Substance D Documentation

Release 0.0

Repoze Developers

November 07, 2018

Contents

1 Overview 1

2 Installation 3

3 Demonstration Application 5

4 Narrative Documentation 7

5 API Documentation 75

6 Support / Reporting Bugs / Development Versions 125

7 Copyright, Trademarks, and Attributions 127

8 Indices and tables 129

Python Module Index 131

CHAPTER 1

Overview

Substance D is an application server built using the *Pyramid* web framework. It can be used as a base to build a general-purpose web application like a blog, a shopping cart application, a scheduling application, or any other web app that requires both an administration and a retail interface.

Substance D owes much of its spirit to the *Zope* application server.

It requires Python 2.7, 3.4, or 3.5.

CHAPTER 2

Installation

See *Installation*.

CHAPTER 3

Demonstration Application

See the application running at http://demo.substanced.net for a demonstration of the Substance D management interface.

CHAPTER 4

Narrative Documentation

4.1 Installation

Install using pip, e.g., (within a virtual environment).

```
$ pip install substanced
```

Warning: During Substance D's alpha period, it may be necessary to use a checkout of Substance D as well as checkouts of the most recent versions of the libraries upon which Substance D depends.

4.1.1 Demonstration application

See the application running at http://demo.substanced.net for a demonstration of the Substance D management interface.

You can deploy the demonstration application locally by performing the following steps.

1. Create a new directory somewhere and `cd` to it:

   ```
   $ virtualenv -p python2.7 hack-on-substanced
   $ cd hack-on-substanced
   ```

2. Install Substance D either from PyPI or from a git checkout:

   ```
   $ bin/pip install substanced
   ```

   OR:

   ```
   $ bin/pip install -e git+https://github.com/Pylons/substanced#egg=substanced
   ```

   Alternatively create a writeable fork on GitHub and check that out.

3. Check that the python-magic library has been installed:

   ```
   $ bin/python -c "from substanced.file import magic; assert magic is not None,
   ↪'python-magic not installed'"
   ```

   If you then see "python-magic not installed" then you will need to take additional steps to install the python-magic library. See *Installing python-magic*.

4. Move back to the parent directory:

   ```
   $ cd ..
   ```

5. Now you should be able to create new Substance D projects by using `pcreate`. The following `pcreate` command uses the scaffold `substanced` to create a new project named `myproj`:

   ```
   $ hack-on-substanced/bin/pcreate -s substanced myproj
   ```

6. Now you can make a virtual environment for your project and move into it:

   ```
   $ virtualenv -p python2.7 myproj
   $ cd myproj
   ```

7. Install that project using `pip install -e` into the virtual environment:

   ```
   $ bin/pip install -e .
   ```

8. Run the resulting project via `bin/pserve development.ini`. The development server listens to requests sent to http://0.0.0.0:6543 by default. Open this URL in a web browser.

9. The initial Administrator password is randomly generated automatically. Use the following command to find the login information:

   ```
   $ grep initial_password *.ini
   development.ini:substanced.initial_password = hNyrGI5nnl
   production.ini:substanced.initial_password = hNyrGI5nnl
   ```

4.1.2 Create a project from a scaffold in Substance D

After creating a development checkout, you can create a new project from the default substanced scaffold by using `pcreate`.

```
$ cd ../env
$ bin/pcreate -s substanced myproj
```

Then install that project using `pip install -e .` into the virtual environment.

```
$ cd myproj
$ ../bin/pip install -e .
```

Run the resulting project.

```
$ ../bin/pserve development.ini
```

Then start hacking on your new project.

4.1.3 Hacking on Substance D

See Hacking on Substance D, or look in your checked out local git repository for `HACKING.txt`, for information and guidelines to develop your application, including testing and internationalization.

4.2 Introduction

A Scanner Darkly

“The two hemispheres of my brain are competing?” Fred said.

“Yes.”

“Why?”

“Substance D. It often causes that, functionally. This is what we expected; this is what the tests confirm. Damage having taken place in the normally dominant left hemisphere, the right hemisphere is attempting to compensate for the impairment. But the twin functions do not fuse, because this is an abnormal condition the body isn’t prepared for. It should never happen. “Cross-cuing”, we call it. Related to splitbrain phenomena. We could perform a right hemispherectomy, but–”

“Will this go away,” Fred interrupted, “when I get off Substance D?”

“Probably,” the psychologist on the left said, nodding. “It’s a functional impairment.”

The other man said, “It may be organic damage. It may be permanent. Time’ll tell, and only after you are off Substance D for a long while. And off entirely.”

“What?” Fred said. He did not understand the answer– was it yes or no? Was he damaged forever or not? Which had they said?

– Philip K. Dick, A Scanner Darkly

Substance D is an application server. It provides the following features:

- Facilities that allow developers to define “content” (e.g., “a blog entry”, “a product”, or “a news item”).
- A management (a.k.a., “admin”) web UI which allows non-expert but privileged users to create, edit, update, and delete developer-defined content, as well as managing other aspects of the system such as users, groups, security, and so on.
- “Undo” capability for actions taken via the management UI.
- A way to make highly granular hierarchical security declarations for content objects (e.g., “Bob can edit *this* post” or “Bob can edit all posts in this collection” as opposed to just “Bob can edit posts”).
- Built-in users and groups management.
- Built-in content workflow.
- Indexing and searching of content (field, keyword, facet, and full-text). Searches can be easily filtered by the security settings of individual pieces of content, by path in the content hierarchy, or by a combination of the two.
- A facility for relating content objects to each other (with optional referential integrity).
- An “evolve” mechanism for evolving content over time as it changes.
- A mechanism to dump your site’s content to the filesystem in a mostly human-readable format, and a mechanism to reload a dump into the system.

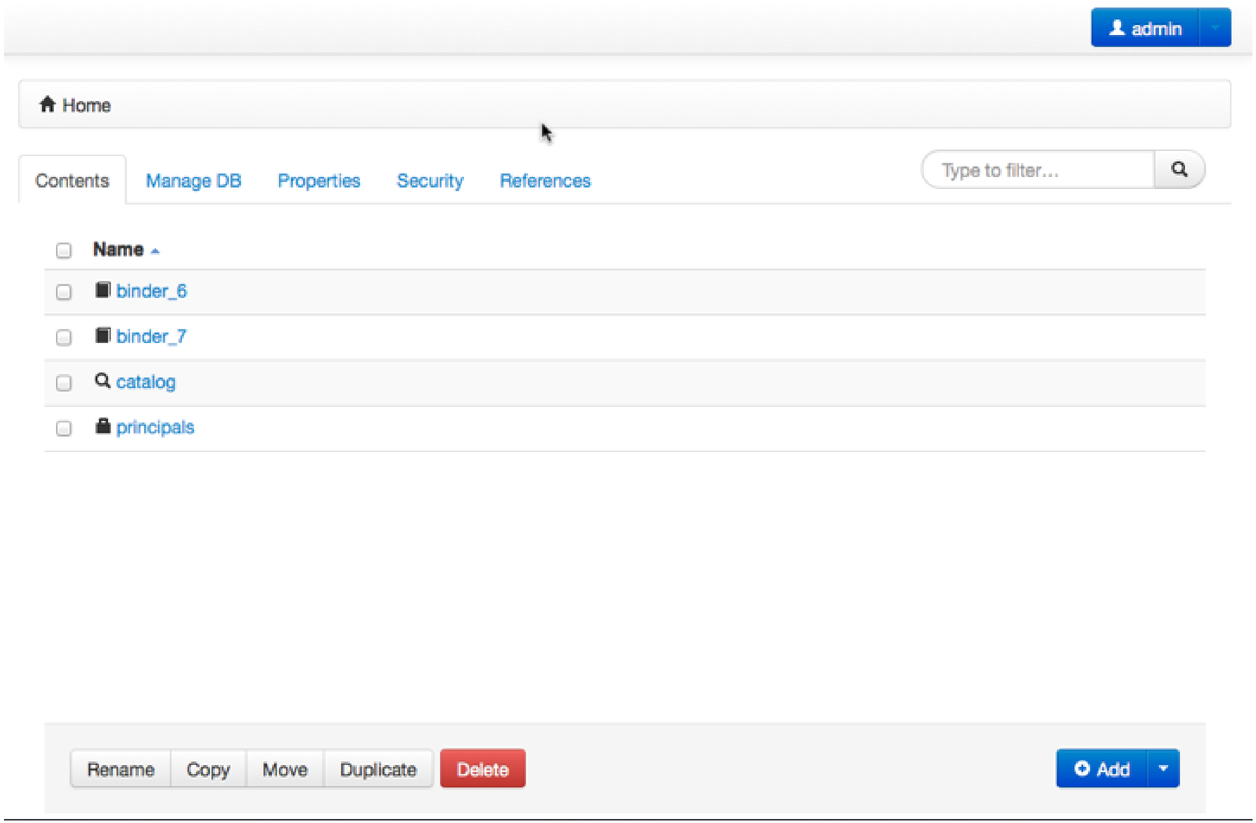
Substance D is built upon the following technologies:

- ZODB
- Pyramid
- hypatia
- colander
- deform

4.3 The Substance D Management Interface

Substance D's prime directive is to help developers quickly build custom content management applications with a custom user experience. For the Substance D parts, though, a polished, stable, and supported management UI is provided.

The Substance D management interface (aka *SDI*) is a set of view registrations that are imposed upon the *resource tree* of your application. The SDI allows you to add, delete, change, and otherwise manage resources and services.



4.3.1 Benefits and features

- Create, Read, Update, Delete (CRUD) operations on content resources
- Extensible actions for each content type via management views
- Built-in support for hierarchies with security
- Already-done UIs for all supported features (e.g., references, principals)

- Undo facility to back out of the last transaction
- Copy and paste

4.3.2 Background and motivation

In prehistoric times there was a Python-based application server, derived from a commercial predecessor released in 1996. Zope and its predecessor had a unique "through-the-web" (TTW) UI for interacting with the system. This UI, called the "Zope Management Interface" (ZMI), had a number of capabilities for a number of audiences. Plone, built on Zope, extended this idea. Other systems, such as Django, have found that providing an out-of-the-box (OOTB) starting point with attractive pixels on the screen can be a key selling point.

Substance D taps into this. In particular, lessons learned from our long experience in this area are applied to the SDI:

- Attractive, official, supported OOTB management UI
- Be successful by being very clear what the SDI *isn't*

4.3.3 What is and isn't the SDI

The SDI is for:

- Developers to use while building their application
- Administrators to use after deployment, to manage certain Substance D or application settings provided by the developer
- Certain power users to use as a behind-the-scenes UI

The SDI is *not* for:

- The *retail UI* for your actual application. Unlike Plone, we don't expect developers to squeeze their UX expectations into an existing UX
- Overridable, customizable, replaceable, frameworky new expectations

The SDI does have a short list of clearly-defined places for developers to plug in parts of their application. As a prime example, you can define a *Management View* that gets added as a new tab on a resource.

The SDI is extensible and allows you to plug your own views into it, so you can present nontechnical users with a way to manage arbitrary kinds of custom content.

Once again, for clarity: the SDI is not a framework, it is an application. It is not your retail UI.

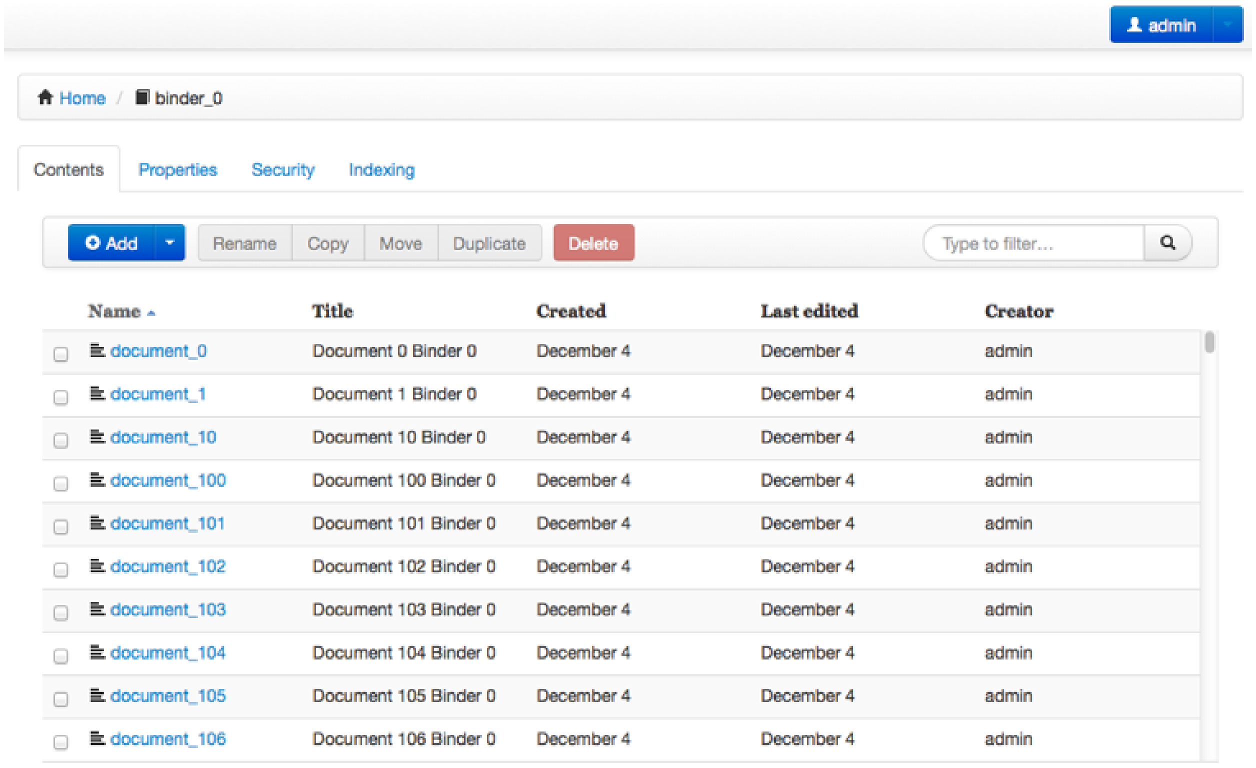
4.3.4 Layout

The SDI has a mostly-familiar layout:

- A *header* that shows the username as a dropdown menu containing a link to the principal screen as well as a logout link
- *Breadcrumbs* with a path from the root
- A series of *tabs* for the management views of the current resource
- Optionally, a *flash message* showing results of the previous operation, a warning, or some other notice
- A *footer*

4.3.5 Folder contents

Folders show a listing of items they contain using a powerful data grid based on SlickGrid:



	Name	Title	Created	Last edited	Creator
☐	document_0	Document 0 Binder 0	December 4	December 4	admin
☐	document_1	Document 1 Binder 0	December 4	December 4	admin
☐	document_10	Document 10 Binder 0	December 4	December 4	admin
☐	document_100	Document 100 Binder 0	December 4	December 4	admin
☐	document_101	Document 101 Binder 0	December 4	December 4	admin
☐	document_102	Document 102 Binder 0	December 4	December 4	admin
☐	document_103	Document 103 Binder 0	December 4	December 4	admin
☐	document_104	Document 104 Binder 0	December 4	December 4	admin
☐	document_105	Document 105 Binder 0	December 4	December 4	admin
☐	document_106	Document 106 Binder 0	December 4	December 4	admin

This dynamic grid features:

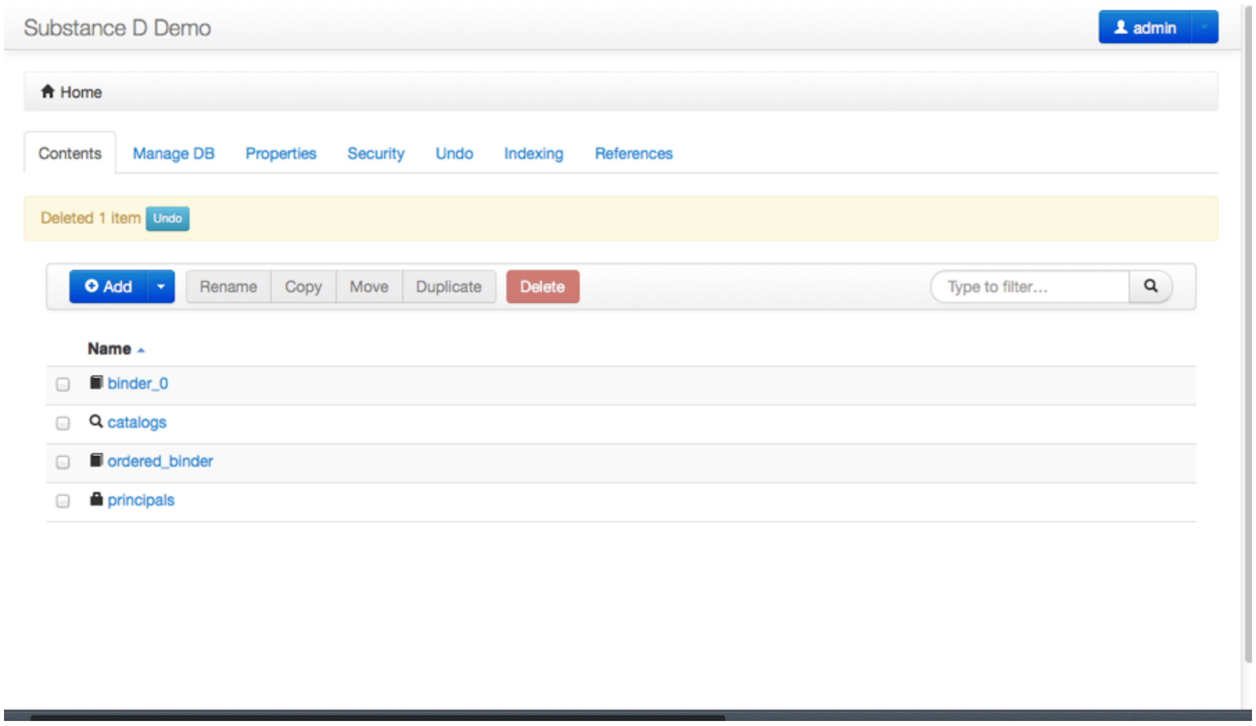
- Only loading the items needed for display, to speed up operations on large folders
- "Infinite scrolling" via the scrollbar to go directly to a batch at any point in the folder
- Column resizing and re-ordering
- Sorting on sort-supported columns
- Filtering based on search string
- Selection of one or more items and performing an operation by clicking on a button
- Styling integrated with Twitter Bootstrap
- Detection and re-layout on responsive design operations

The *Configuring Folder Contents* chapter covers how Substance D developers can plug their custom content types into folder contents.

4.3.6 Undo

In Substance D, many transactions can be undone and redone after a commit. This "Undo" ability is one of the key features that people notice immediately and it has real, deep value to a developer's customers.

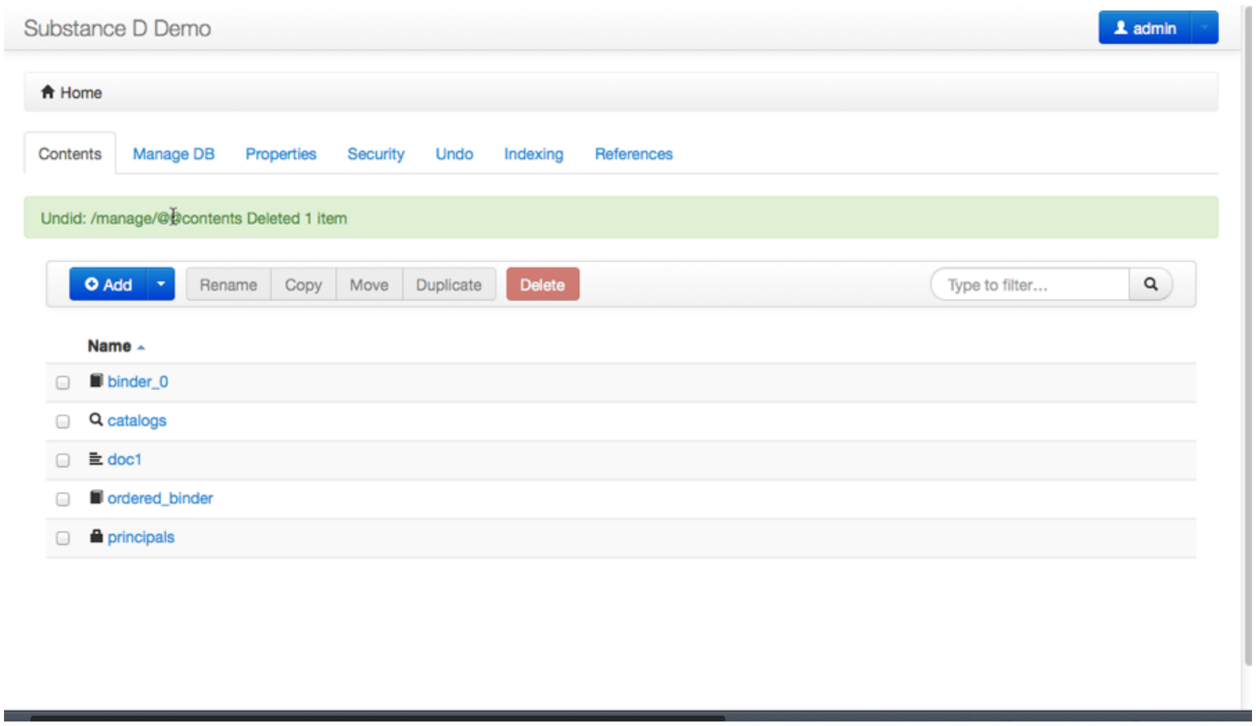
Many of the built-in operations display an undo button. For example, if we delete an item from a folder, we get a "flash" message telling us the deletion was performed, but with a button allowing us to undo it if that was a mistake:



Deleted 1 item Undo

binder_0
catalogs
ordered_binder
principals

Clicking "undo" restores the deleted item, with a flash message offering to redo the undo:

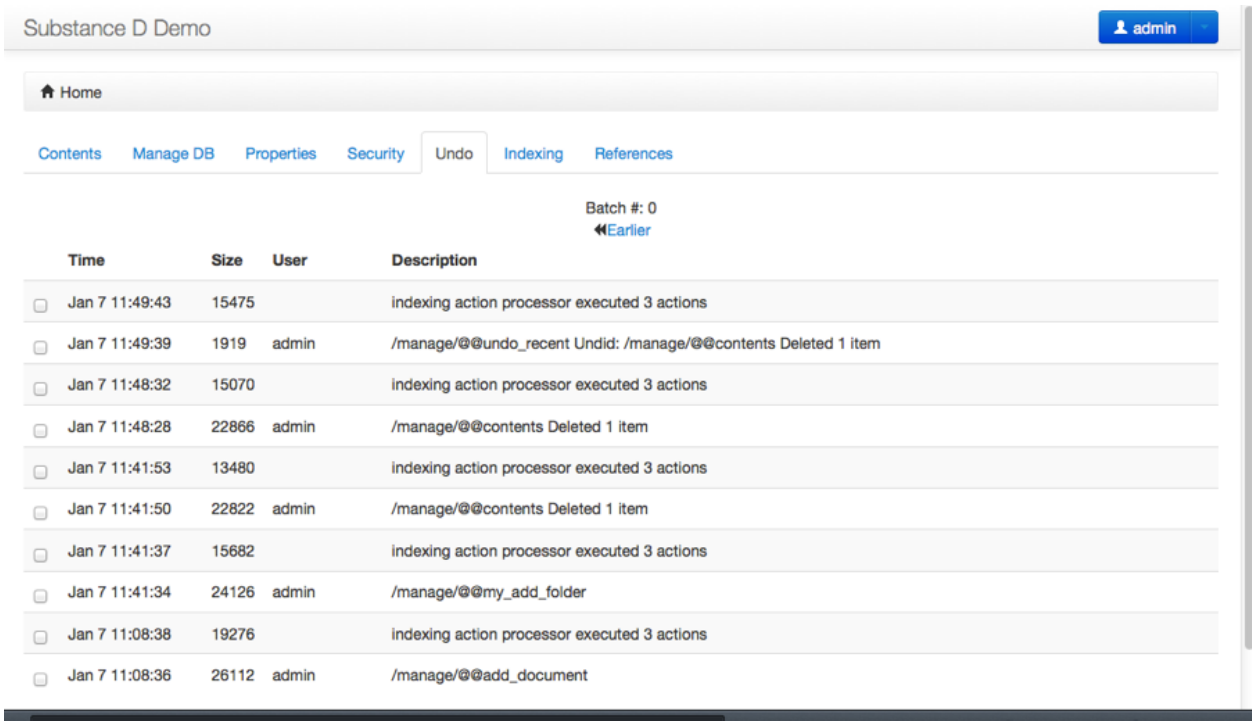


Undid: /manage/@@contents Deleted 1 item

binder_0
catalogs
doc1
ordered_binder
principals

Undo button support is enabled by the developer in their management views that commit data. It isn't available on every kind of change. Instead developers need to wrap their commit with certain information used by the SDI's undo features.

All actions that change data (even ones without undo button support) can be undone. These screenshots show an `Undo` tab on the site's root folder. This provides a global way to see recent transactions and perform an undo:

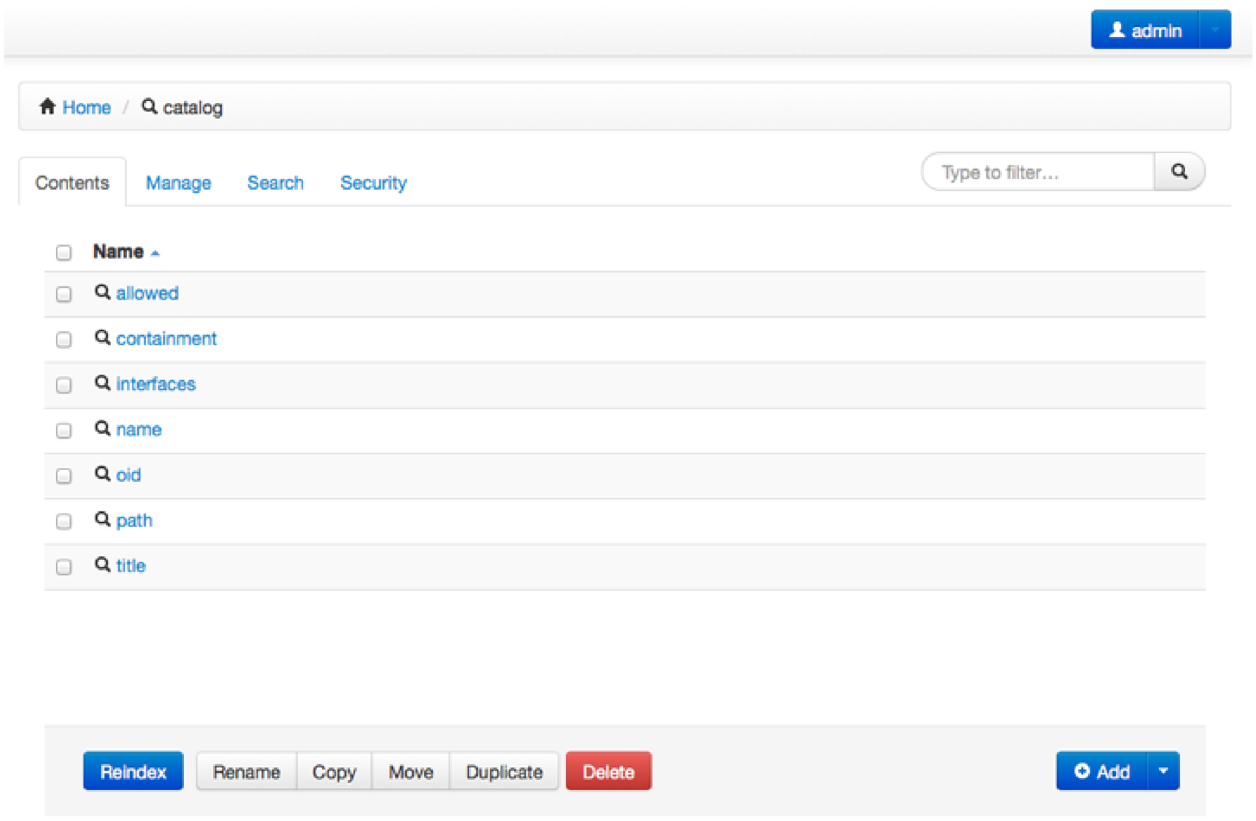


	Time	Size	User	Description
▢	Jan 7 11:49:43	15475		indexing action processor executed 3 actions
▢	Jan 7 11:49:39	1919	admin	/manage/@@undo_recent Undid: /manage/@@contents Deleted 1 item
▢	Jan 7 11:48:32	15070		indexing action processor executed 3 actions
▢	Jan 7 11:48:28	22866	admin	/manage/@@contents Deleted 1 item
▢	Jan 7 11:41:53	13480		indexing action processor executed 3 actions
▢	Jan 7 11:41:50	22822	admin	/manage/@@contents Deleted 1 item
▢	Jan 7 11:41:37	15682		indexing action processor executed 3 actions
▢	Jan 7 11:41:34	24126	admin	/manage/@@my_add_folder
▢	Jan 7 11:08:38	19276		indexing action processor executed 3 actions
▢	Jan 7 11:08:36	26112	admin	/manage/@@add_document

Sometimes a particular transaction cannot be undone without undoing an earlier transaction. For example, if you make three changes to a resource, the first two can't be undone without first undoing the last, as the resource will have been changed by a later transaction.

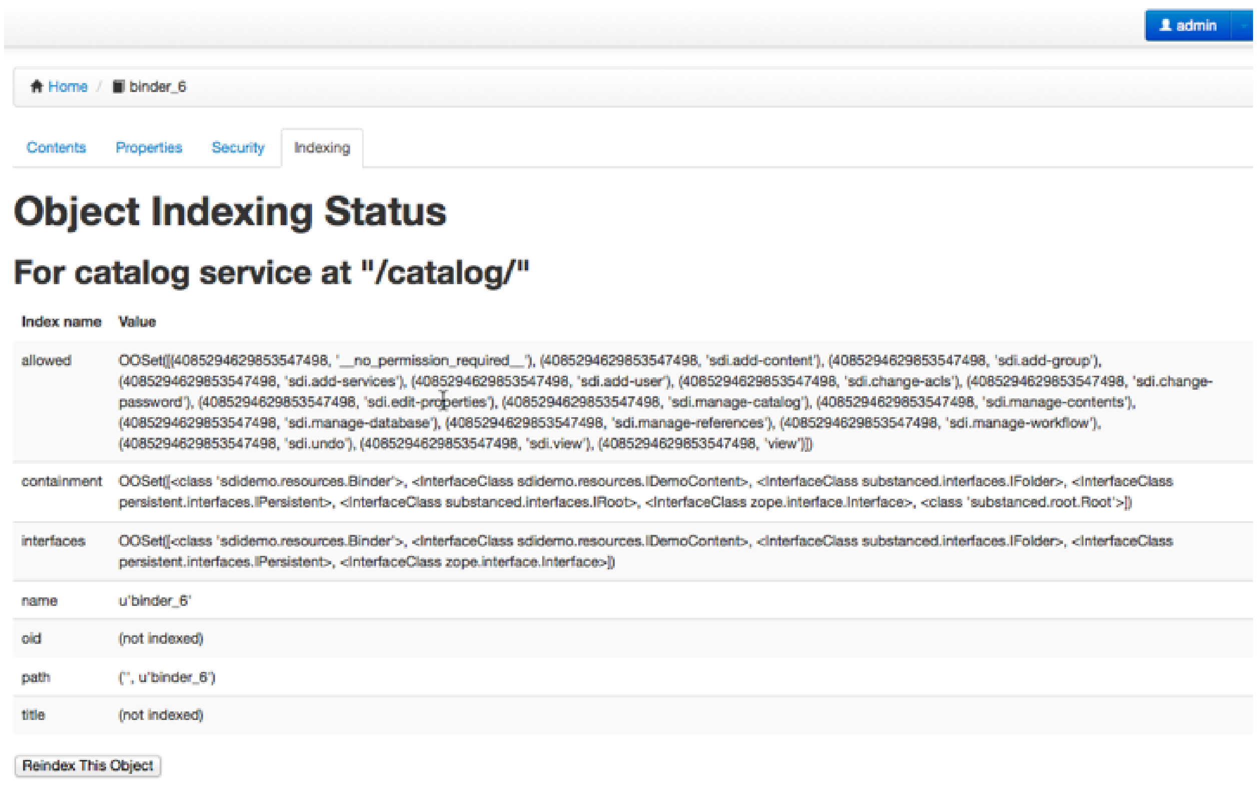
4.3.7 Catalog

With *cataloging* developers have a powerful indexing and searching facility that can be added to their application. Like other first-class parts of Substance D's machinery, it includes an SDI UI for interacting with the catalog:



Catalogues are content, meaning they show up as folder items in the SDI. You can visit a catalog and update its indexes, or see some statistics for those indexes. You can also use the SDI to reindex the contents of an index, if you suspect it has gotten out of sync with the content.

The catalog also registers a management view on content resources, which gain an `Indexing` tab:



admin

Home / binder_6

Contents Properties Security Indexing

Object Indexing Status

For catalog service at "/catalog/"

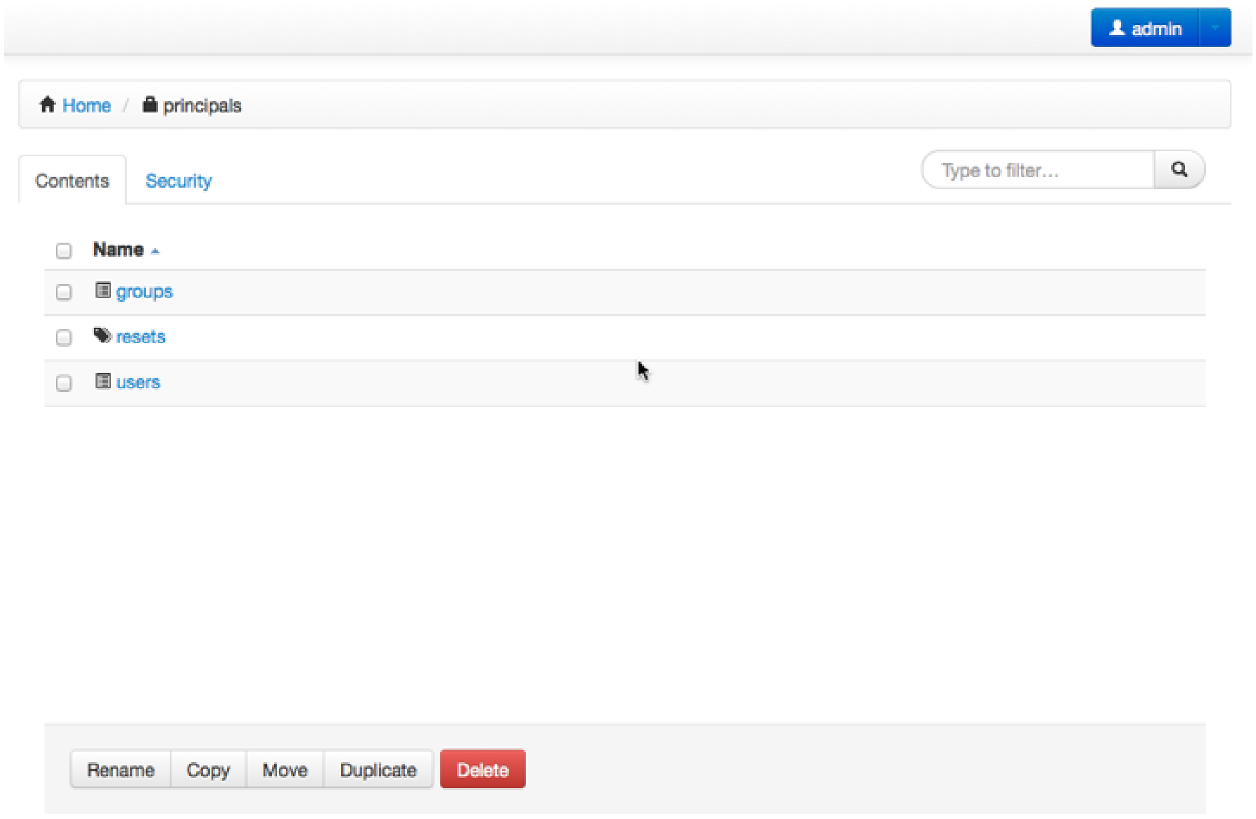
Index name	Value
allowed	OOSet([(4085294629853547498, '__no_permission_required__'), (4085294629853547498, 'sdi.add-content'), (4085294629853547498, 'sdi.add-group'), (4085294629853547498, 'sdi.add-services'), (4085294629853547498, 'sdi.add-user'), (4085294629853547498, 'sdi.change-acls'), (4085294629853547498, 'sdi.change-password'), (4085294629853547498, 'sdi.edit-properties'), (4085294629853547498, 'sdi.manage-catalog'), (4085294629853547498, 'sdi.manage-contents'), (4085294629853547498, 'sdi.manage-database'), (4085294629853547498, 'sdi.manage-references'), (4085294629853547498, 'sdi.manage-workflow'), (4085294629853547498, 'sdi.undo'), (4085294629853547498, 'sdi.view'), (4085294629853547498, 'view')])
containment	OOSet([<class 'sdidemo.resources.Binder'>, <InterfaceClass sdidemo.resources.IDemoContent>, <InterfaceClass substanced.interfaces.IFolder>, <InterfaceClass persistent.interfaces.IPersistent>, <InterfaceClass substanced.interfaces.IRoot>, <InterfaceClass zope.interface.Interface>, <class 'substanced.root.Root'>])
interfaces	OOSet([<class 'sdidemo.resources.Binder'>, <InterfaceClass sdidemo.resources.IDemoContent>, <InterfaceClass substanced.interfaces.IFolder>, <InterfaceClass persistent.interfaces.IPersistent>, <InterfaceClass zope.interface.Interface>])
name	u'binder_6'
oid	(not indexed)
path	('', u'binder_6')
title	(not indexed)

Reindex This Object

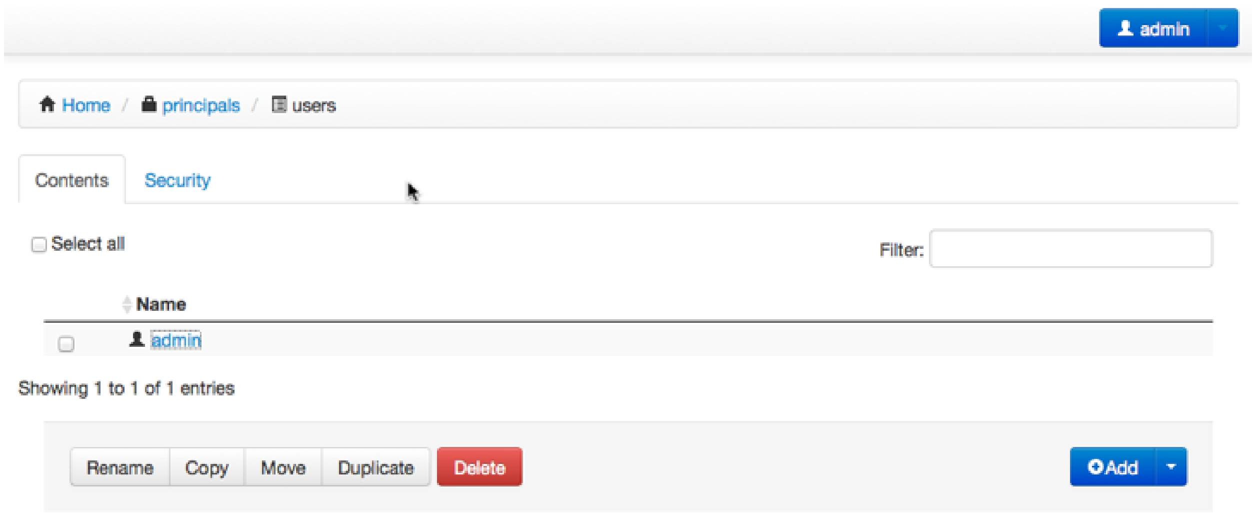
This shows some statistics and allows an SDI user to reindex an individual resource.

