Various Topics

ECS639: Web Programming

(week 8)

Dr Paulo Oliva

Queen Mary
University of London

Week 8

1. Review	Quiz
2. Web app engineering	Quiz
3. Advanced Django	Q&A
4. Git	QM+ video

IT'S QUIZ TIME

https://cloudsurvey.co.uk

More on Web Apps Architecture

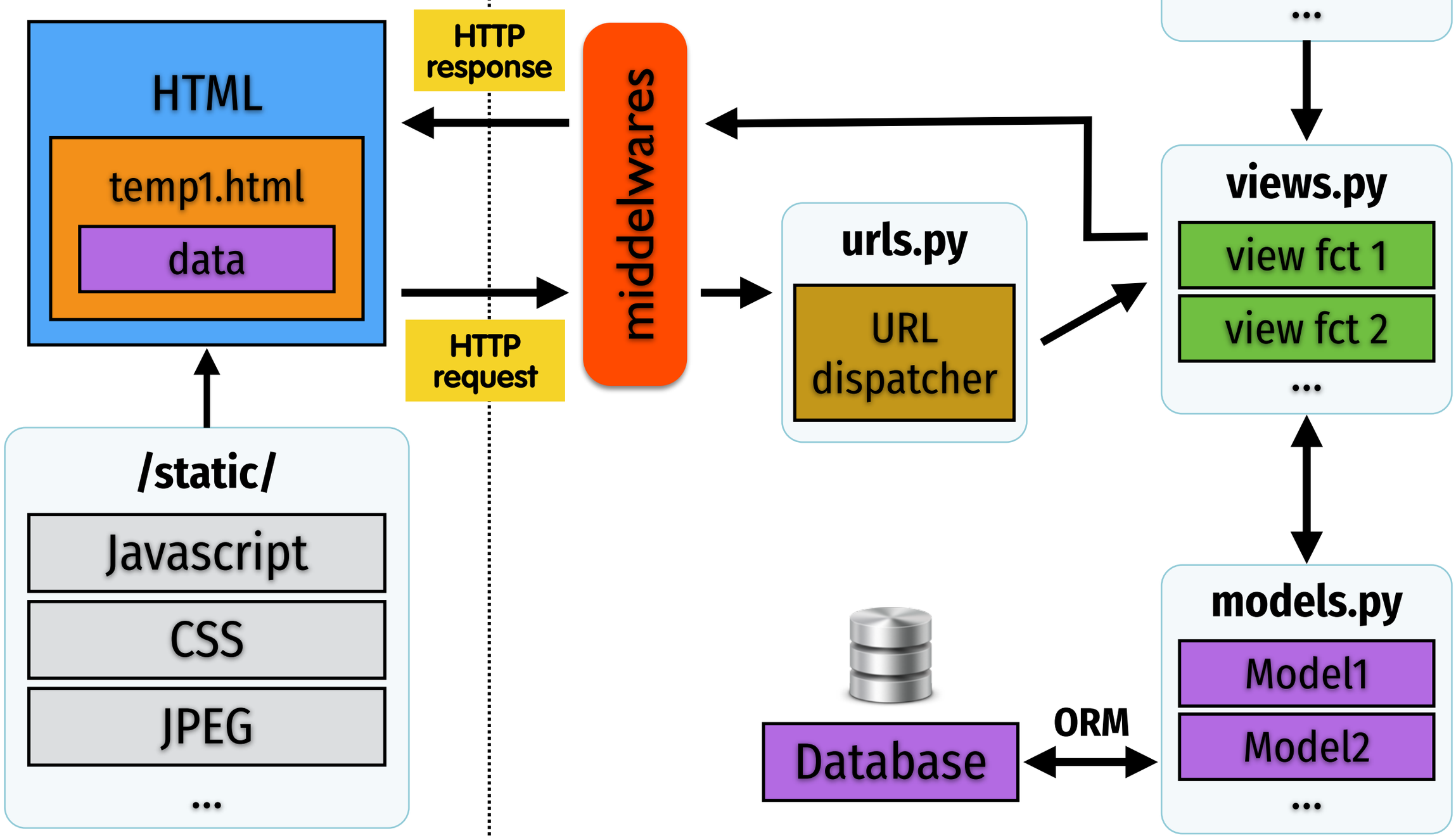
Web Frameworks

- Benefits:
  - Introduces **abstraction** layers
  - Enables **separation of concerns**
  - Enables **DRY**
- Cost:
  - Learn about conventions of framework
  - Details change from one framework to another

