Django 2 beginners guide

by www.pyplane.com

Version 1.0

PART 1: Theory & setup

Why is django cool?

My top 3 choices:

1. Python based web framework
2. MVT architecture
3. Django ORM

Why is it difficult to start?

My top 3 choices:

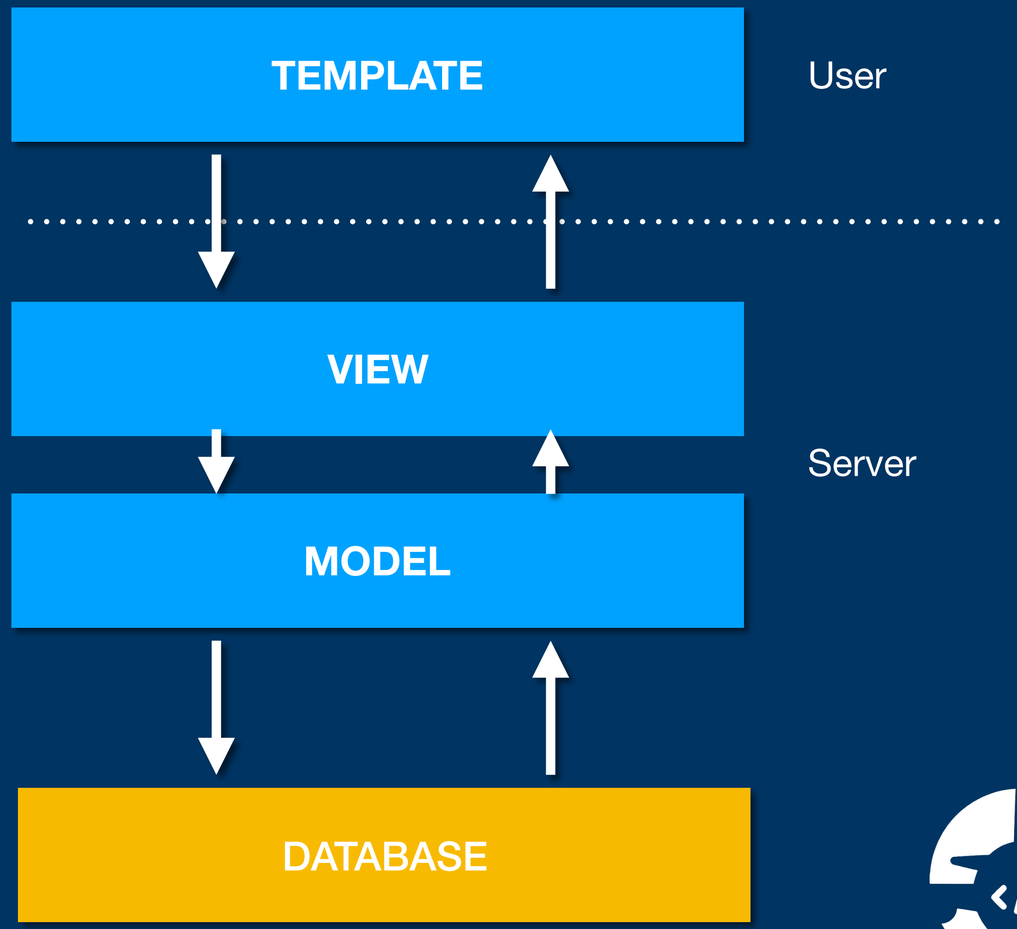
1. Several lines of code in different files just to write a simple „hello world”
2. The concept of MVT (at first may seem confusing)
3. Lack of easy tutorials that help you get started

Python packages

- Looking at django project you will often see __init__.py file. So let’s start with explaining why
- What is a Python package?
  - A Python package is simply an organized collection of python modules. A python module is simply a single python file.
- Why would I want to create a package using __init__.py?
  - Creating a package with __init__.py is all about making it easier to develop large Python projects. It provides an easy way for you to group large folders of many seperate python scripts into a single importable module

MVT

- Model View Template
- The MVT (Model View Template) is a software design pattern in Django's architecture that consists of three components:
  - Model: responsible for the database (ORM)
  - Template: responsible for the presentation layer & user interface
  - View: responsible for the business logic



ORM

- ORM stands for „Object Relational Mapping” and is a neat way to interact with the database
- Every class defined in the models file represents a database table
- Every defined property (field) represents a column in the table
- Each object is created based on a particular model, without having to write any SQL statement
- Advantages for this approach: simplified and cleaner solution, that is based on shorter and more readable querysets
- What is a Queryset? It is a list of objects of a particular model accessed from the database based on given criteria.

ORM

Relational database:

ID	TITLE	CATEGORY	PRICE	LENGTH
1	Movie1	Horror	29.00	120 min
2	Movie2	Thriller	39.90	150 min
3	Movie3	Comedy	24.99	110 min

Object:

Class Movie:
title = ”Movie1”
category = ”Horror”
price = 29.00
length = ”120 min”

Django ORM allows us to use django queries instead of using plain SQL statements, so i.e. if we want to access all the movies from the horror category we would use SQL:

SELECT * FROM Movies WHERE category=”Horror”

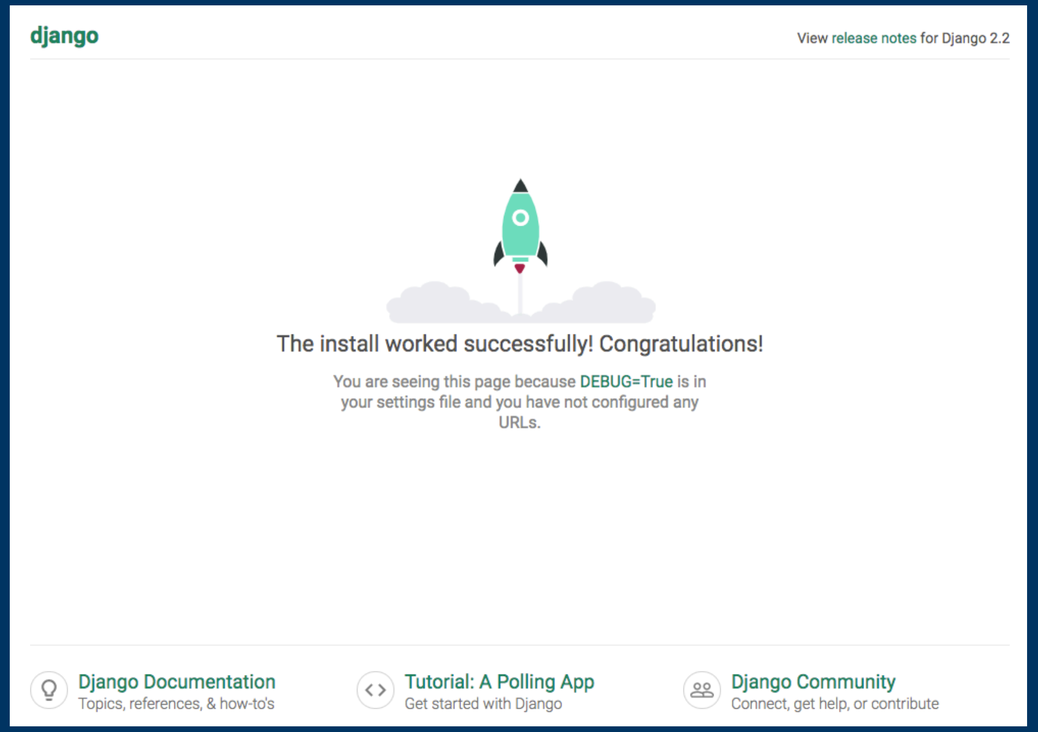
Or django queryset:

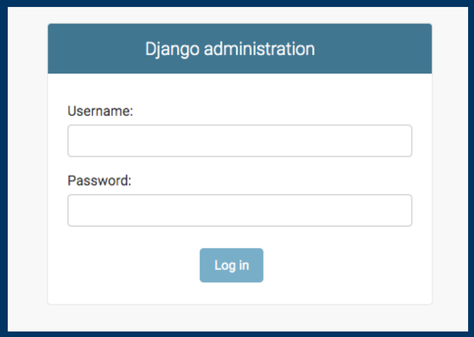
Movie.objects.filter(category=”Horror”)

Project setup 1

Main steps:

1. Choose project folder
2. In terminal run: **virtualenv name**
3. Enter the folder with virtual environment: **cd name**
4. Activate virtual enviroment : **source bin/activate**
5. Install Django : **pip install django==2.2**
6. Start Django project: **django-admin startproject projectname**
7. In terminal rename project file to src: **mv projectname src**
8. Go to the source folder : **cd src**
9. Do the migration : **python manage.py migrate**
10. Create super user : **python manage.py createsuperuser**
11. **Test the server: python manage.py runserver**
12. **Login to admin: in the browser type:** http://127.0.0.1:8000/admin/





Migrate vs makemigrations

REMARK! manage.py is a file which is added to each django project and is created after starting a project. It simply helps with the management of the site

python manage.py makemigrations - generates the sql commands to create the table of each class defined in the models.py

python manage.py migrate - command responsible for executing the commands above (applying migrations). It creates the tables in the database file.

Project setup 2

13. Open your editor and go to settings.py
14. Edit the templates adding „templates” name in the DIRS:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates'), ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                ...
            ],
        },
    },
]
```

15. Add the bottom of settings.py add static and media configuration:

```
STATIC_URL = '/static/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static_project")
]

STATIC_ROOT = os.path.join(os.path.dirname(
    BASE_DIR), "static_cdn", "static_root")

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(os.path.dirname(
    BASE_DIR), "static_cdn", "media_root")
```

Create this folder in src

(how os path works will be explained in the next slides)

16. In urls.py add add the top imports:

```
from django.conf import settings
from django.conf.urls.static import static
```

17. In urls.py at the bottom add:

```
urlpatterns += static(
    settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
urlpatterns += static(
    settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

18. Install you first app in the terminal:

python manage.py startapp appname

19. In settings.py in „INSTALLED_APPS” add the name of the created app

20. Run: **python manage.py collectstatic**

Project setup - remarks

- Static files are dedicated to javascript, css files, images while media files are meant for content that the user uploads
- We distinguished MEDIA_ROOT and STATIC_ROOT as seperate directories within the static_cdn folder
- To the STATIC_ROOT will be copied to this directory after running collectstatic command with the purpose to serve all the static files from a single location
- MEDIA_ROOT will be updated each time a user uploads something to the server
- STATICFILES_DIRS tells Django where to look for additional static files outside of the static folder in the application directory. If apps don’t contain static folders - it will be the place to store static files
- Please note that in order to use static files in html, you need to load static at the top and reference it in the place where you use it. See examples below:

```
{% load static %}
```

```
<div>
    <img src="{% static 'assets/img/tape.png' %}" width="300px" height="auto">
</div>
```

Django users

- A user is a person that simply can be authenticated
- The user can be set to active or not, but only active users are allowed to login.
- Any user assigned to staff status, can login to the admin pages and has no other privileges.
- A superuser is a user with all the permissions (has_perm will return True)

os.path

os.path.basename(path) : returns the name of the file from the given path.

```
import os
file = os.path.basename("/my_folder/my_file.txt")
print(file)

# output: my_file.txt
```

os.path.dirname(path) : returns the directory of the file with the name of the file

```
import os
file = os.path.dirname("/my_folder/my_file.txt")
print(file)

# output: /my_folder
```

os.path.join(path): returns a path merged out of multiple path components

```
import os
path = '/desktop'
print(os.path.join(path, "my_folder", "my_file.txt"))

# output: /desktop/my_folder/my_file.txt
```

Middleware

In very simple terms, middleware is pretty much like function in a view (we'll get there in part 2), that will be executed before and after rendering the view. It enables processing of requests and responses globally, so there is no need for customisation of every view

Examples of operations performed by Middlewares:

- HTTP authentication
- Session management
- Encryption
- Validation

settings.py

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Django middlewares require at least one of the following methods:
process_request, process_response, process_view, and process_exception

More informations in the documentation: https://docs.djangoproject.com/en/2.2/topics/http/middleware/

CRUD

The four main functions related to managing the data in the database:

1. Create
2. Read / Retrieve
3. Update
4. Delete / Destroy

PART 2: MVT

Models

- In part 1, we talked about the ORM concept. In this section we will take a look at practical aspects of using models.
- Models are defined in given apps in the file models.py
- Each class inherits from models.Model
- Example of a model:

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField(max_length=360)
    image = models.ImageField(upload_to='posts/img/',
                              validators=[validate_ext], blank=True, null=True)
    liked = models.ManyToManyField(
        Profile, default=None, blank=True, related_name="liked")
    updated = models.DateTimeField(auto_now=True)
    created = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(
        Profile, on_delete=models.CASCADE, related_name="author")

    def num_likes(self):
        return self.liked.all().count()

    def __str__(self):
        return str(self.title)

    def get_absolute_url(self):
        return reverse("posts:gp-detail", kwargs={"pk": self.pk})

    class Meta:
        ordering = (“-created",)
```

The model contains various fields (attributes) as well as methods:

- num_likes - to get the value of likes for particular post
- __str__ - to get the string representation of the object
- get_absolute_url - to access the the link of a particular post (object
- Using class Meta we defined ordering by „-created” which basically means from the newest to the oldest

Read more in the documentation on field types:
https://docs.djangoproject.com/en/2.2/ref/models/fields/

Documentation for models:
https://docs.djangoproject.com/en/2.2/topics/db/models/

Model relationships

We can distinguish 3 types of relationships

- Many-to-many relationship - many users can like many posts
- Many-to-one relationship - a particular comment can be related to one Post only, but many comments can be associated to a given Post
- One-to-one relationship - one user can have only one profile

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField(max_length=360)
    liked = models.ManyToManyField(
        Profile, default=None, blank=True, related_name="liked")
    updated = models.DateTimeField(auto_now=True)
    created = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(
        Profile, on_delete=models.CASCADE, related_name="author")
```

