AusDTO Discovery Layer

Release 0.0.1-pre-alpha

Commonwealth of Australia, Digital Transformation Office

September 07, 2015

Contents

CHAPTER 1

Overview

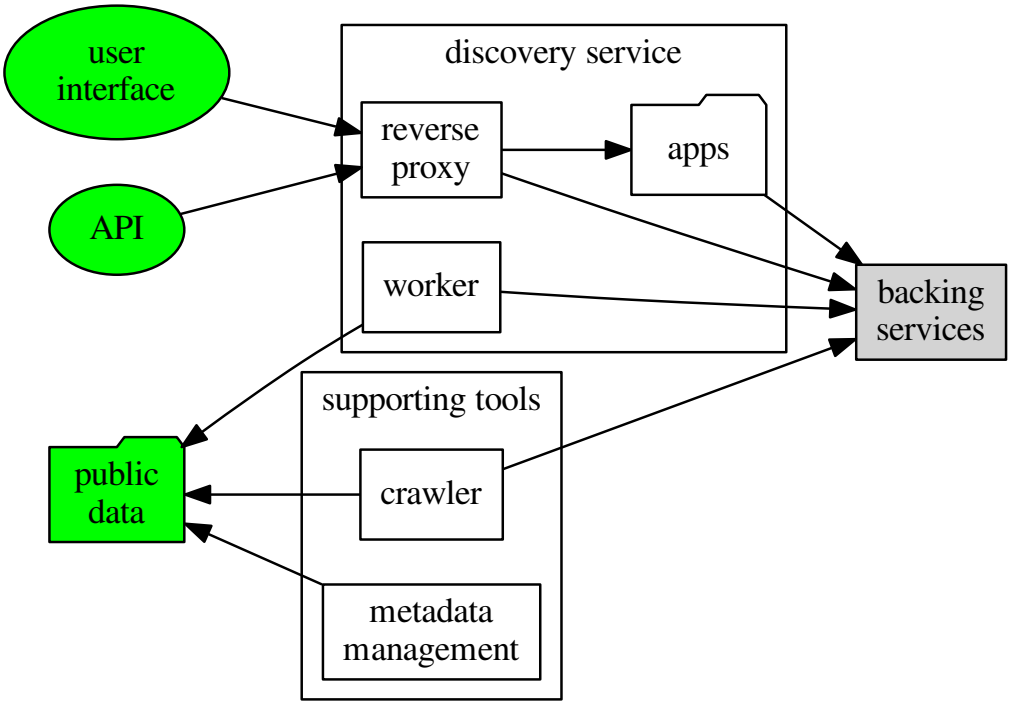
1.1 Copyright

1.2 Introduction

These are technical documents, they are only concerned with what and how. Specifics of who and when are contained in the git logs. This blog post explains why and where:

https://www.dto.gov.au/news-media/blog/making-government-discoverable

The user discovery later aims to provide useful features that enable users and 3rd party applications to discover government resources. It is currently in pre-ALPHA status, meaning a working technical assessment, not yet considered suitable for public use (even by "early-adopters").



TODO: define each box in the above diagram

1.3 Development

Discovery service:

- http://github.com/AusDTO/discoveryLayer Code
- http://github.com/AusDTO/discoveryLayer/issues Discussion
- http://waffle.io/AusDTO/discoveryLayer Kanban
- http://ausdto-discovery-layer.readthedocs.org/ Documentation

Crawler:

- http://github.com/AusDTO/disco_crawler Code
- http://github.com/AusDTO/disco_crawler/issues Discussion
- http://ausdto-disco-crawler.readthedocs.org/ Documentation

