python
- proj
- geos
- libjpeg-devel
- imagemagick
- gcc
- make

Installing Molly

Note: When installing using pip or Git it is possible to install Molly in an isolated environment called a virtualenv, where Molly and its dependencies can be installed seperately from the system-wide Python packages. To do this, you simply need to install the virtualenv tool (`pip install virtualenv`) and then run `virtualenv molly`, followed by `source molly/bin/activate`. Once the virtualenv is activated, then you can install Molly using the directions below. Please note that Molly will only appear installed when the virtualenv is activated by running the activate script as indicated above.

Warning: Please note that the Python 2.6 binary on CentOS 5 is called 'python26'. On CentOS, it is therefore recommended that you work in a virtualenv as detailed above. The virtualenv can be set to use Python 2.6 by creating it as `virtualenv --python=/usr/bin/python26 molly`. Once the virtualenv is activated, `python` refers to version 2.6.

Using pip or easy_install

Note: On Fedora, pip is called 'pip-python' to avoid a clash with Perl's pip

You can install Molly directly from the Python Package Index (PyPI) using the pip or easy_install commands:

```
pip install molly
```

From Git (latest development version)

To install the bleeding edge version of Molly, you can obtain Molly from our Git repository at git://github.com/mollyproject/mollyproject.git:

```
git clone git://github.com/mollyproject/mollyproject.git
```

This will create a clone (local copy) of the full Molly source code repository in the folder called 'mollyproject' in the directory where you ran the command. To install Molly, you can now use the setup.py script inside the newly created 'mollyproject' directory:

```
python setup.py install
```

If you intend on doing development work with Molly, you may prefer to use the development install facility which will allow you to work from your Molly repository without having to re-install after every code change:

```
python setup.py develop
```

Setting up your database

Warning: This is only intended as a quick introduction to configuring Postgres and may not be suitable for production environments.

Molly recommends the use of the PostgreSQL database with the PostGIS extensions as a database backend.

Installing Postgres

On Ubuntu and Debian:

```
sudo apt-get install postgresql-8.4 postgresql-8.4-postgis
```

On Fedora and CentOS:

```
su -c 'yum install postgresql postgresql-server postgresql-devel postgresql-contrib␣
↪postgis'
```

You may then need to start your new Postgres database, this can be done with:

```
sudo service postgresql start
```

Once Postgres is created you must create a template database which can then be used to create the Molly database. This can be done by following the Creating a Spatial Database Template for PostGIS section in the Geodjango documentation.

Creating your database user

> **Warning:** This section assumes a default distribution install of Postgres, if you have changed the default security settings of Postgres, you may need to specify the `-U username -W password` options on the `createuser` command to authenticate as a superuser.

In default installs of Postgres, database usernames must match local usernames in order to authenticate, therefore, the user created on the database should match the username of the user that Molly runs as.

On most default installs, the 'postgres' user is a superuser on the database, and the `createuser` command should be run as this superuser:

```
sudo -u postgres createuser
```

Please enter your local username when creating this user, and the user does not need to be a superuser or able to create new users (roles) or databases.

If you have configured Postgres to require password authentication, the `-P` option should be specified:

```
sudo -u postgres createuser -P
```

In this case, the username may not necessarily be the same as the local user.

Creating your database for Molly

> **Warning:** This section assumes a default distribution install of Postgres, if you have changed the default security settings of Postgres, you may need to specify the `-U username -W password` options on the `createdb` and `psql` commands to authenticate as a superuser.

As with the `createuser` command, the 'postgres' user is considered a superuser and commands should be run as this user. To create the database, assuming the PostGIS template was installed as `template_postgis` you can use:

```
sudo -u postgres createdb -T template_postgis <database>
```

Replacing <database> with your database name. More information on postgis templates can be found in the PostGIS documentation.

And then you can give your user access to this new database:

```
sudo -u postgres psql -c "GRANT ALL ON DATABASE <database> TO <username>;"
```

Replacing <database> and <username> with your database and username respectively.

Creating a site template

Molly has the distinction between the core of Molly, and a Molly site. The core of Molly is the upstream Python package and associated data, whereas a site is a Django project which contains your settings for Molly, as well as any media, templates and other customisations to the Molly core. In order to get started with Molly, you will need to create a site.

The `molly-admin createsite` command will create a template site which you can then go and customise to your exact requirements:

This argument takes one argument which specifies the path to create the template in:

```
molly-admin createsite /PATH/TO/MY/SITE
```

Warning: Don't call your site *site*. It will mess up Djangos with settings finder.

Once your site template has been created, the following files are created which are only templates and require you to edit them:

- `settings.py` - following the configuration guide;
- `apache/molly.wsgi` - if you are deploying Molly as a WSGI app, then you will need to change the DJANGO_SETTINGS_MODULE setting in this file;
- `apache/httpd.conf` - this is a sample Apache config file;
- `templates/base.html` - this is a sample template override - for more information about this, please see the customising guide.

You will also have a `compiled_media` folder, which should be ignored, and a `site_media` folder, which is where you should put any overrides for media on your site.

Running Celery

New in version 1.4.

Molly now runs it's periodic tasks (e.g. importing map data from OpenStreetMap) by using the popular Distributed Task Queue, Celery.

Celery requires us to install a *message broker*, the most popular choice here is RabbitMQ. There are other brokers available, as always we recommend reviewing the excellent Celery documentation to learn more.

Molly's installation will have setup the celery worker, `celeryd` and task scheduler `celerybeat` for us. We just have to start them:

```
python manage.py celeryd
python manage.py celerybeat
```

See also:

Daemonizing Celery – Celery doesn't daemonize itself so you'll want to either run it as an init script or deploy it with some process management like supervisor.

Deploying Molly

Once you have configured your site appropriately, you're almost ready to deploy your site!

This can be done using the command:

```
python manage.py deploy
```

You now have an install of Molly ready to serve to the world. The recommended way of doing this is by using Apache and mod_wsgi. The site template created by the installer consists of a WSGI script and a sample Apache config in the apache/ directory of your site.

The mod_wsgi documentation goes into considerable detail about how to deploy a Django application.

Starting a development server for Molly

This can be done by adding the `--develop` command to the deploy command above:

```
python manage.py deploy --develop
```

Note: Starting a development server also skips the updating Wurfl step, in order to speed up development

Updating Molly

To update Molly, you simply need to rerun the ‘‘./setup.py install‘ command in the new Molly folder. This will recognise that the installation already exists and will update as appropriate.

Configuring Molly

Understanding the Config

Note: This covers advanced topics and are not needed to get a simple Molly install going. We'd recommend coming back to this later, if you need to.

The settings file itself is a standard Django settings file, which is documented in the settings overview in the Django documentation, with the addition of some extra settings which configure Molly itself.

Some of the Django settings must be configured in particular ways for Molly to operate as desired. These are listed below.

Preamble

At the top of the Molly settings files we import some required definitions for use later on in the file:

```
from oauth.oauth import OAuthSignatureMethod_PLAINTEXT
import os.path, imp
from molly.conf.settings import Application, extract_installed_apps, Authentication,␣
↪ExtraBase, Provider
from molly.utils.media import get_compress_groups
```

We also define two variables which are used throughout, one to refer to the location Molly is installed, and the second to refer to the location at which the site lives:

```
molly_root = imp.find_module('molly')[1]
project_root = os.path.normpath(os.path.dirname(__file__))
```

Required Django settings

The following settings are all required by the Molly project, but can be configured freely for Molly to operate normally

- ADMINS by default the people defined here will receive logging e-mails and cronjob output
- ADMIN_MEDIA_PREFIX
- DATABASES note that the database engine should be set to an engine in `django.contrib.gis`; PostGIS is recommended: `django.contrib.gis.db.backends.postgis`
- DEBUG
- LANGUAGE_CODE
- LANGUAGES - this is the list of languages which Molly will display to its users as selectable. If not set, this will default to Django's default, which is probably not what you want.
- LOCALE_PATHS - this is where Molly will find any translation files you have prepared
- MANAGERS Molly gives this setting no special meaning, so it is recommended you set this to the same as ADMINS
- SITE_ID unused in most situations, so leave at 1
- STATIC_ROOT this is the path to where on disk media for your site is served from. It should be an empty directory, which is populated during the build process.
- STATIC_URL this is the URL to the location where your media is served from (note that Django does not serve media in non-development mode, but relies on your web server to do it, for more information see *Deploying Molly*).
- TEMPLATE_DEBUG
- TIME_ZONE

Required Settings for Molly

The following settings are all standard Django settings, but must be configured in a particular way for Molly to operate correctly:

CSRF_FAILURE_VIEW

To render a nice page when a CSRF validation failure occurs, Molly ships with a default page for these circumstances. Django must be told to use this page:

```
CSRF_FAILURE_VIEW = 'molly.utils.views.CSRFFailureView'
```

INSTALLED_APPS

This must be defined after the *APPLICATIONS* setting. This setting is used to inform Django which Molly apps are loaded, as well as any non-Molly applications that are being used. Molly provides one Django application and has dependencies on other Django applications which must be included. INSTALLED_APPS must therefore be at least:

```
INSTALLED_APPS = extract_installed_apps(APPLICATIONS) + (
    'django.contrib.auth',
    'django.contrib.admin',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.gis',
    'django.contrib.comments',
    'molly.batch_processing',
    'django.contrib.staticfiles',
    'pipeline',
    'south',
)
```

With any additional non-Molly apps being added to the bottom of the list.

Logging configuration should be done the same way it is done for every *Django project <https://docs.djangoproject.com/en/dev/topics/logging/>*.

MIDDLEWARE_CLASSES

The setup of middleware can vary between particular installations, however for getting started, the default value below will suffice. More advanced users should refer to the Django reference

At the very least, this setting must include `molly.wurfl.middleware.WurflMiddleware` as the first value and then at any point in the list `molly.auth.middleware.SecureSessionMiddleware` and `molly.url_shortener.middleware.URLShortenerMiddleware`, as well as Django's default middleware.

In order to enable geolocation functionality in Molly, you must enable `molly.utils.middleware.LocationMiddleware`.

A typical setup may look like this:

```
MIDDLEWARE_CLASSES = (
    'molly.wurfl.middleware.WurflMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'molly.utils.middleware.CookieLocaleMiddleware',
    'molly.utils.middleware.ErrorHandlingMiddleware',
    'molly.utils.middleware.LocationMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'molly.auth.middleware.SecureSessionMiddleware',
```

```
    'molly.url_shortener.middleware.URLShortenerMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
)
```

ROOT_URLCONF

Molly will automatically create a urlconf based on your loaded applications, however, you can override this to a custom one if you wish. To use the default urlconf that comes with Molly set this setting like so:

```
ROOT_URLCONF = 'molly.urls'
```

STATICFILES_DIRS

This is a list of locations where media is collected from to be served. Assuming that you have a folder called `site_media` which contains your custom media (this is created in the default layout by the installer), and you wish to fall through to Molly's media if required, then the following setting should suffice:

```
STATICFILES_DIRS = (
    ('', os.path.join(project_root, 'site_media')),
    ('', os.path.join(molly_root, 'media')),
    ('markers', MARKER_DIR),
)
```

Note that the final lines (markers) is required for slippy maps to correctly display markers.

TEMPLATE_CONTEXT_PROCESSORS

Like MIDDLEWARE_CLASSES, this can vary between installations, but the Django reference is a good starting point for more advanced users. To get started, the typical value below should suffice.

If configuring this directly, then it is recommended you use the `molly.utils.context_processors.ssl_media` context processor instead of Django's `django.core.context_processors.media`, especially if you're serving media from a separate server (for more information, read this blog post). The following context processors are required for correct operation of Molly in most settings:

- `molly.wurfl.context_processors.wurfl_device`
- `molly.wurfl.context_processors.device_specific_media`
- `molly.geolocation.context_processors.geolocation`
- `molly.utils.context_processors.full_path`

If you wish to use Google Analytics, `molly.utils.context_processors.google_analytics` is useful.

A typical setup looks like this:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.request',
    'molly.utils.context_processors.languages',
    'molly.utils.context_processors.ssl_media',
    'django.contrib.messages.context_processors.messages',
    'molly.wurfl.context_processors.wurfl_device',
```

```
    'molly.wurfl.context_processors.device_specific_media',
    'molly.geolocation.context_processors.geolocation',
    'molly.utils.context_processors.full_path',
    'molly.utils.context_processors.google_analytics',
    'molly.utils.context_processors.site_name',
    'django.core.context_processors.csrf',
)
```

TEMPLATE_DIRS

These are the directories Django looks in for templates, in order of searching. At it's most minimal, this needs to contain the path to Molly's templates folder, but in most cases will also need to include the path to the folder where templates specific to your deployment (the ones that override the default templates) are being held. Using the shorthand paths set in the *preamble*, and assuming that your templates are stored in a `templates/` folder in your deployment, the following is typical:

```
TEMPLATE_DIRS = (
    os.path.join(project_root, 'templates'),
    os.path.join(molly_root, 'templates'),
)
```

TEMPLATE_LOADERS

This sets how Django looks for templates when rendering a view. This must include Molly's custom template loader, and in almost all circumstances should be set to the following value:

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.ldjanoaders.app_directories.load_template_source',
    'django.template.loaders.eggs.load_template_source',
    'molly.utils.template_loaders.MollyDefaultLoader'
)
```

USE_18N

This settings enables Django's i18n support. As Molly has full i18n support, it is recommended you set this to True:

```
USE_I18N = True
```

django-compress Settings

In order to deliver minified CSS and JavaScript, Molly uses a fork of django-compress, which must be configured appropriately.

The following settings will make this "just work" with Molly:

```
PIPELINE_CSS, PIPELINE_JS = get_compress_groups(STATIC_ROOT)
PIPELINE_AUTO = False
PIPELINE_VERSION = True
PIPELINE_CSS_COMPRESSOR = 'molly.utils.compress.MollyCSSFilter'
PIPELINE_JS_COMPRESSOR = 'pipeline.compressors.jsmin.JSMinCompressor'
```

Celery settings

New in version 1.4.

We include a few sane-defaults for running Celery. These are:

```
BROKER_URL = "amqp://molly:molly@localhost:5672//"
CELERYBEAT_SCHEDULER = "djcelery.schedulers.DatabaseScheduler"
CELERYD_CONCURRENCY = 1
CELERY_RETRY_DELAY = 3 * 60
CELERY_MAX_RETRIES = 3
```

The only setting you should worry about here is the *BROKER_URL*. This setting is passed from Celery to the transport layer library Kombu, which has excellent documentation for the possible broker options and their limitations. The default we provide will attempt to use RabbitMQ with vhost *molly*, on *localhost*, connecting as user *molly*.

