LASS Documentation

Release 1.0

URY Computing Team

August 30, 2013

CONTENTS

Contents:

CHAPTER

ONE

META-DOCUMENTATION

This part of the *LASS* documentation introduces *LASS* and the purposes of the documentation itself.

1.1 Introduction to LASS

Section author: Matt Windsor <matt.windsor@ury.org.uk>

1.1.1 What is LASS?

LASS (the *Longevity Assured Site System*) is a Django project providing the framework for a website suitable for student radio stations.

It was originally written to replace the University Radio York website (http://ury.org.uk), but is available under the GNU General Public Licence version 2 for inspection and adaptation in other sites and contexts.

Technically, *LASS* refers only to the collection of applications that form the URY website codebase; the templates and static data that complement the Django code to form a full website are separate concerns.

1.1.2 Why would I need LASS?

LASS is provided in the hopes that it may be helpful in part or in whole to other amateur, community or student radio stations wishing to build a website, as a free software bundle of existing solutions to problems faced when URY built their website.

It also obviously continues to be developed as the backend codebase for the URY site itself! (At time of writing.) As such, anyone is welcome to extend or help develop it.

1.1.3 What is this documentation?

This is the semi-automated documentation for LASS, which combines docstrings extracted from the actual LASS code with ancillary manual documentation written by humans.

The aim is to serve as one centralised location for all information about LASS and how to use, extend and improve it.

This document covers the integrated LASS project only; for the individual app documentation see the URY project page on ReadTheDocs.

1.1.4 Is this the only documentation?

This is the main source of documentation for LASS; in adherence to the Don't Repeat Yourself principle, most if not all of the documentation for the project is "baked in" as either docstrings or supporting files bundled with the project.

However, this autodocumentation is admittedly likely to be lacking at the start of the project and may lag behind the knowledge of the LASS contributors. If you find that something is poorly documented, see *Contributors* for details on how to contact the LASS contributors for assistance.

1.2 Licence

The licence in full may be found in the source tree as 'COPYING'.

1.2.1 LASS code

Why GPL2?

We realise that the GPL2 is not the most optimal licence for this type of Web application, and that the AGPL or permissive licences would have suited the project better.

However, LASS is under the GPL2 due to the inclusion, at time of writing, of the xapian-haystack search backend, which is indeed under the GPL2.

1.2.2 This documentation

As this documentation is mostly extracted from the GPL2 source, it is also distributed under the terms of GPL2 (for better or worse).

1.3 Contributors

This section lists contributors to the LASS project.

Where possible authors of contributed modules/apps used as dependencies to the project will be recognised; however this list may be outdated.

1.3.1 LASS Codebase

- Matt Windsor <matt.windsor@ury.org.uk> - Initial version code

uryplayer

- Donal Cahill <donal.cahill@ury.org.uk> - Initial URYPlayer podcast code

1.3.2 URY Website

Contributors to the URY website in general, which is at time of writing almost entirely included in the LASS package.

Without these people, the initial deployment of LASS would not have been possible.

Style and Design

- Danny Bell <danny.bell@ury.org.uk> - Initial design implementation
- Andy Durant <andy.durant@ury.org.uk> - Most of the CSS
- Rob Stonehouse <rob.stonehouse@ury.org.uk> - Design ideas, awesomeness

1.3.3 External Modules

TBC

CHAPTER

TWO

INSTALLATION INFORMATION

Section author: Matt Windsor <matt.windsor@ury.org.uk>

LASS is a non-trivial project to install. Work is constantly being carried out to make the project easier to deploy, but at this early stage installation requires a large amount of manual intervention.

2.1 Requirements

LASS carries a `pip` requirements file as `requirements.txt`; you can use this to pull the dependencies and sub-apps in.

2.2 Init scripts

Some things that LASS uses are written in languages other than Python, but nonetheless must be installed into any virtualenv LASS runs in in order for LASS to work. At time of writing, these include SQLite (for running unit tests) and Xapian (search backend).

Generally, scripts to install these into a virtualenv are available in the `init_scripts` directory as Bourne shell scripts.

These will *not* work outside of a virtualenv; in those cases, you will need to use your operating system's native means of installing these dependencies.

2.3 Templates and Static Files

The LASS distribution does *not* include templates; these will eventually be available from the same locations as the code distribution, or you can roll your own if you choose.

Similarly, static files (CSS, images etc.) are not provided - you will almost certainly want to make your own statics as the ones used at URY constitute URY's unique station branding.

