Python Has Power

Paweł Ostrowski

Przemysław Warchoł

STX Next, Wrocław, 15.06.2019

Agenda

1. INTRODUCTION
2. PROJECT INITIALIZATION
3. VIEWS AND URLS
4. MODELS AND ORM
5. TEMPLATES
6. FORMS
7. EXERCISE
8. TESTS

Framework

- Application skeleton
- Abstraction, where an apps behaviour is modified by user-written code
- Defines the app structure and mechanisms
- Enforces conventions and often software design patterns

What is Django?

- Open-Source, high level web framework written in Python
- Uses the MVT (Model-View-Template) pattern, very similar to MVC
- BSD License, maintained by Django Software Foundation
- Battery-included
  - Own ORM and templating language
  - Database migrations
  - Included code for authentication and authorization, security, routing, forms, etc.

Short history of Django

- Created by web programmers of Lawrence Journal-World newspaper in 2003
- Publicly released in 2005 (BSD License)
- Transformed to Django Software Foundation in 2008
- **Current stable release: 2.2.2**
  - **Supports Python 3.6, 3.7**
  - The last version to support Python 2.7 is Django 1.11 LTS

Usage

- Great for writing big, monolithic web apps
- Pretty good at writing small API-driven apps
- Used by a lot of companies, including:
  - Instagram,
  - Pinterest,
  - Disqus,
  - OpenStack,
  - Mozilla,
  - Spotify

Project initialization

To make this workshop easy, please use python3 venv throughout our entire project

```
$ sudo apt-get install python3-venv
...
$ python3 -m venv ~/.virtualenvs/php-advanced
...

$ source ~/.virtualenvs/php-advanced/bin/activate

(php-advanced) $ python --version
Python 3.6.8
```

Install required packages

Let's install Django and all it's basic dependencies

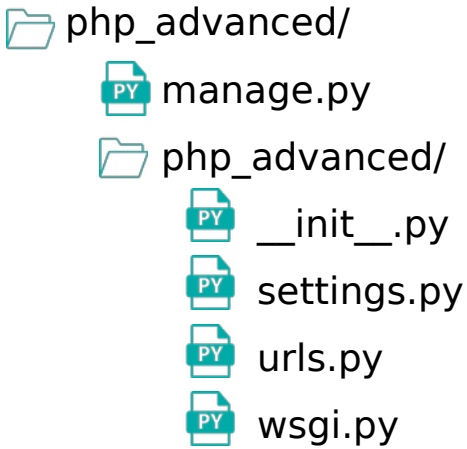
```
(php-advanced) $ pip install Django
Collecting Django
(...)
Successfully installed Django-2.2.2 pytz-2019.1 sqlparse-0.3.0

(php-advanced) $ pip freeze
Django==2.2.2
pkg-resources==0.0.0
pytz==2019.1
sqlparse==0.3.0
```

Setup the project

Bootstrap a django project using the tools provided with django

```
(php-advanced) $ django-admin startproject php_advanced
(php-advanced) $
```

Project structure



Let's start!

Django includes a development server which we'll use throughout this workshop

```
(php-advanced) $ cd php_advanced
(php-advanced) php_advanced $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply the
migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

Django version 2.2.2, using settings 'php_advanced.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Apply migrations

Let's apply the default ones, we'll get back to migrations later

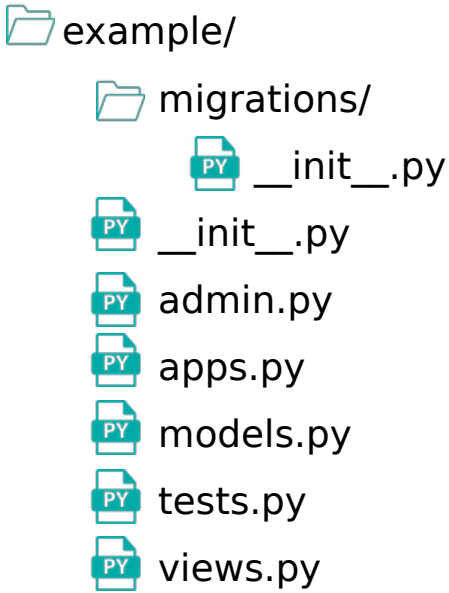
```
(php-advanced) php_advanced $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  ...
  Applying sessions.0001_initial... OK
```

Create an application

Django project consists of smaller django apps and configurations that tie them together