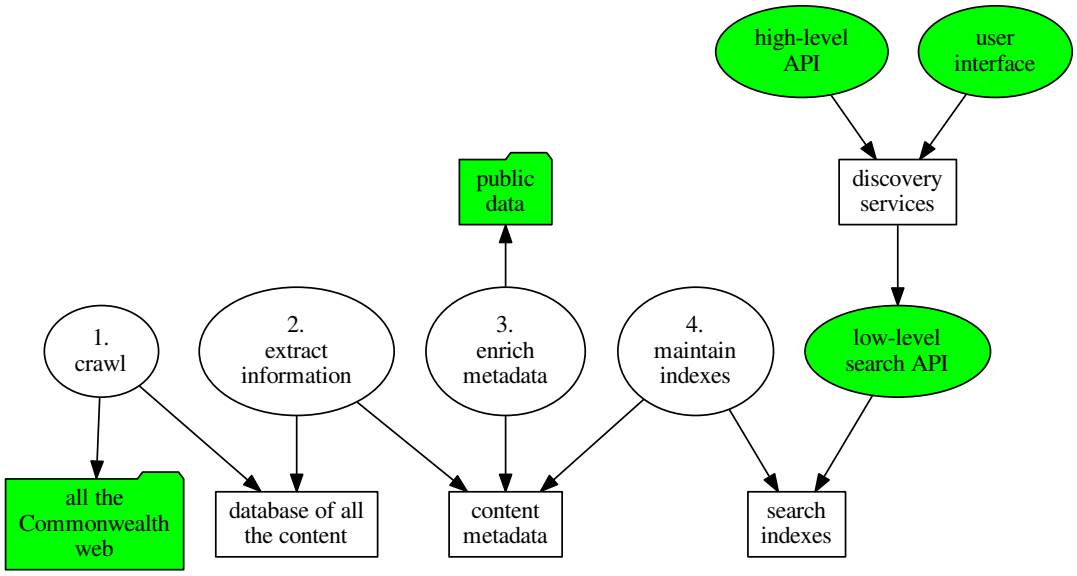
Metadata management (currently service catalogue):

- http://github.com/AusDTO/serviceCatalogue Code
- http://github.com/AusDTO/serviceCatalogue/issues Discussion
- http://ausdto-service-catalogue.readthedocs.org/ Documentation

CHAPTER 2

Design

The discovery layer is designed using the "pipeline" pattern. It processes public data (including all Commonwealth web sites) to produce a search indexes of enriched content metadata. These search indexes provide a public, low-level (native) search API, which is used by the discovery service to power user interface and high-level API features.



Pipeline:

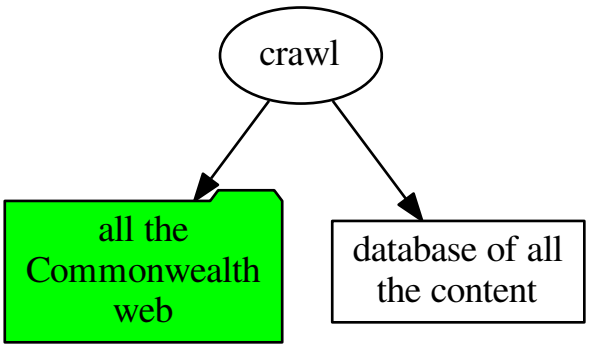
1. Crawl a database of content from the Commonwealth web.
2. Extract information into a metadata repository, from the content database.
3. Enrich content metadata using public data.
4. Maintain search indexes from content metadata.

2.1 Activities

In the above diagram, white ellipses represent activities performed by discovery layer components.

2.1.1 Crawling content

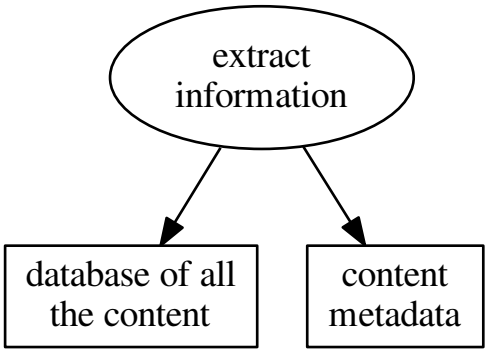
The crawler component is a stand-alone product located in it's own GitHub repository (https://github.com/AusDTO/disco_crawler). It suits our needs OK right now, but at some point we may replace it with a more sophistocated turnkey system such as apache nutch.