Remaining options are best explained in the Celery documentation.

Molly settings

The following settings are additional to the Django configuration and configure Molly directly.

APPLICATIONS

This setting defines your Molly applications and how they are configured. It should consists of a list of Molly Application objects, e.g.,:

```
APPLICATIONS = [
    Application('molly.apps.app1', 'app1', 'App 1',
        ... application config 1 ...
    ),

    Application('molly.apps.app2', 'app2', 'App 2',
        ... application config 2 ...
    ),

    Application('molly.apps.app3', 'app3', 'App 3',
        ... application config 3 ...
    ),
]
```

Details about configuration of each individual application are on the page for that application, and more abstract information about the application framework can be found at *The Application framework*.

Applications bundled with Molly are split into two categories: utility, and batteries-included. Utility apps must always be loaded, whereas batteries-included apps represent optional user-facing functionality. However, some batteries-included apps rely on others, so these must also be loaded.

The apps that are utility apps are:

- *molly.auth – Auth services*
- ../ref/batch-processing
- *molly.external_media – External Media*

- *molly.favourites – Favourites*
- *molly.geolocation – Geolocation*
- *molly.maps – Map Generation*
- *molly.url_shortener – URL shortening*
- *molly.utils – Molly Utilities*
- *molly.wurfl – Device Detection*

The following battery-included apps are:

- *molly.apps.contact – Contact search*
- *molly.apps.desktop – Desktop*
- *molly.apps.feeds.events – Event Listing*
- *molly.apps.feature_vote – Feature Suggestions*
- *molly.apps.feedback – Feedback*
- *molly.apps.feeds – Feed Manager* (required by news, events and webcams)
- *molly.apps.home – Home Screen* (required)
- *molly.apps.library – Library Search*
- *molly.apps.feeds.news – News*
- *molly.apps.places – Places* (required by library and transport)
- *molly.apps.podcasts – Podcasts*
- *molly.apps.sakai – Sakai integration*
- *molly.apps.search – Whole site search*
- *molly.apps.service_status – Service status*
- *molly.apps.tours – Tours*
- *molly.apps.transport – Transport dashboard*
- *molly.apps.weather – Weather*
- *molly.apps.webcams – Webcams*

API_KEYS

This is a dictionary holding various API keys for your deployment. There is no default for these, are you will need to get your own keys. The following two keys are used: Cloudmade for geolocation and Google Analytics, if Google Analytics is enabled. You can get a Cloudmade key from your user profile page on Cloudmade.com, and Google Analytics from the Analytics dashboard.

Sample:

```
API_KEYS = {
    'cloudmade': 'MyCloudmadeKey',
    'google_analytics': 'MyGoogleAnalyticsKey',
}
```

CACHE_DIR

CACHE_DIR should be set to a path where Molly can cache files (this includes generated map tiles, resized images, etc), with no trailing slash.

Sample:

```
CACHE_DIR = '/var/cache/molly' # This must be set, and there is no default
```

DEBUG_SECURE

Whether or not secure parts of the site are in debug mode (this means less rigorous checking of secure sessions and whether or https is required to access parts of the site marked as secure).

Sample:

```
DEBUG_SECURE = DEBUG # There is no default, but this would set it to the same value␣
↪as the global DEBUG setting
```

DISTANCE_UNITS

This setting determines how distances are transformed into human-readable units. The valid settings here are 'imperial', which uses yards and miles, 'metric' which uses metres and kilometres, and 'british', which uses metres and miles. This defaults to 'british'.

Sample:

```
DISTANCE_UNITS = 'british'
```

EXTERNAL_IMAGES_DIR

Where cached external images are stored, by default this is under the cache directory, and is optional.

Sample:

```
EXTERNAL_IMAGES_DIR = '/var/cache/molly-images'
```

IDENTIFIER_SCHEME_PREFERENCE

Each entity has a primary identifier which is used to generate the absolute URL of the entity page. We can define a list of identifier preferences, so that when an entity is imported, these identifier namespaces are looked at in order until a value in that namespace is chosen. This is then used as the primary identifer. The default is shown in the sample below, and is optional:

```
IDENTIFIER_SCHEME_PREFERENCE = ('atco', 'osm', 'naptan', 'postcode', 'bbc-tpeg')
```

MARKER_DIR

Where markers are stored, by default this is under the cache directory. This is optional, but it may be useful to define it, as it needs to be referenced in the STATICFILES_DIRS setting:

```
MARKER_DIR = os.path.join(CACHE_DIR, 'markers')
```

NO_CACHE

When set to true, then cache headers are suppressed from responses. This can be used as a workaround to a MOLLY-177.

SITE_NAME

The name of the service, extensively used in templates to name the service. This defaults to 'Molly Project':

```
SITE_NAME = 'Mobile Oxford'
```

Deploying Molly

Now you've got your Molly install configured and ready to deploy, there are two steps you can follow. The first is to do a local testing deploy using Django's built in web server, and the second is to configure Apache to serve the site.

For development and demonstrations, the local testing install is sufficient, however the Django webserver should not be considered as reliable or secure for anything other than local testing.

Local Testing Installs

There are two ways to start up the Django development server. If you do a development install of Molly (pass the `-d` flag to `setup.py deploy`), then the development server will automatically start at the end of the install.

The alternative way is to start the server by hand. There are two steps to this, the first is to activate the virtualenv, and the second to start the server:

```
source /path/to/install/bin/activate
cd /path/to/install/deploy
python manage.py runserver
```

Configuring Apache

When a new site is created by the Molly installer, than a sample WSGI and Apache configuration file is generated, which can be used to get Molly up and running. If you did not create your site using the Molly installer, then sample files are included below.

WSGI is an interface between web applications like Molly and a webserver, so you must install the mod_wsgi Apache module and then configure mod_wsgi in your Apache configuration to serve your site.

> **Warning:** Molly by default uses a virtualenv to manage dependencies. mod_wsgi must therefore be set up to use this virtualenv. This can be accomplished by adding `WSGIPythonHome /path/to/install/` to your `httpd.conf` or mod_wsgi configuration file.

You must also set up Apache to serve your site media, as compiled in the directory specified by `STATIC_ROOT` (this is the `media` folder in a default install) at the URL specified by the `STATIC_URL` setting.

Sample Apache virtualhost config:

```
<VirtualHost *:80>
    # Change the following settings as appropriate
    ServerName m.example.ac.uk
    ServerAdmin molly@example.ac.uk

    WSGIDaemonProcess molly user=molly group=molly
    WSGIProcessGroup molly
    WSGIScriptAlias / /path/to/my/site/apache/molly.wsgi

    ErrorLog /var/log/apache2/error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog /var/log/apache2/access.log combined

    Alias /media /path/to/my/site/media
    # It is recommended you create these two files and then set up the links
    # as appropriate
    #Alias /robots.txt /path/to/robots.txt
    #Alias /favicon.ico /path/to/favicon.ico

</VirtualHost>
```

Sample WSGI script:

```
#!/usr/bin/env python

import os, os.path
import sys

os.environ['DJANGO_SETTINGS_MODULE'] = 'deploy.settings'
sys.path.insert(0, os.path.abspath(os.path.join(
os.path.dirname(__file__), '..', '..',
)))

import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Customising Molly

The Molly framework contains powerful functionality to allow for customisations of the HTML, CSS and images to be displayed to users. Below, we cover the ways you can use to customise the look and feel of your Molly install.

Changing Images and CSS

The first, and simplest, way to customise Molly is simply to replace whole images in the final result. This can be images, icons, or anything else you want.

Note: This assumes you are using the standard layout generated by the installer. If you are using a non-standard layout, then the folder which is used must be defined in `STATICFILE_DIRS` before the Molly media directory.

This is simply done by placing any images you want to replace in the 'site_media' folder which have exactly the same path and name as the image in the original Molly media.

Note: You can also replace CSS and JavaScript in the same way, although at present you must replace the entire file, rather than individual rules.

There are multiple ways to find the path of the image you want to replace. Firstly, you should check the documentation of the app which should list all of the media provided by that app, and its location.

Warning: At the time of writing, documentation is incomplete and not all apps contain a complete list of media.

Other ways of finding media include looking at the URL of the image, and then removing the URL component, e.g., if your STATIC_URL is set to '/compiled_media/' and the URL of the image is '/media/weather/images/cloudy5.png', then the file to place into the site_media folder is 'weather/images/cloudy5.png'. Alternatively, looking in 'molly/media/' or 'molly/apps/APP/static/' will also find the media associated with a particular app.

Note: Any changes to images won't take effect until the media folder is rebuilt, the quickest way to do this is to re-run the installer.

Customising Templates

As Molly is built on the Django framework, it should come as no surprise that Molly uses Django's templating framework. For a thorough introduction to Django templating, Django's own documentation is the best introduction, however a brief overview is presented below.

When Molly prepares a page to be rendered, it first executes a view, which builds a "context" - a Python dictionary consisting of variables which are available in the view for rendering. It then passes this dictionary to a particular template file for rendering. For the filename of a particular template file which is rendered on a particular view is listed in the documentation for that app.

Warning: At the time of writing, documentation is incomplete and not all apps contain a complete list of templates, views and context variables.

Syntax of a Template

The syntax of a template file consists of HTML interspersed with variables and template tags. Variables are code like `{{ bus_stop }}`, which inserts the value of the variable in the context called `bus_stop`. The `.` seperator can be used to access members of variables, e.g., `{{ entity.title }}` will print the title of an entity object.

Note: The list of variables available in the context is shown in the documentation for that app.

Template variables can also be put through filters, this is done with the | character, and this modifies a variable in a way defined by the filter. e.g., ‘‘ {{ entity.title|upper }}‘‘ results in the title of a passed in entity being displayed in upper case.

Note: Some non-core template tags must first be loaded before they can be used in a template. This is done by placing `{% load LIBRARY %}` at the top of the file, where LIBRARY is the name of the library containing that template tag or filter. The Molly utilities section of the documentation contains a list of all template tag libraries available with Molly.

Also available are template tags, which are like function calls. These use syntax such as: `{% for entity in entities %} ... {% endfor %}`. This code loops over a list in the context (called entities) and exposes one entity at a time in a variable called `entity` which the code inside the for loop can use.

Also available are if statements and code to lookup URLs.

See also:

The Django reference documentation contains a full list of template tags and filters.

Blocks

Templates support inheritance, which means that a template can “inherit” from another template, taking its styling and other content from another template, and then replacing part of the other template with the content you specifically want to render. This is accomplished in Molly by providing a base template, and then having individual templates replace the body of this base template with the actual content to be displayed to the user.

This is indicated in code by `{% extends "base.html" %}`, which indicates that this page “extends” base.html. The blocks then defined in that particular template override the content that would have been shown in that place in base.html, resulting in a consistent template for the user.

Molly extends this further by allowing even smaller parts of a template to be replaced with other content, which allows you to only alter part of a template without having to replace the entire template that comes bundled with Molly.

Replacing Whole Templates

Note: This assumes you are using the standard layout generated by the installer. If you are using a non-standard layout, then the folder which is used must be defined in `TEMPLATE_DIRS` before the Molly templates directory.

If in your site you would like to replace an entire page with your custom template, this is done by creating a file in the `templates/` directory in your site with the same name as the template being overriden. This file is then rendered instead of the file that comes with Molly.

Overriding Parts of Templates

If you wish to only override a part of a template that comes with Molly, rather than the entire template, this is also supported, using the ‘Blocks’ mechanism outlined above. In this case, you can keep almost all of the original template code in Molly, benefitting from any updates or changes that affect it, and then customise only the small part of it you want to do.

See also:

The applications reference pages have a list of blocks that are extendable for each view.

To do this, you must first create a template in the same way as for a whole template, but instead of replacing the entire content, you can simply extend the original template, which is available with the path `molly_default/app/template.html`, and then add a new block for each part of the template you wish to change, rather than the entire template.

base.html - an example

This section works through an example of styling Molly's base template. It may be beneficial to have the Molly development server running (see the *install guide* for details on how to do this) so you can see your changes in real time.

base.html is the very root of the entire template tree, it defines the basic styling which is on every page of the site (apart from the desktop app). Most sites will want to start by styling this.

To start, you must create a file called `base.html` in the `templates/` directory of your site. If you create this as an empty file, you should now start seeing a blank page on your site, which means that your new base template is being rendered. As the base template does not contain anything yet, the blank page should be expected!

Note: The front page is not a good example, as it makes multiple changes to the base template. It would be best to use a sub-page, such as the weather page, to view the changes.

The next step is to make the new base template inherit from the one that comes included in Molly. We can do this by simply adding one line to the new base.html

```
{% extends "molly_default/base.html" %}
```

This should now result in a page which looks exactly like the one that comes with Molly. The key difference now is that all requests are going through our new base template, so we can start adding and changing things to do what we would like to do.

Two things you may want to change to get started are the page name as it appears in the <title> of the document and in the navigation options for non-smart phones. There are two blocks to do this, called `site_title` and `breadcrumb_0`. We can now override these by adding the following two lines to `base.html`.