4.3.8 Principals

Managing users and groups, a.k.a., principals, is more interesting in a system like Substance D with rich hierarchies. You can add a folder of principals to any folder or other kind of container that allows adding principals:



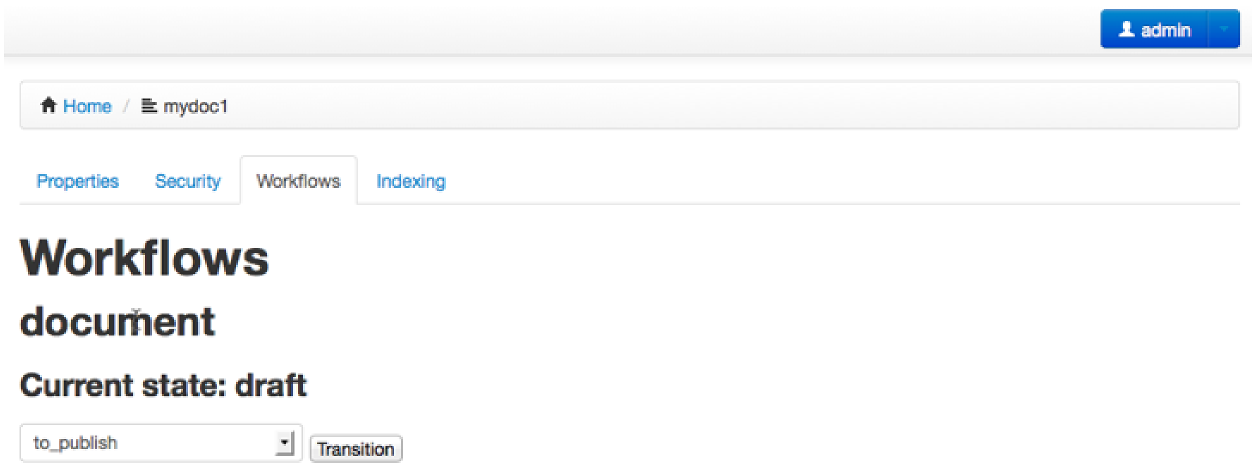
A principals folder allows you to manage (e.g., add, edit, delete, or rename) users and groups via the SDI, as well as password resets. Since users and groups are content, you gain some of the other SDI tabs for managing them (e.g., Security, References):



Users and groups can also grow extra attributes and behavior because they're just content, so you can customize your user model out of the box.

4.3.9 Workflows

The *workflows service* provides a powerful system for managing states and transitions. This service shows up in the SDI as a tab on content types that have workflows registered for them:



Workflows

document

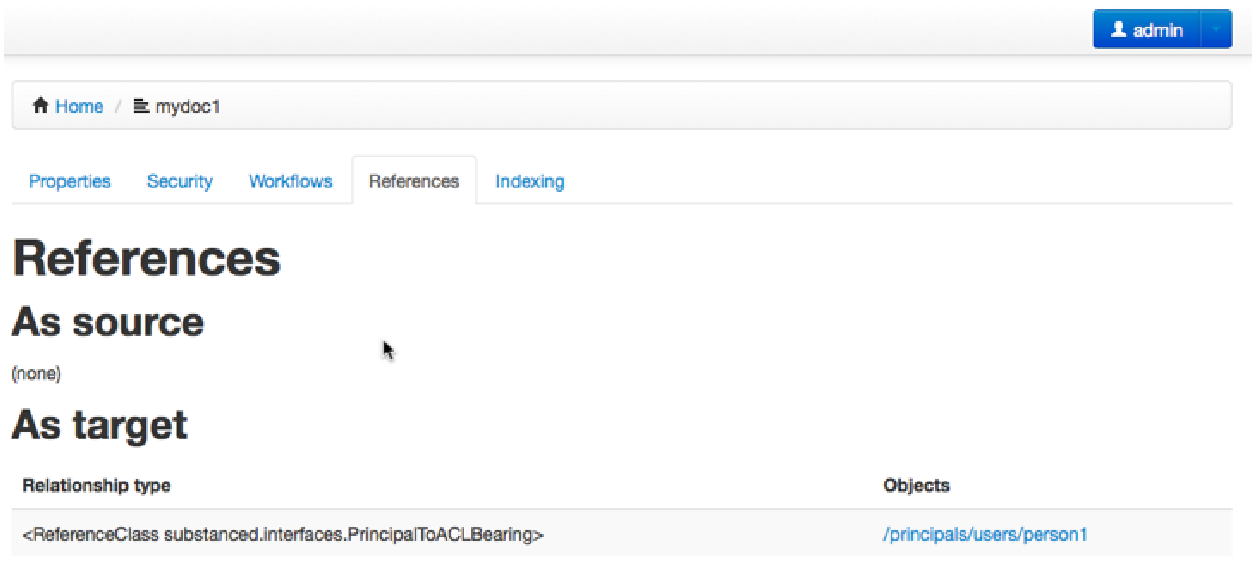
Current state: draft

This provides a way, via the SDI, to transition the workflow state of a resource.

4.3.10 References

With the built-in support for *references*, Substance D helps manage relationships between resources. The SDI provides a UI into the reference service.

If the resource you are viewing has any references, a `References` tab will appear:



References

As source

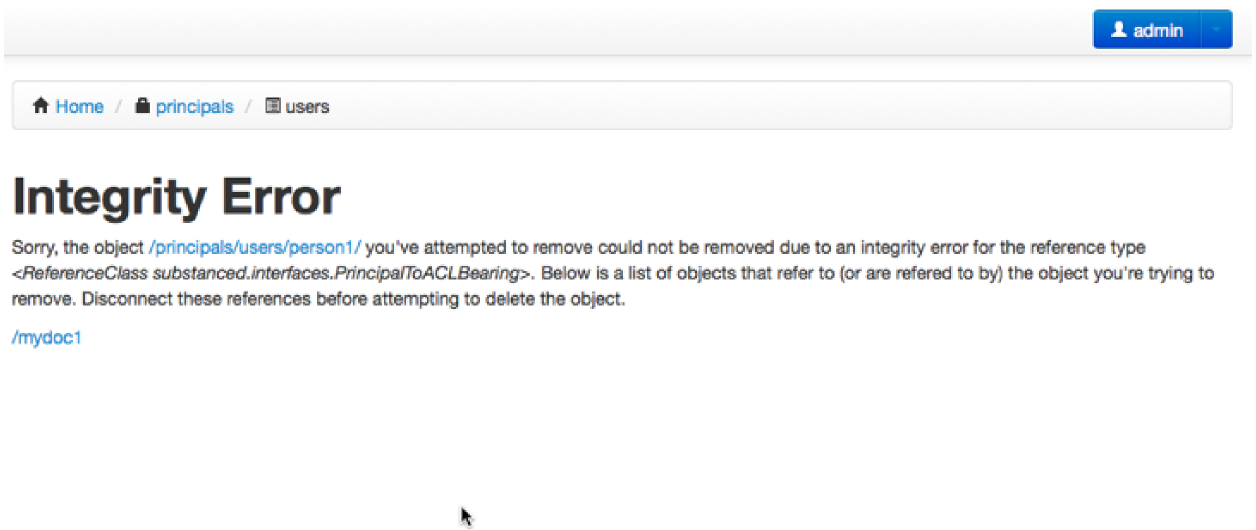
(none)

As target

Relationship type	Objects
<ReferenceClass substanced.interfaces.PrincipalToACLBearing>	/principals/users/person1

In this example, `mydoc1` is a target of an ACL reference from the `admin1` user.

An integrity error can occur if you try to delete a source or target of a reference that claims to be "integral". The SDI will then show this with an explanation:



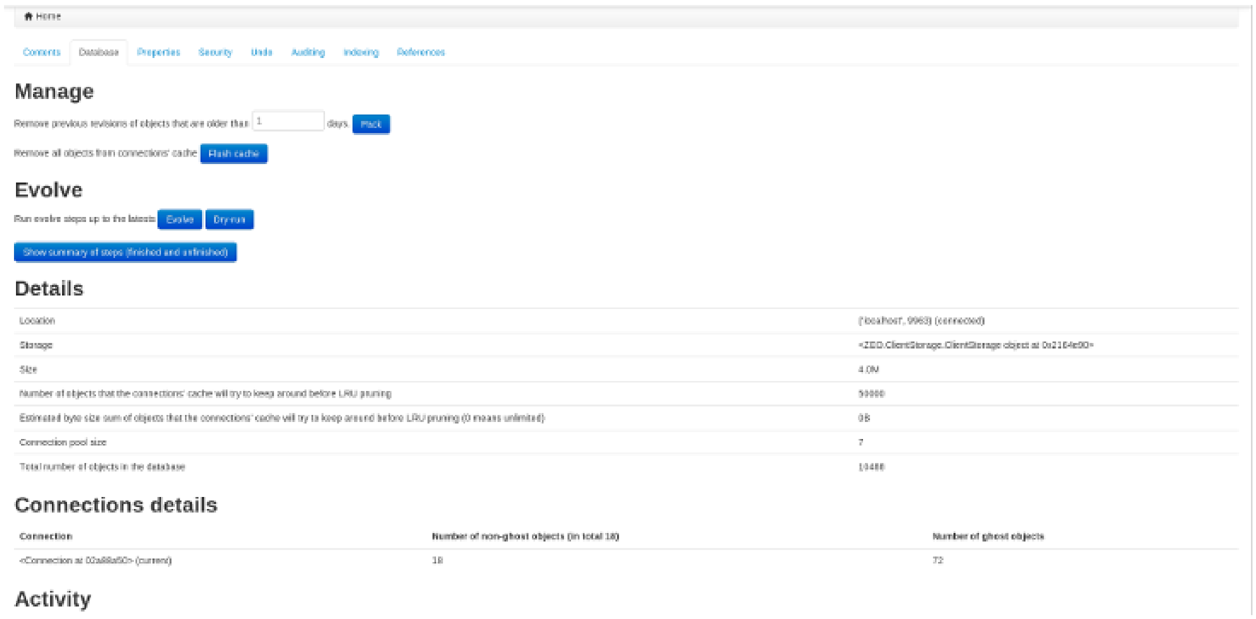
Integrity Error

Sorry, the object /principals/users/person1/ you've attempted to remove could not be removed due to an integrity error for the reference type <*ReferenceClass substanced.interfaces.PrincipalToACLBearing*>. Below is a list of objects that refer to (or are refered to by) the object you're trying to remove. Disconnect these references before attempting to delete the object.

/mydoc1

4.3.11 Manage Database

The object database inside Substance D has some management knobs that can be adjusted via the SDI:



Manage

Remove previous revisions of objects that are older than 1 days.

Remove all objects from connections' cache

Evolve

Run evolve steps up to the latest:

Details

Location	('localhost', 9963) (connected)
Storage	<ZEO.ClientStorage.ClientStorage object at 0x2164e90>
Size	4.0M
Number of objects that the connections' cache will try to keep around before LRU pruning	50000
Estimated byte size sum of objects that the connections' cache will try to keep around before LRU pruning (0 means unlimited)	0B
Connection pool size	7
Total number of objects in the database	10488

Connections details

Connection	Number of non-ghost objects (in total 18)	Number of ghost objects
<Connection at 02a88a50> (current)	18	72

Activity

This tab appears on the root object of the site and lets you:

- Pack the old revisions of objects in the database.

- Inspect and run evolution steps.
- Flush the object cache.
- See details and statistics about the database, the connection, and activity.

4.3.12 Implementation Notes

While it doesn't matter for developers of Substance D applications, some notes are below, regarding how the SDI is implemented:

- We use a high-performance, modern, responsive UI based on Twitter Bootstrap
- We use the upstream LESS variables from Bootstrap in a LESS file for parts of the SDI.
- Our grid is based on SlickGrid.

4.4 Defining Content

Resource is the term that Substance D uses to describe an object placed in the *resource tree*.

Ideally, all resources in your resource tree will be *content*. "Content" is the term that Substance D uses to describe resource objects that are particularly well-behaved when they appear in the SDI management interface. The Substance D management interface (aka *SDI*) is a set of views imposed upon the resource tree that allow you to add, delete, change and otherwise manage resources.

You can convince the management interface that your particular resources are content. To define a resource as content, you need to associate a resource with a *content type*.

4.4.1 Registering Content

In order to add new content to the system, you need to associate a *resource factory* with a *content type*. A resource factory that generates content must have these properties:

- It must be a class, or a factory function that returns an instance of a resource class.
- Instances of the resource class must be *persistent* (it must derive from the `persistent.Persistent` class or a class that derives from Persistent such as *`substanced.folder.Folder`*).
- The resource class or factory must be decorated with the `@content` decorator, or must be added at configuration time via `config.add_content_type`.
- It must have a *type*. A type acts as a globally unique categorization marker, and allows the content to be constructed, enumerated, and introspected by various Substance D UI elements such as "add forms", and queries by the management interface for the icon class of a resource. A type can be any hashable Python object, but it's most often a string.

Here's an example which defines a content resource factory as a class:

```
# in a module named blog.resources

from persistent import Persistent
from substanced.content import content

@content('Blog Entry')
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
```

(continues on next page)

(continued from previous page)

```
        self.title = title
        self.body = body
```

Here's an example of defining a content resource factory using a function instead:

```
# in a module named blog.resources

from persistent import Persistent
from substanced.content import content

class BlogEntry(Persistent):
    def __init__(self, title, body):
        self.title = title
        self.body = body

@content('Blog Entry')
def make_blog_entry(title='', body=''):
    return BlogEntry(title, body)
```

When a resource factory is not a class, Substance D will wrap the resource factory in something that changes the resource object returned from the factory. In the above case, the BlogEntry instance returned from `make_blog_entry` will be changed; its `__factory_type__` attribute will be mutated.

Notice that when we decorate a resource factory class with `@content`, and the class' `__init__` function takes arguments, we provide those arguments with default values. This is mandatory if you'd like your content objects to participate in a "dump". Dumping a resource requires that the resource be creatable without any mandatory arguments. It's a similar story if our factory is a function; the function decorated by the `@content` decorator should provide defaults to any argument. In general, a resource factory can take arguments, but each parameter of the factory's callable should be given a default value. This also means that all arguments to a resource factory should be keyword arguments, and not positional arguments.

In order to activate a `@content` decorator, it must be *scanned* using the Pyramid `config.scan()` machinery:

```
# in a module named blog.__init__

from pyramid.config import Configurator

def main(global_config, **settings):
    config = Configurator()
    config.include('substanced')
    config.scan('blog.resources')
    # .. and so on ...
```

Instead of using the `@content` decorator, you can alternately add a content resource imperatively at configuration time using the `add_content_type` method of the Configurator:

```
# in a module named blog.__init__

from pyramid.config import Configurator
from .resources import BlogEntry

def main(global_config, **settings):
    config = Configurator()
    config.include('substanced')
    config.add_content_type('Blog Entry', BlogEntry)
```

This does the same thing as using the `@content` decorator, but you don't need to `scan()` your resources if you use `add_content_type` instead of the `@content` decorator.

Once a content type has been defined (and scanned, if it's been defined using a decorator), an instance of the resource can be constructed from within a view that lives in your application:

```
# in a module named blog.views

from pyramid.httpexceptions import HTTPFound
from pyramid.view import (
    view_config,
    view_defaults,
    )

@view_config(name='add_blog_entry', request_method='POST')
def add_blogentry(context, request):
    title = request.POST['title']
    body = request.POST['body']
    entry = request.registry.content.create('Blog Entry', title, body)
    context[title] = entry
    return HTTPFound(request.resource_url(entry))
```

The arguments passed to `request.registry.content.create` must start with the content type, and must be followed with whatever arguments are required by the resource factory.

Creating an instance of content this way isn't particularly more useful than creating an instance of the resource object by calling its class `__init__` directly unless you're building a highly abstract system. But even if you're not building a very abstract system, types can be very useful. For instance, types can be enumerated:

```
# in a module named blog.views

@view_config(name='show_types', renderer='show_types.pt')
def show_types(request):
    all_types = request.registry.content.all()
    return {'all_types':all_types}
```

`request.registry.content.all()` will return all the types you've defined and scanned.

4.4.2 Metadata

A content's type can be associated with metadata about that type, including the content type's name, its icon in the SDI management interface, an add view name, and other things. Pass arbitrary keyword arguments to the `@content` decorator or `config.add_content_type` to specify metadata.

Names

You can associate a content type registration with a name that shows up when someone attempts to add such a piece of content using the SDI management interface "Add" tab by passing a `name` keyword argument to `@content` or `config.add_content_type`.

```
# in a module named blog.resources

from persistent import Persistent
from substanced.content import content

@content('Blog Entry', name='Cool Blog Entry')
```

(continues on next page)

(continued from previous page)

```
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
        self.title = title
        self.body = body
```

Once you've done this, the "Add" tab in the SDI management interface will show your content as addable using this name instead of the type name.

Icons

You can associate a content type registration with a management view icon class by passing an `icon` keyword argument to `@content` or `add_content_type`.

```
# in a module named blog.resources

from persistent import Persistent
from substanced.content import content

@content('Blog Entry', icon='glyphicon glyphicon-file')
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
        self.title = title
        self.body = body
```

Once you've done this, content you add to a folder in the sytem will display the icon next to it in the contents view of the management interface and in the breadcrumb list. The available icon class names are listed at http://getbootstrap.com/components/#glyphicons . For glyphicon icons, you'll need to use two classnames: `glyphicon` and `glyphicon-foo`, separated by a space.

You can also pass a callback as an `icon` argument:

```
from persistent import Persistent
from substanced.content import content

def blogentry_icon(context, request):
    if context.body:
        return 'glyphicon glyphicon-file'
    else:
        return 'glyphicon glyphicon-gift'

@content('Blog Entry', icon=blogentry_icon)
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
        self.title = title
        self.body = body
```

A callable used as `icon` must accept two arguments: `context` and `request`. `context` will be an instance of the type and `request` will be the current request; your callback will be called at the time the folder view is drawn. The callable should return either an icon class name or `None`. For example, the above `blogentry_icon` callable tells the SDI to use an icon representing a file if the blogentry has a body, otherwise show an icon representing gift.

Add Views

You can associate a content type with a view that will allow the type to be added by passing the name of the add view as a keyword argument to `@content` or `add_content_type`.

```
# in a module named blog.resources

from persistent import Persistent
from substanced.content import content

@content('Blog Entry', add_view='add_blog_entry')
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
        self.title = title
        self.body = body
```

Once you've done this, if the button is clicked in the "Add" tab for this content type, the related view will be presented to the user.

You can also pass a callback as an `add_view` argument:

```
from persistent import Persistent
from substanced.content import content
from substanced.folder import Folder

def add_blog_entry(context, request):
    if request.registry.content.istype(context, 'Blog'):
        return 'add_blog_entry'

@content('Blog')
class Blog(Folder):
    pass

@content('Blog Entry', add_view=add_blog_entry)
class BlogEntry(Persistent):
    def __init__(self, title='', body=''):
        self.title = title
        self.body = body
```

A callable used as `add_view` must accept two arguments: `context` and `request`. `context` will be the potential parent object of the content (when the SDI folder view is drawn), and `request` will be the current request at the time the folder view is drawn. The callable should return either a view name or `None` if the content should not be addable in this circumstance. For example, the above `add_blog_entry` callable asserts that Blog Entry content should only be addable if the context we're adding to is of type Blog; it returns None otherwise, signifying that the content is not addable in this circumstance.

Obtaining Metadata About a Content Object's Type

Return the icon class name for the blogentry's content type or `None` if it does not exist:

```
request.registry.content.metadata(blogentry, 'icon')
```

Return the icon for the blogentry's content type or `glyphicon glyphicon-file` if it does not exist:

```
request.registry.content.metadata(blogentry, 'icon',
                                  'glyphicon glyphicon-file')
```

4.4.3 Affecting Content Creation

In some cases you might want your resource to perform some actions that can only take place after it has been seated in its container, but before the creation events have fired. The `@content` decorator and `add_content_type` method both support an `after_create` argument, pointed at a callable.

For example:

```
@content(
    'Document',
    icon='glyphicon glyphicon-align-left',
    add_view='add_document',
    propertysheets = (
        ('Basic', DocumentPropertySheet),
        ),
    after_create='after_creation'
    )
class Document(Persistent):

    name = renamer()

    def __init__(self, title, body):
        self.title = title
        self.body = body

    def after_creation(self, inst, registry):
        pass
```

If the value provided for `after_create` is a string, it's assumed to be a method of the created object. If it's a sequence, each value should be a string or a callable, which will be called in turn. The callable(s) are passed the instance being created and the registry. Afterwards, `substanced.event.ContentCreatedEvent` is emitted.

Construction of the root folder in Substance D is a special case. Most Substance D applications will start with:

```
from substanced.db import root_factory
def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings, root_factory=root_factory)
```

The *substanced.db.root_factory()* callable contains the following line:

```
app_root = registry.content.create('Root')
```

In many cases you want to perform some extra work on the `Root`. For example, you might want to create a catalog with indexes. Substance D emits an event when the root is created, so you can subscribe to that event and perform some actions:

```
from substanced.root import Root
from substanced.event import subscribe_created
from substanced.catalog import Catalog

@subscribe_created(Root)
def root_created(event):
    root = event.object
    catalog = Catalog()
    catalogs = root['catalogs']
    catalogs.add_service('catalog', catalog)
```

(continues on next page)

(continued from previous page)

```
    catalog.update_indexes('system', reindex=True)
    catalog.update_indexes('sdidemo', reindex=True)
```

4.4.4 Names and Renaming

A resource's "name" (`__name__`) is important to the system in Substance D. For example, traversal uses the value in URLs and paths to walk through hierarchy. Containers need to know when a resource's `__name__` changes.

To help support this, Substance D provides `substanced.util.renamer()`. You use it as a class attribute wrapper on resources that want "managed" names. These resources then gain a `name` attribute with a getter/setter from `renamer`. Getting the `name` returns the `__name__`. Setting `name` grabs the container and calls the `rename` method on the folder.

For example:

```
class Document(Persistent):
    name = renamer()
```

4.4.5 Special Colander Support

Forms and schemas for resources become pretty easy in Substance D. To make it easier for forms to interact with the Substance D machinery, it includes some special Colander schema nodes you can use on your forms.

NameSchemaNode

If you want your form to affect the `__name__` of a resource, certain constraints become applicable. These constraints might be different, so you might want to know if you are on an add form versus an edit form. `substanced.schema.NameSchemaNode` provides a schema node and default widget that bundles up the common rules for this. For example:

```
class BlogEntrySchema(Schema):
    name = NameSchemaNode()
```

The above provides the basics of support for editing a name property, especially when combined with the `renamer()` utility mentioned above.

By default the name is limited to 100 characters. `NameSchemaNode` accepts an argument that can set a different limit:

```
class BlogEntrySchema(Schema):
    name = NameSchemaNode(max_len=20)
```

You can also provide an `editing` argument, either as a boolean or a callable which returns a boolean, which determines whether the form is rendered in "editing" mode. For example:

```
class BlogEntrySchema(Schema):
    name = NameSchemaNode(
        editing=lambda c, r: r.registry.content.istype(c, 'BlogEntry')
        )
```

`PermissionSchemaNode`

A form might want to allow selection of zero or more permissions from the site's defined list of permissions. `PermissionSchemaNode` collects the possible state from the system, the currently-assigned values, and presents a widget that manages the values.

`MultireferenceIdSchemaNode`

References are a very powerful facility in Substance D. Naturally you'll want your application's forms to assign references. `MultireferenceIdSchemaNode` gives a schema node and widget that allows multiple selections of possible values in the system for references, including the current assignments.

As an example, the built-in `substanced.principal.UserSchema` uses this schema node:

```
class UserSchema(Schema):
    """ The property schema for :class:`substanced.principal.User`
    objects."""
    groupids = MultireferenceIdSchemaNode(
        choices_getter=groups_choices,
        title='Groups',
        )
```

4.4.6 Overriding Existing Content Types

Perhaps you would like to slightly adjust an existing content type, such as `Folder`, without re-implementing it. For exampler, perhaps you would like to override just the `add_view` and provide your own view, such as:

```
@mgmt_view(
    context=IFolder,
    name='my_add_folder',
    tab_condition=False,
    permission='sdi.add-content',
    renderer='substanced.sdi:templates/form.pt'
)
class MyAddFolderView(AddFolderView):

    def before(self, form):
        # Perform some custom work before validation
        pass
```

With this you can override any of the view predicates (such as `permission`) and override any part of the form handling (such as adding a `before` that performs some custom processing.)

To make this happen, you can re-register, so to speak, the content type during startup:

```
from substanced.folder import Folder
from .views import MyAddFolderView
config.add_content_type('Folder', Folder,
                        add_view='my_add_folder',
                        icon='glyphicon glyphicon-folder-close')
```

This, however, keeps the same content type class. You can also go further by overriding the content type definition itself:

```
@content(
    'Folder',
    icon='glyphicon glyphicon-folder-close',
    add_view='my_add_folder',
)
@implementer(IFolder)
class MyFolder(Folder):

    def send_email(self):
        pass
```

The class for the `Folder` content type has now been replaced. Instead of `substanced.folder.Folder` it is `MyFolder`.

Note: Overriding a content type is a pain-free way to make a custom `Root` object. You could supply your own `root_factory` to the `Configurator` but that means replicating all its rather complicated goings-on. Instead, provide your own content type factory, as above, for `Root`.

4.4.7 Adding Automatic Naming for Content

On some sites you don't want to set the name for every piece of content you create. Substance D provides support for this with a special kind of folder. You can configure your site to use the autonaming folder by overriding the standard folder:

```
from substanced.folder import SequentialAutoNamingFolder
from substanced.interaces import IFolder
from zope.interface import implementer

@content(
   'Folder',
    icon='glyphicon glyphicon-folder-close',
    add_view='add_folder',
)
@implementer(IFolder)
class  MyFolder(SequentialAutoNamingFolder):
    """ Override Folder content type """
```

The `add view` for Documents can then be edited to no longer require a name:

```
def add_success(self, appstruct):
    registry = self.request.registry
    document = registry.content.create('Document', **appstruct)
    self.context.add_next(document)
    return HTTPFound(
        self.request.sdiapi.mgmt_path(self.context, '@@contents')
    )
```

Note: This does not apply to the root object.

4.4.8 Affecting the Tab Order for Management Views

The `tab_order` parameter overrides the mgmt_view tab settings for a content type. Its value should be a sequence of view names, each corresponding to a tab that will appear in the management interface. Any registered view names that are omitted from this sequence will be placed after the other tabs.

4.4.9 Handling Content Events

Adding and modifying data related to content is, thanks to the framework, easy to do. Sometimes, though, you want to intervene and, for example, perform some extra work when content resources are added. Substance D has several framework events you can subscribe to using Pyramid events.

The `substanced.events` module imports these events as interfaces from *substanced.interfaces* and then provides decorator subscribers as convenience for each:

- *substanced.interfaces.IObjectAdded* as subscriber `@subscriber_added`
- *substanced.interfaces.IObjectWillBeAdded* as subscriber `@subscriber_will_be_added`
- *substanced.interfaces.IObjectRemoved* as subscriber `@subscriber_removed`
- *substanced.interfaces.IObjectWillBeRemoved* as subscriber `@subscriber_will_be_removed`
- *substanced.interfaces.IObjectModified* as subscriber `@subscriber_modified`
- *substanced.interfaces.IACLModified* as subscriber `@subscriber_acl_modified`
- *substanced.interfaces.IContentCreated* as subscriber `@subscriber_created`

As an example, the `substanced.principal.subscribers.user_added()` function is a subscriber to the `IObjectAdded` event:

```
@subscribe_added(IUser)
def user_added(event):
    """ Give each user permission to change their own password."""
    if event.loading: # fbo dump/load
        return
    user = event.object
    registry = event.registry
    set_acl(
        user,
        [(Allow, get_oid(user), ('sdi.view', 'sdi.change-password'))],
        registry=registry,
        )
```

As with the rest of Pyramid, you can do imperative configuration if you don't like decorator-based configuration, using `config.add_content_subscriber` Both the declarative and imperative forms result in `substanced.event.add_content_subscriber()`.

Note: While the event subscriber is de-coupled logically from the action that triggers the event, both the action and the subscriber run in the same transaction.

The `IACLModified` event (and `@subscriber_acl_modified` subscriber) is used internally by Substance D to re-index information in the system catalog's ACL index. Substance D also uses this event to maintain references between resources and principals. Substance D applications can use this in different ways, for example recording a security audit trail on security changes.

Sometimes when you perform operations on objects you don't want to perform the standard events. For example, in folder contents you can select a number of resources and move them to another folder. Normally this would fire content change events that re-index the files. This is fairly pointless: the content of the file hasn't changed.

If you looked at the interface for one of the content events, you would see some extra information supported. For example, in *substanced.interfaces.IObjectWillBeAdded*:

```
class IObjectWillBeAdded(IObjectEvent):
    """ An event type sent when an before an object is added """
    object = Attribute('The object being added')
    parent = Attribute('The folder to which the object is being added')
    name = Attribute('The name which the object is being added to the folder '
                     'with')
    moving = Attribute('None or the folder from which the object being added '
                       'was moved')
    loading = Attribute('Boolean indicating that this add is part of a load '
                        '(during a dump load process)')
    duplicating = Attribute('The object being duplicated or ``None``')
```

`moving`, `loading`, and `duplicating` are flags that can be set on the event when certain actions are triggered. These help in cases such as the one above: certain subscribers might want "flavors" of standard events and, in some cases, handle the event in a different way. This helps avoid lots of special-case events or the need for a hierarchy of events.

Thus in the case above, the catalog subscriber can see that the changes triggered by the event where in the special case of "moving". This can be seen in `substanced.catalog.subscribers.object_added`.

4.5 Management Views

A *management view* is a view configuration that applies only when the URL is prepended with the *manage prefix*. The manage prefix is usually `/manage`, unless you've changed it from its default by setting a custom `substanced.manage_prefix` in your application's `.ini` file.

This means that views declared as management views will never show up in your application's "retail" interface (the interface that normal unprivileged users see). They'll only show up when a user is using the *SDI* to manage content.

There are two ways to define management views:

- Using the `substanced.sdi.mgmt_view` decorator on a function, method, or class.
- Using the `substanced.sdi.add_mgmt_view()` Configurator (aka. `config.add_mgmt_view`) API.

The former is most convenient, but they are functionally equivalent. `mgmt_view` just calls into `add_mgmt_view` when found via a scan.

Declaring a management view is much the same as declaring a "normal" Pyramid view using `pyramid.view.view_config` with a `route_name` of `substanced_manage`. For example, each of the following view declarations will register a view that will show up when the `/manage/foobar` URL is visited:

```
from pyramid.view import view_config

@view_config(
    renderer='string',
    route_name='substanced_manage',
    name='foobar',
    permission='sdi.view',
    )
```

(continues on next page)

(continued from previous page)

```python
def foobar(request):
    return 'Foobar!'
```

The above is largely functionally the same as this:

```python
from substanced.sdi import mgmt_view

@mgmt_view(renderer='string', name='foobar')
def foobar(request):
    return 'Foobar!'
```

Management views, in other words, are really just plain-old Pyramid views with a slightly shorter syntax for definition. Declaring a view a management view, however, does do some extra things that make it advisable to use rather than a plain Pyramid view registration:

- It registers *introspectable* objects that the SDI interface uses to try to find management interface tabs (the row of actions at the top of every management view rendering).
- It allows you to associate a tab title, a tab condition, and cross-site request forgery attributes with the view.
- It uses the default permission `sdi.view`.

So if you want things to work right when developing management views, you'll use `@mgmt_view` instead of `@view_config`, and `config.add_mgmt_view` instead of `config.add_view`.

As you use management views in the SDI, you might notice that the URL includes `@@` as "goggles". For example, `http://0.0.0.0:6541/manage/@@contents` is the URL for seeing the folder contents. The `@@` is a way to ensure that you point at the URL for a *view* and not get some resource with the `__name__` of `contents`. You can still get to the folder contents management view using `http://0.0.0.0:6541/manage/contents`... until that folder contains something named `contents`.

4.5.1 `mgmt_view` View Predicates

Since `mgmt_view` is an extension of Pyramid's `view_config`, it re-uses the same concept of view predicates as well as some of the same actual predicates:

- `request_type`, `request_method`, `request_param`, `containment`, `attr`, `renderer`, `wrapper`, `xhr`, `accept`, `header`, `path_info`, `context`, `name`, `custom_predicates`, `decorator`, `mapper`, and `http_cache` are supported and behave the same.
- `permission` is the same but defaults to `sdi.view`.

The following are new view predicates introduced for `mgmt_view`:

- `tab_title` takes a string for the label placed on the tab.
- `tab_condition` takes a callable that returns `True` or `False`, or `True` or `False`. If you state a callable, this callable is passed `context` and `request`. The boolean determines whether the tab is listed in a certain situation.
- `tab_before` takes the view name of a `mgmt_view` that this `mgmt_view` should appear after (covered in detail in the next section.)
- `tab_after` takes the view name of a `mgmt_view` that this `mgmt_view` should appear after. Also covered below.
- `tab_near` takes a "sentinel" from `substanced.sdi` (or `None`) that makes a best effort at placement independent of another particular `mgmt_view`. Also covered below. The possible sentinel values are:

```
substanced.sdi.LEFT
substanced.sdi.MIDDLE
substanced.sdi.RIGHT
```

4.5.2 Tab Ordering

If you register a management view, a tab will be added in the list of tabs. If no mgmt view specifies otherwise via its tab data, the tab order will use a default sorting: alphabetical order by the `tab_title` parameter of each tab (or the view name if no `tab_title` is provided.) The first tab in this tab listing acts as the "default" that is open when you visit a resource. Substance D does, though, give you some options to control tab ordering in larger systems with different software registering management views.