The crawler only visits Commonwealth resources (.gov.au domains, excluding state subdomains). The result of all that is that the database fills up with "all the Commonwealth resources", those resources are checked on a regulalar schedule and the database is updated when they change.

2.1.2 Information Extraction

The information extraction step is currently very simple. It ignores everything except html resources, and performs a simple "article extraction" using the python Goose library (https://pypi.python.org/pypi/goose-extractor).



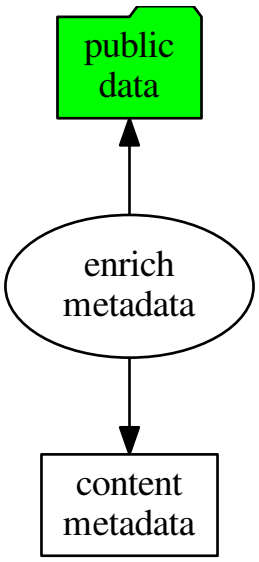
PDF article extraction is yet to be implemented, but shelling-out to the pdftotxt tool from Xpdf (http://www.foolabs.com/xpdf/download.html) might work OK. Encourageing results have been obtained from scanned PDF documents using Teseract (https://github.com/tesseract-ocr/tesseract),

The DBPedia open source project has some much more sophistocated information extraction features (http://dbpedia.org/services-resources/documentation/extractor) which may be relevent as new requirements emerge in this step. Specifically, their distributed extraction framework (https://github.com/dbpedia/distributed-extraction-framework) using Apache Spark seems pretty cool. This might be relevant to us if we wanted to try and

migrate or syncicate Commonwealth web content(however, this might not be fesible doe to the diversity of page structures that would need to be modelled).

2.1.3 Metadata enrichment

The metadata enrichment step combines the extracted information with aditional data from public sources. Currently this is limited to "information about government services" sourced from the service catalogue component.



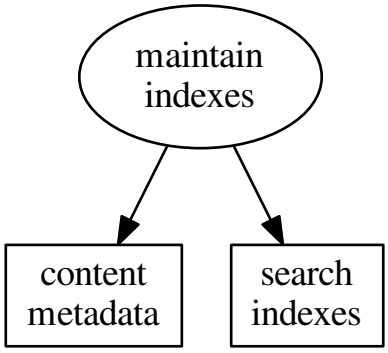
The design intent is that this enrichment step would draw on rich sources of knowledge about government services - essentially, releaving users of the burden of having to understand how the government is structured to access it's content.

Technically this would be when faceting data is incorporated; user journeys (scenarios), information architecture models, web site/page tagging and classification schemes, etc. This metadata might be manually curated/maintained (e.g. web site classification), automatically produced (e.g. natural language processing, automated clustering, web traffic analysis, semantic analysis, etc) or even folksonomically managed. AGLS metadata (enriched with synonyms?) might also be used to produce potentialy useful facets.

Given a feedback loops from passive behavior analysis (web traffic) or navigation choice-decision experiments (A-B split testing, ANOVA/MANOVA designs etc), information extraction could be treated as a behavior laboritory for creating value in search-oriented architecture at other layers. Different information extraction schemes (treatments) could be operated to produce/maintain parallel indexes, and discovery-layer nodes could be randomly assigned to indexes.

2.1.4 Index maintainance

The search indexes are maintained using the excellent django-haystack library (http://haystacksearch.org/). Specifically, using the asynchronous celery_haystack module (https://github.com/django-haystack/celery-haystack).

