



# Backend Technologies



# Prerequisites

## Python in terminal / cmd

Make sure that:

- you have python and package manager PIP installed,
- directory containing python and pip binary is added to PATH environment variable,
- you can execute “***python3 --version***” and “***pip3 --version***” commands from terminal / command line.



## Virtualenv - what is it?

**Virtualenv** is a tool to create isolated Python environments. It creates an environment that has its own installation directories and does not share libraries with other virtualenv environments.

For example, it comes in handy when one of applications uses different library version than another. Without virtualenv one would have to force upgrade of the library version in the older application because they share libraries folder. With virtualenv that is not a problem!



## Preparing environment

1. Install virtualenv package via package manager in terminal / command line by executing command:

***“pip3 install virtualenv”.***



## Preparing environment

2. Create new virtual environment in a directory of your choice in terminal / command line by executing command:

***“python3 -m venv env-name”,***

where ***-m*** parameter points at desired python package to be executed (venv is a shortcut for virtualenv) and ***env-name*** parameter represents desired environment name.



## Preparing environment

3. Navigate to bin directory of virtual environment created in previous step:

in MacOS / Linux: **“env-name/bin”**,  
in Windows **“env-name/Scripts”**

and activate the environment by executing the activate script:

in MacOS / Linux: **“source activate”**,  
in Windows: **“activate.bat”**.



## Preparing environment

4. With virtual environment activated in previous step install django library in terminal / command line by executing command:

***“pip3 install django”.***

Django is now installed in virtual environment directory and ready to use.





# Exercise time!



1. Recreate previous steps in order to prepare your virtual environment for Django applications development.



# Creating first Django project

## Creating first Django project

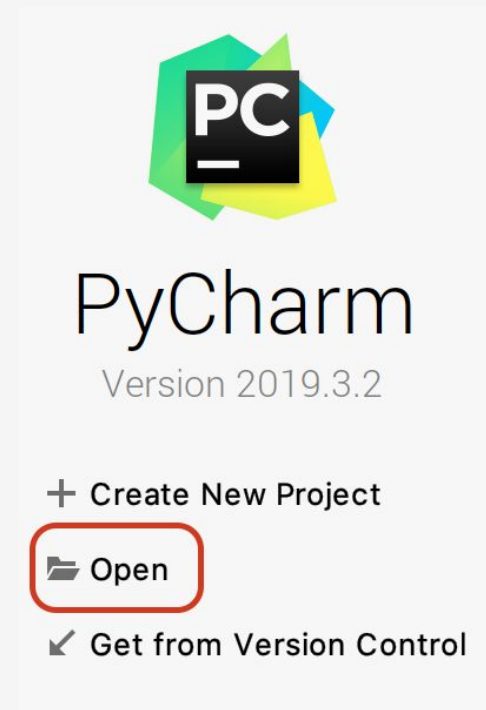
1. To create first project **make sure that you are working in context of desired virtual environment** (step 3 of preparing environment instruction). All future instructions assume you are working in the context of virtualenv.
2. Change directory to desired location for the project you are about to create.
3. Create new django project by executing command:

***“django-admin startproject helloworld”,***

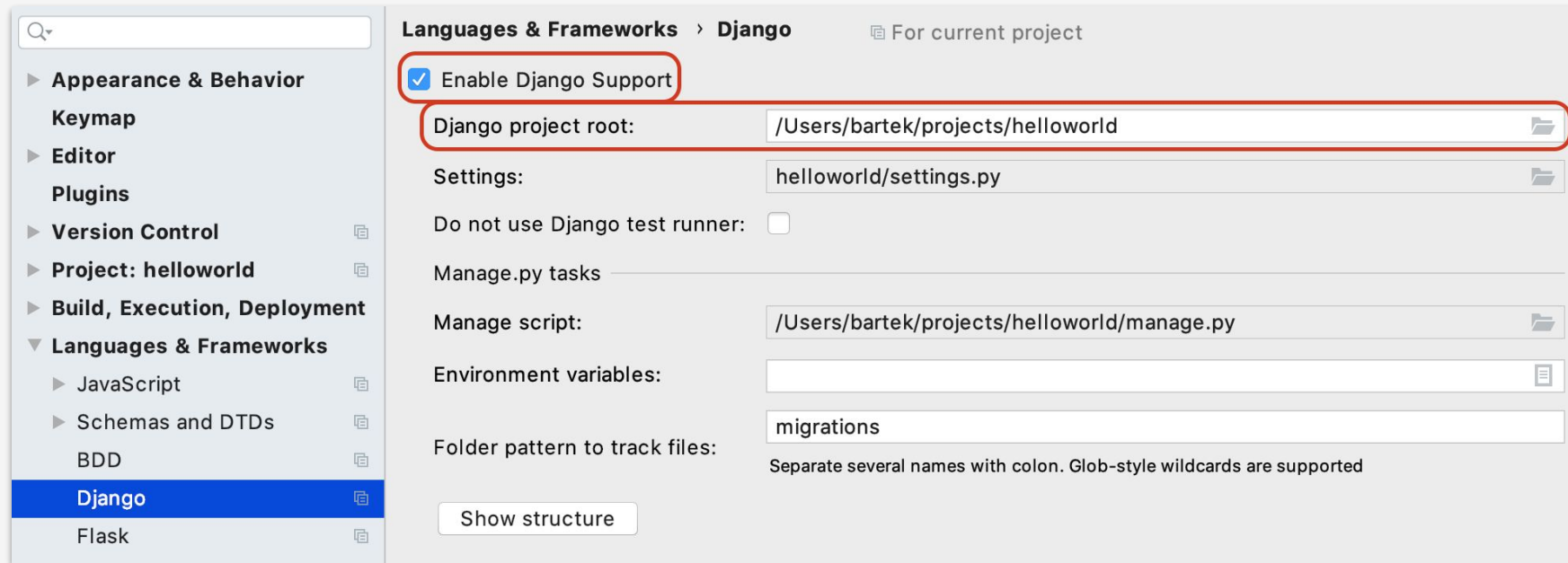
where ***helloworld*** is your desired project name.

# Opening the project in PyCharm

1. Start PyCharm and click Open.
2. Choose created project directory location.
3. In PyCharm open Settings.
4. Navigate to Languages & Frameworks -> Django.
5. Select "Enable Django Support".
6. Choose correct directory for Django project root.