```
(php-advanced) php_advanced $ python manage.py startapp example
(php-advanced) php_advanced $
```

Default application structure

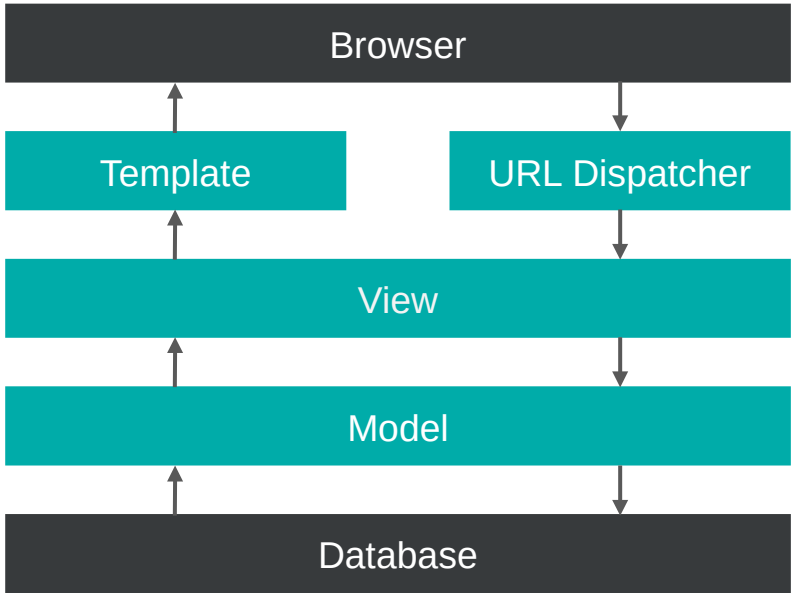


Configuring application

Django needs to know about our application, let's add it to php_advanced/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # (...)
    'example',
]
```

Django architecture (basic components)



Urls & views

URL (Uniform Resource Locator)

http://example.com/post/123/title/

The URL consists of the Protocol, Domain and Path.

A view function (or simply: a view) is a function that takes a web request, and returns a web response.

In its most basic form, it's a function that is pointed at by a defined URL. It should return an instance of HttpResponse object (or anything that inherits from it).

Writing a simple view

Let's write a simple “Hello World” view in example/views.py

```
from django.http import HttpResponse


def hello_world(request):
    return HttpResponse('Hello World!')
```

Pointing urls at views

Let's point an url at our view in php_advanced/urls.py

```
from django.urls import path
from django.contrib import admin

from example import views as example_views

urlpatterns = [
    path('hello/', example_views.hello_world),
    path('admin/', admin.site.urls),
]
```

Writing a (less) simple view

Let's write a simple “Hello {name}” view in example/views.py

```
from django.http import HttpResponse


def hello_world(request):
    return HttpResponse('Hello World!')

def hello_name(request, name):
    return HttpResponse(f'Hello, {name}!')
```

Pointing (another) url at our views

Let's point an url at our view in php_advanced/urls.py

```
from django.paths import path
from django.contrib import admin

from example import views as example_views

urlpatterns = [
    path('hello/', example_views.hello_world),
    path('hello/<str:name>/', example_views.hello_name),
    path('admin/', admin.site.urls),
]
```

Models

A model is a single source of information about your data.

Some basic information:

- Each model is a Python class that inherits from django.db.models.Model
- Each attribute of the model represents a database field
- If done this way, Django gives you access to an automatically generated database access API

Models

Create your first model in php_advanced/example/models.py

```
from django.db import models


class GiftList(models.Model):
    name = models.CharField(max_length=64)
    created_on = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
    guests = models.IntegerField()


    def __str__(self):
        return f'{self.name} for {self.guests} guests'
```

Migrations

A migration is a set of instructions on how to create a database schema based on a model

Create a migration file:

```
(php-advanced) php_advanced $ python manage.py makemigrations example
Migrations for 'example':
  example/migrations/0001_initial.py
    - Create model GiftList
```

and apply it to the database.

```
(php-advanced) php_advanced $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, example, sessions
Running migrations:
  Applying example.0001_initial... OK
```

Models

Let's extend php_advanced/example/models.py by adding a new model

```
class Gift(models.Model):
    name = models.CharField(max_length=128)
    gift_list = models.ForeignKey(GiftList, on_delete=models.CASCADE)

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

Model fields

django.db.models provides various fields, that are used to represent columns of various data types in the database:

- models.BooleanField
- models.CharField
- models.DateTimeField
- models.IntegerField
- models.TextField
- models.ForeignKey
- models.ManyToManyField

Migrations

Don't forget about the migration

```
(php-advanced) php_advanced $ python manage.py makemigrations example
Migrations for 'example':
  example/migrations/0002_gift.py
    - Create model Gift

(php-advanced) php_advanced $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, example, sessions
Running migrations:
  Applying example.0002_gift... OK
```

Django admin

Django provides an automatic admin interface. It's enabled by default but you have to create an user to be able to log in.

