ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA

DJANGO FRAMEWORK

LẬP TRÌNH PYTHON

Khoa CÔNG NGHỆ THÔNG TIN
Nguyễn Thị Lệ Quyên

Nội dung

- Tổng quan Django framework
- Cấu trúc project Django
- MVT Pattern (Model-View-Template)
- Lập trình Web với Django
- Ref: The web framework for perfectionists with deadlines | Django (djangoproject.com)

Tổng quan

- Django là web framework miễn phí, mã nguồn mở để xây dựng các ứng dụng web Python hiện đại
- Django giúp nhanh chóng xây dựng các ứng dụng web bằng cách xây dựng sẵn các thành phần và tái sử dụng như kết nối với cơ sở dữ liệu, xử lý bảo mật, cho phép xác thực người dùng, tạo URL, hiện thị nội dung thông qua các templates và forms, hỗ trợ nhiều CSDL backends, và thiết lập giao diện hiển thị
- Developers chỉ cần tập trung vào xây dựng chức năng của ứng dụng web thay vì phải làm lại các chức năng chuẩn của 1 ứng dụng web
- Django được sử dụng phổ biến

Yêu cầu kỹ thuật

- Python 3.8+
- pip
- Django 4.0+
- Visual Studio (VS) Code
- Sử dụng môi trường ảo (Virtual Environments) (optional)

```
$ python3 -m venv myenv       # Tạo môi trường ảo

$ source myenv/bin/activate   # Active môi trường ảo trong Linux/MacOS
$ .\myenv\Scripts\activate    # Hoặc Active môi trường ảo trong Windows

(myenv) $ pip3 install django
```

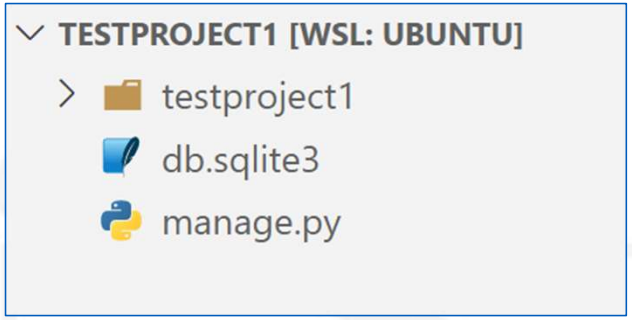
Ref: venv — Creation of virtual environments — Python 3.11.2 documentation

Cấu trúc dự án

- `manage.py`
  - Không nên sửa đổi
  - File giúp thực hiện các thao tác quản trị, vd như chạy máy chủ cục bộ

```
(myenv) $ python3 manage.py runserver
```

- `db.sqlite3`
  - Chứa CSDL
- `testProject`



Cấu trúc thư mục dự án sau khi tạo và chạy máy chủ cục bộ bằng lệnh `runserver`

Create the Django project

- In the VS Code Terminal where your virtual environment is activated, run the following command:
  - `django-admin startproject web_project .`
  - This `startproject` command assumes (by use of . at the end) that the current folder is your project folder.
- Create an empty development database by running the following command:
  - `python manage.py migrate`
  - `When you run the server the first time, it creates a default SQLite database in the file db.sqlite`

Create the Django project

- To verify the Django project, make sure your virtual environment is activated, then start Django's development server using the command:
  - `python manage.py runserver`
  - The server runs on the default port 8000
  - Django's built-in web server is intended only for local development purposes. When you deploy to a web host, however, Django uses the host's web server instead. The wsgi.py and asgi.py modules in the Django project take care of hooking into the production servers.
  - If you want to use a different port than the default 8000, specify the port number on the command line, such as python manage.py runserver 5000.
- Ctrl+click the http://127.0.0.1:8000/ URL in the terminal output window to open your default browser to that address.
- When you're done, close the browser window and stop the server in VS Code using Ctrl+C as indicated in the terminal output window.

Create a Django app

- In the VS Code Terminal with your virtual environment activated, run the administrative utility's startapp command in your project folder (where manage.py resides):
  - `python manage.py startapp hello`
  - The command creates a folder called `hello` that contains a number of code files and one subfolder. Of these, you frequently work with views.py (that contains the functions that define pages in your web app) and models.py (that contains classes defining your data objects).
- The migrations folder is used by Django's administrative utility to manage database versions as discussed later in this tutorial
- There are also the files apps.py (app configuration), admin.py (for creating an administrative interface), and tests.py (for creating tests), which are not covered here.

Create a Django app

- Modify hello/views.py to match the following code, which creates a single view for the app's home page:

```
from django.http import HttpResponse

def home(request):

    return HttpResponse("Hello, Django!")
```

Create a file, `hello/urls.py`, with the contents below. The `urls.py` file is where you specify patterns to route different URLs to their appropriate views. The code below contains one route to map root URL of the app ("") to the views.home function that you just added to hello/views.py:

```
from django.urls import path

from hello import views

urlpatterns = [

    path("", views.home, name="home"),

]
```

Create a Django app

- The web_project folder also contains a urls.py file, which is where URL routing is actually handled.
- Open `web_project/urls.py` and modify it to match the following code (you can retain the instructive comments if you like). This code pulls in the app's hello/urls.py using `django.urls.include`, which keeps the app's routes contained within the app. This separation is helpful when a project contains multiple apps.

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

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

Use a template to render a page

- In hello/urls.py, add a route to the urlpatterns list:

```
path("hello/<name>", views.hello_there, name="hello_there"),
```

- The first argument to path defines a route `"hello/"` that accepts a variable string called name. The string is passed to the views.hello_there function specified in the second argument to path.
- URL routes are case-sensitive. For example, the route `/hello/<name>` is distinct from `/Hello/<name>`. If you want the same view function to handle both, define paths for each variant.
- The name variable defined in the URL route is given as an argument to the hello_there function.