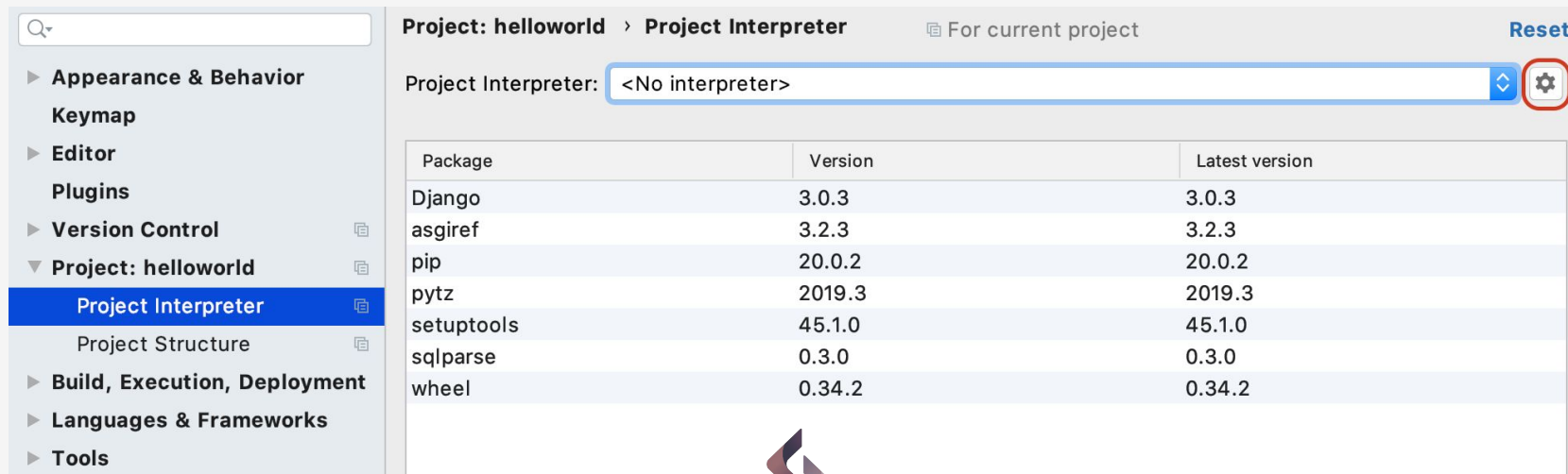


# Opening the project in PyCharm



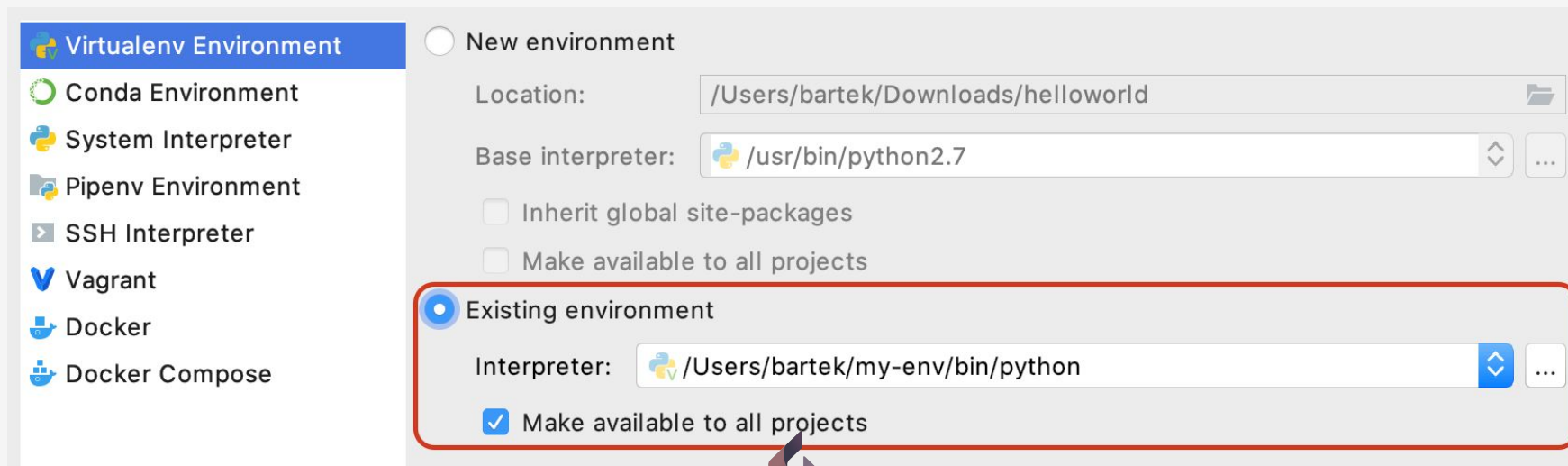
# Opening the project in PyCharm

7. In PyCharm open Settings.
8. Navigate to Project: helloworld -> Project Interpreter and click add new.



## Opening the project in PyCharm

9. Select “Existing environment” and point to python executable in virtual environment directory that you created previously.



## Django files

- **settings.py** defines settings for whole project.
- **urls.py** tells Django which views should be processed and returned on browser's request sent to a particular url.
- **wsgi.py** contains configuration for deploying Django with WSGI, the Pythonic way of deploying web applications.
- **manage.py** is used to execute various Django commands like creating new Django applications or starting local web server.





## Running Django project

To run Django project in terminal / command line execute command:

***“python3 manage.py runserver”.***

Open web browser and navigate to address *127.0.0.1:8000*. You should see your first Django application welcome page. **Congratulations!**





# Exercise time!

1. Recreate previous steps in order to create your first Django application.
2. Start the application server.
3. Verify that application is reachable under address 127.0.0.1:8000 and greets you with a default welcome page.



# Creating first Django application

## Adding new application

- Django uses concept of projects and applications to keep code well structured and organized.
- Single Django project contains one or more applications which work together in order to deliver fully working web application. For example, one application could be used for user authentication and authorization, another for payments realization and processing and so on...
- Each application focuses on a single and separated element of functionality.




## Adding new application

In order to start new Django application:

1. **make sure that you are working in context of desired virtual environment,**
2. make sure that you are in root folder of desired Django project,
3. execute command ***“python3 manage.py startapp pages”***.



## Django application files

- **admin.py** is a configuration file for built-in application called Django Admin used for administrating your application.
- **apps.py** is a configuration file for your application specifically.
- **migrations/** is used to track any changes in file **models.py** in order to keep database and **models.py** synchronized.
- **models.py** is a file where database models are defined and automatically transformed into database tables by Django.
- **views.py** is a file where logic for request and response processing is defined.
- **tests.py** contains application tests. 

## Registration of new application

Even though our new application is created, Django does not know of its existence. We need to explicitly inform Django about new application.

In order to do so, open application's **settings.py** file and find **INSTALLED\_APPS** array. Register application by adding its name to the array.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'pages'  
]
```



## Views and URLs

- In Django, **views** describe what and how to render. They represent a single web page.
- **URLs** on the other hand, describe when a particular **view** should be displayed. When a browser sends a request to a particular URL, Django checks configuration and decides which **view** to call.





## View as a function

In application's **views.py** add a new function:

```
home_page_view(request)
```

It accepts a request object and returns a response object filled with simple text greeting. This text will be rendered by web browser.

