Django notes

# Things to remember

Static pages—use flatpages app: <https://docs.djangoproject.com/en/1.8/ref/contrib/flatpages/>

URL patterns: <https://docs.djangoproject.com/en/1.8/releases/1.8/#django-conf-urls-patterns>

# Code snippets

Add a class to the active menu item:

<ul class = 'nav'>

<li class="{% ifequal request.path 'about/'%} active {% endifequal%}">

<a href="{% url "about" %}">About</a>

</li>

</ul>

Timestamp model:

# core/models.py

from django.db import models

class TimeStampedModel(models.Model):

"""

An abstract base class model that provides selfupdating

``created`` and ``modified`` fields.

"""

created = models.DateTimeField(auto\_now\_add=True)

modified = models.DateTimeField(auto\_now=True)

class Meta:

abstract = True

# flavors/models.py

from django.db import models

from core.models import TimeStampedModel

class Flavor(TimeStampedModel):

title = models.CharField(max\_length=200)

URL namespaces:

# urls.py at root of project

urlpatterns += [

url(r'^tastings/', include(‘tastings.urls’,

namespace='tastings')),

]

# tastings/views.py snippet

class TasteUpdateView(UpdateView):

model = Tasting

def get\_success\_url(self):

return reverse("tastings:detail",

kwargs={"pk": self.object.pk})

# in template

{% extends "base.html" %}

{% block title %}Tastings{% endblock title %}

{% block content %}

<ul>

{% for taste in tastings %}

<li>

<a href="{% url "tastings:detail" taste.pk %}">{{ taste.title }} </a>

<small>

(<a href="{% url "tastings:update" taste.pk %}">update</a>)

</small>

</li>

{% endfor %}

</ul>

{% endblock content %}

Simplest views:

# simplest\_views.py

from django.http import HttpResponse

from django.views.generic import View

# The simplest FBV

def simplest\_view(request):

# Business logic goes here

return HttpResponse("FBV")

# The simplest CBV

class SimplestView(View):

def get(self, request, \*args, \*\*kwargs):

# Business logic goes here

return HttpResponse("CBV")

Mixin example:

from django.views.generic import TemplateView

class FreshFruitMixin(object):

def get\_context\_data(self, \*\*kwargs):

context = super(FreshFruitMixin, self).get\_context\_data(\*\*kwargs)

context["has\_fresh\_fruit"] = True

return context

class FruityFlavorView(FreshFruitMixin, TemplateView):

template\_name = "fruity\_flavor.html"

Model order (<http://www.marinamele.com/taskbuster-django-tutorial/model-creation-onetoone-relationship-signals-django-admin>):

class MyModel(models.Model):

# Relations

# Attributes - Mandatory

# Attributes - Optional

# Object Manager

# Custom Properties

# Methods

# Meta and String

How to use signals (<http://www.marinamele.com/taskbuster-django-tutorial/model-creation-onetoone-relationship-signals-django-admin>):

from django.dispatch import receiver

from django.db.models.signals import post\_save

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL)

def create\_profile\_for\_new\_user(sender, created, instance, \*\*kwargs):

if created:

profile = Profile(user=instance)

profile.save()

# *Two Scoops* Notes

Keep code readable:

* Avoid abbreviating variable names.
* Write out your function argument names.
* Document your classes and methods.
* Comment your code.
* Refactor repeated lines of code into reusable functions or methods.
* Keep functions and methods short. A good rule of thumb is that scrolling should not be necessary to read an entire function or method.

Project layout examples:

<repository\_root>/

<django\_project\_root>/

<configuration\_root>/

icecreamratings\_project/

.gitignore

Makefile

docs/

README.rst

requirements.txt

icecreamratings/

manage.py

media/ # Development ONLY!

products/

profiles/

ratings/

static/

templates/

config/

\_\_init\_\_.py

settings/

urls.py

wsgi.py

Use Cookiecutter to generate your boilerplate from cookiecutter-django.

Check chapter 6 for list of when not to use null=True.

Use get\_object\_or\_404() in views when retrieving a single item.

Guidelines When Working With CBVs

ä Less view code is better.

ä Never repeat code in views.

ä Views should handle presentation logic. Try to keep business logic in models when possible,

or in forms if you must.

ä Keep your views simple.

ä Don’t use CBVs to write custom 403, 404, and 500 error handlers. Use FBVs instead.

ä Keep your mixins simpler

Mixins: In programming, a mixin is a class that provides functionality to be inherited, but isn’t meant for instantiation on its own. In programming languages with multiple inheritance, mixins can be used to add enhanced functionality and behavior to classes.

# Behavior

* User views product page
* User decides to add product to order
  + User chooses a quantity and hits submit
* User is redirected to a page of related items
  + User can add all related items to order from this page
    - Item that has been added is indicated by a color change or message
* User is redirected to a form to fill out her contact information and order information
* User receives email with order information
* Store receives email with order information
* After user is contacted, order is marked complete/contacted

OR:

* User views product page
* User decides to add product to order
* User chooses/enters a quantity and hits “add to order”
* Site checks whether user is logged in
  + If logged in, proceed to related items step
  + If not logged in, give user option to log in or register
    - User is redirected back to product page
      * Can product page save quantity entered before?
* User is directed to a “related items page”
* User adds related items
  + User
* User adds more information about order
  + Expected number of guests
  + Store pickup/deliver location
* User marks order complete
* User receives email with order information
* Store receives email with order information
* Store contacts user
* Store adds notes to order?
  + Create section of site or use Django admin?
* Store marks order contacted/complete
* User gets email with completed order information
* Store gets email with completed order information
* Order is no longer visible on active orders list, but can be retrieved/viewed from completed orders list

User needs to be able to:

* Register
* Log in
* View pending orders
* View completed orders
* Create a new order
* Modify existing order?

Store needs to be able to:

* Log in with user name created by admin
* View pending orders
* View completed orders
* Modify existing orders
* Create a new order

# Packages to use

django-model-utils: <https://django-model-utils.readthedocs.org/en/latest/>

For timestamps

django-vanilla-views; <http://django-vanilla-views.org/>

For easy CBV

django-braces: <https://django-braces.readthedocs.org/en/latest/>

For permissions and form mixins

psycopg2: <http://initd.org/psycopg/>

Needed for Postgres

django-allauth: <http://django-allauth.readthedocs.org/en/latest/>

For authentication with social accounts and Google