Module 16: Python DB and Framework

Q 1. Introduction to embedding HTML within Python using web frameworks like Django or Flask.

Ans.

Embedding HTML within Python using web frameworks like **Django** or **Flask** allows developers to build dynamic web applications. Both frameworks follow the Model-View-Template (MVT) or Model-View-Controller (MVC) pattern, separating the business logic (Python) from the user interface (HTML). Here's an introduction:

**1. What is HTML Embedding in Web Frameworks?**

It refers to **injecting HTML code into web pages** served by Python backend logic. This is typically done through **template engines**, which allow HTML files to be dynamically rendered with Python variables and logic.

**2. Using Flask for HTML Embedding**

a. Install Flask

pip install flask

b. Basic Flask App with Embedded HTML

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

return render\_template("index.html", name="Mohit")

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

c. templates/index.html

<!DOCTYPE html>

<html>

<head>

<title>Flask Page</title>

</head>

<body>

<h1>Hello, {{ name }}!</h1>

</body>

</html>

**3. Using Django for HTML Embedding**

a. Create a Django Project

django-admin startproject mysite

cd mysite

python manage.py startapp myapp

b. In views.py of myapp

from django.shortcuts import render

def home(request):

return render(request, 'home.html', {'name': 'Mohit'})

c. In templates/home.html

<!DOCTYPE html>

<html>

<head>

<title>Django Page</title>

</head>

<body>

<h1>Welcome, {{ name }}!</h1>

</body>

</html>

**d. Configure settings.py**

Make sure TEMPLATES and 'myapp' are added correctly.

**Q 2. Generating dynamic HTML content using Django templates.**

Ans.

1. How Django Templates Work

View (Python logic) → Template (HTML) → Rendered Web Page

**2. Basic Setup**

Assume you already have a Django project and app. If not:

django-admin startproject mysite

cd mysite

python manage.py startapp myapp

Add 'myapp' to INSTALLED\_APPS in settings.py.

3. Creating a View with Dynamic Content

myapp/views.py

from django.shortcuts import render

def homepage(request):

context = {

'user': 'Mohit',

'languages': ['Python', 'JavaScript', 'C++'],

'is\_logged\_in': True

}

return render(request, 'home.html', context)

**4. Creating the Template**

Create a templates folder in your app directory, and a file named home.html inside it:

**myapp/templates/home.html**

<!DOCTYPE html>

<html>

<head>

<title>Dynamic Page</title>

</head>

<body>

<h1>Hello, {{ user }}!</h1>

{% if is\_logged\_in %}

<p>Welcome back!</p>

{% else %}

<p>Please log in to continue.</p>

{% endif %}

<h2>Your favorite programming languages:</h2>

<ul>

{% for lang in languages %}

<li>{{ lang }}</li>

{% endfor %}

</ul>

</body>

</html>

**5. URL Configuration**

In **myapp/urls.py** (create it if it doesn’t exist):

from django.urls import path

from . import views

urlpatterns = [

path('', views.homepage, name='homepage'),

]

And in **mysite/urls.py**, include the app URLs:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('myapp.urls')),

]

6. **Run the Server**

python manage.py runserver

Q 3. Integrating CSS with Django templates.

Ans.

Integrating **CSS with Django templates** enhances the appearance of your web pages by applying custom styling. In Django, CSS files (along with JavaScript, images, etc.) are part of what's called **static files**.

Here’s a clear step-by-step guide to integrating CSS with Django templates:

**🧭 1. Folder Structure**

Make sure your app has the following structure:

myapp/

├── static/

│ └── myapp/

│ └── style.css

├── templates/

│ └── home.html

├── views.py

Note: myapp/static/myapp/style.css is the recommended location. Django finds static files inside each app’s static/ folder.