```
# pages/views.py
from django.http import HttpResponse

def home_page_view(request):
    return HttpResponse('Hello, world!')
```



## URL mapping configuration

In application directory create **urls.py** file and fill it with code instructing Django to return view we have just recently created when browser requests root URL of our application.

```
# pages/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home_page_view, name='home')
]
```



## URL application mapping

We have to include application's **urls.py** file in projects' **urls.py** file. The first parameter of path function determines value which included URLs should be prefixed with.

```
# helloworld/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls'))
]
```





# Exercise time!

1. Recreate previous steps in order to create your first Django application with hello world page.
2. This time, start the application server from PyCharm, by clicking the green triangle in the right upper corner.



3. Verify that application is reachable under address 127.0.0.1:8000 and greets you with your hello world page.



# Django templates

## What is a template?

Django needs a convenient way to generate HTML content dynamically. The most common approach relies on templates. **A template contains the static parts of the desired HTML output** as well as some special syntax describing how and where dynamic content will be inserted.

A Django project can be configured with zero to several template engines. Django ships built-in backends for its own template system called the Django Template Language (DTL).





# What is a template?

- A template is an ordinary text file of any format (HTML, XML, CSV etc.).
- A template contains **variables** that are replaced during view rendering and **tags** that control the template's logic.

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
    <a href="{{ story.get_absolute_url }}">
        {{ story.headline|upper }}
    </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

## Template location

- It is recommended to store templates in a folder called **templates**, located in the project's root folder. We need to point Django at our templates directory via project's **settings.py** configuration file.

```
# helloworld/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```





## Web page recipe

By reviewing what you have learned so far, we can define a simple recipe for creating web pages in Django:

- a template is a static HTML page with special syntax describing where and how dynamic data should be inserted,
- a view is a function that generates dynamic data to be inserted into a template,
- a URL mapping defines which view should be invoked upon receiving a request from web browser to a particular URL.





# Templates - variables

Variables are represented by variable name put inside double curly braces.

```
{{ section.title }}
```

When template engine encounters a variable, it computes the value and replaces the marker with the computation result.

Variable can contain any combination of alphanumeric characters and underscores.

A dot is used to retrieve variable attributes.

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

```
{% block title %}{{ section.title }}{% endblock %}
```

```
{% block content %}
```

```
<h1>{{ section.title }}</h1>
```

```
{% for story in story_list %}
```

```
<h2>
```

```
<a href="{{ story.get_absolute_url }}">
```

```
    {{ story.headline|upper }}
```

```
</a>
```

```
</h2>
```

```
<p>{{ story.tease|truncatewords:"100" }}</p>
```

```
{% endfor %}
```

```
{% endblock %}
```



# Templates - tags

Tags are represented by tag name put inside curly brace and a % character.

```
{% block title %}
```

Tags are more complex than variables. Some of them produce text, some of them control data flow by executing a loop or an if statement and some load external data into the template that should be later used by variables.

Django provides plenty of tags, most popular being: **for**, **if**, **block**, **extends**.

```
{% extends 'base' %}
```

```
{% block title %}{{ section.title }}{% endblock %}
```

```
{% block content %}
```

```
<h1>{{ section.title }}</h1>
```

```
{% for story in story_list %}
```

```
<h2>
```

```
<a href="{{ story.get_absolute_url }}">
```

```
    {{ story.headline|upper }}
```

```
</a>
```

```
</h2>
```

```
<p>{{ story.tease|truncatewords:"100" }}</p>
```

```
{% endfor %}
```

```
{% endblock %}
```



# Templates - tags - for

For tag iterates any collection and performs defined logic for each element in the collection.

In this example, for tag iterates variable **athlete\_list** and for each iteration prints HTML list element with athlete's name. In result, HTML page will contain list with names of athletes.

```
<ul>
    {% for athlete in athlete_list %}
        <li>{{ athlete.name }} %}</li>
    {% endfor %}
</ul>
```



# Templates - tags - url

- If we defined a name for our path in URL configuration file - **pages/urls.py** - then we can use that name with Django URL tag.
- All we have to do is to pass the mapping name and Django will generate appropriate link to our view.
- Thanks to that, we do not need to worry about changing URL mapping in many files. All we need to do is to just change the mapping in the URL configuration file and Django takes care of the rest.

```
# pages/urls.py
from django.urls import path
from pages.views import AboutView

urlpatterns = [
    path('', IndexView.as_view(), name='homepage')
]
```

```
# pages/urls.py
...
<a href="{% url 'homepage' %}">Home</a>
...
```



# Templates - tags - comments

In template files, we can distinguish 2 types of comments.

Ordinary html comments and DTL (Django Template Language) comments.

Text after DTL comments will be displayed as regular HTML text.

```
{# DTL comment #} Displayed html text.
```

```
<!-- Html comment -->
```

## Templates extending

Most powerful and most complex part of Django template engine is inheritance. **Template inheritance** allows to define global style and structure to be used in each application page.

Parent template (extended template) contains all common elements and defines blocks, which children template (extending template) can replace with its own content.





# Templates extending - block

Parent template contains **block** tags. Each block has its own name which is its unique id at the same time.

```
{% block content %}
```

Each block must end with block closing tag.

```
{% endblock %}
```

Everything between opening and closing tag is considered a default block content and may be replaced by children template if children template defines a block with the same name.

```
{# templates/base.html #}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <link rel="stylesheet" href="style.css" />  
  <title>{% block title %}My amazing site{% endblock %}</title>  
</head>  
  
<body>  
  <div id="sidebar">  
    {% block sidebar %}  
    <ul>  
      <li><a href="/">Home</a></li>  
      <li><a href="/blog">Blog</a></li>  
    </ul>  
    {% endblock %}  
  </div>  
  
  <div id="content">  
    {% block content %}{% endblock %}  
  </div>  
</body>  
</html>
```





# Templates extending - extends

Children template can define which template it wants to extend using **extends** tag.

