Pyramid Layout Documentation

Release 0.4

Chris Rossi, Paul Everitt

March 12, 2016

Contents

1 Approach **3**

2 Contents **5**

2.1 About Layouts, Panels 5
2.2 Using Pyramid Layout 6
2.3 Demo App With Pyramid Layout 10
2.4 API Reference 14
2.5 Glossary 17

3 Indices and tables **19**

Python Module Index **21**

Making an attractive, efficient user-experience (UX) is hard. Pyramid Layout provides a layout-based approach to building your global look-and-feel then re-using it across your site. You can then manage your global UX *layout* as a unit, just like models, views, static resources, and other parts of Pyramid.

If you are OCD, and want the same ways to organize and override your UX that you get in your Python code, this *layout* approach is your cup of tea.

CHAPTER 1

Approach

- Make one (or more) *layout* objects of template and template logic
- Do useful things with this unit of layout: registration, dynamic association with a view, pluggability via Pyramid overrides, testing in isolation
- Layouts can share lightweight units called *panels* which are objects of template and code, sharing the same useful things
- Use of any of the common Pyramid templating engines (Chameleon ZPT, Mako, Jinja2) is tested and supported with examples.

CHAPTER 2

Contents

2.1 About Layouts, Panels

If you have a large project with lots of views and templates, you most likely have a lot of repetition. The header is the same, the footer is the same. A lot of CSS/JS is pulled in, etc.

Lots of template systems have ways to share templating between templates. But how do you get the data into the master template? You can put it in the view and pass it in, but then it is hard to know what parts belong to the view versus the main template. Then there's testing, overriding, cases where you have multiple main templates.

Wouldn't it be nice to have a formal concept called "Layout" that gained many of the benefits of Pyramid machinery like views?

This section introduces the concepts of *layout* and "panel".

2.1.1 About Layouts

Most projects have a global look-and-feel and each view plugs into it. In ZPT-based systems, this is usually done with a main template that uses METAL to wrap each view's template.

Having the template, though, isn't enough. The template usually has logic in it and needs data. Usually each view had to pass in that data. Later, Pyramid's renderer globals provided an elegant facility for always having certain data available in all renderings.

In Pyramid Layout, these ideas are brought together and given a name: *layout*. A layout is a combination of templating and logic to which a view template can point. With Pyramid Layout, *layout* becomes a first-class citizen with helper config machinery and defined plug points.

In more complex projects, different parts of the same site need different layouts. Pyramid Layout provides a way for managing the use of different layouts in different places in your application.

2.1.2 About Panels

In your project you might have a number of layouts and certainly many view templates. Reuse is probably needed for little boxes on the screen. Or, if you are using someone else's layout, you might want to change one small part without forking the entire template.

In ZPT, macros provide this functionality. That is, re-usable snippets of templating with a marginal amount of overidability. Like main templates, though, they also have logic and data that need to be schlepped into the template.

Pyramid Layout addresses these re-usable snippets with panels. A *panel* is a box on the screen driven by templating and logic. You make panels, register them, and you can then use them in your view templates or *main templates*.

Moreover, making and using them is a very Pythonic, Pyramid-like process. For example, you call your *panel* as a normal Python callable and can pass it arguments. Registration of panels, like *layouts*, is very similar to registration of views in Pyramid.

2.2 Using Pyramid Layout

To get started with Pyramid Layout, include `pyramid_layout` in your application's config:

```
config = Configurator(...)
config.include('pyramid_layout')
```

Alternately, instead of using the the Configurator's include method, you can activate Pyramid Layout by changing your application's .ini file, using the following line:

```
pyramid.includes = pyramid_layout
```

Including `pyramid_layout` in your application adds two new directives to your configurator: *`add_layout`* and *`add_panel`*. These directives work very much like `add_view`, but add registrations for layouts and panels. Including `pyramid_layout` will also add an attribute, `layout_manager`, to the request object of each request, which is an instance of *`LayoutManager`*.

Finally, three renderer globals are added which will be available to all templates: `layout`, `main_template`, and `panel`. `layout` is an instance of the *layout class* of the current layout. `main_template` is the template object that provides the *main template* (aka, o-wrap) for the view. `panel`, a shortcut for *`LayoutManager.render_panel`*, is a callable used to render *panels* in your templates.