**⚙️ 2. Enable Static Files in settings.py**

In your **settings.py**, make sure these lines exist:

# Already present by default

STATIC\_URL = '/static/'

# Optional but useful for development

STATICFILES\_DIRS = [BASE\_DIR / 'static']

**📝 3. Create the CSS File**

**myapp/static/myapp/style.css**

body {

font-family: Arial, sans-serif;

background-color: #f0f0f0;

padding: 20px;

}

h1 {

color: #333366;

}

ul {

background-color: white;

padding: 10px;

border-radius: 5px;

}

**4. Link CSS in the HTML Template**

**myapp/templates/home.html**

<!DOCTYPE html>

<html>

<head>

<title>Styled Page</title>

{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'myapp/style.css' %}">

</head>

<body>

<h1>Hello, {{ user }}!</h1>

{% if is\_logged\_in %}

<p>Welcome back!</p>

{% else %}

<p>Please log in to continue.</p>

{% endif %}

<h2>Your favorite programming languages:</h2>

<ul>

{% for lang in languages %}

<li>{{ lang }}</li>

{% endfor %}

</ul>

</body>

</html>

The key line is:

{% load static %}

<link rel="stylesheet" href="{% static 'myapp/style.css' %}">

**5. Run the Server and Check**

python manage.py runserver

Open your browser and go to http://127.0.0.1:8000/. You’ll see the page with styles applied from your CSS file.

Q 4. How to serve static files (like CSS, JavaScript) in Django.

Ans.Serving **static files** (like CSS, JavaScript, images) in **Django** is essential for styling and interactivity. Here's a complete, beginner-friendly guide to serving static files during development.

**📁 1. What are Static Files in Django?**

Static files are **assets that don't change** (e.g., CSS, JS, images) and are served directly to the browser without needing server-side rendering.

**2. Folder Structure**

Recommended structure inside your Django app:

myapp/

├── static/

│ └── myapp/

│ ├── style.css

│ └── script.js

├── templates/

│ └── home.html

Place your static files inside:

myapp/static/myapp/

**3. Settings Configuration**

**In settings.py:**

# URL prefix for static files

STATIC\_URL = '/static/'

# (Optional) For global static files in the project root

STATICFILES\_DIRS = [BASE\_DIR / 'static']

# (For production, not needed in dev)

# STATIC\_ROOT = BASE\_DIR / 'staticfiles'

**📝 4. Loading Static Files in Templates**

At the top of your HTML template:

{% load static %}

Then include static files like this:

<link rel="stylesheet" href="{% static 'myapp/style.css' %}">

<script src="{% static 'myapp/script.js' %}"></script>

**5. Example HTML Template**

**myapp/templates/home.html**

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Static Example</title>

<link rel="stylesheet" href="{% static 'myapp/style.css' %}">

</head>

<body>

<h1>Hello from Django with CSS!</h1>

<script src="{% static 'myapp/script.js' %}"></script>

</body>

</html>

**6.Run Development Server**

python manage.py runserver

Now visit http://127.0.0.1:8000/ and your static files should be served.

**7. Troubleshooting Tips**

| **Issue** | **Solution** |
| --- | --- |
| Static file not loading | Ensure {% load static %} is used and path is correct |
| CSS/JS changes not reflecting | Use **Shift + Refresh** to clear browser cache |
| 404 error for static file | Check file path, filename spelling, and app structure |

**8. Serving Static Files in Production**

In production, Django doesn't serve static files by itself. You must:

1. Run:
2. python manage.py collectstatic
3. Configure a web server like **Nginx** or **Apache** to serve files from the STATIC\_ROOT directory.

Q 5. Using JavaScript for client-side interactivity in Django templates.

Ans.

Using **JavaScript for client-side interactivity** in Django templates allows you to make your web pages dynamic and responsive without reloading them from the server. This includes things like toggling menus, validating forms, showing/hiding content, or even making AJAX requests.

Here’s how to integrate JavaScript in a Django project step-by-step:

**1. Folder Structure**

Place your JavaScript inside the app’s static/ directory like this:

myapp/

├── static/

│ └── myapp/

│ └── script.js

├── templates/

│ └── home.html

**2. Example JavaScript File**

**myapp/static/myapp/script.js**

function greetUser() {

alert("Welcome to the Django site!");

}

function toggleMessage() {

const msg = document.getElementById("message");

msg.style.display = msg.style.display === "none" ? "block" : "none";

