1. Python version check: python3 –version
2. Django version check: python3 -m django –version or Django.VERSION
3. Pip3 install to install libraries

Introduction to Django

Django is a high-level Python web framework that enables the rapid development of secure and maintainable websites. It emphasizes reusability and "don't repeat yourself" (DRY) principles, and it includes built-in features such as authentication, an admin interface, and ORM (Object-Relational Mapping). Django makes it easier to build complex web applications by handling common tasks such as routing, handling requests, and rendering templates.

MVT Pattern in Django

Django follows the **Model-View-Template (MVT)** architectural pattern, which is a variant of the Model-View-Controller (MVC) pattern, commonly used in web frameworks. In MVT:

- **Model**: Represents the data and business logic. This is where the database models are defined, and Django's ORM allows you to interact with the database.
- **View**: The view in Django takes care of processing user requests. It acts as the controller in the classic MVC pattern, executing business logic and interacting with models as needed. A view function takes a web request and returns a web response, typically by rendering a template or returning some data.
- **Template**: The template is responsible for rendering the HTML page or any other format (like JSON or XML) that the user interacts with. Templates contain static HTML code mixed with dynamic elements provided by the view.

Unlike MVC, where the **View** is focused on rendering data, Django separates this responsibility, so the rendering is primarily handled by the **Template**, while the **View** in Django focuses on request handling and data logic.

django-admin startproject myproject

The `django-admin startproject myproject` command is used to create a new Django project. Here's what happens:

- **`django-admin`**: This is a Django command-line utility to perform administrative tasks. It is responsible for creating projects, starting applications, and running various Django commands.
- **`startproject myproject`**: This creates a new directory named `myproject` with the following files:

    - `manage.py`: A command-line utility to interact with your project (runserver, migrations, etc.).
    - A folder named `myproject/` (or whatever project name you choose) which contains the core configuration files:
        - `__init__.py`: Marks the directory as a Python package.
        - `settings.py`: Contains the settings for the project (database configurations, installed apps, middleware, etc.).
        - `urls.py`: Defines the URL routing for the project.
        - `asgi.py` and `wsgi.py`: Entry points for the ASGI and WSGI server protocols respectively.

This command essentially sets up the skeleton for your Django project.

python3 manage.py startapp myapp

The command `python3 manage.py startapp myapp` creates a new Django app. In Django, the concept of an app is a self-contained module that performs specific functionality. Apps can be reused across multiple projects, and a single Django project can have multiple apps. For example, you might have an app to handle user authentication, another for managing blog posts, and another for handling comments.

When you run `startapp`, Django creates a folder structure with the following files:

- `admin.py`: Used to define how the app's models will appear in the Django admin interface.
- `apps.py`: Contains the configuration for the app.
- `models.py`: Used to define the data models (database tables) for the app.
- `views.py`: Used to define the logic that handles web requests and returns responses.
- `migrations/`: A folder to track database migrations for this app.
- `tests.py`: A module for writing unit tests for the app.
- `urls.py`: Not created by default but typically added later to define the app's URL routes.

Multiple Apps in a Django Project

Django supports having multiple apps within a single project. For example, after creating a project (`myproject`), you can create multiple apps, like `myapp` and `myapp2`. Each app will handle different aspects of the project. For example, `myapp`could handle user management, while `myapp2` could handle blog posts.

- Each app can have its own `models.py`, `views.py`, and other files, making it modular and easy to maintain. Django encourages this modular approach to web development.

python3 manage.py runserver

The `python3 manage.py runserver` command is used to start Django's built-in development server. It starts a lightweight web server that allows you to run your application locally without the need for any external server configuration. Here's what happens:

- The server listens for HTTP requests on a specified port (usually `localhost:8000` by default, but you can specify a different port if needed).
- When a request is made to the server (e.g., by visiting `http://localhost:8000/` in the browser), Django routes the request to the appropriate view based on the URL configuration in `urls.py`.
- The view processes the request, possibly interacts with the database via the model, and returns a response (usually by rendering a template).