```
(php-advanced) php_advanced $ python manage.py createsuperuser
Username (leave blank to use 'python-hero'): admin
Email address: admin@example.com
Password:
Password (again):
Superuser created successfully.


(php-advanced) php_advanced $ python manage.py runserver
```

Django admin

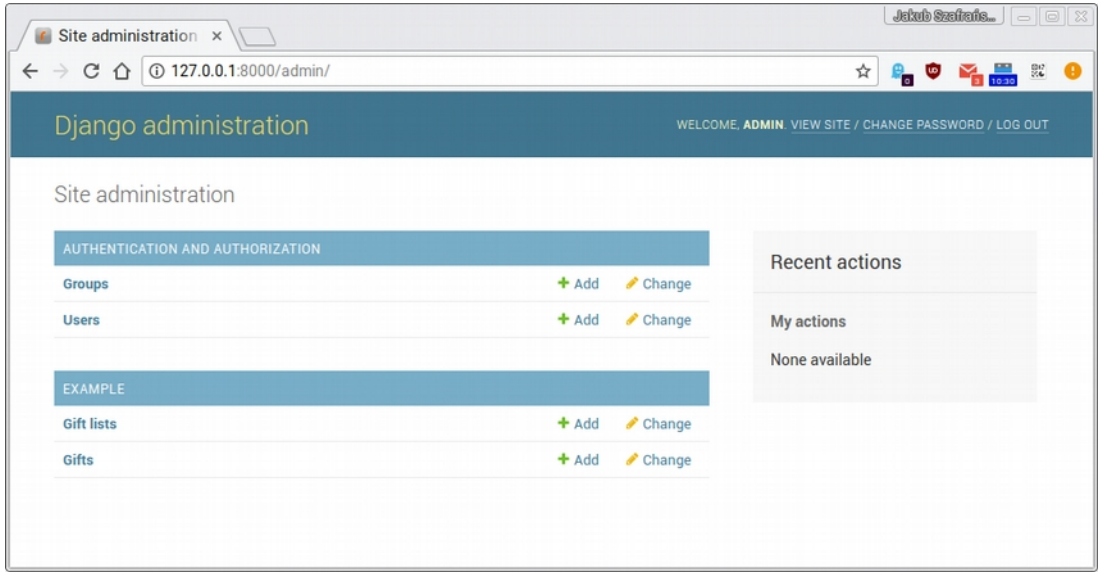
Where are my models? You have to register these in

php_advanced/example/admin.py

```
from django.contrib import admin
from example.models import GiftList, Gift

admin.site.register(GiftList)
admin.site.register(Gift)
```

Django admin

You now should be able to log in to the admin panel at http://127.0.0.1:8000/admin/



Nestled forms in Django Admin

Change a little bit of code in php_advanced/example/admin.py

```
from django.contrib import admin
from example.models import GiftList, Gift

class GiftInLine(admin.TabularInline):
    model = Gift
    extra = 1

@admin.register(GiftList)
class GiftListAdmin(admin.ModelAdmin):
    inlines = (GiftInLine, )

# admin.site.register(GiftList, GiftListAdmin)
admin.site.register(Gift)
```

ORM

Object-relational mapping is a technique used to represent object-oriented architecture to a relational database (or some other system of relational character).

Django ORM gives you access to a rich API that can be used to manipulate data stored in a relational database.

ORM

Let's try out the ORM by using django interactive shell.

First you need import our models:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>>
```

ORM

Let's create a GiftList and save it to the database:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> gift_list = GiftList(name='Example Gift List', guests=4)
>>> gift_list.save()
>>>
```

ORM

The .save() step can be ommited if we use the .create() shortcut:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> gift_list = GiftList(name='Example Gift List', guests=4)
>>> gift_list.save()
>>> gift_list2 = GiftList.objects.create(name='Another gift list', guests=7)
>>>
```

ORM

Let's break down what has been done in the interactive shell so far. The following piece of code:

```
gift_list = GiftList(name='Example Gift List', guests=4)
gift_list.save()
```

Does two things: initializes an object, and saves it to the database. After the save() method has been called, the object is permanently stored to the database, and can be used by other models in the system.