2.2.1 Using Layouts

A *layout* consists of a *layout class* and *main template*. The layout class will be instantiated on a per request basis with the context and request as arguments. The layout class can be omitted, in which case a default layout class will be used, which only assigns *context* and *request* to the layout instance. Generally, though, you will provide your own layout class which can serve as a place to provide API that will be available to your templates. A simple layout class might look like:

```
class MyLayout(object):
    page_title = 'Hooray! My App!'

    def __init__(self, context, request):
        self.context = context
        self.request = request
        self.home_url = request.application_url

    def is_user_admin(self):
        return has_permission(self.request, 'manage')
```

A *layout instance* will be available in templates as the renderer global, `layout`. For example, if you are using Mako or ZPT for templating, you can put something like this in a template:

```
<title>${layout.page_title}</title>
```

For Jinja2:

```
<title>{{layout.page_title}}</title>
```

All *layouts* must have an associated template which is the *main template* for the layout and will be present as `main_template` in renderer globals.

To register a layout, use the *add_layout* method of the configurator:

```
config.add_layout('myproject.layout.MyLayout',
                  'myproject.layout:templates/default_layout.pt')
```

The above registered layout will be the default layout. Layouts can also be named:

```
config.add_layout('myproject.layout.MyLayout',
                  'myproject.layout:templates/admin_layout.pt',
                  name='admin')
```

Now that you have a layout, time to use it on a particular view. Layouts can be defined declaratively, next to your renderer, in the view configuration:

```
@view_config(..., layout='admin')
def myview(context, request):
    ...
```

In Pyramid < 1.4, to use a named layout, call *LayoutManager.use_layout* method in your view:

```
def myview(context, request):
    request.layout_manager.use_layout('admin')
    ...
```

If you are using traversal you may find that in most cases it is unnecessary to name your layouts. Use of the *context* argument to the layout configuration can allow you to use a particular layout whenever the context is of a particular type:

```
from ..models.wiki import WikiPage

config.add_layout('myproject.layout.MyLayout',
                  'myproject.layout:templates/wiki_layout.pt',
                  context=WikiPage)
```

Similarly, the *containment* argument allows you to use a particular layout for an entire branch of your resource tree:

```
from ..models.admin import AdminFolder

config.add_layout('myproject.layout.MyLayout',
                  'myproject.layout:templates/admin_layout.pt',
                  containment=AdminFolder)
```

The decorator *layout_config* can be used in conjuction with `Configurator.scan` to register layouts declaratively:

```
from pyramid_layout.layout import layout_config

@layout_config(template='templates/default_layout.pt')
@layout_config(name='admin', template='templates/admin_layout.pt')
class MyLayout(object):
    ...
```

Layouts can also be registered for specific context types and containments. See the *api docs* for more info.

2.2.2 Using Panels

A *panel* is similar to a view but is responsible for rendering only a part of a page. A panel is a callable which can accept arbitrary arguments (the first two are always `context` and `request`) and either returns an html string or uses a Pyramid renderer to render the html to insert in the page.

Note: You can mix-and-match template languages in a project. Some panels can be implemented in Jinja2, some in Mako, some in ZPT. All can work in layouts implemented in any template language supported by Pyramid Layout.

A *panel* can be configured using the method, `add_panel` of the `Configurator` instance:

```
config.add_panel('myproject.layout.siblings_panel', 'siblings',
                 renderer='myproject.layout:templates/siblings.pt')
```

Because *panels* can be called with arguments, they can be parameterized when used in different ways. The panel callable might look something like:

```
def siblings_panel(context, request, n_siblings=5):
    return [sibling for sibling in context.__parent__.values()
            if sibling is not context][:n_siblings]
```

And could be called from a template like this:

```
${panel('siblings', 8)}  <!-- Show 8 siblings -->
```

If using `Configurator.scan`, you can also register the panel declaratively:

```
from pyramid_layout.panel import panel_config

@panel_config('siblings', renderer='templates/siblings.pt')
def siblings_panel(context, request, n_siblings=5):
    return [sibling for sibling in context.__parent__.values()
            if sibling is not context][:n_siblings]
```