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

The template engine first renders the parent template and then replaces block values with appropriate children template evaluated block values.

```
{# templates/index.html #}  
{% extends 'base.html' %}
```

```
{% block title %}My amazing blog{% endblock %}
```

```
{% block content %}  
  {% for entry in blog_entries %}  
    <h2>{{ entry.title }}</h2>  
    <p>{{ entry.body }}</p>  
  {% endfor %}  
{% endblock %}
```



# Rendering template in view

To instruct view function to render a template, we need to import a Django function and then invoke it with appropriate parameters:

- a request object,
- name of template file to render,
- context containing variables for template file.

```
# pages/views.py
from django.shortcuts import render

def home_page_view(request):
    context = {
        'blog_entries': [
            {
                'title': 'Hello, world!',
                'body': 'I have created my first
template in Django!'
            },
            {
                'title': 'A title',
                'body': 'And a description.'
            }
        ]
    }
    return render(request, 'index.html', context)
```

# Exercise time!



1. Recreate previous steps in order to introduce templates in your application.



# Exercise time!

1. Extend your blog posts page. Each post should have:
  - a. list of comments, each comment should have:
    - i. author name,
    - ii. content,
    - iii. creation date and time,
  - b. author name,
  - c. creation date and time,
  - d. image (you can use a URL link from google images of any image for displaying).
2. All posts from current month should be prefixed with one heading (<h1>) 'Latest posts' for all posts.
3. All posts from previous months should be prefixed with one heading (<h2>) 'Older stories' for all posts.
4. All data for rendering in templates should be statically provided from view function.

## View based on class

Previous versions of Django only provided views based on functions but programmers quickly noticed that they repeat the same patterns:

- write a view that displays all objects of a model,
- write a view that displays only one element of a model,
- and so on...

As a result, Django introduced class-based generic views that were supposed to increase quality of life for programmers and speed things up.





# TemplateView

Let's assume that we only want to display one template - **about.html**. Django has a general view for this purpose - **TemplateView**.

Our class can inherit from it and pass it template name that we want to render.

Next we need to configure this new view in URLconf. Note that **TemplateView** is a class and not a function, so instead we need to call **as\_view()** function that we inherited from **TemplateView**.

```
# pages/views.py
from django.views.generic import
TemplateView
```

```
class AboutView(TemplateView):
    template_name = "about.html"
```

```
# pages/urls.py
from django.urls import path
from pages.views import AboutView

urlpatterns = [
    path('about/', AboutView.as_view())
]
```

## More template views

Django provides even more class-based generic views. To use them effectively we first need to learn database integration as they base on SQL statements and database model classes. Most commonly used are:

- **ListView** - displays list of objects based on an SQL query,
- **DetailView** - displays details an object based on an SQL query,
- **CreateView** - creates and saves an object into the database,
- **UpdateView** - edits and saves an object into the database,
- **DeleteView** - deletes an object from the database.





# Example generic views

```
class PostCreateView(CreateView):  
    model = Post  
    template_name = 'post_new.html'  
    fields = '__all__'
```

```
class PostEditView(UpdateView):  
    model = Post  
    template_name = 'post_edit.html'  
    fields = ['title', 'text']
```

```
class PostDeleteView(DeleteView):  
    model = Post  
    template_name = 'post_delete.html'  
    success_url = reverse_lazy('post_list')
```

```
class PostListView(ListView):  
    model = Post  
    template_name = 'post_list.html'
```

```
class PostDetailView(DetailView):  
    model = Post  
    template_name = 'post_details.html'
```





# Database integration

# Object Oriented Programming

In programming there is a concept of **object oriented programming**. It simply means that instead of creating dull sequences of instructions, we are creating models and define how they interact between each other.

A model or a class in other words is a set of certain properties and activities. For example if we wanted to create a model of a cat, we would create an object named Cat which has some properties like color, age, owner, etc. The owner property can either be empty and then we have a stray cat or can be an another object of Person type.



# Django models

- Model in Django is a class of special purpose - it is saved in the database.
- An object of Model class represents one record from a database table.
- SQLite is the default built-in database engine used in Django and configured in project's settings.py file.

```
# helloworld/settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3')
    }
}
```





# Blog post model

**models.Model** means that the class is a Django database model and should be stored in the database.

We have defined model fields and specified of what type each one of them is:

- **models.CharField** - a text column with limited amount of characters,
- **models.TextField** - a text column with unlimited amount of characters,
- **models.DateTimeField** - date and time column,
- **models.ForeignKey** - a reference to another model (database relation).

```
# pages/models.py
from django.db import models
from django.utils import timezone
```

```
class Post(models.Model):
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE
    )
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(
        default=timezone.now
    )
    published_date = models.DateTimeField(
        blank=True, null=True
    )
```



# Migrations

1. To add model to the database we need to inform Django that we have changes in the **models.py**. Execute command:  
  
***"python3 manage.py makemigrations"***
2. Django has prepared a file with migrations, which we need to import into the database:  
  
***"python3 manage.py migrate pages"***,  
  
where ***pages*** is name of the application.

```
(my-env) bartek@Bartoszs-MacBook-Pro helloworld % python3 manage.py makemigrations
```

## Migrations for 'pages':

**pages/migrations/0001\_initial.py**

- Create model Post

```
(my-env) bartek@Bartoszs-MacBook-Pro helloworld % python3 manage.py migrate pages
```

## Operations to perform:

**Apply all migrations:** pages

## Running migrations:

Applying contenttypes.0001\_initial... **OK**

Applying auth.0001\_initial... **OK**

Applying pages.0001\_initial... **OK**



# Exercise time!

Let's create new project and application in order to memorize all the knowledge and double check if everything is understandable. Since we will be working on our blog application together, we must all have the same project and application names and folders structure.

1. Using the existing virtual environment create new Django project called “**shopping\_paradise**”.
2. In the project, create new application called “**products**”.
3. Open the project in PyCharm and configure the IDE.
4. Configure the project so that the “**products**” application is reachable under address **127.0.0.1:8000/products**.
5. Create model Category containing a text field “name” with maximum length of 30 characters.
6. Create model Product containing:
  - a. a text field “name” with maximum length of 50 characters,
  - b. a real number field “price”,
  - c. a “category” field (One-To-Many relation to Category, on\_deletion=DO\_NOTHING).
7. Generate Django database migrations and execute them.



Django being an ORM framework, provides easy way of retrieving database records basing on some criteria. Each model contains functions useful for retrieving the records. Examples:

```
# Saves the product.
Product(name='MacBook Pro', price=20000.00).save()
# Retrieves product matching provided criteria,
# expects only 1 result and raises Error if more records are a match.
Product.objects.get(name='test')
# Returns all records.
Product.objects.all()
# Returns objects matching criteria.
Product.objects.filter(name__contains='laptop')
# These two examples produce the same result
Product.objects.exclude(category__name__isnull=True)
Product.objects.exclude(category__name=None)
# Retrieves all products and orders them by price,
# skips first 5 and returns next 5 records.
Product.objects.all().order_by('price')[5:10]
```



# SQL queries - how to get started easily?

It might be hard for you to know all the filters and patterns, especially when you are just beginning your journey with Django. Always use IDE to do the work for you;).

```
Product.objects.filter(name)
```

- p name\_\_contains=
- p name=
- p name\_\_endswith=
- p name\_\_exact=
- p name\_\_icontains=
- p name\_\_iendswith=
- p name\_\_iexact=
- p name\_\_in=
- p name\_\_iregex=
- p name\_\_isnull=
- p name\_\_istartswith=
- p name\_\_regex=
- p name\_\_search=
- p name\_\_startswith=
- p category\_\_name=
- p category\_\_name\_\_contains=
- p category\_\_name\_\_endswith=
- p category\_\_name\_\_exact=
- p category\_\_name\_\_icontains=
- p category\_\_name\_\_iendswith=

^↓ and ^↑ will move caret down and up in the editor [Next Tip](#)



# Exercise time!



1. Experience with retrieving data from database. Write some custom methods to get the feeling of it.



# Administration panel

## Django admin panel

- Django offers an administration panel at our disposal.
- It is a great tool used to manage applications in multiple ways.
- By default it is mapped to 127.0.0.1:8000/admin.
- In order to be able to use it, we need to go through simple setup process.