Installing Virtualenv and Why We Need It

`virtualenv` is a tool used to create isolated Python environments. A virtual environment is essentially a self-contained directory that contains all the necessary dependencies, packages, and Python binary files needed to run your project. This means:

- You can install specific versions of libraries and packages for a particular project without affecting the global Python environment on your system or other projects.
- It prevents "dependency conflicts" between different projects. For example, one project may require Django 3.x, while another might require Django 4.x. With virtualenv, you can have separate environments with the appropriate versions installed for each project.

The Command: `virtualenv myenv`

This command creates a virtual environment named `myenv`. Here's how it works:

1. **`virtualenv`**: This is the command to create a virtual environment.
2. **`myenv`**: This is the name of the virtual environment. You can choose any name you like.

When you run this command, `virtualenv` creates a directory named `myenv` which contains:

- A copy of the Python interpreter.
- A `bin` (or `Scripts` on Windows) folder containing executables.
- A `lib` folder containing libraries that the virtual environment will use.

Activating the Virtual Environment

Once you have created a virtual environment, you need to activate it. The steps to activate the environment vary depending on your operating system.

On Linux/macOS:
```
bash
Copy code
source myenv/bin/activate
```
On Windows:
```
Copy code
myenv\Scripts\activate
```

After activation, you will notice that your command prompt changes to indicate that the environment is active, for example:

```
ruby
Copy code
(myenv) $
```

Why Do We Activate the Virtual Environment?

Activating the virtual environment ensures that any Python commands you execute (such as `pip install` or `python`commands) are confined to that specific environment. When a virtual environment is active:

- Python will use the environment's interpreter instead of the global one.
- Libraries and packages installed with `pip` will be installed in the virtual environment, not globally.

This allows you to isolate dependencies and avoid conflicts between projects.

What to Do Next After Activating the Virtual Environment?

After activation, you can begin installing the necessary Python packages for your project using `pip` (Python's package manager). For example:

```
Copy code
pip install django
```

This command installs Django only within the active virtual environment, keeping it isolated from the global Python installation.

Command: `pip3 freeze > requirements.txt`

This command is used to create a list of all the Python packages (along with their versions) currently installed in your virtual environment. Here's what happens:

- **`pip3 freeze`**: Lists all installed packages in the current environment.

- **`>`**: Redirects the output of the command.
- **`requirements.txt`**: The file where the list of installed packages is saved.

The purpose of this file is to capture the exact versions of all the dependencies used in your project. This is useful when sharing the project with others or deploying it to another environment (e.g., production). You can then install all the required packages using this file.

Command: `pip3 install -r requirements.txt`

This command is used to install all the packages listed in the `requirements.txt` file. Here's what it does:

- **`pip3 install`**: Installs Python packages.
- **`-r requirements.txt`**: This tells `pip` to install the packages from the `requirements.txt` file. The file contains the names of the packages and their respective versions, and `pip` installs them exactly as specified.

This is particularly useful when:

- You're setting up a project on a new machine or environment and want to install all the necessary dependencies without manually installing each package.
- You want to ensure that all collaborators or deployment environments have the same dependencies and versions, ensuring consistency.

Example Workflow with Virtualenv

1. **Create the virtual environment**:

   ```
   Copy code
   virtualenv myenv
   ```

2. **Activate the virtual environment**:

   ```
   bash
   Copy code
   source myenv/bin/activate  # (Linux/macOS)
   myenv\Scripts\activate     # (Windows)
   ```

3. **Install packages** (e.g., Django):

   ```
   Copy code
   pip install django
   ```

4. **Save installed packages**:

   ```
   Copy code
   pip freeze > requirements.txt
   ```

5. **Share the `requirements.txt` file** with others or for deployment.
6. **Install packages from the `requirements.txt`** in a new environment:

   ```
   Copy code
   pip install -r requirements.txt
   ```

By using virtual environments, you ensure each project has the right dependencies isolated, making development and deployment easier to manage.

URL Dispatcher, Requests & Responses in Django

Django's URL dispatcher is responsible for mapping URLs to views. Each URL is associated with a specific function (view) that will process the request and return a response.

