

Getting Started with Django

Summary: in this tutorial, you'll learn how to create a new Django project, understand the project structure, and launch the Django web app from a web browser.

Django overview

Django is a [Python web framework](#) that includes a set of components for solving common web development problems.

Django allows you to **rapidly develop web applications with less code** by taking advantage of its framework.

Django follows the **DRY (don't repeat yourself)** principle, which allows you to maximize the code reusability.

Django uses the **MVT (Model-View-Template) pattern**, which is slightly similar to the **MVC (Model-View-Controller) pattern**.

The MVT pattern consists of three main components:

- **Model** – defines the data or contains the logic that interacts with the data in the database.
- **View** – communicates with the database via model and transfers data to the template for representing the data.
- **Template** – defines the template for displaying the data in the web browser.

The Django framework itself acts as a controller. The Django framework uses URL patterns that send the request to an appropriate view.

If you are familiar with **MVC**, the following are equivalent:

- **Template (T) is equivalent to View (V) in MVC**
- **View (V) is equivalent to Controller (C) in MVC**
- **Model (M) is equivalent to Model (M) in MVC**

In practice, you'll often work with models, views, templates, and URLs in the Django application.

Django helps you write software that is:

[Complete](#)

Django follows the "**Batteries included**" **philosophy** and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, follows consistent design principles, and has extensive and [up-to-date documentation](#).

[Versatile](#)

Django can be (and has been) used to build almost any type of website — from **content management systems and wikis, through to social networks and news sites**. It can work with

any client-side framework, and can deliver content in almost any format (including HTML, RSS feeds, JSON, and XML).

Internally, while it provides choices for almost any functionality you might want (e.g. several popular databases, templating engines, etc.), it can also be extended to use other components if needed.

[Secure](#)

Django helps developers avoid many common security mistakes by providing a framework that has been engineered to **"do the right things"** to protect the website automatically. For example, Django provides a secure way to manage user accounts and passwords, **avoiding common mistakes like putting session information in cookies where it is vulnerable** (instead cookies just contain a key, and the actual data is stored in the database) or **directly storing passwords rather** than a password hash.

A password hash is a fixed-length value created by sending the password through a [cryptographic hash function](#). Django can check if an entered password is correct by running it through the hash function and comparing the output to the stored hash value. However due to the "one-way" nature of the function, even if a stored hash value is compromised it is hard for an attacker to work out the original password.

Django enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and [clickjacking](#) (see [Website security](#) for more details of such attacks).

[Scalable](#)

Django uses a **component-based "shared-nothing" architecture** (each part of the architecture is independent of the others, and can hence be replaced or changed if needed). Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers. Some of the busiest sites have successfully scaled Django to meet their demands (e.g. **Instagram** and **Disqus**, to name just two).

[Maintainable](#)

Django code is written using design principles and patterns that encourage the creation of maintainable and reusable code. In particular, it makes use of the **Don't Repeat Yourself (DRY) principle** so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable "applications" and, at a lower level, groups related code into modules (along the lines of the [Model View Controller \(MVC\)](#) pattern).

[Portable](#)

Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform, and can run your applications on many **flavors of Linux**,

Windows, and macOS. Furthermore, Django is well-supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Django sites.

History:

Django was initially developed between **2003 and 2005** by a web team who were responsible for creating and maintaining newspaper websites.

After creating a number of sites, the team began to factor out and reuse lots of common code and design patterns.

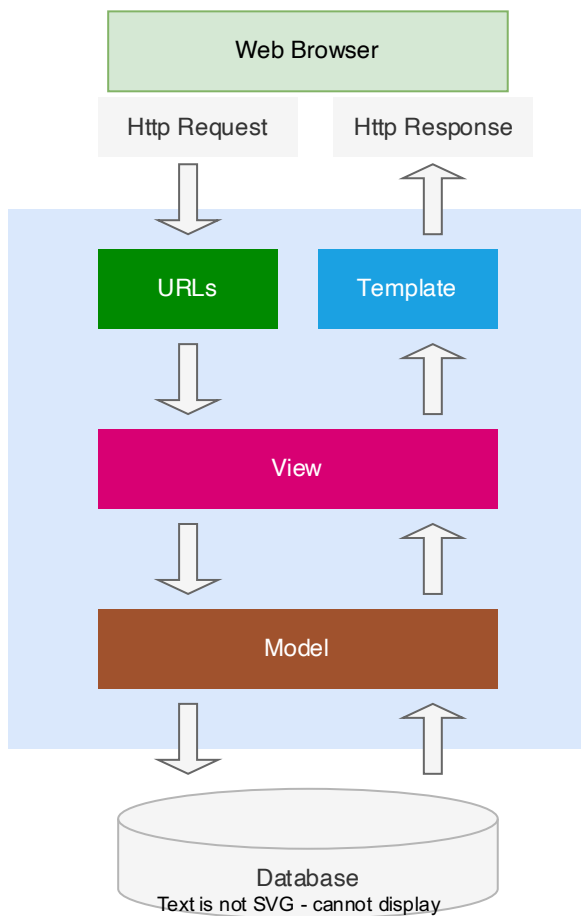
This common code evolved into a **generic web development framework**, which was open-sourced as the **"Django" project in July 2005**.

Django has continued to grow and improve, from its first milestone release (1.0) in **September 2008 through to the version 5.0 in 2023**.

Each release has added new functionality and bug fixes, ranging from support for new types of databases, template engines, and caching, through to the addition of "generic" view functions and classes (which reduce the amount of code that developers have to write for a number of programming tasks).

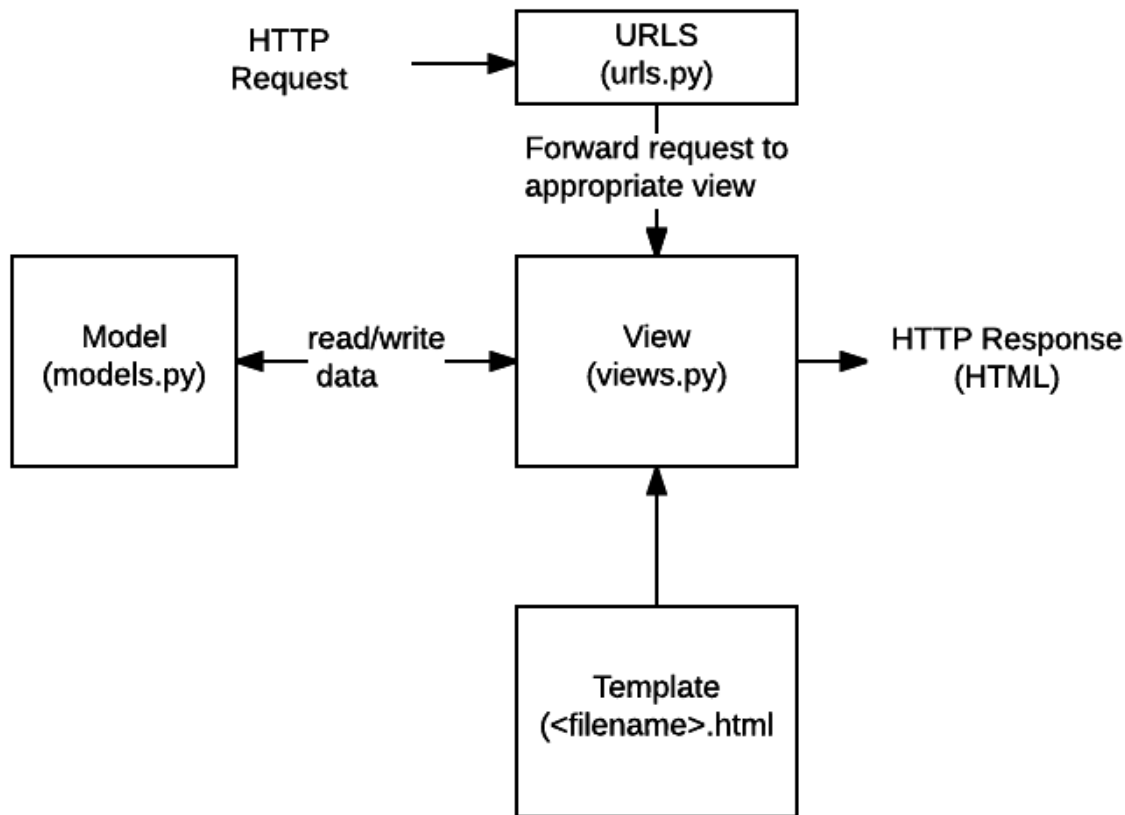
Django architecture

The following picture shows how Django manages the HTTP request/response cycle using its components:



- First, a web browser requests a page specified by a URL from a web server. The web server passes the HTTP request to Django.
- Second, Django matches the URL with URL patterns to find the first match.
- Third, Django calls the View that corresponds to the matched URL.
- Fourth, the view uses a model to retrieve data from the database.
- Fifth, the model returns data to the view.
- Finally, the view renders a template and returns it as an HTTP response.

Django web applications typically group the code that handles each of these steps into separate files:



- **URLs:** While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of **strings or digits that appear in a URL and pass these to a view function as data**.
- **View:** A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests **via *models*, and delegate the formatting of the response to *templates***.
- **Models:** Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (**add, modify, delete**) and query records in the database.
- **Templates:** A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*. A template can be used to define the structure of any type of file; it **doesn't have to be HTML!**

Install pip:

Install command:

```
python -m ensurepip
```

Check the installed version

```
pip -version
```

Upgrade:

```
python3 -m pip install --upgrade pip
```

Install Django:

```
pip install Django
```

```
python -m django --version
```

Exploring Django commands

Django comes with a command-line utility program called **django-admin** that manages administrative tasks such as creating a new project and running the Django development server. To list all available Django commands, you execute the following django-admin command like this:

django-admin

Output:

Type '**django-admin help <subcommand>**' for help on a specific subcommand.

Available subcommands:

```
[django]
  check
  compilemessages
  createcachetable
  dbshell
  diffsettings
  dumpdata
  flush
  inspectdb
  loaddata
  makemessages
  makemigrations
  migrate
  optimizemigration
  runserver
  sendtestemail
  shell
  showmigrations
  sqlflush
```

```
sqlmigrate
sqlsequencereset
squashmigrations
startapp
startproject
test
testserver
```

For now, we're interested in the **startproject** command that creates a new Django project. The following **startproject** command creates a new project called `django_project`:

```
django-admin startproject django_project
```

This command creates a `django_project` directory.

Let's explore the project structure:

```
cd django_project
```

The following shows the `django_project` structure:

```
├── django_project
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   └── __init__.py
└── manage.py
```

Here's a quick overview of each file in the Django project:

- `manage.py` is a command-line program that you use to interact with the project like starting a development server and making changes to the database.

The `django_project` is a Python package that consists of the following files:

- `__init__.py` – is an empty file indicating that the `django_project` directory is a package.
- `settings.py` – contains the project settings such as installed applications, database connections, and template directories.
- `urls.py` – stores a list of routes that map URLs to views.
- `wsgi.py` – contains the configurations that run the project as a web server gateway interface (WSGI) application with WSGI-compatible web servers.
- `asgi.py` – contains the configurations that run the project as an asynchronous web server gateway interface (ASGI) application with ASGI-compatible web servers.

Running the Django development server

Django comes with a built-in web server that allows you to quickly run your Django project for development purposes.

The Django development web server will continuously check for code changes and reload the project automatically. However, you still need to restart the web server manually in some cases such as adding new files to the project.

To run the Django development server, you use the `runserver` command:

```
python manage.py runserver
```

Output:

```
Watching for file changes with StatReloader
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
...
```

```
Django version 4.1.1, using settings 'django_project.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CTRL-BREAK.
```

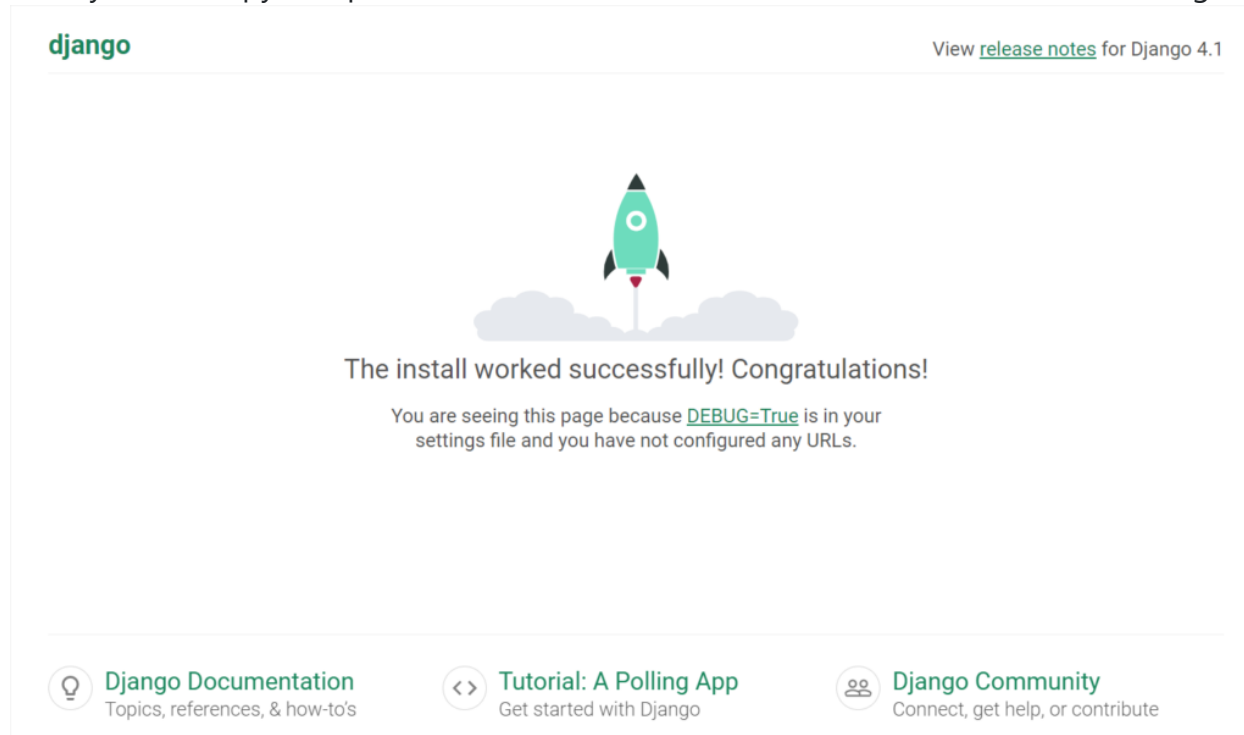
```
Code language: plaintext (plaintext)
```

Once the server is up and running, you can open the web app using the URL listed in the output.

Typically, the URL is something like this:

```
http://127.0.0.1:8000/
```

Now, you can copy and paste the URL to a web browser. It should show the following webpage:



The `urls.py` contains a default route that maps `/admin` path with the `admin.site.urls` view:

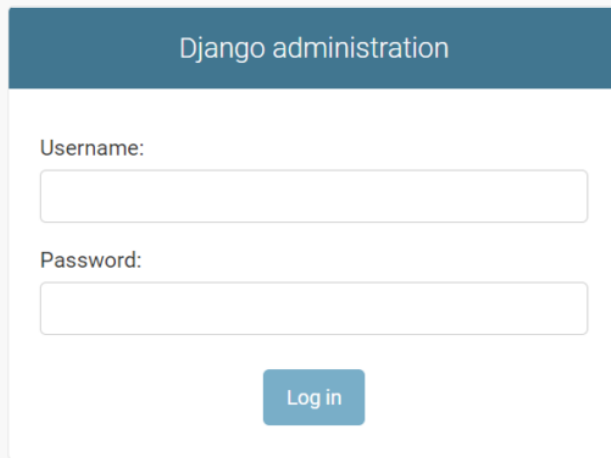
```
from django.contrib import admin
from django.urls import path
```

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

To open the [admin page](#), you use the following URL:

```
http://127.0.0.1:8000/admin
```

It'll show a login page:

A screenshot of the Django administration login interface. It features a dark blue header bar with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". A blue "Log in" button is positioned below the password field. The entire form is centered on a light gray background.

Django administration

Username:

Password:

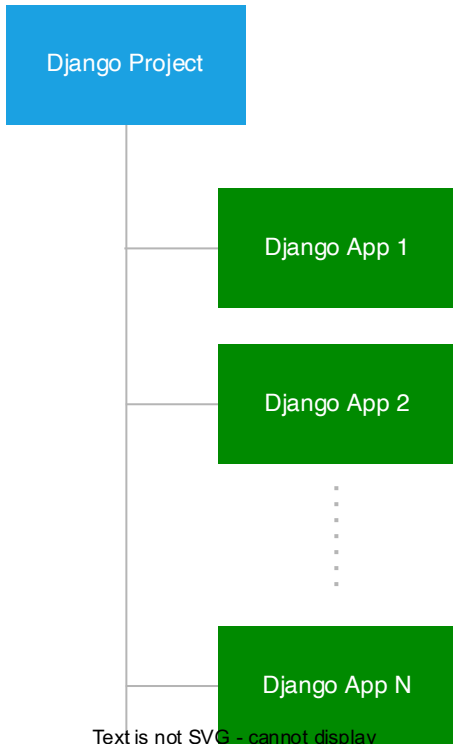
Log in

Django projects and applications

In the Django framework:

- A project is a Django installation with some settings.
- An application is a group of models, views, templates, and URLs.

A Django project may have one or more applications. For example, a project is like a website that may consist of several applications such as blogs, users, and wikis. Typically, you design a Django application that can be reusable in other Django projects. The following picture shows the structure of a Django project and its applications:



Creating a blog application

To create an application, you use the `startapp` command as follows:

```
python manage.py startapp app_name
```

For example, you can create an application called `blog` using the `startapp` command like this:

```
python manage.py startapp blog
```

The command creates a `blog` directory with some files:

```
├── blog
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   ├── models.py
│   ├── tests.py
│   ├── views.py
│   └── __init__.py
├── db.sqlite3
├── django_project
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   ├── __init__.py
│   └── __pycache__
```

└─ manage.py

Registering an application

After creating an application, you need to register it to the project especially when the application uses [templates](#) and interacts with a database.

The `blog` app has the `apps.py` module which contains the `BlogConfig` class like this:

```
from django.apps import AppConfig
```

```
class BlogConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'blog'
```

To register the `blog` app, you add the `blog.apps.BlogConfig` class to the `INSTALLED_APPS` list in the `settings.py` of the project:

```
INSTALLED_APPS = [
    # ...
    'blog.apps.BlogConfig',
]
```

Alternatively, you can use the app name like `blog` in the `INSTALLED_APPS` list like this:

```
INSTALLED_APPS = [
    # ...
    'blog',
]
```

Creating a view

The `views.py` file in the `blog` directory comes with the following default code:

```
from django.shortcuts import render
```

The `views.py` will contain all the views of the application. A view is a function that takes an `HttpRequest` object and returns an `HttpResponse` object. It's equivalent to the controller in the MVC architecture.

To create a new view, you import the `HttpResponse` from the `django.http` into the `views.py` file and [define a new function](#) that accepts an instance of the `HttpRequest` class:

```
from django.shortcuts import render
from django.http import HttpResponse
```

```
def home(request):
    return HttpResponse('<h1>Blog Home</h1>')
```

In this example, the `home()` function returns a new `HttpResponse` object that contains a piece of HTML code. The HTML code includes an `h1` tag.

The `home()` function accepts an instance of an `HttpRequest` object and returns an `HttpResponse` object. It is called a function-based view. Later, you'll learn how to create class-based views.

To map a URL with the `home()` function, you create a new file `urls.py` inside the `blog` directory and add the following code to the `urls.py` file:

```
from django.urls import path
from . import views
```

```
urlpatterns = [
```

```
path('', views.home, name='posts'),
]
```

How it works.

First, import the `path` from `django.urls` module:

```
from django.urls import path
```

Second, import the `views.py` module from the current directory.

```
from . import views
```

Note that this is a relative import that imports the `views` module from the current directory.

Third, define a route that maps the blog URL with the `home()` function using the `path()` function.

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

The `name` keyword argument defines the name of the route. Later, you can reference the URL using the route name instead of the hard-code URL like `blog/`.

By using the name for the path, you can change the URL of the path to something else like `my-blog/` in the `urls.py` instead of changing the hard-coded URL everywhere.

Note that the final argument of the `path` must be a keyword argument like `name='posts'`. If you use a positional argument like this:

```
from django.urls import path
from . import views
```

```
urlpatterns = [
    path('', views.home, 'posts'), # Error
]
```

you'll get the following error:

```
TypeError: kwargs argument must be a dict, but got str.
```

To make the blog's routes work, you need to include the `urls.py` of the `blog` application in the `urls.py` file of the Django project:

```
from django.contrib import admin
from django.urls import path, include # new
```

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

In the `urls.py` of the project, we import the `include` function from the `django.urls` and map the path of the blog to the `blog.urls`.

By doing this, when you navigate to `http://127.0.0.1:8000/blog/`, Django will run the `home()` function of the `views.py` module and returns a webpage that displays a `h1` tag.

Before opening the URL, you need to start the Django development web server:

```
python manage.py runserver
```

When you navigate to `http://127.0.0.1:8000/blog/`, you'll see a webpage that displays the `Blog Home` heading.

Here's the flow:

- First, the web browser sends an HTTP request to the URL `http://127.0.0.1:8000/blog/`

- Second, Django executes the `urls.py` in the `django_project` directory. It matches the `blog/` with the URL in the `urlpatterns` list in the `urls.py`. As a result, it sends `"` to the `urls.py` of the `blog` app.
- Third, Django runs the `urls.py` file in the `blog` application. It matches the `"` URL with the `views.home` function and execute it, which returns an HTTP response that outputs a `h1` tag.
- Finally, Django returns a webpage to the web browser.

Adding more routes

First, define the `about()` function in the `views.py` of the `blog` application:

```
from django.shortcuts import render
from django.http import HttpResponse
```

```
def home(request):
    return HttpResponse('<h1>Blog Home</h1>')
```

```
def about(request):
    return HttpResponse('<h1>About</h1>')
```

Second, add a route to the `urls.py` file:

```
from django.urls import path
from . import views
```

```
urlpatterns = [
    path('', views.home, name='posts'),
    path('about/', views.about, name='about'),
]
```

Third, open the URL `http://127.0.0.1:8000/blog/about/`, and you'll see a page that displays the About page.

Now, if you open the home URL, you'll see a page that displays a page not found with a 404 HTTP status code.

The reason is that the `urls.py` in the `django_project` doesn't have any route that maps the home URL with a view.

To make the `blog` application the homepage, you can change the route from `blog/` to `"` as follows:

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

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

If you open the URL `http://127.0.0.1:8000`, you'll see the blog home page. And navigating to the URL `http://127.0.0.1:8000/about/` will take you to the About page.

Django Templates

Summary: in this tutorial, you'll learn how to create Django templates, pass data from view functions to templates, and display the data in the templates.

Introduction to the Django templates

In the [previous tutorial](#), you learned how to return a `HttpResponse` with a `h1` tag from a view. To return a full HTML page, you'll need to use a template.

Note that it is possible to return a full HTML page by mixing the HTML with Python code. However, it is not practical and doesn't scale well.

A template is a file that contains the static and dynamic parts of a webpage. To generate the dynamic parts of the webpage, Django uses its specific template language called Django template language or DTL.

The Django template engine renders templates that contain variables, constructs, tags, and filters.

Variables

A variable is surrounded by `{{` and `}}`. For example:

```
Hi {{name}}, welcome back!
```

In this template, the `name` is a variable. If the value of the `name` variable is `John`, the Django template engine will render the above template to the following text:

```
Hi John, welcome back!
```

If a variable is a [dictionary](#), you can access the items of the dictionary using the dot notation

(`dict_name.key`).

Suppose you have a `person` dictionary with two keys `name` and `email` :

```
person = {'name': 'John', 'email': 'john@pythontutorial.net'}
```

... you can access the values of the `name` and `email` keys of the `person` dictionary in the template like this:

```
{{ person.name }}  
{{ person.email }}
```

Tags

Tags are responsible for outputting contents, serving a control structure if-else, for-loop, and getting data from a database.

Tags are surrounded by `{%` and `%}` . For example:

```
{% csrf_token %}
```

In this example, the `csrf_token` tag generates a token for preventing CSRF attacks.

Some tags like `if-else` and `for-loop` require beginning and ending tags. For example:

```
{% if user.is_authenticated %}  
Hi {{user.username}}  
{% endif %}
```

Filters

Filters transform the contents of variables and tags argument. For example, to capitalize each word of a string, you use the `title` filter like this:

```
{{ name | title }}
```

If the value of the `name` variable is `john doe`, then the `title` filter will transform it to the following:

```
John Doe
```

Some filters accept an argument. For example, to format a date of the `joined_date` variable in the `Y-m-d` format, you use the following filter:

```
{{ joined_date | date: "Y-m-d" }}
```

Here are the complete [built-in template tags and filters](#).

Comments

The comments will look like this:

```
{# This is a comment in the template #}
```

The Django template engine will not render text inside the comment blocks.