Use a template to render a page

- In the `web_project/settings.py` file, locate the `INSTALLED_APPS` list and add the following entry, which makes sure the project knows about the app so it can handle templating:

```
- 'hello',
```

- Inside the hello folder, create a folder named templates, and then another subfolder named hello to match the app name (this two-tiered folder structure is typical Django convention).
- In the templates/hello folder, create a file named hello_there.html with the contents below.
- This template contains two placeholders for data values named "name", and "date", which are delineated by pairs of curly braces, {{ and }}
- As you can see, template placeholders can also include formatting, the expressions after the pipe | symbols, in this case using Django's built-in date filter and time filter

Use a template to render a page

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Hello, Django</title>
    </head>
    <body>
        <strong>Hello there, {{ name }}!</strong> It's {{ date | date:"l, d F, Y" }} at {{ date | time:"H:i:s" }}
    </body>
</html>
```

Use a template to render a page

- At the top of views.py, add the following import statement:

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

Also in `views.py`, modify the `hello_there` function to use `django.shortcuts.render` method to load a template and to provide the template context.

```
def hello_there(request, name):
    print(request.build_absolute_uri()) #optional
    return render(
        request,
        'hello/hello_there.html',
        {
            'name': name,
            'date': datetime.now()
        }
    )
```

Serve static files

- Static files are pieces of content that your web app returns as-is for certain requests, such as CSS files. Serving static files requires that the `INSTALLED_APPS` list in settings.py contains `django.contrib.staticfiles`, which is included by default.
- When switching to production, navigate to settings.py, set `DEBUG=False`, and change `ALLOWED_HOSTS = ['*']` to allow specific hosts

Ready the app for static files

- In the project's web_project/urls.py, add the following import statement:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns
```

- Refer to static files in a template:
  - In the `hello` folder, create a folder named `static`.
  - Within the `static` folder, create a subfolder named `hello`, matching the app name:
    - The reason for this extra subfolder is that when you deploy the Django project to a production server, you collect all the static files into a single folder that's then served by a dedicated static file server. The static/hello subfolder ensures that when the app's static files are collected, they're in an app-specific subfolder and won't collide with file from other apps in the same project.
  - In the `static/hello` folder, create a file named `site.css` with the following contents:

```
.message {
        font-weight: 600;
        color: blue;
}
```

Ready the app for static files

- In `templates/hello/hello_there.html`, add the following lines after the `<title>` element. The `{% load static %}` tag is a custom Django template tag set, which allows you to use `{% static %}` to refer to a file like the stylesheet.

```
{% load static %}
<link rel="stylesheet" type="text/css" href="{% static 'hello/site.css' %}" />
```

- Also in `templates/hello/hello_there.html`, replace the contents `<body>` element with the following markup that uses the message style instead of a `<strong>` tag:

```
<span class="message">Hello, there {{ name }}!</span> It's {{ date | date:'l, d F, Y' }} at {{ date | time:'H:i:s' }}.
```

Use the collectstatic command

- For production deployments, you typically collect all the static files from your apps into a single folder using the `python manage.py collectstatic` command.
- The following steps show how this collection is made, although you don't use the collection when running with the Django development server.
  - In `web_project/settings.py`, add the following line that defines a location where static files are collected when you use the `collectstatic` command:
    - `STATIC_ROOT = BASE_DIR / 'static_collected'`
  - In the Terminal, run the command `python manage.py collectstatic` and observe that `hello/site.css` is copied into the top level `static_collected` folder alongside `manage.py`
  - In practice, run collectstatic any time you change static files and before deploying into production

Create multiple templates that extend a base template

- Because most web apps have more than one page, and because those pages typically share many common elements, developers separate those common elements into a base page template that other page templates then extend.
- Create a base page template and styles:
  - A base page template in Django contains all the shared parts of a set of pages, including references to CSS files, script files, and so forth
  - Base templates also define one or more block tags with content that extended templates are expected to override
  - A block tag is delineated by `{% block <name> %}` and `{% endblock %}` in both the base template and extended templates.

Create multiple templates that extend a base template

- In the `templates/hello` folder, create a file named `layout.html` with the contents below, which contains blocks named "title" and "content“

- As you can see, the markup defines a simple nav bar structure with links to Home, About, and Contact pages, which you create in a later section

- Notice the use of Django's {% url %} tag to refer to other pages through the names of the corresponding URL patterns rather than by relative path.

Create multiple templates that extend a base template

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>{% block title %}{% endblock %}</title>
    {% load static %}
    <link rel="stylesheet" type="text/css" href="{% static 'hello/site.css' %}"/>
</head>

<body>
<div class="navbar">
    <a href="{% url 'home' %}" class="navbar-brand">Home</a>
    <a href="{% url 'about' %}" class="navbar-item">About</a>
    <a href="{% url 'contact' %}" class="navbar-item">Contact</a>
</div>

<div class="body-content">
    {% block content %}
    {% endblock %}
    <hr/>
    <footer>
        <p>&copy; 2024</p>
    </footer>
</div>
</body>
</html>
```

Create multiple templates that extend a base template

- Add the following styles to `static/hello/site.css` below the existing "message" style, and save the file.

Create multiple templates that extend a base template

```
.navbar {
    background-color: lightslategray;
    font-size: 1em;
    font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
    color: white;
    padding: 8px 5px 8px 5px;
}

.navbar a {
    text-decoration: none;
    color: inherit;
}

.navbar-brand {
    font-size: 1.2em;
    font-weight: 600;
}

.navbar-item {
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    padding: 5px;
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

Create a code snippet

- Because the three pages you create in the next section extend layout.html, it saves time to create a code snippet to initialize a new template file with the appropriate reference to the base template
- A code snippet provides a consistent piece of code from a single source, which avoids errors that can creep in when using copy-paste from existing code.
- In VS Code, select the File (Windows/Linux) or Code (macOS), menu, then select Preferences > User snippets.
- In the list that appears, select html. (The option may appear as "html.json" in the Existing Snippets section of the list if you've created snippets previously.)
- After VS code opens `html.json`, add the code below within the existing curly braces. (The explanatory comments, not shown here, describe details such as how the `$0` line indicates where VS Code places the cursor after inserting a snippet):

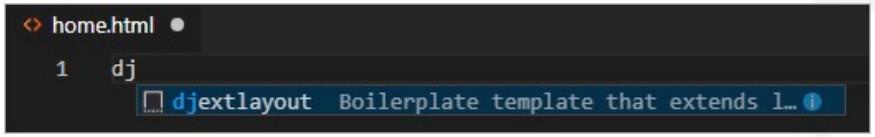
Create a code snippet

```
"Django Tutorial: template extending layout.html": {
    "prefix": "djextlayout",
    "body": [
        "{% extends \"hello/layout.html\" %}",
        "{% block title %}",
        "$0",
        "{% endblock %}",
        "{% block content %}",
        "{% endblock %}"
    ],

    "description": "Boilerplate template that extends layout.html"
},
```

Use the code snippet to add pages

- With the code snippet in place, you can quickly create templates for the Home, About, and Contact pages.
  - In the `templates/hello` folder, create a new file named `home.html`, Then start typing `djext` to see the snippet appear as a completion:



  - At the insertion point in the "title" block, write Home, and in the "content" block, write `<p>Home page for the Visual Studio Code Django tutorial.</p>`, then save the file. These lines are the only unique parts of the extended page template:
  - In the `templates/hello folder`, create `about.html`, use the snippet to insert the boilerplate markup, insert `About us` and `<p>About page for the Visual Studio Code Django tutorial.</p>` in the "title" and "content" blocks, respectively, then save the file.