}

**3. Link JS in Django Template**

**myapp/templates/home.html**

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Django + JS Example</title>

<script src="{% static 'myapp/script.js' %}"></script>

</head>

<body>

<h1 onclick="greetUser()">Click me to get a greeting</h1>

<button onclick="toggleMessage()">Toggle Message</button>

<p id="message" style="display: none;">This is a hidden message!</p>

</body>

</html>

**4. Key Concepts**

| **Concept** | **How It Works** |
| --- | --- |
| External JS file | Stored in static/ and linked with {% static %} |
| DOM Manipulation | Use document.getElementById, querySelector, etc. |
| Event handling | Add onclick, onchange, etc., in HTML or JS file |
| Interactive behavior | Show/hide, validate, animate, etc. |

**5. Testing**

Run the Django development server:

python manage.py runserver

Open http://127.0.0.1:8000/ and interact with your elements.

**6. Bonus: Use JavaScript with Django Context**

Django passes dynamic content to templates using variables, and you can access them in JavaScript like this:

<script>

let username = "{{ user|escapejs }}"; // Pass Django context into JS

console.log("Logged in as: " + username);

</script>

**7. Advanced: AJAX Example (Optional)**

You can use JavaScript with fetch() or XMLHttpRequest to get data without reloading the page.

fetch('/api/data/')

.then(response => response.json())

.then(data => {

console.log("Data received:", data);

});

Then, in Django, you create a view that returns JSON:

from django.http import JsonResponse

def api\_data(request):

return JsonResponse({'message': 'Hello from Django!'})

Q 6. Linking external or internal JavaScript files in Django.

Ans.

Linking **external or internal JavaScript files** in Django is done through the **static files system**. Whether the JS file is hosted **locally (internal)** or from a **CDN (external)**, Django provides a consistent way to include it in your HTML templates.

**1. Internal JavaScript (Local JS Files)**

**Step-by-Step:**

**a. Project structure**

myapp/

├── static/

│ └── myapp/

│ └── script.js

├── templates/

│ └── home.html

**b. JavaScript file: script.js**

**myapp/static/myapp/script.js**

console.log("Script loaded!");

**c. Template file: home.html**

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Django JS Example</title>

<script src="{% static 'myapp/script.js' %}"></script>

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>

**Important:**

* You must add {% load static %} at the top of your template.
* Use {% static 'path/to/file.js' %} to link local JS files.

**2. External JavaScript (CDN or Remote URLs)**

**Just use standard HTML <script> tags:**

<!DOCTYPE html>

<html>

<head>

<title>External JS</title>

<!-- Link to an external JS file (e.g., jQuery CDN) -->

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

</head>

<body>

<h1>Hello with jQuery!</h1>

<script>

$(document).ready(function() {

console.log("jQuery loaded!");

});

</script>

</body>

</html>

**Tips & Best Practices**

| **Practice** | **Recommendation** |
| --- | --- |
| Organize JS | Keep JS files in static/myapp/ folders |
| Cache control | Use {% static %} for Django to handle cache busting |
| Combine JS | For performance, combine/minify multiple scripts |
| Script loading | Place <script> tags just **before </body>** for better performance |

**Testing**

1. Run:
2. python manage.py runserver
3. Visit http://127.0.0.1:8000/
4. Open **browser console** to confirm: "Script loaded!" appears.

Q 7. Overview of Django: Web development framework.

Ans.

**What is Django?**

**Django** is a **high-level, open-source web framework** for Python that enables rapid development of secure and maintainable websites. It follows the **Model-View-Template (MVT)** architecture and emphasizes **reusability**, **scalability**, and **clean code**.

Created in 2005, Django is used by companies like Instagram, Pinterest, Mozilla, and NASA.

**Key Features**

| **Feature** | **Description** |
| --- | --- |
| 🔐 Built-in Security | Protects against CSRF, XSS, SQL Injection, clickjacking, etc. |
| ⚙️ Admin Interface | Auto-generates a fully-featured admin panel from your models. |
| 📦 Batteries-Included | Includes tools for authentication, ORM, routing, sessions, forms, etc. |
| 🧩 Template System | Dynamically render HTML with Django Templates. |
| 🚀 Scalable | Works for both small and large-scale applications. |
| 🌍 ORM (Object Relational Mapper) | Interact with databases using Python objects instead of SQL. |

**Architecture: MVT Pattern**

| **Component** | **Role** |
| --- | --- |
| **Model** | Defines the structure of your data (database). |
| **View** | Handles business logic and fetches data. |
| **Template** | Displays data in HTML with Django’s template engine. |

**How Django Works (Simplified)**

1. **User requests a URL**
2. **URLconf** matches it to a **View**
3. **View** interacts with the **Model** (database)
4. **View** returns an HTML **Template**
5. Browser displays the page

**Basic Project Example**

**1. Create Project & App**

django-admin startproject mysite

cd mysite

python manage.py startapp myapp

**2. Define a View (myapp/views.py)**

from django.http import HttpResponse

def home(request):

return HttpResponse("Hello, Django!")

**3. Map URL (myapp/urls.py)**

from django.urls import path

from . import views

urlpatterns = [

path('', views.home),

]

Include it in **mysite/urls.py**:

from django.urls import include, path

urlpatterns = [

path('', include('myapp.urls')),

]

**Tools and Concepts Built into Django**

* ✅ Forms and form validation
* ✅ Authentication system (users, login/logout)
* ✅ Middleware support
* ✅ Signal system (event handling)
* ✅ Static and media file management
* ✅ Testing framework

**Why Use Django?**

* Fast development
* Secure by default
* Admin panel out of the box
* Large ecosystem and community
* Well-documented and maintained

Q 8. Advantages of Django (e.g.,scalability, security).

Ans.

**1. Security**

* Django has **built-in protection** against common security threats:
  + SQL injection
  + Cross-site scripting (XSS)
  + Cross-site request forgery (CSRF)
  + Clickjacking
* It handles **user authentication**, password hashing, and session management securely by default.

**2. Scalability**

* Django can scale to support high-traffic websites (e.g., Instagram).
* Built-in support for:
  + Caching (Memcached, Redis)
  + Load balancing
  + Database optimization
* Designed for apps that grow over time.

**3. Batteries-Included Philosophy**

* Everything you need is included:
  + Admin panel
  + Authentication system
  + ORM (Object Relational Mapping)
  + Templating engine
  + Routing
  + Forms & validation
  + File handling

**4. Rapid Development**

* You can build and deploy apps **quickly**.
* Auto-generation of models, forms, admin pages, etc.
* DRY principle (Don't Repeat Yourself) is followed.

Q 9.Django vs. Flask comparison: Which to choose and why.

Ans.

**1. Basic Overview**

| **Feature** | **Django** | **Flask** |
| --- | --- | --- |
| Type | Full-stack framework | Micro (minimalist) framework |
| Philosophy | “Batteries-included” | “Do-it-yourself” |
| Project Size | Best for medium to large apps | Great for small to medium apps |

**2. Architecture**

| **Aspect** | **Django** | **Flask** |
| --- | --- | --- |
| Structure | Enforces MVT (Model-View-Template) | Flexible, you design the structure |
| ORM | Built-in ORM | No ORM by default (you can add SQLAlchemy) |
| Admin Panel | Built-in admin interface | No admin, must build or integrate manually |

**3. Features Comparison**

| **Feature** | **Django** | **Flask** |
| --- | --- | --- |
| URL Routing | Built-in | Built-in |
| Form Handling | Built-in (forms, ModelForm) | Use libraries like WTForms |
| Security | High (many features by default) | Low-to-medium (requires manual setup) |
| Authentication | Built-in | Must implement or use extensions |
| REST APIs | Django REST Framework (DRF) | Flask-RESTful or Flask-RESTX |
| Templates | Django Template Engine | Jinja2 (same as Django but more flexible) |

**4. Development Speed**

* **Django**: Faster for large projects due to built-in tools.
* **Flask**: Fast for small apps; flexible but more work for big apps.

**5. Learning Curve**

| **Django** | **Flask** |
| --- | --- |
| Steeper due to more built-in tools | Easier for beginners to grasp |
| Great for learning full-stack dev | Great for learning fundamentals |

**Use Flask if:**

* You’re building a small app, prototype, or API.
* You want full control over architecture and components.
* You want lightweight, fast performance with less overhead.