1) liked field is a many-to-many relationship example

```
class Comment(models.Model):
    user = models.ForeignKey(Profile, on_delete=models.CASCADE)
    post = models.ForeignKey(GeneralPost, on_delete=models.CASCADE)
    body = models.TextField(max_length=360)
    created = models.DateTimeField(auto_now_add=True)
```

2) post field is a foreignkey relationship example

```
class Profile(models.Model):
    user = models.OneToOneField(
        User, on_delete=models.CASCADE, blank=True, null=True)
    bio = models.CharField(max_length=220, blank=True)
    profile_picture = models.ImageField(
        blank=True, default='uploads/profile/profile.png', upload_to=get_file_path, validators=[validate_content])
```

3) user field is a one to one relationship example

Model Manager

- In Django, the Manager is the interface responsible for communicating with the database.
- Each model class by default, has a Manager added with the name „objects”. You can access the data by calling a queryset, i.e. Model.objects.all(), Model.objects.filter() or by defining your own Custom Manager
- In order to do so, the base Manager class must be extended and instantiated in your custom Manager defined in your model.
- Let’s look at an example

Model Manager - example

```python
class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=120)
    slug = models.SlugField(unique=True, blank=True)
    Body = models.TextField()
    draft = models.BooleanField(default=False)
    active = models.BooleanField(default=False)
    updated = models.DateTimeField(auto_now=True)
    created = models.DateTimeField(auto_now_add=True)

    objects = PostManager()

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse("posts:detail", kwargs={"slug": self.slug})

    class Meta:
        ordering = ('-created',)
```

Instantiation

```python
class PostManager(models.Manager):

    def get_all_active(self):
        return super().get_queryset().filter(draft=False)

    def get_all_published(self):
        return super().get_queryset().filter(created__lte=
timezone.now())
```

Now we can access the data with:

- Post.objects.get_all_active()
- Post.objects.get_all_published()

But we can't get all active and all published posts.
To make this work we need to create custom queryset methods

Filtering Querysets

- lte - search for is less than or equal
- gte - search for is greater than or equal
- lt - search for is less than
- gt - search for is greater than
- contains - search for contains and is case sensitive
- icontains - search for contains and is case insensitive
- exact - search for is exactly the same and case sensitive
- iexact - search for is exactly the same and case insensitive
- startswith - search for begins with
- endswith - search for ends with
- isnull - search is null as True or False

Examples:

```
Post.objects.filter(content__icontains=”cool”)
Post.objects.filter(user__isnull=True)
Post.objects.filter(title__startswith=”welcome”)
```

3 options to return a custom queryset

```python
    def get_all_active(self):
        return super().get_queryset().filter(draft=False)
```

```python
    def get_all_active(self):
        return self.filter(draft=False)
```

```python
    def get_all_active(self):
        return Post.objects.filter(draft=False)
```

All 3 options will return the same results. NOTE! When defining a Queryset class the methods in the Manager class will require adjustment

Manager & Queryset

```python
class PostQuerySet(models.QuerySet):

    def get_all_active(self):
        return self.filter(draft=False)

    def get_all_published(self):
        return self.filter(publish__lte=timezone.now())


class PostManager(models.Manager):

    def get_queryset(self):
        return PostQuerySet(self.model, using=self._db)

    def get_all_active(self):
        return self.get_queryset().get_all_active()

    def get_all_published(self):
        return self.get_queryset().get_all_published()
```

By defining additionally custom queryset methods, we can now chain the querysets, like this:

Post.objects.get_all_active().get_all_published()

Views

- After defining a model or many models in the particular app, we can now handle the business logic in our views that later on we will display in templates
- The first challenge will be to choose a view type from:
  1. Function Based Views
  2. Class Based Views
  3. Generic Class Based Views
- In this section we will have a overview of the positions mentioned above

Function Views

- In django function views take as a input a request and returns as an output a response
- The response can be a instruction how to render the context in the defined template, or it can be i.e. a simple HttpResponse (see example below), redirect instruction or page not found 404
- They can import models to access objects from the database according to the logic defined in the particular view, next the view is being imported to urls.py where it’s added to the path list (this point is related to function views as well as to class based views
- Let’s look at some basic examples:

```python
from django.http import HttpResponse
from datetime import datetime

def current_date(request):
    current_date = datetime.now().strftime("%Y/%m/%d/")
    text = "Todays date is {} ".format(current_date)
    return HttpResponse(text)
```

```python
def home_page(request):
    context ={
        "title":"hello world",
    }
    if request.user.is_authenticated:
        user = request.user.username
        context['user_info'] = "User {} Logged in".format(user)
    return render(request, 'main/index.html', context)
```

Class Based Views

from django.views import View
Import „from django.views.generic import NameOfView”
Import „from django.contrib.auth.views import LogoutView”

Class Based View is a django view that inherits directly from the View class
Generic Class Based Views are built-in django Class Based Views created in order to handle most common tasks (get list of objects, show detail of particular object, delete an object etc.

Advantages: easy and clean implementation + extension opportunities via mixins (multiple inheritance)

Most popular Class Views:

- View - the base class for Class Views (see example on the next slide)
- TemplateView - render particular template
- FormView - displays a form
- CreateView - create an object via form submit
- DetailView - displays the detail of the particular object
- ListView - displays collection of objects in the database according to particular queryset
- UpdateView - update an object via form submit
- LogoutView - logs out a particular user
- RedirectView - redirect to a particular URL
- DeleteView - deletes object from database (with confirmation view)

Most popular features: model, queryset, context_object_name, template_name, success_url, login_url, initial ???