Use the code snippet to add pages

- Repeat the previous step to create `templates/hello/contact.html` using `Contact us` and `<p>Contact page for the Visual Studio Code Django tutorial.</p>`.
- In the app's `urls.py`, add routes for the `/about and /contact pages`. Be mindful that the name argument to the path function defines the name with which you refer to the page in the `{% url %}` tags in the templates.
  - `path("about/", views.about, name="about"),`
  - `path("contact/", views.contact, name="contact"),`

Work with data, data models, and migrations

- In Django, a model is a Python class, derived from `django.db.models.Model`, that represents a specific database object, typically a table. You place these classes in an app's `models.py` file.
- With Django, you work with your database almost exclusively through the models you define in code. Django's "migrations" then handle all the details of the underlying database automatically as you evolve the models over time.
- The general workflow is as follows:
  - Make changes to the models in your `models.py` file.
  - Run `python manage.py makemigrations` to generate scripts in the `migrations` folder that migrate the database from its current state to the new state.
  - Run `python manage.py migrate` to apply the scripts to the actual database.
- The migration scripts effectively record all the incremental changes you make to your data models over time. By applying the migrations Django updates the database to match your models.

Work with data, data models, and migrations

- When using the db.sqlite3 file, you can also work directly with the database using a tool like the SQLite browser.

- It's fine to add or delete records in tables using such a tool, but avoid making changes to the database schema because the database will then be out of sync with your app's models

- Instead, change the models, run makemigrations, then run migrate.

Define models

- A Django model is again a Python class derived from `django.db.model.Models`, which you place in the app's `models.py` file

- In the database, each model is automatically given a unique ID field named `id`

- All other fields are defined as properties of the class using types from `django.db.models` such as `CharField` (limited text), `TextField` (unlimited text), `EmailField`, `URLField`, `IntegerField`, `DecimalField`, `BooleanField`. `DateTimeField`, `ForeignKey`, and `ManyToMany`, among others.

- Each field takes some attributes, like `max_length`. The `blank=True` attribute means the field is optional; `null=true` means that a value is optional. There is also a choices attribute that limits values to values in an array of data value/display value tuples.

Define models

- For example, add the following class in models.py to define a data model that represents dated entries in a simple message log:

```
from django.db import models
from django.utils import timezone
class LogMessage(models.Model):
    message = models.CharField(max_length=300)
    log_date = models.DateTimeField("date logged")
    def __str__(self):
        """Returns a string representation of a message."""
        date = timezone.localtime(self.log_date)
        return f"'{self.message}' logged on {date.strftime('%A, %d %B, %Y at
%X')}"
```

Migrate the database

- Because you changed your data models by editing `models.py`, you need to update the database itself.

- In VS Code, open a Terminal with your virtual environment activated (use the Terminal: `Create New Terminal command, Ctrl+Shift+`)), navigate to the project folder, and run the following commands:
  - `python manage.py makemigrations`
  - `python manage.py migrate`

Take a look in the migrations folder to see the scripts that makemigrations generates. You can also look at the database itself to see that the schema is updated.

Use the database through the models

- With your models in place and the database migrated, you can store and retrieve data using only your models.
- In this section, you add a form page to the app through which you can log a message. You then modify the home page to display those messages.
- Because you modify many code files here, be mindful of the details.

➢ 1. In the `hello` folder (where you have `views.py`), create a new file named `forms.py` with the following code, which defines a Django form that contains a field drawn from the data model, `LogMessage`:

➢ 2. In the `templates/hello` folder, create a new template named `log_message.html` with the following contents, which assumes that the template is given a variable named `form` to define the body of the form. It then adds a submit button with the label "Log".

Use the database through the models

```
{% extends "hello/layout.html" %}
{% block title %}
    Log a message
{% endblock %}
{% block content %}
    <form method="POST" class="log-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Log</button>
    </form>
{% endblock %}
```

- Note: Django's `{% csrf_token %}` tag provides protection from cross-site request forgeries. See Cross Site Request Forgery protection in the Django documentation for details.

Use the database through the models

- 3. In the app's static/hello/site.css file, add a rule to make the input form wider:

```
input[name=message] {
    width: 80%;
}
```

- 4. In the app's urls.py file, add a route for the new page:

```
path("log/", views.log_message, name="log"),
```

- 5. In views.py, define the view named `log_message` (as referred to by the URL route). This view handles both `HTTP GET` and `POST` cases. In the `GET` case (the `else:` section), it just displays the form that you defined in the previous steps. In the `POST` case, it retrieves the data from the form into a data object (message), sets the `timestamp`, then saves that object at which point it's written to the database:

Use the database through the models

```
# Add these to existing imports at the top of the file:
from django.shortcuts import redirect
from hello.forms import LogMessageForm
from hello.models import LogMessage
# Add this code elsewhere in the file:
def log_message(request):
    form = LogMessageForm(request.POST or None)
    if request.method == "POST":
        if form.is_valid():
            message = form.save(commit=False)
            message.log_date = datetime.now()
            message.save()
            return redirect("home")
    else:
        return render(request, "hello/log_message.html", {"form": form})
```

Use the database through the models

- 6. One more step before you're ready to try everything out! In templates/hello/layout.html, add a link in the "navbar" div for the message logging page:

```
<!-- Insert below the link to Home -->
<a href="{% url 'log' %}" class="navbar-item">Log Message</a>
```

- 7. Run the app and open a browser to the home page. Select the Log Message link on the nav bar, which should display the message logging page.
- 8. Enter a message, select Log, and you should be taken back to the home page. The home page doesn't yet show any of the logged messages yet (which you remedy in a moment). Feel free to log a few more messages as well. If you want, peek in the database using a tool like SQLite Browser to see that records have been created. Open the database as read-only, or otherwise remember to close the database before using the app, otherwise the app will fail because the database is locked.
- 9. Stop the app when you're done.