**Examples**: Microservices, REST APIs, IoT dashboards, simple web tools.

**Final Recommendation**

| **Goal** | **Recommended Framework** |
| --- | --- |
| Rapid development of full apps | ✅ **Django** |
| Lightweight, flexible web app/API | ✅ **Flask** |
| Built-in admin and security | ✅ **Django** |
| More control with fewer dependencies | ✅ **Flask** |

Q 10. Understanding the importance of a virtual environment in Python projects.

Ans.

**🧪 What is a Virtual Environment?**

A **virtual environment** is an isolated environment in which you can install Python packages **specific to a project**, without affecting other projects or your system-wide Python installation.

**Why is a Virtual Environment Important?**

**1. Project Isolation**

* Keeps dependencies **separate** for each project.
* Avoids version conflicts between different projects.

Example: One project uses Django 3.2 and another uses Django 4.0 — virtual environments allow both to coexist.

**2. No System Pollution**

* Prevents installing packages globally, which may:
  + Break system tools.
  + Cause permission errors.
  + Lead to hard-to-debug conflicts.

**3. Reproducibility**

* Helps generate a requirements.txt file for consistent environments on other machines or servers.

pip freeze > requirements.txt

Then recreate with:

pip install -r requirements.txt

**4. Cleaner Deployment**

* Hosting services (like Heroku, PythonAnywhere, AWS) expect isolated environments.
* Easier deployment with no unexpected system dependencies.

Q 11. Using venv or virtualenv to create isolated environments.

Ans.

**1. Using venv (Recommended, built-in from Python 3.3+)**

**Step 1: Create a virtual environment**

python -m venv myenv

* This creates a folder named myenv containing the isolated environment.

**Step 2: Activate the virtual environment**

* **Windows:**

myenv\Scripts\activate

* **macOS/Linux:**

source myenv/bin/activate

* Your terminal prompt will change to show (myenv) indicating the environment is active.

**Step 3: Install packages inside the environment**

pip install django

**Step 4: Deactivate when done**

deactivate

**2. Using virtualenv (Third-party, works with Python 2 & 3)**

**Step 1: Install virtualenv (if not installed)**

pip install virtualenv

**Step 2: Create a virtual environment**

virtualenv myenv

* Similar to venv, creates a folder myenv.

**Step 3: Activate the environment**

* Same commands as for venv above.

**Step 4: Use pip to install packages inside**

pip install flask

**Step 5: Deactivate**

deactivate

**Key Differences**

| **Feature** | **venv** | **virtualenv** |
| --- | --- | --- |
| Included with Python? | Yes (Python 3.3+) | No (needs installation) |
| Python version support | Python 3 only | Python 2 and 3 support |
| Features | Basic virtual environments | Some extra features (e.g., faster env creation, more options) |
| Usage complexity | Simpler | Slightly more flexible |

Q 12. Steps to create a Django project and individual apps within the project.

Ans.

**Step 1: Set up a Virtual Environment (optional but recommended)**

python -m venv venv

# Activate it:

# Windows

venv\Scripts\activate

# macOS/Linux

source venv/bin/activate

**Step 2: Install Django**

pip install django

**Step 3: Create a New Django Project**

django-admin startproject myproject

* This creates a folder myproject/ with the following structure:

myproject/

manage.py

myproject/

\_\_init\_\_.py

settings.py

urls.py

asgi.py

wsgi.py

**Step 4: Navigate into the Project Directory**

cd myproject

**Step 5: Create a New App within the Project**

python manage.py startapp myapp

* This creates an app folder myapp/ with files like:

myapp/

migrations/

\_\_init\_\_.py

admin.py

apps.py

models.py

tests.py

views.py

**Step 6: Register the App in Project Settings**

* Open myproject/settings.py
* Find the INSTALLED\_APPS list and add your app:

INSTALLED\_APPS = [

# Default Django apps ...

'myapp',

]

**Step 7: Run the Development Server to Check Setup**

python manage.py runserver

* Visit http://127.0.0.1:8000/ in your browser to see the default Django welcome page.

**Step 8: Create Views and Map URLs (Basic Example)**

**In myapp/views.py:**

from django.http import HttpResponse

def home(request):

return HttpResponse("Hello, Django app!")