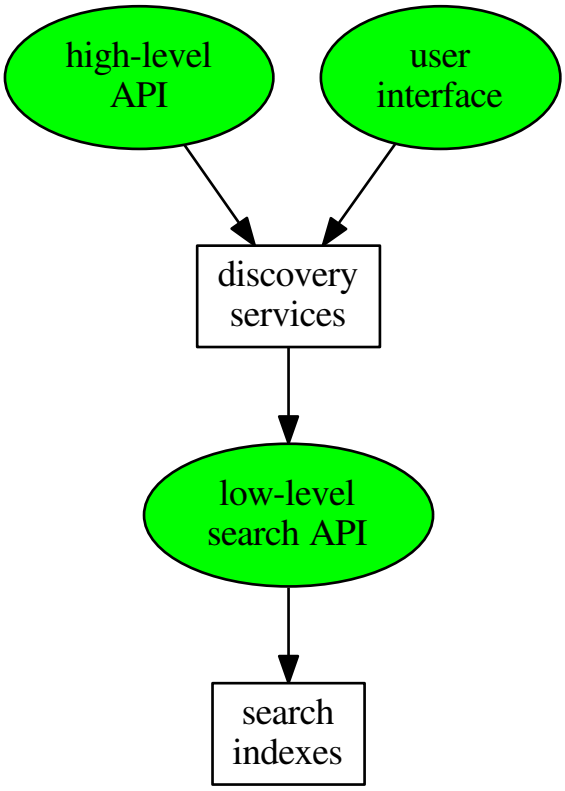


Using celery_haystack, index-management tasks are triggered by "save" signals on the ORM model that the index is based on. Because the crawler is NOT using the ORM, inserts/updates/deleted by the crawler do not automatically trigger these tasks. Instead, scheduled jobs compare content hash fields in the drawler's database and the metadata to detect differences and dispatch metadata updates apropriately.

Note: The US Digital GovSearch service is trying out a search index management feature called i14y (Beta, http://search.digitalgov.gov/developer/) to push CMS content changes to their search layer for reindexing.

That's a nice idea here too; furnish a callback API that dispatches change to the crawler schedule and metadata maintenance. Possibly the GovCMS solr inegration hooks could be extended...

2.2 Interfaces



In the above diagram, green ellipses represent interfaces. The colour green is used to indicate that the items are open for public access.

2.2.1 User interface

The discovery service **user interface** is a mobile-friendly web application. It is a place to impliment "consierge service" type features, that assist people locate government resources. The DEV team consideres it least likely to be important over the long term, but likely to be useful for demonstrations and proofs of concept.

These are imagined to be user-friendly features for finding (searching and/or browsing) Australian Government online resources. The current pre-ALPHA product does not have significant features here yet, because we are just entering "discovery phase" on that project (we are in the process of gathering evidence and analysing user needs).

In adition to conventional search features, the "search oriented architecture" paradigm contains a number of patterns (such as faceted browsing) that are likely to be worthy of experiment during ALPHA and BETA stages of development.

2.2.2 High-level API

The discovery service **high-level API** is a REST integration surface, designed to support/enable discoverability features in other applications (such as Commonwealth web sites). They are essentially wrappers that exploit the power of the low-level search API in a way that is convenient to users. The DEV team considers it highly-likely that signifacant value could be added at this layer.

Two kinds of high-level API features are considered likely to prove useful.

- Machine-consumable equivalents of the user-interface features
- Framework for content analysis

The first type of high-level API is simply a REST endpoint supporting json or xml format, 1:1 exact mapping of functionality. It should be useful for integrating 3rd party software with the discovery layer infrastructure.

The second type of high-level API is the python language interface provided by django-haystack, the framework used to interface and manage the search indexes. This API is used internally to make the first kind of API and the user interfaces. It's also potentially useful for extending the service with new functionality, and analytic use-cases (as evidenced by ipython notebook content analysis, TODO).

2.2.3 Low-level search API

The **low-level search API** is simply the read-only part of the native elasticsearch interface. It's our post-processed data, derived from public web pages and open data, using our open source code. We don't know if or how other people might use this interface, but would be delighted if that happened.

The search index backing service has a REST interface for GETing, POSTing, PUTing and DELETEing the contents of the index. The GET verbs of this interface is published directly through the reverse-proxy component of the discovery layer interface, allowing 3rd parties to reuse our search index (either with code based on our high-level python API, or any other software that supports the same kind of search index).