Use the database through the models

- 10. Now modify the home page to display the logged messages. Start by replacing the contents of app's `templates/hello/home.html` file with the markup below. This template expects a context variable named `message_list`. If it receives one (checked with the `{% if message_list %}` tag), it then iterates over that list (the `{% for message in message_list %}` tag) to generate table rows for each message. Otherwise the page indicates that no messages have yet been logged.

Use the database through the models

```
{% extends "hello/layout.html" %}
{% block title %}
    Home
{% endblock %}
{% block content %}
    <h2>Logged messages</h2>

    {% if message_list %}
        <table class="message_list">
            <thead>
            <tr>
                <th>Date</th>
                <th>Time</th>
                <th>Message</th>
            </tr>
            </thead>
            <tbody>
            {% for message in message_list %}
                <tr>
                    <td>{{ message.log_date | date:'d M Y' }}</td>
                    <td>{{ message.log_date | time:'H:i:s' }}</td>
                    <td>
                        {{ message.message }}
                    </td>
                </tr>
            {% endfor %}
            </tbody>
        </table>
    {% else %}
        <p>No messages have been logged. Use the <a href="{% url 'log' %}">Log Message form</a>.</p>
    {% endif %}
{% endblock %}
```

Use the database through the models

- 11. In static/hello/site.css, add a rule to format the table a little:

```
.message_list th,td {
    text-align: left;
    padding-right: 15px;
}
```

- 12. In views.py, import Django's generic `ListView` class, which we'll use to implement the home page:

```
from django.views.generic import ListView
```

- 13. Also in `views.py`, replace the home function with a class named `HomeListView`, derived from `ListView`, which ties itself to the `LogMessage` model and implements a function `get_context_data` to generate the context for the template.

Use the database through the models

- 11. In static/hello/site.css, add a rule to format the table a little:

```
.message_list th,td {
    text-align: left;
    padding-right: 15px;
}
```

- 12. In views.py, import Django's generic `ListView` class, which we'll use to implement the home page:

```
from django.views.generic import ListView
```

- 13. Also in `views.py`, replace the home function with a class named `HomeListView`, derived from `ListView`, which ties itself to the `LogMessage` model and implements a function `get_context_data` to generate the context for the template.

Use the database through the models

```
# Remove the old home function if you want; it's no longer used

class HomeListView(ListView):
    """Renders the home page, with a list of all messages."""
    model = LogMessage

    def get_context_data(self, **kwargs):
        context = super(HomeListView, self).get_context_data(**kwargs)
        return context
```

Use the database through the models

- 14. In the app's `urls.py`, import the data model:

```
from hello.models import LogMessage
```

- 15. Also in `urls.py`, make a variable for the new view, which retrieves the five most recent `LogMessage` objects in descending order (meaning that it queries the database), and then provides a name for the data in the template context (`message_list`), and identifies the template to use:

```
home_list_view = views.HomeListView.as_view(
    queryset=LogMessage.objects.order_by("-log_date")[:5],  # :5 limits the
results to the five most recent
    context_object_name="message_list",
    template_name="hello/home.html",
)
```

Use the database through the models

- 16. In `urls.py`, modify the path to the home page to use the `home_list_view` variable:

```
# Replace the existing path for ""
    path("", home_list_view, name="home"),
```

- 17. Start the app and open a browser to the home page, which should now display messages
- 18. Stop the app when you're done.

Create a requirements.txt file for the environment

- When you share your app code through source control or some other means, it doesn't make sense to copy all the files in a virtual environment because recipients can always recreate that environment themselves.
- Accordingly, developers typically omit the virtual environment folder from source control and instead describe the app's dependencies using a `requirements.txt` file.
- Although you can create the file by hand, you can also use the pip freeze command to generate the file based on the exact libraries installed in the activated environment:
  1. With your chosen environment selected using the Python: Select Interpreter command, run the Terminal: Create New Terminal command (Ctrl+Shift+`)) to open a terminal with that environment activated.
  2. In the terminal, run `pip freeze > requirements.txt` to create the `requirements.txt` file in your project folder.
- Anyone (or any build server) that receives a copy of the project needs only to run the `pip install -r requirements.txt` command to reinstall the packages on which the app depends within the active environment.

Create a superuser and enable the administrative interface

- By default, Django provides an administrative interface for a web app that's protected by authentication.
- The interface is implemented through the built-in `django.contrib.admin` app, which is included by default in the project's `INSTALLED_APPS` list (`settings.py`), and authentication is handled with the built-in `django.contrib.auth` app, which is also in `INSTALLED_APPS` by default.
- Perform the following steps to enable the administrative interface:

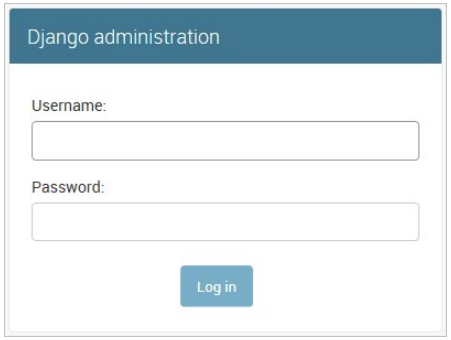
1. Create a superuser account in the app by opening a Terminal in VS Code for your virtual environment, then running the command `python manage.py createsuperuser --username=<username> --email=<email>`, replacing `<username>` and `<email>`, of course, with your personal information. When you run the command, Django prompts you to enter and confirm your password.
2. Add the following URL route in the project-level `urls.py` (`web_project/urls.py` in this tutorial) to point to the built-in administrative interface:

```
# This path is included by default when creating the app
 path("admin/", admin.site.urls),
```