# Django administration panel - setup process

To be able to log in, you need to create a superuser account. Superuser account has unlimited access to the whole website. In order to create the superuser account open terminal / command line and execute commands:

- **`python3 manage.py makemigrations`** in project root directory if it was not yet executed in project before to generate migrations for default Django system tables,
- **`python3 manage.py migrate`** in project root directory to migrate the tables,
- **`python3 manage.py createsuperuser`** in project root directory to create the superuser by providing username, e-mail address and password.

Do not worry that the password is not showing as you type it - it is still being registered, just make sure you do not make mistake.

```
(my-env) bartek@Bartoszs-MacBook-Pro shopping_paradise % python3 manage.py createsuperuser
Username (leave blank to use 'bartek'): bartek
Email address: b.kwiatek@sdacademy.dev
Password:
Password (again):
Superuser created successfully.
```



# Django administration panel - managing database

To be able to manage database records via admin panel, we need to extend admin panel project's configuration in **project's admin.py** file (shopping\_paradise/admin.py).

```
# shopping_paradise/admin.py  
from django.contrib import admin  
from products.models import Category, Product  
  
admin.site.register(Category)  
admin.site.register(Product)
```



# Exercise time!

1. Complete setup process for Django admin panel.
2. Make Category and Product models manageable from Django admin panel.
3. Add 2 categories: “Laptops” and “Desktop computers”.
4. Add 3 products for each category.



# Django templates - continuation

## Product list - ListView

To list all products in the database in a simple HTML table, we need to:

- add a new class in application's **views.py** file,
- create new HTML file in templates directory,
- register the view in application's **urls.py** file.

Note that to add product detail and add product links, the views must be already created and configured.







# Product list - ListView

```
{# templates/product_list.html #}
{% extends 'base.html' %}

{% block content %}
<table>
  <tr>
    <th>PK</th>
    <th>Name</th>
    <th>Price</th>
    <th>Category</th>
  </tr>
  {% for product in products %}
    <tr>
      <td>{{ product.pk }}</td>
      <td>
        <a href="{% url 'product_detail' product.pk %}">
          {{ product.name }}
        </a>
      </td>
      <td>{{ product.price }}</td>
      <td>{{ product.category.name }}</td>
    </tr>
  {% endfor %}
</table>
<a href="{% url 'product_add' %}">Add product</a>
{% endblock %}
```

```
# products/urls.py
from django.urls import path
from .views import ProductListView

urlpatterns = [
    path('list/', ProductListView.as_view(),
        name='product_list')
]
```

```
# products/views.py
from django.views.generic import ListView
from products.models import Product

class ProductListView(ListView):
    model = Product
    template_name = 'product_list.html'
    context_object_name = 'products'
```

## Product create - CreateView

To add new product via form, we need to:

- add a new class in application's **views.py** file,
- create new HTML file in templates directory,
- register the view in application's **urls.py** file.

Note that we also need to define what happens after successful form submit. We will define success url and set it to products list.



## Product create - CreateView

We will also add tag `{% csrf_token %}` which is provided by Django to protect our web form from Cross Site Scripting attack. It should be used for all forms. More information in discussion at Stack Overflow:

<https://stackoverflow.com/questions/5207160/what-is-a-csrf-token-what-is-its-importance-and-how-does-it-work>

To render form fields for database model we will use tag `{{ form.as_p }}`.





# Product create - CreateView

```
{# templates/product_create.html #}
{% extends 'base.html' %}

{% block content %}
    <form action="" method="post"> {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Create" />
    </form>
{% endblock %}
```

```
# products/urls.py
from django.urls import path
from .views import ProductListView

urlpatterns = [
    path('list/', ProductListView.as_view(),
name='product_list'),
    path('create/', ProductCreateView.as_view(),
name='product_create'),
]
```

```
# products/views.py
from django.views.generic import CreateView
from products.models import Product

class ProductCreateView(CreateView):
    model = Product
    template_name = 'product_create.html'
    fields = '__all__'
    success_url = reverse_lazy('product_list')
```

## Product detail - DetailView

To display a particular product details, we need to:

- add a new class in application's **views.py** file,
- create new HTML file in templates directory,
- register the view in application's **urls.py** file with a URL parameter representing primary key of the product in the database.

Note that to add product update and delete links, the views must be already created and configured.





# Product detail - DetailView

```
{# templates/product_detail.html #}
{% extends 'base.html' %}

{% block content %}
    <p>PK: {{ product.pk }}</p>
    <p>Name: {{ product.name }}</p>
    <p>Price: {{ product.price }}</p>
    <p>Category: {{ product.category.name }}</p>

    <br />

    <a href="{% url 'product_update' product.pk %}">
        Update product
    </a>
    <a href="{% url 'product_delete' product.pk %}">
        Delete product
    </a>
{% endblock %}
```

```
# products/urls.py
from django.urls import path
from .views import ProductListView, ProductCreateView,
ProductDetailView

urlpatterns = [
    path('list/', ProductListView.as_view(),
name='product_list'),
    path('create/', ProductCreateView.as_view(),
name='product_create'),
    path('detail/<int:pk>', ProductDetailView.as_view(),
name='product_detail'),
]
```

```
# products/views.py
from django.views.generic.detail import DetailView
from products.models import Product

class ProductDetailView(DetailView):
    model = Product
    template_name = 'product_detail.html'
    context_object_name = 'product'
```

## Product update - UpdateView

To update a particular product, we need to:

- add a new class in application's **views.py** file,
- create new HTML file in templates directory,
- register the view in application's **urls.py** file with a URL parameter representing primary key of the product in the database.

Note that we also need to define what happens after successful form submit. We will define success url and set it to products list.