Like *layouts*, *panels* can also be registered for a context type:

```
from pyramid_layout.panel import panel_config

@panel_config(name='see-also'
              context='myproject.models.Document',
              renderer='templates/see-also.pt')
def see_also(context, request):
    return {'title': context.title,
            'url': request.resource_url(context)}
```

The context to use to look up a panel defaults to the context found during traversal. A different context may be provided by passing a *context* keyword argument to panel call. In this hypothetical template, each *related_content* item can potentially be a different type and wind up invoking a different panel:

```
<h2>Related Content</h2>
<ul>
  <li tal:repeat="item releated_content">
    ${panel('see-also', context=item)}
  </li>
</ul>
```

When registering panels by context, the *name* part of the registration becomes optional. In the example above, we could make the *see-also* panels the default panels for any registered contexts by simply omitting *name*:

```
from pyramid_layout.panel import panel_config

@panel_config(context='myproject.models.Document',
              renderer='templates/see-also.pt')
def see_also(context, request):
```

```
    return {'title': context.title,
            'url': request.resource_url(context)}
```

Also in the template:

```
<h2>Related Content</h2>
<ul>
  <li tal:repeat="item releated_content">
    ${panel(context=item)}
  </li>
</ul>
```

See the *api docs* for more info.

2.2.3 Using the Main Template

The precise syntax for hooking into the *main template* from a view template varies depending on the templating language you're using.

ZPT

If you are a ZPT user, connecting your view template to the *layout* and its *main template* is pretty easy. Just make this the outermost element in your view template:

```
<metal:block use-macro="main_template">
...
</metal:block>
```

You'll note that we're taking advantage of a feature in Chameleon that allows us to use a template instance as a macro without having to explicitly define a macro in the *main template*.

After that, it's about what you'd expect. The *main template* has to define at least one slot. The view template has to fill at least one slot.

Mako

In Mako, to use the *main template* from your *layout*, use this as the first line in your view template:

```
<%inherit file="${context['main_template'].uri}"/>
```

In your *main template*, insert this line at the point where you'd like for the view template to be inserted:

```
${next.body()}
```

Jinja2

For Jinja2, to use the *main template* for your *layout*, use this as the first line in your view template:

```
{% extends main_template %}
```

From there, blocks defined in your *main template* can be overridden by blocks defined in your view template, per normal usage.

2.3 Demo App With Pyramid Layout

Let's see Pyramid Layout in action with the demo application provided in `demo`.

2.3.1 Installation

Normal Pyramid stuff:

1. Make a virtualenv
2. `env/bin/python demo/setup.py develop`
3. `env/bin/pserve demo/development.ini`
4. Open `http://0.0.0.0:6543/` in a browser
5. Click on the `Home Mako`, `Home Chameleon`, and `Home Jinja2` links in the header to see views for that use each.

Now let's look at some of the code.

2.3.2 Registration

Pyramid Layout defines configuration directives and decorators you can use in your project. We need those loaded into our code. The demo does this in the `etc/development.ini` file:

```
pyramid.includes =
    pyramid_debugtoolbar

mako.directories = demo:templates
```

The `development.ini` entry point starts in `demo/__init__.py`:

```
from pyramid.config import Configurator

def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    config.include('pyramid_chameleon')
    config.include('pyramid_jinja2')
    config.include('pyramid_mako')
    config.include('pyramid_layout')
    config.add_static_view('static', 'static', cache_max_age=3600)
    config.add_route('home.mako', '/')
    config.add_route('home.chameleon', '/chameleon')
    config.add_route('home.jinja2', '/jinja2')
    config.scan('.layouts')
    config.scan('.panels')
    config.scan('.views')
    return config.make_wsgi_app()
```

This is all Configurator action. We register a route for each view. We then scan our `demo/layouts.py`, `demo/panels.py`, and `demo/views.py` for registrations.