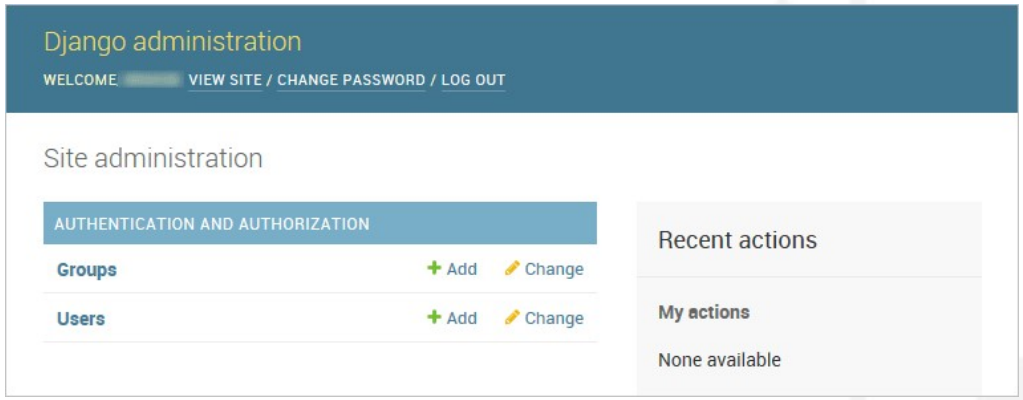
3. Run the server, then open a browser to the app's /admin page (such as `http://127.0.0.1:8000/admin` when using the development server).

Create a superuser and enable the administrative interface

- 4. A login page appears, courtesy of `django.contrib.auth`. Enter your superuser credentials.

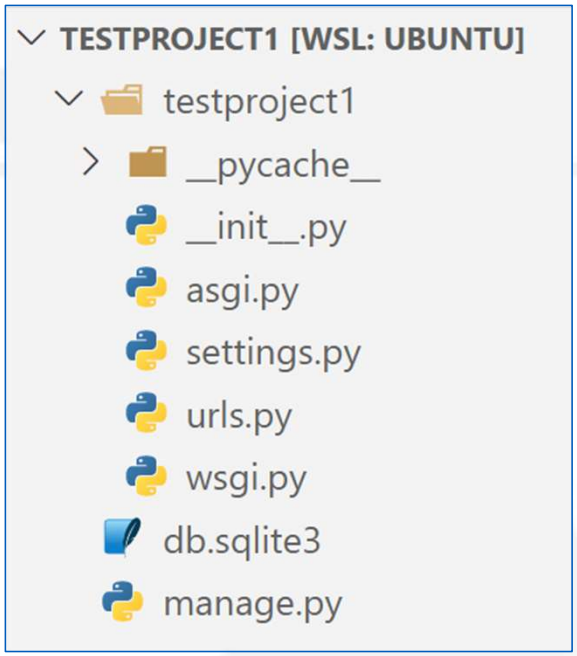


- 5. Once you're authenticated, you see the default administration page, through which you can manage users and groups:



Cấu trúc dự án

- `__pycache__`: Thư mục lưu trữ bytecode được biên dịch khi tạo dự án, mục đích làm dự án bắt đầu nhanh ơn bằng cách lưu code đã biên dịch vào bộ nhớ cache để sau đó nó có thể dễ dàng thực thi
- `__init__.py`: File chỉ định những gì sẽ chạy khi Django khởi chạy lần đầu
- `asgi.py`: File cho phép tùy chọn Asynchronous Server Gateway Interface (Giao diện cổng máy chủ không đồng bộ) chạy
- `settings.py`: File quan trọng chứa cài đặt của dự án
- `urls.py`: File cho Django biết trang nào sẽ hiển thị theo yêu cầu của trình duyệt hoặc URL.
- `wsgi.py`: (Web Server Gateway Interface – Giao diện cổng máy chủ web) giúp Django phục vụ trang web

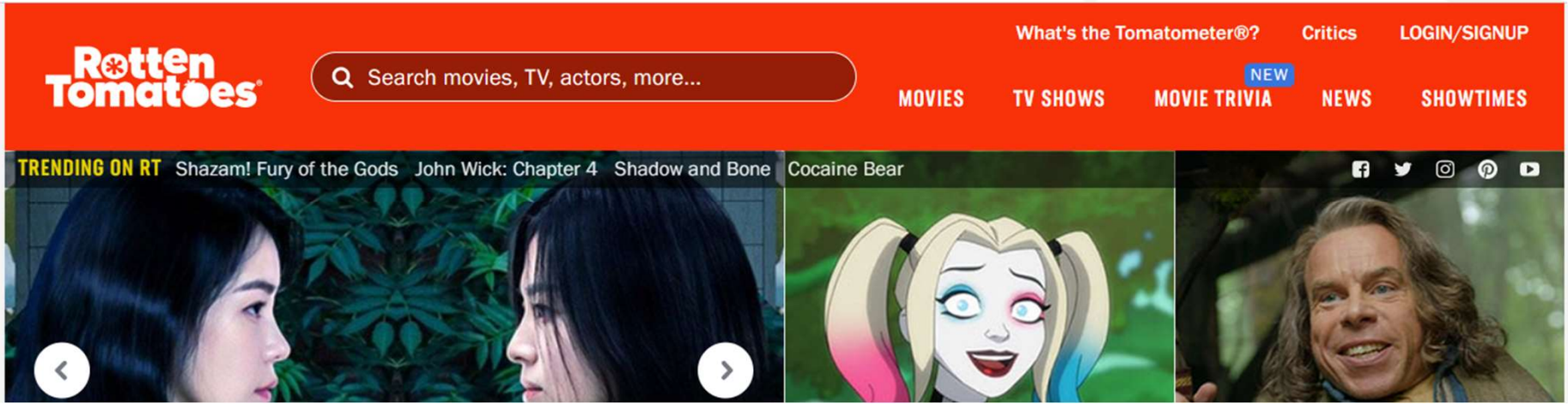


Cấu trúc dự án (settings.py)