Each model has a primary key, which can be accessed by the pk attribute. An instance will not have a primary key until it's saved to the database.

ORM

In the second case, we've done pretty much the same:

```
gift_list2 = GiftList.objects.create(name='Another gift list', guests=7)
```

We've accessed the objects attribute of the GiftList class. It's a special property, called a Manager - a special interface, through which database query operations are provided to the models.

Each model class has a manager, and you can create your own managers if you need to.

ORM

Let's create a few gifts, and attach them to the gift lists created:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> gift_list = GiftList(name='Example Gift List', guests=4)
>>> gift_list.save()
>>> gift_list2 = GiftList.objects.create(name='Another gift list', guests=7)
>>> gift1 = Gift.objects.create(name='RC Car', gift_list=gift_list)
>>> gift2 = Gift.objects.create(name='Bag of candies', gift_list=gift_list)
>>> gift3 = Gift.objects.create(name='Smartphone', gift_list=gift_list2)
>>> gift4 = Gift.objects.create(name='Game console', gift_list=gift_list2)
>>> gift5 = Gift.objects.create(name='A puppy', gift_list=gift_list2)
```

ORM

Objects can be retrieved by any field that's on the model

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> GiftList.objects.get(name='Example Gift List')
<GiftList: Example Gift List for 4 guests>

>>> GiftList.objects.get(name='Nonexistent Gift List')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
(...)
example.models.DoesNotExist: GiftList matching query does not exist.

>>>
```

ORM

We can filter to get a list of matching objects

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> GiftList.objects.filter(name__contains='Gift')
<QuerySet [<GiftList: Example Gift List for 4 guests>, <GiftList: Another gift list for 7 guests>]>

>>>
```

ORM

QuerySet is a special data type. Filters can be chained, and will be executed on the database level:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> GiftList.objects.filter(name__contains='Gift')
<QuerySet [<GiftList: Example Gift List for 4 guests>, <GiftList: Another gift list for 7 guests>]>

>>> GiftList.objects.filter(name__contains='Gift') \
...     .filter(name__contains='Example')
<QuerySet [<GiftList: Example Gift List for 4 guests>]>
```

ORM

QuerySets can be used to check for existence and count:

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)
>>> from example.models import GiftList, Gift
>>> qs = GiftList.objects.filter(name__contains='Gift')
>>> qs.exists()
True

>>> qs.count()
2
```

ORM

Iterating over a QuerySet will yield instances of the model that was retrieved

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)

>>> from example.models import GiftList, Gift
>>> qs = GiftList.objects.filter(name__contains='Gift')
>>> for gl in qs:
...     print(f'{gl.created_on} - {gl.name}')
2017-05-09 07:37:50.611498+00:00 - Example Gift List
2017-05-09 07:37:58.227882+00:00 - Another gift list

>>> gl = GiftList.objects.filter(name__contains='Gift')[0]
>>> print(gl.name)
Example Gift List
```

ORM

QuerySets provide various other methods. Some of them are:

- filter() - as the name suggests, filters a queryset using provied criteria
- exclude() - used to filter-out from a queryset
- order_by() - used to change the ordering field of a queryset
- delete() - removes every object in a queryset from the database

Full api reference is available at

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

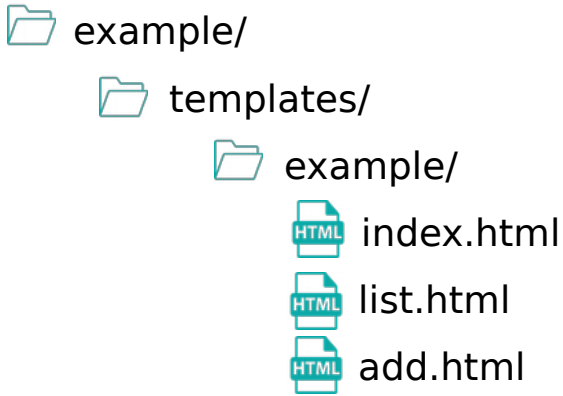
Templates

Django's template engine is a powerful mini-language (DTL) for defining user-facing layer of your application.