Documentation: https://docs.djangoproject.com/en/2.2/topics/class-based-views/

Class Based Views - example

views.py

```
class PostExampleView(View):
    form_class = ExampleForm
    initial = {'name': 'example of name'}
    template_name = 'posts/example.html'

    def get(self, request):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request):
        form = self.form_class(request.POST)
        if form.is_valid():
            form.save()
            return redirect("posts:post-list")
        return render(request, self.template_name, {'form': form})
```

forms.py

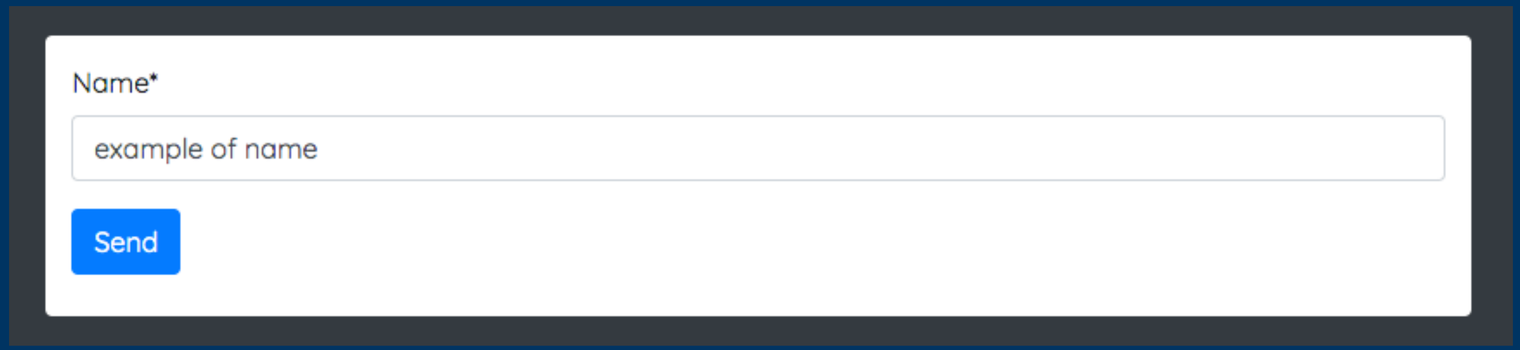
```
class ExampleForm(forms.ModelForm):
    class Meta:
        model = ExamplePost
        fields = '__all__'
```

- Our class inherits from „View” class
- We also defined initial value of field „name” defined in the model
- The output should look like this:

models.py

```
class ExamplePost(models.Model):
    name = models.CharField(max_length=220)
    created = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return str(self.name)
```



Path in urls.py

```
  path('example/', PostExampleView.as_view(), name="post-example"),
```

GCBV examples

ListView

```
class PostListView(ListView):
    model = Post
    template_name = posts/posts.html'
    paginate_by = 5
```

DetailView

```
class PostDetailView(DetailView):
    model = Post
```

- Instead of model we can define queryset i.e: **queryset = Post.objects.all()**
- You don’t need to define template_name if you name the file: **modelname_list.html** (ListView)
- You don’t need to define template_name if you name the file: **modelname_detail.html** (DetailView)

GCBV example - CreateView

```python
class PostCreateView(CreateView):
    form_class = PostForm
    template_name = 'posts/board.html'
    success_url = reverse_lazy('posts:post-list')
```

- This view displays a form defined in the form_class, on success redirects to the view defined in success_url and saves the object in the database. In case of errors - displays validation errors
- You don’t need to define template_name if you name the file: **modelname_form.html**

GCBV example - FormView

```python
class HomeView(FormView):
    template_name = 'reports/home.html'
    form_class = ReportSelectLineForm

    def get_form_kwargs(self):
        kwargs = super(HomeView, self).get_form_kwargs()
        kwargs['user'] = self.request.user
        return kwargs
```

- This view displays a form defined in the form_class, on success redirects to the view defined in success_url. In case of errors - displays validation errors. This view shouldn't be responsible for creating objects. The classical example for the use of the FormView is contact form for sending emails.
- get_form_kwargs is a method that passes data (as a dictionary) to the __init__ method of the given form.
- In the example above, the field user will be automatically filled with the value of the logged in user
- Later we will discuss methods form_valid and form_invalid methods

GCBV example - UpdateView

```
class ReportUpdateView(UpdateView):
    model = Report
    form_class = ReportForm
    template_name = 'reports/update.html'

    def get_success_url(self):
        return self.request.path
```

- In the example above we defined a model, form_class and a template_name for the view. The next step was to override success_url with the get_success_url method to redirect us to the current path (so we can see the updated fields)
- Later we will discuss more methods we can apply to this view

GCBV example - DeleteView

```python
class GeneralPostDelete(DeleteView):
    model = GeneralPost
    template_name = 'posts/confirm.html'
    success_url = reverse_lazy("posts:post-list")
```

- The template of the delete view consists of a form (as in the example below) to confirm if the object in fact should be deleted

In confirm.html :

```html
<form method="post">
    {% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input class="btn btn-primary" type="submit" value="Confirm delete">
</form>
```

CBV vs FBV

- The biggest advantage of Class Based Views is multiple inheritance that allows us to extend the code in a nice and clean way
- However, although the code itself looks nicer, it’s very often harder to read because the flow isn’t obvious (it’s divided into various methods within the given view class)

Business logic in a view: change object field (class attributes)

```
class Profile(models.Model):
    user = models.OneToOneField(
        User, on_delete=models.CASCADE, blank=True, null=True)
    bio = models.CharField(max_length=220, blank=True)
    active = models.BooleanField(default=False)
```

This will be changed for the given object

- Once you'll get a object from the database assigned to a variable you can change all the field properties defined in the model i.e.:

```
profile = Profile.objects.get(name="Luke")
profile.active = True
```

reverse vs reverse_lazy

- Before we go any further, we should discuss what is the concept behind reverse and what is the main difference between reverse and reverse_lazy
- Reverse is away of moving from one view to another by using the name of the view defined in the urls.py
- If we want to reference an app, we need to combine the app namespace with the name of the view: **appnamespace:viewname**

```
    path('messages/', include('user_messages.urls', namespace='messages')),
```

```
    path('write/', MessageCreateView.as_view(), name="message-write"),
```

Main urls.py

App urls.py

```
reverse_lazy('messages:message-write')
```

```
class MessageCreateView(LoginRequiredMixin, FormValidationMixin, CreateView):
    model = Message
    form_class = MessageModelForm
    template_name = "user_messages/new_message.html"
    success_url = reverse_lazy('messages:message-write')

    def form_invalid(self, form):
        messages.warning(self.request, "There was an error..")
        return super().form_invalid(form)
```