```
{% block site_title %}m.ox | {% endblock %}
{% block breadcrumb_0 %}m.ox{% endblock %}
```

This will change the start of the <title> for each page to be "m.ox" followed by a pipe, and then the first navigation breadcrumb for non-smart phones to be "m.ox".

Note: If you're looking to just change the logo, this can be done using the methods for replacing images above - simply place your logo with the same name in the `site_media` folder.

With these simple changes and suitable logo replacements, your Molly instance should now be ready to go!

Referring to other views

To get a link to another view, this can be generated by the Django url tag using the namespace prefix of the app and the name of the view (as documented on that application's page)

E.g., for the 'index' view of the contact app, the following template tag will generate a URL:

```
{% url contact:index %}
```

Writing an application

A Molly application is just a standard Django application, however, the Molly framework provides a few extra features to make applications within Molly consistent, and handling mobile browsers easier.

- *Class-based views*, which provide a powerful framework for rendering which targets different devices
- A consistent *framework for styling*, which provides a consistent look and feel across the site.
- A *framework for searching*, which applications can tie in to.

Note: For a more in-depth look at these features of the Molly framework, please see the documentation linked above, this is a simple overview of how to get started writing a Molly app.

Anatomy of a Molly app

See also:

A Molly app is also a Django app. The Django tutorial is a good introduction to writing Django apps, and it may be beneficial to familiarise yourself with Django concepts before

On disk, a Molly app may look similar to this:

```
myapp/
 migrations/
|   [...]
 providers/
|    __init__.py
|    myprovider.py
 static/
|   myapp/
|      css/
|     |   smart.css
|      js/
|     |   smart.js
|      images/
|          icon.png
|          [...]
 templates/
|    myapp/
|       base.html
|       index.html
|       [...]
 templatetags/
|   __init__.py
|   molly_myapp.py
 __init__.py
 admin.py
 forms.py
 models.py
 search.py
 tests.py
```

```
urls.py
views.py
```

Note: Not all of these files may exist or are necessary in all apps.

Let's break down the content of this folder. `migrations` are used to store migrations which are managed by South. The `providers` folder, unsurprisingly, contains the providers that come bundled with the application. `__init__.py` normally contains the base provider (i.e., an abstract class which demonstrates the signature expected of providers for that class), and then any other subfiles contain the concrete providers, following the signature of the base provider.

The `static` and `templates` folders each have a single folder underneath them, with the same name as the application which it provides. This is due to the way collating templates and media works, so adding an additional level avoids clashes with other applications during the collation process. In this folder are the media and templates for the app which are used for rendering. The media is exposed to the world under the `STATIC_URL` defined in the configuration, and can be referenced in your templates. For the apps that ship with Molly, we have followed a standard of having 3 subdirectories here: `js`, `css`, and `images`.

Note: JavaScript and CSS is automatically minified during the build process when installation is done in non-development mode.

The files `css/smart.css`, `css/dumb.css` and `js/smart.js` in the media folders have special meanings, and are included automatically on pages (when using the standard base template). `smart.css` is served to "smart" phones, `dumb.css` to "dumb" phones and `smart.js` to "smart" phones which Molly considers to have an acceptable level of JavaScript support.

Note: Technically `js/dumb.js` also has a special meaning, but "dumb" phones do not get served JavaScript, so the script will never be included by default.

`templatetags` is a standard implementation of Django's template tags, which Molly gives no special meaning to. Molly applications themselves have standardised on a prefix of `molly_` to the template tags tag name to prevent clashes with any external apps being used.

`__init__.py` typically provides utility functions within the application, and `admin.py` provides the functionality (if any) for this application in the Django admin view. Similarly, `forms.py` is where any Django forms live, `models.py` where the Django models are and `tests.py` where any unit tests for this application stay. This is the same layout as for a typical Django app.

`views.py` is typically where any views for this application are stored, however, unlike in typical Django apps, these views follow the Molly view format, which is documented below. Similarly, `urls.py` is a standard Django URL dispatcher, however in most cases an actual reference to the class, rather than a string, should be used, e.g.:

```
from django.conf.urls.defaults import *

from views import IndexView, FooView, BarView

urlpatterns = patterns('',
    (r'^$', IndexView, {}, 'index'),
    (r'^foo/(?P<arg>.+)$', FooView, {}, 'foo'),
    (r'^bar/$', BarView, {}, 'bar'),
)
```

The first argument in each pattern is a regular expression. Any match groups in the regular expression are then passed

to the methods of the view as arguments. Molly exclusively uses named match groups (which are passed as keyword arguments) to accomplish this.

`search.py` is a file which has special meaning within Molly. If there is a class called `ApplicationSearch` in here, then this is used within the site-wide search framework.

See also:

molly.apps.search – Whole site search

Anatomy of a view

See also:

Explaining Molly's class-based views

Molly provides a powerful framework for writing class-based views by providing a class called `BaseView`. Writing a view generally consists of extending this class and then providing content for a few methods.

Available on each view is also an attribute `conf`, which contains the configuration of the application which this view belongs to. This contains all the configuration arguments specified in the configuration file, as well as:

- `application_name` - the name of the application;
- `local_name` - the local name (i.e., first part of the URL) as configured for this application;
- `title` - the human-readable title of this application;
- `urls` - the Django urlconf for this application;
- `has_urlconf` - whether or not the urlconf is set.

initial_context

When a view is called, then the `initial_context` method is called, along with the request object, as well as any arguments defined in the URL pattern. This function then sets up the context which is used for rendering.

Note: If this class inherits from `ZoomableView` or `FavouritableView`, then you should call the `super` function in `initial_context` in order to correctly setup the context.

This is done by populating a dictionary with various keys and values depending on what needs to be rendered, and then returning this dictionary, e.g.:

```
from molly.utils.views import BaseView

class FooView(BaseView):

    def initial_context(self, request):

        context = {}
        context['rain'] = 'Mainly on the plain'

        return context

    ...
```

When this method is called, then any match groups defined in the URL pattern are also presented alongside it, e.g., if the match group was: `^/(?P<id>\d+)/$`, then this is how the `initial_context` could be written:

```
from molly.utils.views import BaseView
from models import Foo

class FooView(BaseView):

    def initial_context(self, request, id):

        context = {}
        context['thing'] = Foo.objects.get(pk=id)

        return context

    ...
```

Also of note, is the ability to raise `django.http.Http404 <http://docs.djangoproject.com/en/dev/topics/http/views/#the-http404-exception>`_, which will cause the view to render as a 404 error.

handle_*

You will have to write a `handle_*` method for each HTTP method you wish to support. In most cases, this will just be GET, and sometimes POST, although you can support some of the rarer requests if you would like (HEAD by default is handled by rendering it as a GET and then stripping the content).

For whatever method you write, the method is called along with the request object, the context as set up by `initial_context`, as well as any arguments defined in the match group.

This function is expected to return a HttpResponse object, which can be done by calling 2 shortcut functions: `render` or `redirect` which are defined in `BaseView`.

These can be utilised like so:

```
from molly.utils.views import BaseView

class FooView(BaseView):

    def handle_GET(self, request, context):

        return self.render(request, context, 'myapp/template')

    def handle_POST(self, request, context):

        # Handle a form response, which is available in the request.POST
        # dictionary

        return self.redirect(context['uri'], request)

    ...
```

As with `initial_context`, raising `django.http.Http404 <http://docs.djangoproject.com/en/dev/topics/http/views/#the-http404-exception>`_, will cause the view to render as a 404 error.

Breadcrumbs

Molly uses a "breadcrumb trail" approach to navigation across the site. In the default template, this is represented at the top of the screen. In order to generate a breadcrumb trail, each view defines a `breadcrumb` method, which returns a function which evaluates to a tuple providing the following members:

- the name of the application;
- the index view of the application;
- the parent of this page;
- whether or not the parent is the index;
- the title of this page.

In order to simplify this, you can simply return a `Breadcrumb` object from your method, and then decorate it using the `BreadcrumbFactory` decorator.

A Breadcrumb object consists of the name of the application, the URL to the parent page, the title of this particular page, and the URL of this particular page. A typical example may look like:

```
from molly.utils.views import BaseView
from molly.utils.breadcrumbs import Breadcrumb, BreadcrumbFactory
from django.core.urlresolvers import reverse

class FooView(BaseView):

    @BreadcrumbFactory
    def breadcrumb(self, request, context, ...):
        return Breadcrumb(
            'myapp',
            reverse('myapp:index'),
            context['title'],
            reverse('myapp:foo'),
        )

    ...
```

Note: If the view is the top-level page in the app, the second argument should be `None`.

This assumes that `initial_context` adds a `title` key to the context. This could be static text, or some other method of deducing the name of this page. Also, if the pattern for matching this page includes any optional arguments, then these are passed as additional arguments at the end of the method.

Metadata

In some circumstances, you will want to get information about a view, without actually rendering it. This is done, for example, when rendering a search result or displaying information about search results. To provide information for these uses, then views can define a `get_metadata` function, which returns a dictionary with the keys `title`, containing the title of the page being rendered, and an optional `additional` line, which contains additional information about the view:

```
from molly.utils.views import BaseView
from molly.utils.breadcrumbs import Breadcrumb, BreadcrumbFactory
from django.core.urlresolvers import reverse

class FooView(BaseView):

    def get_metadata(self, request):
        return {
            'title': 'Foo Checker',
            'additional': 'Check on the current status of foo',
```

```
        }

    ...
```

Also, if the pattern for matching this page includes any optional arguments, then these are passed as additional arguments at the end of the function.

ZoomableView

If you are rendering maps, and want the ability to make static maps zoomable, then you can instead inherit from `molly.utils.views.ZoomableView`, which will add the ability to zoom in and out of static maps.

> **Warning:** If the device supports slippy maps, then all maps will be zoomable.

To use this, you must also set up the context in `initial_context` using a `super()` call. The context will then contain a key called `zoom` which can be passed to the `Map` constructor to build a map at the correct zoom level.

If you would like to specify a default zoom level, you can do this by adding an attribute to your class called *default_zoom*, e.g.,:

```
from molly.utils.views import ZoomableView

class FooView(ZoomableView):

    default_zoom = 16

    def initial_context(self, request):
        context = super(FooView, self).initial_context(request)
        ...
        return context

    ...
```

FavouritableView

If you would like to make it so that a view can be marked as a *favourite*, then `molly.favourites.views.FavouritableView` is available as a base class, which when used as a base adds values to the context which are used to add the ability to add/remove favourites on those rendered pages:

```
from molly.favourites.views import FavouritableView

class BarView(FavouritableView):

    def initial_context(self, request):
        context = super(BarView, self).initial_context(request)
        ...
        return context

    ...
```

SecureView

Another view available is `molly.auth.views.SecureView`. When extending this view, then all requests to your view must be made over a HTTPS connection (unless `DEBUG_SECURE` is true).

Your First App

Note: This tutorial was first given as a workshop at Dev8D 2011. The code from this workshop has been made available, and as of version 1.1 has been incorporated into the transport app.

Now we've covered the basics of a Molly view and the structure of an app, we can start building our first app. In this worked example, we will build an application to display the status of mass transit systems (particularly the London Underground and Glasgow Subway).

Django provides a simple method to start an app, which should be sufficient as the first step in making any new app for Molly. It doesn't really matter where the code is stored, but it should be on your Python path. In most cases, a good place to put it is in your site (the 'deploy' directory by default).

To get started, we can use the `django-admin` function in your deploy directory to create the framework for your app:

```
django-admin.py startapp transit_status
```

The last argument here is the name of the folder (and subsequentally the app) to be created. Inside this folder, we see the structure of an app as described above, although with a few files missing. From here, we're ready to start, so let's put together a view which does nothing.

The blank view

Creating a simple view in Molly is quite simple, you just need to extend `BaseView` and then provide at least one `handle_*` method - typically `handle_GET` for most pages, and `handle_POST` if you need to deal with forms, and a breadcrumb method.

Open up views.py in your favourite text editor, and then add the following:

```
from molly.utils.views import BaseView
from molly.utils.breadcrumbs import Breadcrumb, BreadcrumbFactory, lazy_reverse

class IndexView(BaseView):

    @BreadcrumbFactory
    def breadcrumb(self, request, context):
        return Breadcrumb('transit_status', None, self.conf.title,
                          lazy_reverse('index'))

    def handle_GET(self, request, context):
        return self.render(request, context, 'transit_status/index')
```

Here, we have two methods: `handle_GET`, which simply renders the template 'transit_status/index' `breadcrumb` which returns the details for the breadcrumb navigation included in the default template.

The breadcrumb method here uses the standard way of generating breadcrumbs in Molly:

- the first argument to the `Breadcrumb` constructor is the name of the app;

- the second is the URL for the parent of this page - in this case there is no parent, as this is the root of the app, so this is None;
- the third is the title of this page - here we're using self.conf.title attribute, which means that the name of the application is also the name of this page. In many pages, this will not necessarily be the case, so the title could be determined from the context, or as a static string;
- the fourth is the URL of this page, the `lazy_reverse` function returns the URL for the `index` page in this app (the `index` page is defined in the URL conf as described below).

As our `handle_GET` method is rendering a template, we will now need to write a template to do this. The most minimal thing we can do here is to create new folders in your application folder called `templates/transit_status`, and then create a blank file called `index.html`. We can add some content to this file later.

The final step to produce a minimal view is to create a urlconf to requests to the view. Urlconf's are standard Django fare, and a fairly standard one could be created which looks like:

```
from django.conf.urls.defaults import *

from views import IndexView

urlpatterns = patterns('',
    (r'^$', IndexView, {}, 'index'),
)
```

With all that done, we now need to add the new app to your `settings.py`, and start up the development server to see our blank page in action.

See also:

Configuring Molly

To do this, at the end of the `APPLICATIONS` list in `settings.py`, an `Application` definition needs to be added. In this case, the following will suffice:

```
Application('transit_status', 'transit_status', 'Tube Status'),
```

Now, start up a development server and browse to your development instance (typically `http://localhost:8000`). There should be a blank icon on the home screen at the end with your new application below it. Clicking on that should take you to a blank page.

Note: The *installation guide* contains information on how to install Molly in development mode, or to start a development server.

Note: Not seeing what you expect? Ask the Molly community, who will be able to help you.

Fleshing it out

Now we have a basic view working, we can start fleshing out our views and templates. One thing that needs adding to the templates is a `get_metadata` method, which allows for pages to appear in search results, as well any further future use, such as favouriting pages. In most cases, this simply needs to be something which returns the title of the current page, as well as any additional information about what the page does. On this view, we can simply add:

```
def get_metadata(self, request):
    return {
        'title': self.conf.title,
        'additional': 'View the status of transit lines'
    }
```

The next step is to add something to our template to make it a bit more than a blank screen, this can be done by adding:

```
{% extends "base.html" %}
```

to the template, which renders the base style of the site, with any additional content to be displayed.

Note: Most Molly apps actually extend `app_name/base.html`, which in turn extend `base.html`. This structure allows for entire apps to be styled consistently to each other, but different to the core styling, if so desired.

At this point, we need to decide on the format of the context to be presented to the template, as well as the format of the data provided by providers.

Note: Molly separates apps into "views" and "providers". Providers should provide abstract interfaces to services which views can call to get the details about the configured services. Views should therefore be service agnostic.

Most applications supply a `BaseProvider` which provides a signature for concrete providers to follow. For this transit line app, a provider which implements a single `get_status` method should suffice. This method is then responsible for querying the service, and then returning the information. For our users, this information is in the form of a list of Python dictionaries, where each Python dictionary provides the name of the line (`line_name`), the current status (`status`) and an optional reason for disruption (`disruption_reason`).

With this decided, we can now define the context. Here, we can simply pass the results from the provider into the context. As the title of the app is configurable (e.g., a London university may set it to 'Tube status', a Mancunian one to 'Metrolink status', etc), we also want this in the context.

Once the context is defined, we can set up the template to render this. To display content when extending the base template, you have to define a block called content, and place your template code in there. For our template, with the context structure defined above, utilising Molly's default CSS structure, we can edit `templates/transit_status/index.html` to look like so