Django template examples

First, create a new directory called `templates` inside the `blog` directory:

```
mkdir templates
```

Second, create a `blog` directory inside the `templates` directory:

```
cd templates  
mkdir blog
```


Note that the directory inside the `templates` directory must have the same name as the application name. In this example, the `blog` directory has the same name as the `blog` app of the Django project.

Third, inside the `templates/blog` directory create two template files `home.html` and `about.html` with the following contents. The

`home.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Home</title>
</head>
<body>
  <h1>Home</h1>
</body>
</html>
```

The `about.html` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>About</title>
</head>
<body>
  <h1>About</h1>
</body>
</html>
```

It's important to note that you should add the `blog` application to the `INSTALLED_APPS` list in the `settings.py` file to make the templates work. Typically, you do it immediately after [creating the new Django application](#).

```
INSTALLED_APPS = [  
    # ...  
    'blog.apps.BlogConfig',  
]
```

Fourth, open the `views.py` file and change the `home()` and `about()` view functions to the following:

```
from django.shortcuts import render  
  
def home(request):  
    return render(request, 'blog/home.html')  
  
def about(request):  
    return render(request, 'blog/about.html')
```

In this `views.py` file, we import the `render()` function from the `django.shortcuts`.

The `render()` function accepts an `HttpRequest` object and a path to a template. It renders the template and returns an `HttpResponse` object.

Fifth, run the Django development server:

```
python manage.py runserver
```

Finally, open the URL `http://127.0.0.1:8000/` and the URL `http://127.0.0.1:8000/about/`, you'll see full HTML pages that come from the `home.html` and `about.html` templates.

[Passing variables to a template](#)

We'll create dummy blog post data and pass it to the `home.html` template. Later, you'll learn how to get the post data from the database.

The `views.py` will look like this:

```
from django.shortcuts import render

posts = [
    {
        'title': 'Beautiful is better than ugly',
        'author': 'John Doe',
        'content': 'Beautiful is better than ugly',
        'published_at': 'October 1, 2022'
    },
    {
        'title': 'Explicit is better than implicit',
        'author': 'Jane Doe',
        'content': 'Explicit is better than implicit',
        'published_at': 'October 1, 2022'
    }
]

def home(request):
    context = {
        'posts': posts
    }
    return render(request, 'blog/home.html', context)

def about(request):
    return render(request, 'blog/about.html')
```

How it works.

- First, create a new list (`posts`) that stores the dummy post data.
- Second, define a new dictionary `context` inside the `home()` function with the key `posts` and pass it to the `render()` function as the third argument.

Inside `home.html` template, you can access the post data via the `posts` variable.

The following `home.html` template that displays the posts:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Blog</title>
</head>
<body>
    {% for post in posts %}
        <h2>{{ post.title }}</h2>
        <small>Published on {{ post.published_at }} by {{ post.author}}</small>
        <p>{{ post.content }}</p>
    {% endfor %}
</body>
</html>
```

How it works.

- First, use a `for` loop to iterate over the `posts` variable. The `for` loop ends with `endfor`. Both `for` and `endfor` are surrounded by `{%}` and `%}`.
- Second, place the value of each item in the dictionary using dot notation (`.`).

If you save the `home.html` and open the URL `http://127.0.0.1:8000/`, you'll see the postdata displayed on the page.

Besides the `for` loop, you can use another conditional statement like `if-else`. For example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>{% if title %} {{title}} {% else %} Blog {% endif %}</title>
```

```
</head>
<body>
    {% for post in posts %}
        <h2>{{ post.title }}</h2>
        <small>Published on {{ post.published_at }} by {{ post.author}}</small>
        <p>{{ post.content }}</p>
    {% endfor %}
</body>
```

This example uses an `if-else` statement to show the `title` variable if it is available or the `Blog` otherwise.

To pass the `title` variable to the `home.html` template, you add a new entry to the `context` dictionary with the key `title` in the `home()` function like this:

```
def home(request):
    context = {
        'posts': posts,
        'title': 'Zen of Python'
    }
    return render(request, 'blog/home.html', context)
```

If you refresh the home URL `http://127.0.0.1:8000/`, you'll see the new title.

Typically, a website has some common sections like a header, footer, and sidebar. To avoid repeating them in every template, you can use a base template.