- **URLconf**: It stands for URL configuration and is used to define the routing of URLs to views. It maps a URL pattern to a view function. In Django, this is typically done in the `urls.py` file.
- **Requests**: When a user makes a request to the Django server (such as visiting a URL in their browser), Django processes that request and looks for the appropriate view function based on the URL pattern.
- **Responses**: Once the view function processes the request, it returns a response, typically in the form of an `HttpResponse` or `JsonResponse`.

Step-by-Step Explanation

1. Modifying `settings.py` with the app's name

Once you have created an app (in this case, `myfirstapp`), you need to add it to your Django project's `INSTALLED_APPS`setting. This lets Django know about the app and includes it in the project.

Edit `settings.py` in your project and add `myfirstapp`:

```
python
Copy code
# Inside settings.py

INSTALLED_APPS = [
    # Default Django apps
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Your custom app
    'myfirstapp',  # Add your app here
]
```

By adding `myfirstapp` to `INSTALLED_APPS`, Django can recognize and manage the app during runtime, migrations, and other operations.

2. Setting up URL Routing for the Project (`myfirstproject/urls.py`)

Django uses a `urls.py` file at both the project level and app level to manage routing. The project's `urls.py` defines high-level routes and includes the URL patterns from individual apps.

Here's the content of your `myfirstproject/urls.py` file:

```
python
Copy code
# Inside myfirstproject/urls.py

from django.contrib import admin
from django.urls import path, include  # Import include for including app-
specific URLs

urlpatterns = [
    path('admin/', admin.site.urls),   # Route for the Django admin interface
    path('', include('myfirstapp.urls')),  # Include the URLs from myfirstapp
]
```

This line:

```
python
Copy code
path('', include('myfirstapp.urls')),
```

means that when users visit the root URL (i.e., `/`), Django will use the URL patterns defined in the `myfirstapp`'s `urls.py`file.

3. Creating URL Routing for the App (`myfirstapp/urls.py`)

In `myfirstapp`, you define its own `urls.py` file that specifies the routing for the app. For each path, you define the view function that will handle the request and generate a response.

Here's the content of `myfirstapp/urls.py`:

```
python
```

```
Copy code
# Inside myfirstapp/urls.py

from django.urls import path
from . import views  # Import views from the current app directory

urlpatterns = [
    path('', views.myfunctioncall, name='index'),  # Route for the home page
    path('about', views.myfunctionabout, name='about'),  # Route for the
about page
    path('add/<int:a>/<int:b>', views.add, name='add'),  # Route for
addition, with integer parameters
    path('intro/<str:name>/<int:age>', views.intro, name='intro'),  # Route
for introduction, with string and integer parameters
]
```

- **`path('', views.myfunctioncall, name='index')`**: The empty string `''` signifies the root URL of the app (i.e., `/`). When users visit the app's root, Django calls the `myfunctioncall` function in `views.py`.
- **`path('about', views.myfunctionabout, name='about')`**: This maps `/about` to the `myfunctionabout` view, returning a response when users visit `/about`.
- **`path('add/<int:a>/<int:b>', views.add, name='add')`**: This URL takes two integers (`a` and `b`) as part of the path, like `/add/3/5`, and passes them to the `add` view function.
- **`path('intro/<str:name>/<int:age>', views.intro, name='intro')`**: This URL expects a string `name` and an integer `age`, like `/intro/John/25`, and passes them to the `intro` view function.

4. Views in Django (`myfirstapp/views.py`)

The views in Django handle the request and generate a response based on the URL routing. Here's what each view does:

```
python
Copy code
# Inside myfirstapp/views.py

from django.shortcuts import render
from django.http import HttpResponse, JsonResponse

# Function that returns a simple "Hello, World!" response
def myfunctioncall(request):
    return HttpResponse("Hello, World!")

# Function that returns an "About" page response
def myfunctionabout(request):
    return HttpResponse("About Response")

# Function that adds two numbers passed in the URL
def add(request, a, b):
    return HttpResponse(f"Sum of {a} and {b} is {a+b}")
```

```
# Function that returns a JSON response with name and age
def intro(request, name, age):
    mydictionary = {
        "name": name,
        "age": age
    }
    return JsonResponse(mydictionary)
```