2.3.3 Layout

Let's start with the big picture: the global look-and-feel via a *layout*:

```
1  from pyramid_layout.layout import layout_config
2
3
4  @layout_config(template='demo:templates/layouts/layout.mako')
5  @layout_config(
6      name='chameleon',
7      template='demo:templates/layouts/layout.pt'
8      )
9  @layout_config(
10     name='jinja2',
11     template='demo:templates/layouts/layout.jinja2'
12     )
13 class AppLayout(object):
14
15     def __init__(self, context, request):
16         self.context = context
17         self.request = request
18         self.home_url = request.application_url
19         self.headings = []
20         self.portlets = (Thing1(), Thing2(), LittleCat("A"))
21
22     @property
23     def project_title(self):
24         return 'Pyramid Layout App!'
25
26     def add_heading(self, name, *args, **kw):
27         self.headings.append((name, args, kw))
28
29
30 class Thing1(object):
31     title = "Thing 1"
32     content = "I am Thing 1!"
33
34
35 class Thing2(object):
36     title = "Thing 2"
37     content = "I am Thing 2!"
38
39
40 class LittleCat(object):
41     talent = "removing pink spots"
42
43     def __init__(self, name):
44         self.name = name
```

The `@layout_config` decorator comes from Pyramid Layout and allows us to define and register a *layout*. In this case we've stacked 3 decorators, thus making 3 layouts, one for each template language.

Note: The first `@layout_config` doesn't have a `name` and is thus the layout that you will get if your view doesn't specifically choose which layout it wants.

Lines 21-24 illustrates the concept of keeping templates and the template logic close together. All views need to show the `project_title`. It's part of the global look-and-feel *main template*. So we put this logic on the *layout*, in one

place as part of the global contract, rather than having each view supply that data/logic.

Let's next look at where this is used in the template for one of the 3 layouts. In this case, the Mako template at `demo/templates/layouts/layout.mako`:

```
<title>${layout.project_title}, from Pylons Project</title>
```

Here we see an important concept and some important magic: the template has a top-level variable `layout` available. This is an instance of your *layout class*.

For the ZPT crowd, if you look at the master template in `demo/templates/layouts/layout.pt`, you might notice something weird at the top: there's no `metal:define-macro`. Since Chameleon allows a template to be a top-level macro, Pyramid Layout automatically binds the entire template to the macro named `main_template`.

How does your view know to use a *layout*? Let's take a look.

2.3.4 Connecting Views to a Layout

Our demo app has a very simple set of views:

```
from pyramid.view import view_config

@view_config(
    route_name='home.mako',
    renderer='demo:templates/home.mako'
    )
@view_config(
    route_name='home.chameleon',
    renderer='demo:templates/home.pt',
    layout='chameleon'
    )
@view_config(
    route_name='home.jinja2',
    renderer='demo:templates/home.jinja2',
    layout='jinja2'
    )
def home(request):
    lm = request.layout_manager
    lm.layout.add_heading('heading-mako')
    lm.layout.add_heading('heading-chameleon')
    lm.layout.add_heading('heading-jinja2')
    return {}
```

We again have one callable with 3 stacked decorators. The decorators are all normal Pyramid `@view_config` stuff.

The second one points at a Chameleon template in `demo/templates/home.pt`:

```
<metal:block use-macro="main_template">

  <div metal:fill-slot="content">
    <!-- Main hero unit for a primary marketing message or call to action -->
    ${panel('hero', title='Chameleon')}

    <!-- Example row of columns -->
    <div class="row">
      <p>${panel('headings')}</p>
    </div>
    <div class="row">
        <p>${panel('contextual_panels')}</p>
```

```
    </div>
    <div class="row">
        <h2>User Info</h2>
        <p>${panel('usermenu',
            user_info={
            'first_name': 'Jane',
            'last_name': 'Doe',
            'username': 'jdoe'}
            )}</p>
    </div>
  </div>

</metal:block>
```

The first line is the one that opts the template into the *layout*. In `home.jinja2` that line looks like:

```
{% extends main_template %}
```

For both of these, `main_template` is inserted by Pyramid Layout, via a Pyramid renderer global, into the template's global namespace. After that, it's normal semantics for that template language.

Back to `views.py`. The view function grabs the `Layout Manager`, which Pyramid Layout conveniently stashes on the request. The `LayoutManager`'s primary job is getting/setting the current *layout*. Which, of course, we do in this function.

Our function then grabs the *layout instance* and manipulates some state that is needed in the global look and feel. This, of course, could have been done in our `AppLayout` class, but in some cases, different views have different values for the headings.

2.3.5 Re-Usable Snippets with Panels

Our *main template* has something interesting in it:

```
  <body>

    ${panel('navbar')}

    <div class="container">

      ${next.body()}

      <hr>

      <footer>
        ${panel('footer')}
      </footer>

    </div> <!-- /container -->

    <!-- Le javascript
    ================================================== -->
    <!-- Placed at the end of the document so the pages load faster -->
    <script src="${request.static_url('demo:static/js/jquery-1.8.0.min.js')}"></script>
    <script src="${request.static_url('demo:static/js/bootstrap.min.js')}"></script>

  </body>
```

Here we break our global layout into reusable parts via *panels*. Where do these come from? `@panel_config` decorators, as shown in `panels.py`. For example, this:

```
${panel('navbar')}
```

...comes from this:

```
@panel_config(
    name='navbar',
    renderer='demo:templates/panels/navbar.mako'
    )
def navbar(context, request):
    def nav_item(name, url):
        active = request.current_route_url() == url
        item = dict(
            name=name,
            url=url,
            active=active
            )
        return item
    nav = [
        nav_item('Mako', request.route_url('home.mako')),
        nav_item('Chameleon', request.route_url('home.chameleon')),
        nav_item('Jinja2', request.route_url('home.jinja2'))
        ]
    return {
        'title': 'Demo App',
```

The `@panel_config` registered a panel under the name `navbar`, which our template could then use **or override**.

The `home.mako` view template has a more interesting panel:

```
${panel('hero', title='Mako')}
```

...which calls:

```


@panel_config(
    name='hero',
    renderer='demo:templates/panels/hero.mako'
    )
```

This shows that a *panel* can be parameterized and used in different places in different ways.

2.4 API Reference

2.4.1 pyramid_layout.config

pyramid_layout.config.**add_layout**(*config*, *layout=None*, *template=None*, *name=''*, *context=None*, *containment=None*)

Add a layout configuration to the current configuration state.

Arguments

layout

A layout class or dotted Python name which refers to a layout class. This argument is not required. If the layout argument is not provided, a default layout class is used which merely has `context` and `request` as instance attributes.

template

A string implying a path or an asset specification for a template file. The file referred to by this argument must have a suffix that maps to a known renderer. This template will be available to other templates as the renderer global `main_template`. This argument is required.

name

The layout name.

context

An object or a dotted Python name referring to an interface or class object that the context must be an instance of, *or* the interface that the context must provide in order for this layout to be found and used. This predicate is true when the context is an instance of the represented class or if the context provides the represented interface; it is otherwise false.

containment

This value should be a Python class or interface (or a dotted Python name) that an object in the lineage of the context must provide in order for this view to be found and called. The nodes in your object graph must be "location-aware" to use this feature.

`pyramid_layout.config.`**`add_panel`**(*config*, *panel=None*, *name=''*, *context=None*, *renderer=None*, *attr=None*)

Add a panel configuration to the current configuration state.

Arguments

panel

A panel callable or a dotted Python name which refers to a panel callable. This argument is required unless a `renderer` argument also exists. If a `renderer` argument is passed, and a `panel` argument is not provided, the panel callable defaults to a callable that returns an empty dictionary.

attr

The panel machinery defaults to using the `__call__` method of the panel callable (or the function itself, if the panel callable is a function) to obtain a response. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your panel was a class, and the class has a method named `index` and you wanted to use this method instead of the class' `__call__` method to return the response, you'd say `attr="index"` in the panel configuration for the panel. This is most useful when the panel definition is a class.

renderer

This is either a single string term (e.g. `json`) or a string implying a path or asset specification (e.g. `templates/panels.pt`) naming a renderer implementation. If the `renderer` value does not contain a dot `.`, the specified string will be used to look up a renderer implementation, and that renderer implementation will be used to construct a response from the panel return value. If the `renderer` value contains a dot (`.`), the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the renderer implementation, which will be passed the full path. The renderer implementation will be used to construct a response from the panel return value.

Note that if the panel itself returns an instance of *basestring* (or just *str* in Python 3), the specified renderer implementation is never called.

When the renderer is a path, although a path is usually just a simple relative pathname (e.g. `templates/foo.pt`, implying that a template named "foo.pt" is in the "templates" directory relative to the directory of the current package of the Configurator), a path can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternately be an asset specification in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in a separate package.