BETA version of the discovery layer probably requires throttling and/or other forms of protection from queries that would potentially degrade performance.

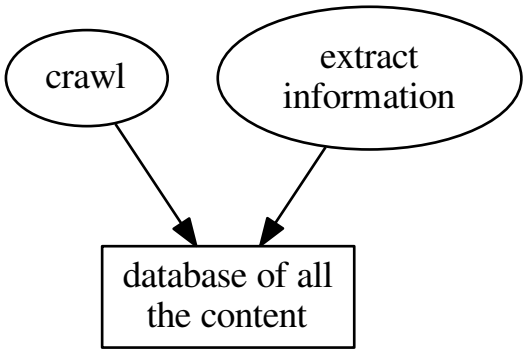
2.3 Components

In the diagrams on this page, ellipses are "verbish" (interfaces and activities) and rectangles are "nounish" (components of the discovery layer system).

2.3.1 Content database

Pipeline:

- Crawl a database of content from the Commonwealth web.
- Extract information into a metadata repository, from the content database.

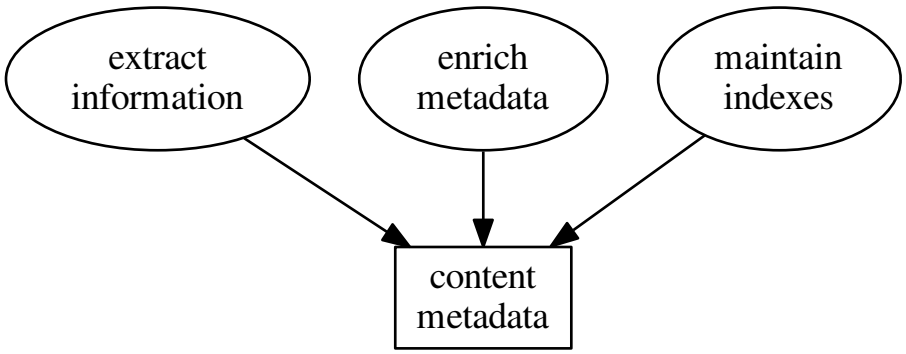


The content_database is shared with the disco_crawler component. Access from python is via the ORM wrapper in */crawler/models.py*. See also *crawler/tasks.py* for the synchronisation jobs that drive information extraction process.

2.3.2 Content metadata

Pipeline:

- Extract information into a metadata repository, from the content database.
- Enrich content metadata using public data.
- Maintain search indexes from content metadata.

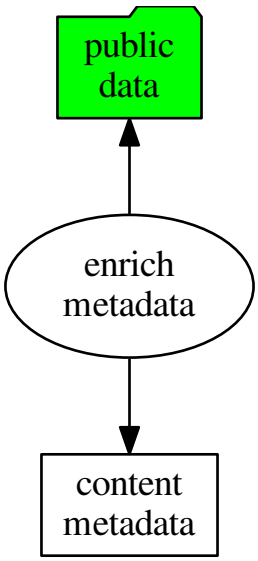


Content metadata is managed from python code through the django ORM layer (see *<app>/models.py* in the repo), primarially by asynchronous worker processes (celery tasks, see *<app>/tasks.py*).

2.3.3 Public data

Pipeline:

- Enrich content metadata using public data.



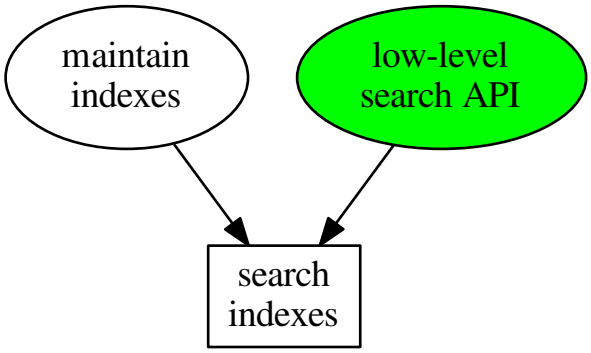
The initial design intent was to draw all public data from the CKAN API at data.gov.au, although any open public API would be OK.