**In myapp/urls.py (create this file):**

from django.urls import path

from . import views

urlpatterns = [

path('', views.home),

]

**In main project myproject/urls.py:**

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('myapp.urls')),

]

Q 13. Understanding the role of manage.py, urls.py, and views.py

Ans.

**1. manage.py**

* **What it is:**  
  A command-line utility script automatically created when you start a Django project.
* **Role:**  
  It acts as a **wrapper around Django's administrative commands**. You use it to run your project, create apps, apply database migrations, run tests, and much more.
* **Common commands:**
* python manage.py runserver # Start the development server
* python manage.py startapp myapp # Create a new app
* python manage.py migrate # Apply database migrations
* python manage.py createsuperuser # Create admin user
* python manage.py test # Run tests
* **Where is it?**  
  At the **root of your project directory**, alongside your main project folder.

**2. urls.py**

* **What it is:**  
  A Python module responsible for **mapping URLs (web addresses) to views**.
* **Role:**  
  It works like a **traffic controller** — when a user visits a URL, urls.py tells Django **which view to run** for that URL.
* **Two levels:**
  + **Project-level urls.py** (in your main project folder) routes top-level URLs and often includes app URLs.
  + **App-level urls.py** (optional, created inside each app) manages URLs specific to that app.
* **Example:**
* # project-level urls.py
* from django.urls import path, include
* urlpatterns = [
* path('admin/', admin.site.urls),
* path('blog/', include('blog.urls')), # Include URLs from blog app
* ]

**3. views.py**

* **What it is:**  
  A Python module inside each app that **contains functions or classes called views**.
* **Role:**  
  Views **process user requests**, perform logic (like fetching data from the database), and **return responses** (usually HTML pages or JSON data).
* **Example:**
* from django.http import HttpResponse
* def home(request):
* return HttpResponse("Hello, world!")
* Views receive a request object, process it, and return a response object.

Q 14.Django’s MVT (Model-View-Template) architecture and how it handles request-response cycles.

Ans.

**Django’s MVT Architecture Explained**

Django follows the **MVT (Model-View-Template)** design pattern, which is similar to MVC (Model-View-Controller) but slightly different in naming and roles:

| **Component** | **Role** | **Equivalent in MVC** |
| --- | --- | --- |
| **Model** | Manages data and database logic | Model |
| **View** | Handles business logic and processes requests | Controller |
| **Template** | Defines the presentation layer (HTML, UI) | View |

**1. Model**

* Represents the **data structure** of your application.
* Defines database schema through Python classes.
* Responsible for **querying, inserting, updating, and deleting data**.
* Located in models.py inside each app.

**2. View**

* Acts as the **business logic layer**.
* Receives HTTP requests, interacts with models to get data.
* Passes data to templates.
* Returns an HTTP response.
* Defined as functions or classes in views.py.

**3. Template**

* The **presentation layer** that renders HTML pages.
* Uses Django Template Language (DTL) to dynamically display data.
* Stored in templates/ directories.

**How Django Handles the Request-Response Cycle**

1. **User sends a request** by entering a URL in the browser.
2. **URL dispatcher (urls.py) receives the request** and matches the URL pattern to a view function.
3. **View function is called** with the request object.
4. **View processes the request:**
   * Interacts with the **Model** to fetch or modify data.
   * Loads and prepares data for display.
5. **View calls a Template** with context data to render HTML.
6. **Template generates an HTML response** that is sent back to the user’s browser.

**Flow Diagram**

User Request (URL)

↓

URL Dispatcher (urls.py)

↓

View (views.py)

↓

Model (models.py) ←→ Database

↓

Template (HTML + DTL)

↓

HTTP Response (HTML) sent back to user

**Example**

# urls.py

from django.urls import path

from . import views

urlpatterns = [

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

]

# views.py

from django.shortcuts import render

from .models import Article

def home(request):

articles = Article.objects.all() # Fetch data from Model

return render(request, 'home.html', {'articles': articles})

# home.html (template)

{% for article in articles %}

<h2>{{ article.title }}</h2>

<p>{{ article.content }}</p>

{% endfor %}