# Product update - UpdateView

```
{# templates/product_update.html #}
{% extends 'base.html' %}

{% block content %}
    <form action="" method="post">
        {% csrf_token %}
        {# Next line automatically generates form based on
        model and puts them in paragraphs. #}
        {{ form.as_p }}
        <input type="submit" value="Update" />
    </form>
{% endblock %}
```

```
# products/urls.py
from django.urls import path
from .views import ProductListView, ProductCreateView,
ProductDetailView, ProductUpdateView

urlpatterns = [
    ...
    path('create/', ProductCreateView.as_view(),
name='product_create'),
    path('detail/<int:pk>', ProductDetailView.as_view(),
name='product_detail'),
    path('update/<int:pk>', ProductUpdateView.as_view(),
name='product_update'),
]
```

```
# products/views.py
from django.views.generic import UpdateView
from products.models import Product

class ProductUpdateView(UpdateView):
    model = Product
    template_name = 'product_update.html'
    context_object_name = 'product'
    fields = '__all__'
    success_url = reverse_lazy('product_list')
```



## Product delete - DeleteView

To delete a particular product, we need to:

- add a new class in application's **views.py** file,
- create new HTML file in templates directory,
- register the view in application's **urls.py** file with a URL parameter representing primary key of the product in the database.

Note that we also need to define what happens after successful form submit. We will define success url and set it to products list.





# Product delete - DeleteView

```
{# templates/product_delete.html #}
{% extends 'base.html' %}

{% block content %}
    <form action="" method="post"> {% csrf_token %}
        <p>
            Are you sure to delete post {{ post.title }}?
        </p>
        <input type="submit" value="Delete" />
    </form>
{% endblock %}
```

```
# products/urls.py
from django.urls import path
from .views import ProductListView, ProductCreateView,
ProductDetailView, ProductUpdateView, ProductDeleteView

urlpatterns = [
    ...
    path('detail/<int:pk>', ProductDetailView.as_view(),
name='product_detail'),
    path('update/<int:pk>', ProductUpdateView.as_view(),
name='product_update'),
    path('delete/<int:pk>', ProductDeleteView.as_view(),
name='product_delete'),
]
```

```
# products/views.py
from django.views.generic import DeleteView
from products.models import Product

class ProductDeleteView(DeleteView):
    model = Product
    template_name = 'product_update.html'
    context_object_name = 'product'
    success_url = reverse_lazy('product_list')
```



# Exercise time!

1. Create products list page.
2. Create product details page.
3. Create add product page.
4. Create edit product page.
5. Create delete product page.
6. Create category list page.
7. Create category details page.
8. Create add category page.
9. Create edit category page.
10. Create delete category page.
11. Find a way to display categories in product form pages with their names instead of default values "Category object (1)". Hint: `__str__(self)`:
12. Find a way to reduce boilerplate html pages for pages from tasks 1-10.



# Forms

# Forms

It is not hard to imagine a situation where an application needs to provide a form but it is not represented by any table in the database. For this reason Django provides Form class.

If we want to create custom form, we should inherit from Django Form class and introduce required fields. All forms are best stored in application's **forms.py** file.





# Forms

Fields in forms are defined almost exactly the same as in models. In application's **forms.py** file create the form definition.

**forms.CharField** will generate HTML validation for input fields.

It is however strongly advised that we also create custom validation on server side and never rely only client side validation.

```
from django import forms

class MyForm(forms.Form):
    name = forms.CharField(label='Name', max_length=30)
    city = forms.CharField(label='City', max_length=25)
    CHOICES = [('yes', 'Yes, I am happy!'),
               ('no', 'No, I am not happy.')]
    is_happy = forms.ChoiceField(choices=CHOICES, widget=forms.RadioSelect)

    def clean(self):
        # Data from the form is fetched using super function.
        super(MyForm, self).clean()

        # Extract the username and text field from the data.
        name = self.cleaned_data.get('name')
        city = self.cleaned_data.get('city')

        # Conditions to be met for the name and city length.
        if len(name) > 30:
            self._errors['name'] = self.error_class([
                'Maximum 30 characters.'])
        if len(city) > 25:
            self._errors['city'] = self.error_class([
                'Maximum 25 characters.'])

        # Return any errors if found.
        return self.cleaned_data
```



# Forms

We also have to define the view function that will handle both form rendering as well as form processing.

We do it in application's **views.py** file.

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import MyForm

def my_form(request):
    # If this is a POST request we need to process the form data.
    if request.method == 'POST':
        # Create a form instance and populate it
        # with data from the request.
        form = MyForm(request.POST)
        # Check whether it is valid.
        if form.is_valid():
            # Does any data require extra processing?
            # If so, do it in form.cleaned_data as required.
            return HttpResponseRedirect('/thank-you')
        else:
            # Redirect back to the same page if the data was
            invalid.
            return render(request, 'my_form.html', {'form': form})
    # If a GET (or any other method) we will create a blank form.
    else:
        form = MyForm()

    return render(request, 'my_form.html', {'form': form})
```

# Forms

The last step would be to create template to be rendered for the form and map view in URL conf.

```
<form action="{% url 'my_form' %}" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```







# Exercise time!

1. Create your form that consists of at least 4 different types of input field.
2. Validate the form on server side.



# Sessions

# Sessions

All communication between web browsers and servers happens via the HTTP protocol. The **HTTP protocol is stateless** what means there is no notion of “sequence” or behaviour based on previous messages.

Sessions are the mechanism used by Django (and most of the Internet) for keeping track of the “state” between the site and a particular browser. Sessions allow to store arbitrary data per browser and have this data available to the site whenever the browser connects.



# Sessions

Django uses a **cookie containing a special session id** to identify each browser and its associated session with the site. The actual session **data is stored in the website database** by default (this is more secure than storing the data in a cookie, where they are more vulnerable to malicious users).

We can configure Django to store the session data in other places (cache, files, “secure” cookies), but the default location is a good and relatively secure option.



# Sessions

In Django sessions are enabled by default and defined in project's **settings.py** file.

```
INSTALLED_APPS = [  
    ...  
    'django.contrib.sessions',  
    ...  
]  
  
MIDDLEWARE = [  
    ...  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    ...  
]
```



## How to use the session?

You can access the session attribute in the view from the request parameter (an `HttpRequest` passed in as the first argument to the view).

This session attribute represents the specific connection to the current user (or to be more precise, the connection to the current browser, as identified by the session id in the browser's cookie for this site).



## How to use the session?

The session attribute is a dictionary-like object that you can read and write as many times as you like in your view, modifying it as wished.

You can do all the normal dictionary operations, including clearing all data, testing if a key is present, looping through data, etc. Most of the time though, you'll just use the standard “dictionary” API to get and set values.