- Các thuộc tính trong file `settings.py`:
  - `BASE_DIR`: Xác định vị trí của dự án trên máy
  - `SECRET_KEY`: Được sử dụng khi có dữ liệu vào và ra khỏi trang web. Không nên chia sẻ.
  - `DEBUG`: Dự án đang chạy ở chế độ gỡ lỗi hay không
  - `INSTALLED_APPS`: Cho phép đưa các đoạn code khác nhau vào dự án
  - `MIDDLEWARE`: Đề cập đến các chức năng tích hợp của Djando để xử lý các yêu cầu/ phản hồi của ứng dụng, bao gồm xác thực, phiên (session) và bảo mật
  - `ROOT_URLCONF`: Chỉ định vị trí của các URL
  - `TEMPLATES`: Xác định lớp công cụ mẫu, danh sách các thư mục và công cụ sẽ tìm các file nguồn template và cài đặt template cụ thể
  - `AUTH_PASSWORD_VALIDATORS`: Cho phép chỉ định các xác thực mong muốn trên mật khẩu, vd độ dài tối thiểu.
  - Ngoài ra còn nhiều thuộc tính khác như `LANGUAGE_CODE`, `TIME_ZONE`

Cấu trúc dự án (ứng dụng - app)

- 1 dự án Django có thể chứa 1 hoặc nhiều ứng dụng hoạt động cùng nhau để hỗ trợ ứng dụng web.
- App giống 1 phần của trang web. Có thể code toàn bộ bằng 1 app, nhưng chia nó thành nhiều app, mỗi app có một chức năng rõ ràng sẽ hữu ích hơn.
- VD: Trang đánh giá phim như Rotten Tomatoes có 1 app liệt kê phim, 1 app để liệt kê tin tức, 1 app để thanh toán, 1 app để xác thực người dùng, ...



Cấu trúc dự án (ứng dụng – app)

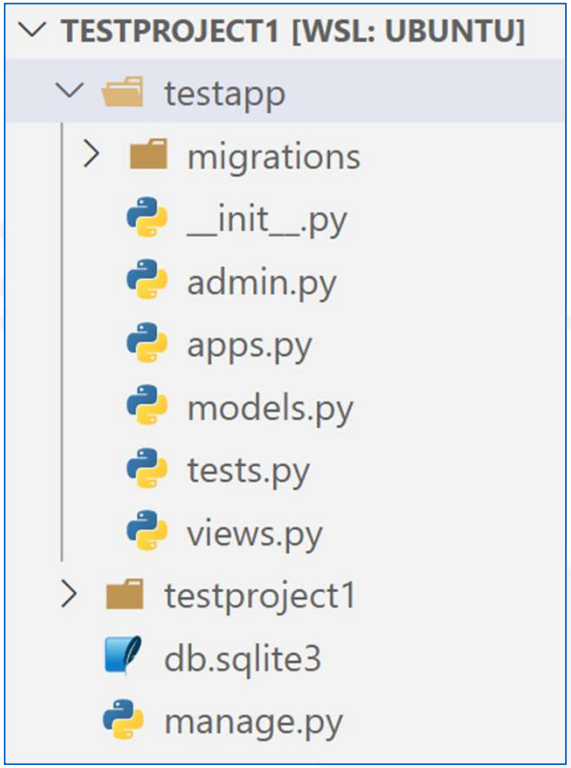
- Tạo mới app bằng câu lệnh:

```
(myenv) $ python3 manage.py startapp <name of app>
```

- Mặc dù app mới tạo tồn tại trong dự án của Django nhưng phải thêm vào `settings.py` thì Django mới nhận ra

settings.py

```
...
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp'
]
...
```



URLs

- urls.py

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views as testviews

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

Ref: [django.urls functions for use in URLconfs | Django documentation | Django (djangoproject.com)]

URLs

- urls.py

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views as testviews

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

views.py

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

def home(request):
    return HttpResponse('<h1>Welcome to Home Page</h1>')

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

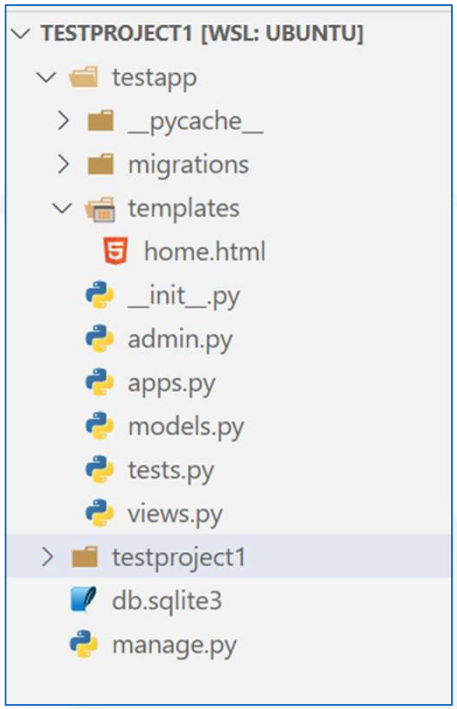
Ref: [django.urls functions for use in URLconfs | Django documentation | Django (djangoproject.com)]

Tạo trang HTML với Templates

views.py

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

def home(request):
    return render(request, 'home.html')
```



Tạo trang HTML với Templates

- Truyền dữ liệu vào templates

views.py

```
...
def home(request):
    return render(request, 'home.html', {'name':'Quyen Nguyen'})
...
```

home.html

```
...
<body>
    <h1>Welcome to Home Page, {{ name }}</h1>
    <h2>This is the full home page</h2>
</body>
...
```

Ref: Django Template Language (DTL)

Models

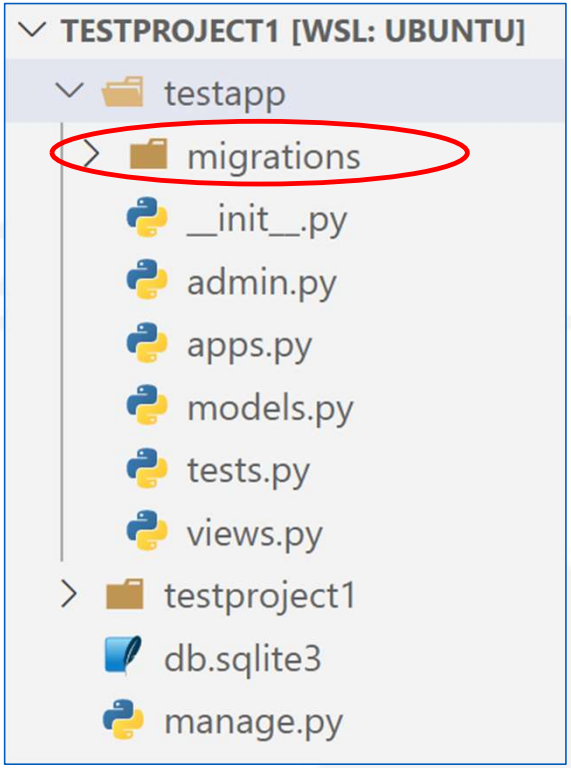
```python
from django.db import models