Let's render a simple template in example/views.py

```
from django.shortcuts import render

def hello_world(request):
    return render(request, 'example/index.html')
```

Template - folder structure



example/templates/example/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello world</title>
</head>
<body>
    <div>Hello world from template</div>
</body>
</html>
```

Templates - simple list view

example/templates/example/list.html

```
<h1>Python Has Power App</h1>
<h2>List of Gifts list</h2>
<ul>
   {% for gfl in gfl_entries %}
      <li>{{ gfl.name }}</li>
   {% endfor %}
</ul>
```

example/views.py

```
def simple_list_view(request):
   gfl_entries = GiftList.objects.all()
   return render(
      request,
      'example/list.html',
      {'gfl_entries': gfl_entries}
   )
```

php_advanced/urls.py

```
path('gift_list_by_func_view', example_views.simple_list_view),
```

Templates - tags and filters

example/templates/example/list.html

```
<h1>List of Gifts list {{ gfl_entries|length }}</h1>
<ul>
   {% for gfl in gfl_entries %}
      <li>{{ gfl.name|upper }} - {{ gfl.modified|date:'Y-m-d'}}</li>
   {% endfor %}
</ul>
```

https://docs.djangoproject.com/en/2.2/ref/templates/builtins/

Templates - class based generic views

Let’s build the same page in simpler way example/views.py

```
from django.views.generic import ListView

class GiftListView(ListView):
    model = GiftList
    template_name = 'example/list.html'
    context_object_name = 'gfl_entries'
```

php_advanced/urls.py

```
path('gift_list_by_class_view', example_views.GiftListView.as_view(), name=’list_gfl’),
```

Let's add some beauty to our application

In terminal type

pip install django-bootstrap3

in php_advanced/settings.py add

```
INSTALLED_APPS = [
    # ...
    # do not remove previous apps,
    # append this line below

    'bootstrap3',
]
```

in example/templates/example/list.html add

```
{% load bootstrap3 %}
{% bootstrap_css %}

<h1>Python Has Power App</h1>
<div class="panel panel-default">
    <div class="panel-heading">List of Gift List</div>
    <div class="panel-body">
        <ul>
            {% for gfl in gfl_entries %}
                <li>{{ gfl.name }}</li>
            {% endfor %}
        </ul>
    </div>
</div>
```

Extending templates

example/templates/example/index.html

```
{% load bootstrap3 %}
{% bootstrap_css %}

<body>
   <h1>Python Has Power App</h1>

   <div class="wrapper">
      {% block content %}{% endblock %}
   </div>
</body>
```

example/templates/example/list.html

```
{% extends 'example/index.html' %}

{% block content %}
   <div class="panel panel-default">
      <div class="panel-heading">List of Gift List</div>
      <div class="panel-body">
         <ul>
            {% for gfl in gfl_entries %}
               <li>{{ gfl.name }}</li>
            {% endfor %}
         </ul>
   </div>
{% endblock %}
```

Forms

We are able to build our own forms based on templates and views but it is not the best solution!

Django provides “auto” form system (similar to **Models**) which give us:

- automatic validation
- consistency of data
- automatic parsing of data types
- save time
- highly customizable
- own widgets
- less chance of confusion

Forms

Let’s create our first form in example/forms.py

```
from django import forms
from example.models import GiftList


class GiftListForm(forms.ModelForm):
    class Meta:
        model = GiftList
        exclude = ('modified', )
        # fields = (...)
        # fields = '__all__'
```

Forms

We can check our form in shell

```
(php-advanced) php_advanced $ python manage.py shell
(InteractiveConsole)

>>> from example.forms import GiftListForm
>>> gift_list_form = GiftListForm()
>>> print(gift_list_form)
<tr><th><label for="id_name">Name:</label></th><td><input type="text" name="name" maxlength="64"
required id="id_name"></td></tr>
<tr><th><label for="id_guests">Guests:</label></th><td><input type="number" name="guests" required
id="id_guests"></td></tr>

>>> print(gif_list_form.as_p())
<p><label for="id_name">Name:</label> <input type="text" name="name" maxlength="64" required
id="id_name"></p>
<p><label for="id_guests">Guests:</label> <input type="number" name="guests" required id="id_guests"></p>
```

Forms

Let’s try this in practice!

To add a form we need the knowledge we acquired today:

- form (extended ModelForm - we already have one)
- view
- template
- url

Create Form

Let’s create view for form in example/views.py

```
from django.views.generic import CreateView, ListView
from django.urls import reverse_lazy

from example.forms import GiftListForm
from example.models import GiftList


class GiftListCreateView(CreateView):
    model = GiftList
    form_class = GiftListForm
    success_url = reverse_lazy('list_gfl')
    template_name = 'example/add.html'
```

Create Form

Let’s create template in example/templates/example/add.html