```
{% extends "base.html" %}

{% block content %}

    <div class="section">
        <div class="header">
            <h2>{{ title }}</h2>
        </div>
        <table class="content">
            <thead>
                <tr>
                    <th>Line</th>
                    <th>Status</th>
                </tr>
            </thead>
            <tbody>
                {% for status in statuses %}
                <tr>
                    <td>{{ status.line_name }}</td>
```

```
                    <td>{{ status.status }}
                    {% if status.disruption_reason %}<br/>{{ status.disruption_reason␣
↪}}{% endif %}
                    </td>
                </tr>
                {% endfor %}
            </tbody>
        </table>
    </div>

{% endblock %}
```

See also:

Customising Molly

We now need to set up the context to actually provide this information to the template. We can do this by adding an `initial_context` method to `IndexView` which returns our context dictionary. In the context, we need to provide two things:

- the title of the page, from `self.conf.title`;
- the status of the lines, by calling the provider.

Our `initial_context` method should therefore look like this:

```
def initial_context(self, request):
    return {
        'title': self.conf.title,
        'statuses': self.conf.provider.get_status()
    }
```

At this point, we also need to write our base provider, and also alter the configuration of the application to include a `provider` attribute.

To create a base provider, then the following can be included in a new file, `providers/__init__.py`:

```
class BaseTransitLineStatusProvider(object):

    def get_status(self):
        # Return a list of dictionaries, where the dictionaries have keys
        # of "line_name", "status" and optional "disruption_reason"
        return []
```

See also:

The Application framework

We can now alter our `settings.py` application configuration to point to this provider, and our app should now render the page as expected (with no line statuses showing quite yet). To do this, an argument to the `Application` contructor called `provider` should be added, which is itself is a `Provider`, constructed with the classpath of the provider. i.e.:

```
Application('transit_status', 'transit_status', 'Tube Status',
    provider=Provider('transit_status.providers.BaseTransitLineStatusProvider')),
```

The finishing touches

Now we have the basis of an app actually working, that's all the Molly specific stuff over. All that remains is for us to add an actual provider. In a new file, `providers/tfl.py`, the following can be pasted:

```
import urllib
from xml.dom import minidom

from transit_status.providers import BaseTransitLineStatusProvider

class TubeStatusProvider(BaseTransitLineStatusProvider):

    LINESTATUS_URL = 'http://cloud.tfl.gov.uk/TrackerNet/LineStatus'

    def get_status(self):

        statuses = []

        status_xml = minidom.parse(urllib.urlopen(self.LINESTATUS_URL))

        for node in status_xml.documentElement.childNodes:
            if node.nodeType == node.ELEMENT_NODE and node.tagName == 'LineStatus':
                line_status = {
                    'disruption_reason': node.getAttribute('StatusDetails'),
                }
                for child in node.childNodes:
                    if child.nodeType == child.ELEMENT_NODE and child.tagName == 'Line
↪':
                        line_status['line_name'] = child.getAttribute('Name')
                    elif child.nodeType == child.ELEMENT_NODE and child.tagName ==
↪'Status':
                        line_status['status'] = child.getAttribute('Description')
                statuses.append(line_status)

        return statuses
```

Then, the provider in the application configuration can be changed as below to use this new provider:

```
Application('transit_status', 'transit_status', 'Tube Status',
    provider=Provider('transit_status.providers.tfl.TubeStatusProvider')),
```

We now have a complete application for displaying the status of the London Underground lines!

With this split of views and providers, it makes it very simple to adjust an app for use by others, in other contexts. The following provider, if placed in `providers/spt.py`, would allow access for status of the Glaswegian subway:

```
from transit_status.providers import BaseTransitLineStatusProvider
from lxml import etree
import urllib2

class SubwayStatusProvider(BaseTransitLineStatusProvider):

    JOURNEYCHECK_URL = 'http://www.spt.co.uk/journeycheck/index.aspx'

    def get_status(self):
        statuses = []
        xml = etree.parse(urllib2.urlopen(self.JOURNEYCHECK_URL), parser = etree.
↪HTMLParser())
        ul = xml.find(".//ul[@id='jc']")
```

```
        for li in ul:
            statuses.append({
                'line_name': ''.join(li.itertext()).strip(),
                'status': li.find(".//img").attrib['alt']
            })
        return statuses
```

And if the configuration of the app was changed as below, this app is now also suitable for a Glaswegian university:

```
Application('transit_status', 'transit_status', 'Subway Status',
    provider=Provider('transit_status.providers.spt.SubwayStatusProvider')),
```

Of course, this is a very simplistic application, it doesn't utilise the database, only has one view and doesn't deal with forms, but those features are part of Django, which is well-documented, rather than particular to the Molly framework.

CHAPTER 2

Developing with Molly

Molly as an API

Molly's approach to views lends itself to a large amount of flexibility. One such flexibility is the ability to vary the format of the rendered view based on the Accept headers of the client, or a manual override of the data format based on an additional GET parameter: `format`.

To disable this functionality on a particular view (e.g., if the context contains data that should not be exposed), you can set `exposes_user_data` to True in the context thus implying that the data is private and should not be exposed in the API.

The data formats supported are:

- XML
- JSON/JSONP
- YAML

And in each of these, a serialised form of the context is presented.

The JSON API also provides a field called `view_name` which can be used to uniquely identify the format of the response. This is discussed further below.

An additional format, 'fragment' is also available. This provides the body and title HTML of a template encoded using JSON. It's used for the AJAX functionality introduced in Molly 1.1.

Adding serialisation to your apps

Molly will attempt to serialise many standard Python and Django objects, however in some cases, this is too simplistic for many uses. Classes can therefore define a `simplify_for_render` method which can be used to manually specify how the object can be simplified.

View names and page names

A view name is a unique identifier for the page which rendered the view. It should uniquely identify the format of the response. It is in the format of `application:view_id`. The details of what are available for each view

Page names uniquely identify a particular instance of a page. Molly can handle multiple instantiations of an app, so this refers to the specific instance of the app. This is best illustrated with an example.

Let's assume that we have two different phonebooks, one for people and another for departments. We may want to instantiate this as two different instances of the contact search app with the configuration below:

```
Application('molly.apps.contact', 'people', 'People search',
    provider = 'example.providers.contact.PeopleSearchProvider',
),

Application('molly.apps.contact', 'departments', 'Department Phonebook',
    provider = 'example.providers.contact.DepartmentSearch',
),
```

The page at http://example.com/people/ will therefore have a page name of `people:index` and the page at http://example.com/contact/ will have a page name of `departments:index`, however both pages when rendered will be rendered by the view in the Molly app `contact` so will have a view name of `contact:index`.

Page names are therefore used in breadcrumbs and on the home page apps listing as these refer to particular instances of apps, and these should be passed to the reverse API in order to get the URL of those particular instances. View name only exists as a convenience to help API users determine how to handle the response.

The reverse API

Django includes functionality to "reverse" a URL, that is, take a page or view name and arguments to it and then return the URL on that Molly instance. This endpoint exists at the `/reverse/` URL at the highest level of the instance (e.g., `http://example.com/reverse/`) and takes two arguments: `name` and `arg`, which can be specified multiple times. These correspond to the Django URL resolver definition in the app's `urls.py`.

See also:

Django's documentation on reversing URLs

This page returns a simple string containing the URL. It is recommended that you cache these somewhere to avoid lookups.

For example, to resolve the URL for Doncaster rail station, we can simply call the URL `http://example.com/reverse/?name=places:entity&arg=crs&arg=DON` which will return the fully qualified URL for that page.

This functionality allows for applications to be written against Molly as a generic API, rather than one tied to the (URL configuration) of a particular Molly instance.

Application media

Note: This is information for people developing applications and wanting to use Molly's media framework. If you're just interested in customising the look and feel of Molly, see the *customising tutorial*.

Molly uses django-staticfiles and django-compress to collate and merge media files from various sources.

Source media locations

A set of base media are pulled in from `molly/media/`. Each molly application may also define a `media/` subdirectory, from which Molly will pull media related to that particular application. Finally, the site may provide additional media to override the defaults in `<project_root>/site_media/`.

In each of these locations media are organised within a directory bearing the name of the application that uses them. For example:

```
molly/apps/contact/
  media/
    contact/
      css/
        [...]
      js/
        [...]
      images/
        [...]

molly/
  media/
    base/
      css/
        [...]
      [...]

your_site/
  site_media/
    contact/
      images/
        override_some_icon.png
    base/
      css/
        [Add your own style here]
```

Collating media

Media are collated into `settings.STATIC_ROOT`, which is defined to be the same as `settings.MEDIA_ROOT`. To invoke `django-staticfiles` to collate the media:

```
$ python manange.py build_static -l --noinput
```

The `-l` option causes it to use symlinks instead of copying files. This has the added advantage that when editing your media during development you don't have to invoke `build_static` after every change.

Note: You will however need to invoke `build_static` if you add new media files, so as to create the new symlinks from `STATIC_ROOT`.

Compressing media

Molly encourages the use of `django-compress` to merge and tidy up CSS and JavaScript files. Media are merged into a directory `c/` within `STATIC_ROOT`, with the first fragment of the path removed.

`django-compress` also adds a timestamp to each compressed file's filename.

Thus, `contact/css/index.css` and `base/css/index.css` would be merged into `c/css/index.r12345678.css`.

To reference the combined media group from a template you should load the `compressed` template library using `{% load compressed %}` at the beginning of the template file. You may then use `{% compressed_js <group-name> %}` and `{% compressed_css <group-name> %}` to generate the required `<script/>` and `<link rel="stylesheet"\>` tags.

The group name is the hyphenated path within the `c/` directory without any `js-` or `css-` prefix. Thus `contact/css/types/clever.css` would have a group of `types-clever`.

Note: For more information about `django-compress`, please see their documentation.

`django-compress` will automatically update the generated files if it discovers that a constituent file has changed on disk. However, if you want to sync them manually, it can be invoked as follows:

```
$ python manage.py synccompress
```

To only enable compression when not in `DEBUG` mode, use the following line in your settings file:

```
COMPRESS = not DEBUG
```

Using compressed media in your templates

The Application framework

Molly extends Django by formalising the concept of an application. This is acheived by instantiating *`Application`* objects and adding then to `APPLICATIONS` in your `settings` module.

This document explains how the Application framework works, and should not be a necessary part of your reading unless you intend to write a new Molly application, or have sufficient levels of curiosity.

If you don't know whether you want to write an application or a provider, the following section may be useful. For information on writing providers, see topics/writing_providers.

Difference between applications and providers

Molly is intended to be relatively pluggable, allowing the deploying institution to take whatever provided functionality they choose and, if necessary, integrate it with their own applications.

This integration can be done at two levels, the application level, or the provider level.

An application is a Python package, usually containing `urls`, `models` and `views` modules. An application provides a data model and interface for a particular class of information, for example, PC availability or lecture timetables.

A provider is usually a single Python class that connects an application to a data source. A provider might implement some well-known protocol or file format (e.g. RSS for the feeds application), or might connect to a local bespoke system.

It is intended that in the majority of cases the implementor should be able to take an already-existing application and need only write the provider that performs the interaction with other systems.

Overview of the Application framework

An *Application* object is a wrapper around a Python package which hooks in providers and configuration objects for easy access by the application. The definition is as follows:

class molly.conf.settings.**Application**

__init__(*application_name*, *local_name*, *title*, ***kwargs*)

Instantiates an Application object. None of the module or package paths are dereferenced yet. kwargs are mostly left alone and attached to the *~molly.conf.settings.ApplicationConf* class. Some, defined later, have special meanings.

Parameters

- **application_name** (*str*) – a Python package path to the application to be used, e.g. 'molly.apps.places'.
- **local_name** (*str*) – a local unique identifier for this instance of the *Application*. In most cases this will be identical to the last part of application_name. This will be used in almost all cases where an application needs to be referenced. It is also used by the default urlconf creator to determine the URL prefix this site is served under.
- **title** (*unicode or str*) – A descriptive title for the application instance to be shown to the user.

get()

Used internally. Creates a configuation object and dereferences all the paths provided to it. Where a urls module exists it will call *add_conf_to_pattern()* to walk the urlconf to attach the configuration object to the views.

The ApplicationConf returned will have attributes reflecting the kwargs passed to *__init__()*. The urlconf will be exposed as a urls attribute.

Return type ApplicationConf subclass

add_conf_to_pattern(*pattern*, *conf*, *bases*)

Used internally. Maps the conf and bases onto the views contained in pattern. This method creates a new view, with conf in the class dictionary and the view and bases as base classes. Specifically:

```
new_callback = type(callback.__name__ + 'WithConf',
                    (callback,) + bases,
                    { 'conf': conf })
```

This dynamically creates a new class object. For more information, see the second definition of type().

When given a RegexURLPattern it will return a new RegexURLPattern with its callback replaced as above. When given a RegexURLResolver it will descend recursively before returning a new RegexURLResolver instance.

Returns A copy of the first argument with views replaced as described above.

Return type RegexURLResolver or RegexURLPattern instance

In the vast majority of cases, you will only need to use the constructor, and only in your settings module.

There are a few keyword arguments with special meanings:

providers An iterable of Provider instances to load providers from.

provider (or any keyword ending with provider) A shorthand for providers = [`provider`].

display_to_user A `bool` used by *molly.apps.home* to determine whether a link should be rendered for the application on the home page.

extra_bases An iterable of `ExtraBase` instances, defining extra base classes to add to all views in the application. With suitably defined extra base classes one can override functionality. Application-level authentication may also be added in this manner.

secure A `bool` which if `True` will add `SecureView` as a base class of all views in the application. `SecureView` forces all requests to be made over HTTPS, and provides a `secure_session` attribute on `HttpRequest` objects.

urlconf A module path to the `urls` module to use for this application. May be useful if an application uses a non-standard naming, or if you want to override the application-provided urlconf. If not provided, defaults to `application_name + '.urls'`

to_email This is optional, and defaults to the admins setting, and refers to the default target for e-mails generated by this app.

from_email This is optional, and sets the e-mail address e-mails generated by this app appears from

Here's an example:

```
APPLICATIONS = [
    # ...
    Application('example.apps.dictionary', 'dictionary', 'Dictionary',
        provider = Provider('isihac.providers.apps.dictionary.uxbridge'),
        max_results = 10,
    ),
    # ...
]
```

Here we want to use a dictionary application with at most ten results from the Uxbridge English Dictionary. If we wanted to expose two different dictionaries we may wish to do the following:

```
APPLICATIONS = [
    # ...
    Application('example.apps.dictionary', 'uxbridge_dictionary', 'Uxbridge English␣
↪Dictionary',
        provider = Provider('isihac.providers.apps.dictionary.uxbridge'),
        max_results = 10,
    ),
    Application('example.apps.dictionary', 'oxford_dictionary', 'Oxford English␣
↪Dictionary',
        provider = Provider('oxford.providers.apps.dictionary.oed'),
        max_results = 20,
    ),
    # ...
]
```

Once hooked into the root urlconf, this would present two links on the home page. Alternatively, if the `example.apps.dictionary` application supported multiple providers, we could do this:

```
APPLICATIONS = [
    # ...
    Application('example.apps.dictionary', 'dictionary', 'Dictionaries',
        providers = (
            Provider('isihac.providers.apps.dictionary.uxbridge',
                slug='uxbridge',
                title='Uxbridge English Dictionary'),
            Provider('oxford.providers.apps.dictionary.oed',
```

```
                slug='oed',
                title='Oxford English Dictionary'),
        ),
        max_results = 10,
    ),
    # ...
]
```

Of course, this assumes that the application knows to pick the `slug` and `title` from each of its providers. To determine the interface between applications and providers, consult the application's documentation.

Providers

A provider maps an external interface onto the model used by the application.

Most applications provide a `providers.BaseProvider` class which specifies an interface to be implemented by a provider for that application.

New in version 1.4: Providers now all subclass `molly.conf.provider.Provider`

Task Processing

New in version 1.4.

Celery is used to provide asynchronous task processing. For an introduction to the basics of Celery we recommend you take a look at the "Getting Started with Celery" guide.

Molly uses a modified version of the Celery task decorator located in `molly.conf.provider.task` this should be used in a similar the previous @batch decorator to identify any methods on a provider to run async via celery.

See this (simplified) example from `molly.apps.feeds.providers.rss`:

```
@task(run_every=timedelta(minutes=60))
def import_data(self, **metadata):
    """
    Pulls RSS feeds
    """

    from molly.apps.feeds.models import Feed
    for feed in Feed.objects.filter(provider=self.class_path):
        logger.debug("Importing: %s - %s" % (feed.title, feed.rss_url))
        self.import_feed.delay(feed)
    return metadata

# Override CELERY_RETRY_DELAY and CELERY_MAX_RETRIES
@task(default_retry_delay=5, max_retries=2)
def import_feed(self, feed):
    from molly.apps.feeds.models import Item
    feed_data = feedparser.parse(feed.rss_url)
    # Do stuff with feed_data
```

We can iterate through all feeds and launch tasks to import them asynchronously using `task.delay()`. This convention has been applied through all the standard providers packaged with Molly. Note the default_retry_delay and max_retries are overridden on import_feed. This means each feed will only be retried 2 times, with 5 seconds between each of those retries.

Explaining Molly's class-based views

Note: This has been lifted verbatim from a blog post and still needs to be tidied up to fit the documentation.

When an HTTP request is handled by a Django website, it attempts to match the local part of the URL against a series of regular expressions. Upon finding a match it passes an object representing the request as an argument to a callable associated with the regular expression. In Python, most callables you will find are class or instance methods, or functions. The Django documentation only briefly refers to the fact that one can use callables other than functions.

The flow of a request

Django despatches an incoming request to the callback given in a urlconf which, in Molly, is a class object. Calling a class object is mapped to calling it's `__new__` method. Ordinarily, this returns an instance of that class, but in Molly it returns the response. Specifically:

class `FooView`

`__new__(request, *args, **kwargs)`:

Parameters **`request`** (`HttpRequest`) – The request from the client to be processed.

Return type `HttpResponse`

Unless overridden, the method called is `molly.utils.views.BaseView.__new__()`. This performs the following steps:

- Checks there is a handler available for the HTTP method specified. If not, it immediately returns a `405 Method Not Acceptable` response.
- Calls `cls.initial_context(request, *args, **kwargs)`, which can provide context for all method handlers. A developer can use this to factor out code common between each of the handlers.
- Evaluates the breadcrumbs and adds the resulting information to the context.
- Calls the relevent handler, the name of which is determined by appending the method name to `handle_`, e.g. `handle_GET`. The handler is passed the request, context and any positional and keyword arguments.
- The handler will update the context and perform any required actions as necessary.
- The handler will generally return a `HttpResponse` subclass directly, or call the `render()` method on `BaseView`.
- In the latter case, `render()` will determine which format to serialize to using a `format` query parameter or content negotiation, and despatch to a format-specific rendering method (e.g. `render_html()`).
- The renderer will return a `HttpResponse` object, which will then be passed back up the callstack to Django to be sent back to the client.

Class-based views and metaclasses

We're using class-based views, a concept that doesn't seem to have much of a presence on the Internet. The usual approach is to define a method `__call__(self, request, ...)` on a class, an instance of which is then placed in an urlconf. Our approach is the following:

- Have a base view called, oddly enough, BaseView.

- Define a method __new__(cls, request, ...) that despatches to other class methods depending on the HTTP method.
- The __new__ method also calls a method to add common context for each of the handlers.
- We use a metaclass to save having to put @classmethod decorators in front of every method.
- We never create instances of the view classes; instead, __new__ returns an HttpResponse object and the class itself is place in the urlconf.

Here's the code:

```
from inspect import isfunction
from django.template import RequestContext

class ViewMetaclass(type):
     def __new__(cls, name, bases, dict):
         # Wrap all functions but __new__ in a classmethod before
         # constructing the class
         for key, value in dict.items():
             if isfunction(value) and key != '__new__':
                 dict[key] = classmethod(value)
         return type.__new__(cls, name, bases, dict)

class BaseView(object):
    __metaclass__ = ViewMetaclass

    def method_not_acceptable(cls, request):
        """
        Returns a simple 405 response.
        """

        response = HttpResponse(
            'You can't perform a %s request against this resource.' %
                request.method.upper(),
            status=405,
        )
        return response

        # We could go on defining error status handlers, but there's
        # little need. These can also be overridden in subclasses if
        # necessary.

        def initial_context(cls, request, *args, **kwargs):
            """
            Returns common context for each of the HTTP method
            handlers. You will probably want to override this in
            subclasses.
            """

            return {}

        def __new__(cls, request, *args, **kwargs):
            """
            Takes a request and arguments from the URL despatcher,
            returning an HttpResponse object.
            """

            method_name = 'handle_%s' % request.method
            if hasattr(cls, method_name):
```

```
                # Construct the initial context to pass to the HTTP
                # handler
                context = RequestContext(request)
                context.update(cls.initial_context(request,
                                                   *args, **kwargs))

                # getattr returns a staticmethod , which we pass the
                # request and initial context
                handler_method = getattr(cls, method_name)
                return handler_method(request, context,
                                      *args, **kwargs)
            else:
                # Our view doesn't want to handle this method; return
                # a 405
                return cls.method_not_acceptable(request)
```

Our actual view code can then look a little something like this (minus all the faff with input validation and authentication):

```
class CheeseView(BaseView):
    def initial_context(cls, request, slug):
        return {
            'cheese': get_object_or_404(Cheese, slug=slug)
        }

    def handle_GET(cls, request, context, slug):
        return render_to_response('cheese_detail.html', context)

    def handle_DELETE(cls, request, context, slug):
        context['cheese'].delete()
        # Return a 204 No Content response to acknowledge the cheese
        # has gone.
        return HttpResponse('', status=204)

    def handle_POST(cls, request, context, slug):
        # Allow a user to change the smelliness of the cheese
        context['cheese'].smelliness = request.POST['smelliness']
        context['cheese'].save()
        return HttpResponse('', status=204)
```

For those who aren't familiar with metaclasses, I'll give a brief description of class creation in Python. First, the class statement executes all the code in the class body, using the newly bound objects (mostly the methods) to populate a dictionary. This dictionary is then passed to the __new__ method on the metaclass, along with the name of the class and its base classes. Unless otherwise specified, the metaclass will be type, but the __metaclass__ attribute is used to override this. The __new__ method can alter the name, base classes and attribute dictionary as it sees fit. In our case we are wrapping the functions in class method constructors so that they do not become instance methods.

Other things we could do are:

- Override handle_DELETE in a subclass to return a 403 Forbidden if the cheese is important (calling super(cls, cls).handle_DELETE if it isn't)
- Despatch to other methods from a handler to keep our code looking modular and tidy
- Subclass __new__ to add more parameters to the handlers on subclasses

As an example of the last point, we have an OAuthView that ensures an access token for a service and adds an urllib2 opener to the parameters which contains the necessary credentials to access a remote resource.

The subclassing view can then simply call opener.open(url) without having to worry about achieving the requisite authorisation.

Using class-based views allows us to define other methods on the views to return metadata about the resource being requested. As an example, we have a method that constructs the content for the breadcrumb trail, and another that returns the metadata for displaying in search results.

Achieving such extensibility with function-based views would be nigh on impossible.

Coding guidelines

Blocks in template should use an underscore to separate words (i.e. double_word should be preferred over doubleword or double-word).

JavaScript with Molly

Molly uses device detection to decide whether or not a device should be given JavaScript or not.

Adding JavaScript to your app

The single most important thing to remember when adding JS functionality to your app is that of graceful degradation. Not all devices that Molly supports support JS - the *Supported Devices* page gives the full breakdown.

Molly comes with JQuery 1.2.6 (Symbian 9.4 struggles on newer versions), so this is available for you to use if you would like.

Putting JavaScript into page headers is not supported, as this will not work when the page is asynchronously loaded. All JavaScript must be put in a file called *static/APP/js/smart.js* (where *APP* is the name of your application package), which will be included automatically on every page of the site, as well as being minified and collated into one large file when served.

As your JS will be loaded on every page, it is important not to do anything that may interfere with other pages. The section below covers this in more detail.

JavaScript with the asynchronous page loading framework

Because of the AJAX added to Molly in 1.1, JavaScript is not quite as straight forward as it used to be. As multiple pages can be loaded and rendered within the same session using AJAX, the document.ready event can not be relied upon to set up any functionality in your page.

To work around this, Molly provides a new event which is called when an AJAX page change is triggered: 'molly-page-change', with one argument, the URL of the new page. You will then probably want to check the url attribute to ensure that it's a page that you should be handling.

```
$(document).bind('molly-page-change', function(event, url){
    if (url == '/my-app/') {
        /* Set up my event handlers here */
    }
});
```

Anything that would be in a document.ready handler or a <script> tag in the head will then need to go on there. This event also get's called on the first page load, so you can put everything that you will ever need to handle in there.

If you do things like set up timers in the event handler, you will also have to remember to clear those timers if the page transitions to another page.

If you add your own AJAX handlers to redraw the page, etc, you will then be responsible for ensuring the listeners are set up on any <a> and <form> elements you create. Fortunately, this is quite simple - just call 'capture_outbound()' at the end of your AJAX callback and the correct event listeners will be created.

To suppress the default action of AJAX page transitions, returning false or similar is not enough to stop the default page transition happening. To do this, you must add an 'has-ajax-handler' class to any links and forms you want your code to handle, before the 'capture_outbound()' function is called. If you add this class whilst inside the 'molly-page-change' event handler, then you don't need to worry any further, as this is automatically called after the event has been triggered.

Catching location changes

It is also possible to catch occasions when the user updates their location, or the location is updated for them automatically by listening to an event in a similar way. You may want to do this to, for example, trigger a refresh of location-specific data.

The event to listen to is on the document object, and is 'molly-location-update'.

```
$(document).bind('molly-location-update', function(){
    /* react to how the location changed here */
});
```

Styling Molly

Supported Devices

The following devices are supported by Molly:

OS/Browser Type	Style type	JavaScript support?
Apple iOS	smart	yes
Symbian 9.2+	smart	On Symbian 9.4+ only
Maemo/Meego	smart	yes
RIM (Blackberry)	smart	yes
Android	smart	yes
Web OS (Palm)	smart	yes
Windows Mobile 6	dumb	no
Windows Phone 7	smart	yes
Kindle	smart	yes
Desktop web browsers	smart	yes
Opera	smart	yes, except on Opera Mini before version 5
Skyfire	dumb	No
Everything Else	dumb	No

Using Molly's Utilities

CHAPTER 3

Reference

Utilities

`molly.auth` – Auth services

This is a utility app which provides authentication services to other applications.

Configuration

- unify_identifiers: Which identifier namespaces are guaranteed to contain unique values for a user (used to log a user back in to an old session when they log in with one service)

Sample:

```
Application('molly.auth', 'auth', 'Authentication',
    display_to_user = False,
    secure = True,
    unify_identifiers = ('oxford:sso', 'oxford:oss', 'weblearn:id', 'oxford_ldap'),
),
```

Using in your app

Extra configuration

Applications which use the authentication services provided by this app have the following additional settings they can set:

- enforce_timeouts (optional, defaults to True): if true, then authentication times out after a period of inactivity
- service_name: The name of the service which requires authentication

- oauth_authorize_interstitial (optional, defaults to True): whether or not an institial page should be shown when requesting OAuth authorisation

Extra bases

Bases are provided so that other applications can have them as a parent class in order to provide authentication.

molly.auth.oauth.views.OAuthView

Provides OAuth authentication to an application. It has the following options:

- secret: The OAuth secret
- signature_method: A signature method instance from the *oauth.oauth* package
- base_url: The URL for the OAuth provider
- request_token_url: The URL (added on to the base) of the request_token service
- access_token_url: The URL (added on to the base) of the access_token service
- authorize_url: The URL (added on to the base) of the authorisation service

Views

molly.external_media – External Media

This is a utility app for handling off-site media, including resizing for mobile devices.

Configuration

This app has no configuration.

Sample:

```
Application('molly.external_media', 'external_media', 'External Media',
    display_to_user = False,
),
```

Views

molly.favourites – Favourites

Configuration

Sample:

Providers

Writing Your Own Providers

Views

molly.geolocation – Geolocation

This is a utility app that provides geolocation functionality and the user experience to other apps and users.

Configuration

- prefer_results_near (optional, defaults to None): If set, it prefers geolocation results near the (latitude, longitude, distance) tuple
- providers: providers which provide geocoding
- history_size (optional, defaults to 5): How many locations should be stored in the history for each user
- location_request_period (optional, defaults to 180): how often to ask for new locations from users which are updating their location automatically, in seconds

Sample:

```
Application('molly.geolocation', 'geolocation', 'Geolocation',
    prefer_results_near = (-1.25821, 51.75216, 5000),
    providers = [
        Provider('molly.geolocation.providers.PlacesGeolocationProvider'),
        Provider('molly.geolocation.providers.CloudmadeGeolocationProvider',
            search_around = (-1.25821, 51.75216),
            search_distance = 5000,
        ),
    ],
    location_request_period = 900,
    display_to_user = False,
),
```

Providers

molly.geolocation.providers.PlacesGeolocationProvider

This is a provider which matches geocoding requests against entities known in the places app and returns the location, if appropriate. It has one, optional, option:

- search_identifiers: A limit on which identifier namespaces are to be searched (list of strings)

molly.geolocation.providers.CloudmadeGeolocationProvider

This provider asks Cloudmade to geocode results. You have to retrieve an API key from Cloudmade to use this provider. It has three optional parameters:

- search_around: (latitude, longitude) tuple, supposed the centre of the area
- search_distance: distance to the centre of the area
- **get_area: make a second call to the API to display the name of the area, for** each search result

Writing Your Own Providers

Views

molly.maps – Map Generation

This is a utility app which provides maps to other applications.

Configuration

This app has no configuration

Sample:

```
Application('molly.maps', 'maps', 'Maps',
    display_to_user = False,
),
```

Views

molly.url_shortener – URL shortening

This is a utility function which allows views to have shortened URLs generated and mapping of those URLs when input to their views.

Configuration

This app has no extra configuration.

Sample:

```
Application('molly.url_shortener', 'url_shortener', 'URL Shortener',
    display_to_user = False,
),
```

Views

molly.utils – Molly Utilities

This is a utility service containing the core of Molly.

Configuration

This app has no extra configuration

Sample:

```
Application('molly.utils', 'utils', 'Molly utility services',
    display_to_user = False,
),
```

Views

molly.utils.middleware – Molly Middleware

A library containing middleware classes for Molly

LocationMiddleware

Configuration

Add `molly.utils.middleware.LocationMiddleware` to your `MIDDLEWARE_CLASSES` setting and you're done.

Usage

LocationMiddleware enables location awareness in Molly. Locations can be provided in a number of ways.

1. **Stored as a session variable** - this is the default for people visiting the site with their browsers. Molly redirects the user to a view that gets their location if they visit a view that requires their location and their current session doesn't already store it.

2. **Provided as an HTTP header** - useful for third party apps accessing Molly via API calls. The format of the HTTP header is as follows:

```
X-Current-Location:  latitude=x.xx,longitude=y.yy[,accuracy=z]
```

For example:

```
X-Current-Location:  latitude=51.40392,longitude=1.0203
```

```
X-Current-Location:  latitude=51.40392,longitude=1.0203,accuracy=5
```

3. **Provided as HTTP query string parameters** - an alternative to HTTP headers where it's easier to use a query string. The format is as follows:

```
latitude=x.xx&longitude=y.yy[&accuracy=z]
```

For example:

```
http://mydomain.com/places/nearby/?latitude=51.40392&longitude=1.0203
```

```
http://mydomain.com/places/nearby/?latitude=51.40392&longitude=1.
0203&accuracy=5
```

Note: When providing location as an HTTP header or as query string parameters, latitude and longitude are mandatory, and will be interpreted using the WSG84 datum. Accuracy is optional.

The order of evaluation is:

- If query string params are provided, use those; else:
- If the X-Current-Location header is set, use that; else:
- If the geolocation:location session variable is set, use that

molly.wurfl – Device Detection

This is a utility app which provides device detection

Configuration

- expose_view: If defined, this exposes a single page which allows users to see what their device is being identified as.

Sample:

```
Application('molly.wurfl', 'device_detection', 'Device detection',
    display_to_user = False,
    expose_view = True,
),
```

Views

Troubleshooting

The WURFL database is the part of the Molly framework that needs the most upkeep due to the ever-changing nature of the mobile market. The installation process for Molly will update your Wurfl database to the most recent version at every install and update (except when in development mode), but new devices may not yet appear in the Wurfl, and the Wurfl neglects to cover user agents belonging to desktop browsers. Therefore, Molly maintains a "local patch" to the Wurfl in `molly/wurfl/data/local_patch.xml`. This patch file format is documented by Wurfl and is merged into the main Wurfl file at update time. This file lives in the main Molly repository, and if you have come across a device which the Wurfl does not recognise, we would encourage you to commit it back to the main Molly Project as a patch so all users can benefit.

When modifying this file, you must first identify the user agent of the device, and if this device is a newer version of an already existing device, the Wurfl ID of the older version of the device (assuming that the newer device inherits the attributes of the older version). You can then simply add a new line like so:

```
<device user_agent="User-Agent-Of-My-New-Device" fall_back="wurfl_id_of_old_device"␣
↪id="my_new_wurfl_id"/>
```

New devices can be added following the format specified in the main Wurfl docs.

The Wurfl will cover most mobile devices eventually, so you should be able to remove this patch after a period of time. Desktop browsers appear to be slower to be updated in the Wurfl desktop patch.

"Batteries-Included" Apps

`molly.apps.contact` – Contact search

This application provides contact search functionality

Configuration

- provider: the provider which provides contact search results

Sample:

```
Application('molly.apps.contact', 'contact', 'Contact search',
    provider = Provider('molly.apps.contact.providers.LDAPContactProvider'
                        url='ldap://ldap.mit.edu:389', base_dn='dc=mit,dc=edu'),
),
```

Providers

molly.apps.contact.providers.LDAPContactProvider

This queries an LDAP server for contact details. It takes three options:

- url: The URL to the LDAP server to use
- base_dn: The base DN to use when searching the LDAP tree
- phone_prefix (optional, defaults to blank): A prefix to add to phone numbers, e.g., if your LDAP servers only store extension numbers, you can add the prefix to make it externally callable here (e.g., '+44190443'). This is not used when `phone_formatter` is set.
- phone_formatter (optional, defaults to nothing): A custom "number formatter". The option expects to have a callable (either a function defined elsewhere in your settings file, or a lambda defined there) which is called with the raw phone number from LDAP and is expected to return a normalised phone number. Use this is you need more advanced logic than the phone_prefix option can give you.
- alphabetical (defaults to False): A boolean which indicates whether or not results from the LDAP server should be sorted alphabetically by surname.
- query (defaults to '(sn={surname})'): This is the query which is passed to the LDAP server. It uses new style string formatting and has two fields available 'forename' and surname.

Writing Your Own Providers

A contact provider must inherit `BaseContactProvider` and expose the following interface:

Views

index

This view lives at the `/` URL of this app and is handled by `molly.apps.contact.views.IndexView`.

This view renders `contact/index.html`, providing the following context:

- `form`: A form defined by the provider to use as the input for searching
- `medium_choices`: A list of tuples, representing the different media the provider can search (e.g., phone book, e-mail address list), in the format specified by

There are no overridable blocks provided by this template, but the search form is rendered by the separate template `contact/search_form.html`

result_list

This view lives at `results/` in this app and is handled by `molly.apps.contact.views.ResultListView`.

This view renders `contact/result_list.html` providing the following context:

- `form`: The form used to perform the search
- `medium`: The medium selected for this query
- `results`: The list of results (as returned by *molly.apps.contact.providers.BaseContactProvider.perform_query*)
- `message`: If set, any error messages generated in the search

There are no overridable blocks provided by this template, but each individual result is rendered by `contact/result.html`

result_detail

The view lives at `results/:ID:` in this app, and is handled by `molly.apps.contact.views.ResultDetailView`.

This view renders `contact/result_detail.html` providing the following context:

- `result`: The result object (as returned by *molly.apps.contact.providers.BaseContactProvider.perform_query* to be rendered)

molly.apps.desktop – Desktop

This shows a desktop site to desktop browsers (desktop browsers hitting the main page are redirected here)

Configuration

- `twitter_username` (optional): If set, this imports a Twitter feed which the desktop templates can display (this settings refers to Twitter's screen_name, not the user_id as precised in Twitter's documentation)
- `twitter_ignore_urls` (optional): If set, this filters out tweets in the Twitter feed which contain this URL prefix - this can be used, for example, if you're importing a news blog to your Twitter feed to avoid duplicate content
- `blog_rss_url` (optional): If set, this imports an RSS news feed which can be displayed to your users

Sample:

```
Application('molly.apps.desktop', 'desktop', 'Desktop',
    display_to_user = False,
    twitter_username = 'mobileox',
    twitter_ignore_urls = 'http://post.ly/',
    blog_rss_url = 'http://feeds.feedburner.com/mobileoxford',
),
```

Views

molly.apps.feeds.events – Event Listing

Used to display event feeds to the user.

Configuration

This module has no configuration

Sample:

```
Application('molly.apps.feeds.events', 'events', 'Events'),
```

Views

molly.apps.feature_vote – Feature Suggestions

This allows users to suggest features and then vote on them.

Configuration

This app has no additional configuration.

Sample:

```
Application('molly.apps.feature_vote', 'feature_vote', 'Feature suggestions',
    display_to_user = False,
),
```

Use

This app has a Django admin interface, under 'Feature vote' in the admin panel. When you get a feature request (which is sent to the admins as defined in the config) by e-mail, you can then go into the admin panel and either edit it to make it public, by editing the individual issue and ticking 'Is public' and clicking 'Save'. You can also delete features from here (e.g., spam).

When a feature is implemented, you can do this by setting the date of when it is implemented in the same edit screen. After 4 weeks, the feature will disappear from the suggestions screen. Also, marking a feature as implemented will notify the original user by e-mail of this, as well as letting all users who voted know by a message on the home page.

Feature

Views

index

This view lives at the / URL of this app and is handled by `molly.apps.feature_vote.views.IndexView`.

This view renders the list of currently active features, and handles voting (via POST) on these features, as well as submission of new features. New features which are submitted are e-mailed to the admins using the template `feature_vote/feature_create.eml`.

The template used to render requests is `feature_vote/index.html`, and the context procided to this template consists of:

- features - a list of currently public and active Feature objects
- form - A Django form for new feature submission
- submitted - a Boolean indicating whether or not the user just submitted a new idea

feature-detail

This view lives at the `/:id:/` URL of this app and is handled by `molly.apps.feature_vote.views.FeatureDetailView`. It takes one argument, id, which is the ID of the specific feature to be rendered. Non-existant or non-public features throw a 404.

This view renders detail about a particular feature. The template used to render this file is `feature_vote/feature_detail.html`. The following things are provided in the context:

- feature - the feature object being detailed

Templates

feature_vote/feature_create.eml

This is the template used for e-mails sent to the site administrators when a new feature is submitted. The following things are in the context:

- name - the name of the submitting user
- email - the e-mail of the submitting user
- title - the title of the submitted idea
- description - the description of the submitted idea
- feature - the feature object which was submitted

This template has no blocks.

feature_vote/feature_detail.html

This template has no blocks.

feature_vote/implemented.eml

This is the template used when notifying a submitting user that their idea has been implemented on the site. It has one thing in the context:

- feature - the feature object which the user submitted and has been implemented

This template has no blocks.

feature_vote/index.html

This template has no blocks.

feature_vote/vote.html

This is used by `feature_vote/index.html` and `feature_vote/feature_detail.html` to abstract the voting box (up and down arrows) on features. It has no blocks.

Styling

feature_vote/css/smart.css

This file defines the styling for the voting box, and how comments from superusers are displayed.

Media

The following images come with this app, and they correspond to the up and down voting arrows, in both an active and disabled ("voted") state.

- `feature_vote/images/vote-up.png`
- `feature_vote/images/vote-down.png`
- `feature_vote/images/vote-up-voted.png`
- `feature_vote/images/vote-down-voted.png`

`molly.apps.feedback` – Feedback

This allows for users to e-mail the site admins with feedback.

Configuration

This app has no configuration.

Sample:

```
Application('molly.apps.feedback', 'feedback', 'Feedback',
    display_to_user = False,
),
```

Views

`molly.apps.feeds` – Feed Manager

This is a utility app for importing feeds. The feeds to be imported are configured in the admin interface.

To manage your feeds, you can go to the admin interface (at http://m.example.com/adm/) and then select the 'Feeds' option in the 'Feeds' box on that page. On the following page you will see a list of feeds. To edit an existing feed, you simply need to click on the feed title. To create a new feed, simply click the 'Add Feed' button in the top right corner. You can also bulk delete feeds from this screen.

The Add and Edit screens for feeds are the same, with the Edit field being prepopulated with pre-existing values for that feed. The following fields exist:

- Title
- Entity - if this feed belongs to a place (for example, University department, bar, etc) which is in the places database, you can associate it with an entity here.
- RSS URL - this is the URL to the RSS (or Atom) feed. In future, other providers may exist, in which case this will be a URL to the feed to be parsed by that provider. Django will check that this URL is valid (responds to a HEAD request) upon saving.
- Slug - this is the unique identifier for this feed which is used to generate the URL the feed is displayed at, i.e., http://m.example.com/news/SLUG/.
- Language - in multi-lingual setups, you can set a language on a feed which means it is only shown to users in that language.
- Ptype - this is the type of feed that is being imported - event or news

- Provider - at present, only the RSS provider is supported, however in future more providers may be supported. This can therefore be used to specify the type of feed. Note that the RSS provider can also handle Atom feeds.
- Tags - this feature is currently unused by Molly, but allows for feeds to be tagged and possibly grouped together in future

---

Note: A common mistake is to leave the Provider value unset. This will result in the feed appearing, but no items appearing, as the RSS provider does not import it!

To save changes, or to create a new feed, simply use the 'Save' buttom at the bottom of the screen.

Once the feed has been created, the contents of it will be imported the next time the RSS importer runs (by default, hourly).

Configuration

- providers: A list of providers of feed importers

Sample:

```
Application('molly.apps.feeds', 'feeds', 'Feeds',
    providers = [
        Provider('molly.apps.feeds.providers.RSSFeedsProvider'),
    ],
    display_to_user = False,
),
```

Providers

molly.apps.feeds.providers.RSSFeedsProvider

Imports RSS (and Atom) feeds, takes no options.

Writing Your Own Providers

Views

This app has no views. Instead, the rendering of the two different types of feeds are handled seperately by two different applications:

- *molly.apps.feeds.news – News*
- *molly.apps.feeds.events – Event Listing*

molly.apps.home – Home Screen

This application renders the home screen. This app must be included.

Configuration

This app takes no additional configuration.

Sample:

```
Application('molly.apps.home', 'home', 'Home',
    display_to_user = False,
)
```

Views

molly.apps.library – Library Search

This application provides a front-end to electronic library catalogues.

Configuration

- provider: the provider which provides search results
- library_identifier: if set, this is used to look up entities (from the places app) in this identifier namespace which have the value of the library code as returned by the query

Sample:

```
Application('molly.apps.library', 'library', 'Library search',
            provider = Provider('molly.apps.library.providers.Z3950',
                        host = 'z3950.copac.ac.uk',
                        syntax = 'XML',
                        database = 'COPAC'),
),
```

Providers

molly.apps.library.providers.Z3950

This provides the library app with the ability to query library catalogues using the Z39.50 protocol. It supports the following options:

- host: The hostname of the Z39.50 server to connect to
- database: The name of the database on the server to use
- port (optional, defaults to 210): The port on which the Z39.50 server listens
- syntax (optional, defaults to USMARC): The syntax which the server uses for responses (currently supported: USMARC and XML)
- charset (optional, defaults to UTF-8): The encoding to use when communicating with the server
- control_number_key (optional, defaults to 12): The 'use attribute' to query on when doing a control number lookup,
- results_encoding (optional, defaults to marc8): The encoding results come back in. Valid values are 'marc8' and 'unicode'. Aleph needs 'unicode'.

Writing Your Own Providers

Views

molly.apps.feeds.news – News

Used to display news feeds to the user.

Configuration

This module has no configuration in setup.py. Rather, feeds are controlled in the database as discussed in *molly.apps.feeds – Feed Manager*.

Sample:

```
Application('molly.apps.feeds.news', 'news', 'News'),
```

Views

molly.apps.places – Places

A database of places with locations

Configuration

- providers: A list of providers of entities and information about entities
- nearby_entity_types: A list of tuples of the form (heading, [entity_types...]), where the entity_types are a list of entity type slugs to be included in this category. Used for deciding which category is shown on the Nearby page
- associations: A list of tuples which allow entities to be associated with one another, in the form ((scheme1, value1, (heading, ((scheme2, value2) , (scheme3, value 3)))), which would associate the entities identified by scheme2:value2 and scheme3:value3 with scheme1:value1 (currently this means that real time departure information for bus stops are additionally shown on the page)

Sample:

```
Application('molly.apps.places', 'places', 'Places',
    providers = [
        Provider('molly.apps.places.providers.NaptanMapsProvider',
            areas=('329',)
        ),
        Provider('molly.apps.places.providers.PostcodesMapsProvider',
            codepoint_path = CACHE_DIR + '/codepo_gb.zip',
            import_areas = ('OX',),
        ),
        'molly.apps.places.providers.ACISLiveMapsProvider',
        Provider('molly.apps.places.providers.OSMMapsProvider',
                 lat_north=52.1, lat_south=51.5,
                 lon_west=-1.6, lon_east=-1.0
        ),
        Provider('molly.apps.places.providers.ACISLiveRouteProvider',
            urls = ('http://www.oxontime.com',),
        ),
        Provider('molly.apps.places.providers.LiveDepartureBoardPlacesProvider',
```

```
            token = 'MyNationalRailToken'
        ),
        Provider('molly.apps.places.providers.BBCTPEGPlacesProvider',
            url='http://www.bbc.co.uk/travelnews/tpeg/en/local/rtm/oxford_tpeg.xml',
        ),
    ],
    nearby_entity_types = (
        ('Transport', (
            'bicycle-parking', 'bus-stop', 'car-park', 'park-and-ride',
            'taxi-rank', 'train-station')),
        ('Amenities', (
            'atm', 'bank', 'bench', 'medical', 'post-box', 'post-office',
            'public-library', 'recycling', 'bar', 'food', 'pub')),
        ('Leisure', (
            'cinema', 'theatre', 'museum', 'park', 'swimming-pool',
            'sports-centre', 'punt-hire')),
    ),
    associations = (
        ('atco', '9100OXFD', ( # Railway station
            ('Station Forecourt',
             (
                ('atco', '340000006R1'),
                ('atco', '340000006R2'),
                ('atco', '340000006R3'),
                ('atco', '340000006R4'),
                ('atco', '340000006R5'),
                ('atco', '340000006R6'),
             )
            ),
          )
        ),
    )
),
```

Providers

molly.apps.places.providers.ACISLiveMapsProvider

ACIS Live provides real time bus information in various UK counties. When enabled, this adds real time bus information to those counties (note, it's probably a good idea to check with your local council before enabling this!). This has no options.

The areas supported by this provider are:

- Bristol
- Buckinghamshire
- Cambridgeshire
- Gloucestershire
- Kent
- Lancashire
- York/North Yorkshire
- Oxfordshire

- South Yorkshire
- Wessex
- West Yorkshire
- Cardiff

molly.apps.places.providers.ACISLiveRouteProvider

This provider scrapes an ACIS Live instance to try and get route data for the real-time routes ACIS Live knows about. It is heavily recommended you consult with your council before enabling this, as it makes a lot of requests!

This has one option:

- urls (optional, defaults to all instances): the base URLs of the ACIS Live instances to be scraped for route data

molly.apps.places.providers.AtcoCifTimetableProvider

Note: This only implements ATCO-CIF as far as TfGM data is concerned. There are no other public releases of ATCO-CIF data to test against.

This imports ATCO-CIF (public transport timetable and route) data. This allows for bus routes to be rendered, as well as scheduled departure times from a bus stop.

This has one optional option:

- url: A .zip file containing the .CIF files to be imported

molly.apps.places.providers.BBCTPEGPlacesProvider

Warning: The BBC appear to have deactivated their TPEG feeds, so this importer may not give useful information

This imports TPEG (travel alert) data from the BBC. This has one optional option:

- url (optional, defaults to UK wide feed): the TPEG feed to import (the BBC provide individual ones for individual counties)

molly.apps.places.providers.LiveDepartureBoardPlacesProvider

This gives rail stations live departure (and arrival) boards. This has one required and 2 optional options:

- token: Your National Rail Enquiries token
- max_services (optional, defaults to 15): The maximum number of services to fetch
- max_results (optional, defaults to 1): How many boards to fetch at once

molly.apps.places.providers.cloudamber.CloudAmberBusRtiProvider

This provider scrapes a CloudAmber instance to try and get route data for the real-time routes CloudAmber knows about. It is heavily recommended you consult with your council before enabling this, as it makes a lot of requests!

This has one mandatory option:

- url: the base URL of the CloudAmber instance to be scraped for route data

molly.apps.places.providers.NaptanMapsProvider

This imports entities from the NaPTAN (bus stops, etc) database. This has the following option:

- areas (optional): a list of ATCO area codes which to import the data from

molly.apps.places.providers.OSMMapsProvider

This imports points of interest from the OpenStreetMap database. It has the following options, all of which are optional:

- lat_north: A northern bound on latitude which data is imported for (if not set imports all)
- lat_south: A southern bound on latitude which data is imported for (if not set imports all)
- lon_west: A western bound on longitude which data is imported for (if not set imports all)
- lon_east: An eastern bound on longitude which data is imported for (if not set imports all)
- url: The URL to the OpenStreetMap dataset to be imported (defaults to the England dataset)
- entity_type_data_file: A YAML file (see below) which contains the entity type definitions for the OSM importer
- osm_tags_data_file: A YAML file (see below) which contains the OSM tags to be imported, and how they map to Molly's entity types (defined in entity_type_data_file)
- identities_file: A YAML file (see below) which contains a mapping between any OpenStreetMap nodes or ways and an external data source which defines when the OSM entity is identical to the entity in the external data source.

Entity Type data files

This file defines a dictionary in YAML entity-type definitions for Molly, where the key is the slug of the entity type to be created. Any entity type which you want an OSM type to be matched to should exist in this file.

Each definition defines the name of the entity type, the singular and plural forms of it, the name of the category it belongs to, and defaults for whether or not they are shown in the nearby and category lists (these can be overriden by the database, and these settings are only respected on first import). Also included are a list of parent types (e.g., "ice cream cafe" objects also belong to the "food" type, so "food" is a parent of "ice cream cafe").

This file is optional, and Molly ships with a sensible set of defaults (see `molly/apps/places/providers/data/osm-entity-types.yaml`).

Note: Note for translators! This file isn't in Python, so isn't marked up! You will, however, still want the names in here to be translated, so you need to make sure your .po files include things defined here.

Example:

```
bank:
  category: Amenities
  parent-types: []
  show_in_category_list: true
  show_in_nearby_list: true
  verbose_name: bank
  verbose_name_plural: banks
  verbose_name_singular: a bank
church:
  category: Amenities
  parent-types: [place-of-worship]
  show_in_category_list: false
  show_in_nearby_list: false
  verbose_name: church
  verbose_name_plural: churches
  verbose_name_singular: a church
```

The example above defines two entity types for the OSM importer to deal with. The OSM tag data file can then refer to the slugs (the keys: e.g., 'bank' and 'church') when mapping OSM tags to Molly entity types.

OSM tag data files

This file defines which OSM tags are to be imported, and also which entity types in Molly they map on to. Each entity in the list contains at least two keys: 'entity-type' and 'osm-tag'. This means OSM items which have a tag which matches osm-tag will be imported, and assigned to entity-type.

In this file, you can also define subtags, which allow for specialisations of OSM tags and Molly types. For example, you may have an OSM item of type 'place_of_worship', but in Molly represent different faiths as different types. In this case, you can look at the religion tag, to determine a more specialised entity type to tag with. Another way to think about this is as an AND clause for the OSM tags. e.g.,

```
- entity-type: place-of-worship
  osm-tag: amenity=place_of_worship
  subtags:
  - {entity-type: chapel, osm-tag: place_of_worship=chapel}
```

In this case, OSM entities which have tags 'amenity=place_of_worship' AND 'place_of_worship=chapel' will be imported with the entity type of 'chapel', and entities which only have a tag 'amenity=place_of_worship' will be imported with the entity type 'place-of-worship'.

Note: All entity types referred to here must be defined in the corresponding OSM entity type data file.

Example:

```
- {entity-type: museum, osm-tag: amenity=museum}
- entity-type: car-park
  osm-tag: amenity=parking
  subtags:
  - {entity-type: park-and-ride, osm-tag: park_ride=bus}
```

In the example above, this imports OSM items with a tag 'amenity=museum' as the Molly entity type 'museum', OSM items with a tag 'amenity=parking' as entity type 'car-park', and OSM items with the tags 'amenity=parking' AND 'park_ride=bus' as entity type 'park-and-ride'.

Identity files

The file defines which OpenStreetMap nodes or ways are identical to an entity already in your database, which has been imported from another source. When such a node or way is encountered (and would be imported because it contains a tag in the entity type data file), then the OSM metadata is then added to the other entity, rather than a new entity being created.

```
W23679234: oxpoints:23232416
W23679567: oxpoints:23232405
```

In the example above, the OpenStreetMap way with ID 23679234 (which would normally get imported with URL http://example.com/places/osm:W23679234/) is instead mapped on to the entity at http://example.com/places/oxpoints:23232405/, i.e., /places/osm:W23679234/ is identical to /places/oxpoints:23232416/.

molly.apps.places.providers.PostcodesMapsProvider

This imports postcodes from the Code-Point Open database. It has the following options:

- codepoint_path: A path to where the Code-Point Open zip file is get on disk. If the file does not exist, it is obtained from freepostcodes.org.uk
- import_areas (optional): If set, it is a list of postcode prefixes which limits the area which is imported (this is highly recommended due to the size of the postcode database!)

Writing Your Own Providers

Views

Entities

molly.apps.podcasts – Podcasts

Provides an interface to browsing different podcast feeds.

Configuration

- providers: a list of providers which can be used to import podcasts

Sample:

```
Application('molly.apps.podcasts', 'podcasts', 'Podcasts',
    providers = [
        Provider('molly.apps.podcasts.providers.OPMLPodcastsProvider',
            url = 'http://www.bbc.co.uk/radio/opml/bbc_podcast_opml_v2.xml',
            rss_re = r'http://downloads.bbc.co.uk/podcasts/(.+)/rss.xml'
        ),
        Provider('molly.apps.podcasts.providers.RSSPodcastsProvider',
            podcasts = [
                ('top-downloads', 'http://rss.oucs.ox.ac.uk/oxitems/topdownloads.xml
↪'),
            ],
```

```
        ),
    ]
),
```

Providers

molly.apps.podcasts.providers.OPMLPodcastsProvider

This imports RSS feeds as defined in a single OPML file. This provider may be more useful as a base in which to write a custom parser to correctly get metadata, due to the wide variety of methods in which OPML files represent data. It supports the following options:

- url: The URL the OPML file lives at
- rss_re: A regular expression string which extracts the slug

molly.apps.podcasts.providers.RSSPodcastsProvider

This imports individually specified RSS feeds. It supports the following options:

- podcasts: A list of tuples in the form (slug, url) of RSS feeds to import
- medium: Whether these feeds are audio or video (or undefined)

molly.apps.podcasts.providers.PodcastProducerPodcastsProvider

This imports podcast producer feeds. It supports one option:

- url: The URL to import

Writing Your Own Providers

Views

molly.apps.sakai – Sakai integration

Provides a mobile view using the Sakai API. This should be authenticated using the OAuth authentication API.

Configuration

- host: The host of the Sakai instance
- tools: A list of tuples of slugs and names of implemented Sakai tools
- identifiers: A list of tuples of (namespace, (search)) of identifiers from Sakai and how they match to Molly user identifier namespaces

Sample:

```
Application('molly.apps.sakai', 'weblearn', 'WebLearn',
    host = 'https://weblearn.ox.ac.uk/',
    service_name = 'WebLearn',
    secure = True,
    tools = [
        ('signup', 'Sign-ups'),
        ('poll', 'Polls'),
    ],
    extra_bases = (
        ExtraBase('molly.auth.oauth.views.OAuthView',
            secret = SECRETS.weblearn,
            signature_method = OAuthSignatureMethod_PLAINTEXT(),
            base_url = 'https://weblearn.ox.ac.uk/oauth-tool/',
            request_token_url = 'request_token',
            access_token_url = 'access_token',
            authorize_url = 'authorize',
        ),
    ),
    enforce_timeouts = False,
    identifiers = (
        ('oxford:sso', ('props', 'aid',)),
        ('weblearn:id', ('id',)),
        ('oxford:oss', ('props', 'oakOSSID',)),
        ('oxford:ldap', ('props', 'udp.dn',)),
        ('weblearn:email', ('email',)),
    ),
),
```

Views

`molly.apps.search` – Whole site search

The search application allows the user to enter a query and retrieve results from across the site. Results are produced by the application's providers, and may determine results in whatever manner they choose.

Search providers may access an index for the entire site, or be application-specific. An institution may have a Google Search Appliance from which results are retrieved, or alternatively they may wish searches for ISBNs to go straight to a library catalogue page.

Configuration

- form (optional, defaults to a built-in form): A Django form to use when searching
- query_expansion_file: A file to use for query expansion
- providers: A list of providers of search results

Sample:

```
Application('molly.apps.search', 'search', 'Search',
    providers = [
        Provider('molly.apps.search.providers.ApplicationSearchProvider'),
        Provider('molly.apps.search.providers.GSASearchProvider',
            search_url = 'http://googlesearch.oucs.ox.ac.uk/search',
            domain = 'm.ox.ac.uk',
            params = {
```

```
                'client': 'oxford',
                'frontend': 'mobile',
            },
            title_clean_re = r'm\.ox \| (.*)',
        ),
    ],
    query_expansion_file = os.path.join(project_root, 'data', 'query_expansion.txt'),
    display_to_user = False,
),
```

Providers

GSASearchProvider

This search provider retrieves results from a GSA (Google Search Appliance) as XML. Results are augmented using `get_metadata()`.

Options:

- search_url: The URL of the GSA
- domain: The domain of your deployment (used to restrict search results to)
- params (optional, defaults to nothing): Optional parameters to pass with the search request
- title_clean_re (optional, defaults to nothing): A regular expression to tidy up page titles when returned

`params` are added to the query string of the URL used when fetching request from the GSA. Further information about valid parameters can be found in the Google Seach Appliance documentation.

Where provided, `title_clean_re` is a regular expression containing a single group (i.e. paranthesised expression). If the title of a page as returned by the GSA matches the regular expression, it is substituted with the matched group. This can be used to remove common elements of titles.

ApplicationSearchProvider

This provider allows for Molly apps to return search results (e.g., library books, places, etc). It has no options.

Adding Search Capability to your App

Writing Your Own Providers

Providers should extend the `BaseSearchProvider` interface:

`BaseSearchProvider` also provides the following utility methods you may find useful in your implementations:

Search results

Individual search results are represented as dictionaries where the following keys have particular meanings:

`url` (*required*) The local part of the URL for the page. Will be used as an `href`.

`title` (*required*) The page title.

application (*recommended*) The name of the application that handles the URL. May be used to display an icon next to each result.

excerpt The bit of the page relevant to the query. May contain HTML (i.e. should be marked safe in a template).

additional More information about the resource represented at the URL. For example, the *places* application returns the type of entity and a distance from the user's location.

redirect_if_sole_result A boolean, default `False`, which will cause the search page to redirect to the URL if only one result is returned.

exclude_from_search A boolean, default `False`, which will exclude the page from any displayed search results. Can be used to exclude irrelevant results or those not intended for mobile devices.

Views

This application defines just one view:

index

Presents a single-field form to the user, and where a query has been submitted retrieves results from all configured search providers.

Results from multiple providers are presented in the order they were configured, i.e. the topmost configured provider's results are given precedence. Where more than one provider returns the same result, metadata are combined (with the location given by its first occurance) and only one result is displayed.

This view renders to the `search/index.html` template, passing a `SearchForm` instance as `form` and (where a search has been performed) a list of results as `results`.

Query Expansion

molly.apps.service_status – Service status

The service status application is intended to display whether services are up or down (or somewhere in between).

Configuration

- providers: the information sources

Sample:

```
Application('molly.apps.service_status', 'service_status', 'Service status',
    providers = [
        Provider('molly.apps.service_status.providers.RSSModuleServiceStatusProvider',
            name='University of Example IT Services',
            slug='it',
            url='http://www.example.ac.uk/it/status.rss')
    ],
),
```

Providers

molly.apps.service_status.providers.RSSModuleServiceStatusProvider

In the most part, such information would be provided by an RSS feed. We encourage the use of the service status RSS module, the concepts of which are used in the provider interface.

It supports the following options:

- name: The name of the service this feed includes (displayed to the user)
- slug: The feed slug
- url: The URL of the feed

Writing Your Own Providers

Views

molly.apps.tours – Tours

This module allows users to build their own walking tours by selecting destinations from a list of points of interest. It will then give the user directions to the start of the route using long-distance public transport routes or arrival points, taking into account park and ride routes.

Configuration

Views

Templates

Styling

Media

molly.apps.transport – Transport dashboard

A dashboard page which takes transport related data from the places app and displays it in a friendlier way

Configuration

- `train_station`: A string in the form 'scheme:value' identifying the entity to use as the train station on the page (this is the same form as in /places/scheme:value/)
- `train_station_nearest`: A boolean (defaulting to False) specifying that if the user has a location set, whether or not they should instead be shown their closest rail station instead.
- `nearby`: A dictionaries of entity types where values are in the form: (entity-type-slug, number to show). The keys form the values of the page URLs in the rendered remplate.
- `*_status_provider`: A provider which provides statuses for currently running services. X should be replaced with the type of service that that provider serves and should match up with the key in 'nearby' for that type of entities and then it is shown on that page, e.g., in the example below, to add a provider to deal with bus stops, the setting would be `bus_stops_status_provider`

- park_and_rides: if specified, then the park and ride entities are shown on the page in the order specified (scheme:value strings)
- travel_alerts: Whether or not to display travel alerts on this page

Sample:

```
Application('molly.apps.transport', 'transport', 'Transport',
    train_station = 'crs:OXF',
    nearby = {
        'bus_stops': ('bus-stop', 5),
        'tube': ('tube-station', 3),
    },
    park_and_rides = ('osm:W4333225', 'osm:W4329908', 'osm:W34425625',
                      'osm:W24719725', 'osm:W2809915'),
    travel_alerts = True,
    tube_status_provider = 'molly.apps.transport.providers.TubeStatusProvider',
),
```

Providers

molly.apps.transport.providers.TubeStatusProvider

This is a provider which provides the current running Tube line status. It has no options.

Writing Your Own Providers

The Transport app chiefly uses providers from the places app, however, if you want to show current "transit line" status, then you can do this by writing a custom provider and providing that to the transit_status_provider setting.

Transit status providers are simple and only have to provide one method, following the format below:

Views

All transport views contain the following in their context:

- location: the location of the current user (which was used to search)
- train_station: whether or not the rail departures page is enabled
- travel_alerts: whether or not the tranport page is enabled
- park_and_rides: whether or not the Park & Ride page is enabled
- public_transport: a dictionary of enabled public transport pages (value is a Boolean indicating whether or not it's active)

index

This view is a simple view that lives at the / URL. It redirects users to the last subpage of the transport app that they were on. If they've never visited the transport section of the site before, they are redirected to the bus page. This is handled by molly.apps.transport.views.IndexView.

public-transport

This page handles different types of generic transport entities, such as bus stops or tram stops. The types that are handled are defined by the `nearby` option in the application configuration. It lives at the URL `/:slug:/` URL inside this app, where the slug is any valid key given in the transport app. This view is handled by `molly.apps.transport.views.PublicTransportView`.

This view is location sensitive and renders the template `transport/public_transport.html` with the following in its context:

- `type`: the EntityType for these objects
- `entities`: the list of Entities to be rendered (ordered by distance)
- `pageslug`: the key for this page
- `favourites`: a list of Entities of this type which have been favourited
- `route_ids`: a sorted list of route IDs for routes which serve these types of entities
- `selected_routes`: the Route objects which match up to the IDs the user selected
- `line_status`: the results of the `get_status` call on your line provider

All entities are fully annotated with metadata.

This page can limit with services are shown by passing the `route` parameter as a GET parameter. Multiple results can be obtained by specifying it multiple times.

routes

This page shows a full list of routes which serve the type of entity specified in the slug. This is a subpage to `public-transport` and the allowed types are handled in the same way. This lives at `/:slug:/routes/` inside the app, and handled in `molly.apps.transport.views.RoutesView`.

This view is location sensitive and renders the template `transport/routes.html` with the following in its context:

- **`routes`: all Route objects which serve this type of entity, sorted by** service_id

If the user's location is known, each route will be annotated with two additional fields `nearest`, `nearest_distance` and `nearest_bearing`, the nearest Entity which serves this route, the distance and bearing to that entity respectively.

rail

This page shows the live departure board for the closest train station if `train_station_nearest` is set to True in the config and the user has set a location, or otherwise the value of the `train_station` setting.

This page lives at `/rail/` inside the app, and handled by `molly.apps.transport.views.RailView`.

The template rendered is `transport/rail.html` which has the following in its context:

- `entity`: the rail station to be rendered
- `board`: the board currently being shown

The parameter `board` can be specified in the GET query string and this can be either 'arrivals' or 'departures' and is used to determine which board is rendered. This defaults to departures.

park-and-ride

travel-news

This view lives at `/travel-news/` inside the app and is handled by `molly.apps.transport.views.TravelNewsView`.

This provides a list of entities of type `travel-alert` from the places database, ordered by distance from the user (or alternatively, title, if no location is known). This is provided by a single item in the context called `travel_alerts`.

Templates

transport/base.html

This template contains the following blocks:

- `transport_links`: A list of buttons which canbe used to jump to particular parts of the page. This defaults to Bus Stops, Park and Rides, Rail Stations and Travel Alerts. If any of those things are not being rendered, then they are hidden. Any additional anchors should be added to this block.

This template is used by all subtemplates to provide a consistent look and feel.

transport/park_and_ride.html

This page contains no blocks, and renders the list of park and rides, including the capacity of each park and ride using the `park_and_ride` key on the entity metadata.

transport/public_transport.html

This page contains no blocks and renders the transit line status (if defined), followed by the list of favourite entities, and then nearby entities.

transport/rail.html

This renders a live departure board from `places/metadata/ldb.html`. It contains no blocks.

transport/routes.html

This renders the list of routes. It contains no blocks

transport/travel_news.html

This renders travel alerts as a list with a link to the travel alert. It contains no blocks.

Styling

transport/css/smart.css

This is an overridable CSS file. It defines CSS selectors for the anchor buttons at the top of the page, park and ride capacity indicators, background images for the appropriate sections and styling for London Underground transit status displays.

Media

The following images are bundled in the `transport/images/` directory:

- `bus.png`, `bus.svg` and `bus-small.png`: Icons to indicate the section of the page dealing with bus stops, in various forms.
- `pride.png`, `pride.svg` and `pride-small.png`: Icons to indicate the park and ride section of the page.
- `train.png`, `train.svg` and `train-small.png`: Icons to indicate the rail station section of the page.
- `warn.png`, `warn.svg` and `warn-small.png`: Icons to indicate the travel alerts section of the page.

`molly.apps.weather` – Weather

This shows weather to users (and by default, is embedded in the home page)

Configuration

- location_id: the code of the weather to use (use 'bbc/ID', where ID is the location ID for the BBC)
- provider: the weather information source

Sample:

```
Application('molly.apps.weather', 'weather', 'Weather',
    location_id = 'bbc/25',
    provider = Provider('molly.apps.weather.providers.BBCWeatherProvider',
        location_id = 25,
    ),
),
```

Providers

molly.apps.weather.providers.BBCWeatherProvider

Uses the BBC Backstage datafeed to get weather information. The ID for the weather location can be obtained by looking at the URLs when doing a weather search. It has one option:

- location_id: The BBC ID for this weather location

Writing Your Own Providers

Views

molly.apps.webcams – Webcams

This shows webcams to users. The webcams to be imported are configured from the admin interface.

Configuration

This has no additional configuration in the settings file, just in the database.

Sample:

```
Application('molly.apps.webcams', 'webcams', 'Webcams')
```

Managing Webcams

This app can be managed by going to the admin interface at http://m.example.com/adm/ and logging in using the username and password created at install time. From here, you can select the 'Webcams' option from the 'Webcams' box on the front page to manage webcam feeds that are being displayed to users. To create a new webcam, you simply need to click on the 'Add webcam' button, whereas to edit a webcam, you can click on the webcam title. You can also bulk delete webcams from this screen.

On the following screen, there are three fields and a box to allow for multi-lingual content to be configured:

- slug - this is the unique identifier that forms the URL for this webcam, i.e., http://m.example.com/webcam/SLUG/
- URL - this is the URL for the static image for the webcam. Molly proxies this image, so if the webcam is private, it can be set so only Molly can access it. The formats supported depend on the support that PIL was compiled with, but by default should include JPEG and PNG. Django ensures
- Fetch period - this is how long Molly should wait between requests (i.e., how long the image should be cached for). Note that Molly only requests the webcam when it is dealing with a request for the webcam, so this is not a constant fetch rate.

The box below is then split into rows with one row per language. You can then specify details for the webcam in multiple languages to support multi-lingual websites. For single language websites, you must fill in the top line and can leave the rest blank.

The four columns are:

- Language code: The language of this row
- Title: The name of this webcam (shown as the page title and also on the list of webcams).
- Description: In the default template, this is shown below the webcam image.
- Credit: In the default template, this is shown below the description.

Once you have entered this information, you can click 'Save' at the bottom to save the changes.

Webcam

Views

index

This view lives at the / URL of this app and is handled by `molly.apps.webcams.views.IndexView`. It is rendered with a week-long cache.

This view provides a list of all webcams and is used to render a list of webcams for the user to choose. The rendered template is `webcams/index.html` and a single item is provided in the context: `webcams` which is an (unordered) list of all `molly.apps.webcams.models.Webcam` objects known to Molly.

webcam

This view lives at the `/:slug:/` URL of this app, and is handled by `molly.apps.webcams.views.WebcamDetailView`.

The context consists of a single item: `webcam`, which corresponds to the `molly.apps.webcams.models.Webcam` object in the database where the slug matches the `:slug:` parameter in the URL. The rendered template is `webcams/webcam_detail.html`.

Templates

webcams/index.html

This template defines no new blocks and uses standard styling to render a link list to all webcams known about.

webcams/webcam_detail.html

This template defines no new blocks and uses standard styling to render the webcam image with description and credit beneath it, with title as the page header.

It uses the `extrahead` block to define a meta refresh period of 60 seconds.

Styling

Webcams uses core Molly styles and defines non-itself.

Media

Webcams does not use any additional media.

CHAPTER 4

Miscellany

License

Python Module Index

m

Index

Symbols

A

F

G

M