When using Generic Class Based Views with the „success_url” use reverse_lazy

When using functions/methods use reverse

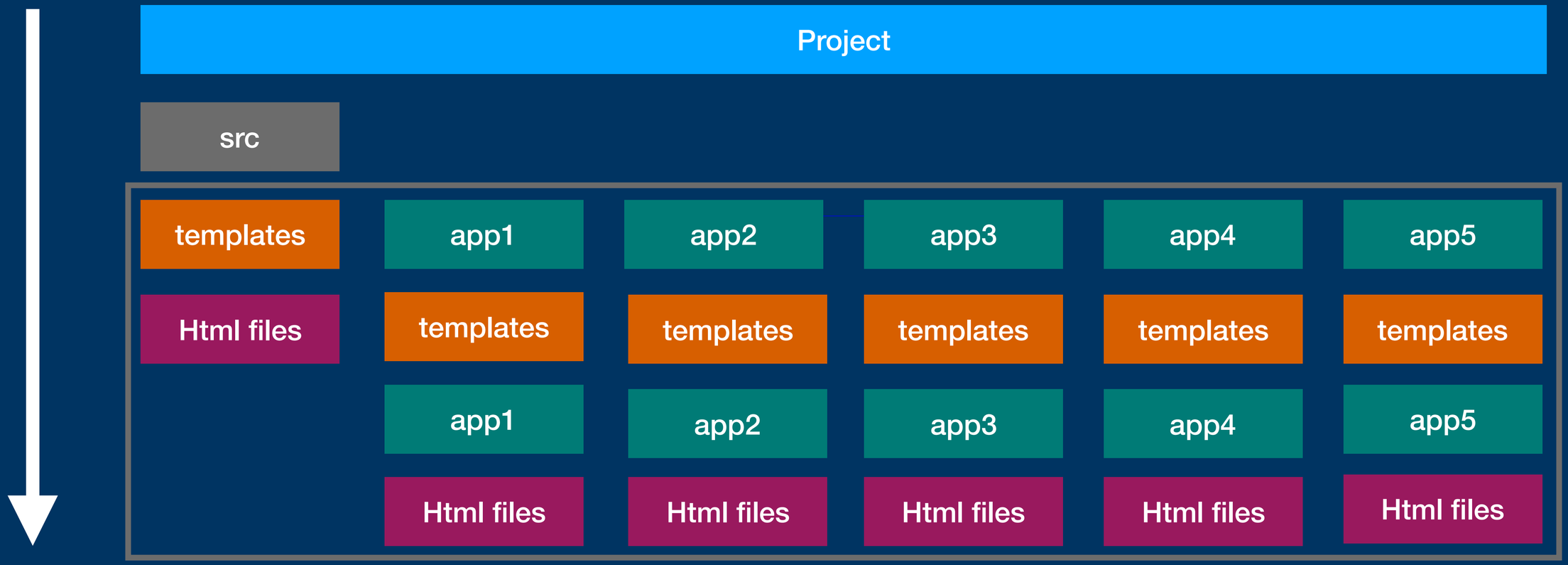
```
get_absolute_url(self):
        return reverse(„messages:message-detail", kwargs={"pk": self.pk})
```

Templates

- Inside the ”src” folder we need to create our main “templates” where we will define among others: base.html, navbar.html, footer.html
- Inside each app folder we should define a separate “templates” folder for views related to this app
- Inside of that folder we need to create another folder with the name of the app
- Finally, inside of that folder we can create our html files (templates) referenced in the views
- This way we avoid mistakes because we pass in the views a unique path, i.e.: **template_name = ”posts/posts_list.html”**

Templates

The structure of templates folders inside the project file



Templates

- The base.html template is the core template for others in the project
- It consists of html and python tags. See example below:

base.html

```
{% load static %}
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!--Custom css-->
    <link rel="stylesheet" href="{% static 'assets/css/style.css' %}">
    <!--bootstrap css & js-->
    <link href="//maxcdn.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" rel="stylesheet" id="bootstrap-css">
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>

    <title>Project Name | {% block title %} {% endblock title %}</title>
  </head>
  <body>
  {% include 'navbar.html' %}

    {% block content %}
    {% endblock content %}

  {% include 'footer.html' %}

    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
  </body>
</html>
```

Within these blocks we will write html code for every file that will extend base.html

Templates

- In the previous slide we defined „block” tags that together with „extends” tag are part of a powerful feature in django called template inheritance.
- To extend simply means that in particular template that inherits from base.html (i.e. home.html) we have all the code defined in the „parent” file + we can write some additional code within the defined blocks
- Let’s look at a simple example:

```
{% extends 'base.html' %}

{% block title %} home {% endblock title %}

{% block content %}

<h1> Hello World </h1>

{% endblock content %}
```

```
{% extends 'base.html' %}

{% block title %} contact {% endblock title %}

{% block content %}

<h1> Contact us </h1>

{% endblock content %}
```

- The code above

PART 3: Working with django

Django admin

- Most popular features:
  A. list_display - fields available in the preview
  B. Fields - fields displayed in the admin to which we have direct access
  C. list_display_links - fields that are hyperlinks to access the object
  D. Ordering - rule for listing in order objects in the admin
  E. search_fields - what fields can be used to filter the data

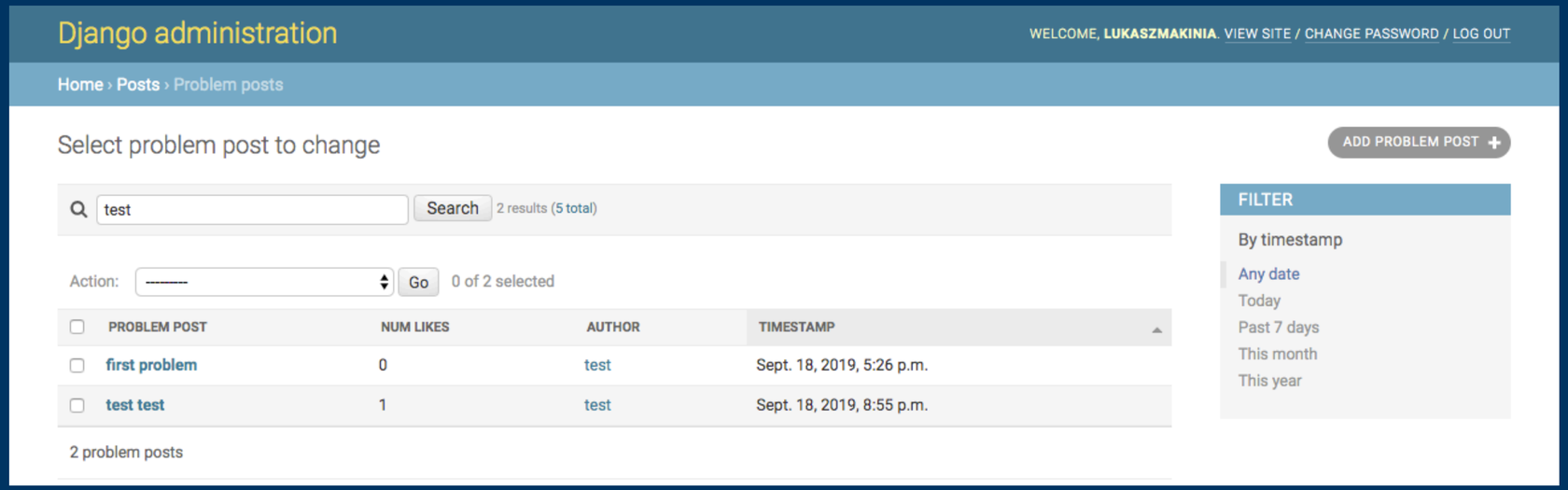
admin.py

```
class ProblemPostModelForm(admin.ModelAdmin):
    list_display = ['__str__', 'num_likes', 'author', 'timestamp']
    fields = ('liked', 'author', 'report', 'problem_reported')
    list_display_links = ('__str__', 'author')
    list_filter = ('timestamp',)

    ordering = ('timestamp',)
    search_fields = ('author__user__username',)

    class Meta:
        model = ProblemPost

admin.site.register(ProblemPost, ProblemPostModelForm)
```