The `renderer` attribute is optional. If it is not defined, the "null" renderer is assumed (no rendering is performed and the value is passed back to the upstream Pyramid machinery unmodified).

name

The optional panel name, which defaults to an empty string.

context

An object or a dotted Python name referring to an interface or class object that the context must be an instance of, *or* the interface that the context must provide in order for this panel to be found and called. This predicate is true when the context is an instance of the represented class or if the context provides the represented interface; it is otherwise false.

2.4.2 pyramid_layout.layout

class `pyramid_layout.layout.`**`LayoutManager`**(*context*, *request*)

An instance of LayoutManager will be available as the `layout_manager` attribute of the `request` object in views and allows the view to access or change the current layout.

`layout`

Property which gets the current layout.

`render_panel`(*name=''*, **args*, ***kw*)

Renders the named panel, returning a *unicode* object that is the rendered HTML for the panel. The panel is looked up using the current context (or the context given as keyword argument, to override the context in which the panel is called) and an optional given name (which defaults to an empty string). The panel is called passing in the current context, request and any additional parameters passed into the *render_panel* call. In case a panel isn't found, *None* is returned.

`use_layout`(*name*)

Makes a layout with the given name the current layout. By default an unnamed layout which matches the current context and containment will be the current layout. By specifying a named layout using *`LayoutManager.use_layout()`*, a named view matching the current context, containment, and given name will be used.

`pyramid_layout.layout.`**`layout_config`**(*name=''*, *context=None*, *template=None*, *containment=None*)

A class decorator which allows a developer to create layout registrations.

For example, this code in a module `layout.py`:

```
@layout_config(name='my_layout', template='mypackage:templates/layout.pt')
class MyLayout(object):

    def __init__(self, context, request):
        self.context = context
        self.request = request
```

The following arguments are supported as arguments to *`pyramid_layout.layout.layout_config`*: `context`, `name`, `template`, `containment`.

The meanings of these arguments are the same as the arguments passed to *pyramid_layout.config.add_layout()*.

2.4.3 pyramid_layout.panel

pyramid_layout.panel.**panel_config**(*name=''*, *context=None*, *renderer=None*, *attr=None*)
A function, class or method decorator which allows a developer to create panel registrations.

For example, this code in a module `panels.py`:

```
from resources import MyResource

@panel_config(name='my_panel', context=MyResource):
def my_panel(context, request):
    return 'OK'
```

The following arguments are supported as arguments to *pyramid_layout.panel.panel_config*: `context`, `name`, `renderer`, `attr`.

The meanings of these arguments are the same as the arguments passed to *pyramid_layout.config.add_panel()*.

2.5 Glossary

layout The basic unit of reusable look and feel, a layout consists of a *main template* and a *layout class*.

layout class A class registered with a layout that can be used as a place to consolidate API that is common to all or many templates across a project.

layout instance An instance of a *layout class*. For each view, a *layout* is selected and that layout's *layout class* is instantiated for the current request and context and made available to templates as the renderer global, `layout`.

main template Also known as the *o-wrap* or *outer wrapper*, this is a template which contains HTML that is common to all views that share a particular layout. View templates are derived from the main template and inject their own HTML into the HTML defined by the main template.

panel A panel is a reusable component that defines the HTML for small piece of an entire page. Panels are callables, like views, and may either return an HTML string or use a renderer for generating HTML to embed in the page.

CHAPTER 3

Indices and tables

- genindex
- modindex

p

pyramid_layout.config, 14
pyramid_layout.layout, 16
pyramid_layout.panel, 17

A

add_layout() (in module pyramid_layout.config), 14
add_panel() (in module pyramid_layout.config), 15

L

layout, **17**
layout (pyramid_layout.layout.LayoutManager attribute), 16
layout class, **17**
layout instance, **17**
layout_config() (in module pyramid_layout.layout), 16
LayoutManager (class in pyramid_layout.layout), 16

M

main template, **17**

P

panel, **17**
panel_config() (in module pyramid_layout.panel), 17
pyramid_layout.config (module), 14
pyramid_layout.layout (module), 16
pyramid_layout.panel (module), 17

R

render_panel() (pyramid_layout.layout.LayoutManager method), 16

U

use_layout() (pyramid_layout.layout.LayoutManager method), 16