- **`myfunctioncall(request)`**: This is called when users visit the home page (`/`). It returns a simple HTTP response with the text "Hello, World!".
- **`myfunctionabout(request)`**: This function handles requests to `/about` and returns "About Response".
- **`add(request, a, b)`**: This function expects two integer parameters (`a` and `b`). It returns the sum of those two numbers in the response. Example: visiting `/add/3/5` will return `Sum of 3 and 5 is 8`.
- **`intro(request, name, age)`**: This function expects a string `name` and an integer `age`. It returns a JSON response with these values. Example: visiting `/intro/John/25` will return `{"name": "John", "age": 25}` as JSON.

Summary of How It All Works Together

1. **Request**: When a user visits a URL, Django's URL dispatcher (defined in `urls.py`) looks for a matching URL pattern.
2. **Routing**: Once Django finds a match in the URL patterns (`urlpatterns`), it calls the corresponding view function.
3. **View Function**: The view function processes the request, performs any logic (e.g., calculating the sum or creating a dictionary), and returns a response (either HTML or JSON).
4. **Response**: The response is sent back to the browser, which could be a simple string (`HttpResponse`) or structured data like JSON (`JsonResponse`).

What is Bootstrap?

Bootstrap is a popular open-source front-end framework used for designing responsive and mobile-first websites. It includes pre-built CSS styles, JavaScript components, and HTML templates that make it easier to create a clean and modern user interface. Bootstrap provides components like a grid system, buttons, forms, navigation bars, and modals, helping developers speed up the design process while ensuring the site is responsive across different screen sizes.

Templates in Django

In Django, **templates** are used to define the structure of the HTML that is rendered to the user. Templates are plain HTML files that can contain **Django Template Language (DTL)**, which allows you to insert dynamic content, iterate over data, and extend base templates.

Django looks for templates in a designated folder named `templates`. You can structure your templates into subdirectories for different apps, making them easier to manage.

Template Folder

The **template folder** is where you store all your HTML templates. Django looks for this folder when rendering views that require HTML. By default, you have to create this folder and configure Django to recognize it in the `settings.py` file.

For example, in your project structure, you might have:

```
Copy code
myproject/
├── myfirstapp/
│   ├── templates/
│   │   └── myfirstapp/
│   │       └── index.html
├── manage.py
```

Setting Up Templates in `settings.py`

To tell Django where to find your templates, you need to add the template directory path to the `TEMPLATES` setting in `settings.py`:

```
python
Copy code
# Inside settings.py

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],  # Specify your template directory
here
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # other context processors
            ],
        },
    },
]
```

Here, `DIRS` is a list of directories where Django will look for templates. You can add multiple directories if needed. The line `APP_DIRS: True` allows Django to automatically search for a `templates` folder inside each app.

Static Folder

The **static folder** is used to store static files such as CSS, JavaScript, images, and fonts that don’t change dynamically. In Django, static files are usually placed inside a folder named `static` within your app directory.

Setting Up Static Files in `settings.py`

To serve static files correctly, you need to define the location of your static files in `settings.py`. Here's how you do it:

```
python
Copy code
# Inside settings.py

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

- **`STATIC_URL`**: Defines the URL path for accessing static files in the browser (e.g., `/static/`).
- **`STATICFILES_DIRS`**: Specifies directories where Django should look for static files.

Now, inside your project structure, you might have something like this:

```
arduino
Copy code
myproject/
├── myfirstapp/
│   ├── static/
│   │   └── myfirstapp/
│   │       └── style.css
│   │
```

In your HTML templates, you can reference static files using Django's `{% static %}` tag:

```
html
Copy code
<link rel="stylesheet" href="{% static 'myfirstapp/style.css' %}">
```

Pages and Extending Templates with `{% extends %}` and `{% block %}`

Django allows you to reuse common sections of HTML (like headers, footers, and navigation bars) by using template inheritance with the `{% extends %}` and `{% block %}` tags.

Example 1: Base Template (`index.html`)

You can define a base template that contains common elements like the navigation bar or footer that you want to reuse on every page.

```
html
Copy code
<!-- Inside templates/myfirstapp/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <title>{% block title %}My Website{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'myfirstapp/style.css' %}">
</head>
<body>
    <nav>
        <!-- Common navigation bar -->
    </nav>

    <div>
        {% block content %}
        <!-- Placeholder for page-specific content -->
        {% endblock %}
    </div>

    <footer>
        <!-- Common footer -->
    </footer>
</body>
</html>
```

Example 2: Extending the Base Template (`second.html`)

Now, to create a second page, you can extend the `index.html` template and insert page-specific content using the `{% block %}` tag.

```
html
Copy code
<!-- Inside templates/myfirstapp/second.html -->
{% extends 'myfirstapp/index.html' %}

{% block title %}
My Second Page
{% endblock %}

{% block content %}
<h1>Second Page extends first page with a common navbar</h1>
<p>This page inherits the common layout from the base template.</p>
{% endblock %}
```

Explanation of Template Tags

- **`{% extends 'index.html' %}`**: This tag indicates that the current template (e.g., `second.html`) extends another template (in this case, `index.html`). The `index.html` template serves as the base template.
- **`{% block title %}`**: Defines a block where you can insert a title that will appear in the `<title>` tag. The default title in the base template can be overridden here (e.g., `My Second Page`).
- **`{% block content %}`**: This block allows you to insert page-specific content that will be placed in the `content`section of the base template. In the example, the second page includes a heading and a paragraph inside the content block.

Using these template tags, you can easily create multiple pages that share a consistent layout (like navigation bars or footers), making your project more maintainable and reducing duplicate code.

Forms with Django

Django has a powerful form handling mechanism. Forms in Django allow you to easily manage and validate user input. Forms can be created directly in your HTML templates or through Django's `forms.py` to manage complex inputs.

Creating Forms in Django

1. **Creating a Form in `forms.py`**: You typically define your forms in the `forms.py` file within your app.

   ```
   python
   Copy code
   # Inside myfirstapp/forms.py
   from django import forms

   class ContactForm(forms.Form):
       name = forms.CharField(label='Your Name', max_length=100)
       email = forms.EmailField(label='Your Email')
       message = forms.CharField(widget=forms.Textarea, label='Your
   Message')
   ```

2. **Using the Form in a View**: Once the form is created, you can use it in a view.

   ```
   python
   Copy code
   # Inside myfirstapp/views.py
   from django.shortcuts import render
   from .forms import ContactForm

   def contact_view(request):
       if request.method == 'POST':
           form = ContactForm(request.POST)
           if form.is_valid():
               # Process form data
               name = form.cleaned_data['name']
               email = form.cleaned_data['email']
               message = form.cleaned_data['message']
               return HttpResponse(f'Thank you, {name}!')
       else:
           form = ContactForm()

       return render(request, 'myfirstapp/contact.html', {'form': form})
   ```

3. **Rendering the Form in a Template**: In your HTML template, you can render the form with Django's `{{ form.as_p }}` method or manually render each form field.

```
html
Copy code
<!-- Inside templates/myfirstapp/contact.html -->
<h1>Contact Us</h1>
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Send</button>
</form>
```

- **`{% csrf_token %}`**: Adds a CSRF token for security, preventing cross-site request forgery attacks.
- **`{{ form.as_p }}`**: Renders the form fields automatically, wrapping each field in a `<p>` tag.

Handling Form Submissions

Django forms handle both GET and POST requests. After form submission, Django validates the form data. If the data is valid, you can access the cleaned form data through `form.cleaned_data`.

By combining Django forms with templates and views, you can build fully functional user interfaces for collecting and processing data, such as login forms, contact forms, or registration forms.

Conclusion

- **Bootstrap** helps style your templates.
- **Templates** provide the structure for rendering dynamic content.
- **Static folder** stores CSS/JS files, and **settings.py** is configured to use them.
- **Template inheritance** with `{% extends %}` and `{% block %}` enables code reuse across pages.
- **Forms in Django** provide a way to handle user input efficiently, with built-in validation and security.