Perhaps a developer wants to ensure that one of her tabs appears first in the list and another appears last, no matter what other management views have been registered by Substance D or any add-on packages. `@mgmt_view` (or the imperative call) allow a keyword of `tab_before` or `tab_after`. Each take the string tab `name` of the management view to place before or after. If you don't care (or don't know) which view name to use as a `tab_before` or `tab_after` value, use `tab_near`, which can be any of the sentinel values `MIDDLE`, `LEFT`, or `RIGHT`, each of which specifies a target "zone" in the tab order. Substance D will make a best effort to do something sane with `tab_near`.

As in many cases, an illustration is helpful:

```
from substanced.sdi import LEFT, RIGHT

@mgmt_view(
    name='tab_1',
    tab_title='Tab 1',
    renderer='templates/tab.pt'
    )
def tab_1(context, request):
    return {}


@mgmt_view(
    name='tab_2',
    tab_title='Tab 2',
    renderer='templates/tab.pt',
    tab_before='tab_1'
    )
def tab_2(context, request):
    return {}


@mgmt_view(
    name='tab_3',
    tab_title='Tab 3',
    renderer='templates/tab.pt',
    tab_near=RIGHT
    )
def tab_3(context, request):
    return {}


@mgmt_view(
    name='tab_4',
```

(continues on next page)

(continued from previous page)

```
    tab_title='Tab 4',
    renderer='templates/tab.pt',
    tab_near=LEFT
    )
def tab_4(context, request):
    return {}


@mgmt_view(
    name='tab_5',
    tab_title='Tab 5',
    renderer='templates/tab.pt',
    tab_near=LEFT
    )
def tab_5(context, request):
    return {}
```

This set of management views (combined with the built-in Substance D management views for `Contents` and `Security`) results in:

```
Tab 4 | Tab 5 | Contents | Security | Tab 2 | Tab 1 | Tab 3
```

These management view arguments apply to any content type that the view is registered for. What if you want to allow a content type to influence the tab ordering? As mentioned in the *content type docs*, the `tab_order` parameter overrides the mgmt_view tab settings, for a content type, with a sequence of view names that should be ordered (and everything not in the sequence, after.)

4.5.3 Filling Slots

Each management view that you write plugs into various parts of the SDI UI. This is done using normal ZPT `fill-slot` semantics:

- `page-title` is the `<title>` in the `<head>`
- `head-more` is a place to inject CSS and JS in the `<head>` *after* all the SDI elements
- `tail-more` does the same, just before the `</body>`
- `main` is the main content area

4.5.4 SDI API

All templates in the SDI share a common "layout". This layout needs information from the environment to render markup that is common to every screen, as well as the template used as the "main template."

This "template API" is known as the `SDI API`. It is an instance of the `sdiapi` class in `substanced.sdi.__init__.py` and is made available as `request.sdiapi`.

The template for your management view should start with a call to `requests.sdiapi`:

```
<div metal:use-macro="request.sdiapi.main_template">
```

The `request.sdiapi` object has other convenience features as well. See the Substance D interfaces documentation for more information.

4.5.5 Flash Messages

Often you perform an action on one view that needs a message displayed by another view on the next request. For example, if you delete a resource, the next request might confirm to the user "Deleted 1 resource." Pyramid supports this with "flash messages."

In Substance D, your applications can make a call to the `sdiapi` such as:

```
request.sdiapi.flash('ACE moved up')
```

. . . and the next request will process this flash message:

- The message will be removed from the stack of messages
- It will then be displayed in the appropriate styling based on the "queue"

The `sdiapi` provides another helper:

request.sdiapi.flash_with_undo('ACE moved up')

This displays a flash message as before, but also provides an `Undo` button to remove the previous transaction.

- title, content, flash messages, head, tail

4.6 Forms

When writing a Substance D application, you are free to use any library you would like for forms and schemas. This applies both for your retail views and for the management views that you plug into the SDI.

For the built-in content types and management views, you will see that Substance D has standardized on *Colander* and *Deform* for schemas and forms. Additionally, Substance D defines a `substanced.form.FormView` class, discussed below.

4.6.1 `FormView`

Form handling is ground that is frequently covered, usually in different ways. Substance D provides a class to help implement common patterns in form handling.

Imagine this example:

```
@mgmt_view(
    context=IFolder,
    name='add_document',
    tab_title='Add Document',
    permission='sdi.add-content',
    renderer='substanced.sdi:templates/form.pt',
    tab_condition=False,
)
class AddDocumentView(FormView):
    title = 'Add Document'
    schema = DocumentSchema()
    buttons = ('add',)

    def add_success(self, appstruct):
        registry = self.request.registry
        name = appstruct.pop('name')
        document = registry.content.create('Document', **appstruct)
```

(continues on next page)

(continued from previous page)

```
        self.context[name] = document
        return HTTPFound(
            self.request.mgmt_path(self.context, '@@contents'))
```

This `mgmt_view` adds a view `add_document` to resources with the `IFolder` interface. The form gets a `title`, a Colander schema, and asks for just one button.

Since the `mgmt_view` is associated with a renderer, we have an SDI template `form.pt` which does the basics of laying out the rendering before handing the work over to Deform.

The `@action` of the form is the `mgmt_view` itself, making it a self-posting form. The button that was clicked causes the `FormView` to, upon validation success, route processing to a handler for that button. By convention, `FormView` looks for a method starting with the name of the button (e.g. `add`) and finishing with `_success` (e.g. `add_success`.) The class also supports a similar protocol for `_failure`.

`FormView` also supports the following methods that can be overridden:

- `before(self, form)` is called before validation and processing of any `_success` or `_failure` methods
- `failure(self, e)` is called with the exception, if the there is no button-specific `_failure` method
- `show(self, form)` returns `{'form':form.render()}` and thus can be a place to affect form rendering

4.7 Services

A *service* is a name for a content object that provides a service to application code. It looks just like any other content object, but services that are added to a site can be found by name using various Substance D APIs.

Services expose APIs that exist for the benefit of application developers. For instance, the `catalogs` service provides an API that allows a developer to index and query for content objects using a structured query API. The `principals` service allows a developer to add and enumerate users and groups.

A service is added to a folder via the `substanced.folder.Folder.add_service()` API.

An existing service can be looked up in one of two ways: using the `substanced.util.find_service()` API or the `substanced.folder.Folder.find_service()` API. They are functionally equivalent. The latter exists only as a convenience so you don't need to import a function if you know you're dealing with a *folder*.

Either variant of `find_service` will look down the resource hierarchy towards the root until it finds a parent folder that has had `add_service` called on it. If the name passed in matches the service name, the object will be returned, otherwise the search will continue down the tree.

Note that a content object may exist in the folder with the same name as you're looking for via `find_service`, but if that object was not added via `add_service` (instead it's just a "normal" content object), it won't be found by `find_service`.

Here's how to use `substanced.util.find_service()`:

```
from substanced.util import find_service
principals = find_service(somecontext, 'principals')
```

`somecontext` above is any *resource* in the *resource tree*. For example, `somecontext` could be a "document" object you've added to a folder.

Here's how to use `substanced.folder.Folder.find_service()`:

```
principals = somefolder.find_service('principals')
```

`somefolder` above is any `substanced.folder.Folder` object (or any object which inherits from that class) present in the *resource tree*.

There is also the find-multiple-services variants `substanced.util.find_services()` and `substanced.folder.Folder.find_services()`.

4.8 Cataloging

Substance D provides application content indexing and querying via a *catalog*. A catalog is an object named `catalog` which lives in a service named `catalogs` within your application's resource tree. A catalog has a number of indexes, each of which keeps a certain kind of information about your content.

4.8.1 The Default Catalog

A default catalog named `system` is installed into the root folder's `catalogs` subfolder when you start Substance D. This `system` catalog contains a default set of indexes:

- path (a `path` index)

  Represents the path of the content object.

- name (a `field` index), uses `content.__name__` exclusively

  Represents the local name of the content object.

- interfaces (a `keyword` index)

  Represents the set of interfaces possessed by the content object.

- content_type (a `field` index)

  Represents the Substance D content-type of an object.

- allowed (an `allowed` index)

  An index which can be used to filter resultsets using principals and permissions.

- text (a `text` index)

  Represents the text searched for when you use the filter box within the folder contents view of the SDI.

4.8.2 Adding a Catalog

The `system` catalog won't have enough information to form all the queries you need. You'll have to add a catalog via code related to your application. The first step is adding a catalog factory.

A catalog factory is a collection of index descriptions. Creating a catalog factory doesn't actually add a catalog to your database, but it makes it possible to add one later.

Here's an example catalog factory named `mycatalog`:

```
from substanced.catalog import (
    catalog_factory,
    Text,
    Field,
```

(continues on next page)

(continued from previous page)

```
    )

@catalog_factory('mycatalog')
class MyCatalogFactory(object):
    freaky = Text()
    funky = Field()
```

In order to activate a `@catalog_factory` decorator, it must be *scanned* using the Pyramid `config.scan()` machinery. This will allow you to use `substanced.catalog.CatalogsService.add_catalog()` to add a catalog with that factory's name:

```
# in a module named blog.__init__

from pyramid.config import Configurator

def main(global_config, **settings):
    config = Configurator()
    config.include('substanced')
    config.scan('blog.catalogs')
    # .. and so on ...
```

Once you've done this, you can then add the catalog to the database in any bit of code that has access to the database. For example, in an event handler when the root object is created for the first time.

```
from substanced.root import Root
from substanced.event import subscribe_created

@subscribe_created(Root)
def created(event):
    root = event.object
    service = root['catalogs']
    service.add_catalog('mycatalog', update_indexes=True)
```

4.8.3 Object Indexing

Once a new catalog has been added to the database, each time a new *catalogable* object is added to the site, its attributes will be indexed by each catalog in its lineage that "cares about" the object. The object will always be indexed in the "system" catalog. To make sure it's cataloged in custom catalogs, you'll need to do some work. To index the object in a custom application index, you will need to create an *index view* for your content using `substanced.catalog.indexview`, and **scan** the resulting index view using `pyramid.config.Configurator.scan()`:

For example:

```
from substanced.catalog import indexview

class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview(catalog_name='mycatalog')
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)
```

An index view class should be a class that accepts a single argument, (conventionally named `resource`), in its constructor, and which has one or more methods named after potential index names. When it comes time for the system to index your content, Substance D will create an instance of your indexview class, and it will then call one or more of its methods; it will call methods on the indexview object matching the `attr` passed in to `add_indexview`. The `default` value passed in should be returned if the method is unable to compute a value for the content object.

Once this is done, whenever an object is added to the system, a value (the result of the `freaky(default)` method of the catalog view) will be indexed in the `freaky` field index.

You can attach multiple index views to the same index view class:

```python
from substanced.catalog import indexview

class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview(catalog_name='mycatalog')
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)

    @indexview(catalog_name='mycatalog')
    def funky(self, default):
        return getattr(self.resource, 'funky', default)
```

You can use the "index_name" parameter to `indexview` to tell the system that the index name is not the same as the method name in the index view:

```python
from substanced.catalog import indexview

class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview(catalog_name='mycatalog')
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)

    @indexview(catalog_name='mycatalog', index_name='funky')
    def notfunky(self, default):
        return getattr(self.resource, 'funky', default)
```

You can use the `context` parameter to `indexview` to tell the system that this particular index view should only be executed when the class of the resource (or any of its interfaces) matches the value of the context:

```python
from substanced.catalog import indexview

class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview(catalog_name='mycatalog', context=FreakyContent)
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)

    @indexview(catalog_name='mycatalog', index_name='funky')
    def notfunky(self, default):
        return getattr(self.resource, 'funky', default)
```

You can use the `indexview_defaults` class decorator to save typing in each `indexview` declaration. Keyword argument names supplied to `indexview_defaults` will be used if the `indexview` does not supply the same keyword:

```
from substanced.catalog import (
    indexview,
    indexview_defaults,
    )

@indexview_defaults(catalog_name='mycatalog')
class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview()
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)

    @indexview()
    def notfunky(self, default):
        return getattr(self.resource, 'funky', default)
```

The above configuration is the same as:

```
from substanced.catalog import indexview

class MyCatalogViews(object):
    def __init__(self, resource):
        self.resource = resource

    @indexview(catalog_name='mycatalog')
    def freaky(self, default):
        return getattr(self.resource, 'freaky', default)

    @indexview(catalog_name='mycatalog')
    def notfunky(self, default):
        return getattr(self.resource, 'funky', default)
```

You can also use the *substanced.catalog.add_indexview()* directive to add index views imperatively, instead of using the `@indexview` decorator.

4.8.4 Querying the Catalog

You execute a catalog query using APIs of the catalog's indexes.

```
from substanced.util import find_catalog

catalog = find_catalog(resource, 'system')
name = catalog['name']
path = catalog['path']
# find me all the objects that exist under /somepath with the name 'somename'
q = name.eq('somename') & path.eq('/somepath')
resultset = q.execute()
for contentob in resultset:
    print contentob
```

The calls to `name.eq()` and `path.eq()` above each return a query object. Those two queries are ANDed together into a single query via the `&` operator between them (there's also the `|` character to OR the queries together, but we don't use it above). Parentheses can be used to group query expressions together for the purpose of priority.

Different indexes have different query methods, but most support the `eq` method. Other methods that are often supported by indexes: `noteq`, `ge`, `le`, `gt`, `any`, `notany`, `all`, `notall`, `inrange`, `notinrange`. The *AllowedIndex* supports an additional *allows()* method.

Query objects support an `execute` method. This method returns a *hypatia.util.ResultSet*. A *hypatia.util.ResultSet* can be iterated over; each iteration returns a content object. *hypatia.util.ResultSet* also has methods like `one` and `first`, which return a single content object instead of a set of content objects. A *hypatia.util.ResultSet* also has a `sort` method which accepts an index object (the sort index) and returns another (sorted) *hypatia.util.ResultSet*.

```
catalog = find_catalog(resource, 'system')
name = catalog['name']
path = catalog['path']
# find me all the objects that exist under /somepath with the name 'somename'
q = name.eq('somename') & path.eq('/somepath')
resultset = q.execute()
newresultset = resultset.sort(name)
```

Note: If you don't call `sort` on the *hypatia.util.ResultSet* you get back, the results will not be sorted in any particular order.

4.8.5 Querying Across Catalogs

In many cases, you might only have one custom attribute that you need indexed, while the `system` catalog has everything else you need. You thus need an efficient way to combine results from two catalogs, before executing the query:

```
system_catalog = find_catalog(resource, 'system')
my_catalog = find_catalog(resource, 'mycatalog')
path = system_catalog['path']
funky = my_catalog['funky']
# find me all funky objects that exist under /somepath
q = funky.eq(True) & path.eq('/somepath')
resultset = q.execute()
newresultset = resultset.sort(system_catalog['name'])
```

4.8.6 Filtering Catalog Results Using the Allowed Index

The Substance D system catalog at `substanced.catalog.system.SystemCatalogFactory` contains a number of default indexes, including an `allowed` index. Its job is to index security information to allow security-aware results in queries. This index allows us to filter queries to the system catalog based on whether the principal issuing the request has a permission on the matching resource.

For example, the below query will find:

- all of the subresources inside a folder
- which is of content type `News Item`
- which the current user also possesses the `view` permission against

```
system_catalog = find_catalog(resource, 'system')
path = system_catalog['path']
```

(continues on next page)

(continued from previous page)

```
content_type = system_catalog['content_type']
allowed = system_catalog['allowed']
q = ( path.eq(resource, depth=1, include_origin=False) &
      content_type.eq('News Item') &
      allowed.allows(request, 'view')
    )
return q
```

4.8.7 Filtering Catalog Results Using The Objectmap

It is possible to postfilter catalog results using the *substanced.objectmap.ObjectMap.allowed()* API. For example:

```
def get_allowed_to_view(context, request):

    catalog = find_catalog(context, 'system')
    q = catalog['content_type'].eq('News Item')
    resultset = q.execute()

    objectmap = find_objectmap(context)
    return objectmap.allowed(
                resultset.oids, request.effective_principals, 'view')
```

The result of *allowed()* is a generator which returns oids, so the result must be listified if you intend to index into it, or slice it, or what-have-you.

4.8.8 Setting ACLs

The objectmap keeps track of ACLs in a cache to make catalog security functionality work. Note that for the object map's cached version of ACLs to be correct, you will need to set ACLs in a way that helps keep track of all the contracts. For this, the helper function *substanced.util.set_acl()* can be used. For example, the site root at *substanced.root.Root* finishes with:

```
set_acl(
    self,
    [(Allow, get_oid(admins), ALL_PERMISSIONS)],
    registry=registry,
    )
```

Using `set_acl` this way will generate an event that will keep the objectmap's cache updated. This will allow the `allowed` index to work and the *substanced.objectmap.ObjectMap.allowed()* method to work.

4.8.9 Deferred Indexing and Mode Parameters

As a lesson learned from previous cataloging experience, Substance D natively supports deferred indexing. As an example, in many systems the text indexing can be done after the change to the object is committed in the web request's transaction. Doing so has a number of performance benefits: the user's request processes more quickly, the work to extract text from a Word file can be performed later, less chance to have a conflict error, etc.

As such, the `substanced.catalog.system.SystemCatalogFactory`, by default, has indexes that aren't updated immediately when a resource is changed. For example:

```
# name is MODE_ATCOMMIT for next-request folder contents consistency
name = Field()

text = Text(action_mode=MODE_DEFERRED)
content_type = Field()
```

The `Field` indexes use the default of *MODE_ATCOMMIT*. The `Text` overrides the default and set `action_mode` to *MODE_DEFERRED*.

There are three such catalog "modes" for indexing:

- *substanced.interfaces.MODE_IMMEDIATE* means indexing action should take place as immediately as possible.
- *substanced.interfaces.MODE_ATCOMMIT* means indexing action should take place at the successful end of the current transaction.
- *substanced.interfaces.MODE_DEFERRED* means indexing action should be performed by an external indexing processor (e.g. `drain_catalog_indexing`) if one is active at the successful end of the current transaction. If an indexing processor is unavailable at the successful end of the current transaction, this mode will be taken to imply the same thing as `MODE_ATCOMMIT`.

4.8.10 Running an Indexer Process

Great, we've now deferred indexing to a later time. What exactly do we do at that later time?

Indexer processes are easy to write and schedule with `supervisor`. Here is an example of a configuration for `supervisor.conf` that will run in indexer process every five seconds:

```
[program:indexer]
command = %(here)s/../bin/sd_drain_indexing %(here)s/production.ini
redirect_stderr = true
stdout_logfile = %(here)s/../var/indexing.log
autostart = true
startsecs = 5
```

This calls `sd_drain_indexing` which is a console script that Substance D automatically creates in your `bin` directory. Indexing messages are logged with standard Python logging to the file that you name. You can view these messages with the `supervisorctl` command `tail indexer`. For example, here is the output from `sd_drain_indexing` when changing a simple `Document` content type:

```
2013-01-07 11:07:38,306 INFO  [substanced.catalog.deferred][MainThread] no actions to␣
↪execute
2013-01-07 11:08:38,329 INFO  [substanced.catalog.deferred][MainThread] executing
↪<substanced.catalog.deferred.IndexAction object oid 5886459017869105529 for index u
↪'text' at 0x106e52910>
2013-01-07 11:08:38,332 INFO  [substanced.catalog.deferred][MainThread] executing
↪<substanced.catalog.deferred.IndexAction object oid 5886459017869105529 for index u
↪'interfaces' at 0x106e52dd0>
2013-01-07 11:08:38,333 INFO  [substanced.catalog.deferred][MainThread] executing
↪<substanced.catalog.deferred.IndexAction object oid 5886459017869105529 for index u
↪'content_type' at 0x1076e2ed0>
2013-01-07 11:08:38,334 INFO  [substanced.catalog.deferred][MainThread] committing
2013-01-07 11:08:38,351 INFO  [substanced.catalog.deferred][MainThread] committed
```

4.8.11 Overriding Default Modes Manually

Above we set the default mode used by an index when Substance D indexes a resource automatically. Perhaps in an evolve script, you'd like to override the default mode for that index and reindex immediately.

The `index_resource` on an index can be passed an `action_mode` flag that overrides the configured mode for that index, and instead, does exactly what you want for only that call. It does not permanently change the configured default for indexing mode. This applies also to `reindex_resource` and `unindex_resource`. You can also grab the catalog itself and reindex with a mode that overrides all default modes on each index.

4.8.12 Autosync and Autoreindex

If you add `substanced.catalogs.autosync = true` within your application's `.ini` file, all catalog indexes will be resynchronized with their catalog factory definitions at application startup time. Indices which were added to the catalog factory since the last startup time will be added to each catalog which uses the index factory. Likewise, indices which were removed will be removed from each catalog, and indices which were modified will be modified according to the catalog factory. Having this setting in your `.ini` file is like pressing the `Update indexes` button on the `Manage` tab of each of your catalogs. The `SUBSTANCED_CATALOGS_AUTOSYNC` environment variable can also be used to turn this behavior on. For example `export SUBSTANCED_CATALOGS_AUTOSYNC=true`.

If you add `substanced.catalogs.autoreindex = true` within your application's `.ini` file, all catalogs that were changed as the result of an auto-sync will automatically be reindexed. Having this setting in your `.ini` file is like pressing the `Reindex catalog` button on the `Manage` tab of each catalog which was changed as the result of hitting `Update indexes`. The `SUBSTANCED_CATALOGS_AUTOREINDEX` environment variable can also be used to turn this behavior on. For example `export SUBSTANCED_CATALOGS_AUTOREINDEX=true`.

4.8.13 Forcing Deferral of Indexing

There may be times when you'd like to defer all catalog indexing operations, such as during a bulk load of data from a script. Normally, only indexes marked with `MODE_DEFERRED` use deferred indexing, and actions associated with those indexes are even then only actually deferred if an index processor is active.

You can force Substance D to defer all catalog indexing using the `substanced.catalogs.force_deferred` flag in your application's `.ini` file. When this flag is used, all catalog indexing operations will be added to the indexer's queue, even those indexes marked as `MODE_IMMEDIATE` or `MODE_ATCOMMIT`. Deferral will also happen whether or not the indexer is running, unlike during normal operations.

When you use this flag, you can stop the indexer process, do your bulk load, and start the indexer again when it's convenient to have all the content indexing done in the background.

The `SUBSTANCED_CATALOGS_FORCE_DEFERRED` environment variable can also be used to turn this behavior on. For example `export SUBSTANCED_CATALOGS_FORCE_DEFERRED=true`.

4.9 References

Objects that live in the Substance D resource tree can be related to one another using references.

The most user-visible facet of references is the SDI "References" tab, which is presented to SDI admin users when the object they're looking at is involved in a reference relation. For example, you'll notice that the built-in user and group implementations already have references to each other, and you can visit their References tabs to see them. Likewise, when you use the Security tab to change the ACL associated with an object, and include in the ACL a user or group that lives in the principals folder, a relation is formed between the ACL-bearing object and the principal. So, as you can see, references aren't just for application developers; Substance D itself uses references under the hood to do its job too.

A reference has a type and a direction. A reference is formed using methods of the *object map*.

```
from substanced.interfaces import ReferenceType
from substanced.objectmap import find_objectmap

class ContextToRoot(ReferenceType):
    pass

def connect_reference(context, request):
    objectmap = find_objectmap(context)
    root = request.root
    objectmap.connect(context, root, ContextToRoot)
```

A reference type is a class (not an instance) that inherits from `substanced.interfaces.ReferenceType`. The reference's name should indicate its directionality.

Warning: One caveat: reference types are *pickled*, so if you move a reference type from one location to another, you'll have to leave behind a backwards compatibility import in its original location "forever", so choose its name and location wisely. We recommend that you place it in an `interfaces.py` file in your project.

A reference can be removed using the object map too:

```
from substanced.interfaces import ReferenceType
from substanced.objectmap import find_objectmap

class ContextToRoot(ReferenceType):
    pass

def disconnect_reference(context, request):
    objectmap = find_objectmap(context)
    root = request.root
    objectmap.disconnect(context, root, ContextToRoot)
```

The first two arguments to *`connect()`* or *`disconnect()`* are *source* and *target*. These can be either resource objects or oids. The third argument to these functions is the reference type.

Once a reference is formed between two objects, you can see the reference within the "References" tab in the SDI. The References tab of either side of the reference (in the above example, either the root or the context) when visited in the SDI will display the reference to the other side.

Once a reference is made between two objects, the object map can be queried for objects which take part in the reference.

```
from substanced.interfaces import ReferenceType
from substanced.objectmap import find_objectmap

class ContextToRoot(ReferenceType):
    pass

def query_reference_sources(context, request):
    objectmap = find_objectmap(context)
    return objectmap.sourceids(request.root, ContextToRoot)

def query_reference_targets(context, request):
    objectmap = find_objectmap(context)
    return objectmap.targetids(context, ContextToRoot)
```

The `sourceids()` method returns the set of objectids which are *sources* of the object and reference type it's passed. The `targetids()` method returns the set of objectids which are *targets* of the object and reference type it's passed. If no objects are involved in the relation, an empty set will be returned in either case. `sources()` and `targets()` methods also exist which are analogous, but return the actual objects involved in the relation instead of the objectids:

```python
from substanced.interfaces import ReferenceType
from substanced.objectmap import find_objectmap

class ContextToRoot(ReferenceType):
    pass

def query_reference_sources(context, request):
    objectmap = find_objectmap(context)
    return objectmap.sources(request.root, ContextToRoot)

def query_reference_targets(context, request):
    objectmap = find_objectmap(context)
    return objectmap.targets(context, ContextToRoot)
```

A reference type can claim that it is "integral", which just means that the deletion of either the source or the target of a reference will be prevented. Here's an example of a "source integral" reference type:

```python
from substanced.interfaces import ReferenceType

class UserToGroup(ReferenceType):
    source_integrity = True
```

This reference type will prevent any object on the "user" side of the UserToGroup reference (as opposed to the group side) from being deleted. When a user attempts to delete a user that's related to a group using this reference type, a `substanced.objectmap.SourceIntegrityError` will be raised and the deletion will be prevented. Only when the reference is removed or the group is deleted will the user deletion be permitted.

The flip side of this is target integrity:

```python
from substanced.interfaces import ReferenceType

class UserToGroup(ReferenceType):
    target_integrity = True
```

This is the inverse. The reference will prevent any object on the "group" side of the UserToGroup reference from being deleted unless the associated user is first removed or the reference itself is no longer active. When a user attempts to delete a user that's related to a group using this reference type, a `substanced.objectmap.TargetIntegrityError` will be raised and the deletion will be prevented.

`substanced.objectmap.SourceIntegrityError` and `substanced.objectmap.TargetIntegrityError` both inherit from `substanced.objectmap.ReferentialIntegrityError`, so you can catch either in your code.

There are convenience functions that you can add to your resource objects that give them special behavior: `reference_sourceid_property()`, `reference_targetid_property()`, `reference_source_property()`, `reference_target_property()`, `multireference_sourceid_property()`, `multireference_targetid_property()`, `reference_source_property()`, and `reference_target_property()`.

Here's use of a reference property:

```python
from persistent import Persistent
from substanced.objectmap import reference_sourceid_property
```

(continues on next page)

(continued from previous page)

```python
from substanced.interfaces import ReferenceType

class LineItemToOrder(ReferenceType):
    pass

class LineItem(Persistent):
    order = reference_target_property(LineItemToOrder)
```

Once you've seated a resource object in a folder, you can then begin to use its special properties:

```python
from mysystem import LineItem, Order

lineitem = LineItem()
folder['lineitem'] = lineitem
lineitem.order = Order()
```

This is just a nicer way to use the objectmap query API; you don't have to interact with it at all, just assign and ask for attributes of your object. The `multireference_*` variants are similar to the reference variants, but they allow for more than one object on the "other side".

4.9.1 ACLs and Principal References

When an ACL is modified on a resource, a statement is being made about a relationship between that resource and a principal or group of principals. Wouldn't it be great if a reference was established, allowing you to then see such connections in the SDI?

This is indeed exactly how Substance D behaves: a source-integral PrincipalToACLBearing reference is set up between an ACL-bearing resource and the principals referred to within the ACL.

4.10 Workflows

A workflow is a collection of *transitions* that *transition* between *states*. Specifically, `substanced.workflow` implements event-driven finite-state machine workflows.

Workflows are used to ease following tasks when content goes through the lifecycle:

- updating security (adding/removing permissions)
- sending emails
- ...

States and transitions together with metadata are stored on the `Workflow`. Workflows are stored in `config.registry.workflows`. The only thing that content has from the workflow machinery is `content.__workflow_state__` attribute that stores a dict of all workflow types and corresponding states assigned. When content is added to the database (`ObjectAdded` event is emitted), all relevant registered workflows are initialized for it.

4.10.1 Features

- Site-wide workflows
- Multiple workflows per object

- Content type specific workflows
- Restrict transitions by permission
- Configurable callbacks when entering state
- Configurable callbacks when executing transition
- Reset workflow to initial state

4.10.2 Adding a workflow

Suppose we want to add a simple workflow:

```
/-----\    <-- to_draft -----    /---------\
|draft|                          |published|
\-----/    --- to_publish -->    \---------/
```

Using `add_workflow()` Pyramid configuration directive:

```
>>> workflow = Workflow(initial_state="draft", type="article")
>>> workflow.add_state("draft")
>>> workflow.add_state("published")
>>> workflow.add_transition('to_publish', from_state='draft', to_state='published')
>>> workflow.add_transition('to_draft', from_state='published', to_state='draft')

...

>>> config.add_workflow(workflow, ('News',))
```

4.10.3 Interaction with the workflow

Retrieve a `Workflow` instance using the `substanced.workflow.get_workflow()`:

```
>>> from substanced.workflow import get_workflow

>>> workflow = get_workflow(request, type='article', content_type='News')
```

Suppose there is a *context* object at hand, you can `reset()` its workflow to initial state:

```
>>> workflow.reset(context, request)
```

You could check it `has_state()` and assert `state_of()` *context* is initial state name of the workflow:

```
>>> assert workflow.has_state(context) == True
>>> assert workflow.state_of(context) == workflow.initial_state
```

List possible transitions from the current state of the workflow with `get_transitions()`:

```
>>> workflow.get_transitions(context, request)
[{'from_state': 'draft',
  'callback': None,
  'permission': None,
  'name': 'to_publish',
  'to_state': 'published'}]
```

Execute a `transition()`:

```
>>> workflow.transition(context, request, 'to_publish')
```

List all states of the workflow with *get_states()*:

```
>>> workflow.get_states(context, request)
[{'name': 'draft',
  'title': 'draft',
  'initial': True,
  'current': False,
  'transitions': [{'from_state': 'draft',
                   'callback': None,
                   'permission': None,
                   'name': 'to_publish',
                   'to_state': 'published'}],
  'data': {'callback': None}},
 {'name': 'published',
  'title': 'published',
  'initial': False,
  'current': True,
  'transitions': [{'from_state': 'published',
                   'callback': None,
                   'permission': None,
                   'name': 'to_draft',
                   'to_state': 'draft'}],
  'data': {'callback': None}}]
```

Execute a *transition_to_state()*:

```
>>> workflow.transition_to_state(context, request, 'draft')
```

4.10.4 Using callbacks

Typically you will want to define custom actions when transition is executed or when content enters a specific state. Let's define a transition with a callback:

```
>>> def cb(context, **kw):
...     print "keywords: ", kw

>>> workflow.add_transition('to_publish_with_callback',
...                         from_state='draft',
...                         to_state='published',
...                         callback=cb)
```

When you execute the transition, callback is called:

```
>>> workflow.transition(context, request, 'to_publish_with_callback')
keywords: {'workflow': <Workflow ...>, 'transition': {'to_state': 'published', 'from_
↪state': 'draft', ...}, request=<Request ...>}
```

To know more about callback parameters, read *add_transition()* signature.

4.11 Dumping Content to Disk

Substance D's object database stores native Python representations of resources. This is easy enough to work with: you can run `bin/pshell` to get an interactive prompt, write longer ad-hoc console scripts, or just put code into your application.

However, production sites usually want exportable representations of important data stored in a long-term format. For this, Substance D provides a dump facility for content types to be serialized in a YAML representation on disk.

Note: You'll note in the following the absence of docs on *loading* data. This is intentional. The process of loading data into a new, or semi-new, or newer-than-new site has many policy implications. Too many to fit into a single loading script. Substance D considers the particulars of loading data to be in the province of the application developer.

4.11.1 Dumping Resources Using `sd_dump`

The `sd_dump` console script loads your Substance D application, connects to your object database, and writes serialized representations of resources to disk in a directory hierarchy:

```
$ ../bin/sd_dump --help
Usage: sd_dump [options]

 Dump an object (and its subobjects) to the filesystem:  sd_dump [--source
=ZODB-PATH] [--dest=FILESYSTEM-PATH] config_uri   Dumps the object at ZODB-
PATH and all of its subobjects to a   filesystem path.  Such a dump can be
loaded (programmatically)   by using the substanced.dump.load function   e.g.
sd_dump --source=/ --dest=/my/dump etc/development.ini

Options:
  -h, --help            show this help message and exit
  -s ZODB-PATH, --source=ZODB-PATH
                        The ZODB source path to dump (e.g. /foo/bar or /)
  -d FILESYSTEM-PATH, --dest=FILESYSTEM-PATH
                        The destination filesystem path to dump to.
```

For example:

```
$ ../bin/sd_dump ../etc/development.ini
2013-01-07 13:27:03,939 INFO  [ZEO.ClientStorage][MainThread] ('localhost', 9963)␣
↪ClientStorage (pid=93148) created RW/normal for storage: 'main'
2013-01-07 13:27:03,941 INFO  [ZEO.cache][MainThread] created temporary cache file '
↪<fdopen>'
2013-01-07 13:27:03,981 WARNI [ZEO.zrpc][Connect([(2, ('localhost', 9963))])] (93148)␣
↪CW: error connecting to ('fe80::1%lo0', 9963): EHOSTUNREACH
2013-01-07 13:27:03,982 WARNI [ZEO.zrpc][Connect([(2, ('localhost', 9963))])] (93148)␣
↪CW: error connecting to ('fe80::1%lo0', 9963): EHOSTUNREACH
2013-01-07 13:27:04,002 WARNI [ZEO.zrpc][Connect([(2, ('localhost', 9963))])] (93148)␣
↪CW: error connecting to ('::1', 9963): EINVAL
2013-01-07 13:27:04,003 INFO  [ZEO.ClientStorage][Connect([(2, ('localhost',␣
↪9963))])] ('localhost', 9963) Testing connection <ManagedClientConnection ('127.0.0.
↪1', 9963)>
2013-01-07 13:27:04,004 INFO  [ZEO.zrpc.Connection(C)][('localhost', 9963) zeo client␣
↪networking thread] (127.0.0.1:9963) received handshake 'Z3101'
2013-01-07 13:27:04,105 INFO  [ZEO.ClientStorage][Connect([(2, ('localhost',␣
↪9963))])] ('localhost', 9963) Server authentication protocol None
```

(continues on next page)

(continued from previous page)

```
2013-01-07 13:27:04,106 INFO  [ZEO.ClientStorage][Connect([(2, ('localhost',␣
↪9963))])] ('localhost', 9963) Connected to storage: ('localhost', 9963)
2013-01-07 13:27:04,108 INFO  [ZEO.ClientStorage][Connect([(2, ('localhost',␣
↪9963))])] ('localhost', 9963) No verification necessary -- empty cache
2013-01-07 13:27:04,727 INFO  [substanced.catalog][MainThread] system update_indexes:␣
↪no indexes added or removed
2013-01-07 13:27:04,730 INFO  [substanced.catalog][MainThread] sdidemo update_
↪indexes: no indexes added or removed
2013-01-07 13:27:04,732 INFO  [substanced.dump][MainThread] Dumping /
2013-01-07 13:27:04,749 INFO  [substanced.dump][MainThread] Dumping /principals
2013-01-07 13:27:04,754 INFO  [substanced.dump][MainThread] Dumping /principals/users
2013-01-07 13:27:04,760 INFO  [substanced.dump][MainThread] Dumping /principals/users/
↪admin
2013-01-07 13:27:04,779 INFO  [substanced.dump][MainThread] Dumping /principals/resets
2013-01-07 13:27:04,783 INFO  [substanced.dump][MainThread] Dumping /principals/groups
```

…with logging messages being emitted until all known content is dumped. A `dump` subdirectory in the current directory is created (if no argument is provided) containing:

```
$ ls
acl.yaml    propsheets      references.yaml resource.yaml   resources
```

Note: To correctly encode as much meaning as possible, the dump files contain some advanced and custom YAML constructs when needed.