Documentation: https://docs.djangoproject.com/en/2.2/ref/contrib/admin/

Customizing CBV

- get() - the purpose of this method is to write some business logic for a view and process initial data before the template is rendered
- get_object() - related to DetailView with the purpose to select particular object to be attached and passed with the context
- get_context_data() - responsible for sending the data to the template. You can add additional querysets and process some data according to the defined logic
- get_queryset() - related to the ListView and responsible for retrieving all the data from the database according to the defined querysets. Instead of returning all the objects, you can override this method to retrieve objects by given criteria i.e. in the filter method or methods defined in the custom Manager

Querysets methods

Here are some popular methods for querysets:

- all() - default method on a queryset, returns all objects of the given model
- get() - returns an object according to given lookup parameters
- filter() - returns objects according to given lookup parameters
- exclude() - returns a queryset of objects that don’t match the given parameters
- create() - creates an object according to given parameters
- get_or_create() - explained on a seperate slide
- order_by() - with this method you can override ordering defined in the class Meta. If you use ”?” as a parameter, objects will be returned randomly. I.e. Post.objects.order_by(”?”)
- aggregate() - aggregate a collection of objects i.e. (summarize, calculate average). Example: Report.objects.aggregate(Sum(’plan'))
- count() - returns the quantity of objects. I.e. Post.objects.all().count()
- distinct() - used to eliminate duplicates
- values() - returns a queryset of dictionaries
- value_list() - returns a queryset of list of tuples

Documentation: https://docs.djangoproject.com/en/2.2/ref/models/querysets/

Get or create

- Get or create is a simplified method for looking up an object according to given kwargs
- It returns a tuple of: object and created
- If object exists, created will equal to True and the object we are looking for will be returned
- Otherwise created will return False, and a object will be created
- It’s a simplified version of running the following code:

```
try:
    obj = Post.objects.get(title="post1")
except Post.DoesNotExist:
    Post.objects.create(title='Post1', description='content for post1', active=True)
```

- The same can be accomplished running single line of code

```
obj, created = Post.objects.get_or_create(title='Post1', description__icontains=’content for post1', active=True)
```

get_object_or_404

- Instead of using i.e. Report.objects.get(pk=1) which should be wrapped in try, except blocks (otherwise DoesNotExist exception will be raised), there is an alternative called get_object_or_404:

```
report = get_object_or_404(Report, pk=1)
```

- If the object isn't found it raises Http404
- This approach with get():

```
try:
    report = Report.objects.get(pk=1)
except Report.DoesNotExists:
    raise Http404("nothing here...")
except:
    pass
```

Context processors

- A context processor has a very simple interface: It’s just a Python function that takes one argument, an HttpRequest object, and returns a dictionary that gets added to the template context. Each context processor must return a dictionary and can be used in all templates in the project

context_processors.py

```python
def profile_pic(request):
    if request.user.is_authenticated:
        user = request.user
        try:
            profile_obj = Profile.objects.get(user=user)
            pic = profile_obj.profile_picture
        except Profile.DoesNotExist:
            pic = None
        return {"picture": pic}
    return {}
```

In all templates you can use it now as {{ picture }}

```html
<img class="img-avatar" src="{{ picture.url }}" alt="Avatar" class="avatar">
```

In settings.py add the new context processor using the pattern:
appname.context_processors.function_name

```python
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates'), ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                ...
                'profiles.context_processors.profile_pic',
                ...
            ],
        },
    },
]
```

Django forms: forms vs modelforms

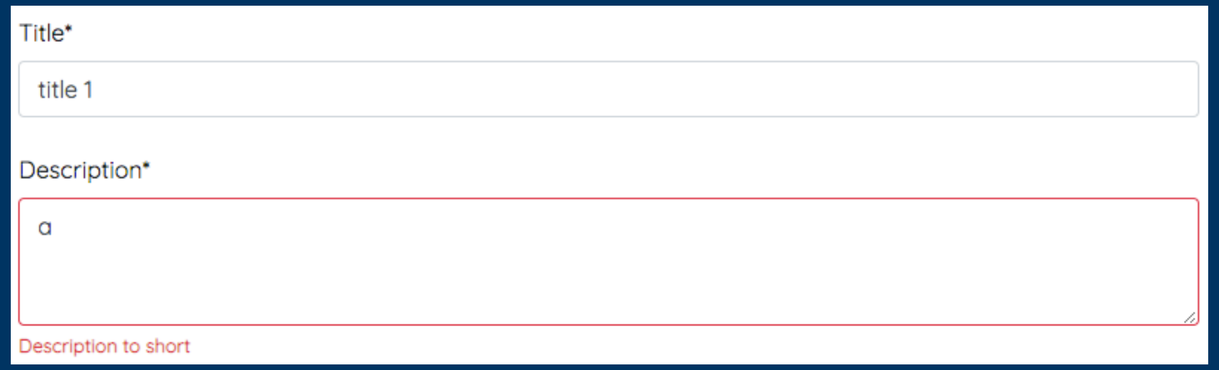
- Form - form created regardless of the model. Fields are created from scratch by the programmer - being manually defined without not having any relation to the database. We should use them i.e. to do searching/filtering, subscriptions, send emails etc.
- ModelForm - creates a form based on the selected fields from a given model. Recommended when we want to save an object to the database, i.e. : save a post or comment
- Django provides built-in validation for each field of the form that is defined in the given Form class
- For form validation are responsible methods: form.is_valid() for function views and form_valid() / form_invalid() for class views

Custom validation

You can perform custom form validation in forms.py, models.py or in validators.py (next page)