# Session manipulation in view function

```
# products/views.py
def session_manipulating_page(request):
    # Get a session value by its key (e.g. animal). A KeyError occurs if the key is not present
    animal = request.session['animal']

    # Get a session value, setting a default if it is not present ('Dog')
    animal = request.session.get('animal', 'Dog')

    # Set a session value
    request.session['animal'] = 'Dog'

    # Delete a session value
    del request.session['animal']
```





# Session manipulation in view function

By default, Django only saves to the session database and sends the session cookie to the client when the session has been modified (assigned) or deleted. If you're updating some data using its session key as shown in the previous section, then you don't need to worry about this! For example:

```
...  
# This is detected as an update to the session, so session data is saved.  
request.session['animal'] = 'dog'  
...
```



# Session manipulation in view function

If you're updating some information within session data, then Django will not recognise that you have made a change to the session and save the data (for example, if you were to change “breed” data inside your “animal” data, as shown below). In this case you will need to explicitly mark the session as having been modified.

...

*# Session object not directly modified, only data within the session.  
Session changes are not saved!*

```
request.session['animal']['breed'] = 'Beagle'
```

*# Set session as modified to force data updates/cookie to be saved.*

```
request.session.modified = True
```

...

# Exercise time!



1. Create a simple page counting how many times user has visited the page.



# Exercise solution

```
# products/views.py
```

```
def counter(request):
```

```
    # Number of visits to this view, stored in the session variable.
```

```
    visits_count = request.session.get('visits_count', 0)
```

```
    request.session['visits_count'] = visits_count + 1
```

```
    context = {
```

```
        'visits_count': visits_count,
```

```
    }
```

```
    # Render the HTML template passing data in the context.
```

```
    return render(request, 'counter.html', context=context)
```

```
# templates/counter.html
```

```
<p>You have visited this page {{ visits_count }}{% if visits_count = 1 %} time{%  
else %} times{% endif %}.</p>
```



# Authentication and permissions

# Authentication vs Authorization

**Authentication** is all about verifying your identity. The process checks if you really are who you say you are.

A good example is an ordinary login page, where you provide a username and a password. System then verifies if password you provided for a particular username matches the password stored in the database. If it matches, it assumes you are who you said you are.



# Authentication vs Authorization

**Authorization** is about deciding if an authenticated user (a user who passed authentication process, so a logged in user) has access privileges to a particular part of the application.

Imagine a simple blog web application. Regular users can comment on blog posts and nothing more. Owner of the blog on the other hand can access special part of the application designed to manage all blog posts and add new ones. To make sure that regular users cannot access this special part of the application, authorization is used.



## Django permissions

Django provides an authentication and authorization system, built on top of the session framework which you have already learned about. In the admin panel you can find built-in models for:

- Users and Groups,
- permissions that designate what a particular user has access to,
- tools for restricting content and views for logging in users.

Groups are an easy way of applying permissions to more than one user at the same time.





## Enabling Django permissions

In Django permissions are enabled by default and defined in project's **settings.py** file.

```
MIDDLEWARE = [  
    ...  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware'  
    ...  
]  
INSTALLED_APPS = [  
    ...  
    'django.contrib.auth',  
    'django.contrib.contenttypes'  
    ...  
]
```



## How to create new user?

You can create new user and groups in two ways. Either using GUI in admin panel or programmatically.

```
from django.contrib.auth.models import User

# Create user and save to the database
user = User.objects.create_user('sda', 'sda@sdacademy.dev', 'password')

# Update fields and then save again
user.first_name = 'John'
user.last_name = 'Doe'
user.save()
```





# Exercise time!

1. Open Django admin panel (127.0.0.1:8000/admin).
2. Create new group “customers”, do not add any permissions yet.
3. Create new user “johndoe”.
4. Add user “johndoe” to group “customers”.

## Setting up authentication views

Django provides almost everything you need in terms of user authentication. This **includes a URL mapper, views and forms**. However, it does not include one thing, that every application would probably like to create differently - the templates. In project's **urls.py** file include Django authentication urls:

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('products/', include('products.urls')),  
    path('accounts/', include('django.contrib.auth.urls'))  
]
```



## Adding authentication templates

The URLs and views that we just added expect to find their associated templates in a directory named **registration** in the templates directory of the project. Django expects that we provide HTML files named:

- password\_reset\_form.html
- password\_reset\_done.html
- password\_reset\_email.html
- password\_reset\_confirm.html
- password\_reset\_complete.html
- login.html
- logged\_out.html





# login.html

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

{% block content %}

    {% if form.errors %}
        <p>Your username and password didn't match. Please try
again.</p>
    {% endif %}

    {% if next %}
        {% if user.is_authenticated %}
            <p>Your account doesn't have access to this page. To
proceed,
            please login with an account that has access.</p>
        {% else %}
            <p>Please login to see this page.</p>
        {% endif %}
    {% endif %}
    ...
```



```
...
<form method="post" action="{% url 'login' %}">
    {% csrf_token %}
    <table>
        <tr>
            <td>{{ form.username.label_tag }}</td>
            <td>{{ form.username }}</td>
        </tr>
        <tr>
            <td>{{ form.password.label_tag }}</td>
            <td>{{ form.password }}</td>
        </tr>
    </table>
    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next }}" />
</form>

<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}
```

## Login template

By default, Django assumes that upon successfully logging in, user should be redirected to profile page (**accounts/profile**), which is not always true. To define redirection url, add this line in project's **settings.py** file:

```
LOGIN_REDIRECT_URL = '/products/list'
```



## Logout template

By navigating to URL **127.0.0.1:8000/accounts/logout** you will be logged out and by default redirected to admin logout success page. To change that, create **logged\_out.html** template in **templates/registration** directory:

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

{% block content %}
    <p>You have been logged out!</p>
    <a href="{% url 'login' %}">Click here to login again.</a>
{% endblock %}
```





## Password reset process

Password reset process is very simple and consists of 5 HTML pages:

- **password\_reset\_form.html** - a page collecting user's email address,
- **password\_reset\_done.html** - a page informing that an email with reset link has been sent,
- **password\_reset\_email.html** - a password reset email content as HTML page with link to password reset,
- **password\_reset\_confirm.html** - a page allowing for password change,
- **password\_reset\_complete.html** - a page informing that the password has been changed.





# password\_reset\_form.html

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

{% block content %}
    <form action="" method="post">
        {% csrf_token %}
        {% if form.email.errors %}
            {{ form.email.errors }}
        {% endif %}
        <p>{{ form.email }}</p>
        <input type="submit" class="btn btn-default btn-lg" value="Reset password">
    </form>
{% endblock %}
```



# password\_reset\_done.html

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

```
{% block content %}
```

```
    <p>An email with password reset instructions has been sent to your address.</p>
```

```
{% endblock %}
```



# password\_reset\_email.html

Someone asked for password reset for email {{ email }}. Follow the link below:  
{{ protocol }}://{{ domain }}{% url 'password\_reset\_confirm' uidb64=uid  
token=token %}