A "vanilla" set of CSS/LESS stylesheets may be provided in the future.

2.4 Required Data

You will need to populate the database with some initial data. Where possible, the data is provided as fixtures to load with Django's `loaddata` command.

2.4.1 Filler show

The filler show must be created in order to allow the schedule filling algorithms to work properly.

The provided `schedules/fixtures/filler_show.json` fixture gives an example configuration for the filler show. Note that this fixture is incomplete: it does not *approve* or assign a *creator* to the show location, and the primary keys will likely need to be adjusted.

In addition, `filler_show.json` does *not* contain show metadata; you will need to add that manually for the time being.

CHAPTER

THREE

CONFIGURATION

Section author: Matt Windsor<matt.windsor@ury.org.uk>

LASS uses the Django configuration system, but in a slightly customised manner using YAML files for most of the configuration options.

Some config is still baked into the *settings.py* file; usually this is either things that cannot be expressed well in YAML, things that should be common to all LASS installations, and also the bootstrap for the YAML configuration system.

3.1 Configuration files

You will generally need to supply some Django settings to LASS before it works, most notably the database and assets locations. These use the standard Django names (`DATABASES`, `TEMPLATE_DIRS` etc), but are read from YAML files (ending in '.yml') in the `private` directory. (The name is an artefact from its use to store URY-private settings.)

LASS loads all `*.yml` files in Python string sort order, with each settings file clobbering any previous Django settings loaded. This means that you can store site and server-specific settings in files named such as to come after more general settings files, and indeed this is how URY resolve differences between environments.

3.1.1 File format

Each file should contain a map of Django values to the YAML representations of their settings, as interpreted by *PyYAML*. An example:

```
# This can be overridden by YAML files loaded later
DEBUG: False

# Paths
MEDIA_ROOT: /usr/local/assets/media
STATIC_ROOT: /usr/local/assets/static

# Admins
ADMINS:
    - - Head of Everything
      - hoe@radio.example.com
    - - Webmaster
      - sucker@radio.example.com
SERVER_EMAIL: noreply@example.com
```

Files can obviously contain any YAML syntax that *PyYAML* understands.

3.1.2 Defaults

Some Django settings have default values baked into `settings.py` in the event that they are not specified in any private YAML. See the code file for information.

CHAPTER

FOUR

STRUCTURE OF THE LASS SYSTEM

Section author: Matt Windsor<matt.windsor@ury.org.uk>

This is a brief overview of the structure of the *LASS* project as a whole, including its sub-apps.

In cases where the current state of the code differs from this outline, the code takes precedence. LASS is a constantly changing beast, and thus this can go out of date very quickly.

4.1 Why is there so much boilerplate/wheel reinventing?

Often, LASS has needed to reinvent things that Django or existing applications already do.

The usual reason for this is that it is very important that LASS's database usage plays fair with existing URY applications both legacy and new, and thus the tendency is to eschew any system that would introduce "djangoisms" into the database. This includes things like content-types, MySQL-style prefixed tables (URY runs PostgreSQL), as well as the desire for LASS to be able to use URY's database table naming conventions where possible.

Usually when it comes to things that LASS doesn't need to share with other applications, our main philosophy is to reuse where possible, which is why LASS pulls in a large amount of external apps at the website level.

Another reason is that we're only human, and genuinely might have overlooked a better solution!

4.2 Base modules

Most of LASS is implemented in a few highly generalised apps, which provide a class framework for the higher levels of the system to use.

These are referred to here, with documentation links, in a vague order from low-level to high-level. Usually items referenced lower down the list will have dependencies on packages higher up.

4.2.1 Utils

The "root" of the system is the `django-lass-utils` app, which is intended to have no other dependencies from the LASS project.

The utils app contains a large amount of extremely general mixins and other features used by practically everything else, including:

- The `AttachableMixin` pattern for quickly creating models that provide extra data to other models, in a manner that doesn't pollute the database with Django-specific idioms;
- The `Type` abstract model, used for anything that refers to a type or category of other model (for example, show types; metadata keys)
- Various other odds and ends.

The documentation for `django-lass-utils` is here.

4.2.2 People

Large amounts of data in LASS are tied to people, for example show credits, data change approval, metadata creators and so on. The `django-lass-people` app provides mixins, models and other items for dealing with these.

The documentation for `django-lass-people` is here.

4.2.3 Metadata

A large part of the LASS backbone is the `django-lass-metadata` app, which implements a general metadata system using the `AttachableMixin` from `django-lass-utils` that can be used to attach typed key-value metadata (with history and both single and multiple-value support) to arbitrary models.