Due to the nature of duct tape, chewing gum and number 8 wire employed in pre-alpha development, none of the data is currently being drawn from APIs at the moment. Currently it's only the service catalogue drawn from a repository hosted in github.com.

2.3.4 Search indexes

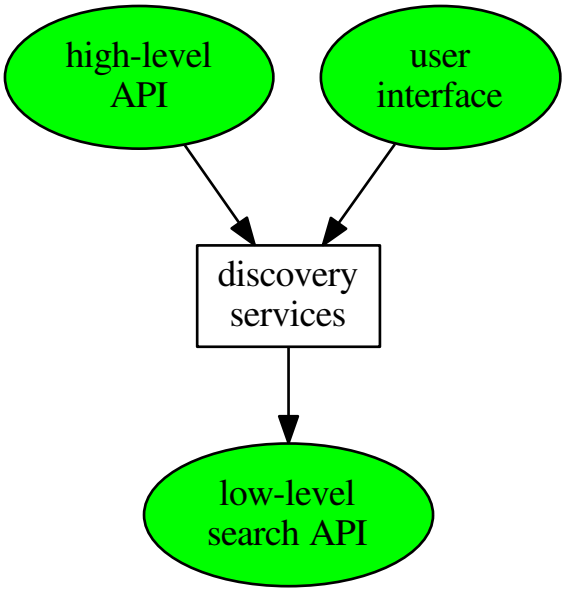
Pipeline:

- Maintain search indexes from content metadata.



Search indexes are currently ElasticSearch, although theoretically could be any index backend supported by django-haystack.

2.3.5 Discovery services



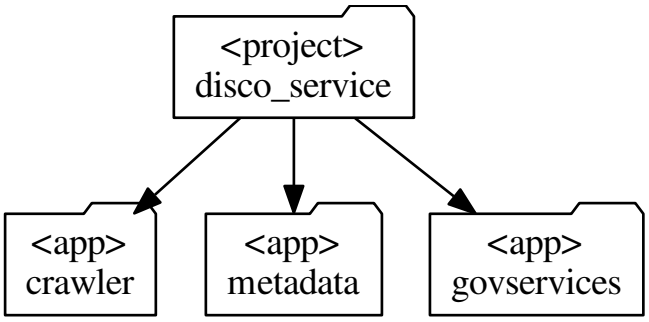
The disco services are implemented as python/django applications, run in a stateless wsgi container (gunicorn) behind a reverse proxy (nginx). Django is used to produce both the user interface (responsive web) and high-level API (REST).

See Dockerfile for specific details of how this is component is packaged, configured and run.

CHAPTER 3

Code

The code is organised into packages, in the standard django way.



The following documentation is incomplete (work in progress), for the timebeing it's better to reffer to the actual sources.

3.1 Package: disco_service

This is a django project, containing the usual settings.py, urls.py and wsgi.py

Note: Also contains *celery.py*, which is configuration for async worker nodes

3.2 Package: crawler

This django app is a simple wrapper. crawler app does not have an admin interface.

3.2.1 crawler.models

An ORM interface to the DB which is shared with the disco_crawler node.js app.

class `crawler.models.`**`WebDocument`**`(`**args*, ***kwargs*`)`
Resource downloaded by the disco_crawler node.js app.

The document attribute is a copy of the resource which was downloaded.

url uniquely defines the resource (there is no numeric primary key). host, path, port and protocol are attributes about the HTTP request used to retrieve the resource. lastfetchdatetime and nextfetchdatetime are heuristically determined and drive the behavior of the crawler. _hash is indexed and has a coresponding attribute in the metadata.Resource class (these are compared to determine if the metadata is dirty).

The rest of the attributes are derived from the content of the document.

3.2.2 crawler.tasks

This module contains integration tasks for synchronising this DB with the metadata used in the rest of the discovery layer.