`acl.yaml` For Security Settings

This YAML file contains security settings for this resource. For example:

```
- !!python/tuple [Allow, 1644064392535565429, !all_permissions '']
```

`references.yaml` for Reference Information

Data about references aren't stored on the resources involved in the reference. Instead, they are stored in the objectmap. This file contains the reference information for the resource identified at the current dump directory. For example:

```
!interface 'substanced.interfaces.PrincipalToACLBearing':
  sources: [1644064392535565429]
```

`workflow.yaml` for Workflow Settings

The workflow engine can contain information about resource state. For example:

```
!!python/object:persistent.mapping.PersistentMapping
data: {document: draft}
```

`propsheets` Directory for Property Sheet Data

Resources can have multiple system-defined or application-defined property sheets on resources. These are serialized as subdirectories under `propsheets`, with a directory for each property sheet. For example, a resources `propsheets/Basic/properties.yaml` might contain:

```
{body: !!python/unicode 'The quick brown fox jumps over the lazy dog. The quick brown
    fox jumps over the lazy dog.  The quick brown fox jumps over the lazy dog.
    The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the
    lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps
    over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown
    fox jumps over the lazy dog. ', name: !!python/unicode 'document_0', title: !!
↪python/unicode 'Document
    0 Binder 0'}
```

`resource.yaml` for Content Type Information

Each directory after the top corresponds to a resource in the database. As such, the resource likely has content type information. The dump script encodes this into a YAML file in the resource's dump directory:

```
{content_type: Root, created: !!timestamp '2013-01-07 14:23:23.133436', is_service:␣
↪false,
  name: null, oid: 1644064392535565415}
```

`resources` for Contained Resources in Containers

If the resource at a current dump directory is a `Folder` or some other kind of container, it will contain a `resources` subdirectory. This might contain more subfolders and thus subdirectories. It might also contain individual resources, as a subdirectory named with the resource name.

4.11.2 Custom Dumping with `__dump__`

The built-in facilities allow automatic dumping of most information for your content, including information in your property sheets, the content type, security settings, references, workflows, etc.

If you do need extra information dumped to YAML about your content type, Substance D has a Python protocol using an `__dump__` on your `@content` class. As an example, :py:meth:`substanced.principal.User.dump` is a callable which returns a mapping of simple Python objects. The dumper checks to see if a resource has a `__dump__` method. If so, it calls the method, encodes the result to YAML, and writes it to an `adhoc.yaml` file in the dumped-resource's directory.

The inverse is also true. If a content type has a `__load__` method, information from that method is added to the state that is loaded.

4.11.3 Adding New Dumpers

The `adhoc.yaml` file that we just saw is an example of the `AdhocAttrDumper`. There are seven other dumpers built-in: acl, workflow, references, sdiproperties, interfaces, order, and propsheets.

If you would like a custom dumper, you can register it with `config.add_dumper`. For example, `substanced.dump.includeme()` registers the existing dumpers and their dumper factories:

```
def includeme(config):
    DEFAULT_DUMPERS = [
        ('acl', ACLDumper),
        ('workflow', WorkflowDumper),
        ('references', ReferencesDumper),
        ('sdiproperties', SDIPropertiesDumper),
        ('interfaces', DirectlyProvidedInterfacesDumper),
        ('order', FolderOrderDumper),
        ('propsheets', PropertySheetDumper),
        ('adhoc', AdhocAttrDumper),
        ]
    config.add_directive('add_dumper', add_dumper)
    for dumper_name, dumper_factory in DEFAULT_DUMPERS:
        config.add_dumper(dumper_name, dumper_factory)
```

4.12 Changing Resource Structure With Evolution

As you develop your software and make changes to structures, your existing content will be in an old state. Whether in production or during development, you need a facility to correct out-of-date data.

Evolution provides a rich facility for “evolving” your resources to match changes during development. Substance D’s evolution facility gives Substance D developers full control over the data updating process:

- Write scripts for each package that get called during an update
- Set revision markers in the data to indicate the revision level a database is at
- Console script and SDI GUI that can be run to “evolve” a database

4.12.1 Running an Evolution from the Command Line

Substance D applications generate a console script at `bin/sdi_evolve`. Running this without arguments displays some help:

```
$ bin/sd_evolve
Requires a config_uri as an argument

    sd_evolve [--latest] [--dry-run] [--mark-finished=stepname] [--mark-
↪unfinished=stepname] config_uri
      Evolves new database with changes from scripts in evolve packages
         - with no arguments, evolve displays finished and unfinished steps
         - with the --latest argument, evolve runs scripts as necessary
         - with the --dry-run argument, evolve runs scripts but does not issue any␣
↪commits
         - with the --mark-finished argument, marks the stepname as finished
         - with the --mark-unfinished argument, marks the stepname as unfinished

    e.g. sd_evolve --latest etc/development.ini
```

Running with your INI file, as explained in the help, shows information about the version numbers of various packages:

```
$ bin/sd_evolve etc/development.ini

Finished steps:

```

(continues on next page)

(continued from previous page)

```
    2013-06-14 13:01:28 substanced.evolution.legacy_to_new

Unfinished steps:
```

This shows that one evolution step has already been run and that there are no unfinished evolution steps.

4.12.2 Running an Evolution from the SDI

The Evolution section of the `Database` tab of the Substance D root object allows you to do what you might have otherwise done using the `sd_evolve` console script described above.

In some circumstances when Substance D itself needs to be upgraded, you may need to use the `sd_evolve` script rather than the GUI. For example, if the way that Substance D `Folder` objects work is changed and folder objects need to be evolved, it may be impossible to view the evolution GUI, and you may need to use the console script.

4.12.3 Autoevolve

If you add `substanced.autoevolve = true` within your application .ini file, all pending evolution upgrade steps will be run when your application starts. Alternately you can use the `SUBSTANCED_AUTOEVOLVE` evnironment variable (e.g. `export SUBSTANCED_AUTOEVOLVE=true`) to do the same thing.

4.12.4 Adding Evolution Support To a Package

Let's say we have been developing an `sdidemo` package and, with content already in the database, we want to add evolution support. Our `sdidemo` package is designed to be included into a site, so we have the traditional Pyramid `includeme` support. In there we add the following:

```
import logging

logger = logging.getLogger('evolution')

def evolve_stuff(root, registry):
    logger.info('Stuff evolved.')

def includeme(config):
    config.add_evolution_step(evolve_stuff)
```

We've used the *`substanced.evolution.add_evolution_step()`* API to add an evolution step in this package's `includeme` function.

Running `sd_evolve` *without* `--latest` (meaning, without performing an evolution) shows that Substance D's evolution now knows about our package:

```
$ bin/sd_evolve etc/development.ini

Finished steps:

    2013-06-14 13:01:28 substanced.evolution.legacy_to_new

Unfinished steps:

                        sdidemo.evolve_stuff
```

Let's now run `sd_evolve` "for real". This will cause the evolution step to be executed and marked as finished.

```
$ bin/sd_evolve --latest etc/development.ini

2013-06-14 13:22:51,475 INFO  [evolution][MainThread] Stuff evolved.
Evolution steps executed:
   substanced.evolution.evolve_stuff
```

This examples shows a number of points:

- Each package can easily add evolution support via the `config.add_evolution_step()` directive. You can learn more about this directive by reading its API documentation at *`substanced.evolution.add_evolution_step()`*.
- Substance D's evolution service looks at the database to see which steps haven't been run, then runs all the needed evolve scripts, sequentially, to bring the database up to date.
- All changes within an evolve script are in the scope of a transaction. If all the evolve scripts run to completion without exception, the transaction is committed.

4.12.5 Manually Marking a Step As Evolved

In some cases you might have performed the work in an evolve step by hand and you know there is no need to re-perform that work. You'd like to mark the step as finished for one or more evolve scripts, so these steps don't get run. The `--mark-step-finished` argument to `sd_evolve` accomplishes this. The "Mark finished" button in the SDI evolution GUI does the same.

4.12.6 Baselining

Evolution is baselined at first startup. When there's no initial list of finished steps in the database. Substance D, in the root factory, says: "I know all the steps participating in evolution, so when I first create the root object, I will set all of those steps to finished."

If you wish to perform something after *`Root`* was created, see *Affecting Content Creation*.

4.13 Configuring Folder Contents

The folder contents, as mentioned previously in *Folder contents*, the SDI's folder contents uses a powerful datagrid to view and manage items in a folder. This chapter covers how your content types can plug into the folder contents view.

4.13.1 Adding Columns

Perhaps your system has content types with extra attributes that are meaningful and you'd like your contents listings to show that column. You can change the columns available on folder contents listings by passing in a `columns` argument to the `@content` directive. The value of this argument is a callable which returns a sequence of mappings conforming to the datagrid's contract. For example:

```python
def binder_columns(folder, subobject, request, default_columnspec):
    subobject_name = getattr(subobject, '__name__', str(subobject))
    objectmap = find_objectmap(folder)
    user_oid = getattr(subobject, 'creator', None)
    created = getattr(subobject, 'created', None)
```

(continues on next page)

(continued from previous page)

```
    modified = getattr(subobject, 'modified', None)
    if user_oid is not None:
        user = objectmap.object_for(user_oid)
        user_name = getattr(user, '__name__', 'anonymous')
    else:
        user_name = 'anonymous'
    if created is not None:
        created = created.isoformat()
    if modified is not None:
        modified = modified.isoformat()
    return default_columnspec + [
        {'name': 'Title',
        'value': getattr(subobject, 'title', subobject_name),
        },
        {'name': 'Created',
        'value': created,
        'formatter': 'date',
        },
        {'name': 'Last edited',
        'value': modified,
        'formatter': 'date',
        },
        {'name': 'Creator',
        'value': user_name,
        }
        ]

@content(
    'Binder',
    icon='glyphicon glyphicon-book',
    add_view='add_binder',
    propertysheets = (
        ('Basic', BinderPropertySheet),
        ),
    columns=binder_columns,
    )
```

The callable is passed the folder, a subobject, the `request`, and a set of default column specifications. To display the datagrid column headers, your callable is invoked on the first resource. Later, this callable is used to get the value for the fields of each column for each resource in a request's batch.

The mappings returned can indicate whether a particular column should be sorted. If you want your column to be sortable, you must provide a `sorter` key in the mapping. If supplied, the `sorter` value must either be `None` if the column is not sortable, or a function which accepts a resource (the folder), a "resultset", a `limit` keyword argument, and a `reverse` keyword argument and which must return a sorted result set. Here's an example sorter:

```
from substanced.util import find_index

def sorter(folder, resultset, reverse=False, limit=None):
    index = find_index(folder, 'mycatalog', 'date')
    if index is not None:
        resultset = resultset.sort(index, reverse=reverse, limit=limit)
    return resultset

def my_columns(folder, subobject, request, default_columnspec):
    return default_columnspec + [
```

(continues on next page)

(continued from previous page)

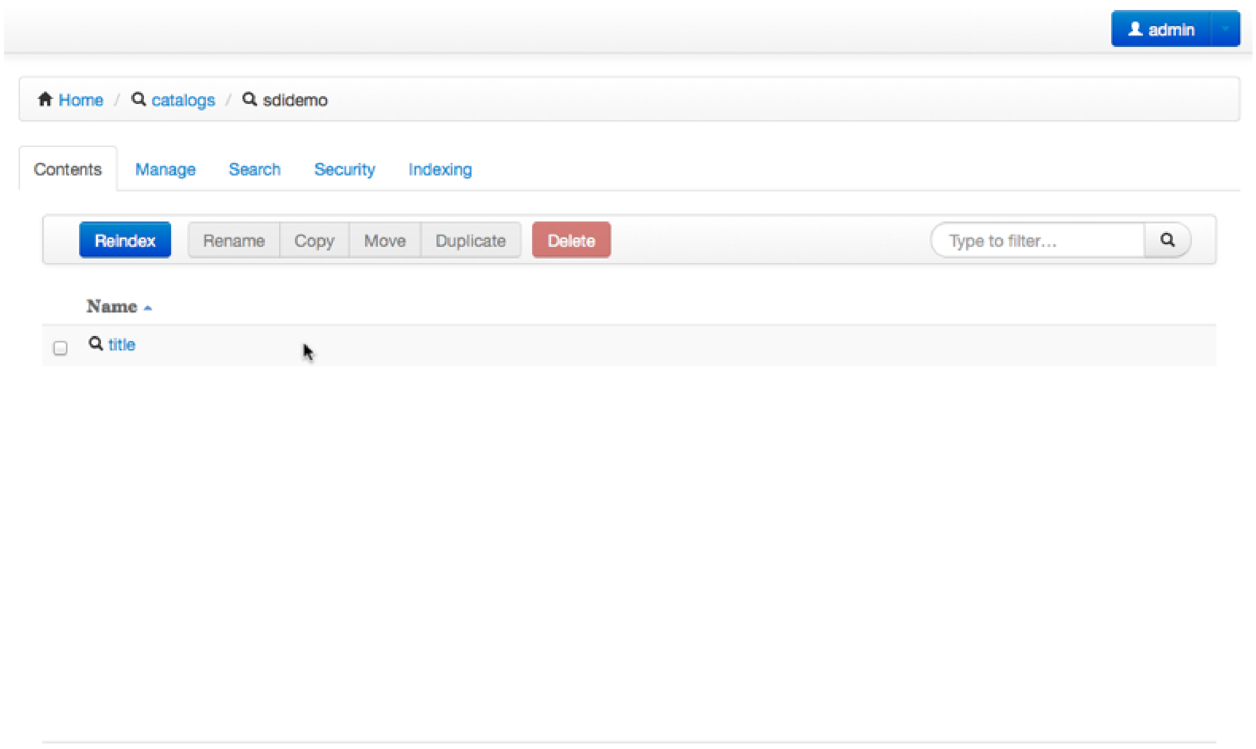
```
        {'name': 'Date',
        'value': getattr(subobject, 'title', subobject_name),
        'sorter': 'sorter',
        },
```

Most often, sorting is done by passing a catalog index into the resultset.sort method as above (resultset.sort returns another resultset), but sorting can be performed manually, as long as the sorter returns a resultset.

4.13.2 Buttons

As we just showed, you can extend the folder contents with extra columns to display and possibly sort on. You can also add new buttons that will trigger operations on selected resources.

As with columns, we pass a new argument to the `@content` directive. For example, the folder contents view for the catalogs folder allows you to reindex multiple indexes at once:



The `Reindex` button illustrates a useful facility for performing many custom operations at once.

The *substanced.catalog* module's `@content` directive has a `buttons` argument:

```
@content(
    'Catalog',
    icon='glyphicon glyphicon-search',
    service_name='catalog',
    buttons=catalog_buttons,
    )
```

This points at a callable:

```
def catalog_buttons(context, request, default_buttons):
    """ Show a reindex button before default buttons in the folder contents
    view of a catalog"""
    buttons = [
        {'type':'single',
         'buttons':
         [
             {'id':'reindex',
              'name':'form.reindex',
              'class':'btn-primary btn-sdi-sel',
              'value':'reindex',
              'text':'Reindex'}
             ]
         }
        ] + default_buttons
    return buttons
```

In this case, the `Reindex` button was inserted before the other buttons, in the place where an add button would normally appear.

The `class` on your buttons affect behavior in the datagrid:

- `btn-primary` gives this button the styling for the primary button of a form, using Twitter Bootstrap form styling
- `btn-sdi-act` makes the button always enabled
- `btn-sdi-sel` disables the button until one or more items are selected
- `btn-sdi-one` disables the button until exactly one item is selected
- `btn-sdi-del` disables the button if any of the selected resources is marked as "non-deletable" (discussed below)

When clicked, this button will do a form `POST` of the selected docids to a view that you have implemented. Which view? The `'name': 'form.reindex'` item sets the parameter on the POST. You can then register a view against this. `substanced.catalog.views.catalog` shows this:

```
@mgmt_view(
    context=IFolder,
    content_type='Catalog',
    name='contents',
    request_param='form.reindex',
    request_method='POST',
    renderer='substanced.folder:templates/contents.pt',
    permission='sdi.manage-contents',
    tab_condition=False,
    )
def reindex_indexes(context, request):
    toreindex = request.POST.getall('item-modify')
    if toreindex:
        context.reindex(indexes=toreindex, registry=request.registry)
        request.sdiapi.flash(
            'Reindex of selected indexes succeeded',
            'success'
            )
    else:
        request.sdiapi.flash(
            'No indexes selected to reindex',
```

(continues on next page)

(continued from previous page)

```
            'danger'
            )

    return HTTPFound(request.sdiapi.mgmt_path(context, '@@contents'))
```

4.13.3 Selection and Button Enabling

As mentioned above, some buttons are driven by the selection. If nothing is selected, the button is disabled.

Buttons can also be disabled if any selected item is "non-deletable". How does that get signified? An item is 'deletable' if the user has the `sdi.manage-contents` permission on `folder` *and* if the subobject has a `__sdi_deletable__` attribute which resolves to a boolean `True` value.

It is also possible to make button enabling and disabling depend on some application-specific condition. To do this, assign a callable to the `enabled_for` key in the button spec. For example:

```
def catalog_buttons(context, request, default_buttons):
    def is_indexable(folder, subobject, request):
        """ only enable the button if subobject is indexable """
        return subobject.is_indexable()

    buttons = [
        {'type':'single',
         'buttons':
         [
             {'id':'reindex',
              'name':'form.reindex',
              'class':'btn-primary btn-sdi-sel',
              'value':'reindex',
              'enabled_for': is_indexable,
              'text':'Reindex'}
             ]
         }
        ] + default_buttons
    return buttons
```

In the example above, we define a button similar to our previous reindex button, except this time we have an `enabled_for` key that is assigned the `is_indexable` function. When the buttons are rendered, each element is passed to this function, along with the folder and request. If *any one* of the folder subobjects returns `False` for this call, the button will not be enabled.

4.13.4 Filtering What Can Be Added

Not all kinds of resources make sense to be added inside a certain kind of container. For example, *`substanced.catalog.Catalog`* is a content type that can hold only indexes. That is,it isn't meant to hold any arbitrary kind of thing.

To tell the SDI what can be added inside a container content type, add a `__sdi_addable__` method to your content type. This method is passed the folder object representing the place the object might be added, and a Substance D introspectable for a content type. When Substance D tries to figure out whether an object is addable to a particular folder, it will call the `__sdi_addable__` method of your folderish type once for each content type.

The introspectable is a dictionary-like object which contains information about the content type. The introspectable contains the following keys:

meta A dictionary representing "meta" values passed to *add_content_type()*. For example, if you pass add_view='foo' to *add_content_type()*, the meta of the content type will be {'add_view':'foo'}.

content_type The content type value passed to *add_content_type()*.

factory_type The factory_type value passed to *add_content_type()*.

original_factory The original content factory (without any wrapping) passed to *add_content_type()*.

factory The potentially wrapped content factory derived from the original factory in *add_content_type()*.

See *Registering Content* for more information about content type registration and what the above introspectable values mean.

Your __sdi_addable__ method can perform some logic using the values it is passed, and then it must return a filtered sequence.

As an example, the __sdi_addable__ method on the Catalog filters out the kinds of things that can be added in a catalog.

4.13.5 Extending Which Columns Are Displayed

The folder contents grid displays a number of columns by default. If you are managing content with custom properties, in some cases you want to list those properties in the columns the grid can display. You can do so on custom folder content types by adding a columns argument to your @content decorator.

As an example, imagine a Binder kind of container. It has a content type declaration:

```
@content(
    'Binder',
    icon='glyphicon glyphicon-book',
    add_view='add_binder',
    propertysheets = (
        ('Basic', BinderPropertySheet),
        ),
    columns=binder_columns,
    )
```

The binder_columns points to a callable where we perform the work to both add the column to the list of columns, but also specify how to get the row data for that column:

```
def binder_columns(folder, subobject, request, default_columnspec):
    subobject_name = getattr(subobject, '__name__', str(subobject))
    objectmap = find_objectmap(folder)
    user_oid = getattr(subobject, 'creator', None)
    created = getattr(subobject, 'created', None)
    modified = getattr(subobject, 'modified', None)
    if user_oid is not None:
        user = objectmap.object_for(user_oid)
        user_name = getattr(user, '__name__', 'anonymous')
    else:
        user_name = 'anonymous'
    if created is not None:
        created = created.isoformat()
    if modified is not None:
        modified = modified.isoformat()
    return default_columnspec + [
        {'name': 'Title',
```

(continues on next page)

(continued from previous page)

```
        'value': getattr(subobject, 'title', subobject_name),
        },
        {'name': 'Created',
        'value': created,
        'formatter': 'date',
        },
        {'name': 'Last edited',
        'value': modified,
        'formatter': 'date',
        },
        {'name': 'Creator',
        'value': user_name,
        }
        ]
```

Here we add four columns to the standard set of grid columns, whenever we are in a `Binder` folder.

4.13.6 Adding New Folder Contents Buttons

The grid in folder contents makes it easy to select multiple resources then click a button to perform an action. Wouldn't it be great, though, if we could add a new button to all or certain folders, to perform custom actions?

In the previous section we saw how to pass another argument to the `@content` decorator. We do the same for new buttons. A content type can pass in `buttons=callable` to modify the list of buttons on a particular kind of folder.

For example, the `substanced.catalog.catalog_buttons()` callable adds a new `Reindex` button in front of the standard set of buttons:

```
def catalog_buttons(context, request, default_buttons):
    """ Show a reindex button before default buttons in the folder contents
    view of a catalog"""
    buttons = [
        {'type':'single',
         'buttons':
         [
             {'id':'reindex',
              'name':'form.reindex',
              'class':'btn-primary btn-sdi-sel',
              'value':'reindex',
              'text':'Reindex'}
             ]
         }
        ] + default_buttons
    return buttons
```

The button is disabled until one or more resources are selected which have the correct permission (discussed above.) If our new button is clicked, the form is posted with the `form.reindex` value in post data. You can then make a `@mgmt_view` with `request_param='form.reindex'` in the declaration to handle the form post when that button is clicked.

4.13.7 Broken Objects and Class Aliases

Let's assume that there's an object in your database that is an instance of the class `myapplication.resources.MyCoolResource`. If that class is subsequently renamed to `myapplication.resources.MySuperVeryCoolResource`, the `MyCoolResource` object that exists in the database will become broken.

This is because the ZODB database used by Substance D uses the Python `pickle` persistence format, and `pickle` writes the literal class name into the record associated with an object instance. Therefore, if a class is renamed or moved, when you come along later and try to deserialize a pickle with the old name, it will not work as it used to.

Persistent objects that exist in the database but which have a class that cannot be resolved are called "broken objects". If you ask a Substance D folder (or the object map) for an object that turns out to be broken in this way, it will hand you back an instance of the `pyramid.util.BrokenWrapper` class. This class tries to behave as much as possible like the original object for data that exists in the original objects' `__dict__` (it defines a custom `__getattr__` that looks in the broken object's state). However, you won't able to call methods of the original class against a broken object.

You can usually delete broken objects using the SDI folder contents view if necessary.

If you must rename or move a class, you can leave a class alias behind for backwards compatibility to avoid seeing broken objects in your database. For example:

```
class MySuperVeryCoolResource(Persistent):
    pass

MyCoolResource = MySuperVeryCoolResource # bw compat alias
```

4.14 Using Auditing

Substance D keeps an audit log of all meaningful operations performed against content if you have an audit database configured. At the time of this writing, "meaningful" is defined as:

- When an ACL is changed.
- When a resource is added or removed.
- When a resource is modified.

The audit log is of a fixed size (currently 1,000 items). When the audit log fills up, the oldest audit event is thrown away. Currently we don't have an archiving mechanism in place to keep around the items popped off the end of the log when it fills up; this is planned.

You can extend the auditing system by using the *substanced.audit.AuditLog*, writing your own events to the log.

4.14.1 Configuring the Audit Database

In order to enable auditing, you have to add an `audit` database to your Substance D configuration. This means adding a key to your application's section in the `.ini` file associated with the app:

```
zodbconn.uri.audit = <some ZODB uri>
```

An example of "some ZODB URI" above might be (for a FileStorage database, if your application doesn't use multiple processes):

```
zodbconn.uri.audit = file://%(here)s/auditlog.fs
```

Or if your application uses multiple processes, use a ZEO URL.

The database cannot be your main database. The reason that the audit database must live in a separate ZODB database is that we don't want undo operations to undo the audit log data.

Note that if you do not configure an audit database, real-time SDI features such as your folder contents views updating without a manual refresh will not work.

Once you've configured the audit database, you need to add an audit log object to the new database. You can do this using pshell:

```
[chrism@thinko sdnet]$ bin/pshell etc/development.ini
Python 3.3.2 (default, Jun  1 2013, 04:46:52)
[GCC 4.6.3] on linux
Type "help" for more information.

Environment:
  app          The WSGI application.
  registry     Active Pyramid registry.
  request      Active request object.
  root         Root of the default resource tree.
  root_factory Default root factory used to create `root`.

>>> from substanced.audit import set_auditlog
>>> set_auditlog(root)
>>> import transaction; transaction.commit()
```

Once you've done this, the "Auditing" tab of the root object in the SDI should no longer indicate that auditing is not configured.

4.14.2 Viewing the Audit Log

The root object will have a tab named "Auditing". You can view the currently active audit log entries from this page. Accessing this tab requires the `sdi.view-auditlog` permission.

4.14.3 Adding an Audit Log Entry

Here's an example of adding an audit log entry of type `NailsFiled` to the audit log:

```
from substanced.util import get_oid, get_auditlog

def myview(context, request):
    auditlog = get_auditlog(context)
    auditlog.add('NailsFiled', get_oid(context), type='fingernails')
    ...
```

Warning: If you don't have an audit database defined, the *`get_auditlog()`* API will return `None`.

This will add a``NailsFiled`` event with the payload `{'type':'fingernails'}` to the audit log. The payload will also automatically include a UNIX timestamp as the key `time`. The first argument is the audit log typename. Audit entries of the same kind should share the same type name. It should be a string. The second argument is the oid of the content object which this event is related to. It may be `None` indicating that the event is global, and unrelated to any particular piece of content. You can pass any number of keyword arguments to *`substanced.audit.AuditLog.add()`*, each will be added to the payload. Each value supplied as a keyword argument *must* be JSON-serializable. If one is not, you will receive an error when you attempt to add the event.

4.14.4 Using The `auditstream-sse` View

If you have auditing enabled, you can use a view named `auditstream-sse` against any resource in your resource tree using JavaScript. It will return an event stream suitable for driving an HTML5 `EventSource` (an HTML 5 feature, see http://www.html5rocks.com/en/tutorials/eventsource/basics/ for more information). The event stream will contain auditing events. This can be used for progressive enhancement of your application's UI. Substance D's SDI uses it for that purpose. For example, when an object's ACL is changed, a user looking at the "Security" tab of that object in the SDI will see the change immediately, rather than upon the next page refresh.

Obtain events for the context of the view only:

```
var source = new EventSource(
   "${request.sdiapi.mgmt_path(context, 'auditstream-sse')}");
```

Obtain events for a single OID unrelated to the context:

```
var source = new EventSource(
   "${request.sdiapi.mgmt_path(context, 'auditstream-sse', query={'oid':'12345'})}");
```

Obtain events for a set of OIDs:

```
var source = new EventSource(
   "${request.sdiapi.mgmt_path(context, 'auditstream-sse', query={'oid':['12345',
↪'56789']})}");
```

Obtain all events for all oids:

```
var source = new EventSource(
   "${request.sdiapi.mgmt_path(context, 'auditstream-sse', query={'all':'1'})}");
```

The executing user will need to possess the `sdi.view-auditstream` permission against the context on which the view is invoked. Each event payload will contain detailed information about the audit event as a string which represents a JSON dictionary.

See the `acl.pt` template in the `substanced/sdi/views/templates` directory of Substance D to see a "real-world" usage of this feature.

4.15 Using Locking

Substance D allows you to lock content resources programmatically. When a resource is locked, its UI can change to indicate that it cannot be edited by someone other than the user holding the lock.

Locking a resource *only* locks the resource, not its children. The locking system is not recursive at this time.

4.15.1 Locking a Resource

To lock a resource:

```
from substanced.locking import lock_resource
from pyramid.security import has_permission

if has_permission('sdi.lock', someresource, request):
    lock_resource(someresource, request.user, timeout=3600)
```

If the resource is already locked by the owner supplied as `owner_or_ownerid` (the parameter filled by `request.user` above), calling this function will refresh the lock. If the resource is not already locked by another user, calling this function will create a new lock. If the resource is already locked by a different user, a *`substanced.locking.LockError`* will be raised.

Using the *`substanced.locking.lock_resource()`* function has the side effect of creating a "Lock Service" (named `locks`) in the Substance D root if one does not already exist.

Warning: Callers should assert that the owner has the `sdi.lock` permission against the resource before calling *`lock_resource()`* to ensure that a user can't lock a resource he is not permitted to.

4.15.2 Unlocking a Resource

To unlock a resource:

```
from substanced.locking import unlock_resource
from pyramid.security import has_permission

if has_permission('sdi.lock', someresource, request):
    unlock_resource(someresource, request.user)
```

If the resource is already locked by a user other than the owner supplied as `owner_or_ownerid` (the parameter filled by `request.user` above) or the resource isn't already locked with this lock type, calling this function will raise a *`substanced.locking.UnlockError`* exception. Otherwise the lock will be removed.

Using the *`substanced.locking.unlock_resource()`* function has the side effect of creating a "Lock Service" (named `locks`) in the Substance D root if one does not already exist.

Warning: Callers should assert that the owner has the `sdi.lock` permission against the resource before calling *`unlock_resource()`* to ensure that a user can't lock a resource he is not permitted to.

To unlock a resource using an explicit lock token:

```
from substanced.locking import unlock_token
from pyramid.security import has_permission

if has_permission('sdi.lock', someresource, request):
    unlock_token(someresource, token, request.user)
```

If the lock identified by `token` belongs to a user other than the owner supplied as `owner_or_ownerid` (the parameter filled by `request.user` above) or if no lock exists under `token`, calling this function will raise a *`substanced.locking.LockError`* exception. Otherwise the lock will be removed.

Using the `substanced.locking.unlock_token()` function has the side effect of creating a "Lock Service" (named `locks`) in the Substance D root if one does not already exist.

Warning: Callers should assert that the owner has the `sdi.lock` permission against the resource before calling `unlock_token()` to ensure that a user can't lock a resource he is not permitted to.

4.15.3 Discovering Existing Locks

To discover any existing locks for a resource:

```
from substanced.locking import discover_resource_locks

locks = discover_resource_locks(someresource)
# "locks" will be a sequence
```

The *substanced.locking.discover_resource_locks()* function will return a sequence of *substanced.locking.Lock* objects related to the resource for the lock type provided to the function. By default, only valid locks are returned. Invalid locks for the resource may exist, but they are not returned unless the `include_invalid` argument passed to :*discover_resource_locks()* is `True`.

Under normal circumstances, the length of the sequence returned will be either 0 (if there are no locks) or 1 (if there is any lock). In some special circumstances, however, when the *substanced.locking.lock_resource()* API is not used to create locks, there may be more than one lock related to a resource of the same type.

By default, the `discover_resource_locks` API returns locks for the provided object, plus locks on any object in its lineage. To suppress this default, pass `include_lineage=False`, e.g.:

```
locks = discover_resource_locks(someresource)
# "locks" will be only those set on 'someresource'
```

In some applications, the important thing is to ensure that a particular user *could* lock a resource before updating it (e.g., from a browser view on a property sheet). The :`could_lock_resource()` API is designed for these cases: if the supplied userid could not lock the resource, it raises a *substanced.locking.LockError* exception:

```
from substanced.locking import could_lock_resource, LockError

try:
    could_lock_resource(someresource, request.user)
except LockError as e:
    raise FormError('locked by "%s"' % e.lock.owner.__name__)
```

4.15.4 Viewing The Lock Service

Once some locks have been created, a *lock service* will have been created. The lock service is an object named `locks` in the Substance D root.

You can use the SDI UI of this locks service to delete and edit existing locks. It's a good idea to periodically use the "Delete Expired" button in this UI to clear out any existing expired locks that were orphaned by buggy or interrupted clients.

4.16 Configuration

While writing a Substance D application is very similar to writing a Pyramid application, there are a few extra considerations to keep in mind.

4.16.1 Scan and Include

When writing Pyramid applications, the Configurator supports `config.include` and `config.scan` Because of ordering effects, do all your `config.include` calls before any of your `config.scan` calls.

4.16.2 Using RelStorage

Content in Substance D is stored in a Python object database called the ZODB. The ZODB has deep integration with Pyramid. When developing Python applications that use ZODB, you have a number of storage options:

- `FileStorage` is the simplest and is used in the development scaffolds for Substance D. That is, `development.ini` is configured use `FileStorage`. Just a file on disk, no long-running server process.
- `ZEO` keeps a file on disk but runs a server process that manages transactions over a socket. This allows multiple app servers on multiple boxes, or background processes such as deferred indexing, to access the database.
- RelStorage stores and retrieves the Python objects from a relational database. This is the preferred deployment option for applications that need trusted reliability and scalability.

Switching between storages is mostly a matter of editing your configuration file and choosing a different storage.

Note: While RelStorage uses an RDBMS for transactions, storage, retrieval, failover, and other features, it does *not* use SQL or decompose your Python objects into columns and joined tables.

Although RelStorage supports a number of RDBMS packages, we'll focus on PostgreSQL in these docs.

RelStorage + PostgreSQL Configuration

First, read the RelStorage docs, focusing on the PostgreSQL section and the command line needed for database setup. In particular, make sure that you:

- Have a system user account named `database`:

  ```
  $ sudo su - postgres
  $ createuser --pwprompt zodbuser
  $ createdb -O zodbuser zodb
  ```

- The user that you created (e.g. `zodbuser`) can make local connections

Next, we'll make some changes to some of the configuration files. In your `setup.py`, indicate that you need the `RelStorage` package as well as the `psycopg2` Python binding for PostgreSQL. This presumes that the binaries for the PostgreSQL client are available on your path.

In your configuration file (e.g. `production.ini`), the `[app:main]` section should have:

```
zodbconn.uri = zconfig://%(here)s/relstorage.conf
```

We thus need a `relstorage.conf` file:

```
%import relstorage
<zodb main>
  <relstorage>
    blob-dir ../var/blobs
    <postgresql>
      dsn dbname='zodb' user='zodbuser' host='localhost' password='zodbuser'
    </postgresql>
  </relstorage>
  cache-size 100000
</zodb>
```

Resetting Your Substance D Database

During development you frequently need to blow away all your data and start over. You can do this via evolution, but usually it isn't worth the work.

This is very easy with `FileStorage`: just `rm var/Data.fs*` and restart your app server. It is also easy with ZEO: shut down the supervisor service, remove the data as above, restart it, and restart the app server.

With RelStorage, you get a rich set of existing tools such as `pgadmin` to browse and modify table data. You can, though, do it the quickie way via `bin/pshell` and just delete the root object, then commit the transaction.

If you need to remove evolve data as well, open up pshell and do `root._p_jar.root()`. You'll see the *ZODB* root (not the app root). Inside of it is the app root and the evolve data.

4.17 Gathering Runtime Statistics

Problems can come up in production. When they do, you usually want forensics that show aspects of the system under load, over a period of time.

Of course, you don't want the collection of such data to affect performance. What's needed is a mechanism to log data all the time, in a lightweight way, that can later be analyzed in productive ways. This system needs both built-in hooks at the Substance D framework level as well as extension points to analyze function points in the application you are writing.

Three components are involved in the process of collecting statistics:

- *`substanced.stats`* exposes Python API to collect data and sends it to to a *StatsD <https://github.com/etsy/statsd>* agent
- The StatsD agent aggregates data and sends it to backend service
- A backend service displays graphs based on stored data. The service can be self-hosted such as Graphite or it can be a SaaS solution such as DataDog.

4.17.1 Setting Up

To enable statistics gathering in your site, edit your `.ini` configuration file and add the following lines to your `[app:main]` section:

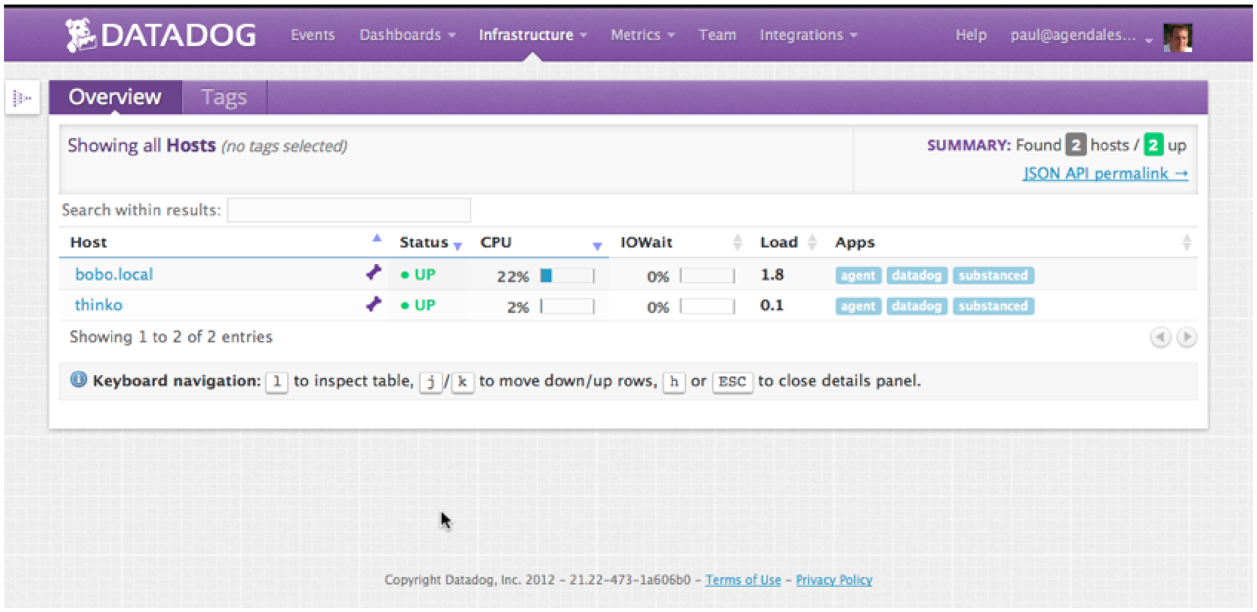
```
substanced.statsd.enabled = true
substanced.statsd.host = localhost
substanced.statsd.port = 8125
substanced.statsd.prefix = substanced
```

Using DataDog with SubstanceD statistics

Substance D supports *DataDog*, a Software-as-a-Service (SaaS) provider for monitoring and visualizing performance data. DataDog installs an *dogstatsd* agent for sending custom metrics on your local system. The agent is based on StatsD.

Using DataDog is an an easy way to get started with Substance D statistics. Sign up for an account with DataDog. This will provide you with the instructions for downloading and running the local agent. You'll need to get the agent installed before proceeding.

Once you've got the agent installed, and the proper settings in your Substance D ini file, you will be able to see statistics in the DataDog user interface. Once you log into your DataDog dashboard, click on `Infrastructure` and you'll see any hosts configured as part of your account:



Showing all **Hosts** *(no tags selected)*

SUMMARY: Found 2 hosts / 2 up

JSON API permalink →

Search within results:

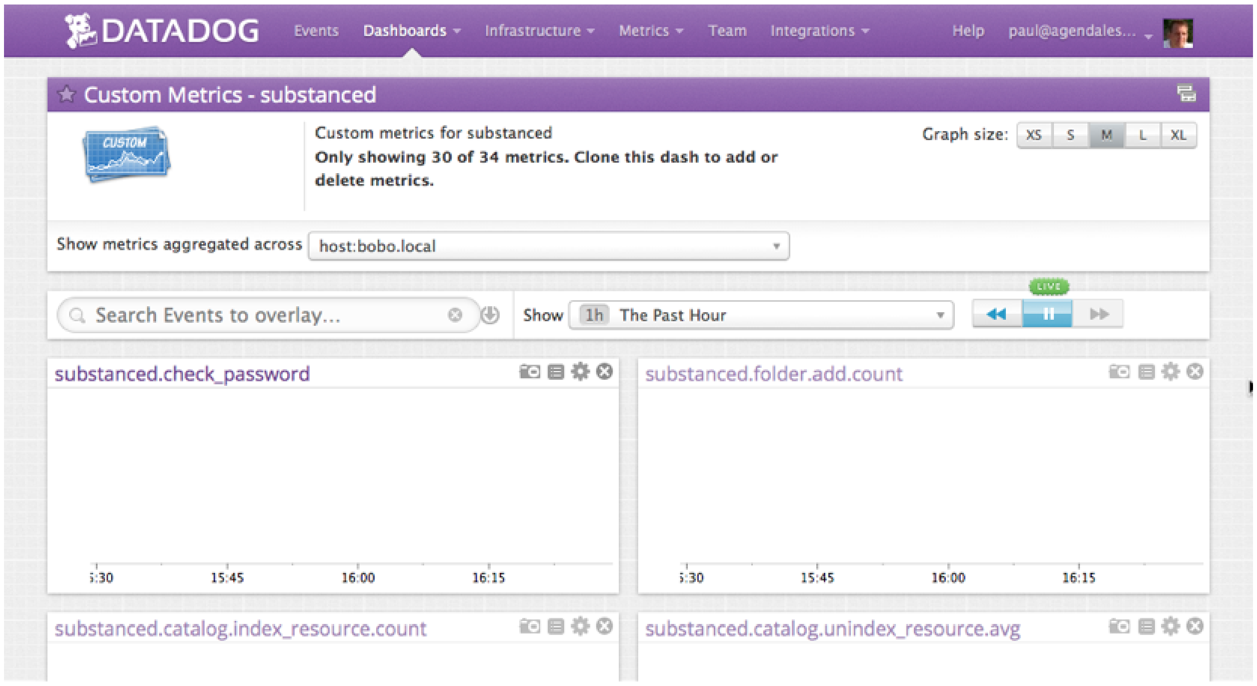
Host	Status	CPU	IOWait	Load	Apps
bobo.local	● UP	22%	0%	1.8	agent datadog substanced
thinko	● UP	2%	0%	0.1	agent datadog substanced

Showing 1 to 2 of 2 entries

Keyboard navigation: 1 to inspect table, j / k to move down/up rows, h or ESC to close details panel.

Copyright Datadog, Inc. 2012 – 21.22-473-1a606b0 – Terms of Use – Privacy Policy

The `substanced` entry in `Apps` table column is from the `substanced.statsd.prefix` configured in *Settings up* section. Clicking on that brings up Substance D specific monitoring in DataDog:



Custom Metrics - substanced

Custom metrics for substanced

Only showing 30 of 34 metrics. Clone this dash to add or delete metrics.

Show metrics aggregated across host:bobo.local

substanced.check_password

substanced.folder.add.count

substanced.catalog.index_resource.count

substanced.catalog.unindex_resource.avg

Clicking settings symbol on a graph will lead you to graph editor, where you can change how DataDog interprets and renders your graphs. A good resource how the editor works is Graphing Primer.

DataDog also supports Metric Alerts allowing you to send alerts when your statistics reach certain state.

Logging Custom Statistics

Over time, Substance D itself will include more framework points where statistics are collected. Most likely, though, you'll want some statistics that are very meaningful to your application's specific functionality.

If you look at the docs for the Python statsd module you will see three main types:

- *Counters* for simply incrementing a value,
- *Timers* for logging elapsed time in a code block, and
- *Gauges* for tracking a constant at a particular point in time

Each of these map to methods in `substanced.stats.StatsdHelper`. This class is available as an instance available via import:

```
from substanced.stats import statsd_gauge
```

Your application code can then make calls to these stats-gathering methods. For example, `substanced.principal.User` does the following to note that check password was used:

```
statsd_gauge('check_password', 1)
```

Here is an example in *`substanced.catalog.Catalog.index_resource()`* that measures elapsed indexing time inside a Python `with` block:

```
with statsd_timer('catalog.index_resource'):
    if oid is None:
        oid = oid_from_resource(resource)
    for index in self.values():
        index.index_resource(resource, oid=oid, action_mode=action_mode)
    self.objectids.insert(oid)
```

4.18 Virtual Rooting

You can present a folder other than the physical Substance D root object as the "SDI root" to people. For example, if you have the following structure from your physical Substance D root:

```
root--
      \-- folder1
      |
      |-- folder2
```

You can present either `folder1` or `folder2` to the user as a virtual root when people log in to the SDI.

To do so, you have to pass an `X-Vhm-Root` header to SubstanceD in each request. It's easiest to do this with Apache or another frontend web server. Here's a sample configuration which assumes you are telling Apache to proxy to a Substance D application that runs on localhost on port 6543:

```
<VirtualHost *:80>
    ServerAdmin webmaster@agendaless.com
    ServerName  example.com
    ErrorLog    /var/log/apache2/example.com-error.log
    CustomLog   /var/log/apache2/example.com-access.log combined
    RewriteEngine On
    RewriteRule ^(.*) http://127.0.0.1:6543/$1 [L,P]
```

(continues on next page)

(continued from previous page)

```
    ProxyPreserveHost On
    RequestHeader add X-Vhm-Root /folder1
</VirtualHost>
```

In the above configuration, when users log in on `http://example.com/manage`, the root they see in the SDI will be `/folder1` instead of the real root. They will not be able to access the real root.

Note that retail requests (requests without `/manage`) to the same hostname will *also* be rooted at `folder1`.

This feature requires Pyramid version 1.4.4 or better.

4.19 Building a Retail Application

It's not the intent that normal unprivileged users of an application you build using Substance D ever see the *SDI* management interface. That interface is reserved for privileged users, like you and your staff.

To build a "retail" application, you just use normal Pyramid view configuration to associate objects with view logic based on the content types provided to you by Substance D and the content types you've defined.

For example, here's a view that will respond on the root Substance D object and return its SDI title:

```python
from pyramid.view import view_config

@view_config(content_type='Root')
def hello(request):
    html = u'<html><head></head><body>Hello from %s!</body></html>'
    request.response.text = html % request.context.sdi_title
    return request.response
```

Note that we did *not* use the `substanced.sdi.mgmt_view` decorator. Instead we used the `pyramid.view.view_config` decorator, which will expose the view to normal site visitors, not just those visiting the resource via the *SDI*.

To see that code working, create a `retail` package within the `myproj` package (that is the inner `myproj` folder that contains the `__init__.py`, `resources.py` and `views.py` files). The package will have two files: an empty `__init__.py` and a `views.py` with the code snippet above. If you now visit `http://localhost:6543/` you will see the "Hello from..." message.

To display actual content stored in the database, Substance D exposes a *resource tree* that you can hang views from to build your application. You'll want to read up on traversal to understand how to associate view configuration with *resource* objects.

4.20 Substance D Command-Line Utilities

Substance D installs a number of helper scripts for performing admin-related tasks. To get full command-line syntax for any script, run it with the option `--help`.

4.20.1 `sd_adduser`

Add a new user, making them part of the 'admins' group. Useful when recovering from a forgotten password for the default 'admin' user. E.g.:

```
$ /path/to/virtualenv/bin/sd_adduser /path/to/virtualenv/etc/production.ini phred␣
↪password
```

4.20.2 `sd_drain_indexing`

Process deferred indexing actions. E.g., run this from a **`cron`** job to drain the queue every two minutes:

```
0-59/2 * * * * /path/to/virtualenv/bin/sd_drain_indexing /path/to/virtualenv/etc/
↪production.ini
```

4.20.3 `sd_dump`

Dump an object (and its subobjects) to the filesystem:

```
sd_dump [--source=ZODB-PATH] [--dest=FILESYSTEM-PATH] config_uri
Dumps the object at ZODB-PATH and all of its subobjects to a
filesystem path.  Such a dump can be loaded (programmatically)
by using the substanced.dump.load function
```

E.g.:

```
$ /path/to/virtualenv/bin/sd_dump --source=/ --dest=/tmp/dump /path/to/virtualenv/etc/
↪development.ini
```

4.20.4 `sd_evolve`

Query for pending evolution steps, or run them to get the database up-to-date. See *Running an Evolution from the Command Line*.

4.20.5 `sd_reindex`

Reindex the catalog. E.g.:

```
$ /path/to/virtualenv/bin/sd_reindex /path/to/virtualenv/etc/development.ini
```

4.21 Installing `python-magic`

Use of the *`substanced.file.USE_MAGIC`* constant for guessing file types from stream content requires the `python-magic` library, which works without extra help on most systems, but may require special dependency installations on Mac OS and Windows systems. You'll need to follow these steps on those platforms to use this feature:

Mac OS X

  http://www.brambraakman.com/blog/comments/installing_libmagic_in_mac_os_x_for_python-magic/

Windows

  "Installation on Win32" in https://github.com/ahupp/python-magic

CHAPTER 5

API Documentation

5.1 `substanced` API

`substanced.`**`includeme`**`(`*config*`)`

Do the work of *`substanced.include()`*, then *`substanced.scan()`*. Makes `config.include(substanced)` work.

`substanced.`**`include`**`(`*config*`)`

Perform all `config.include` tasks required for Substance D and the default aspects of the SDI to work.

`substanced.`**`scan`**`(`*config*`)`

Perform all `config.scan` tasks required for Substance D and the default aspects of the SDI to work.

5.2 `substanced.audit` API

`class` `substanced.audit.`**`AuditLog`**`(`*max_layers=10*, *layer_size=100*, *entries=None*`)`

`add``(`*_name*, *_oid*, ***kw*`)`

Add a record the audit log. `_name` should be the event name, `_oid` should be an object oid or `None`, and `kw` should be a json-serializable dictionary

`latest_id``()`

Return the generation and the index id as a tuple, representing the latest audit log entry

`newer``(`*generation*, *index_id*, *oids=None*`)`

Return the events newer than the combination of `generation` and `oid`. Filter using `oids` if supplied.

5.3 `substanced.catalog` API

`class` `substanced.catalog.`**`Text`**`(`***kw*`)`

class substanced.catalog.**Field**(***kw*)

class substanced.catalog.**Keyword**(***kw*)

class substanced.catalog.**Facet**(***kw*)

class substanced.catalog.**Allowed**(***kw*)

class substanced.catalog.**Path**(***kw*)

class substanced.catalog.**Catalog**(*family=None*)

__setitem__(*name*, *other*)

Set object `other` into this folder under the name `name`.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding.

`name` cannot be the empty string.

When `other` is seated into this folder, it will also be decorated with a `__parent__` attribute (a reference to the folder into which it is being seated) and `__name__` attribute (the name passed in to this function. It must not already have a `__parent__` attribute before being seated into the folder, or an exception will be raised.

If a value already exists in the foldr under the name `name`, raise `KeyError`.

When this method is called, the object will be added to the objectmap, an *`substanced.event.ObjectWillBeAdded`* event will be emitted before the object obtains a `__name__` or `__parent__` value, then a *`substanced.event.ObjectAdded`* will be emitted after the object obtains a `__name__` and `__parent__` value.

__getitem__(*name*)

Return the object named `name` added to this folder or raise `KeyError` if no such object exists. `name` must be a Unicode object or directly decodeable to Unicode using the system default encoding.

Retrieve an index.

get(*name*, *default=None*)

Return the object named by `name` or the default.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding.

Retrieve an index or return failobj.

flush(*all=True*)

Flush pending indexing actions for all indexes in this catalog.

If `all` is `True`, all pending indexing actions will be immediately executed regardless of the action's mode.

If `all` is `False`, pending indexing actions which are *`MODE_ATCOMMIT`* will be executed but actions which are *`MODE_DEFERRED`* will not be executed.

index_resource(*resource*, *oid=None*, *action_mode=None*)

Register the resource in indexes of this catalog using `oid` as the indexing identifier. If `oid` is not supplied, the `__oid__` attribute of the `resource` will be used as the indexing identifier.

`action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`*, indicating when the updates should take effect. The `action_mode` value will overrule any action mode a member index has been configured with except `None` which explicitly indicates that you'd like to use the index's action_mode value.

reindex(*dry_run=False*, *commit_interval=3000*, *indexes=None*, *path_re=None*, *output=None*, *registry=None*)

Reindex all objects in the catalog using the existing set of indexes immediately.

If `dry_run` is `True`, do no actual work but send what would be changed to the logger.

`commit_interval` controls the number of objects indexed between each call to `transaction.commit()` (to control memory consumption).

`indexes`, if not `None`, should be a list of index names that should be reindexed. If `indexes` is `None`, all indexes are reindexed.

`path_re`, if it is not `None` should be a regular expression object that will be matched against each object's path. If the regular expression matches, the object will be reindexed, if it does not, it won't.

`output`, if passed should be one of `None`, `False` or a function. If it is a function, the function should accept a single message argument that will be used to record the actions taken during the reindex. If `False` is passed, no output is done. If `None` is passed (the default), the output will wind up in the `substanced.catalog` Python logger output at `info` level.

`registry`, if passed, should be a Pyramid registry. If one is not passed, the `get_current_registry()` function will be used to look up the current registry. This function needs the registry in order to access content catalog views.

reindex_resource(*resource*, *oid=None*, *action_mode=None*)

Register the resource in indexes of this catalog using `oid` as the indexing identifier. If `oid` is not supplied, the `__oid__` attribute of `resource` will be used as the indexing identifier.

`action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`* indicating when the updates should take effect. The `action_mode` value will overrule any action mode a member index has been configured with except `None` which explicitly indicates that you'd like to use the index's action_mode value.

The result of calling this method is logically the same as calling `unindex_resource`, then `index_resource` with the same resource, but calling those two methods in succession is often more expensive than calling this single method, as member indexes can choose to do smarter things during a reindex than what they would do during an unindex followed by a successive index.

reset()

Reset all indexes in this catalog and clear self.objectids.

transaction = <module 'transaction' from '/home/docs/checkouts/readthedocs.org/user_bu

unindex_resource(*resource_or_oid*, *action_mode=None*)

Deregister the resource in indexes of this catalog using the indexing identifier `resource_or_oid`. If `resource_or_oid` is an integer, it will be used as the indexing identifier; if `resource_or_oid` is a resource, its `__oid__` attribute will be used as the indexing identifier.

`action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`* indicating when the updates should take effect. The `action_mode` value will overrule any action mode a member index has been configured with except `None` which explicitly indicates that you'd like to use the index's action_mode value.

update_indexes(*registry=None*, *dry_run=False*, *output=None*, *replace=False*, *reindex=False*, ***reindex_kw*)

Use the candidate indexes registered via `config.add_catalog_factory` to populate this catalog. Any indexes which are present in the candidate indexes, but not present in the catalog will be created. Any indexes which are present in the catalog but not present in the candidate indexes will be deleted.

`registry`, if passed, should be a Pyramid registry. If one is not passed, the `get_current_registry()` function will be used to look up the current registry. This function needs the registry in order to access content catalog views.

If `dry_run` is `True`, don't commit the changes made when this function is called, just send what would have been done to the logger.

`output`, if passed should be one of `None`, `False` or a function. If it is a function, the function should accept a single message argument that will be used to record the actions taken during the reindex. If `False` is passed, no output is done. If `None` is passed (the default), the output will wind up in the `substanced.catalog` Python logger output at `info` level.

This function does not reindex new indexes added to the catalog unless `reindex=True` is passed.

Arguments to this method captured as `kw` are passed to *`substanced.catalog.Catalog.reindex()`* if `reindex` is True, otherwise `kw` is ignored.

If `replace` is `True`, an existing catalog index that is not in the `category` supplied but which has the same name as a candidate index will be replaced. If `replace` is `False`, existing indexes will never be replaced.

class substanced.catalog.**CatalogsService**(*data=None*, *family=None*)

class Catalog(*family=None*)

flush(*all=True*)

Flush pending indexing actions for all indexes in this catalog.

If `all` is `True`, all pending indexing actions will be immediately executed regardless of the action's mode.

If `all` is `False`, pending indexing actions which are *`MODE_ATCOMMIT`* will be executed but actions which are *`MODE_DEFERRED`* will not be executed.

index_resource(*resource*, *oid=None*, *action_mode=None*)

Register the resource in indexes of this catalog using `oid` as the indexing identifier. If `oid` is not supplied, the `__oid__` attribute of the `resource` will be used as the indexing identifier.

`action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`*, indicating when the updates should take effect. The `action_mode` value will overrule any action mode a member index has been configured with except `None` which explicitly indicates that you'd like to use the index's action_mode value.

reindex(*dry_run=False*, *commit_interval=3000*, *indexes=None*, *path_re=None*, *output=None*, *registry=None*)

Reindex all objects in the catalog using the existing set of indexes immediately.

If `dry_run` is `True`, do no actual work but send what would be changed to the logger.

`commit_interval` controls the number of objects indexed between each call to `transaction.commit()` (to control memory consumption).

`indexes`, if not `None`, should be a list of index names that should be reindexed. If `indexes` is `None`, all indexes are reindexed.

`path_re`, if it is not `None` should be a regular expression object that will be matched against each object's path. If the regular expression matches, the object will be reindexed, if it does not, it won't.

`output`, if passed should be one of `None`, `False` or a function. If it is a function, the function should accept a single message argument that will be used to record the actions taken during the reindex. If `False` is passed, no output is done. If `None` is passed (the default), the output will wind up in the `substanced.catalog` Python logger output at `info` level.

registry, if passed, should be a Pyramid registry. If one is not passed, the get_current_registry() function will be used to look up the current registry. This function needs the registry in order to access content catalog views.

reindex_resource(*resource*, *oid=None*, *action_mode=None*)

Register the resource in indexes of this catalog using oid as the indexing identifier. If oid is not supplied, the __oid__ attribute of resource will be used as the indexing identifier.

action_mode, if supplied, should be one of None, *MODE_IMMEDIATE*, *MODE_ATCOMMIT* or *MODE_DEFERRED* indicating when the updates should take effect. The action_mode value will overrule any action mode a member index has been configured with except None which explicitly indicates that you'd like to use the index's action_mode value.

The result of calling this method is logically the same as calling unindex_resource, then index_resource with the same resource, but calling those two methods in succession is often more expensive than calling this single method, as member indexes can choose to do smarter things during a reindex than what they would do during an unindex followed by a successive index.

reset()

Reset all indexes in this catalog and clear self.objectids.

transaction = <module 'transaction' from '/home/docs/checkouts/readthedocs.org/use

unindex_resource(*resource_or_oid*, *action_mode=None*)

Deregister the resource in indexes of this catalog using the indexing identifier resource_or_oid. If resource_or_oid is an integer, it will be used as the indexing identifier; if resource_or_oid is a resource, its __oid__ attribute will be used as the indexing identifier.

action_mode, if supplied, should be one of None, *MODE_IMMEDIATE*, *MODE_ATCOMMIT* or *MODE_DEFERRED* indicating when the updates should take effect. The action_mode value will overrule any action mode a member index has been configured with except None which explicitly indicates that you'd like to use the index's action_mode value.

update_indexes(*registry=None*, *dry_run=False*, *output=None*, *replace=False*, *reindex=False*, ***reindex_kw*)

Use the candidate indexes registered via config.add_catalog_factory to populate this catalog. Any indexes which are present in the candidate indexes, but not present in the catalog will be created. Any indexes which are present in the catalog but not present in the candidate indexes will be deleted.

registry, if passed, should be a Pyramid registry. If one is not passed, the get_current_registry() function will be used to look up the current registry. This function needs the registry in order to access content catalog views.

If dry_run is True, don't commit the changes made when this function is called, just send what would have been done to the logger.

output, if passed should be one of None, False or a function. If it is a function, the function should accept a single message argument that will be used to record the actions taken during the reindex. If False is passed, no output is done. If None is passed (the default), the output will wind up in the substanced.catalog Python logger output at info level.

This function does not reindex new indexes added to the catalog unless reindex=True is passed.

Arguments to this method captured as kw are passed to *substanced.catalog.Catalog.reindex()* if reindex is True, otherwise kw is ignored.

If replace is True, an existing catalog index that is not in the category supplied but which has the same name as a candidate index will be replaced. If replace is False, existing indexes will never be replaced.

add_catalog(*name*, *update_indexes=True*)

Create and add a catalog named `name` to this catalogs service. Return the newly created catalog object. If a catalog named `name` already exists in this catalogs service, an exception will be raised.

Example usage in a root created subscriber:

```
@subscribe_created(content_type='Root')
def created(event):
    root = event.object
    service = root['catalogs']
    catalog = service.add_catalog('app1', update_indexes=True)
```

If `update_indexes` is True, indexes in the named catalog factory will be added to the newly created catalog.

`substanced.catalog.`**is_catalogable**(*resource*, *registry=None*)

`substanced.catalog.`**catalog_factory**(*name*)

Decorator for a class which acts as a template for index creation.:

```
from substanced.catalog import Text

@catalog_factory('myapp')
class MyAppIndexes(object):
    text = Text()
    title = Field()
```

When scanned, this catalog factory will be added to the registry as if *`substanced.catalog.add_catalog_factory()`* were called like:

```
config.add_catalog_factory('myapp', MyAppIndexes)
```

`substanced.catalog.`**includeme**(*config*)

`substanced.catalog.`**add_catalog_factory**(*config*, *name*, *cls*)

Directive which adds a named catalog factory to the configuration state. The `cls` argument should be a class that has named index factory instances as attributes. The `name` argument should be a string.

`substanced.catalog.`**add_indexview**(*self*, **arg*, ***kw*)

Directive which adds an index view to the configuration state state. The `view` argument should be function that is an indeview function, or or a class with a `__call__` method that acts as an indexview method. For example:

```
def title(resource, default):
    return getattr(resource, 'title', default)

config.add_indexview(title, catalog_name='myapp', index_name='title')
```

Or, a class:

```
class IndexViews(object):
    def __init__(self, resource):
        self.resource = resource

    def __call__(self, default):
        return getattr(self.resource, 'title', default)

config.add_indexview(
    IndexViews, catalog_name='myapp', index_name='title'
    )
```

If an `attr` arg is supplied to `add_indexview`, you can use a different attribute of the class instad of `__call__`:

```
class IndexViews(object):
    def __init__(self, resource):
        self.resource = resource

    def title(self, default):
        return getattr(self.resource, 'title', default)

    def name(self, default):
        return getattr(self.resource, 'name', default)

config.add_indexview(
    IndexViews, catalog_name='myapp', index_name='title', attr='title'
    )
config.add_indexview(
    IndexViews, catalog_name='myapp', index_name='name', attr='name'
    )
```

In this way you can use the same class to represent a bunch of different index views. An index view will be looked up by the cataloging machinery when it wants to insert value into a particular catalog type's index. The `catalog_name` you use specify which catalog name this indeview is good for; it should match the string passed to `add_catalog_factory` as a `name`. The `index_name` argument should match an index name used within such a catalog.

Index view lookups work a bit like Pyramid view lookups: you can use the `context` argument to pass an interface or class which should be used to register the index view; such an index view will only be used when the resource being indexed has that class or interface. Eventually we'll provide a way to add predicates other than `context` too.

The *`substanced.catalog.indexview`* decorator provides a declarative analogue to using this configuration directive.

class `substanced.catalog.`**`indexview`**(***settings*)
A class decorator which, when applied to an index view class method, will mark the method as an index view. This decorator accepts all the arguments accepted by *`substanced.catalog.add_indexview()`*, and each has the same meaning.

class `substanced.catalog.`**`indexview_defaults`**(***settings*)
A class decorator which, when applied to a class, will provide defaults for all index view configurations defined in the class. This decorator accepts all the arguments accepted by *`substanced.catalog.indexview()`*, and each has the same meaning.

5.4 `substanced.catalog.indexes` API

class `substanced.catalog.indexes.`**`FieldIndex`**(*discriminator=None*, *family=None*, *action_mode=None*)

class `substanced.catalog.indexes.`**`KeywordIndex`**(*discriminator=None*, *family=None*, *action_mode=None*)

class `substanced.catalog.indexes.`**`TextIndex`**(*discriminator=None*, *lexicon=None*, *index=None*, *family=None*, *action_mode=None*)

class `substanced.catalog.indexes.`**`FacetIndex`**(*discriminator=None*, *facets=None*, *family=None*, *action_mode=None*)

class `substanced.catalog.indexes.`**`PathIndex`**(*discriminator=None*, *family=None*)

Uses the `substanced.objectmap.ObjectMap.pathlookup()` to apply a query to retrieve object identifiers at or under a path.

path can be passed to methods as:

- resource object
- tuple of strings (usually returned value of `pyramid.traverse.resource_path_tuple()`)
- a string path (e.g. /foo/bar)

Query methods accept following parameters:

- *include_origin* (by default True), see `substanced.objectmap.ObjectMap.pathlookup()` for explanation.
- *depth* (by default None) see `substanced.objectmap.ObjectMap.pathlookup()` for explanation.

Query types supported:

- Eq
- NotEq

class `substanced.catalog.indexes.`**`AllowedIndex`**(*discriminator*, *family=None*)

An index which defers to `objectmap.allowed` as part of a query intersection.

`allows`(*principals*, *permission*)

`principals` may either be 1) a sequence of principal indentifiers, 2) a single principal identifier, or 3) a Pyramid request, which indicates that all the effective principals implied by the request are used.

`permission` must be a permission name.

5.5 `hypatia.query` API

5.5.1 Comparators

class `hypatia.query.`**`Contains`**(*index*, *value*)

Contains query.

CQE equivalent: 'foo' in index

class `hypatia.query.`**`Eq`**(*index*, *value*)

Equals query.

CQE equivalent: index == 'foo'

class `hypatia.query.`**`NotEq`**(*index*, *value*)

Not equal query.

CQE eqivalent: index != 'foo'

class `hypatia.query.`**`Gt`**(*index*, *value*)

Greater than query.

CQE equivalent: index > 'foo'

class `hypatia.query.`**`Lt`**(*index*, *value*)

Less than query.

CQE equivalent: index < 'foo'

class `hypatia.query.`**`Ge`**(*index*, *value*)
Greater (or equal) query.

CQE equivalent: index >= 'foo'

class `hypatia.query.`**`Le`**(*index*, *value*)
Less (or equal) query.

CQE equivalent: index <= 'foo

class `hypatia.query.`**`Contains`**(*index*, *value*)
Contains query.

CQE equivalent: 'foo' in index

class `hypatia.query.`**`NotContains`**(*index*, *value*)
CQE equivalent: 'foo' not in index

class `hypatia.query.`**`Any`**(*index*, *value*)
Any of query.

CQE equivalent: index in any(['foo', 'bar'])

class `hypatia.query.`**`NotAny`**(*index*, *value*)
Not any of query (ie, None of query)

CQE equivalent: index not in any(['foo', 'bar'])

class `hypatia.query.`**`All`**(*index*, *value*)
All query.

CQE equivalent: index in all(['foo', 'bar'])

class `hypatia.query.`**`NotAll`**(*index*, *value*)
NotAll query.

CQE equivalent: index not in all(['foo', 'bar'])

class `hypatia.query.`**`InRange`**(*index*, *start*, *end*, *start_exclusive=False*, *end_exclusive=False*)
Index value falls within a range.

CQE eqivalent: lower < index < upper lower <= index <= upper

class `hypatia.query.`**`NotInRange`**(*index*, *start*, *end*, *start_exclusive=False*, *end_exclusive=False*)
Index value falls outside a range.

CQE eqivalent: not(lower < index < upper) not(lower <= index <= upper)

5.5.2 Boolean Operators

class `hypatia.query.`**`Or`**(**queries*)
Boolean Or of multiple queries.

class `hypatia.query.`**`And`**(**queries*)
Boolean And of multiple queries.

class `hypatia.query.`**`Not`**(*query*)
Negation of a query.

5.5.3 Other Helpers

class hypatia.query.**Name**(*name*)

A variable name in an expression, evaluated at query time. Can be used to defer evaluation of variables used inside of expressions until query time.

Example:

```
from hypatia.query import Eq
from hypatia.query import Name

# Define query at module scope
find_cats = Eq('color', Name('color')) & Eq('sex', Name('sex'))

# Use query in a search function, evaluating color and sex at the
# time of the query
def search_cats(catalog, resolver, color='tabby', sex='female'):
    # Let resolver be some function which can retrieve a cat object
    # from your application given a docid.
    params = dict(color=color, sex=sex)
    count, docids = catalog.query(find_cats, params)
    for docid in docids:
        yield resolver(docid)
```

hypatia.query.**parse_query**(*expr*, *catalog*, *optimize_query=True*)

Parses the given expression string and returns a query object. Requires Python >= 2.6.

5.6 `hypatia.util` API

class hypatia.util.**ResultSet**(*ids*, *numids*, *resolver*, *sort_type=None*)

Implements `hypatia.interfaces.IResultSet`

intersect(*docids*)

Intersect this resultset with a sequence of docids or another resultset. Returns a new ResultSet.

interface hypatia.interfaces.**IResultSet**

Iterable sequence of documents or document identifiers.

sort(*index*, *reverse=False*, *limit=None*, *sort_type=None*, *raise_unsortable=True*)

Return another IResultSet sorted using the `index` (an IIndexSort) passed to it after performing the sort using the index and the `limit`, `reverse`, and `sort_type` parameters.

If `sort_type` is not `None`, it should be the value `hypatia.interfaces.STABLE` to specify that the sort should be stable or `hypatia.interfaces.OPTIMAL` to specify that the sort algorithm chosen should be optimal (but not necessarily stable). It's usually unnecessary to pass this value, even if you're resorting an already-sorted set of docids, because the implementation of IResultSet will internally ensure that subsequent sorts of the returned result set of an initial sort will be stable; if you don't want this behavior, explicitly pass `hypatia.interfaces.OPTIMAL` on the second and subsequent sorts of a set of docids.

If `raise_unsortable` is `True` (the default), if the index cannot resolve any of the docids in the set of docids in this result set, a `hypatia.exc.Unsortable` exception will be raised during iteration over the sorted docids.

all(*resolve=True*)

Return a sequence representing all elements in the resultset. *If ``resolve`* is True, and the result set has a valid resolver, return an iterable of the resolved documents, otherwise return an iterable containing the document id of each document.

`ids`
An iterable sequence of document identifiers

`one`(*resolve=True*)
Return the element in the resultset, asserting that there is only one result. If the resultset has more than one element, raise an `hypatia.exc.MultipleResults` exception. If the resultset has no elements, raise an `hypatia.exc.NoResults` exception. *If ``resolve`* is True, and the result set has a valid resolver, return the resolved document, otherwise return the document id of the document.

`__iter__`()
Return an iterator over the results of `self.all()`

`resolver`
A callable which accepts a document id and which returns a document. May be `None`, in which case, resolution performed by result set methods is not performed, and document identifiers are returned unresolved.

`__len__`()
Return the length of the result set

`first`(*resolve=True*)
Return the first element in the sequence. If `resolve` is True, and the result set has a valid resolver, return the resolved document, otherwise return the document id of the first document.

5.7 `substanced.content` API

class `substanced.content.`**`content`**(*content_type*, *factory_type=None*, ***meta*)
Use as a decorator for a content factory (usually a class). Accepts a content type, a factory type (optionally), and a set of meta keywords. These values mean the same thing as they mean for *`substanced.content.add_content_type()`*. This decorator attaches information to the object it decorates which is used to call *`add_content_type()`* during a scan.

class `substanced.content.`**`service`**(*content_type*, *factory_type=None*, ***meta*)
This class is meant to be used as a decorator for a content factory that creates a service object (aka a service factory). A service object is an instance of a content type that can be looked up by name and which provides a service to application code. Services have well-known names within a folder. For example, the `principals` service within a folder is 'the principals service', the `catalog` object within a folder is 'the catalog service' and so on.

This decorator accepts a content type, a factory type (optionally), and a set of meta keywords. These values mean the same thing as they mean for the *`substanced.content.content`* decorator and *`substanced.content.add_content_type()`*. The decorator attaches information to the object it decorates which is used to call *`add_content_type()`* during a scan.

There is only one difference between using the *`substanced.content.content`* decorator and the `substanced.service.service` decorator. The `service` decorator honors a `service_name` keyword argument. If this argument is passed, and a service already exists in the folder by this name, the service will not be shown as addable in the add-content dropdown in the SDI UI.

`substanced.content.`**`add_content_type`**(*config*, *content_type*, *factory*, *factory_type=None*, ***meta*)
Configurator directive method which register a content type factory with the Substance D type system. Call via `config.add_content_type` during Pyramid configuration phase.

`content_type` is a hashable object (usually a string) representing the content type.

`factory` is a class or function which produces a content instance. It must be a *global object* (e.g. it cannot be a callable which is a method of a class or a callable instance). If `factory` is a function rather than a class, a *factory wrapper* is used (see below).

`**meta` is an arbitrary set of keywords associated with the content type in the content registry.

Some of the keywords in `**meta` have special meaning:

- If `meta` contains the keyword `propertysheets`, the content type will obtain a tab in the SDI that allows users to manage its properties.
- If `meta` contains the keyword `icon`, this value will be used as the icon for the content type that shows up next to the content in a folder content view.

Other keywords in `meta` will just be stored, and have no special meaning.

`factory_type` is an optional argument that can be used if the same factory must be used for two different content types; it is used during content type lookup (e.g. `substanced.util.get_content_type()`) to figure out which content type a constructed object is an instance of; it only needs to be used when the same factory is used for two different content types. Note that two content types cannot have the same factory type, unless it is `None`.

If `factory_type` is passed, the supplied factory will be wrapped in a factory wrapper which adds a `__factory_type__` attribute to the constructed instance. The value of this attribute will be used to determine the content type of objects created by the factory.

If the factory is a function rather than a class, a factory wrapper is used unconditionally.

The upshot wrt to `factory_type`: if your factory is a class and you pass a `factory_type` *or* if your factory is a function, you won't be able to successfully use the 'bare' factory callable to construct an instance of this content in your code, because the resulting instance will not have a `__factory_type__` attribute. Instead, you'll be required to use `substanced.content.Content.create()` to create an instance of the object with a proper `__factory_type__` attribute. But if your factory is a class, and you don't pass `factory_type` (the 'garden path'), you'll be able to use the class' constructor directly in your code to create instances of your content objects, which is more convenient and easier to unit test.

`substanced.content.`**`add_service_type`**(*config*, *content_type*, *factory*, *factory_type=None*, ***meta*)

Configurator directive method which registers a service factory. Call via `config.add_service_type` during Pyramid configuration phase. All arguments mean the same thing as they mean for the `substanced.content.add_content_type`.

A service factory is a special kind of content factory. A service factory creates a service object. A service object is an instance of a content type that can be looked up by name and which provides a service to application code. Services often have well-known names within the services folder. For example, the `principals` object within a services folder is 'the principals service', the `catalog` object within a services folder is 'the catalog service' and so on.

There is only one difference between using the `substanced.content.add_content_type` function and the `substanced.service.add_service_type` decorator. The `add_service_type` function honors a `service_name` keyword argument in its `**meta`. If this argument is passed, and a service already exists in a folder by this name, the service will not be shown as addable in the add-content dropdown in the SDI UI of the folder.

class `substanced.content.`**`ContentRegistry`**(*registry*)

An object accessible as `registry.content` (aka `request.registry.content`, aka `config.registry.content`) that contains information about Substance D content types.

`add`(*content_type*, *factory_type*, *factory*, ***meta*)

Add a content type to this registry

`all``()`
Return all content types known my this registry as a sequence.

`create``(`*content_type*, **arg*, ***kw*`)`
Create an instance of `content_type` by calling its factory with `*arg` and `**kw`. If the meta of the content type has an `after_create` value, call it (if it's a string, it's assumed to be a method of the created object, and if it's a sequence, each value should be a string or a callable, which will be called in turn); then send a `substanced.event.ContentCreatedEvent`. Return the created object.

If the key `__oid` is in the `kw` arguments, it will be used as the created object's oid.

`exists``(`*content_type*`)`
Return `True` if `content_type` has been registered within this content registry, `False` otherwise.

`factory_type_for_content_type``(`*content_type*`)`
Return the factory_type value for the content_type requested

`find``(`*resource*, *content_type*`)`
Return the first object in the lineage of the `resource` that supplies the `content_type` or `None` if no such object can be found.

See also `pyramid.traversal.find_interface()` to find object by an interface or a class.

`istype``(`*resource*, *content_type*`)`
Return `True` if `resource` is of content type `content_type`, `False` otherwise.

`metadata``(`*resource*, *name*, *default=None*`)`
Return a metadata value for the content type of `resource` based on the `**meta` value passed to `add()`. If a value in that content type's metadata was passed using `name` as its name, the value will be returned, otherwise `default` will be returned.

`typeof``(`*resource*`)`
Return the content type of `resource`

`substanced.content.`**`includeme`**`(`*config*`)`

5.8 `substanced.db` API

`substanced.db.`**`root_factory`**`(`*request*, *t=<module 'transaction' from '/home/docs/checkouts/readthedocs.org/user_builds/substanced/envs/latest/local/lib/python2.7/ packages/transaction/__init__.pyc'>*, *g=<function get_connection>*, *mark_unfinished_as_finished=<function mark_unfinished_as_finished>*`)`
A function which can be used as a Pyramid `root_factory`. It accepts a request and returns an instance of the `Root` content type.

5.9 `substanced.dump` API

`substanced.db.`**`includeme`**`(`*config*`)`

5.10 `substanced.editable` API

`interface` `substanced.editable.`**`IEditable`**
Adapter interface for editing content as a file.

put (*fileish*)
Update context based on the contents of `fileish`.

- `fileish` is a file-type object: its `read` method should return the (new) file representation of the context.

get ()
Return `(body_iter, mimetype)` representing the context.

- `body_iter` is an iterable, whose chunks are bytes represenating the context as an editable file.
- `mimetype` is the MIMEType corresponding to `body_iter`.

class `substanced.editable.`**FileEditable** (*context*, *request*)
IEditable adapter for stock SubstanceD 'File' objects.

`substanced.editable.`**register_editable_adapter** (*config*, *adapter*, *iface*)
Configuration directive: register `IEditable` adapter for `iface`.

- `adapter` is the adapter factory (a class or other callable taking `(context, request)`).
- `iface` is the interface / class for which the adapter is registered.

`substanced.editable.`**get_editable_adapter** (*context*, *request*)
Return an editable adapter for the context

Return `None` if no editable adapter is registered.

5.11 `substanced.event` API

class `substanced.event.`**ObjectAdded** (*object*, *parent*, *name*, *duplicating=False*, *moving=False*, *loading=False*)
An event sent just after an object has been added to a folder.

class `substanced.event.`**ObjectWillBeAdded** (*object*, *parent*, *name*, *duplicating=False*, *moving=False*, *loading=False*)
An event sent just before an object has been added to a folder.

class `substanced.event.`**ObjectRemoved** (*object*, *parent*, *name*, *removed_oids*, *moving=False*, *loading=False*)
An event sent just after an object has been removed from a folder.

class `substanced.event.`**ObjectWillBeRemoved** (*object*, *parent*, *name*, *moving=False*, *loading=False*)
An event sent just before an object has been removed from a folder.

removed_oids
Helper property that caches oids that will be removed as the result of this event. Will return an empty sequence if objectmap cannot be found on self.parent.

class `substanced.event.`**ObjectModified** (*object*)
An event sent when an object has been modified.

class `substanced.event.`**ACLModified** (*object*, *old_acl*, *new_acl*)

class `substanced.event.`**LoggedIn** (*login*, *user*, *context*, *request*)

class `substanced.event.`**RootAdded** (*object*)

class `substanced.event.`**AfterTransition** (*object*, *old_state*, *new_state*, *transition*)
Event sent after any workflow transition happens

class `substanced.event.`**`subscribe_added`**(*obj=None*, *container=None*, ***predicates*)
Decorator for registering an object added event subscriber (a subscriber for ObjectAdded).

`event = <InterfaceClass substanced.interfaces.IObjectAdded>`

class `substanced.event.`**`subscribe_removed`**(*obj=None*, *container=None*, ***predicates*)
Decorator for registering an object removed event subscriber (a subscriber for ObjectRemoved).

`event = <InterfaceClass substanced.interfaces.IObjectRemoved>`

class `substanced.event.`**`subscribe_will_be_added`**(*obj=None*, *container=None*, ***predicates*)
Decorator for registering an object will-be-added event subscriber (a subscriber for ObjectWillBeAdded).

`event = <InterfaceClass substanced.interfaces.IObjectWillBeAdded>`

class `substanced.event.`**`subscribe_will_be_removed`**(*obj=None*, *container=None*, ***predicates*)
Decorator for registering an object will-be-removed event subscriber (a subscriber for ObjectWillBeRemoved).

`event = <InterfaceClass substanced.interfaces.IObjectWillBeRemoved>`

class `substanced.event.`**`subscribe_modified`**(*obj=None*, ***predicates*)
Decorator for registering an object modified event subscriber (a subscriber for ObjectModified).

`event = <InterfaceClass substanced.interfaces.IObjectModified>`

class `substanced.event.`**`subscribe_acl_modified`**(*obj=None*, ***predicates*)
Decorator for registering an acl modified event subscriber (a subscriber for ObjectModified).

`event = <InterfaceClass substanced.interfaces.IACLModified>`

class `substanced.event.`**`subscribe_logged_in`**(***predicates*)
Decorator for registering an event listener for when a user is logged in

`event = <InterfaceClass substanced.interfaces.ILoggedIn>`

class `substanced.event.`**`subscribe_root_added`**(***predicates*)
Decorator for registering an event listener for when a root object has a database connection

`event = <InterfaceClass substanced.interfaces.IRootAdded>`

class `substanced.event.`**`subscribe_after_transition`**(***predicates*)
Decorator for registering an event listener for when a transition has been done on an object

`event = <InterfaceClass substanced.interfaces.IAfterTransition>`

5.12 `substanced.evolution` API

`substanced.evolution.`**`add_evolution_step`**(*config*, *func*, *before=None*, *after=None*, *name=None*)
A configurator directive which adds an evolution step. An evolution step can be used to perform upgrades or migrations of data structures in existing databases to meet expectations of new code.

`func` should be a function that performs the evolution logic. It should accept two arguments (conventionally-named) `root` and `registry. ``root` will be the root of the ZODB used to serve your Substance D site, and `registry` will be the Pyramid application registry.

`before` should either be `None`, another evolution step function, or the dotted name to such a function. By default, it is `None`, which means execute in the order defined by the calling order of `add_evolution_step`.

`after` should either be `None`, another evolution step function, or the dotted name to such a function. By default, it is `None`.

`name` is the name of the evolution step. It must be unique between all registered evolution steps. If it is not provided, the dotted name of the function used as `func` will be used as the evolution step name.

`substanced.evolution.`**`mark_unfinished_as_finished`**`(`*app_root*, *registry*, *t=None*`)`
Given the root object of a Substance D site as `app_root` and a Pyramid registry, mark all pending evolution steps as completed without actually executing them.

`substanced.evolution.`**`includeme`**`(`*config*`)`

5.13 `substanced.file` API

`substanced.file.`**`USE_MAGIC`**
A constant value used as an argument to various methods of the *`substanced.file.File`* class.

`class` `substanced.file.`**`File`**`(`*stream=None*, *mimetype=None*, *title=u''*`)`

`__init__``(`*stream=None*, *mimetype=None*, *title=u''*`)`
The constructor of a File object.

`stream` should be a filelike object (an object with a `read` method that takes a size argument) or `None`. If stream is `None`, the blob attached to this file object is created empty.

`title` must be a string or Unicode object.

`mimetype` may be any of the following:

- `None`, meaning set this file object's mimetype to `application/octet-stream` (the default).
- A mimetype string (e.g. `image/gif`)
- The constant *`substanced.file.USE_MAGIC`*, which will derive the mimetype from the stream content (if `stream` is also supplied) using the `python-magic` library.

Warning: On non-Linux systems, successful use of *`substanced.file.USE_MAGIC`* requires the installation of additional dependencies. See *Installing python-magic*.

`blob`
The ZODB blob object associated with this file.

`mimetype`
The mimetype of this file object (a string).

`get_etag``()`
Return a token identifying the "version" of the file.

`get_response``(`***kw*`)`
Return a WebOb-compatible response object which uses the blob content as the stream data and the mimetype of the file as the content type. The `**kw` arguments will be passed to the `pyramid.response.FileResponse` constructor as its keyword arguments.

`get_size``()`
Return the size in bytes of the data in the blob associated with the file

`upload``(`*stream*, *mimetype_hint=None*`)`
Replace the current contents of this file's blob with the contents of `stream`. `stream` must be a filelike object (it must have a `read` method that takes a size argument).

`mimetype_hint` can be any of the following:

- `None`, meaning don't reset the current mimetype. This is the default. If you already know the file's mimetype, and you don't want it divined from a filename or stream content, use `None` as the `mimetype_hint` value, and set the `mimetype` attribute of the file object directly before or after calling this method.
- A string containing a filename that has an extension; the mimetype will be derived from the extension in the filename using the Python `mimetypes` module, and the result will be set as the mimetype attribute of this object.
- The constant *`substanced.file.USE_MAGIC`*, which will derive the mimetype using the `python-magic` library based on the stream's actual content. The result will be set as the mimetype attribute of this object.

> **Warning:** On non-Linux systems, successful use of *`substanced.file.USE_MAGIC`* requires the installation of additional dependencies. See *Installing python-magic*.

5.14 `substanced.folder` API

class substanced.folder.**FolderKeyError**

class substanced.folder.**Folder**(*data=None*, *family=None*)

A folder implementation which acts much like a Python dictionary.

Keys must be Unicode strings; values must be arbitrary Python objects.

`__init__`(*data=None*, *family=None*)

Constructor. Data may be an initial dictionary mapping object name to object.

`order`

A tuple of name values. If set, controls the order in which names should be returned from `__iter__()`, `keys()`, `values()`, and `items()`. If not set, use an effectively random order.

`add`(*name*, *other*, *send_events=True*, *reserved_names=()*, *duplicating=None*, *moving=None*, *loading=False*, *registry=None*)

Same as `__setitem__`.

If `send_events` is False, suppress the sending of folder events. Don't allow names in the `reserved_names` sequence to be added.

If `duplicating` not `None`, it must be the object which is being duplicated; a result of a non-`None` duplicating means that oids will be replaced in objectmap. If `moving` is not `None`, it must be the folder from which the object is moving; this will be the `moving` attribute of events sent by this function too. If `loading` is `True`, the `loading` attribute of events sent as a result of calling this method will be `True` too.

This method returns the name used to place the subobject in the folder (a derivation of `name`, usually the result of `self.check_name(name)`).

`add_service`(*name*, *obj*, *registry=None*, ***kw*)

Add a service to this folder named `name`.

`check_name`(*name*, *reserved_names=()*)

Perform all the validation checks implied by *`validate_name()`* against the `name` supplied but also fail with a *`FolderKeyError`* if an object with the name `name` already exists in the folder.

`clear`(*registry=None*)

Clear all items from the folder. This is the equivalent of calling `.remove` with each key that exists in the folder.

copy(*name*, *other*, *newname=None*, *registry=None*)
Copy a subobject named `name` from this folder to the folder represented by `other`. If `newname` is not none, it is used as the target object name; otherwise the existing subobject name is used.

find_service(*service_name*)
Return a service named by `service_name` in this folder *or any parent service folder* or `None` if no such service exists. A shortcut for `substanced.service.find_service()`.

find_services(*service_name*)
Returns a sequence of service objects named by `service_name` in this folder's lineage or an empty sequence if no such service exists. A shortcut for `substanced.service.find_services()`

get(*name*, *default=None*)
Return the object named by `name` or the default.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding.

is_ordered()
Return true if the folder has a manually set ordering, false otherwise.

is_reorderable()
Return true if the folder can be reordered, false otherwise.

items()
Return an iterable sequence of (name, value) pairs in the folder.

Respect `order`, if set.

keys()
Return an iterable sequence of object names present in the folder.

Respect order, if set.

load(*name*, *newobject*, *registry=None*)
A replace method used by the code that loads an existing dump. Events sent during this replace will have a true `loading` flag.

move(*name*, *other*, *newname=None*, *registry=None*)
Move a subobject named `name` from this folder to the folder represented by `other`. If `newname` is not none, it is used as the target object name; otherwise the existing subobject name is used.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events as well as the Added and WillBeAdded events sent will indicate that the object is moving.

order
Return an iterable sequence of object names present in the folder.

Respect order, if set.

pop(*name*, *default=<object object>*, *registry=None*)
Remove the item stored in the under `name` and return it.

If `name` doesn't exist in the folder, and `default` **is not** passed, raise a `KeyError`.

If `name` doesn't exist in the folder, and `default` **is** passed, return `default`.

When the object stored under `name` is removed from this folder, remove its `__parent__` and `__name__` values.

When this method is called, emit an *`substanced.event.ObjectWillBeRemoved`* event before the object loses its `__name__` or `__parent__` values. Emit an *`substanced.event.ObjectRemoved`* after the object loses its `__name__` and `__parent__` value,

remove(*name*, *send_events=True*, *moving=None*, *loading=False*, *registry=None*)
Same thing as `__delitem__`.

If `send_events` is false, suppress the sending of folder events.

If `moving` is not `None`, the `moving` argument must be the folder to which the named object will be moving. This value will be passed along as the `moving` attribute of the events sent as the result of this action. If `loading` is `True`, the `loading` attribute of events sent as a result of calling this method will be `True` too.

rename(*oldname*, *newname*, *registry=None*)
Rename a subobject from oldname to newname.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events sent will indicate that the object is moving.

reorder(*names*, *before*)
Move one or more items from a folder into new positions inside that folder. `names` is a list of ids of existing folder subobject names, which will be inserted in order before the item named `before`. All other items are left in the original order. If `before` is `None`, the items will be appended after the last item in the current order. If this method is called on a folder which does not have an order set, or which is not reorderable, a `ValueError` will be raised. A `KeyError` is raised, if `before` does not correspond to any item, and is not `None`.

replace(*name*, *newobject*, *send_events=True*, *registry=None*)
Replace an existing object named `name` in this folder with a new object `newobject`. If there isn't an object named `name` in this folder, an exception will *not* be raised; instead, the new object will just be added.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events will be sent for the old object, and the WillBeAdded and Added events will be sent for the new object.

set_order(*names*, *reorderable=None*)
Sets the folder order. `names` is a list of names for existing folder items, in the desired order. All names that currently exist in the folder must be mentioned in `names`, or a `ValueError` will be raised.

If `reorderable` is passed, value, it must be `None`, `True` or `False`. If it is `None`, the reorderable flag will not be reset from its current value. If it is anything except `None`, it will be treated as a boolean and the reorderable flag will be set to that value. The `reorderable` value of a folder will be returned by that folder's *`is_reorderable()`* method. The *`is_reorderable()`* method is used by the SDI folder contents view to indicate that the folder can or cannot be reordered via the web UI.

If `reorderable` is set to `True`, the *`reorder()`* method will work properly, otherwise it will raise a `ValueError` when called.

unset_order()
Remove set order from a folder, making it unordered, and non-reorderable.

validate_name(*name*, *reserved_names=()*)
Validate the `name` passed to ensure that it's addable to the folder. Returns the name decoded to Unicode if it passes all addable checks. It's not addable if:

- the name is not decodeable to Unicode.
- the name starts with `@@` (conflicts with explicit view names).
- the name has slashes in it (WSGI limitation).
- the name is empty.

If any of these conditions are untrue, raise a `ValueError`. If the name passed is in the list of `reserved_names`, raise a `ValueError`.

values()

Return an iterable sequence of the values present in the folder.

Respect `order`, if set.

class `substanced.folder.`**SequentialAutoNamingFolder**(*data=None*, *family=None*, *autoname_length=None*, *autoname_start=None*)

An auto-naming folder which autonames a subobject by sequentially incrementing the maximum key of the folder.

Example names: `0000001`, then `0000002`, and so on.

This class implements the `substanced.interfaces.IAutoNamingFolder` interface and inherits from `substanced.folder.Folder`.

This class is typically used as a base class for a custom content type.

__init__(*data=None*, *family=None*, *autoname_length=None*, *autoname_start=None*)

Constructor. Data may be an initial dictionary mapping object name to object. Autoname length may be supplied. If it is not, it will default to 7. Autoname start may be supplied. If it is not, it will default to -1.

add_next(*subobject*, *send_events=True*, *duplicating=None*, *moving=None*, *registry=None*)

Add a subobject, naming it automatically, giving it the name returned by this folder's `next_name` method. It has the same effect as calling `substanced.folder.Folder.add()`, but you needn't provide a name argument.

This method returns the name of the subobject.

next_name(*subobject*)

Return a name string based on:

- intifying the maximum key of this folder and adding one.
- zero-filling the left hand side of the result with as many zeroes as are in the value of this folder's `autoname_length` constructor value.

If the folder has no items in it, the initial value used as a name will be the value of this folder's `autoname_start` constructor value.

add(*name*, *other*, *send_events=True*, *reserved_names=()*, *duplicating=None*, *moving=None*, *loading=False*, *registry=None*)

The `add` method of a SequentialAutoNamingFolder will raise a `ValueError` if the `name` it is passed is not intifiable, as its `next_name` method relies on controlling the types of names that are added to it (they must be intifiable). It will also zero-fill the name passed based on this folder's `autoname_length` constructor value. It otherwise just calls its superclass' `add` method and returns the result.

class `substanced.folder.`**RandomAutoNamingFolder**(*data=None*, *family=None*, *autoname_length=None*)

An auto-naming folder which autonames a subobject using a random string.

Example names: `MXF937A`, `FLTP2F9`.

This class implements the `substanced.interfaces.IAutoNamingFolder` interface and inherits from `substanced.folder.Folder`.

This class is typically used as a base class for a custom content type.

__init__(*data=None*, *family=None*, *autoname_length=None*)

Constructor. Data may be an initial dictionary mapping object name to object. Autoname length may be supplied. If it is not, it will default to 7.

add_next(*subobject*, *send_events=True*, *duplicating=None*, *moving=None*, *registry=None*)

Add a subobject, naming it automatically, giving it the name returned by this folder's `next_name` method.

It has the same effect as calling `substanced.folder.Folder.add()`, but you needn't provide a name argument.

This method returns the name of the subobject.

`next_name`(*subobject*)
Return a name string based on generating a random string composed of digits and uppercase letters of a length determined by this folder's `autoname_length` constructor value. It tries generatoing values continuously until one that is unused is found.

5.15 `substanced.folder.views` API

5.16 `substanced.form` API

class `substanced.form.`**`Form`**(*schema*, *action=''*, *method='POST'*, *buttons=()*, *formid='deform'*, *use_ajax=False*, *ajax_options='{}'*, *autocomplete=None*, ***kw*)
Subclass of `deform.form.Form` which uses a custom resource registry designed for Substance D. XXX point at deform docs.

class `substanced.form.`**`FormView`**(*context*, *request*)
A class which can be used as a view which introspects a schema to present the form. XXX describe better using `pyramid_deform` documentation.

`form_class`
alias of `Form`

class `substanced.form.`**`FileUploadTempStore`**(*request*)
A Deform `FileUploadTempStore` implementation that stores file upload data in the Pyramid session and on disk. The request passed to its constructor must be a fully-initialized Pyramid request (it have a `registry` attribute, which must have a `settings` attribute, which must be a dictionary). The `substanced.uploads_tempdir` variable in the `settings` dictionary must be set to the path of an existing directory on disk. This directory will temporarily store file upload data on behalf of Deform and Substance D when a form containing a file upload widget fails validation.

See the *Deform* documentation for more information about `FileUploadTempStore` objects.

5.17 `substanced.locking` API

Advisory exclusive DAV-style locks for content objects.

When a resource is locked, it is presumed that its SDI UI will display a warning to users who do not hold the lock. The locking service can also be used by add-ons such as DAV implementations.

class `substanced.locking.`**`Lock`**(*infinite=False*, *timeout=3600*, *comment=None*, *last_refresh=None*)
A persistent object representing a lock.

`ownerid`
The owner oid for this lock.

`owner`
The owner object of this lock (a User).

`resourceid`
The oid of the resource related to this lock.

resource
: The resource object related to this lock.

commit_suicide()
: Remove this lock from the lock service.

expires()
: Return the future datetime at which this lock will expire.

  For invalid locks, the returned value indicates the point in the past at which the lock expired.

is_valid(*when=None*)
: Return True if the lock has not expired and its resource exists.

refresh(*timeout=None*, *when=None*)
: Refresh the lock.

  If the timeout is not None, set the timeout for this lock too.

class `substanced.locking.`**LockError**(*lock*)
: Raised when a lock cannot be created due to a conflicting lock.

  Instances of this class have a `lock` attribute which is a *`substanced.locking.Lock`* object, representing the conflicting lock.

class `substanced.locking.`**UnlockError**(*lock*)
: Raised when a lock cannot be removed

  This may be because the owner suplied in the unlock request does not match the owner of the lock, or becaues the lock no longer exists.

  Instances of this class have a `lock` attribute which is a *`substanced.locking.Lock`* object, representing the conflicting lock, or `None` if there was no lock to unlock.

`substanced.locking.`**lock_resource**(*resource*, *owner_or_ownerid*, *timeout=None*, *comment=None*, *locktype=<ReferenceClass substanced.interfaces.WriteLock>*, *infinite=False*)
: Lock a resource using the lock service.

  If the resource is already locked by the owner supplied as owner_or_ownerid, refresh the lock using `timeout`.

  If the resource is not already locked by another user, create a new lock with the given values.

  If the resource is already locked by a different user, raise a *`substanced.locking.LockError`*

  If a Lock Service does not already exist in the lineage, a `ValueError` will be raised.

  > **Warning:** Callers should assert that the owner has the `sdi.lock` permission against the resource before calling this function to ensure that a user can't lock a resource he is not permitted to.

`substanced.locking.`**unlock_resource**(*resource*, *owner_or_ownerid*, *locktype=<ReferenceClass substanced.interfaces.WriteLock>*)
: Unlock a resource using the lock service.

  If the resource is already locked by a user other than the owner supplied as `owner_or_ownerid` or the resource isn't already locked with this lock type, raise a *`substanced.locking.UnlockError`* exception.

  Otherwise, remove the lock.

  If a Lock Service does not already exist in the lineage, a `ValueError` will be raised.

Warning: Callers should assert that the owner has the `sdi.lock` permission against the resource before calling this function to ensure that a user can't lock a resource he is not permitted to.

`substanced.locking.`**`discover_resource_locks`**(*resource*, *include_invalid=False*, *include_lineage=True*, *locktype=<ReferenceClass substanced.interfaces.WriteLock>*)

Return locks related to `resource` for the given `locktype`.

Return a sequence of *`substanced.locking.Lock`* objects.

By default, only valid locks are returned.

Invalid locks for the resource may exist, but they are not returned unless `include_invalid` is `True`.

Under normal circumstances, the length of the sequence returned will be either 0 (if there are no locks) or 1 (if there is any lock).

In some special circumstances, however, when the *`substanced.locking.lock_resource`* API is not used to create locks, there may be more than one lock of the same type related to a resource.

5.18 `substanced.objectmap` API

`class` `substanced.objectmap.`**`ObjectMap`**(*root*, *family=None*)

`add`(*obj*, *path_tuple*, *duplicating=False*, *moving=False*)

Add a new object to the object map at the location specified by `path_tuple` (must be the path of the object in the object graph as a tuple, as returned by Pyramid's `resource_path_tuple` function).

If `duplicating` is `True`, replace the oid of the added object even if it already has one and adjust extents involving the new oid.

If `moving` is `True`, don't add any extents.

It is an error to pass a true value for both `duplicating` and `moving`.

`allowed`(*oids*, *principals*, *permission*)

For the set of oids present in `oids`, return a sequence of oids that are permitted `permission` against each oid if the implied user is a member of the set of principals implied by `principals`. This method uses the data collected via the `set_acl` method of this class.

`connect`(*source*, *target*, *reftype*)

Connect a source object or objectid to a target object or objectid using reference type `reftype`

`disconnect`(*source*, *target*, *reftype*)

Disconnect a source object or objectid from a target object or objectid using reference type `reftype`

`get_extent`(*name*, *default=()*)

Return the extent for `name` (typically a factory name, e.g. the dotted name of the content class). It will be a TreeSet composed entirely of oids. If no extent exist by this name, this will return the value of `default`.

`get_reftypes`()

Return a sequence of reference types known by this objectmap.

`has_references`(*obj*, *reftype=None*)

Return true if the object participates in any reference as a source or a target. `obj` may be an object or an oid.

`new_objectid`()
Obtain an unused integer object identifier

`object_for`(*objectid_or_path_tuple*, *context=None*)
Returns an object or `None` given an object id or a path tuple

`objectid_for`(*obj_or_path_tuple*)
Returns an objectid or `None`, given an object or a path tuple

`order_sources`(*targetid*, *reftype*, *order=<object object>*)
Set the ordering of the source ids of a reference relative to the `targetid`. `order` should be a tuple or list of oids or objects in the order that they should be kept in the reference map. If the reftyp+targetid combination has existing reference values, the values in `order` must mention all of their oids, or a `ValueError` will be raised. You can unset an order for this targetid+reftype combination by passing `None` as the order.

`order_targets`(*sourceid*, *reftype*, *order=<object object>*)
Set the ordering of the target ids of a reference type. `order` should be a tuple (or list) of oids or objects in the order that they should be kept in the reference map. If the reference type has existing reference values, the values in `order` must mention all of their oids, or a `ValueError` will be raised. You can unset an ordering by passing `None` as the order.

`path_for`(*objectid*)
Returns an path or `None` given an object id

`pathcount`(*obj_or_path_tuple*, *depth=None*, *include_origin=True*)
Return the total number of objectids under a given path given an object or a path tuple. If `depth` is None, count all object ids under the path. If `depth` is an integer, use that depth instead. If `include_origin` is `True`, count the object identifier of the object that was passed, otherwise omit it.

`pathlookup`(*obj_or_path_tuple*, *depth=None*, *include_origin=True*)
Return a set of objectids under a given path given an object or a path tuple. If `depth` is None, return all object ids under the path. If `depth` is an integer, use that depth instead. If `include_origin` is `True`, include the object identifier of the object that was passed, otherwise omit it from the returned set.

`remove`(*obj_objectid_or_path_tuple*, *moving=False*)
Remove an object from the object map give an object, an object id or a path tuple. If `moving` is `False`, also remove any references added via `connect` and any extents related to the removed objects.

Return a set of removed oids (including the oid related to the object passed).

`set_acl`(*obj_objectid_or_path_tuple*, *acl*)
For the resource implied by `obj_objectid_or_path_tuple`, set the cached version of its ACL (for later used by `allowed`) to the ACL passed as `acl`

`sourceids`(*obj*, *reftype*)
Return a set of object identifiers of the objects connected to `obj` a source using reference type `reftype`

`sources`(*obj*, *reftype*)
Return a generator which will return the objects connected to `obj` as a source using reference type `reftype`

`targetids`(*obj*, *reftype*)
Return a set of object identifiers of the objects connected to `obj` a target using reference type `reftype`

`targets`(*obj*, *reftype*)
Return a generator which will return the objects connected to `obj` as a target using reference type `reftype`

class `substanced.objectmap.`**`Multireference`**(*context*, *objectmap*, *reftype*, *ignore_missing*, *resolve*, *orientation*, *ordered=False*)
An iterable of objects (if `resolve` is true) or oids (if `resolve` is false). Also supports the Python sequence protocol.

Additionally supports `connect`, `disconnect`, and `clear` methods for mutating the relationships implied by the reference.

`clear`()
Clear all references in this relationship.

`connect`(*objects*, *ignore_missing=None*)
Connect `objects` to this reference's relationship. `objects` should be a sequence of content objects or object identifiers.

`disconnect`(*objects*, *ignore_missing=None*)
Disconnect `objects` from this reference's relationship. `objects` should be a sequence of content objects or object identifiers.

`substanced.objectmap.`**`reference_sourceid_property`**(*reftype*)
Returns a property which, when set, establishes an *object map reference* between the property's instance (the source) and another object in the objectmap (the target) based on the reference type `reftype`. It is comparable to a Python 'weakref' between the persistent object instance which the property is attached to and the persistent target object id; when the target object or the object upon which the property is defined is removed from the system, the reference is destroyed.

The `reftype` argument is a *reference type*, a hashable object that describes the type of the relation. See `substanced.objectmap.ObjectMap.connect()` for more information about reference types.

You can set, get, and delete the value. When you set the value, a relation is formed between the object which houses the property and the target object id. When you get the value, the related value (or `None` if no relation exists) is returned, when you delete the value, the relation is destroyed and the value will revert to `None`.

For example:

```python
# definition

from substanced.content import content
from substanced.objectmap import reference_sourceid_property

@content('Profile')
class Profile(Persistent):
    user_id = reference_sourceid_property('profile-to-userid')

# subsequent usage of the property in a view...

profile = registry.content.create('Profile')
somefolder['profile'] = profile
profile.user_id = get_oid(request.user)
print profile.user_id # will print the oid of the user

# if the user is later deleted by unrelated code...

print profile.user_id # will print None

# or if you delete the value explicitly...

del profile.user_id
print profile.user_id # will print None
```

`substanced.objectmap.`**`reference_source_property`**(*reftype*)
Exactly like `substanced.objectmap.reference_sourceid_property()`, except its getter returns the *instance* related to the target instead of the target object id. Likewise, its setter will accept another persistent object instance that has an object id.

For example:

```python
# definition

from substanced.content import content
from substanced.objectmap import reference_source_property

@content('Profile')
class Profile(Persistent):
    user = reference_source_property('profile-to-user')

# subsequent usage of the property in a view...

profile = registry.content.create('Profile')
somefolder['profile'] = profile
profile.user = request.user
print profile.user # will print the user object

# if the user is later deleted by unrelated code...

print profile.user # will print None

# or if you delete the value explicitly...

del profile.user
print profile.user # will print None
```

`substanced.objectmap.`**`reference_targetid_property`**(*reftype*)

Same as *`substanced.objectmap.reference_sourceid_property()`*, except the object upon which the property is defined is the *target* of the reference and any object assigned to the property is the source.

`substanced.objectmap.`**`reference_target_property`**(*reftype*)

Same as *`substanced.objectmap.reference_source_property()`*, except the object upon which the property is defined is the *target* of the reference and any object assigned to the property is the source.

`substanced.objectmap.`**`multireference_sourceid_property`**(*reftype*, *ignore_missing=False*, *ordered=None*)

Like *`substanced.objectmap.reference_sourceid_property()`*, but maintains a *`substanced.objectmap.Multireference`* rather than an object id. If `ignore_missing` is `True`, attempts to connect or disconnect unresolveable object identifiers will not cause an exception. If `ordered` is `True`, the relative ordering of references in a sequence will be maintained when you assign that sequence to the property and when you use the `.connect` method of the property. If `ordered` is `None`, defers to the appropriate attribute on the `reftype`.

`substanced.objectmap.`**`multireference_source_property`**(*reftype*, *ignore_missing=False*, *ordered=None*)

Like *`substanced.objectmap.reference_source_property()`*, but maintains a *`substanced.objectmap.Multireference`* rather than a single object reference. If `ignore_missing` is `True`, attempts to connect or disconnect unresolveable object identifiers will not cause an exception. If `ordered` is `True`, the relative ordering of references in a sequence will be maintained when you assign that sequence to the property and when you use the `.connect` method of the property. If `ordered` is `None`, defers to the appropriate attribute on the `reftype`.

`substanced.objectmap.`**`multireference_targetid_property`**(*reftype*, *ignore_missing=False*, *ordered=None*)

Like *`substanced.objectmap.reference_targetid_property()`*, but maintains a *`substanced.objectmap.Multireference`* rather than an object id. If `ignore_missing` is

`True`, attempts to connect or disconnect unresolveable object identifiers will not cause an exception. If `ordered` is `True`, the relative ordering of references in a sequence will be maintained when you assign that sequence to the property and when you use the `.connect` method of the property. If `ordered` is `None`, defers to the appropriate attribute on the `reftype`.

`substanced.objectmap.`**`multireference_target_property`**(*reftype*, *ignore_missing=False*, *ordered=None*)

Like *`substanced.objectmap.reference_target_property()`*, but maintains a *`substanced.objectmap.Multireference`* rather than a single object reference. If `ignore_missing` is `True`, attempts to connect or disconnect unresolveable object identifiers will not cause an exception. If `ordered` is `True`, the relative ordering of references in a sequence will be maintained when you assign that sequence to the property and when you use the `.connect` method of the property. If `ordered` is `None`, defers to the appropriate attribute on the `reftype`.

`class` `substanced.objectmap.`**`ReferentialIntegrityError`**(*obj*, *reftype*, *oids*)

Exception raised when a referential integrity constraint is violated. Raised before an object involved in a relation with an integrity constraint is removed from a folder.

Attributes:

```
obj: the object which would have been removed were its removal not
     prevented by the raising of this exception

reftype: the reference type (usually a class)

oids: the oids that reference the to-be-removed object.
```

`get_objects`()

Return the objects which hold a reference to the object inovlved in the integrity error.

`class` `substanced.objectmap.`**`SourceIntegrityError`**(*obj*, *reftype*, *oids*)

`class` `substanced.objectmap.`**`TargetIntegrityError`**(*obj*, *reftype*, *oids*)

5.19 `substanced.principal` API

5.20 `substanced.property` API

`class` `substanced.property.`**`PropertySheet`**(*context*, *request*)

Bases: `object`

Convenience base class for concrete property sheet implementations

`before_render`(*form*)

Hook: allow subclasses to scribble on form.

Called by `substanced.property.views.PropertySheetsView.before`, after building the form but before rendering it.

`substanced.property.`**`add_propertysheet`**(*self*, **arg*, ***kw*)

Add a propertysheet for the content types implied by `iface` and `predicates`.

The `propsheet` argument represents a propertysheet class (or a dotted Python name which identifies such a class); it will be called with two objects: `context` and `request` whenever Substance D determines that the propertysheet is necessary to display. The `iface` may be an interface or a class or a dotted Python name to a global object representing an interface or a class.

Using the default `iface` value, `None` will cause the propertysheet to be registered for all content types.

Any number of predicate keyword arguments may be passed in `**predicates`. Each predicate named will narrow the set of circumstances in which the propertysheet will be invoked. Each named predicate must have been registered via `pyramid.config.Configurator.add_propertysheet_predicate()` before it can be used.

`substanced.property.`**`add_propertysheet_predicate`**(*self*, **arg*, ***kw*)

Adds a property sheet predicate factory. The associated property sheet predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_propertysheet()` in the `**predicates` anonymous keyword argument dictionary.

`name` should be the name of the predicate. It must be a valid Python identifier (it will be used as a `**predicates` keyword argument to `add_propertysheet()`).

`factory` should be a predicate factory or dotted Python name which refers to a predicate factory.

class `substanced.property.`**`PropertySheet`**(*context*, *request*)

Convenience base class for concrete property sheet implementations

`before_render`(*form*)

Hook: allow subclasses to scribble on form.

Called by `substanced.property.views.PropertySheetsView.before`, after building the form but before rendering it.

5.21 `substanced.schema` API

class `substanced.schema.`**`Schema`**(**arg*, ***kw*)

A `colander.Schema` subclass which generates and validates a CSRF token automatically. You must use it like so:

```python
from substanced.schema import Schema as CSRFSchema
import colander

class MySchema(CSRFSchema):
    my_value = colander.SchemaNode(colander.String())
```

And in your application code, *bind* the schema, passing the request as a keyword argument:

```python
def aview(request):
    schema = MySchema().bind(request=request)
```

In order for the CRSFSchema to work, you must configure a *session factory* in your Pyramid application. This is usually done by Substance D itself, but may not be done for you in extremely custom configurations.

`schema_type`

alias of `RemoveCSRFMapping`

class `substanced.schema.`**`NameSchemaNode`**(**arg*, ***kw*)

Convenience Colander schemanode used to represent the name (aka `__name__`) of an object in a propertysheet or add form which allows for customizing the detection of whether editing or adding is being done, and setting a max length for the name.

By default it uses the context's `check_name` API to ensure that the name provided is valid, and limits filename length to a default of 100 characters. Some usage examples follow.

This sets up the name_node to assume that it's in 'add' mode with the default 100 character max limit.:

```
name_node = NameSchemaNode()
```

This sets up the name_node to assume that it's in 'add' mode, and that the maximum length of the name provided is 20 characters:

```
name_node = NameSchemaNode(max_len=20)
```

This sets up the name_node to assume that it's in 'edit' mode (`check_name` will be called on the **parent** of the bind context, not on the context itself):

```
name_node = NameSchemaNode(editing=True)
```

This sets up the name_node to condition whether it's in edit mode on the result of a function:

```
def i_am_editing(context, request):
    return request.registry.content.istype(context, 'Document')

name_node = NameSchemaNode(editing=i_am_editing)
```

class `substanced.schema.`**`PermissionsSchemaNode`**(**arg*, ***kw*)

A SchemaNode which represents a set of permissions; uses a widget which collects all permissions from the introspection system. Deserializes to a set.

5.22 `substanced.sdi` API

`substanced.schema.`**`LEFT`**

`substanced.schema.`**`MIDDLE`**

`substanced.schema.`**`RIGHT`**

5.23 `substanced.root` API

class `substanced.root.`**`Root`**(*data=None*, *family=None*)

An object representing the root of a Substance D application (the object represented in the root of the SDI). It is a subclass of *`substanced.folder.Folder`*.

When created as the result of `registry.content.create`, an instance of a Root will contain a `principals` service. The principals service will contain a user whose name is specified via the `substanced.initial_login` deployment setting with a password taken from the `substanced.initial_password` setting. This user will also be a member of an `admins` group. The `admins` group will be granted the `ALL_PERMISSIONS` special permission in the root.

If this class is created by hand, its `after_create` method must be called manually to create its objectmap, the services, the user, and the group.

5.24 `substanced.stats` API

`substanced.stats.`**`statsd_timer`**()

Return a context manager that can be used for statsd timing, e.g.:

```
with statsd_timer('addlotsofstuff'):
    # add lots of stuff
```

name is the statsd stat name, rate is the sample rate (a float between 0 and 1), and registry can be passed to speed up lookups (it should be the Pyramid registry).

substanced.stats.**statsd_gauge**()
Register a statsd gauge value. For example:

```
statsd_gauge('connections', numconnections)
```

name is the statsd stat name, rate is the sample rate (a float between 0 and 1), and registry can be passed to speed up lookups (it should be the Pyramid registry).

substanced.stats.**statsd_incr**()
Incremement or decrement a statsd counter value. For example:

```
    statsd_incr('hits', 1)

To decrement::

    statsd_incr('numusers', -1)
```

name is the statsd stat name, rate is the sample rate (a float between 0 and 1), and registry can be passed to speed up lookups (it should be the Pyramid registry).

5.25 substanced.util API

substanced.util.**acquire**(*resource*, *name*, *default=<object object>*)

substanced.util.**get_oid**(*resource*, *default=<object object>*)
Return the object identifer of resource. If resource has no object identifier, raise an AttributeError exception unless default was passed a value; if default was passed a value, return the default in that case.

substanced.util.**set_oid**(*resource*, *oid*)
Set the object id of the resource to oid.

substanced.util.**get_acl**(*resource*, *default=<object object>*)
Return the ACL of the object or the default if the object has no ACL. If no default is passed, an AttributeError will be raised if the object doesn't have an ACL.

substanced.util.**set_acl**(*resource*, *new_acl*, *registry=None*)
Change the ACL on resource to new_acl, which may be a valid ACL or None. If new_acl is None, any existing non-None __acl__ attribute of the resource will be removed (via del resource.__acl__). Otherwise, if the resource's __acl__ and the new_acl differ, set the resource's __acl__ to new_acl via setattr.

If the new ACL and the object's original ACL differ, send a *substanced.event.ACLModified* event with the new ACL and the original ACL (the __acl__ attribute of the resource, or None if it doesn't have one) as arguments to the event.

This function will return True if a mutation to the resource's __acl__ was performed, and False otherwise.

If registry is passed, it should be a Pyramid registry; if it is not passed, this function will use the current threadlocal registry to send the event.

substanced.util.**get_interfaces**(*obj*, *classes=True*)
Return the set of interfaces provided by obj. Include its __class__ if classes is True.

substanced.util.**get_content_type**(*resource*, *registry=None*)
: Return the content type of a resource or `None` if the object has no content type. If `registry` is not supplied, the current Pyramid registry will be looked up as a thread local in order to find the Substance D content registry.

substanced.util.**find_content**(*resource*, *content_type*, *registry=None*)
: Return the first object in the lineage of the resource that supplies the `content_type`. If `registry` is not supplied, the current Pyramid registry will be looked up as a thread local in order to find the Substance D content registry.

substanced.util.**find_service**(*resource*, *name*, **subnames*)
: Find the first service named `name` in the lineage of `resource` or return `None` if no such-named service could be found.

  If `subnames` is supplied, when a service named `name` is found in the lineage, it will attempt to traverse the service as a folder, finding a content object inside the service, and it will return it instead of the service object itself. For example, `find_service(resource, 'principals', 'users')` would find and return the `users` subobject in the `principals` service. `find_service(resource, 'principals', 'users', 'fred')` would find and return the fred subobject of the users subobject of the principals service, and so forth. If `subnames` are supplied, and the named object cannot be found, the lineage search continues.

substanced.util.**find_services**(*resource*, *name*, **subnames*)
: Finds all services named `name` in the lineage of `resource` and returns a sequence containing those service objects. The sequence will begin with the most deepest nested service and will end with the least deeply nested service. Returns an empty sequence if no such-named service could be found.

  If `subnames` is supplied, when a service named `name` is found in the lineage, it will attempt to traverse the service as a folder, finding a content object inside the service, and this API will append this object rather than the service itself to the list of things returned. For example, `find_services(resource, 'principals', 'users')` would find the `users` subobject in the `principals` service. `find_services(resource, 'principals', 'users', 'fred')` would find the fred subobject of the users subobject of the principals service, and so forth. If `subnames` are supplied, whether or not the named object can be found, the lineage search continues.

substanced.util.**find_objectmap**(*context*)
: Returns the object map for the root object in the lineage of the `context` or `None` if no objectmap can be found.

substanced.util.**find_catalogs**(*resource*, *name=None*)
: Return all catalogs in the lineage. If `name` is supplied, return only catalogs that have this name in the lineage, otherwise return all catalogs in the lineage.

substanced.util.**find_catalog**(*resource*, *name*)
: Return the first catalog named `name` in the lineage of the resource

substanced.util.**find_index**(*resource*, *catalog_name*, *index_name*)
: Find the first catalog named `catalog_name` in the lineage of the resource, and ask it for its `index_name` index; return the resulting index. If either a catalog of the provided name or an index of the provided name does not exist, this function will return `None`.

substanced.util.**get_principal_repr**(*principal_or_id*)
: Given as `principal_or_id` a resource object that has a `__principal_repr__` method, return the result of calling that method (without arguments); it must be a string that uniquely identifies the principal amongst all principals in the system.

  Given as `principal_or_id` a resource object that does **not** have a `__principal_repr__` method, return the result of the stringification of the `__oid__` attribute of the resource object.

  Given an integer as `principal_or_id`, return a stringification of the integer.

  Given any other string value, return it.

`substanced.util.`**`is_folder`**`(`*resource*`)`
Return `True` if the object is a folder, `False` if not.

`substanced.util.`**`is_service`**`(`*resource*`)`
Returns `True` if the resource is a service, `False` if not.

`substanced.util.`**`get_factory_type`**`(`*resource*`)`
If the resource has a __factory_type__ attribute, return it. Otherwise return the full Python dotted name of the resource's class.

`substanced.util.`**`coarse_datetime_repr`**`(`*date*`)`
Convert a datetime to an integer with 100 second granularity.

The granularity reduces the number of index entries in a fieldindex when it's used in an indexview to convert a datetime value to an integer.

`substanced.util.`**`postorder`**`(`*startnode*`)`
Walks over nodes in a folder recursively. Yields deepest nodes first.

`substanced.util.`**`merge_url_qs`**`(`*url*, ***kw*`)`
Merge the query string elements of a URL with the ones in `kw`. If any query string element exists in `url` that also exists in `kw`, replace it.

`substanced.util.`**`chunks`**`(`*stream*, *chunk_size=10000*`)`
Return a generator that will iterate over a stream (a filelike object) `chunk_size` bytes at a time.

`substanced.util.`**`renamer`**`()`
Returns a property. The getter of the property returns the `__name__` attribute of the instance on which it's defined. The setter of the property calls `rename()` on the `__parent__` of the instance on which it's defined if the new value doesn't match the existing `__name__` of the instance (this will cause `__name__` to be reset if the parent is a normal Substance D folder). Sample usage:

```
class SomeContentType(Persistent):
    name = renamer()
```

`substanced.util.`**`get_dotted_name`**`(`*g*`)`
Return the dotted name of a global object.

`substanced.util.`**`get_icon_name`**`(`*resource*, *request*`)`
Returns the content registry icon name of the resource or `None` if the resource type has no icon in the content registry.

`substanced.util.`**`get_auditlog`**`(`*context*`)`
Returns the current *`substanced.audit.AuditLog`* object or `None` if no audit database is configured

class `substanced.util.`**`Batch`**`(`*seq*, *request*, *url=None*, *default_size=10*, *toggle_size=40*, *seqlen=None*`)`
Given a sequence named `seq`, and a Pyramid request, return an object with the following attributes:

`items`

A list representing a slice of `seq`. It will contain the number of elements in `request.params['batch_size']` or the `default_size` number if such a key does not exist in request.params or the key is invalid. The slice will begin at `request.params['batch_num']` or zero if such a key does not exist in `request.params` or the `batch_num` key could not successfully be converted to a positive integer.

This value can be iterated over via the `__iter__` of the batch object.

`size`

The value obtained from `request.params['batch_size']` or `default_size` if no `batch_size` parameter exists in `request.params` or the `batch_size` parameter could not successfully be converted to a positive interger.

`num`

The value obtained from `request.params['batch_num']` or `0` if no `batch_num` parameter exists in `request.params` or if the `batch_num` parameter could not successfully be converted to a positive integer. Batch numbers are indexed from zero, so batch `0` is the first batch, batch `1` the second, and so forth.

`length`

This is length of the current batch. It is usually equal to `size` but may be different in the very last batch. For example, if the `seq` is `[1,2,3,4]` and the batch size is `3`, the first batch's `length` will be `3` because the batch content will be `[1,2,3]`; but the second and final batch's `length` will be `1` because the batch content will be `[4]`.

`last`

The batch number computed from the sequence length of the last batch (indexed from zero).

`first_url`

The URL of the first batch. This will be a URL with `batch_num` and `batch_size` in its query string. The base URL will be taken from the `url` value passed to this function. If a `url` value is not passed to this function, the URL will be taken from `request.url`. This value will be `None` if the current `batch_num` is 0.

`prev_url`

The URL of the previous batch. This will be a URL with `batch_num` and `batch_size` in its query string. The base URL will be taken from the `url` value passed to this function. If a `url` value is not passed to this function, the URL will be taken from `request.url`. This value will be `None` if there is no previous batch.

`next_url`

The URL of the next batch. This will be a URL with `batch_num` and `batch_size` in its query string. The base URL will be taken from the `url` value passed to this function. If a `url` value is not passed to this function, the URL will be taken from `request.url`. This value will be `None` if there is no next batch.

`last_url`

The URL of the next batch. This will be a URL with `batch_num` and `batch_size` in its query string. The base URL will be taken from the `url` value passed to this function. If a `url` value is not passed to this function, the URL will be taken from `request.url`. This value will be `None` if there is no next batch.

`required`

`True` if either `next_url` or `prev_url` are `True` (meaning batching is required).

`multicolumn`

`True` if the current view should be rendered in multiple columns.

`toggle_url`

The URL to be used for the multicolumn/single column toggle button. The `batch_size`, `batch_num`, and `multicolumn` parameters are converted to their multicolumn or single column equivalents. If a user is viewing items 40-80 in multiple columns, the toggle will switch to items

40-50 in a single column. If a user is viewing items 50-60 in a single column, the toggle will switch to items 40-80 in multiple columns.

`toggle_text`

The text to display on the multi-column/single column toggle.

`make_columns`

A method to split `items` into a nested list representing columns.

`seqlen`

This is total length of the sequence (across all batches).

`startitem`

The item number that starts this batch (indexed from zero).

`enditem`

The item number that ends this batch (indexed from zero).

`make_columns`(*column_size=10*, *num_columns=4*)
Break `self.items` into a nested list representing columns.

5.26 `substanced.workflow` API

`class` `substanced.workflow.`**`ACLState`**(*acl=None*, ***kw*)
Bases: `dict`

`class` `substanced.workflow.`**`ACLWorkflow`**(*initial_state*, *type*, *name=''*, *description=''*)
Bases: `substanced.workflow.Workflow`

`class` `substanced.workflow.`**`Workflow`**(*initial_state*, *type*, *name=''*, *description=''*)
Bases: `object`

Finite state machine.

Implements *substanced.interfaces.IWorkflow*.

Parameters

- **`initial_state`** (*`string`*) – Initial state of the workflow assigned to the content
- **`type`** (*`string`*) – Identifier to separate multiple workflows on same content.
- **`name`** (*`string`*) – Display name.
- **`description`** (*`string`*) – Not used internally, provided as help text to describe what workflow does.

`add_state`(*state_name*, *callback=None*, ***kw*)
Add a new workflow state.

Parameters

- **`state_name`** – Unique name of the state for this workflow.
- **`callback`** (*`callable`*) – Will be called when content enters this state. Meaning `Workflow.reset()`, `Workflow.initialize()`, `Workflow.transition()` and `Workflow.transition_to_state()` will trigger callback if entering this state.
- **`**kw`** – Metadata assigned to this state.

Raises *`WorkflowError`* if state already exists.

Callback is called with **content** as a single positional argument and the keyword arguments **workflow**, **transition**, and **request**. Be aware that methods as *`Workflow.initialize()`* pass **transition** as an empty dictionary.

Note: `**kw` must not contain the key `callback`. This name is reserved for internal use.

`add_transition`(*transition_name*, *from_state*, *to_state*, *callback=None*, *permission=None*, ***kw*)
Add a new workflow transition.

Parameters

- **`transition_name`** – Unique name of transition for this workflow.
- **`callback`** (*`callable`*) – Will be called when transition is executed. Meaning *`Workflow.transition()`* and *`Workflow.transition_to_state()`* will trigger callback if this transition is executed.
- **`**kw`** – Metadata assigned to this transition.

Raises *`WorkflowError`* if transition already exists.

Raises *`WorkflowError`* if from_state or to_state don't exist.

Callback is called with **content** as a single positional argument and the keyword arguments **workflow**, **transition**, and **request**.

Note: `**kw` must not contain any of the keys `from_state`, `name`, `to_state`, or `callback`; these are reserved for internal use.

`check`()
Check the consistency of the workflow state machine.

Raises *`WorkflowError`* if workflow is inconsistent.

`get_states`(*content*, *request*, *from_state=None*)
Return all states for the workflow.

Parameters

- **`content`** – Object to be operated on
- **`request`** – *pyramid.request.Request* instance
- **`from_state`** – State of the content. If None, *`Workflow.state_of()`* will be used on **content**.

Return type list of dicts

Returns Where dictionary contains information about the transition, such as **title**, **initial**, **current**, **transitions** and **data**. **transitions** is return value of *`Workflow.get_transitions()`* call for current state. **data** is a dictionary containing at least **callback**.

Note: States that fail *has_permission* check for their transition are left out.

`get_transitions`(*content*, *request*, *from_state=None*)
Get all transitions from the content state.

Parameters

- **`content`** – Object to be operated on.
- **`request`** – *pyramid.request.Request* instance
- **`from_state`** – Name of the state to retrieve transitions. If None, `Workflow.state_of()` will be used on **content**.

Return type list of dicts

Returns Where dictionary contains information about the transition, such as **from_state**, **to_state**, **callback**, **permission** and **name**.

Note: Transitions that fail *has_permission* check are left out.

`has_state`(*content*)
Return True if the content has state for this workflow, False if not.

`initialize`(*content*, *request=None*)
Initialize the content object to the initial state of this workflow.

Parameters

- **`content`** – Object to be operated on
- **`request`** – *pyramid.request.Request* instance

Returns (initial_state, msg)

msg is a string returned by the state *callback*.

`reset`(*content*, *request=None*)
Reset the content workflow by calling the callback of it's current state and setting its state attr.

If content has no current state, it will be initialized for this workflow (see initialize).

msg is a string returned by the state callback.

Parameters

- **`content`** – Object to be operated on
- **`request`** – *pyramid.request.Request* instance

Returns (state, msg)

`state_of`(*content*)
Return the current state of the content object or None if the content object does not have this workflow.

`transition`(*content*, *request*, *transition_name*)
Execute a transition using a **transition_name** on **content**.

Parameters

- **`content`** – Object to be operated on.
- **`request`** – *pyramid.request.Request* instance
- **`transition_name`** – Name of transition to execute.

Raises `WorkflowError` if no transition is found

Raises `WorkflowError` if transition doesn't pass *has_permission* check

transition_to_state (*content*, *request*, *to_state*, *skip_same=True*)

Execute a transition to another state using a state name (**to_state**). All possible transitions towards **to_state** will be tried until one if found that passes without exception.

Parameters

- **content** – Object to be operated on.
- **request** – *pyramid.request.Request* instance
- **to_state** – State to transition to.
- **skip_same** – If True and the **to_state** is the same as the content state, no transition is issued.

Raises *WorkflowError* if no transition is found

exception substanced.workflow.**WorkflowError**

Bases: exceptions.Exception

Exception raised for anything related to *substanced.workflow*.

substanced.workflow.**add_workflow** (*config*, *workflow*, *content_types=(None,)*)

Configurator method for adding a workflow.

If no **content_types** is given, workflow is registered globally.

Parameters

- **config** – Pyramid configurator
- **workflow** – *Workflow* instance
- **content_types** (*iterable*) – Register workflow for given content_types

Raises ConfigurationError if *Workflow.check()* fails

Raises ConfigurationError if **content_type** does not exist

Raises DoesNotImplement if **workflow** does not implement IWorkflow

substanced.workflow.**get_workflow** (*request*, *type*, *content_type=None*)

Return a workflow based on a content_type and the workflow type.

Parameters

- **request** – *pyramid.request.Request* instance
- **type** – Workflow type
- **content_type** – Substanced content type or None for default workflow.

5.27 substanced.interfaces

These represent interfaces implemented by various Substance D objects.

interface substanced.interfaces.**IACLModified**

Extends: zope.interface.interfaces.IObjectEvent

May be sent when an object's ACL is modified

old_acl

The object ACL before the modification

object
The object being modified

new_acl
The object ACL after the modification

interface `substanced.interfaces.`**IAfterTransition**
An event type sent after a transition has been done

transition
The transition name

new_state
The new state of the object

object
The object on which the transition has been done

initial_state
The initial state of the object

interface `substanced.interfaces.`**ICatalog**
A collection of indices.

reset()
Clear all indexes in this catalog and clear self.objectids.

__getitem__(*name*)
Return the index named `name`

update_indexes(*registry=None*, *dry_run=False*, *output=None*, *replace=False*, *reindex=False*, ***kw*)
Use the candidate indexes registered via `config.add_catalog_factory` to populate this catalog.

reindex_resource(*resource*, *oid=None*, *action_mode=None*)
Register the resource in indexes of this catalog using objectid `oid`. If `oid` is not supplied, the __oid__ of the `resource` will be used. `action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`* indicating when the updates should take effect. The `action_mode` value will overrule any action mode that a member index has been configured with.

The result of calling this method is logically the same as calling `unindex_resource`, then `index_resource` for the same resource/oid combination, but calling those two methods in succession is often more expensive than calling this single method, as member indexes can choose to do smarter things during a reindex than what they would do during an unindex then an index.

reindex(*dry_run=False*, *commit_interval=200*, *indexes=None*, *path_re=None*, *output=None*)
Reindex all objects in this collection of indexes.

If `dry_run` is `True`, do no actual work but send what would be changed to the logger.

`commit_interval` controls the number of objects indexed between each call to `transaction.commit()` (to control memory consumption).

`indexes`, if not `None`, should be a list of index names that should be reindexed. If `indexes` is `None`, all indexes are reindexed.

`path_re`, if it is not `None` should be a regular expression object that will be matched against each object's path. If the regular expression matches, the object will be reindexed, if it does not, it won't.

`output`, if passed should be one of `None`, `False` or a function. If it is a function, the function should accept a single message argument that will be used to record the actions taken during the reindex. If

`False` is passed, no output is done. If `None` is passed (the default), the output will wind up in the `substanced.catalog` Python logger output at `info` level.

`unindex_resource`(*resource_or_oid*, *action_mode=None*)
Deregister the resource in indexes of this catalog using objectid or resource `resource_or_oid`. If `resource_or_oid` is an integer, it will be used as the oid; if `resource_or_oid` is a resource, its `__oid__` attribute will be used as the oid. `action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`*.

`index_resource`(*resource*, *oid=None*, *action_mode=None*)
Register the resource in indexes of this catalog using objectid `oid`. If `oid` is not supplied, the `__oid__` of the `resource` will be used. `action_mode`, if supplied, should be one of `None`, *`MODE_IMMEDIATE`*, *`MODE_ATCOMMIT`* or *`MODE_DEFERRED`*.

`objectids`
a sequence of objectids that are cataloged in this catalog

`flush`(*immediate=True*)
Flush any pending indexing actions for all indexes in this catalog. If `immediate` is `True`, *all* actions will be immediately executed. If `immediate` is `False`, *`MODE_DEFERRED`* actions will be sent to the actions processor if one is active, and all other actions will be executed immediately.

`interface` `substanced.interfaces.`**`IContentCreated`**
An event type sent when a Substance D content object is created via `registry.content.create`

`object`
The freshly created content object. It will not yet have been seated into any folder.

`meta`
The metainformation about the content type in the content registry

`content_type`
The content type of the object that was created

`interface` `substanced.interfaces.`**`IDefaultWorkflow`**
Marker interface used internally for workflows that aren't associated with a particular content type

`interface` `substanced.interfaces.`**`IEditable`**
Adapter interface for editing content as a file.

`put`(*fileish*)
Update context based on the contents of `fileish`.

- `fileish` is a file-type object: its `read` method should return the (new) file representation of the context.

`get`()
Return `(body_iter, mimetype)` representing the context.

- `body_iter` is an iterable, whose chunks are bytes represenating the context as an editable file.
- `mimetype` is the MIMEType corresponding to `body_iter`.

`interface` `substanced.interfaces.`**`IEvolutionSteps`**
Utility for obtaining evolution step data

`interface` `substanced.interfaces.`**`IFile`**
An object representing file content

`mimetype`
The mimetype of the file content

upload(*stream*, *mimetype_hint=False*)
Replace the current contents of this file's blob with the contents of `stream`. `mimetype_hint` can be any of the folliwing:

- `None`, meaning don't reset the current mimetype. This is the default.
- A string containing a filename with an extension; the mimetype will be derived from the extension in the filename.
- The constant *`substanced.file.USE_MAGIC`*, which will derive the content type using the `python-magic` library based on the stream's actual content.

get_size()
Return the size in bytes of the data in the blob associated with the file

blob
The ZODB blob object holding the file content

get_response(***kw*)
Return a WebOb-compatible response object which uses the blob content as the stream data and the mimetype of the file as the content type. The `**kw` arguments will be passed to the `pyramid.response.FileResponse` constructor as its keyword arguments.

interface substanced.interfaces.**IFolder**
A Folder which stores objects using Unicode keys.

All methods which accept a `name` argument expect the name to either be Unicode or a byte string decodable using the default system encoding or the UTF-8 encoding.

rename(*oldname*, *newname*)
Rename a subobject from oldname to newname.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events sent will indicate that the object is moving.

load(*name*, *newobject*)
Same as *`substanced.interfaces.IFolder.replace()`* except it causes the `loading` flag of added and removed events sent during the add and remove events implied by the replacement to be `True`.

move(*name*, *other*, *newname=None*)
Move a subobject named `name` from this folder to the folder represented by `other`. If `newname` is not none, it is used as the target object name; otherwise the existing subobject name is used.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events sent will indicate that the object is moving.

set_order(*value*, *reorderable=None*)
Makes the folder orderable and sets its order to the list of names provided in value. Names should be existing names for objects contained in the folder at the time order is set.

If `reorderable` is passed, value, it must be `None`, `True` or `False`. If it is `None`, the reorderable flag will not be reset from its current value. If it is anything except `None`, it will be treated as a boolean and the reorderable flag will be set to that value. The `reorderable` value of a folder will be returned by that folder's *`is_reorderable()`* method.

The *`is_reorderable()`* method is used by the SDI folder contents view to indicate that the folder can or cannot be reordered via the web UI.

If `reorderable` is set to `True`, the *`reorder()`* method will work properly, otherwise it will raise a `ValueError` when called.

pop(*name*, *default=None*)
Remove the item stored in the under `name` and return it.

If `name` doesn't exist in the folder, and `default` **is not** passed, raise a `KeyError`.

If `name` doesn't exist in the folder, and `default` **is** passed, return `default`.

When the object stored under `name` is removed from this folder, remove its `__parent__` and `__name__` values.

When this method is called, emit an `IObjectWillBeRemoved` event before the object loses its `__name__` or `__parent__` values. Emit an `ObjectRemoved` after the object loses its `__name__` and `__parent__` value,

`replace`(*name*, *newobject*)
Replace an existing object named `name` in this folder with a new object `newobject`. If there isn't an object named `name` in this folder, an exception will *not* be raised; instead, the new object will just be added.

This operation is done in terms of a remove and an add. The Removed and WillBeRemoved events will be sent for the old object, and the WillBeAdded and Add events will be sent for the new object.

`__contains__`(*name*)
Does the container contains an object named by name?

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding or the UTF-8 encoding.

`is_reorderable`()
Return true if the folder can be reordered, false otherwise.

`keys`()
Return an iterable sequence of object names present in the folder.

Respect `order`, if set.

`add`(*name*, *other*, *send_events=True*, *reserved_names=()*, *duplicating=None*, *moving=None*, *loading=False*, *registry=None*)
Same as `__setitem__`.

If `send_events` is false, suppress the sending of folder events. Disallow the addition of the name provided is in the `reserved_names` list. If `duplicating` is not None, it must be the object being duplicated; when non-None, the ObjectWillBeAdded and ObjectAdded events sent will be marked as 'duplicating', which typically has the effect that the subobject's object id will be overwritten instead of reused. If `registry` is passed, it should be a Pyramid registry object; otherwise the `pyramid.threadlocal.get_current_registry()` function is used to look up the current registry.

This method returns the name used to place the subobject in the folder (a derivation of `name`, usually the result of `self.check_name(name)`).

`__len__`()
Return the number of subobjects in this folder.

`sort`(*oids*, *reverse=False*, *limit=None*)
Return the intersection of the oids of the folder's order with the oids passed in. If `reverse` is True, reverse the result set. If `limit` is an integer, return only that number of items (after reversing, if reverse is True).

`__getitem__`(*name*)
Return the object represented by `name` in this folder or raise a KeyError if no such object exists.

`get`(*name*, *default=None*)
Return the object named by `name` or the default.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding or the UTF-8 encoding.

is_ordered()
Return `True` if the folder has a manual ordering (e.g. its `order` attribute has been set), `False` otherwise.

unset_order()
Removes the folder internal ordering, making it an unordered folder.

validate_name(*name*, *reserved_names=()*)
Checks the name passed for validity. If the name is valid and is not present in `reserved_names` returns a validated name. Otherwise a `ValueError` will be raised.

__iter__()
An alias for `keys`.

Respect `order`, if set.

check_name(*name*, *reserved_names=()*)
Performs all checks associated with `validate_name` but also raises a *`substanced.folder.FolderKeyError`* if an object with the name `name` already exists in the folder. Returns the name (with any modifications) returned by `validate_name`.

__delitem__(*name*)
Remove the object from this folder stored under `name`.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding or the UTF-8 encoding.

If no object is stored in the folder under `name`, raise a `KeyError`.

When the object stored under `name` is removed from this folder, remove its `__parent__` and `__name__` values.

When this method is called, emit an `IObjectWillBeRemoved` event before the object loses its `__name__` or `__parent__` values. Emit an `IObjectRemoved` after the object loses its `__name__` and `__parent__` value,

__nonzero__()
Always return True

items()
Return an iterable sequence of (name, value) pairs in the folder.

Respect `order`, if set.

clear()
Clear all objects from the folder. Calling this is equivalent to calling `.remove` for each key in the folder.

__setitem__(*name*, *other*)
Set object `other` into this folder under the name `name`.

`name` must be a Unicode object or a bytestring object.

If `name` is a bytestring object, it must be decodable using the system default encoding or the UTF-8 encoding.

`name` cannot be the empty string.

When `other` is seated into this folder, it will also be decorated with a `__parent__` attribute (a reference to the folder into which it is being seated) and `__name__` attribute (the name passed in to this function.

If a value already exists in the foldr under the name `name`, raise `KeyError`.

When this method is called, emit an `IObjectWillBeAdded` event before the object obtains a `__name__` or `__parent__` value. Emit an `IObjectAdded` event after the object obtains a `__name__` and `__parent__` value.

remove (*name*, *send_events=True*, *moving=None*, *loading=False*)
Same thing as `__delitem__`.

If `send_events` is false, suppress the sending of folder events. If `moving` is not `None`, it should be the folder object from which the object is being moved; the events sent will indicate that a move is in process.

values ()
Return an iterable sequence of the values present in the folder.

Respect `order`, if set.

reorder (*items*, *before*)
Move one or more items from a folder into new positions inside that folder. `items` is a list of ids of existing folder items, which will be inserted in order before the item named `before`. All other items are left in the original order. If this method is called on a folder which does not have an order set, or which is not reorderable, a `ValueError` will be raised.

interface `substanced.interfaces.`**IGroup**
Extends: *`substanced.interfaces.IPrincipal`*

Marker interface representing a group

interface `substanced.interfaces.`**IGroups**
Marker interface representing a collection of groups

interface `substanced.interfaces.`**IIndexFactory**
A factory for an index

interface `substanced.interfaces.`**IIndexingActionProcessor**
Processor of deferred indexing/unindexing actions of catalogs in the system

interface `substanced.interfaces.`**ILock**
Represents a lock to be applied by the lock service

interface `substanced.interfaces.`**ILoggedIn**
An event type sent when a user supplies a valid username and password to a login view. Note that this event is not sent on *every* request that the user initiates, just ones which result in an interactive login.

request
The request which resulted in the login

user
The user object computed by Substance D

context
The context resource that was active during login

login
The login name used by the user

interface `substanced.interfaces.`**IObjectAdded**
Extends: `zope.interface.interfaces.IObjectEvent`

An event type sent when an object is added

loading
Boolean indicating that this add is part of a load (during a dump load process)

name
The name of the object within the folder

parent
The folder to which the object is being added

object
The object being added

duplicating
The object being duplicated or `None`

moving
None or the folder from which the object being added was moved

interface `substanced.interfaces.`**IObjectMap**
A map of objects to paths and a reference engine

pathlookup (*obj_or_path_tuple*, *depth=None*, *include_origin=True*)
Returns an iterator of document ids within obj_or_path_tuple (a traversable object or a path tuple). If depth is specified, returns only objects at that depth. If `include_origin` is `True`, returns the docid of the object passed as `obj_or_path_tuple` in the returned set, otherwise it omits it.

disconnect (*src*, *target*, *reftype*)
Disonnect `src_object` from `target_object` using the reference type `reftype`. `src` and `target` may be objects or object identifiers

path_for (*objectid*)
Return the path tuple for objectid

objectid_for (*obj_or_path_tuple*)
Return the object id for obj_or_path_tuple

remove (*obj_objectid_or_path_tuple*)
Removes an object from the object map using the object itself, an object id, or a path tuple. Returns a set of objectids (children, inclusive) removed as the result of removing this object from the object map.

targets (*obj*, *reftype*)
Return a generator consisting of objects which have `obj` as a relationship target using `reftype`. `obj` can be an object or an object id.

sources (*obj*, *reftype*)
Return a generator consisting of objects which have `obj` as a relationship source using `reftype`. `obj` can be an object or an object id.

add (*obj*)
Add a new object to the object map. Assigns a new objectid to obj.__oid__ to the object if it doesn't already have one. The object's path or objectid must not already exist in the map. Returns the object id.

connect (*src*, *target*, *reftype*)
Connect `src_object` to `target_object` using the reference type `reftype`. `src` and `target` may be objects or object identifiers.

targetids (*obj*, *reftype*)
Return a set of objectids which have `obj` as a relationship target using `reftype`. `obj` can be an object or an object id.

object_for (*objectid*)
Return the object associated with `objectid` or `None` if the object cannot be found.

sourceids (*obj*, *reftype*)
Return a set of objectids which have `obj` as a relationship source using `reftype`. `obj` can be an object or an object id.

interface `substanced.interfaces.`**`IObjectModified`**

Extends: `zope.interface.interfaces.IObjectEvent`

May be sent when an object is modified

`object`
The object being modified

interface `substanced.interfaces.`**`IObjectRemoved`**

Extends: `zope.interface.interfaces.IObjectEvent`

An event type sent when an object is removed

`loading`
Boolean indicating that this remove is part of a load (during a dump load process)

`name`
The name of the object within the folder

`parent`
The folder from which the object is being removed

`object`
The object being removed

`moving`
None or the folder to which the object being removed will be moved

`removed_oids`
The set of oids removed as the result of this object being removed (including the oid of the object itself). This may be any number of oids if the object was folderish

interface `substanced.interfaces.`**`IObjectWillBeAdded`**

Extends: `zope.interface.interfaces.IObjectEvent`

An event type sent when an before an object is added

`loading`
Boolean indicating that this add is part of a load (during a dump load process)

`name`
The name which the object is being added to the folder with

`parent`
The folder to which the object is being added

`object`
The object being added

`duplicating`
The object being duplicated or `None`

`moving`
None or the folder from which the object being added was moved

interface `substanced.interfaces.`**`IObjectWillBeRemoved`**

Extends: `zope.interface.interfaces.IObjectEvent`

An event type sent before an object is removed

`loading`
Boolean indicating that this remove is part of a load (during a dump load process)

`name`
The name of the object within the folder

parent
: The folder from which the object is being removed

object
: The object being removed

moving
: None or the folder to which the object being removed will be moved

interface `substanced.interfaces.`**`IPasswordReset`**
: Marker interface represent a password reset request

interface `substanced.interfaces.`**`IPasswordResets`**
: Marker interface representing a collection of password reset requests

interface `substanced.interfaces.`**`IPrincipal`**
: Marker interface representing a user or group

interface `substanced.interfaces.`**`IPrincipals`**
: Marker interface representing a container of users and groups

interface `substanced.interfaces.`**`IPropertySheet`**
: Interface for objects with a set of properties defined by a Colander schema. The class *`substanced.property.PropertySheet`* (which is meant to be subclassed for specialization) implements this interface.

set (*struct*, *omit=()*)
: Accept `struct` (a dictionary representing the property state) and persist it to the context, refraining from persisting the keys in the struct that are named in `omit` (a sequence of strings or a string). The data structure will have already been validated against the propertysheet schema.

  You can return a value from this method. It will be passed as `changed` into the `after_set` method. It should be `False` if your `set` implementation *did not* change any persistent data. Any other return value will be conventionally interpreted as the implementation having changed persistent data.

get ()
: Return a dictionary representing the current property state compatible with the schema for serialization

request
: The current request

context
: The context of the property sheet (a resource)

after_set (*changed*)
: Perform operations after a successful set. `changed` is the value returned from the `set` method.

  The default propertysheet implementation sends an ObjectModified event if the `changed` value is not `False`.

schema
: The Colander schema instance which defines the fields related to this property sheet

interface `substanced.interfaces.`**`IRootAdded`**
: An event type sent when the Substance D root object has a connection to the database as its `_p_jar` attribute.

object
: The root object

interface `substanced.interfaces.`**`ISDIAPI`**
: Easy access to common templating operations on all views. This object is available as `request.sdiapi`.

mgmt_path(*obj*, **arg*, ***kw*)
Return the route_path inside the SDI for an object

sdi_title()
The `sdi_title` of the virtual root or "Substance D" if not defined

mgmt_views(*context*)
The list of management views on a resource

breadcrumbs()
Return a sequence of dicts for the breadcrumb information. Each dict contains:

- `url`: The `request.mgmt_path` to that resource
- `name`: The resource `__name__` or 'Home' for the root
- `active`: Boolean representing whether the resource is in the breadcrumb is the current context
- `icon`: The full path to the icon for that resource type

get_macro(*asset_spec*, *name=None*)
Return a Chameleon template macro based on the asset spec (e.g. `somepackage:templates/foo.pt`) and the name. If the name is None, the bare template implementation is returned, otherwise the named macro from within the template is returned.

mgmt_url(*obj*, **arg*, ***kw*)
Return the route_url inside the SDI for an object

main_template
The loaded `master.pt` which can be used in view templates with `metal:use-macro="request.sdiapi.main_template"`.

flash_with_undo(*msg*, *queue=''*, *allow_duplicate=True*)
Display a Pyramid `flash message` to the appropriate queue with a button to allow an undo of the commit.

interface `substanced.interfaces.`**IService**
Marker for items which are showin in the "Services" tab.

interface `substanced.interfaces.`**IUser**
Extends: *`substanced.interfaces.IPrincipal`*

Marker interface representing a user

interface `substanced.interfaces.`**IUserLocator**
Adapter responsible for returning a user by his login name and/or userid as well as group objects of a user by his userid.

get_groupids(*userid*)
Return all the group-related principal identifiers for a user with the user principal identifier `userid` as a sequence. If no user exists under `userid`, return `None`.

get_user_by_email(*email*)
Return an IUser object or `None` if no such user exists. The `email` argument is the *email address* of the user.

get_user_by_userid(*userid*)
Return an IUser object or `None` if no such user exists. The `userid` argument is the *user id* of the user (usually an oid).

get_user_by_login(*login*)
Return an IUser object or `None` if no such user exists. The `login` argument is the *login name* of the user, not an oid.

interface `substanced.interfaces.`**`IUsers`**
Marker interface representing a collection of users

interface `substanced.interfaces.`**`IWorkflow`**

`reset`(*content*, *request=None*)

`has_state`(*content*)

`get_transitions`(*content*, *request*, *from_state=None*)

`add_state`(*name*, *callback=None*, ***kw*)

`transition`(*content*, *request*, *transition_name*)

`state_of`(*content*)

`check`()

`get_states`(*content*, *request*, *from_state=None*)

`initialize`(*content*, *request=None*)

`transition_to_state`(*content*, *request*, *to_state*, *skip_same=True*)

`add_transition`(*name*, *from_state*, *to_state*, *callback=None*, ***kw*)

interface `substanced.interfaces.`**`MODE_ATCOMMIT`**
Sentinel indicating that an indexing action should take place at the successful end of the current transaction.

interface `substanced.interfaces.`**`MODE_DEFERRED`**
Sentinel indicating that an indexing action should be performed by an external indexing processor (e.g. `drain_catalog_indexing`) if one is active at the successful end of the current transaction. If an indexing processor is unavailable at the successful end of the current transaction, this mode will be taken to imply the same thing as *`MODE_ATCOMMIT`*.

interface `substanced.interfaces.`**`MODE_IMMEDIATE`**
Sentinel indicating that an indexing action should take place as immediately as possible.

class `substanced.interfaces.`**`ReferenceClass`**(**arg*, ***kw*)
Bases: `zope.interface.interface.InterfaceClass`

interface `substanced.interfaces.`**`UserToLock`**
Extends: `substanced.interfaces.ReferenceType`

A reference type which represents the relationship from a user to his set of locks

interface `substanced.interfaces.`**`WriteLock`**
Extends: `substanced.interfaces.ReferenceType`

Represents a DAV-style writelock. It's a Substance D reference type from resource object to lock object

5.28 Substance D SDI Permission Names

sdi.add-content

Protects views which allow users to add content to a folder.

sdi.add-group

Protects views which add groups to a groups collection within a principals service.

sdi.add-services

Protects views which add built-in Substance D services.

sdi.add-user

Protects views which add users to a users collection within a principals service.

sdi.change-acls

Protects arbitrary locations, allowing certain people to execute views the under that location which change ACLs associated with a resource.

sdi.change-password

Protects views of a user which allow for the changing of passwords.

sdi.lock

Protects views which allow users to lock or unlock a resource.

sdi.manage-catalog

Protects views which allow users to manage catalog data and indexes within a catalog service.

sdi.manage-contents

Protects views which allow users to add, remove, and rename items within folders.

sdi.manage-database

Protects the "manage database" view at the root.

sdi.manage-references

Protects views which allow users to manage the references associated with a resource.

sdi.manage-user-groups

Protects views which allow admin users to update groups for users.

sdi.manage-workflow

Protects the views associated with managing the workflows of an object.

sdi.undo

Protects the capability of users to execute views which undo transactions.

sdi.view

Protects whether a user can view the SDI management pages associated with a resource.

sdi.view-services

Protects whether a user can view the "Services" tab in a folder.

sdi.edit-properties

Allows for the editing of the properties of a property sheet for an object.

sdi.view-auditlog

Allows the user to view the audit log event stream (`auditstream-sse`) view.

CHAPTER 6

Support / Reporting Bugs / Development Versions

Visit http://github.com/Pylons/substanced to download development or tagged versions.

Visit http://github.com/Pylons/substanced/issues to report bugs.

The mailing list exists at https://groups.google.com/group/substanced-users

The IRC channel is at irc://freenode.net/#substanced

CHAPTER 7

Copyright, Trademarks, and Attributions

7.1 Copyright, Trademarks, and Attributions

7.1.1 Attributions

Editor: TBD

Contributors: Steve Piercy, Eric Rasmussen, Domen Kožar, Paul Everitt, Carlos de la Guardia, Balazs Ree, Douglas Cerna, and a number of people with only pseudonyms on GitHub.

SubstanceD.net Website Theme: The assets in the directory *assets* are *not* open source; they are copyrighted by Tamerlan Soziev and released under a proprietary license. To purchase a license for these assets, visit https://wrapbootstrap.com/theme/venera-responsive-multipurpose-template- WB059C895

Documentation Template: Steve Piercy based on the Venera theme. Used by permission.

Some Substance D Interface images copyright Rokey, in particular http://www.iconarchive.com/show/smooth-icons-by-rokey/capsule-icon.html

7.1.2 Contacting The Publisher

Please send documentation licensing inquiries, translation inquiries, and other business communications to Agendaless Consulting. For software and other technical queries see *Support / Reporting Bugs / Development Versions*.

7.1.3 HTML Version and Source Code

The source code for the examples used in this documentation are available within the Substance D software distribution, available via https://github.com/Pylons/substanced

CHAPTER 8

Indices and tables

- *Glossary*
- genindex
- modindex
- search

8.1 Glossary

Colander A schema library which can be used used to describe arbitrary data structures. See http://docs.pylonsproject.org/projects/colander/en/latest/ for more information.

Content A *resource* which is particularly well-behaved when viewed via the Substance D management interface.

Content type An interface associated with a particular kind of content object. A content type also has metadata like an icon, an add view name, and other things.

DataDog A Software-as-a-Service (SaaS) provider for monitoring and visualizing performance data that is compatible with the `statsd` statistics output channel used by Substance D. See http://www.datadoghq.com

Deform A form library that draws and validates forms based on *Colander* schemas. See http://docs.pylonsproject.org/projects/deform/en/latest/ for more information.

Factory Wrapper A function that wraps a content factory when the content factory is not a class or when a `factory_name` is used within the content type declaration.

Folder A resource object which contains other resource objects. See *`substanced.folder.Folder`*.

Global Object A Python object that can be obtained via an `import` statement.

Manage prefix The prepended portion of the URL (usually `/manage`) which signifies that view lookup should be done only amongst the set of views registered as *management view* types. This can be changed by setting the `substanced.manage_prefix` key in your `development.ini` or `production.ini` configuration files.

Management view A view configuration that is only invoked when a user visits a URL prepended with the *manage prefix*.

Object Map A Substance D *service* which maps the object IDs of persistent objects to paths and object IDs to other object IDs in the system.

Object Map Reference A relationship kept in the *object map* between two persistent objects. It is composed of a source, some number of targets, and a *reference type*.

Pyramid A web framework.

Reference Type A hashable object describing the type of relationship between two objects in the *object map*. It's usually a string.

Resource An object representing a node in the *resource tree* of your Substance D application. A resource becomes the context of a view when someone visits a URL in your application.

Resource factory An object which creates a *resource* when it's called. It's often just a class that implements the resource itself.

Resource tree A nested set of *folder* objects and other kinds of content objects, each of which is a *resource*. Your content objects are laid out hierarchically in the resource tree as they're added.

SDI An acronym for the "Substance D (Management) Interface". What you see when you visit `/manage`.

Service A persistent object in the *resource tree* that exposes an API to application developers. For example, the `principals` service.

Service A Substance D content object which provides a service to application code (such as a catalog or a principals service).

State

States TODO

Transition

Transitions TODO

Workflow

Workflows TODO

Zope An application server from which much of the spirit of Substance D is derived. See http://zope.org.

Python Module Index

h

hypatia.query, 82
hypatia.util, 84

s

substanced, 75
substanced.audit, 75
substanced.catalog, 75
substanced.catalog.indexes, 81
substanced.content, 85
substanced.db, 87
substanced.editable, 87
substanced.event, 88
substanced.evolution, 89
substanced.file, 90
substanced.folder, 91
substanced.form, 95
substanced.interfaces, 111
substanced.locking, 95
substanced.objectmap, 97
substanced.property, 101
substanced.root, 103
substanced.schema, 102
substanced.stats, 103
substanced.util, 104
substanced.workflow, 108

Index

Symbols

__contains__() (substanced.interfaces.IFolder method), 115
__delitem__() (substanced.interfaces.IFolder method), 116
__getitem__() (substanced.catalog.Catalog method), 76
__getitem__() (substanced.interfaces.ICatalog method), 112
__getitem__() (substanced.interfaces.IFolder method), 115
__init__() (substanced.file.File method), 90
__init__() (substanced.folder.Folder method), 91
__init__() (substanced.folder.RandomAutoNamingFolder method), 94
__init__() (substanced.folder.SequentialAutoNamingFolder method), 94
__iter__() (hypatia.interfaces.IResultSet method), 85
__iter__() (substanced.interfaces.IFolder method), 116
__len__() (hypatia.interfaces.IResultSet method), 85
__len__() (substanced.interfaces.IFolder method), 115
__nonzero__() (substanced.interfaces.IFolder method), 116
__setitem__() (substanced.catalog.Catalog method), 76
__setitem__() (substanced.interfaces.IFolder method), 116

A

ACLModified (class in substanced.event), 88
ACLState (class in substanced.workflow), 108
ACLWorkflow (class in substanced.workflow), 108
acquire() (in module substanced.util), 104
add() (substanced.audit.AuditLog method), 75
add() (substanced.content.ContentRegistry method), 86
add() (substanced.folder.Folder method), 91
add() (substanced.folder.SequentialAutoNamingFolder method), 94
add() (substanced.interfaces.IFolder method), 115
add() (substanced.interfaces.IObjectMap method), 118
add() (substanced.objectmap.ObjectMap method), 97
add_catalog() (substanced.catalog.CatalogsService method), 79
add_catalog_factory() (in module substanced.catalog), 80
add_content_type() (in module substanced.content), 85
add_evolution_step() (in module substanced.evolution), 89
add_indexview() (in module substanced.catalog), 80
add_next() (substanced.folder.RandomAutoNamingFolder method), 94
add_next() (substanced.folder.SequentialAutoNamingFolder method), 94
add_propertysheet() (in module substanced.property), 101
add_propertysheet_predicate() (in module substanced.property), 102
add_service() (substanced.folder.Folder method), 91
add_service_type() (in module substanced.content), 86
add_state() (substanced.interfaces.IWorkflow method), 122
add_state() (substanced.workflow.Workflow method), 108
add_transition() (substanced.interfaces.IWorkflow method), 122
add_transition() (substanced.workflow.Workflow method), 109
add_workflow() (in module substanced.workflow), 111
after_set() (substanced.interfaces.IPropertySheet method), 120
AfterTransition (class in substanced.event), 88
All (class in hypatia.query), 83
all() (hypatia.interfaces.IResultSet method), 84
all() (substanced.content.ContentRegistry method), 86
Allowed (class in substanced.catalog), 76
allowed() (substanced.objectmap.ObjectMap method), 97
AllowedIndex (class in substanced.catalog.indexes), 82
allows() (substanced.catalog.indexes.AllowedIndex method), 82
And (class in hypatia.query), 83
Any (class in hypatia.query), 83
AuditLog (class in substanced.audit), 75

B

Batch (class in substanced.util), 106
before_render() (substanced.property.PropertySheet method), 101, 102
blob (substanced.file.File attribute), 90
blob (substanced.interfaces.IFile attribute), 114
breadcrumbs() (substanced.interfaces.ISDIAPI method), 121

C

Catalog (class in substanced.catalog), 76
catalog_factory() (in module substanced.catalog), 80
CatalogsService (class in substanced.catalog), 78
CatalogsService.Catalog (class in substanced.catalog), 78
check() (substanced.interfaces.IWorkflow method), 122
check() (substanced.workflow.Workflow method), 109
check_name() (substanced.folder.Folder method), 91
check_name() (substanced.interfaces.IFolder method), 116
chunks() (in module substanced.util), 106
clear() (substanced.folder.Folder method), 91
clear() (substanced.interfaces.IFolder method), 116
clear() (substanced.objectmap.Multireference method), 99
coarse_datetime_repr() (in module substanced.util), 106
Colander, **129**
commit_suicide() (substanced.locking.Lock method), 96
connect() (substanced.interfaces.IObjectMap method), 118
connect() (substanced.objectmap.Multireference method), 99
connect() (substanced.objectmap.ObjectMap method), 97
Contains (class in hypatia.query), 82, 83
Content, **129**
content (class in substanced.content), 85
Content type, **129**
content_type (substanced.interfaces.IContentCreated attribute), 113
ContentRegistry (class in substanced.content), 86
context (substanced.interfaces.ILoggedIn attribute), 117
context (substanced.interfaces.IPropertySheet attribute), 120
copy() (substanced.folder.Folder method), 91
create() (substanced.content.ContentRegistry method), 87

D

DataDog, **129**
Deform, **129**
disconnect() (substanced.interfaces.IObjectMap method), 118
disconnect() (substanced.objectmap.Multireference method), 99
disconnect() (substanced.objectmap.ObjectMap method), 97
discover_resource_locks() (in module substanced.locking), 97
duplicating (substanced.interfaces.IObjectAdded attribute), 118
duplicating (substanced.interfaces.IObjectWillBeAdded attribute), 119

E

Eq (class in hypatia.query), 82
event (substanced.event.subscribe_acl_modified attribute), 89
event (substanced.event.subscribe_added attribute), 89
event (substanced.event.subscribe_after_transition attribute), 89
event (substanced.event.subscribe_logged_in attribute), 89
event (substanced.event.subscribe_modified attribute), 89
event (substanced.event.subscribe_removed attribute), 89
event (substanced.event.subscribe_root_added attribute), 89
event (substanced.event.subscribe_will_be_added attribute), 89
event (substanced.event.subscribe_will_be_removed attribute), 89
exists() (substanced.content.ContentRegistry method), 87
expires() (substanced.locking.Lock method), 96

F

Facet (class in substanced.catalog), 76
FacetIndex (class in substanced.catalog.indexes), 81
Factory Wrapper, **129**
factory_type_for_content_type() (substanced.content.ContentRegistry method), 87
Field (class in substanced.catalog), 75
FieldIndex (class in substanced.catalog.indexes), 81
File (class in substanced.file), 90
FileEditable (class in substanced.editable), 88
FileUploadTempStore (class in substanced.form), 95
find() (substanced.content.ContentRegistry method), 87
find_catalog() (in module substanced.util), 105
find_catalogs() (in module substanced.util), 105
find_content() (in module substanced.util), 105
find_index() (in module substanced.util), 105
find_objectmap() (in module substanced.util), 105
find_service() (in module substanced.util), 105
find_service() (substanced.folder.Folder method), 92
find_services() (in module substanced.util), 105
find_services() (substanced.folder.Folder method), 92
first() (hypatia.interfaces.IResultSet method), 85
flash_with_undo() (substanced.interfaces.ISDIAPI method), 121
flush() (substanced.catalog.Catalog method), 76

flush() (substanced.catalog.CatalogsService.Catalog method), 78
flush() (substanced.interfaces.ICatalog method), 113
Folder, **129**
Folder (class in substanced.folder), 91
FolderKeyError (class in substanced.folder), 91
Form (class in substanced.form), 95
form_class (substanced.form.FormView attribute), 95
FormView (class in substanced.form), 95

G

Ge (class in hypatia.query), 82
get() (substanced.catalog.Catalog method), 76
get() (substanced.editable.IEditable method), 88
get() (substanced.folder.Folder method), 92
get() (substanced.interfaces.IEditable method), 113
get() (substanced.interfaces.IFolder method), 115
get() (substanced.interfaces.IPropertySheet method), 120
get_acl() (in module substanced.util), 104
get_auditlog() (in module substanced.util), 106
get_content_type() (in module substanced.util), 104
get_dotted_name() (in module substanced.util), 106
get_editable_adapter() (in module substanced.editable), 88
get_etag() (substanced.file.File method), 90
get_extent() (substanced.objectmap.ObjectMap method), 97
get_factory_type() (in module substanced.util), 106
get_groupids() (substanced.interfaces.IUserLocator method), 121
get_icon_name() (in module substanced.util), 106
get_interfaces() (in module substanced.util), 104
get_macro() (substanced.interfaces.ISDIAPI method), 121
get_objects() (substanced.objectmap.ReferentialIntegrityError method), 101
get_oid() (in module substanced.util), 104
get_principal_repr() (in module substanced.util), 105
get_reftypes() (substanced.objectmap.ObjectMap method), 97
get_response() (substanced.file.File method), 90
get_response() (substanced.interfaces.IFile method), 114
get_size() (substanced.file.File method), 90
get_size() (substanced.interfaces.IFile method), 114
get_states() (substanced.interfaces.IWorkflow method), 122
get_states() (substanced.workflow.Workflow method), 109
get_transitions() (substanced.interfaces.IWorkflow method), 122
get_transitions() (substanced.workflow.Workflow method), 109
get_user_by_email() (substanced.interfaces.IUserLocator method), 121
get_user_by_login() (substanced.interfaces.IUserLocator method), 121
get_user_by_userid() (substanced.interfaces.IUserLocator method), 121
get_workflow() (in module substanced.workflow), 111
Global Object, **129**
Gt (class in hypatia.query), 82

H

has_references() (substanced.objectmap.ObjectMap method), 97
has_state() (substanced.interfaces.IWorkflow method), 122
has_state() (substanced.workflow.Workflow method), 110
hypatia.query (module), 82
hypatia.util (module), 84

I

IACLModified (interface in substanced.interfaces), 111
IAfterTransition (interface in substanced.interfaces), 112
ICatalog (interface in substanced.interfaces), 112
IContentCreated (interface in substanced.interfaces), 113
IDefaultWorkflow (interface in substanced.interfaces), 113
ids (hypatia.interfaces.IResultSet attribute), 85
IEditable (interface in substanced.editable), 87
IEditable (interface in substanced.interfaces), 113
IEvolutionSteps (interface in substanced.interfaces), 113
IFile (interface in substanced.interfaces), 113
IFolder (interface in substanced.interfaces), 114
IGroup (interface in substanced.interfaces), 117
IGroups (interface in substanced.interfaces), 117
IIndexFactory (interface in substanced.interfaces), 117
IIndexingActionProcessor (interface in substanced.interfaces), 117
ILock (interface in substanced.interfaces), 117
ILoggedIn (interface in substanced.interfaces), 117
include() (in module substanced), 75
includeme() (in module substanced), 75
includeme() (in module substanced.catalog), 80
includeme() (in module substanced.content), 87
includeme() (in module substanced.db), 87
includeme() (in module substanced.evolution), 90
index_resource() (substanced.catalog.Catalog method), 76
index_resource() (substanced.catalog.CatalogsService.Catalog method), 78
index_resource() (substanced.interfaces.ICatalog method), 113
indexview (class in substanced.catalog), 81
indexview_defaults (class in substanced.catalog), 81
initial_state (substanced.interfaces.IAfterTransition attribute), 112

initialize() (substanced.interfaces.IWorkflow method), 122
initialize() (substanced.workflow.Workflow method), 110
InRange (class in hypatia.query), 83
intersect() (hypatia.util.ResultSet method), 84
IObjectAdded (interface in substanced.interfaces), 117
IObjectMap (interface in substanced.interfaces), 118
IObjectModified (interface in substanced.interfaces), 118
IObjectRemoved (interface in substanced.interfaces), 119
IObjectWillBeAdded (interface in substanced.interfaces), 119
IObjectWillBeRemoved (interface in substanced.interfaces), 119
IPasswordReset (interface in substanced.interfaces), 120
IPasswordResets (interface in substanced.interfaces), 120
IPrincipal (interface in substanced.interfaces), 120
IPrincipals (interface in substanced.interfaces), 120
IPropertySheet (interface in substanced.interfaces), 120
IResultSet (interface in hypatia.interfaces), 84
IRootAdded (interface in substanced.interfaces), 120
is_catalogable() (in module substanced.catalog), 80
is_folder() (in module substanced.util), 105
is_ordered() (substanced.folder.Folder method), 92
is_ordered() (substanced.interfaces.IFolder method), 116
is_reorderable() (substanced.folder.Folder method), 92
is_reorderable() (substanced.interfaces.IFolder method), 115
is_service() (in module substanced.util), 106
is_valid() (substanced.locking.Lock method), 96
ISDIAPI (interface in substanced.interfaces), 120
IService (interface in substanced.interfaces), 121
istype() (substanced.content.ContentRegistry method), 87
items() (substanced.folder.Folder method), 92
items() (substanced.interfaces.IFolder method), 116
IUser (interface in substanced.interfaces), 121
IUserLocator (interface in substanced.interfaces), 121
IUsers (interface in substanced.interfaces), 121
IWorkflow (interface in substanced.interfaces), 122

K

keys() (substanced.folder.Folder method), 92
keys() (substanced.interfaces.IFolder method), 115
Keyword (class in substanced.catalog), 76
KeywordIndex (class in substanced.catalog.indexes), 81

L

latest_id() (substanced.audit.AuditLog method), 75
Le (class in hypatia.query), 83
LEFT (in module substanced.schema), 103
load() (substanced.folder.Folder method), 92
load() (substanced.interfaces.IFolder method), 114
loading (substanced.interfaces.IObjectAdded attribute), 117
loading (substanced.interfaces.IObjectRemoved attribute), 119
loading (substanced.interfaces.IObjectWillBeAdded attribute), 119
loading (substanced.interfaces.IObjectWillBeRemoved attribute), 119
Lock (class in substanced.locking), 95
lock_resource() (in module substanced.locking), 96
LockError (class in substanced.locking), 96
LoggedIn (class in substanced.event), 88
login (substanced.interfaces.ILoggedIn attribute), 117
Lt (class in hypatia.query), 82

M

main_template (substanced.interfaces.ISDIAPI attribute), 121
make_columns() (substanced.util.Batch method), 108
Manage prefix, **129**
Management view, **130**
mark_unfinished_as_finished() (in module substanced.evolution), 90
merge_url_qs() (in module substanced.util), 106
meta (substanced.interfaces.IContentCreated attribute), 113
metadata() (substanced.content.ContentRegistry method), 87
mgmt_path() (substanced.interfaces.ISDIAPI method), 120
mgmt_url() (substanced.interfaces.ISDIAPI method), 121
mgmt_views() (substanced.interfaces.ISDIAPI method), 121
MIDDLE (in module substanced.schema), 103
mimetype (substanced.file.File attribute), 90
mimetype (substanced.interfaces.IFile attribute), 113
MODE_ATCOMMIT (interface in substanced.interfaces), 122
MODE_DEFERRED (interface in substanced.interfaces), 122
MODE_IMMEDIATE (interface in substanced.interfaces), 122
move() (substanced.folder.Folder method), 92
move() (substanced.interfaces.IFolder method), 114
moving (substanced.interfaces.IObjectAdded attribute), 118
moving (substanced.interfaces.IObjectRemoved attribute), 119
moving (substanced.interfaces.IObjectWillBeAdded attribute), 119
moving (substanced.interfaces.IObjectWillBeRemoved attribute), 120
Multireference (class in substanced.objectmap), 98
multireference_source_property() (in module substanced.objectmap), 100

multireference_sourceid_property() (in module substanced.objectmap), 100
multireference_target_property() (in module substanced.objectmap), 101
multireference_targetid_property() (in module substanced.objectmap), 100

N

Name (class in hypatia.query), 84
name (substanced.interfaces.IObjectAdded attribute), 117
name (substanced.interfaces.IObjectRemoved attribute), 119
name (substanced.interfaces.IObjectWillBeAdded attribute), 119
name (substanced.interfaces.IObjectWillBeRemoved attribute), 119
NameSchemaNode (class in substanced.schema), 102
new_acl (substanced.interfaces.IACLModified attribute), 112
new_objectid() (substanced.objectmap.ObjectMap method), 97
new_state (substanced.interfaces.IAfterTransition attribute), 112
newer() (substanced.audit.AuditLog method), 75
next_name() (substanced.folder.RandomAutoNamingFolder method), 95
next_name() (substanced.folder.SequentialAutoNamingFolder method), 94
Not (class in hypatia.query), 83
NotAll (class in hypatia.query), 83
NotAny (class in hypatia.query), 83
NotContains (class in hypatia.query), 83
NotEq (class in hypatia.query), 82
NotInRange (class in hypatia.query), 83

O

object (substanced.interfaces.IACLModified attribute), 111
object (substanced.interfaces.IAfterTransition attribute), 112
object (substanced.interfaces.IContentCreated attribute), 113
object (substanced.interfaces.IObjectAdded attribute), 118
object (substanced.interfaces.IObjectModified attribute), 119
object (substanced.interfaces.IObjectRemoved attribute), 119
object (substanced.interfaces.IObjectWillBeAdded attribute), 119
object (substanced.interfaces.IObjectWillBeRemoved attribute), 120
object (substanced.interfaces.IRootAdded attribute), 120
Object Map, **130**
Object Map Reference, **130**
object_for() (substanced.interfaces.IObjectMap method), 118
object_for() (substanced.objectmap.ObjectMap method), 98
ObjectAdded (class in substanced.event), 88
objectid_for() (substanced.interfaces.IObjectMap method), 118
objectid_for() (substanced.objectmap.ObjectMap method), 98
objectids (substanced.interfaces.ICatalog attribute), 113
ObjectMap (class in substanced.objectmap), 97
ObjectModified (class in substanced.event), 88
ObjectRemoved (class in substanced.event), 88
ObjectWillBeAdded (class in substanced.event), 88
ObjectWillBeRemoved (class in substanced.event), 88
old_acl (substanced.interfaces.IACLModified attribute), 111
one() (hypatia.interfaces.IResultSet method), 85
Or (class in hypatia.query), 83
order (substanced.folder.Folder attribute), 91, 92
order_sources() (substanced.objectmap.ObjectMap method), 98
order_targets() (substanced.objectmap.ObjectMap method), 98
owner (substanced.locking.Lock attribute), 95
ownerid (substanced.locking.Lock attribute), 95

P

parent (substanced.interfaces.IObjectAdded attribute), 117
parent (substanced.interfaces.IObjectRemoved attribute), 119
parent (substanced.interfaces.IObjectWillBeAdded attribute), 119
parent (substanced.interfaces.IObjectWillBeRemoved attribute), 119
parse_query() (in module hypatia.query), 84
Path (class in substanced.catalog), 76
path_for() (substanced.interfaces.IObjectMap method), 118
path_for() (substanced.objectmap.ObjectMap method), 98
pathcount() (substanced.objectmap.ObjectMap method), 98
PathIndex (class in substanced.catalog.indexes), 81
pathlookup() (substanced.interfaces.IObjectMap method), 118
pathlookup() (substanced.objectmap.ObjectMap method), 98
PermissionsSchemaNode (class in substanced.schema), 103
pop() (substanced.folder.Folder method), 92
pop() (substanced.interfaces.IFolder method), 114

postorder() (in module substanced.util), 106
PropertySheet (class in substanced.property), 101, 102
put() (substanced.editable.IEditable method), 87
put() (substanced.interfaces.IEditable method), 113
Pyramid, **130**

R

RandomAutoNamingFolder (class in substanced.folder), 94
Reference Type, **130**
reference_source_property() (in module substanced.objectmap), 99
reference_sourceid_property() (in module substanced.objectmap), 99
reference_target_property() (in module substanced.objectmap), 100
reference_targetid_property() (in module substanced.objectmap), 100
ReferenceClass (class in substanced.interfaces), 122
ReferentialIntegrityError (class in substanced.objectmap), 101
refresh() (substanced.locking.Lock method), 96
register_editable_adapter() (in module substanced.editable), 88
reindex() (substanced.catalog.Catalog method), 76
reindex() (substanced.catalog.CatalogsService.Catalog method), 78
reindex() (substanced.interfaces.ICatalog method), 112
reindex_resource() (substanced.catalog.Catalog method), 77
reindex_resource() (substanced.catalog.CatalogsService.Catalog method), 79
reindex_resource() (substanced.interfaces.ICatalog method), 112
remove() (substanced.folder.Folder method), 92
remove() (substanced.interfaces.IFolder method), 117
remove() (substanced.interfaces.IObjectMap method), 118
remove() (substanced.objectmap.ObjectMap method), 98
removed_oids (substanced.event.ObjectWillBeRemoved attribute), 88
removed_oids (substanced.interfaces.IObjectRemoved attribute), 119
rename() (substanced.folder.Folder method), 93
rename() (substanced.interfaces.IFolder method), 114
renamer() (in module substanced.util), 106
reorder() (substanced.folder.Folder method), 93
reorder() (substanced.interfaces.IFolder method), 117
replace() (substanced.folder.Folder method), 93
replace() (substanced.interfaces.IFolder method), 115
request (substanced.interfaces.ILoggedIn attribute), 117
request (substanced.interfaces.IPropertySheet attribute), 120
reset() (substanced.catalog.Catalog method), 77
reset() (substanced.catalog.CatalogsService.Catalog method), 79
reset() (substanced.interfaces.ICatalog method), 112
reset() (substanced.interfaces.IWorkflow method), 122
reset() (substanced.workflow.Workflow method), 110
resolver (hypatia.interfaces.IResultSet attribute), 85
Resource, **130**
resource (substanced.locking.Lock attribute), 95
Resource factory, **130**
Resource tree, **130**
resourceid (substanced.locking.Lock attribute), 95
ResultSet (class in hypatia.util), 84
RIGHT (in module substanced.schema), 103
Root (class in substanced.root), 103
root_factory() (in module substanced.db), 87
RootAdded (class in substanced.event), 88

S

scan() (in module substanced), 75
Schema (class in substanced.schema), 102
schema (substanced.interfaces.IPropertySheet attribute), 120
schema_type (substanced.schema.Schema attribute), 102
SDI, **130**
sdi_title() (substanced.interfaces.ISDIAPI method), 121
SequentialAutoNamingFolder (class in substanced.folder), 94
Service, **130**
service (class in substanced.content), 85
set() (substanced.interfaces.IPropertySheet method), 120
set_acl() (in module substanced.util), 104
set_acl() (substanced.objectmap.ObjectMap method), 98
set_oid() (in module substanced.util), 104
set_order() (substanced.folder.Folder method), 93
set_order() (substanced.interfaces.IFolder method), 114
sort() (hypatia.interfaces.IResultSet method), 84
sort() (substanced.interfaces.IFolder method), 115
sourceids() (substanced.interfaces.IObjectMap method), 118
sourceids() (substanced.objectmap.ObjectMap method), 98
SourceIntegrityError (class in substanced.objectmap), 101
sources() (substanced.interfaces.IObjectMap method), 118
sources() (substanced.objectmap.ObjectMap method), 98
State, **130**
state_of() (substanced.interfaces.IWorkflow method), 122
state_of() (substanced.workflow.Workflow method), 110
States, **130**
statsd_gauge() (in module substanced.stats), 104
statsd_incr() (in module substanced.stats), 104
statsd_timer() (in module substanced.stats), 103

subscribe_acl_modified (class in substanced.event), 89
subscribe_added (class in substanced.event), 88
subscribe_after_transition (class in substanced.event), 89
subscribe_logged_in (class in substanced.event), 89
subscribe_modified (class in substanced.event), 89
subscribe_removed (class in substanced.event), 89
subscribe_root_added (class in substanced.event), 89
subscribe_will_be_added (class in substanced.event), 89
subscribe_will_be_removed (class in substanced.event), 89
substanced (module), 75
substanced.audit (module), 75
substanced.catalog (module), 75
substanced.catalog.indexes (module), 81
substanced.content (module), 85
substanced.db (module), 87
substanced.editable (module), 87
substanced.event (module), 88
substanced.evolution (module), 89
substanced.file (module), 90
substanced.folder (module), 91
substanced.form (module), 95
substanced.interfaces (module), 111
substanced.locking (module), 95
substanced.objectmap (module), 97
substanced.property (module), 101
substanced.root (module), 103
substanced.schema (module), 102
substanced.stats (module), 103
substanced.util (module), 104
substanced.workflow (module), 108

T

targetids() (substanced.interfaces.IObjectMap method), 118
targetids() (substanced.objectmap.ObjectMap method), 98
TargetIntegrityError (class in substanced.objectmap), 101
targets() (substanced.interfaces.IObjectMap method), 118
targets() (substanced.objectmap.ObjectMap method), 98
Text (class in substanced.catalog), 75
TextIndex (class in substanced.catalog.indexes), 81
transaction (substanced.catalog.Catalog attribute), 77
transaction (substanced.catalog.CatalogsService.Catalog attribute), 79
Transition, **130**
transition (substanced.interfaces.IAfterTransition attribute), 112
transition() (substanced.interfaces.IWorkflow method), 122
transition() (substanced.workflow.Workflow method), 110
transition_to_state() (substanced.interfaces.IWorkflow method), 122
transition_to_state() (substanced.workflow.Workflow method), 110
Transitions, **130**
typeof() (substanced.content.ContentRegistry method), 87

U

unindex_resource() (substanced.catalog.Catalog method), 77
unindex_resource() (substanced.catalog.CatalogsService.Catalog method), 79
unindex_resource() (substanced.interfaces.ICatalog method), 113
unlock_resource() (in module substanced.locking), 96
UnlockError (class in substanced.locking), 96
unset_order() (substanced.folder.Folder method), 93
unset_order() (substanced.interfaces.IFolder method), 116
update_indexes() (substanced.catalog.Catalog method), 77
update_indexes() (substanced.catalog.CatalogsService.Catalog method), 79
update_indexes() (substanced.interfaces.ICatalog method), 112
upload() (substanced.file.File method), 90
upload() (substanced.interfaces.IFile method), 113
USE_MAGIC (in module substanced.file), 90
user (substanced.interfaces.ILoggedIn attribute), 117
UserToLock (interface in substanced.interfaces), 122

V

validate_name() (substanced.folder.Folder method), 93
validate_name() (substanced.interfaces.IFolder method), 116
values() (substanced.folder.Folder method), 93
values() (substanced.interfaces.IFolder method), 117

W

Workflow, **130**
Workflow (class in substanced.workflow), 108
WorkflowError, 111
Workflows, **130**
WriteLock (interface in substanced.interfaces), 122

Z

Zope, **130**