1) In forms.py

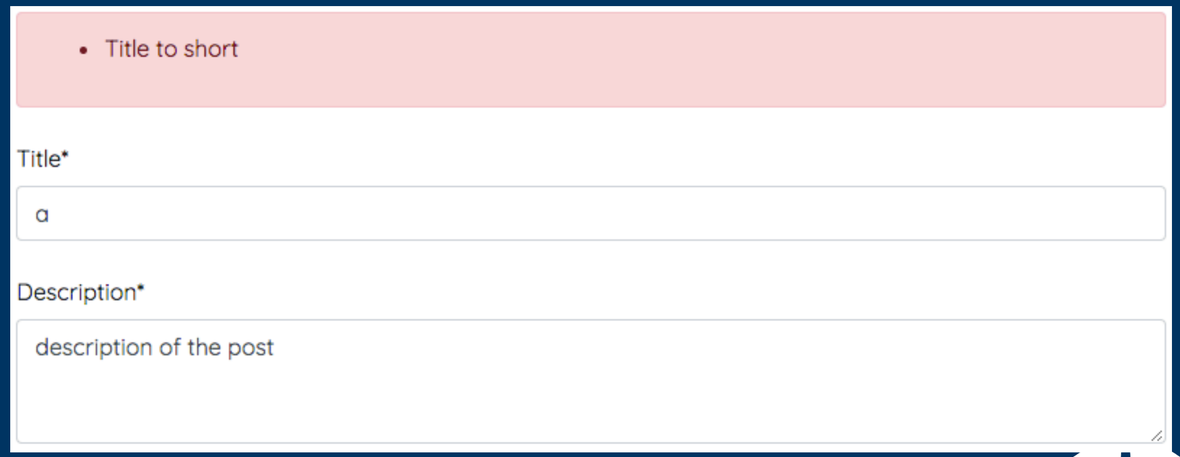
```
def clean_description(self):
        desc = self.cleaned_data.get('description')
        if len(desc) < 10:
            raise ValidationError("Description too short")
        return desc
```



Validation of a specific field - methodName_fieldName

2) In models.py

```
    def clean(self) :
        if len(self.title) < 5:
            raise ValidationError("Title too short")
        return super(GeneralPost, self).clean()
```



Non field validation - use clean method without specifying the field name

Custom validation

3) as validators

Import FileExtensionValidator and defined in the field of a model all the allowed formats (as a list) like in the example below

Reference in the model:

models.py

```
from django.core.validators import FileExtensionValidator

class GeneralPost(Post):
    title = models.CharField(max_length=200)
    description = models.TextField(max_length=360)
    image = models.ImageField(upload_to=get_upload_path,
                              validators=[FileExtensionValidator(['png', 'jpg', 'jpeg'])], blank=True, null=True)
```

Or create custom file extension validation: 1) using os 2) using magic
Note: you need to create validators.py file within the app and import created function inside this file to your models

1) Need to import os

```
def validate_ext(value):
    ext = os.path.splitext(value.name)[-1]
    ext_allowed = ['jpg', 'jpeg', 'png']
    if not ext.lower() in ext_allowed:
        raise ValidationError('This image format is not allowed')
```

OR

2) Need to install via brew magic: brew install lib magic, and import magic

```
def validate_ext(value):
    filetype = magic.from_buffer(value.read())
    ext_allowed = ['jpg', 'jpeg', 'png']
    ext_list = []

    for ext in ext_allowed:
        if ext in filetype.lower():
            ext_list.append(ext)

    if len(ext_list) == 0:
        raise ValidationError('This image format is not allowed')
```

Forms in templates

- Django has bulit-in protection functionality for handling requests to prevent from Cross Site Request Forgeries (CSRF)
- An CSRF vulnerability enables the attacker to perform some important action (i.e. sending the attacker money) without the knowledge of the victim
- Imagine you are in a hotel. You walk out to do some sightseeing and in the meantime your room is being cleaned. The door is open and basically anybody could walk in and say that they forgot something (money, laptop etc.).
- In simple words, wee need to make our requests secure and assure that they are coming from the actual users
- CSRF token is the way to do that. This is a random, basically impossible to guess string which is generated by the server and added to the forms as hidden field. It is assigned to each request to the server with it's value stored in a users session
- After the form is submitted but before rendering the view the server compares the values of the csrf tokens send by the form and the one remembered by the server
- By default, django checks for csrf token with every POST request and it verifies the token before rendering the view.
- To reference the example above, the difference with having the CSRF token would be that cleaning lady at the door entrance would have to verify if the person who wants to enter the room is really you.

Example of adding csrf token to the form:

```
<form action="." method="POST">
            {% csrf_token %}
            {{ form|crispy }}
            <button type="submit">Send</button>
        </form>
```

Documentation: https://docs.djangoproject.com/en/2.2/ref/csrf/#how-it-works

Form with initialized value

To pass a parameter into a FormView you need to override get_form_kwargs method

```python
class HomeView(FormView):
    template_name = 'reports/home.html'
    form_class = ReportSelectLineForm

    def get_form_kwargs(self):
        kwargs = super(HomeView, self).get_form_kwargs()
        kwargs['user'] = self.request.user
        return kwargs
```

```python
class ReportSelectLineForm(forms.Form):

    production_line = forms.ModelChoiceField(queryset=ProductionLine.objects.none())

    def __init__(self, user, *args, **kwargs):
        self.user = user
        super().__init__(*args, **kwargs)
        self.fields['production_line'].queryset = ProductionLine.objects.filter(team_leader__user__username=user)
```

Mixins

- Mixins are classes which are added to class views that contain some methods and attributes
- Using mixins is possible thanks to multiple inheritance - we can extend our class view with multiple mixins that help us organize our code
- Mixins can be extremely useful when we need to reuse certain code among multiple class views (instead of writing the same code in several views, we can simply „attach it” with mixins)
- The simplest example is the use of one of built-in mixins which handles login requirements (LoginRequiredMixin). Let’s have a look:

```python
class ProblemDetailPost(LoginRequiredMixin, DetailView):
    model = ProblemPost
    template_name = 'posts/detail.html'
    pk_url_kwarg = 'pk1'
    login_url = reverse_lazy('account_login')
```

```python
class GeneralPostDelete(LoginRequiredMixin, DeleteView):
    model = GeneralPost
    template_name = 'posts/confirm.html'
    success_url = reverse_lazy("posts:post-list")
```

Documentation: https://docs.djangoproject.com/en/2.2/topics/class-based-views/mixins/

Mixins

Mixins defined in the „Improveo project” (1):

```python
class FormUserOwnerMixin(object):
    def form_valid(self, form):
        profile = Profile.objects.get(user=self.request.user)
        if form.instance.author == profile:
            return super(FormUserOwnerMixin, self).form_valid(form)
        else:
            form.add_error(None, "You have to be the owner of this post to modify it")
            return self.form_invalid(form)
```

```python
class FormUserRequiredMixin(object):
    def form_valid(self, form):
        if self.request.user.is_authenticated:
            profile = Profile.objects.get(user=self.request.user)
            form.instance.author = profile
            return super(FormUserRequiredMixin, self).form_valid(form)
        else:
            form.add_error(None, "user must be logged in")
            return self.form_invalid(form)
```

Adding non-field error to the form

This mixins provides extra functionality that checks if the user is the author of the form (post)

This mixin checks if the user is logged in and if so, it assigns to the field „author” of the Post model the value of profile (retrieved based on the user that is logged in). If user is not authenticated, it displays non field error

Mixins

Mixins defined in the „Improveo project” (2):

```python
class FormValidationMixin(object):
    def form_valid(self, form):
        profile = Profile.objects.get(user=self.request.user)
        if form.instance.message_to == profile:
            form.add_error('message_to', "Can't send a message to yourself")
            return self.form_invalid(form)
        else:
            form.instance.message_from = self.request.user
            messages.success(self.request, "Message send")
            return super().form_valid(form)
```

Adding error to the form field message_to

This mixins checks if we aren’t by accident trying to send a message to ourselfs. If yes, the message is not send and an error is displayed. Otherwise, we receive a confirmation (success message) that the message was send.