`crawler.tasks.`**`sync_from_crawler`**`()`
dispatch metadata.Resource inserts for **new** crawler.WebDocuments

`crawler.tasks.`**`sync_updates_from_crawler`**`()`
dispatch metadata.Resource updates for **changed** crawler.WebDocuments

3.3 Package: metadata

This django app manages the content metadata.

3.3.1 metadata.models

class `metadata.models.`**`Resource`**`(`*args*, ***kwargs*`)`
ORM class wrapping persistent data of the web resource

Contains hooks into the code for resource processing

`_article``()`
Analyse resource content, return Goose interface

`_decode``()`
Lookup content of the coresponding WebDocument.document

`excerpt``()`
Attempt to produce a plain text version of resource content

`sr_summary``()`
Search result summary.

This is a rude hack, it doesn't even break on word boundaries. There should be much smarter ways of doing this.

`title``()`
Attempt to produce a single line description of the resource

3.3.2 metadata.tasks

`metadata.tasks.`**`insert_resource_from_row`**`()`
Wrap metadata.Resource constructor

Stupidly, doesn't even do any input validation.

`metadata.tasks.`**`update_resource_from_row`**`()`
ORM lookup then update

No input validation and foolishly assumes the lookup won't miss.

3.4 Package: govservices

This app wraps public data about government services.

3.4.1 govservices.models

class govservices.models.**Agency**(*id*, *acronym*)

exception DoesNotExist

exception Agency.**MultipleObjectsReturned**

Agency.**dimension_set**

Agency.**objects = <django.db.models.manager.Manager object>**

Agency.**service_set**

Agency.**subservice_set**

class govservices.models.**SubService**(*id*, *cat_id*, *desc*, *name*, *info_url*, *primary_audience*, *agency*)

exception DoesNotExist

exception SubService.**MultipleObjectsReturned**

SubService.**agency**

SubService.**objects = <django.db.models.manager.Manager object>**

class govservices.models.**ServiceTag**(*id*, *label*)

exception DoesNotExist

exception ServiceTag.**MultipleObjectsReturned**

ServiceTag.**objects = <django.db.models.manager.Manager object>**

ServiceTag.**service_set**

class govservices.models.**LifeEvent**(*id*, *label*)

exception DoesNotExist

exception LifeEvent.**MultipleObjectsReturned**

LifeEvent.**objects = <django.db.models.manager.Manager object>**

LifeEvent.**service_set**

class govservices.models.**ServiceType**(*id*, *label*)

exception DoesNotExist

exception ServiceType.**MultipleObjectsReturned**

ServiceType.**objects = <django.db.models.manager.Manager object>**

ServiceType.**service_set**

class govservices.models.**Service**(*id*, *src_id*, *agency*, *old_src_id*, *json_filename*, *info_url*, *name*, *acronym*, *tagline*, *primary_audience*, *analytics_available*, *incidental*, *secondary*, *src_type*, *description*, *comment*, *current*, *org_acronym*)

exception DoesNotExist

exception Service.**MultipleObjectsReturned**

Service.**agency**

Service.**life_events**

Service.**objects = <django.db.models.manager.Manager object>**

Service.**service_tags**

Service.**service_types**

class govservices.models.**Dimension**(*id*, *dim_id*, *agency*, *name*, *dist*, *desc*, *info_url*)

exception DoesNotExist

exception Dimension.**MultipleObjectsReturned**

Dimension.**agency**

Dimension.**objects = <django.db.models.manager.Manager object>**

3.4.2 govservices.tests

Suite of tests assuring that the code which manipulates govservices is working correctly.

3.4.3 govservices.management.commands.update_servicecatalogue

It would be highly preferable to refactor this to use a REST API to interrogate the service catalogue, rather than messing about with the ServiceJsonRepository.

class govservices.management.commands.update_servicecatalogue.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)

manage.py extension. Call with:

python manage.py update_servicecatalogue

or:

python manage.py update_servicecatalogue <entity>

where <entity> is the name of one of the classes in metadata.models

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

c

d

g

m

Symbols

A

C

D

E

G

I

L

M

O

R

S

T

U

W