Abstraction

Feature	Abstracts…
Models and QuerySets	Concrete Database
Request/Response objects	HTTP Protocol
	Cookies
	Session Management
Manage.py	Maintain, Test and Debug

Separation of Concerns


/templates/
temp1.html
temp2.html
...
HTML
temp1.html
data
HTTP response
HTTP request
middelwares
urls.py
URL dispatcher
views.py
view fct 1
view fct 2
...
/static/
Javascript
CSS
JPEG
...
models.py
Model1
Model2
...
Database
ORM

Separation of Concerns

Role	Player
Frontend Content	HTML + template variables
Frontend Presentation Logic	Template Language
Frontend Dynamics	jQuery
Frontend Style	CSS / BootStrap
Business Logic	View Functions
High Level Model	Python Classes
Low Level Data	ORM
Application URLs	URL dispatcher

URL Reversing

```
<!-- signup.html template -->
...
<form method='POST' action="/social/register/">
...
```

```
action="{% url 'register' %}"
```

```
# urls.py
urlpatterns = [
  ...
  # signup page
  path('register/', views.register, name='register'),
  ...
]
```

https://docs.djangoproject.com/en/stable/topics/http/urls/#reverse-resolution-of-urls

URL reversing allows for separation of **URL paths** (URL dispatcher) and **HTML code** (templates)

CSS allows for separation of **content style** (CSS) and **content structure** (HTML and templates)

jQuery allows for separation of **page dynamics** (jQuery and Javascript) and **page contents** (HTML and templates)

QuerySet API (ORM) allows for separation of **application logic** (view functions) and **data storage/retrieval** (database)

DRY

(Don’t Repeat Yourself)

```
class Teacher(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    picture = models.ImageField()
    students = models.ManyToManyField(Student)
```

```
class Secretary(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    picture = models.ImageField()
    job_description = models.TextField()
```

avoid repetition in **model description** using **class inheritance**

https://docs.djangoproject.com/en/stable/topics/db/models/#model-inheritance

Option 1: Abstract base class

Staff inherits from models.Model

```
class Staff(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    picture = models.ImageField()
    class Meta:
        abstract = True
```

Model not used to create a database table

Both Teacher and Secretary inherit from Staff

```
class Teacher(Staff):
    students = models.ManyToManyField(Student)
```

```
class Secretary(Staff):
    job_description = models.TextField()
```

Option 2: Multi-table inheritance

Will have DB table for Staff with 1-to-1 relationship to Teacher and Staff tables