```
{% extends 'example/index.html' %}

{% block content %}
<form method="post">
   {% csrf_token %}
   {{ form }}
   <input type="submit" value=”Save”>
</form>
{% endblock %}
```

Let’s create url in php_advanced/urls.py

```
urlpatterns = [
   # add url in urls.py
   path('gift_list/add/', example_views.GiftListCreateView.as_view(), name='add_gfl'),
]
```

Edit Form

Let’s create view for edit, in example/views_forms.py

```
from django.views.generic import CreateView, ListView, UpdateView
from django.urls import reverse_lazy

class GiftListUpdateView(UpdateView):
    model = GiftList
    form_class = GiftListForm
    template_name = 'example/add.html'
    success_url = reverse_lazy('list_gfl')
```

Let’s create url for edit, in php_advanced/urls.py

```
# add url in urls.py
path('gift_list/edit/<int:pk>/', example_views.GiftListUpdateView.as_view(), name='edit_gfl'),
```

Validators

```
class SlugField(CharField):
    # using defaults validators
    default_validators = [validators.validate_slug]


slug = forms.SlugField()
# is equivalent to:
slug = forms.CharField(validators=[validators.validate_slug])
```

Custom field validation

```
# our own field class

class MultiEmailField(forms.Field):
    def to_python(self, value):
        """Normalize data to a list of strings."""
        if not value:
            return []
        return value.split(',')

    def validate(self, value):
        """Check if value consists only of valid emails."""
        # Use the parent's handling of required fields, etc.
        super().validate(value)
        for email in value:
            validate_email(email)
```

Field validator

Let’s create our first validator in example/forms.py:

```
def clean_name(self):
    name = self.cleaned_data['name']
    if 'curse' in name.lower():
        raise forms.ValidationError('You cannot use forbidden words')

    # Always return the cleaned data, whether you have changed it or not.
    return name
```

Validating multiple fields

```
def clean(self):  # main validation function
    super(ContactForm, self).clean()
    newsletter = self.cleaned_data.get('newsletter')
    email = self.cleaned_data.get('email')
    if newsletter and not email:
        raise forms.ValidationError('You cannot sign in for our newsletter without an email')
```

Exercise time!

Using knowledge gained on the workshop, create a simple “Gift List” application:

- Ability to add/edit/delete a Gift List
  - A list of Gift Lists, an add form, an edit form, a delete view
- Ability to add/edit/remove a Gift to a Gift List
  - When viewing a Gift List: a list of Gifts, an add form, an edit form, a delete view
- Helpful links:
  - https://docs.djangoproject.com/en/2.2/topics/http/views/
  - https://docs.djangoproject.com/en/2.2/topics/db/models/
  - https://docs.djangoproject.com/en/2.2/topics/http/urls/
  - https://docs.djangoproject.com/en/2.2/topics/db/queries/

Tests

Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests – a test suite – to solve, or avoid, a number of problems:

- When you’re writing new code, you can use tests to validate your code works as expected.
- When you’re refactoring or modifying old code, you can use tests to ensure your changes haven’t affected your application’s behavior unexpectedly.

Flashback: simple unit test

Let’s remind how a simple unit test looks like in plain Python:

```
import unittest


class SampleTestCase(unittest.TestCase):
    def test_simple_function(self):
        self.assertEqual(some_function(), 123)
```

Testing in Django

In Django, tests can be written in the same way, but usually you'll see usage of

django.test.TestCase

```
from django.test import TestCase
from example.models import Gift, GiftList
from example.forms import GiftListForm

class GiftTestCase(TestCase):
    def setUp(self):
        self.gift_list = GiftList.objects.create(name='Christmas presents', guests=4)
        self.gift = Gift.objects.create(name='RC Car', gift_list=self.gift_list)

    def test_max_length(self):
        gf = GiftListForm(data={'name': 'X' * 100})
        self.assertFalse(gf.is_valid())

    def test_initial(self):
        gf = GiftListForm(instance=self.gift_list)
        self.assertEqual(gf.initial['name'], self.gift_list.name)
```

Running tests

Classes inheriting from django.test.TestCase should be placed in app/tests.py, or in separate files like app/tests/test_example.py.

Test suite can be run by invoking `python manage.py test`

Running tests

Example output from running unit tests:

```
$ python manage.py test
Creating test database for alias 'default'...
----------------------------------------------------------------------
.....................................................................................................................
.....................................................................................................................
.....................................................................................
Ran 427 tests in 17.409s

OK
Destroying test database for alias 'default'...
```

THANK YOU :-)