Creating a base template

First, create a new `templates` directory in the project directory (not the `blog` app):

```
└─ blog
└─ db.sqlite3
└─ django_project
└─ manage.py
└─ templates
```

```
└─ users
```

Next, add the template directory to the `TEMPLATES` option in the `settings.py` file of the project:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates' ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

Note that `BASE_DIR` is a `Path` object that comes from the `pathlib` built-in module. The forward-slash `/` is an operator that concatenates the `BASE_DIR` object with the `'templates'` string. This feature is called [operator overloading](#) in Python.

Then, create `base.html` in the `templates` directory with the following code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>{% if title %} {{title}} {% else %} Blog {% endif %}</title>
  </head>
  <body>
    {% block content %}
```

```
        {% endblock %}

    </body>

</html>
```

The `base.html` is served as the base template for other templates. The name of the base template can be anything like `main.html` .

After that, change the `home.html` template inside the `templates/blog` directory as follows:

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

{% block content %}
    <h1>My Posts</h1>
    {% for post in posts%}
    <h2>{{ post.title }}</h2>
    <small>Published on {{ post.published_at }} by {{ post.author}}</small>
    <p>{{ post.content }}</p>
    {% endfor%}
{% endblock %}
```

The `home.html` extends the `base.html` template using the `extends` tag. The `home.html` template has its section for the `content` block.

Also, change the `about.html` template that extends the `base.html` template:

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

{% block content %}
    <h1>About</h1>
{% endblock content %}
```

Finally, restart the Django development server and open URL `http://127.0.0.1:8000/` , and you'll see the changes.

Configure static files

The static files are CSS, JavaScript, and image files that you use in the templates. To use the static files in the templates, you follow these steps:

First, create a `static` directory inside the project directory:

```
mkdir static
```

The project directory will look like this:

```
├── blog
├── db.sqlite3
├── manage.py
├── mysite
├── static
└── templates
```

Second, set the `STATICFILES_DIRS` in the `settings.py` after the `STATIC_URL` file so that Django can find the static files in the static directory:

```
STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / 'static']
```

Third, create three directories `js`, `css`, and `images` directory inside the `static` directory:

```
├── static
│   ├── css
│   ├── images
│   └── js
```

Fourth, create `style.css` inside the CSS directory with the following contents.

```
h1{
    color:#0052EA
}
```

```
form {  
    max-width: 400px;  
}  
  
label, input, textarea, select{  
    display:block;  
    width:100%;  
}  
  
input[type="submit"]{  
    display:inline-block;  
    width:auto;  
}  
  
.errorlist {  
    padding:0;  
    margin:0;  
}  
.errorlist li{  
    color:red;  
    list-style:none;  
}  
  
.alert{  
    padding:0.5rem;  
}  
  
.alert-success{  
    background-color: #dfd  
}  
.alert-error{  
    background-color:#ba2121;  
    color:#fff;  
}  
}
```

Note that we use only some simple CSS rules to make the tutorials easier to follow. Our primary focus is Django, not CSS or JavaScript.

Fifth, create the `app.js` inside the `js` directory with the following code:

```
setTimeout(() => {  
    alert('Welcome to my site!');  
}, 3000);
```

This code shows an alert after the page is loaded for 3 seconds.

Sixth, edit the `base.html` template to load the `style.css` and `app.js` files:

```
{%load static %}  
<!DOCTYPE html>  
<html lang="en">  
    <head>  
        <meta charset="UTF-8" />  
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
        <link rel="stylesheet" href="{% static 'css/style.css' %}" />  
        <script src="{% static 'js/app.js' %}" defer></script>  
        <title>My Site</title>  
    </head>  
    <body>  
        {%block content%}  
        {%endblock content%}  
    </body>  
</html>
```

Seventh, restart the Django development server, open the URL `http://127.0.0.1:8000/`, and you'll see that the color of the heading changes according to the CSS rule.

Also, you'll see an alert after about 3 seconds because the `JavaScript` code in the `app.js` runs:

127.0.0.1:8000 says

Welcome to my site!

OK

Since we're not focusing on the JavaScript part, you can remove the code in the `app.js` file to continue the next tutorial.

Summary

- A Django template contains both static and dynamic parts of a web page.
- Django uses Django Template Language (DTL) by default to create templates.
- Use `{{ variable_name }}` to display the value of the `variable_name` in a template.
- Use `{% control_tag %}` to include a control tag in a template.
- Use the `static` tag to load the static files including CSS, JavaScript, and images.