class Movie(models.Model):
    title = models.CharField(max_length=100)
    description = models.CharField(max_length=250)
    image = models.ImageField(upload_to='movie/images/')
    url = models.URLField(blank=True)
```

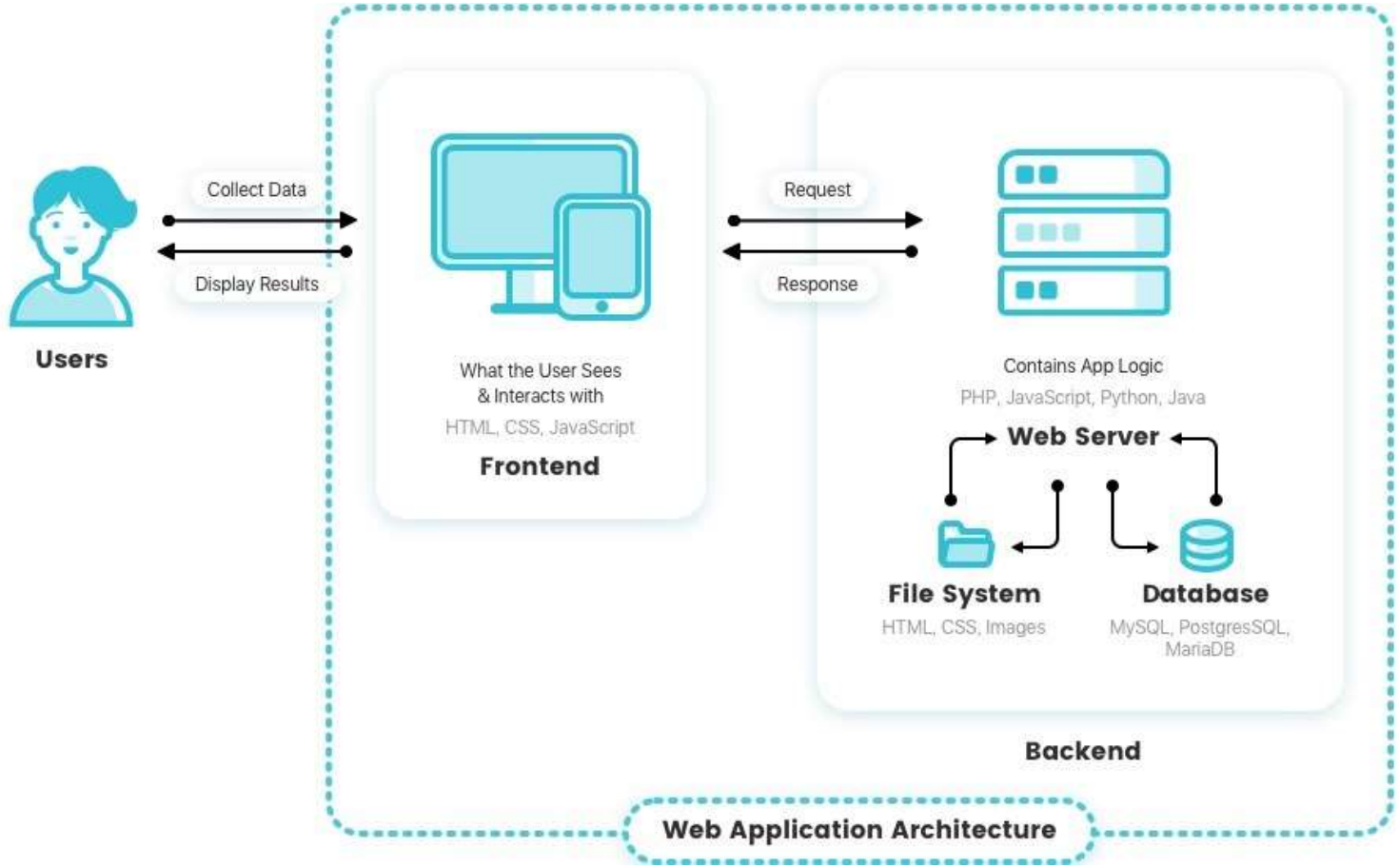
- Module `models` giúp xác định và ánh xa các trường của mô hình vào CSDL
- Lớp Movie kế thừa từ lớp Model. Lớp Model cho phép tương tác với CSDL, tạo bảng, truy xuất và thực hiện các thay đổi đối với dữ liệu trong CSDL
- Ref: Model field reference | Django documentation | Django (djangoproject.com)

Migrations

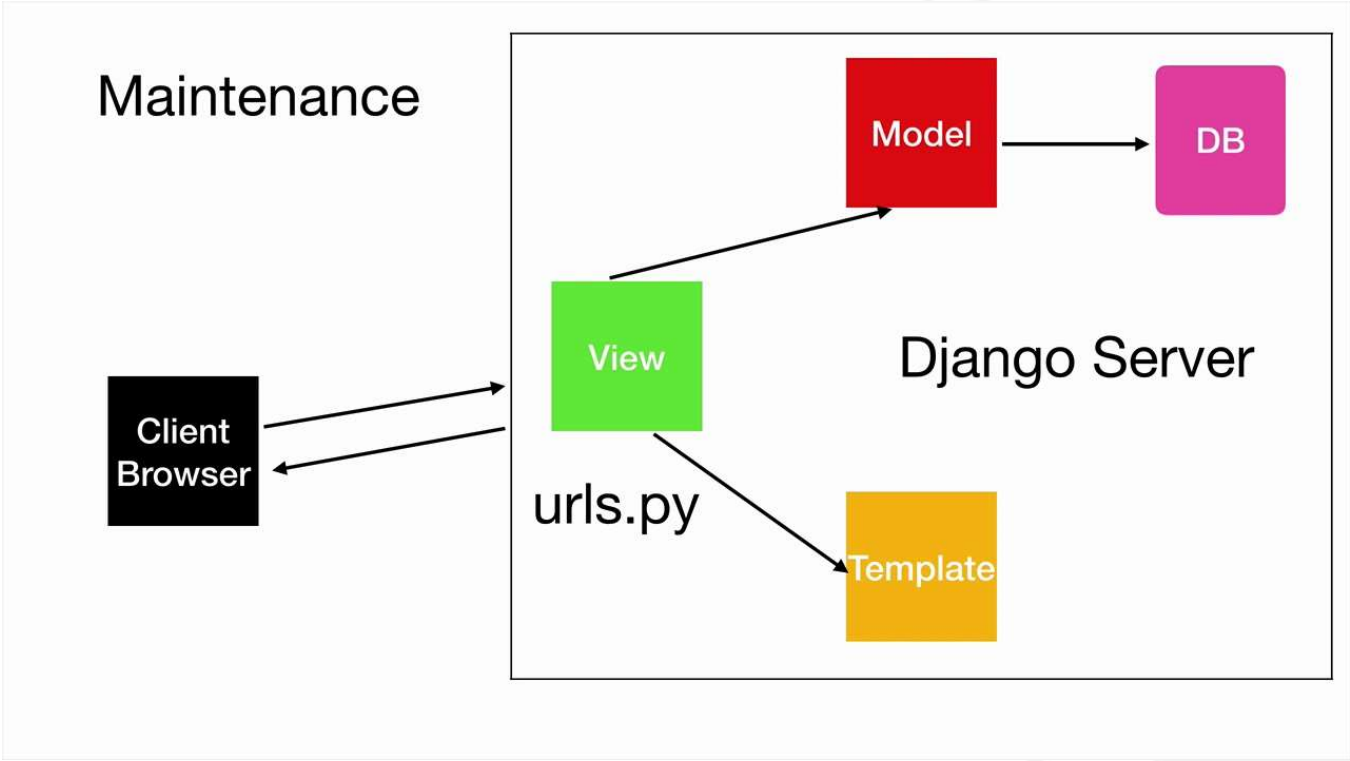
- Migrations cho phép tạo 1 lược đồ CSDL dựa trên code model