# password\_reset\_confirm.html

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

{% block content %}
{% if validlink %}
<p>Please enter new password.</p>
<form action="" method="post">
{% csrf_token %}
<table>
<tr>
<td>
{{ form.new_password1.errors }}
<label for="id_new_password1">
New password:
</label>
</td>
<td>{{ form.new_password1 }}</td>
</tr>
<tr>
...
<td>
{{ form.new_password2.errors }}
<label for="id_new_password2">
Confirm password:
</label>
</td>
<td>{{ form.new_password2 }}</td>
</tr>
<tr>
<td></td>
<td><input type="submit" value="Change password" /></td>
</tr>
</table>
</form>
{% else %}
<h1>Password reset failed</h1>
<p>The password reset link was invalid or already used. Please
request a new password reset.</p>
{% endif %}
{% endblock %}
```

```
graph TD
    A["{% if validlink %}"] --> B["..."]
    B --> C["<table>"]
    C --> D["..."]
    D --> E["{% else %}"]
```



# password\_reset\_complete.html

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

{% block content %}
    <h1>The password has been changed!</h1>
    <p><a href="{% url 'login' %}">Log in.</a></p>
{% endblock %}
```

## Sending the reset password email

To send emails from the application we would need to enable email support in Django and connect to any mail provider (e.g. GMail). There are some really nice articles and documentation files on the Internet about how to achieve this.

For now, simple email content logging in console will work for us. In project's **settings.py** file add this line:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```





# Exercise time!

1. Implement login, logout and password reset templates.
2. Go through the process of resetting the password (remember that emails are printed in console instead of being actually sent).



## Verifying if user is authenticated

Having authentication system created and configured, it is time to start verifying if user is logged in and has access to certain parts of the application. In **base.html** add the following code and see what changed:

```
<ul class="sidebar-nav">
  {% if user.is_authenticated %}
    <li>User: {{ user.get_username }}</li>
    <li><a href="{% url 'logout' %}?next={{ request.path }}">Logout</a></li>
  {% else %}
    <li><a href="{% url 'login' %}?next={{ request.path }}">Login</a></li>
  {% endif %}
</ul>
```



## Verifying if user is authenticated

You can also test against authenticated user in function views:

```
from django.contrib.auth.decorators import login_required
```

```
@login_required
```

```
def some_view(request):
```

```
...
```

or in class based views:

```
from django.contrib.auth.mixins import LoginRequiredMixin
```

```
class SomeView(LoginRequiredMixin, View):
```

```
...
```



# Permissions

**Permissions** are associated with models and define the operations that can be performed on a model instance by a user who has the permission. By default, Django automatically gives add, change, and delete permissions to all models.



## Permissions - models

Defining permissions is done on the model “class Meta” section, using the permissions field. You can specify as many permissions as you need in a tuple, each permission itself being defined in a nested tuple containing the permission name and permission display value. Adding new permissions to models require to run database migrations again.

For example, we might define a permission to allow a user to mark that a product is on sale.



## Permissions - models

```
class Product(models.Model):  
    name = models.CharField(max_length=50)  
    price = models.DecimalField(max_digits=10, decimal_places=2)  
    category = models.ForeignKey(Category,  
on_delete=models.DO_NOTHING)  
  
    class Meta:  
        permissions = (("can_put_on_sale", "Put product on sale"),)
```



## Permissions - templates

The current user's permissions are stored in a template variable called `{{ perms }}`. You can check whether the current user has a particular permission using the specific variable name within the associated Django application (e.g. `{{ perms.products.can_put_on_sale }}`).

```
{% if perms.products.can_put_on_sale %}  
...  
{% endif %}
```



## Permissions - views

Permissions can be tested in function view using the **permission\_required** decorator or in a class-based view using the **PermissionRequiredMixin**.



## Permissions - function views

```
from django.contrib.auth.decorators import permission_required  
  
@permission_required('products.can_put_on_sale')  
@permission_required('products.can_remove_from_sale')  
def my_view(request):  
    ...
```





## Permissions - class-based views

```
from django.contrib.auth.mixins import PermissionRequiredMixin  
  
class SomeView(PermissionRequiredMixin, View):  
    permission_required = 'products.can_put_on_sale'  
    permission_required = ('products.can_put_on_sale',  
        'products.can_remove_from_sale')
```





# Django Rest Framework

## Django Rest Framework

If we want to introduce Angular or any other modern JS framework to our project, we can still use Django as our backend application operating as REST API. We just need to install **django rest framework** via PIP in virtual environment where we installed Django and use for development:

***“pip install djangorestframework”.***



# Django Rest Framework

The steps to create basic REST API are quite straight forward:

1. set up Django,
2. create a model in the database,
3. set up the Django REST Framework,
4. serialize the model from step 2,
5. create the URI endpoints to view the serialized data.



# Django Rest Framework

In project's `settings.py` file add `rest_framework` to `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'products',  
    'rest_framework'  
]
```





# Django Rest Framework

Create new file **serializers.py** in application's directory. The file should contain classes, where each class is treated as one serializer that serializes one model with defined fields:

```
from rest_framework import serializers
from products.models import Product, Category

class ProductSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Product
        fields = ['name', 'price', 'category']

class CategorySerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Category
        fields = ['name']
```



# Django Rest Framework

Having the serializers created, let's create views in application's **views.py** that will represent API endpoints.

```
from rest_framework import viewsets
from products.models import Product, Category
from products.serializers import ProductSerializer, CategorySerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all().order_by('name')
    serializer_class = ProductSerializer

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer
```



# Django Rest Framework

Because we are using viewsets instead of views, we can automatically generate the URL conf for our API, by simply registering the viewsets with a router class. If we need more control over the API URLs we can simply drop down to using regular class-based views, and writing the URL conf explicitly.

```
from django.urls import path, include
from rest_framework import routers
from .views import ProductViewSet, CategoryViewSet

router = routers.DefaultRouter()
router.register(r'products', ProductViewSet)
router.register(r'categories', CategoryViewSet)

urlpatterns = [
    ...
    path('', include(router.urls)),
    path('rest-api/', include('rest_framework.urls', namespace='rest_framework'))
]
```





# Django Rest Framework

We can now navigate to `127.0.0.1:8000/products/categories` and we will be routed to appropriate **ViewSet**. Once we open the address with web browser, a well formed HTML page is returned. If we have called the endpoint with a REST call (with appropriate headers) the response would be a json.

Django REST framework

bartek ▾

Api Root / Category List

Category List

OPTIONS GET ▾

GET /products/categories/

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

[  
 {  
 "name": "TestCategory"  
 }  
]

Raw data HTML form

Name

POST



# Exercise time!

1. Introduce Django Rest Framework to your application.
2. Provide endpoints for Product and Category models.
3. Check out the possibilities of the well formed HTML page that Django Rest Framework renders.



Thank you for your attention!