It also provides, through `MetadataSubjectMixin`, a very simple API for accessing metadata that has been attached to models. Often, currently valid metadata can be accessed as if it was an attribute on the model, making transition to and from the metadata system easy.

LASS uses this nearly everywhere where items of data need titles, descriptions, tags, attached images, or internal notes.

The documentation for `django-lass-metadata` is here.

CHAPTER

FIVE

INTERNAL APPLICATIONS

This section of the LASS documentation contains documentation for each of the applications directly forming the LASS project at time of compilation.

5.1 Application index

5.1.1 Schedule

schedule - radio station scheduling

The *schedule* app provides models, views and services related to the URY radio schedule and retrieving portions of it.

In addition, *schedule* contains and exposes the *show database*, or the collection of all shows, seasons and timeslots URY manages.

It is also responsible for providing schedule-related information to other apps: the page header at time of writing embeds a `schedule` view in order to provide information about the up-coming shows.

models

utils

Utility functions for the schedule.

These functions mainly concern retrieving contiguous "chunks" of the schedule.

filler

list

range

tests

admin

search_indexes

5.1.2 uryplayer - Radio content "on demand"

The URY Player system, which provides external access to URY's non-live media (at time of writing: podcasts).

tests

Tests for the URYPlayer system.

class uryplayer.tests.**AdminTest**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests to make sure that the admin snap-ins validate correctly.

setUp()

test_admin_registration()
Tests that registering the admin hooks with an admin site yields no exceptions.

views

Views for the URY Player system.

uryplayer.views.**home_podcasts**(*request*, *amount=5*, *block_id=None*)
Snap-in view for the URY Player box on the front page.

uryplayer.views.**podcast_channel_latest**(*request*, *channel*)
Redirects to the latest podcast in a given channel.

The channel can be specified by name, ID or object.

admin

class uryplayer.admin.**PodcastAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

description(*obj*)

inlines = [<class 'uryplayer.admin.PodcastPackageEntryInline'>, <class 'uryplayer.admin.PodcastTextMetadataIn

list_display = ('title', 'description', 'date_submitted', 'id')

media

title(*obj*)

class uryplayer.admin.**PodcastChannelAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

inlines = [<class 'uryplayer.admin.PodcastChannelTextMetadataInline'>, <class 'uryplayer.admin.PodcastChanne

list_display = ('name', 'description')

media

class uryplayer.admin.**PodcastChannelTextMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.TextMetadataInline

media

model
alias of PodcastChannelTextMetadata

class uryplayer.admin.**PodcastChannelTextMetadataRuleInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.TabularInline

media

model
alias of PodcastChannelTextMetadataRule

class uryplayer.admin.**PodcastImageMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.ImageMetadataInline

media

model
alias of PodcastImageMetadata

class uryplayer.admin.**PodcastPackageEntryInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.PackageEntryInline

media

model
alias of PodcastPackageEntry

class uryplayer.admin.**PodcastTextMetadataInline**(*parent_model*, *admin_site*)
Bases: metadata.admin_base.TextMetadataInline

media

model
alias of PodcastTextMetadata

uryplayer.admin.**register**(*site*)
Registers the uryplayer admin hooks with an admin site.

5.1.3 *metadata* - the LASS metadata system

The *metadata* app contains the generic metadata system used in LASS, which allows objects in the LASS model set to contain key-value stores of textual, image-based and other formats of metadata by inheriting a mixin and providing a subclass of the standard metadata models.

Metadata system

The *metadata* app is dedicated to the LASS *metadata system*, which allows various different *strands* of data (generally text, but also image-based and other formats) to be attached to items.

LASS uses the metadata system, for example, to provide shows with names and descriptions that have full recorded history and hooks for an approval system. It is also used to associate images (thumbnails and player insets) with podcasts.

Strands

Each model can have zero or more *strands* of metadata attached to it. Each strand is its own model (see below for more information on how to create a metadata provider model), and represents a specific collection of metadata on objects in the subject model.

Strands are indexed by name; an entire strand (as a dictionary-like object) can be retrieved from an implementor of *MetadataSubjectMixin* with `object.metadata()['strand-name']`. Generally, there will be a `text` strand containing all textual metadata, and an `images` strand containing thumbnail images and other related pictorial metadata.

Implementation

All metadata strands are implemented as key-value stores, the key store being implemented as one unified model for simplicity reasons and the value stores being separate for each strand for each model.