- Mỗi khi model thay đổi (thêm trường, đổi tên trường), migration sẽ được tạo ra → theo dõi quá trình phất triển của lược đồ CSDL (dưới dạng hệ thống kiểm soát phiên bản)



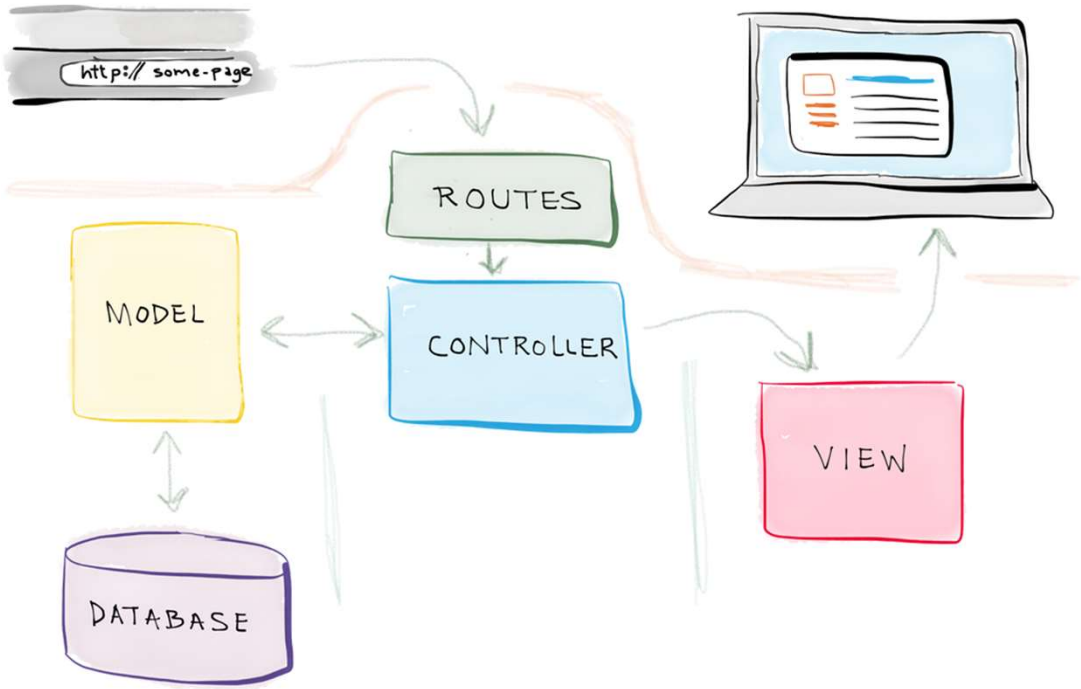
Một ứng dụng web hoạt động như thế nào?

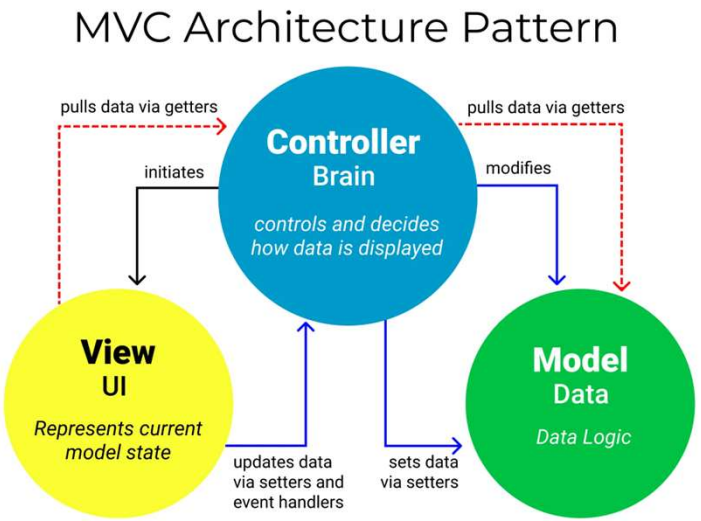


Mô hình MVT


Maintenance
Model
DB
View
Django Server
Client
Browser
urls.py
Template

Mô hình MVC


http:// some-page
ROUTES
MODEL
CONTROLLER
VIEW
DATABASE


MVC Architecture Pattern
pulls data via getters
Controller
Brain
controls and decides how data is displayed
initiates
modifies
pulls data via getters
View
UI
Represents current model state
updates data via setters and event handlers
sets data via setters
Model
Data
Data Logic