+ LoginRequiredMixin - built-in mixing imported from django.contrib.auth.mixins

This mixin requires user to logged in in order to display the view

Messages

- We use the django messages framework to display notifications, i.e.: if we want to confirm that the message was send, or a post has been created
- We can distinguish several types of messages: debug, success, warning, info, error

Add conditional logic on how messages should be executed in the view:

```
def review_create(request):
    form = PostForm(request.POST or None, request.FILES or None)
    if form.is_valid():
        instance = form.save(commit=False)
        instance.title = form.cleaned_data.get('title')
        instance.content = form.cleaned_data.get('content')
        instance.save()
        messages.success(request, "Review created")
    elif form.errors:
        messages.error(request, "There was an error")

    form = PostForm()
    context = {
        'form': form,
    }
    return render(request, 'main/post_create.html', context)
```

Remember to do the import:

```
from django.contrib import messages
```

Display only success message in template:

```
{% if messages %}
    {% for message in messages %}
        {% if 'success' in message.tags %}
        <div class=„form-control”>{{ message }}</div>
        {% endif %}
    {% endfor %}
{% endif %}
```

Documentation: https://docs.djangoproject.com/en/2.2/ref/contrib/messages/

Signals

- Signals in Django allow us to send to a specific application information on a defined event that took place. Often used while creating/ modifying one of the models to execute some action.
- Mos popular signals:
  1. pre_save
  2. post_save
  3. m2m_changed

Example: in our site every time a user is created we want to automatically create a profile for this user, containing some additional informations (bio, profile picture etc.)

Documentation: https://docs.djangoproject.com/en/2.2/topics/signals/

Signals: setup

App name : profiles

apps.py

```
from django.apps import AppConfig


class ProfilesConfig(AppConfig):
    name = 'profiles'

    def ready(self):
        import profiles.signals
```

__init__.py

```
default_app_config = 'profiles.apps.ProfilesConfig'
```

signals.py

```
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import Profile


@receiver(post_save, sender=User)
def post_save_create_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

Inside the app you will be operating with 3 files:

- __init__.py - already exists
- apps.py * - already exists
- signals.py - need to create

* apps.py is a file created in order to help with the configuration of the app. Based on the app name, the AppConfig class is being creted (in our case profiles -> ProfilesConfig)

As an alternative you can define the signals directly in the models.py (not recommended)

Signals: pre_save

```python
from django.utils.text import slugify

@receiver(pre_save, sender=Post)
def post_pre_save_receiver(sender, instance, *args, **kwargs):
    words = instance.description
    words_count = len(words.split(" "))
    instance.length = words_count

    slugy = slugify(instance.title)[:100]
    qs = Post.objects.filter(slug=slugy)
    instance.slug = '{}'.format(slugy)
```

This example allows us to calculate number of words in a post and add it to the object field before the save of the model. Every time we create/modify a post we will calculate the amount of words and update the right field thanks to pre_save signal

Signals: post_save

```python
import random

el = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
     'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

def create_problem_id():
    random.shuffle(el)
    code = [str(x) for x in el[:12]]
    str_code = ''.join(code)

    return str_code
```

We have a random code generator function stored in the utils.py file

```python
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import ProblemReported, Report
from .utils import create_problem_id

@receiver(post_save, sender=ProblemReported)
def post_save_problem_id(sender, instance, created, *args, **kwargs):
    if created:
        if instance.problem_id is None:
            instance.problem_id = create_problem_id()
            instance.save()
```

We check if the created parameter is True, so if the ProblemReported object is created we will assign this randomly
generated code to the problem_id field after the models is saved

Filters custom filters

Example: Calculate estimated time of reading particular article

1) Create „templatetags” folder inside the app where you want to apply your own filter

2) Inside the „templatetags” folder create an empty „__init__.py” file and a another with the name of the filter, in our case - timely.py

3) Write down the code on the right

4) Appy in the template:
   - First load: {% load timely %}
   - Than execute: {{ object.content | timely }}

```python
from django import template

register = template.Library()


@register.filter
def timely(value):

    time = value/60
    real_time = round(time * 0.7)
    return "{} minutes".format(real_time)
```

PART 4: Other practical topics

Popular imports

Some of popular and useful imports:

- from django.views.generic import CreateView, ListView, DetailView …
- from django.contrib import messages
- from django.http import HttpResponseRedirect
- from django.contrib.auth.mixins import LoginRequiredMixin
- from django.shortcuts import render, redirect, reverse, get_object_or_404
- from django.urls import path, include, reverse_lazy
- from django.core.validators import FileExtensionValidator
- from django.core.exceptions import ValidationError
- from django.db.models.signals import post_save, pre_save, m2m_changed
- from django.dispatch import receiver
- from django import template
- from django import forms
- import uuid
- import os
- from django.conf import settings
- from django.contrib.auth.models import User
- from django.utils.deprecation import MiddlewareMixin
- from django.urls import resolve
- from django.http import Http404

Custom middleware

A simple example to restrict the view of the board (post-list). If the ip is equivalent to localhost and the path name is post-list Http404 will be raised

custommiddleware.py

```python
from django.utils.deprecation import MiddlewareMixin
from django.urls import resolve
from django.http import Http404

class CustomMiddlewareExample(MiddlewareMixin):
    def process_request(self, request):
        ip = request.META.get('REMOTE_ADDR')
        path = resolve(request.path).url_name
        if ip == '127.0.0.1' and path=="post-list":
            raise Http404("No page for this user")
        else:
            print('something')
            return None
```

```python
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    ,django.middleware.clickjacking.XFrameOptionsMiddleware',

    'posts.custommiddleware.CustomMiddlewareExample',
]
```

appName.fileName.className

Merging Querysets

```
qs1 = ProblemPost.objects.public_only()
qs2 = GeneralPost.objects.all()
qs = sorted(chain(qs1, qs2), reverse=True,
                      key=lambda obj: obj.created)
```

The example above shows a mechanism for merging 2 querysets and displaying them in reverse order by

www.pyplane.com