Two classes (*metadata.mixins.MetadataSubjectMixin* and *metadata.models.GenericMetadata*) provide the core framework for defining a metadata subject and a metadata strand. There are descendents of *GenericMetadata* available for specific commonly used strand types.

Examples

In the LASS project, examples of how to use the metadata system can be found in *schedule.models.show*, *ury-player.models.podcast* and *people.models.role*.

Models

Mixins

Misc

Administration hooks

Unit tests

admin_base

Hooks

Queries

5.1.4 *laconia* - Lightweight external exposure of LASS data

The *laconia* module holds machine-readable views of information in LASS models. It's like an ad-hoc API.

Its name comes from the area from which the Spartans originated; it often returns information in short, terse bursts in the Spartan tradition.

Things that belong in *laconia*:

1. Application-specific formatted views of data
2. Temporary glue logic that will eventually be replaced with APIs
3. Anything that is intended to be embedded in a script but does not require a heavyweight retrieval mechanism

Much of *laconia* is temporary and anticipates the creation of proper structured API exposure of data. As such, only depend on *laconia* runoffs if you're reasonably confident that they are either permanent or will be easily replacable in your code in the future.

tests

views

5.1.5 *grid* - basic framework for dynamic grids of widgets

The *grid* app contains plumbing for a very basic dynamic widget grid system.

grid was originally designed for the URY front page, which is composed of various rectangular sections that each display a separate view. It was intended to separate mechanism from policy by allowing the website templates to contain no information about the specific make-up of the front site grid and the exact positions of parts of it.

At time of writing, *grid* is very feature-poor and inflexible, having effectively been created as a quick method of making the aforementioned home page dynamic. However, it is hoped that it can be generalised in the future to other types of widget grid.

Models

Models for the home page grid system.

class grid.models.**Grid**(**args*, ***kwargs*)
Bases: lass_utils.models.type.Type

An object representing a grid of ‘GridBlock‘s.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Grid.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Grid.**blocks**

Grid.**gridblockinstance_set**

Grid.**objects = <django.db.models.manager.Manager object at 0x2f10210>**

Grid.**ordered_list**()
Returns a list of blocks active on this grid, ordered to render.

Provides all the blocks This grid has instances for Ordered Y then X

class grid.models.**GridBlock**(**args*, ***kwargs*)
Bases: lass_utils.models.type.Type,metadata.mixins.metadata_subject.MetadataSubjectMix

A block in the home page grid.

Grid blocks are akin to widgets or dashboard items in other analogous systems.

A *GridBlock* does not contain any positioning information; the *GridBlockInstance* and *Grid* models are concerned with this.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception GridBlock.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

GridBlock.**grid_set**

GridBlock.**gridblockinstance_set**

GridBlock.**gridblocktextmetadata_set**

classmethod GridBlock.**make_foreign_key**()
Shortcut for creating a field that links to a grid.

GridBlock.**metadata_strands**()
Returns the set of metadata strands available for this grid block.

GridBlock.**objects = <django.db.models.manager.Manager object at 0x2f0c250>**

GridBlock.**podcast_channel**

class grid.models.**GridBlockInstance**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

An instance of a grid block within a grid.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception GridBlockInstance.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

GridBlockInstance.**grid**

GridBlockInstance.**grid_block**

GridBlockInstance.**objects = <django.db.models.manager.Manager object at 0x2f107d0>**

Template tags

The implementation-level side of the *grid* framework is contained in a template tag library, *grid_tags*. Template tags for the *grid* system.

grid.templatetags.grid_tags.**grid**(*context*, *grid_id*)
Renders the grid of the given ID.

This block simply outputs the blocks instanced in the grid in sequence; you will generally want to ensure that your grid CSS automatically breaks the grid stream into rows and columns.

Parameters **grid_id** (string, integer or *GridBlock*) – the ID (name or primary key) of the grid

Return type a template tag node

grid.templatetags.grid_tags.**grid_block**(*context*, *block_id*)
Renders the grid block of the given ID.

Parameters **block_id** (string, integer or *GridBlock*) – the ID (name or primary key) of the block

Return type a template tag node

5.2 people - radio station members and contacts

5.2.1 tests

5.2.2 admin

5.3 music - Radio music team input

5.3.1 models

5.3.2 tests

5.3.3 views

5.4 website

5.4.1 tests

5.4.2 views

5.4.3 admin

5.5 getinvolved

5.5.1 tests

5.5.2 views

5.6 search

5.6.1 tests

5.6.2 views

CHAPTER

SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX