django-widgy Documentation

Release 0.7.1

Programmers at Fusionbox

August 18, 2015

Contents

1 Design **3**
1.1 Data Model . . . 3
1.2 Versioning . . . 4
1.3 Customization . . . 4
1.4 Owners . . . 4
1.5 Editor . . . 5

2 Contrib Packages **7**
2.1 Page Builder . . . 7
2.2 Form Builder . . . 8
2.3 Widgy Mezzanine . . . 10
2.4 Review Queue . . . 11

3 API Reference **13**
3.1 Base Models . . . 13
3.2 Widgy Site . . . 17
3.3 Links Framework . . . 20
3.4 Template Tags . . . 21

4 Tutorials **23**
4.1 Quickstart . . . 23
4.2 Building Widgy's JavaScript With RequireJS . . . 27
4.3 Writing Your First Widget . . . 27
4.4 Proxy Widgy Model Tutorial . . . 30

5 Changelog **33**
5.1 0.7.1 (2015-08-18) . . . 33
5.2 0.7.0 (2015-07-31) . . . 33
5.3 0.6.1 (2015-05-01) . . . 33
5.4 0.6.0 (2015-04-30) . . . 34
5.5 0.5.0 (2015-04-17) . . . 34
5.6 0.4.0 (2015-03-12) . . . 34
5.7 0.3.5 (2015-01-30) . . . 35
5.8 0.3.4 (2015-01-22) . . . 35
5.9 0.3.3 (2014-12-22) . . . 35
5.10 0.3.2 (2014-10-16) . . . 35
5.11 0.3.1 (2014-10-01) . . . 35
5.12 0.3.0 (2014-09-24) . . . 36

5.13 0.2.0 (2014-08-04) . . . 37
5.14 0.1.6 (2014-09-09) . . . 38
5.15 0.1.5 (2013-11-23) . . . 38
5.16 0.1.4 (2013-11-04) . . . 38
5.17 0.1.3 (2013-10-25) . . . 38
5.18 0.1.2 (2013-10-23) . . . 38
5.19 0.1.1 (2013-10-21) . . . 39
5.20 0.1.0 (2013-10-18) . . . 39

6 Development **41**

django-widgy is a heterogeneous tree editor for Django that is well-suited for use as a CMS. A heterogeneous tree is a tree where each node can be a different type—just like HTML. Widgy provides the representation for heterogeneous trees as well as an interactive JavaScript editor for them. Widgy supports Django 1.4+.

Widgy was originally created for powerful content management, but it can have many different uses.

CHAPTER 1

Design

django-widgy is a heterogeneous tree editor for Django. It enables you to combine models of different types into a tree structure.

The django-widgy project is split into two main pieces. Widgy core provides `Node`, the `Content` abstract class, versioning models, views, configuration helpers, and the JavaScript editor code. Much like in Django, django-widgy has many contrib packages that provide the batteries.

1.1 Data Model

Central to Widgy are Nodes, Contents, and Widgets. `Node` is a subclass of Treebeard's `MP_Node`. Nodes concern themselves with the tree structure. Each Node is associated with an instance of `Content` subclass. A Node + Content combination is called a Widget.

Storing all the structure data in Node and having that point to any subclass of Content allows us to have all the benefits of a tree, but also the flexibility to store very different data within a tree.

`Nodes` are associated with their `Content` through a `GenericForeignKey`.

This is what a hypothetical Widgy tree might look like.:

```
Node (TwoColumnLayout)
|
+-- Node (MainBucket)
|    |
|    +-- Node (Text)
|    |
|    +-- Node (Image)
|    |
|    +-- Node (Form)
|         |
|         +-- Node (Input)
|         |
|         +-- Node (Checkboxes)
|         |
|         +-- Node (SubmitButton)
|
+-- Node (SidebarBucket)
     |
     +-- Node (CallToAction)
```

1.2 Versioning

Widgy comes with an optional but powerful versioning system inspired by Git. Versioning works by putting another model called a version tracker between the owner and the root node. Just like in Git, each `VersionTracker` has a reference to a current working copy and then a list of commits. A `VersionCommit` is a frozen snapshot of the tree.

Versioning also supports delayed publishing of commits. Normally commits will be visible immediately, but it is possible to set a publish time for a commit that allows a user to future publish content.

To enable versioning, all you need to do is use `widgy.db.fields.VersionedWidgyfield` instead of `widgy.db.fields.WidgyField`.

Todo

diagram

1.3 Customization

There are two main ways to customize the behavior of Widgy and existing widgets. The first is through the *`WidgySite`*. *`WidgySite`* is a centralized source of configuration for a Widgy instance, much like Django's `AdminSite`. You can also configure each widget's behavior by subclassing it with a proxy.

1.3.1 WidgySite

- tracks installed widgets
- stores URLs
- provides authorization
- allows centralized overriding of compatibility between components
- accomodates for multiple instances of widgy

1.3.2 Proxying a Widget

Widgy uses a special subclass of `GenericForeignKey` that supports retrieving proxy models. Subclassing a model as a proxy is a lightweight method for providing custom behavior for widgets that you don't control. A more in-depth tutorial on proxying widgets can be found at the Proxy Widgy Model Tutorial.

1.4 Owners

A Widgy owner is a model that has a `WidgyField`.

1.4.1 Admin

Use `WidgyAdmin` (or a `WidgyForm` for your admin form)

Use `get_action_links` to add a preview button to the editor.

1.4.2 Page Builder

If layouts should extend something other than `layout_base.html`, set the `base_template` property on your owner.

1.4.3 Form Builder

`widgy.contrib.form_builder` requires a `get_form_action` method on the owner. It accepts the form widget and the Widgy context, and returns a URL for forms to submit to. You normally submit to your own view and mix in `HandleFormMixin` to help with handling the form submission. Make sure re-rendering after a validation error works.

Todo

tutorials/owner

1.4.4 Tutorial

- It's probably a good idea to render the entire page through Widgy, so I've used a template like this:

  ```
  {# product_list.html #}

  {% include widgy_tags %}{% render_root category 'content' %}
  ```

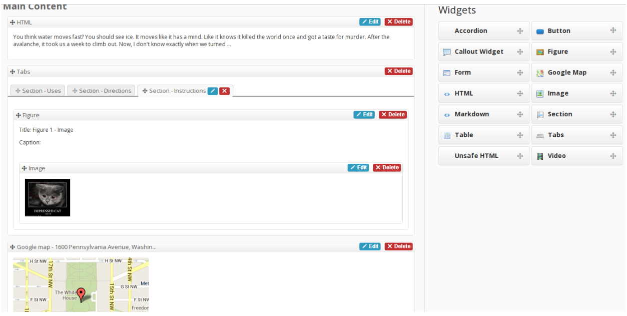
  I have been inserting the 'view' style functionality, in this case a list of products in a category, with `ProductList` widget.

You'll probably have to add support for `root_node_override` to your view, like this:

```
root_node_pk = self.kwargs.get('root_node_pk')
if root_node_pk:
    site.authorize_view(self.request, self)
    kwargs['root_node_override'] = get_object_or_404(Node, pk=root_node_pk)
elif hasattr(self, 'form_node'):
    kwargs['root_node_override'] = self.form_node.get_root()
```

1.5 Editor

Widgy provides a drag and drop JavaScript editor interface to the tree in the form of a Django formfield.

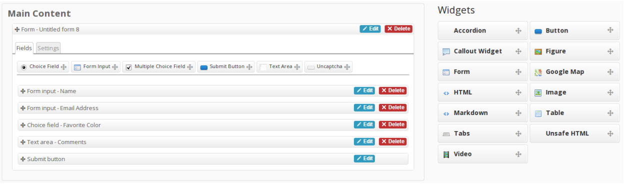


You think water moves fast? You should see ice. It moves like it has a mind. Like it knows it killed the world once and got a taste for murder. After the avalanche, it took us a week to climb out. Now, I don't know exactly when we turned ...

Title: Figure 1 - Image

Caption:

The editor is built on Backbone.js and RequireJS to provide a modular and customizable interface.


Main Content
Form - Untitled form 8
Edit
Delete
Fields
Settings
Choice Field
Form Input
Multiple Choice Field
Submit Button
Text Area
Uncaptcha
Form input - Name
Edit
Delete
Form input - Email Address
Edit
Delete
Choice field - Favorite Color
Edit
Delete
Text area - Comments
Edit
Delete
Submit button
Edit
Widgets
Accordion
Button
Callout Widget
Figure
Form
Google Map
HTML
Image
Markdown
Table
Tabs
Unsafe HTML
Video

CHAPTER 2

Contrib Packages

Here is where we keep the batteries. These packages are Django apps that add functionality or widgets to Widgy.

2.1 Page Builder

Page builder is a collection of widgets for the purpose of creating HTML pages.

2.1.1 Installation

Page builder depends on the following packages:

- django-filer
- markdown
- bleach
- sorl-thumbnail

You can install them manually, or you can install them using the django-widgy package:

```
$ pip install django-widgy[page_builder]
```

2.1.2 Widgets

class `widgy.contrib.page_builder.models.`**`DefaultLayout`**

> **Todo**
>
> Who actually uses DefaultLayout?

class `widgy.contrib.page_builder.models.`**`MainContent`**

class `widgy.contrib.page_builder.models.`**`Sidebar`**

class `widgy.contrib.page_builder.models.`**`Markdown`**

class `widgy.contrib.page_builder.models.`**`Html`**

The HTML Widget provides a CKEditor field. It is useful for large blocks of text that need simple inline styling.

It purposefully doesn't have the capability to add images or tables, because there are already widgets that the developer can control.

Note: There is a possible permission escalation vulnerability with allowing any admin user to add HTML. For this reason, the *Html* widget sanitizes all the HTML using bleach. If you want to add unsanitized HTML, please use the *UnsafeHtml* widget.

class `widgy.contrib.page_builder.models.`**`UnsafeHtml`**

This is a widget which allows the user to output arbitrary HTML. It is unsafe because a non-superuser could gain publishing the unsafe HTML on the website with XSS code to cause permission escalation.

Warning: The `page_builder.add_unsafehtml` and `page_builder.edit_unsafehtml` permissions are equivalent to `is_superuser` status because of the possibility of a staff user inserting JavaScript that a superuser will execute.

class `widgy.contrib.page_builder.models.`**`CalloutWidget`**

class `widgy.contrib.page_builder.models.`**`Accordion`**

class `widgy.contrib.page_builder.models.`**`Tabs`**

class `widgy.contrib.page_builder.models.`**`Section`**

class `widgy.contrib.page_builder.models.`**`Image`**

class `widgy.contrib.page_builder.models.`**`Video`**

class `widgy.contrib.page_builder.models.`**`Figure`**

class `widgy.contrib.page_builder.models.`**`GoogleMap`**

class `widgy.contrib.page_builder.models.`**`Button`**

2.1.3 Tables

class `widgy.contrib.page_builder.models.`**`Table`**

class `widgy.contrib.page_builder.models.`**`TableRow`**

class `widgy.contrib.page_builder.models.`**`TableHeaderData`**

class `widgy.contrib.page_builder.models.`**`TableData`**

2.1.4 Database Fields

class `widgy.contrib.page_builder.db.fields.`**`ImageField`**

A FilerFileField that only accepts images. Includes sensible defaults for use in Widgy — `null=True`, `related_name='+'` and `on_delete=PROTECT`.

2.2 Form Builder

Form builder is a collection of tools built on top of Page Builder that help with the creation of HTML forms.

To enable Form Builder, add `widgy.contrib.form_builder` to your `INSTALLED_APPS`.

2.2.1 Installation

Form builder depends on the following packages:

- django-widgy[page_builder]
- django-extensions
- html2text
- phonenumbers

You can install them manually, or you can install them using the django-widgy package:

```
$ pip install django-widgy[page_builder,form_builder]
```

2.2.2 Success Handlers

When a user submits a *Form*, the *Form* will loop through all of the success handler widgets to do the things that you would normally put in the `form_valid` method of a `django.views.generic.FormView`, for example. Form Builder provides a couple of built-in success handlers that do things like saving the data, sending emails, or submitting to Salesforce.

2.2.3 Widgets

class `widgy.contrib.form_builder.models.`**`Form`**

This widget corresponds to the HTML `<form>` tag. It acts as a container and also can be used to construct a Django Form class.

`build_form_class`(*self*)

Returns a Django Form class based on the FormField widgets inside the form.

class `widgy.contrib.form_builder.models.`**`Uncaptcha`**

class `widgy.contrib.form_builder.models.`**`FormField`**

FormField is an abstract base class for the following widgets. Each *FormField* has the following fields which correspond to the same attributes on `django.forms.fields.Field`.

`label`

Corresponds with the HTML `<label>` tag. This is the text that will go inside the label.

`required`

Indicates whether or not this field is required. Defaults to True.

`help_text`

A TextField for outputting help text.

class `widgy.contrib.form_builder.models.`**`FormInput`**

This is a widget for all simple `<input>` types. It supports the following input types: `text`, `number`, `email`, `tel`, `checkbox`, `date`. Respectively they correspond with the following Django formfields: `CharField`, `IntegerField`, `EmailField`, `PhoneNumberField`, `BooleanField`, `DateField`.

class `widgy.contrib.form_builder.models.`**`Textarea`**

class `widgy.contrib.form_builder.models.`**`ChoiceField`**

class `widgy.contrib.form_builder.models.`**`MultipleChoiceField`**

2.2.4 Owner Contract

For custom *Widgy owners*, Form Builder needs to have a view to use for handling form submissions.

1. Each widgy owner should implement a `get_form_action_url(form, widgy_context)` method that returns a URL that points to a view (see step 2).
2. Create a view to handle form submissions for each owner. Form Builder provides the class-based views mixin, *`HandleFormMixin`*, to make this easier.

2.2.5 Views

class `widgy.contrib.form_builder.views.`**`HandleFormMixin`**

An abstract view mixin for handling form_builder.Form submissions. It inherits from `django.views.generic.edit.FormMixin`.

It should be registered with a URL similar to the following.

```
url('^form/(?P<form_node_pk>[^/]*)/$', 'your_view')
```

`HandleFormMixin` does not implement a GET method, so your subclass should handle that. Here is an example of a fully functioning implementation:

```
from django.views.generic import DetailView
from widgy.contrib.form_builder.views import HandleFormMixin

class EventDetailView(HandleFormMixin, DetailView):
    model = Event

    def post(self, *args, **kwargs):
        self.object = self.get_object()
        return super(EventDetailView, self).post(*args, **kwargs)
```

`widgy.contrib.widgy_mezzanine.views.HandleFormView` provides an even more robust example implementation.

2.3 Widgy Mezzanine

This app provides integration with the Mezzanine project. Widgy Mezzanine uses Mezzanine for site structure and Widgy for page content. It does this by providing a subclass of Mezzanine's Page model called *`WidgyPage`* which delegates to Page Builder for all content.

The dependencies for Widgy Mezzanine (Mezzanine and Widgy's Page Builder app) are not installed by default when you install widgy, you can install them yourself:

```
$ pip install Mezzanine django-widgy[page_builder]
```

or you can install them using through the widgy package:

```
$ pip install django-widgy[page_builder,widgy_mezzanine]
```

In order to use Widgy Mezzanine, you must provide `WIDGY_MEZZANINE_SITE` in your settings. This is a fully-qualified import path to an instance of *`WidgySite`*. You also need to install the URLs.

```
url(r'^widgy-mezzanine/', include('widgy.contrib.widgy_mezzanine.urls')),
```

class `widgy.contrib.widgy_mezzanine.models.`**`WidgyPage`**

The `WidgyPage` class is swappable like `User`. If you want to override it, specify a `WIDGY_MEZZANINE_PAGE_MODEL` in your settings. the `widgy.contrib.widgy_mezzanine.models.WidgyPageMixin` mixin is provided for ease of overriding. Any code that references a *`WidgyPage`* should use the `widgy.contrib.widgy_mezzanine.get_widgypage_model()` to get the correct class.

2.4 Review Queue

Some companies have stricter policies for who can edit and who can publish content on their websites. The review queue app is an extension to versioning which collects commits for approval by a user with permissions.

The `review_queue.change_reviewedversioncommit` permission is used to determine who is allowed to approve commits.

To enabled the review queue,

1. Add `widgy.contrib.review_queue` to your `INSTALLED_APPS`.
2. Your *`WidgySite`* needs to inherit from `ReviewedWidgySite`.
3. Register a subclass of *`VersionCommitAdminBase`*.

```
from django.contrib import admin
from widgy.contrib.review_queue.admin import VersionCommitAdminBase
from widgy.contrib.review_queue.models import ReviewedVersionCommit

class VersionCommitAdmin(VersionCommitAdminBase):
    def get_site(self):
        return my_site

admin.site.register(ReviewedVersionCommit, VersionCommitAdmin)
```

4. If upgrading from a non-reviewed site, a `widgy.contrib.review_queue.models.ReviewedVersionCommit` object must be created for each `widgy.models.VersionCommit`. There is a management command to do this for you. It assumes that all existing commits should be approved.

```
./manage.py populate_review_queue
```

class `admin.`**`VersionCommitAdminBase`**

This an abstract `ModelAdmin` class that displays the pending changes for approval. Any it abstract, because it doesn't know which *`WidgySite`* to use.

`get_site`(*self*)

The *`WidgySite`* that this specific *`VersionCommitAdminBase`* needs to work on.

Note: The review queue's undo (it can undo approvals) support requires Django >= 1.5 or the session-based `MESSAGE_STORAGE`:

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'
```

CHAPTER 3

API Reference

3.1 Base Models

class widgy.models.base.**Content**

node

Accessor for the *Node* that the *Content* belongs to.

Tree Traversal

With the exception *depth_first_order()*, the following methods are all like the traversal API provided by Treebeard, but instead of returning *Nodes*, they return *Contents*.

get_root (*self*)

get_ancestors (*self*)

get_parent (*self*)

get_next_sibling (*self*)

get_children (*self*)

depth_first_order (*self*)

Convenience method for iterating over all the *Contents* in a subtree in order. This is similar to Treebeard's get_descendants(), but includes itself.

Tree Manipulation

The following methods mirror those of *Node*, but accept a *WidgySite* as the first argument. You must call these methods on *Content* and not on *Node*.

```
>>> root = Layout.add_root(widgy_site)
>>> main = root.add_child(widgy_site, MainContent)
>>> sidebar = main.add_sibling(widgy_site, Sidebar, title='Alerts')
# move the sidebar to the left of the main content.
>>> sidebar.reposition(widgy_site, right=main)
```

classmethod add_root (*cls*, *site*, ***kwargs*)

Creates a root node widget. Any kwargs will be passed to the Content class's initialize method.

add_child(*self*, *site*, *cls*, ***kwargs*)
Adds a new instance of `cls` as the last child of the current widget.

add_sibling(*self*, *site*, *cls*, ***kwargs*)
Adds a new instance of `cls` to the right of the current widget.

reposition(*self*, *site*, *right=None*, *parent=None*)
Moves the current widget to the left of `right` or to the last child position of `parent`.

post_create(*self*, *site*)
Hook for doing things after a widget has been created (a `Content` has been created and put in the tree). This is useful if you want to have default children for a widget, for example.

delete(*self*, *raw=False*)
If `raw` is `True` the widget is being deleted due to a failure in widget creation, so `post_create` will not have been run yet.

clone(*self*)
This method is called by `Node.clone_tree()`. You may wish to override it if your Content has special needs like a ManyToManyField.

> **Warning:** Clone is used to freeze tree state in Versioning. If your `clone()` method is incorrect, your history will be corrupt.

Editing

display_name
A human-readable short name for widgets. This defaults to the `verbose_name` of the widget.

> **Hint:** You can use the `@property` decorator to make this dynamic.

> **Todo**
>
> screenshot

tooltip
A class attribute that sets the tooltip for this widget on the shelf.

css_classes
A list of CSS classes to apply to the widget element in the Editor. Defaults to `app_label` and `module_name` of the widget.

shelf = False
Denotes whether this widget have a shelf. Root nodes automatically have a shelf. The shelf is where the widgets exist in the interface before they are dragged on. It is useful to set `shelf` to `True` if there are a large number of widgets who can only go in a specfic subtree.

component_name = 'widget'
Specifies which JavaScript component to use for this widget.

> **Todo**
>
> Write documentation about components.

pop_out = CANNOT_POP_OUT
It is possible to open a subtree in its own editing window. `pop_out` controls if a widget can be popped out. There are three values for `pop_out`:

CANNOT_POP_OUT

CAN_POP_OUT

MUST_POP_OUT

form = ModelForm
The form class to use for editing. Also see *get_form_class()*.

formfield_overrides = {}
Similar to `ModelAdmin`, *Content* allows you to override the form fields for specific model field classes.

draggable = True
Denotes whether this widget may be moved through the editing interface.

deletable = True
Denotes whether this widget may be deleted through the editing interface.

editable = False
Denotes whether this widget may be edited through the editing interface. Widgy will automatically generate a `ModelForm` to provide the editing functionality. Also see `form` and *get_form_class()*.

preview_templates
A template name or list of template names for rendering in the widgy Editor. See *get_templates_hierarchy()* for how the default value is derived.

edit_templates
A template name or list of template names for rendering the edit interface in the widgy Editor. See *get_templates_hierarchy()* for how the default value is derived.

get_form_class (*self*, *request*)
Returns a `ModelForm` class that is used for editing.

get_form (*self*, *request*, ***form_kwargs*)
Returns a form instance to use for editing.

classmethod get_templates_hierarchy (*cls*, ***kwargs*)
Loops through MRO to return a list of possible template names for a widget. For example the preview template for something like *Tabs* might look like:

- `widgy/page_builder/tabs/preview.html`
- `widgy/mixins/tabbed/preview.html`
- `widgy/page_builder/accordion/preview.html`
- `widgy/page_builder/bucket/preview.html`
- `widgy/models/content/preview.html`
- `widgy/page_builder/preview.html`
- `widgy/mixins/preview.html`
- `widgy/page_builder/preview.html`
- `widgy/models/preview.html`
- `widgy/preview.html`

Frontend Rendering

render (*self*, *context*, *template=None*)

The method that is called by the *render()* template tag to render the Content. It is useful to override this if you need to inject things into the context.

get_render_templates (*self*, *context*)

Returns a template name or list of template names for frontend rendering.

Compatibility

Widgy provide robust machinery for compatibility between Contents. Widgy uses the compatibility system to validate the relationships between parent and child Contents.

Compatibility is checked when rendering the shelf and when adding or moving widgets in the tree.

accepting_children = False

An easy compatibility configuration attribute. See *valid_parent_of()* for more details.

valid_parent_of (*self*, *cls*, *obj=None*)

If `obj` is provided, return `True` if it could be a child of the current widget. `cls` is the type of `obj`.

If `obj` isn't provided, return `True` if a new instance of `cls` could be a child of the current widget.

`obj` is `None` when the child widget is being created or Widgy is checking the compatibility of the widgets on the shelf. If it is being moved from another location, there will be an instance. A parent and child are only compatible if both *valid_parent_of()* and *valid_child_of()* return `True`. This defaults to the value of `accepting_children`.

Here is an example of a parent that only accepts three instances of `B`:

```
class A(Content):
    def valid_parent_of(self, cls, obj=None):
        # If this is already my child, it can stay my child.
        # This works for obj=None because self.get_children()
        # will never contain None.
        if obj in self.get_children():
            return True
        else:
            # Make sure it is of type B
            return (issubclass(cls, B)
            # And that I don't already have three children.
                and len(self.get_children()) < 3)
```

classmethod **valid_child_of** (*cls*, *parent*, *obj=None*)

If `obj` is provided, return `True` if it can be a child of `parent`. `obj` will be an instance of `cls`—it may feel like an instance method.

If `obj` isn't provided, return `True` if a new instance of `cls` could be a child of `parent`.

This defaults to `True`.

Here is an example of a Content that can not live inside another instance of itself:

```
class Foo(Content):
    @classmethod
    def valid_child_of(cls, parent, obj=None):
        for p in list(parent.get_ancestors()) + [parent]:
            if isinstance(p, Foo):
```

```
                        return False
                return super(Foo, cls).valid_child_of(parent, obj)
```

equal (*self*, *other*)
Should return `True` if `self` is equal to `other`. The default implementation checks the equality of each widget's `get_attributes()`.

class `widgy.models.base.`**Node**

content
A generic foreign key point to our *`Content`* instance.

is_frozen
A boolean field indicating whether this node is frozen and can't be changed in any way. This is used to preserve old tree versions for versioning.

render (*self*, **args*, ***kwargs*)
Renders this subtree and returns a string. Normally you shouldn't call it directly, use `widgy.db.fields.WidgyField.render()` or *`widgy.templatetags.widgy_tags.render()`*.

depth_first_order (*self*)
Like *`Content.depth_first_order()`*, but over nodes.

prefetch_tree (*self*)
Efficiently fetches an entire tree (or subtree), including content instances. It uses `1 + m` queries, where `m` is the number of distinct content types in the tree.

classmethod **prefetch_trees** (*cls*, **root_nodes*)
Prefetches multiple trees. Uses `n + m` queries, where `n` is the number of trees and `m` is the number of distinct content types across *all* the trees.

maybe_prefetch_tree (*self*)
Prefetches the tree unless it has been prefetched already.

classmethod **find_widgy_problems** (*cls*, *site=None*)
When a Widgy tree is edited without protection from a transaction, it is possible to get into an inconsistent state. This method returns a tuple containing two lists:

1. A list of node pks whose *content* pointer is dangling – pointing to a content that doesn't exist.
2. A list of node pks whose *content_type* doesn't exist. This might happen when you switch branches and remove the code for a widget, but still have the widget in your database. These are represented by `UnknownWidget` instances.

3.2 Widgy Site

class `widgy.site.`**WidgySite**

get_all_content_classes (*self*)

Returns a list (or set) of available Content classes (widget classes). This is used

- To find layouts from `root_choices`
- To find widgets to put on the shelf (using *`validate_relationship()`* against all existing widgets in a tree)

urls(*self*)

Returns the urlpatterns needed for this Widgy site. It should be included in your urlpatterns:

```
('^admin/widgy/', include(widgy_site.urls)),
```

get_urls(*self*)

This method only exists due to the example `ModelAdmin` sets.

Todo

is `urls` or `get_urls` the preferred interface?

reverse(*self*, **args*, ***kwargs*)

Todo

explain reverse

authorize_view(*self*, *request*, *view*)

Every Widgy view will call this before doing anything. It can be considered a 'view' or 'read' permission. It should raise a `PermissionDenied` when the request is not authorized. It can be used to implement permission checking that should happen on every view, like limiting access to staff members:

```
def authorize_view(self, request, view):
    if not request.user.is_staff:
        raise PermissionDenied
    super(WidgySite, self).authorize_view(request, value)
```

has_add_permission(*self*, *request*, *content_class*)

Given a `Content` class, can this request add a new instance? Returns `True` or `False`. The default implementation uses the Django Permission framework.

has_change_permission(*self*, *request*, *obj_or_class*)

Like *`has_add_permission()`*, but for changing. It receives an instance if one is available, otherwise a class.

has_delete_permission(*self*, *request*, *obj_or_class_or_list*)

Like *`has_change_permission()`*, but for deleting. `obj_or_class_or_list` can also be a list, when attempting to delete a widget that has children.

validate_relationship(*self*, *parent*, *child*)

The single compatibility checking entry point. The default implementation delegates to *`valid_parent_of()`* of *`valid_child_of()`*.

`parent` is always an instance, `child` can be a class or an instance.

valid_parent_of(*self*, *parent*, *child_class*, *child=None*)

Does `parent` accept the `child` instance, or a new `child_class` instance, as a child?

The default implementation just delegates to `Content.valid_parent_of`.

valid_child_of(*self*, *parent*, *child_class*, *child=None*)

Will the `child` instance, or a new instance of `child_class`, accept `parent` as a parent?

The default implementation just delegates to `Content.valid_child_of`.

get_version_tracker_model(*self*)

Returns the class to use as a `VersionTracker`. This can be overridden to customize versioning behavior.

Views

Each of these properties returns a view callable. A urlpattern is built in `get_urls()`. It is important that the same callable is used for the lifetime of the site, so `django.utils.functional.cached_property` is helpful.

node_view(*self*)

content_view(*self*)

shelf_view(*self*)

node_edit_view(*self*)

node_templates_view(*self*)

node_parents_view(*self*)

commit_view(*self*)

history_view(*self*)

revert_view(*self*)

diff_view(*self*)

reset_view(*self*)

Media Files

Note: These properties are cached at server start-up, so new ones won't be detected until the server restarts. This means that when using `runserver`, you have to manually restart the server when adding a new file.

scss_files

Returns a list of SCSS files to be included on the front-end. Widgets can add SCSS files just by making a file available at a location determined by its app label and name (see `widgy.models.Content.get_templates_hierarchy()`). For example:

```
widgy/page_builder/html.scss
```

js_files

Like `scss_files`, but JavaScript files.

admin_scss_files

Like `scss_files`, but for the back-end editing interface. These paths look like, for an app:

```
widgy/page_builder/admin.scss
```

and for a widget:

```
widgy/page_builder/table.admin.scss
```

If you want to included JavaScript for the editing interface, you should use a `component`.

3.3 Links Framework

Widgy core also provides a linking framework that allows any model to point to any other model without really knowing which models are available for linking. This is the mechanism by which Page Builder's *Button* can link to Widgy Mezzanine's *WidgyPage* without even knowing that `WidgyPage` exists. There are two components to the links framework, *LinkField* and the link registry.

3.3.1 Model Field

class `widgy.models.links.`**`LinkField`**

LinkField is a subclass of `django.contrib.contenttypes.generic.GenericForeignKey`. If you want to add a link to any model, you can just add a *LinkField* to it.

```
from django.db import models
from widgy.models import links

class MyModel(models.Model):
    title = models.Charfield(max_length=255)
    link = links.LinkField()
```

LinkField will automatically add the two required fields for GenericForeignKey, the `ContentType` ForeignKey and the PositiveIntegerField. If you need to override this, you can pass in the `ct_field` and `fk_field` options that GenericForeignKey takes.

Note: Unfortunately, because Django currently lacks support for composite fields, if you need to display the *LinkField* in a form, there are a couple of things you need to do.

1. Your Form class needs to mixin the `LinkFormMixin`.
2. You need to explicitly define a `LinkFormField` on your Form class.

Hopefully in future iterations of Django, these steps will be obsoleted.

3.3.2 Registry

If you want to expose your model to the link framework to allow things to link to it, you need to do two things.

1. You need to register your model with the links registry.

```
from django.db import models
from widgy.models import links

class Blog(models.Model):
    # ...

links.register(Blog)
```

The `register()` function also works as a class decorator.

```
from django.db import models
from widgy.models import links

@links.register
class Blog(models.Model):
    # ...
```

2. You need to make sure that your model defines a `get_absolute_url` method.

3.4 Template Tags

To use these, you'll need to `{% load widgy_tags %}`.

`widgy.templatetags.widgy_tags.`**`render`**`(`*node*`)`

Renders a node. Use this in your `render.html` templates to render any node that isn't a root node. Under the hood, this template tag calls `Content.render` with the current context.

Example:

```
{% for child in self.get_children %}
  {% render child %}
{% endfor %}
```

`widgy.templatetags.widgy_tags.`**`scss_files`**`(`*site*`)`

`widgy.templatetags.widgy_tags.`**`js_files`**`(`*site*`)`

These template tags provide a way to extract the *`WidgySite.scss_files`* off of a site. This is required if you don't have access to the site in the context, but do have a reference to it in your settings file. *`scss_files()`* and *`js_files()`* can also accept an import path to the site.

```
{% for js_file in 'WIDGY_MEZZANINE_SITE'|js_files %}
  <script src="{% static js_file %}"></script>
{% endfor %}
```

`widgy.templatetags.widgy_tags.`**`render_root`**`(`*owner*, *field_name*`)`

The template entry point for rendering a tree. It delegates to `WidgyField.render`. The `root_node_override` template context variable can be used to override the root node that is rendered (for preview).

```
{% render_root owner_obj 'content' %}
```

CHAPTER 4

Tutorials

4.1 Quickstart

This quickstart assumes you wish to use the following packages:

- Widgy Mezzanine
- Page Builder
- Form Builder

Install the Widgy package:

```
pip install django-widgy[all]
```

Add Mezzanine apps to `INSTALLED_APPS` in `settings.py`:

```
'mezzanine.conf',
'mezzanine.core',
'mezzanine.generic',
'mezzanine.pages',
'django_comments',
'django.contrib.sites',
'filebrowser_safe',
'grappelli_safe',
```

add Widgy to `INSTALLED_APPS`:

```
'widgy',
'widgy.contrib.page_builder',
'widgy.contrib.form_builder',
'widgy.contrib.widgy_mezzanine',
```

add required Widgy apps to `INSTALLED_APPS`:

```
'filer',
'easy_thumbnails',
'compressor',
'argonauts',
'sorl.thumbnail',
```

`django.contrib.admin` should be installed after Mezzanine and Widgy, so move it under them in `INSTALLED_APPS`.

add Mezzanine middleware to `MIDDLEWARE_CLASSES`:

```
'mezzanine.core.request.CurrentRequestMiddleware',
'mezzanine.core.middleware.AdminLoginInterfaceSelectorMiddleware',
'mezzanine.pages.middleware.PageMiddleware',
```

Mezzanine settings:

```
# settings.py
PACKAGE_NAME_FILEBROWSER = "filebrowser_safe"
PACKAGE_NAME_GRAPPELLI = "grappelli_safe"
ADMIN_MEDIA_PREFIX = STATIC_URL + "grappelli/"
TESTING = False
GRAPPELLI_INSTALLED = True
SITE_ID = 1
```

If you want mezzanine to use *`WidgyPage`* as the default page, you can add the following line to `settings.py`:

```
ADD_PAGE_ORDER = (
    'widgy_mezzanine.WidgyPage',
)
```

add Mezzanine's context processors. If you don't already have `TEMPLATE_CONTEXT_PROCESSORS` in your settings file, you should copy the default before adding Mezzanine's:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    # Defaults
    "django.contrib.auth.context_processors.auth",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.static",
    "django.core.context_processors.media",
    "django.core.context_processors.request",
    # Mezzanine
    "mezzanine.conf.context_processors.settings",
    "mezzanine.pages.context_processors.page",
)
```

make a *`Widgy site`* and set it in settings:

```
# demo/widgy_site.py
from widgy.site import WidgySite

class WidgySite(WidgySite):
    pass

site = WidgySite()

# settings.py
WIDGY_MEZZANINE_SITE = 'demo.widgy_site.site'
```

Configure django-compressor:

```
# settings.py
STATICFILES_FINDERS = (
    'compressor.finders.CompressorFinder',
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
)

COMPRESS_ENABLED = True
```

```
COMPRESS_PRECOMPILERS = (
    ('text/x-scss', 'django_pyscss.compressor.DjangoScssFilter'),
)
```

Note: Widgy requires that django-compressor be configured with a precompiler for `text/x-scss`. Widgy uses the django-pyscss package for easily integrating the pySscs library with Django.

Note: If you are using a version of Django older than 1.7, you will need use South 1.0 or set SOUTH_MIGRATION_MODULES.

Then run the following command:

```
$ python manage.py migrate
```

Note: If you are on a version of Django older than 1.7, you will need to run the following command as well:

```
$ python manage.py syncdb
```

add urls:

```
# urls.py
from django.conf.urls import patterns, include, url
from demo.widgy_site import site as widgy_site

urlpatterns = patterns('',
    # ...
    # widgy admin
    url(r'^admin/widgy/', include(widgy_site.urls)),
    # widgy frontend
    url(r'^widgy/', include('widgy.contrib.widgy_mezzanine.urls')),
    url(r'^', include('mezzanine.urls')),
)
```

Make sure you have a url pattern named `home` or the admin templates will not work right.

If you are using `GenericTemplateFinderMiddleware`, use the one from `fusionbox.mezzanine.middleware`. It has been patched to work with Mezzanine.

4.1.1 How to edit home page

1. Add the homepage to your urls.py:

   ```
   url(r'^$', 'mezzanine.pages.views.page', {'slug': '/'}, name='home'),
   ```

   Note: it must be a named URL, with the name 'home'

2. Make a page with the slug `/` and publish it.

3. Make a template called `pages/index.html` and put:

   ```
   {% extends "pages/widgypage.html" %}
   ```

   Note: If you don't do this you will likely get the following error:

   ```
   AttributeError: 'Settings' object has no attribute 'FORMS_EXTRA_FIELDS'
   ```

This is caused by Mezzanine falling back its own template pages/index.html which tries to provide the inline editing feature, which requires mezzanine.forms to be installed.

4.1.2 Admin center

A nice ADMIN_MENU_ORDER:

```
# settings.py
ADMIN_MENU_ORDER = [
    ('Widgy', (
        'pages.Page',
        'page_builder.Callout',
        'form_builder.Form',
        ('Review queue', 'review_queue.ReviewedVersionCommit'),
        ('File manager', 'filer.Folder'),
    )),
]
```

4.1.3 urlconf include

urlconf_include is an optional application that allows you to install urlpatterns in the Mezzanine page tree. To use it, put it in INSTALLED_APPS,:

```
'widgy.contrib.urlconf_include',
```

then add the urlconf_include middleware,:

```
'widgy.contrib.urlconf_include.middleware.PatchUrlconfMiddleware',
```

then set URLCONF_INCLUDE_CHOICES to a list of allowed urlpatterns. For example:

```
URLCONF_INCLUDE_CHOICES = (
    ('blog.urls', 'Blog'),
)
```

4.1.4 Adding Widgy to Mezzanine

If you are adding widgy to an existing mezzanine site, there are some additional considerations.

If you have not done so already, add the root directory of your mezzanine install to INSTALLED_APPS.

Also, take care when setting the WIDGY_MEZZANINE_SITE variable in your settings.py file. Because mezzanine is using an old Django directory structure, it uses your root directory as your project file:

```
# Use:
WIDGY_MEZZANINE_SITE = 'myproject.demo.widgy_site.site'
# Not:
WIDGY_MEZZANINE_SITE = 'demo.widgy_site.site'
```

4.1.5 Common Customizations

If you only have *WidgyPages*, you can choose to unregister the Mezzanine provided RichTextPage. Simply add this to an admin.py file in your directory and add this code:

```
from django.contrib import admin

from mezzanine.pages.models import RichTextPage

admin.site.unregister(RichTextPage)
```

4.2 Building Widgy's JavaScript With RequireJS

Widgy's editing interface uses RequireJS to handle dependency management and to encourage code modularity. This is convenient for development, but might be slow in production due to the many small JavaScript files. Widgy supports building its JavaScript with the RequireJS optimizer to remedy this. This is entirely optional and only necessary if the performance of loading many small JavaScript and template files bothers you.

To build the JavaScript,

- Install `django-require`:

```
pip install django-require
```

- Add the settings for django-require:

```
REQUIRE_BUILD_PROFILE = 'widgy.build.js'
REQUIRE_BASE_URL = 'widgy/js'
STATICFILES_STORAGE = 'require.storage.OptimizedStaticFilesStorage'
```

- Install `node` or `rhino` to run `r.js`. django-require will detect which one you installed. `rhino` is nice because you can apt-get it:

```
apt-get install rhino
```

Now the JavaScript will automatically built during `collectstatic`.

4.3 Writing Your First Widget

In this tutorial, we will build a Slideshow widget. You probably want to read the Quickstart to get a Widgy site running before you do this one.

We currently have a static slideshow that we need to make editable. Users need to be able to add any number of slides. Users also want to be able to change the delay between each slide.

Here is the current slideshow HTML that is using jQuery Cycle2:

```
<div class="cycle-slideshow"
     data-cycle-timeout="2000"
     data-cycle-caption-template="{% templatetag openvariable %}alt{% templatetag closevariable %}">
  <div class="cycle-caption"></div>

  <img src="http://placekitten.com/800/300" alt="Cute cat">
  <img src="http://placekitten.com/800/300" alt="Fuzzy kitty">
  <img src="http://placekitten.com/800/300" alt="Another cute cat">
  <img src="http://placekitten.com/800/300" alt="Awwww">
</div>
```

See also:

`templatetag` This template tag allows inserting the `{{` and `}}` characters needed by Cycle2.

4.3.1 1. Write the Models

The first step in writing a widget is to write the models. We are going to make a new Django app for these widgets.

```
$ python manage.py startapp slideshow
```

(Don't forget to add slideshow to your INSTALLED_APPS).

Now let's write the models. We need to make a Slideshow model as the container and a Slide model that represents the individual images.

```
# slideshow/models.py
from django.db import models
import widgy
from widgy.models import Content

@widgy.register
class Slideshow(Content):
    delay = models.PositiveIntegerField(default=2,
        help_text="The delay in seconds between slides.")

    accepting_children = True
    editable = True

@widgy.register
class Slide(Content):
    image = models.ImageField(upload_to='slides/', null=True)
    caption = models.CharField(max_length=255)

    editable = True
```

All widget classes inherit from *widgy.models.base.Content*. This creates the relationship with *widgy.models.base.Node* and ensures that all of the required methods are implemented.

In order to make a widget visible to Widgy, you have to add it to the registry. There are two functions in the widgy module that help with this, widgy.register() and widgy.unregister(). You should use the widgy.register() class decorator on any model class that you wish to use as a widget.

Both widgets need to have editable set to True. This will make an edit button appear in the editor, allowing the user to set the image, caption, and delay values.

Slideshow has accepting_children set to True so that you can put a Slide in it. The default implementation of *valid_parent_of()* checks accepting_children. We only need this until we override *valid_parent_of()* in *Step 3*.

Note: As you can see, the image field is null=True. It is necessary for all fields in a widget either to be null=True or to provide a default. This is because when a widget is dragged onto a tree, it needs to be saved without data.

CharFields don't need to be null=True because if they are non-NULL, the default is an empty string. For most other field types, you must have null=True or a default value.

Now we need to generate migration for this app.

```
$ python manage.py schemamigration --initial slideshow
```

And now run the migration.

```
$ python manage.py migrate
```

4.3.2 2. Write the Templates

After that, we need to write our templates. The templates are expected to be named `widgy/slideshow/slideshow/render.html` and `widgy/slideshow/slide/render.html`.

To create the slideshow template, add a file at `slideshow/templates/widgy/slideshow/slideshow/render.html`.

```
{% load widgy_tags %}
<div class="cycle-slideshow"
  data-cycle-timeout="{{ self.get_delay_milliseconds }}"
  data-cycle-caption-template="{% templatetag openvariable %}alt{% templatetag closevariable %}">
  <div class="cycle-caption"></div>

  {% for child in self.get_children %}
    {% render child %}
  {% endfor %}
</div>
```

For the slide, it's `slideshow/templates/widgy/slideshow/slide/render.html`.

```
<img src="{{ self.image.url }}" alt="{{ self.caption }}">
```

See also:

`Content.get_templates_hierarchy` Documentation for how templates are discovered.

The current `Slideshow` instance is available in the context as `self`. Because jQuery Cycle2 only accepts milliseconds instead of seconds for the delay, we need to add a method to the `Slideshow` class.

```
class Slideshow(Content):
    # ...
    def get_delay_milliseconds(self):
        return self.delay * 1000
```

The *`Content`* class mirrors several methods of the `TreeBeard API`, so you can call *`get_children()`* to get all the children. To render a child *`Content`*, use the *`render()`* template tag.

Caution: You might be tempted to include the HTML for each `Slide` inside the render template for `Slideshow`. While this does work, it is a violation of the single responsibility principle and makes it difficult for slides (or subclasses thereof) to change how they are rendered.

4.3.3 3. Write the Compatibility

Right now, the `Slideshow` and `Slide` render and could be considered complete; however, the way we have it, `Slideshow` can accept any widget as a child and a `Slide` can go in any parent. To disallow this, we have to write some *Compatibility* methods.

```
class Slideshow(Content):
    def valid_parent_of(self, cls, obj=None):
        # only accept Slides
        return issubclass(cls, Slide)

class Slide(Content):
    @classmethod
    def valid_child_of(cls, parent, obj=None):
        # only go in Slideshows
        return isinstance(parent, Slideshow)
```

Done.

4.3.4 Addendum: Limit Number of Children

Say you want to limit the number of `Slide` children to 3 for your `Slideshow`. You do so like this:

```
class Slideshow(Content):
    def valid_parent_of(self, cls, obj=None):
        if obj in self.get_children():
            # If it's already one of our children, it is valid
            return True
        else:
            # Make sure it's a Slide and that you aren't full
            return (issubclass(cls, Slide) and
                    len(self.get_children()) < 3)
```

4.4 Proxy Widgy Model Tutorial

Widgy was developed with a batteries included philosophy like Django. When building your own widgy project, you may find that you need to change the behavior of certain widgets. With `widgy.unregister()`, you can disable existing widgy models and re-enable it with your custom proxy model with `widgy.register()`.

This tutorial will cover a simple case where we add HTML classes to the `<input>` tags in the contrib module, Form Builder . This tutorial assumes that you have a working widgy project. Please go through the Quickstart if you do not have a working project.

In a sample project, we are adding Bootstrap styling to our forms. Widgy uses an easy template hierarchy to replace all of the templates for styling; however, when we get to adding the styling class, 'form-control', to each of our input boxes in the forms, there is no template to replace.

See also:

`Content.get_templates_hierarchy` Documentation for how templates are discovered.

Widgy uses the power of Django to create a widget with predefined attributes.

To insert the class, you will need to override the attribute `widget_attrs` in *`widgy.contrib.form_builder.models.FormInput`*. Start by creating a models.py file in your project and add your project to the `INSTALLED_APPS` if you have not done so already. Then add this to your models.py file:

```
import widgy

from widgy.contrib.form_builder.models import FormInput

widgy.unregister(FormInput)

@widgy.register
class BootstrapFormInput(FormInput):

    class Meta:
        proxy = True
        verbose_name = 'Form Input'
        verbose_name_plural = 'Form Inputs'

    @property
    def widget_attrs(self):
        attrs = super(BootstrapFormInput, self).widget_attrs
        attrs['class'] = attrs.get('class', '') + ' form-control'
        return attrs
```

This code simply unregisters the existing FormInput and reregisters our proxied version to replace the attribute `widget_attrs`.

To test it, simply create a form with a form input field and preview it in the widgy interface. When you view the HTML source for that field, you will see that the HTML class form-control is now added to `<input>`.

In another example, if you wanted to override the compatibility and `verbose_name` for Page Builder's `CalloutBucket`, you could do the following:

```
import widgy
from widgy.contrib.page_builder.models import CalloutBucket

widgy.unregister(CalloutBucket)

@widgy.register
class MyCalloutBucket(CalloutBucket):
    class Meta:
        proxy = True
        verbose_name = 'Awesome Callout'

    def valid_parent_of(self, cls, obj=None):
        return issubclass(cls, (MyWidget)) or \
            super(MyCalloutBucket, self).valid_parent_of(self, cls, obj)
```

Finally, when using proxy models, if you proxy and unregister a model that already has saved instances in the database, the old class will be used. If you still need to use the existing widgets for the new proxy model, you will need to write a database migration to update the content types. Here is a sample of what may be required for this migration:

```
Node.objects.filter(content_type=old_content_type).update(content_type=new_content_type)
```

More info on proxying models can be found on Django's documentation on proxy models

Changelog

5.1 0.7.1 (2015-08-18)

- Fix python 3 compatibility: SortedDict.keys() was returning an iterator instead of a view. This was causing `form_builder/forms/XX` not to display properly.

5.2 0.7.0 (2015-07-31)

- **Possible Breaking Change** Updated the django-pyscss dependency. Please see the django-pyscss changelog for documentation on how/if you need to change anything.
- Django 1.8 compatibility.
- Python 3 compatibility
- Django 1.7 is now the minimum supported version
- Mezzanine 4.0 is now the minimum supported version
- `Content.clone` now copies simple many-to-many relationships. If you have a widget with a many-to-many field and an overridden clone method that calls super, you should take this into account. If you have many-to-many relationships that use a custom through table, you will have to continue to override clone to clone those.
- **Backwards Incompatible** `WidgySite.has_add_permission` signature changed.
- Multisite support
  - One widgy project can now respond to multiple domains. Use cases could be Widgy as a Service or multi-franchise website.
  - This feature depends on Mezzanine multi-tenancy
  - Callouts are now tied to a django site
  - This feature is provided by `widgy.contrib.widgy_mezzanine.site.MultiSitePermissionMixin`

5.3 0.6.1 (2015-05-01)

- Fix non-determinism bug with find_media_files.

5.4 0.6.0 (2015-04-30)

- Improved the compatibility error messages [#299, #193]
- Remove the recommendation to use mezzanine.boot as it was not required [#291]
- **Possible Breaking Change** Updated the django-pyscss dependency. Please see the django-pyscss changelog for documentation on how/if you need to change anything.
- By default, Widgy views are restricted to staff members. Previously any authenticated user was allowed. This effects the preview view and pop out edit view, among others. If you were relying on the ability for any user to access those, override `authorize_view` in your `WidgySite`. [#267]:

```
class MyWidgySite(WidgySite):
    def authorize_view(self, request, view):
        if not request.user.is_authenticated()
            raise PermissionDenied
```

5.5 0.5.0 (2015-04-17)

- **Backwards Incompatible** RichTextPage is no longer unregistered by default in widgy_mezzanine. If you wish to unregister it, you can add the following to your admin.py file:

```
from django.contrib import admin
from mezzanine.pages.models import RichTextPage
admin.site.unregister(RichTextPage)
```

- Bugfix: Previously, the Widgy editor would break if CSRF_COOKIE_HTTPONLY was set to True [#311]
- Switched to py.test for testing. [#309]

5.6 0.4.0 (2015-03-12)

- Django 1.7 support. Requires upgrade to South 1.0 (Or use of SOUTH_MIGRATION_MODULES) if you stay on Django < 1.7. You may have to –fake some migrations to upgrade to the builtin Django migrations. Make sure your database is up to date using South, then upgrade Django and run:

```
./manage.py migrate --fake widgy
./manage.py migrate --fake easy_thumbnails
./manage.py migrate
```

- Support for installing Widgy without the dependencies of its contrib apps. The ‘django-widgy’ package only has dependencies required for Widgy core. Each contrib package has a setuptools ‘extra’. To install everything, replace ‘django-widgy’ with ‘django-widgy[all]’. [#221]
- Switched to tox for test running and allow running core tests without contrib. [#294]
- Stopped relying on urls with consecutive ‘/’ characters [#233]. This adds a new urlpattern for widgy_mezzanine’s preview page and form submission handler. The old ones will keep working, but you should reverse with ‘page_pk’ instead of ‘slug’. For example:

```
url = urlresolvers.reverse('widgy.contrib.widgy_mezzanine.views.preview', kwargs={
    'node_pk': node.pk,
    'page_pk': page.pk,
})
```

- Treat help_text for fields in a widget form as safe (HTML will not be escaped) [#298]. If you were relying on HTML special characters being escaped, you should replace `help_text="1 is < 2"` with `help_text=django.utils.html.escape("1 is < 2")`.
- Reverse URLs in form_builder admin with consideration for Form subclasses [#274].

5.7 0.3.5 (2015-01-30)

Bugfix release:

- Set model at runtime for ClonePageView and UnpublishView [Rocky Meza, #286]

5.8 0.3.4 (2015-01-22)

Bugfix release:

- Documentation fixes [Rocky Meza and Gavin Wahl]
- Fixes unintentional horizontal scrolling of Widgy content [Justin Stollsteimer]
- Increased spacing after widget title paragraphs [Justin Stollsteimer]
- Fixed styles in ckeditor to show justifications [Zachery Metcalf, #279]
- Eliminated the margins for InvisibleMixin [Rocky Meza]
- CSS support for adding fields to Image. [Rocky Meza]
- Additional mezzanine container style overflow fixes [Justin Stollsteimer]
- Fix r.js optimization errors with daisydiff [Rocky Meza]
- Remove delete button from widgypage add form [Gavin Wahl]

5.9 0.3.3 (2014-12-22)

Bugfix release:

- Allow cloning with an overridden WIDGY_MEZZANINE_PAGE_MODEL [Zach Metcalf, #269]
- SCSS syntax error [Rivo Laks, #271]

5.10 0.3.2 (2014-10-16)

Bugfix release:

- Allow WidgyAdmin to check for ReviewedWidgySite without review_queue installed [Scott Clark, #265]
- Fix handling of related_name on ProxyGenericRelation [#264]

5.11 0.3.1 (2014-10-01)

Bugfix release for 0.3.0. #261, #263.

5.12 0.3.0 (2014-09-24)

This release mainly focusses on the New Save Flow feature, but also includes several bug fixes and some nice CSS touch ups. There have been some updates to the dependencies, so please be sure to check the *How to Upgrade* section to make sure that you get everything updated correctly.

5.12.1 Major Changes

- New Save Flow **Requires upgrading Mezzanine to at least 3.1.10** [Gavin Wahl, Rocky Meza, #241]

  We have updated the workflow for WidgyPage. We consider this an experiment that we can hopefully expand to all WidgyAdmins in the future. We hope that this new save flow is more intuitive and less tedious.

  Screenshot of before:

  Screenshot of after:

  As you can see, we have rearranged some of the buttons and have gotten rid of the Published Status button. The new save buttons on the bottom right now will control the publish state as well as the commit status. This means that now instead of committing and saving being a two-step process, it all lives in one button. This should make editing and saving a smoother process. Additionally, we have renamed some buttons to make their intent more obvious.

5.12.2 Bug Fixes

- Updated overridden directory_table template for django-filer 0.9.6. **Requires upgrading django-filer to at least 0.9.6**. [Scott Clark, #179]
- Fix bug in ReviewedVersionTrackerQuerySet.published [Gavin Wahl, #240]
- Made commit buttons not look disabled [Justin Stollsteimer, #250, #205]
- (Demo) Added ADD_PAGE_ORDER to demo settings [Zach Metcalf, #248]
- (Demo) Updated demo project requirements [Scott Clark, #234]
- Make Widgy's jQuery private to prevent clashes with other admin extensions [Gavin Wahl, #246]

5.12.3 Documentation

- Update recommend ADMIN_MENU_ORDER to clarify django-filer [Gavin Wahl, #249]

5.12.4 How to Upgrade

In this release, widgy has udpated two of its dependencies:

- The minimum supported version of django-filer is now 0.9.6 (previously 0.9.5).
- The minimum supported version of Mezzanine is now 3.1.10 (previously 1.3.0).

If you `pip install django-widgy==0.3.0`, it should upgrade the dependencies for you, but just to be sure, you may want to also run

```
pip install 'django-filer>=0.9.6' 'Mezzanine>=3.1.10'
```

to make sure that you get the updates.

Note: Please note that if you are upgrading from an older version of Mezzanine, that the admin center has been restyled a little bit.

5.13 0.2.0 (2014-08-04)

5.13.1 Changes

- Widgy is now Apache Licensed
- **Breaking Change** Use django-pyscss for SCSS compilation. [Rocky Meza, #175]

  Requires an update to the `COMPRESS_PRECOMPILERS` setting:

  ```
  COMPRESS_PRECOMPILERS = (
      ('text/x-scss', 'django_pyscss.compressor.DjangoScssFilter'),
  )
  ```

  You may also have to update `@import` statements in your SCSS, because django-pyscss uses a different (more consistent) rule for path resolution. For example, `@import ’widgy_common’` should be changed to `@import ’/widgy/css/widgy_common’`
- Added help_text to Section to help user avoid bug [Zach Metcalf, #135]
- Allow UI to updated based on new data after reposition [Gavin Wahl, #199]
- Changed Button’s css_classes in shelf [Rocky Meza, #203]
- Added loading cursor while ajax is in flight [Gavin Wahl, #215, #208]
- Get rid of “no content” [Gavin Wahl, #206]
- Use sprites for the widget icons [Gavin Wahl and Rocky Meza, #89, #227]
- Only show approve/unapprove buttons for interesting commits [Gavin Wahl, #228]
- Updated demo app to have new design and new widgets [Justin Stollsteimer, Gavin Wahl, Antoine Catton and Rocky Meza, #129, #176]
- Added cloning for WidgyPages [Gavin Wahl, #235]
- Use a more realistic context to render pages for search [Gavin Wahl, #166]
- Add default children to Accordion and Tabs [Rocky Meza, #238]

5.13.2 Bugfixes

- Fix cursors related to dragging [Gavin Wahl, #155]
- Update safe urls [Gavin Wahl, #212]
- Fix widgy_mezzanine preview for Mezzanine==3.1.2 [Rocky Meza, #201]
- Allow RichTextPage in the admin [Zach Metcalf, #197]
- Don’t assume the response has a content-type header [Gavin Wahl, #216]
- Fix bug with FileUpload having empty values [Rocky Meza, #217]
- Fix urlconf_include login_required handling [Gavin Wahl, #200]

- Patch fancybox to work with jQuery 1.9 [Gavin Wahl, #222]
- Fix some import errors in SCSS [Rocky Meza, #230]
- Fix restore page in newer versions of Mezzanine [Gavin Wahl, #232]
- Use unicode format strings in review queue [Gavin Wahl, #236]

5.13.3 Documentation

- Updated quickstart to cover south migrations with easy_thumbnails [Zach Metcalf, #202]
- Added Proxy Widgy Model Tutorial [Zach Metcalf, #210]

5.14 0.1.6 (2014-09-09)

- Fix migrations containing unsupported KeywordsField from mezzanine [Scott Clark]
- Rename package to django-widgy

5.15 0.1.5 (2013-11-23)

- Fix Widgy migrations without Mezzanine [Gavin Wahl]
- Drop target collision detection [Gavin Wahl]
- Fix Figure and StrDisplayNameMixin [Gavin Wahl]
- Avoid loading review_queue when it's not installed [Scott Clark]
- Fix multi-table inheritance with LinkFields [Gavin Wahl]

5.16 0.1.4 (2013-11-04)

- Add StrDisplayNameMixin

5.17 0.1.3 (2013-10-25)

- Fix image widget validation with the S3 storage backend

5.18 0.1.2 (2013-10-23)

- Fix Widgy admin for static files hosted on a different domain

5.19 0.1.1 (2013-10-21)

- Adjust `MANIFEST.in` to fix PyPi install.
- Fix layout having a unicode `verbose_name`

5.20 0.1.0 (2013-10-18)

First release.

Basic features:

- Heterogeneous tree editor (`widgy`)
- CMS (`widgy.contrib.widgy_mezzanine`)
- CMS Plugins (`widgy.contrib.urlconf_include`)
- Widgets (`widgy.contrib.page_builder`)
- Form builder (`widgy.contrib.form_builder`)
- Multilingual pages (`widgy.contrib.widgy_i18n`)
- Review queue (`widgy.contrib.review_queue`)

CHAPTER 6

Development

You can follow and contribute to Widgy's development on GitHub. There is a developers mailing list available at widgy@fusionbox.com

Index

A

Accordion (class in widgy.contrib.page_builder.models), 8
add_child() (widgy.models.base.Content method), 13
add_root() (widgy.models.base.Content class method), 13
add_sibling() (widgy.models.base.Content method), 14
admin.VersionCommitAdminBase (class in widgy.contrib.review_queue), 11
admin_scss_files (widgy.site.WidgySite attribute), 19
authorize_view() (widgy.site.WidgySite method), 18

B

build_form_class() (widgy.contrib.form_builder.models.Form method), 9
Button (class in widgy.contrib.page_builder.models), 8

C

CalloutWidget (class in widgy.contrib.page_builder.models), 8
CAN_POP_OUT (widgy.models.base.Content attribute), 15
CANNOT_POP_OUT (widgy.models.base.Content attribute), 15
ChoiceField (class in widgy.contrib.form_builder.models), 9
clone() (widgy.models.base.Content method), 14
commit_view (widgy.site.WidgySite attribute), 19
Content (class in widgy.models.base), 13
content (widgy.models.base.Node attribute), 17
content_view (widgy.site.WidgySite attribute), 19
css_classes (widgy.models.base.Content attribute), 14

D

DefaultLayout (class in widgy.contrib.page_builder.models), 7
delete() (widgy.models.base.Content method), 14
depth_first_order() (widgy.models.base.Content method), 13
depth_first_order() (widgy.models.base.Node method), 17
diff_view (widgy.site.WidgySite attribute), 19
display_name (widgy.models.base.Content attribute), 14

E

edit_templates (widgy.models.base.Content attribute), 15
equal() (widgy.models.base.Content method), 17

F

Figure (class in widgy.contrib.page_builder.models), 8
find_widgy_problems() (widgy.models.base.Node class method), 17
Form (class in widgy.contrib.form_builder.models), 9
FormField (class in widgy.contrib.form_builder.models), 9
FormInput (class in widgy.contrib.form_builder.models), 9

G

get_all_content_classes() (widgy.site.WidgySite method), 17
get_ancestors() (widgy.models.base.Content method), 13
get_children() (widgy.models.base.Content method), 13
get_form() (widgy.models.base.Content method), 15
get_form_class() (widgy.models.base.Content method), 15
get_next_sibling() (widgy.models.base.Content method), 13
get_parent() (widgy.models.base.Content method), 13
get_render_templates() (widgy.models.base.Content method), 16
get_root() (widgy.models.base.Content method), 13
get_site() (widgy.contrib.review_queue.admin.VersionCommitAdminBase method), 11
get_templates_hierarchy() (widgy.models.base.Content class method), 15
get_urls() (widgy.site.WidgySite method), 18
get_version_tracker_model() (widgy.site.WidgySite method), 18
GoogleMap (class in widgy.contrib.page_builder.models), 8

H

has_add_permission() (widgy.site.WidgySite method), 18
has_change_permission() (widgy.site.WidgySite method), 18
has_delete_permission() (widgy.site.WidgySite method), 18
help_text (widgy.contrib.form_builder.models.FormField attribute), 9
history_view (widgy.site.WidgySite attribute), 19
Html (class in widgy.contrib.page_builder.models), 7

I

Image (class in widgy.contrib.page_builder.models), 8
ImageField (class in widgy.contrib.page_builder.db.fields), 8
is_frozen (widgy.models.base.Node attribute), 17

J

js_files (widgy.site.WidgySite attribute), 19
js_files() (in module widgy.templatetags.widgy_tags), 21

L

label (widgy.contrib.form_builder.models.FormField attribute), 9
LinkField (class in widgy.models.links), 20

M

MainContent (class in widgy.contrib.page_builder.models), 7
Markdown (class in widgy.contrib.page_builder.models), 7
maybe_prefetch_tree() (widgy.models.base.Node method), 17
MultipleChoiceField (class in widgy.contrib.form_builder.models), 9
MUST_POP_OUT (widgy.models.base.Content attribute), 15

N

Node (class in widgy.models.base), 17
node (widgy.models.base.Content attribute), 13
node_edit_view (widgy.site.WidgySite attribute), 19
node_parents_view (widgy.site.WidgySite attribute), 19
node_templates_view (widgy.site.WidgySite attribute), 19
node_view (widgy.site.WidgySite attribute), 19

P

post_create() (widgy.models.base.Content method), 14
prefetch_tree() (widgy.models.base.Node method), 17
prefetch_trees() (widgy.models.base.Node class method), 17
preview_templates (widgy.models.base.Content attribute), 15

R

render() (in module widgy.templatetags.widgy_tags), 21
render() (widgy.models.base.Content method), 16
render() (widgy.models.base.Node method), 17
render_root() (in module widgy.templatetags.widgy_tags), 21
reposition() (widgy.models.base.Content method), 14
required (widgy.contrib.form_builder.models.FormField attribute), 9
reset_view (widgy.site.WidgySite attribute), 19
reverse() (widgy.site.WidgySite method), 18
revert_view (widgy.site.WidgySite attribute), 19

S

scss_files (widgy.site.WidgySite attribute), 19
scss_files() (in module widgy.templatetags.widgy_tags), 21
Section (class in widgy.contrib.page_builder.models), 8
shelf_view (widgy.site.WidgySite attribute), 19
Sidebar (class in widgy.contrib.page_builder.models), 7

T

Table (class in widgy.contrib.page_builder.models), 8
TableData (class in widgy.contrib.page_builder.models), 8
TableHeaderData (class in widgy.contrib.page_builder.models), 8
TableRow (class in widgy.contrib.page_builder.models), 8
Tabs (class in widgy.contrib.page_builder.models), 8
Textarea (class in widgy.contrib.form_builder.models), 9
tooltip (widgy.models.base.Content attribute), 14

U

Uncaptcha (class in widgy.contrib.form_builder.models), 9
UnsafeHtml (class in widgy.contrib.page_builder.models), 8
urls() (widgy.site.WidgySite method), 17

V

valid_child_of() (widgy.models.base.Content class method), 16
valid_child_of() (widgy.site.WidgySite method), 18
valid_parent_of() (widgy.models.base.Content method), 16
valid_parent_of() (widgy.site.WidgySite method), 18
validate_relationship() (widgy.site.WidgySite method), 18
Video (class in widgy.contrib.page_builder.models), 8

W

widgy.contrib.form_builder.views.HandleFormMixin (class in widgy.contrib.form_builder.models), 10

widgy.contrib.widgy_mezzanine.models.WidgyPage (built-in class), 10

WidgySite (class in widgy.site), 17