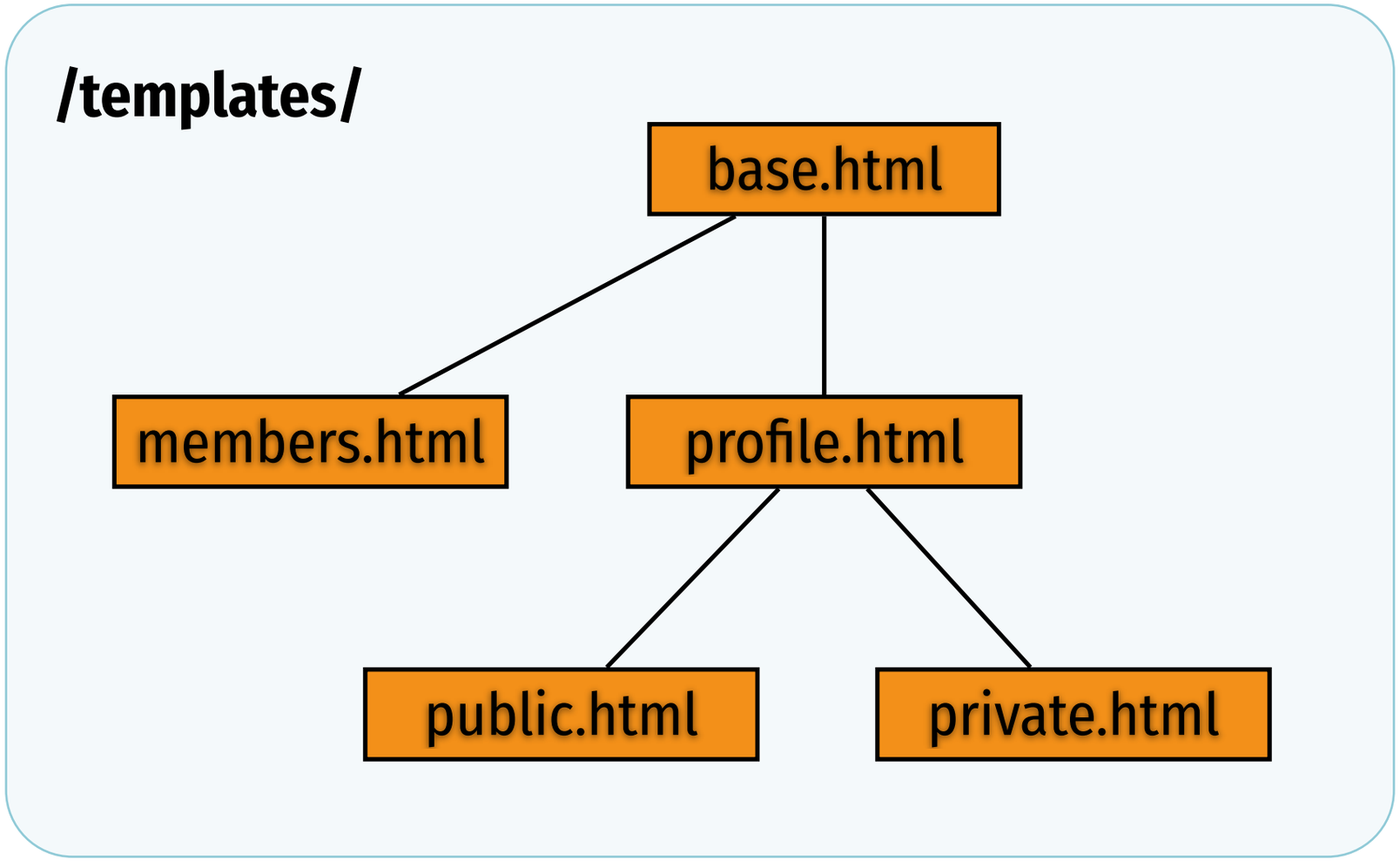
```
class Staff(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    picture = models.ImageField()
```

Both Teacher and Secretary inherit from Staff

```
class Teacher(Staff):
    students = models.ManyToManyField(Student)
```

```
class Secretary(Staff):
    job_description = models.TextField()
```

avoid repetition in **templates** using **template inheritance**



```
def profile(request):
    if 'username' in request.session:
        template = 'myapp/profile.html'
        return render(request,template,{})
    else:
        template = 'myapp/not-logged-in.html'
        return render(request,template,{})
```

```
def messages(request):
    if 'username' in request.session:
        template = 'myapp/messages.html'
        return render(request,template,{})
    else:
        template = 'myapp/not-logged-in.html'
        return render(request,template,{})
```

avoid repetition in **view functions** using **decorators**

```
def loggedin(view_function):
   def new_view_function(request):
      if 'username' in request.session:
         return view_function(request)
      else:
         template = 'myapp/not-logged-in.html'
         return render(request,template,context)
   return new_view_function
```

```
@loggedin
def profile(request):
   template = 'myapp/profile.html'
   return render(request,template,{})
```

```
@loggedin
def messages(request):
   template = 'myapp/messages.html'
   return render(request,template,{})
```

IT'S QUIZ TIME

https://cloudsurvey.co.uk

More on Django

Getting the most of…

- Django models
- Django admin
- Django forms
- Django commands
- Django template tags and filters

Django Models

Meta class and model properties

```
class Place(models.Model):
    type = models.ForeignKey(Type, models.CASCADE)
    name = models.CharField(max_length=300)
    phone = models.CharField(max_length=300)
    rating = models.IntegerField(default=5)

    class Meta:
        db_table = 'venue'
        ordering = ['type', 'name']

    @property
    def is_top(self):
        return self.rating >= 4
```

choose name of databse table

choose default ordering of places

this is a “defined” property, call as my_place.is_top

https://docs.djangoproject.com/en/stable/ref/models/options/

Class Meta Options

- Meta classes: Used to customise a class

```
class Member(models.Model):
    name = models.CharField(...)
    age = models.IntegerField(...)
    email = models.EmailField(...)

    class Meta:
      app_label = 'mainapp'
      db_table = 'member'
      ordering = ['name','-age']
      unique_together = ('name','age')
```

https://docs.djangoproject.com/en/stable/ref/models/options/

Django Admin

list display and list editable

NAME	PHONE	TYPE	RATING
Minister of Sound	090902909	Night club	5
Nandos	023423423	Restaurant	5

model inlines

Change type

Description: Night club

PLACES

NAME	PHONE	LOCATION LAT	LOCATION LON
Fabric			
Fabric	000000000	0.0	0.0
Minister of Sound			
Minister of Sound	090902909	2.0	2.0

list display

```
class PlaceAdmin(admin.ModelAdmin):
    list_display = ['name', 'phone', 'type', 'rating']
    list_editable = ['type']

class PlaceInline(admin.TabularInline):
     model = Place

class TypeAdmin(admin.ModelAdmin):
    inlines = [PlaceInline]

admin.site.register(Type, TypeAdmin)
admin.site.register(Place, PlaceAdmin)
```

list editable

model inlines

Django Forms

mainapp/forms.py

```
from django.forms import ModelForm

from .models import Place

class PlaceForm(ModelForm):
    class Meta:
        model = Place
        fields = ['name', 'type', 'phone', 'rating']
```

mainapp/views.py

```
if request.method == 'GET':
    places = Place.objects.all()
    context = {
        'title' : 'Welcome to Date Night!',
        'places' : places,
        'form' : PlaceForm(),
    }
    return render(request, 'mainapp/list.html', context)
```

mainapp/templates/mainapp/list.html

```
<form method='POST'>
    {% csrf_token %}
    {{ form.as_p }}
    <button class='btn btn-success'>Save</button>
</form>
```

Django Commands

mainapp/management/commands/cleandb.py

```python
from django.core.management.base import BaseCommand, CommandError
from mainapp.models import Place

class Command(BaseCommand):
   help = 'Fill places with empty phone number'

   def handle(self, *args, **options):
      places = Place.objects.all()
      for place in places:
         if place.phone == '':
            print('Need to find phone number for', place)
```

```
» python manage.py

Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[django]
    check
    compilemessages
    ...

[mainapp]
    cleandb
...
```

Custom Template Filters and Tags

Template Tags

{% **csrf_token** %}	*CSRF protection mechanism*
{% **extends** “parent.html” %}	*template extends parent.html*
{% **include** “snippet.html” %}	*renders template with current context*
{% **url** ‘url-name’ v1 v2 %}	*name URL, without domain name*
{% **cycle** x_1 x_2 … %}	*outputs one of x_i each time*
{% **now** “js \o\f F” %}	*4th of September*
{% **load** t1 t2 **from** mylib %}	*loads custom template tags/filters*

Template Tags

{% **block** <name> %} … {% **endblock** %}	*block to be overridden by child template*
{% **for** user **in** users %} … {% **endfor** %}	*for loop over items of given array*
{% **if** users %} … {% **endif** %}	*test if variable exists, is not empty, and is not a false boolean value*
{% **spaceless** %} … {% **endspaceless** %}	*removes whitespace between HTML tags*
{% **autoescape** on/off %} … {% **endautoescape** %}	*turn automatic escaping on and off*

Template Filters

value	Template	Output
"django"	{{ value|**capfirst** }}	"Django"
"I'm Fine"	{{ value|**lower** }}	"i'm fine"
"I'm fine"	{{ value|**cut**:" " }}	"I'mfine"
datetime obj	{{ value|**date**:"D d M Y" }}	Wed 09 Jan 2008
list of dictionaries	{{ value|**dicsort**:"key" }}	sorted list
["Paulo","Oliva"]	{{ value|**join**:"." }}	"Paulo.Oliva"
123456789	{{ value|**filesizeformat** }}	117.7 MB

Custom Template Filters

- You can write your own **template tags/filters**!
- Define these using Python
- Place them in **templatetags** folder

```
polls/
    __init__.py
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

Custom Template Filters

```
# templatetags/extras.py
from django import template
# custom filter: delete arg from the given string arg
def cut(value, arg):
    return value.replace(arg, '')
register = template.Library()
register.filter('cut', cut)
```

```
<!-- some template that uses custom filter -->
{% load extras %}
...
String sum without 0's: {{ sum|cut:"0" }}
```

https://docs.djangoproject.com/en/stable/howto/custom-template-tags/