Django Basics - Getting Started

1. Creating a Python Virtual Environment

What is a Virtual Environment?

A virtual environment is an isolated Python environment that allows you to install packages and dependencies specific to a project without affecting your system-wide Python installation.

Why Create a Virtual Environment?

- **Dependency Isolation**: Different projects may require different versions of the same package. Virtual environments prevent version conflicts.
- **Clean Project Structure**: Keeps your project dependencies organized and separate from other projects.
- **Reproducibility**: Makes it easier to share your project with others by maintaining a consistent environment.
- **System Protection**: Prevents accidental modifications to system-wide Python packages.
- **Easy Cleanup**: You can delete a virtual environment without affecting other projects or your system.

Creating a Virtual Environment

Using venv (Python 3.3+):

```
# Create virtual environment
python -m venv myenv

# Activate on Windows
myenv\Scripts\activate

# Activate on macOS/Linux
source myenv/bin/activate

# Deactivate
deactivate
```

Using virtualenv:

```
# Install virtualenv
pip install virtualenv

# Create virtual environment
virtualenv myenv

# Activation is same as above
```

---

2. Installing Django and Running the Server

Installing Django

After activating your virtual environment:

```
# Install latest version
pip install django

# Install specific version
pip install django==4.2

# Verify installation
python -m django --version
```

Creating a Django Project

```
# Create new project
django-admin startproject myproject

# Navigate to project directory
cd myproject
```

Running the Development Server

```
# Start server (default: http://127.0.0.1:8000)
python manage.py runserver

# Run on specific port
python manage.py runserver 8080

# Run on specific IP and port
python manage.py runserver 0.0.0.0:8000
```

Creating a Django App

```
# Create an app within your project
python manage.py startapp myapp
```

Note: Remember to add your app to `INSTALLED_APPS` in `settings.py`:

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

---

3. Features of Django

Why Use Django?

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the “batteries-included” philosophy.

Key Features

MVT Architecture (Model-View-Template) - **Model**: Defines data structure and database schema - **View**: Contains business logic and handles requests - **Template**: Handles presentation layer (HTML)

Built-in Features

- **ORM (Object-Relational Mapping)**: Work with databases using Python code instead of SQL
- **Admin Interface**: Automatically generated admin panel for managing data
- **Authentication System**: Built-in user authentication and permissions
- **URL Routing**: Clean and elegant URL design
- **Template Engine**: Powerful templating system for dynamic HTML
- **Form Handling**: Easy form creation, validation, and processing
- **Security**: Protection against SQL injection, XSS, CSRF, and clickjacking
- **Internationalization**: Support for multiple languages
- **Caching**: Built-in caching framework for performance optimization
- **Testing Framework**: Comprehensive testing tools

Additional Advantages

- **Scalability**: Used by high-traffic sites like Instagram, Pinterest, and Mozilla
- **DRY Principle**: Don’t Repeat Yourself - promotes code reusability
- **Excellent Documentation**: Comprehensive and well-maintained docs
- **Large Community**: Active community with extensive third-party packages
- **REST Framework**: Django REST framework for building APIs
- **Versatile**: Suitable for various applications from simple sites to complex platforms

---

4. Django Project Structure

When you create a Django project, the following structure is generated:

```
myproject/
├── manage.py
└── myproject/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

When you create an app:

```
myapp/
├── __init__.py
├── admin.py
├── apps.py
├── migrations/
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Project vs App

- **Project**: The entire Django project containing settings and configuration
- **App**: A web application that does something (e.g., blog, forum, poll)
- A project can contain multiple apps
- An app can be used in multiple projects

Typical App Structure Breakdown

- **migrations/**: Database migration files
- **admin.py**: Register models for admin interface
- **apps.py**: App configuration
- **models.py**: Define database models
- **tests.py**: Write tests for your app
- **views.py**: Define view functions/classes

---

5. Important Django Files Explained

manage.py

A command-line utility for interacting with your Django project.

Purpose: Wrapper around `django-admin` that sets up the project environment

Common Commands:

```
python manage.py runserver      # Start development server
python manage.py makemigrations # Create database migrations
python manage.py migrate        # Apply migrations to
        database
python manage.py createsuperuser # Create admin user
python manage.py shell          # Interactive Python shell
python manage.py test           # Run tests
python manage.py collectstatic  # Collect static files
```

Note: You should never need to edit this file.

---

settings.py

The configuration file for your Django project containing all settings and configuration.

Key Settings:

```
# Secret key for cryptographic signing
SECRET_KEY = 'your-secret-key'

# Debug mode (set to False in production)
DEBUG = True

# Allowed hosts for the application
ALLOWED_HOSTS = []

# Installed applications
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    # ... your apps
]

# Middleware components
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    # ...
]

# Database configuration
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}

# Static files configuration
STATIC_URL = 'static/'
```

```
# Template configuration
TEMPLATES = [...]

# Internationalization
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
```

---

urls.py

URL configuration file that maps URLs to views.

Purpose: Define URL patterns for your application (routing)

Example:

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

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

Key Concepts: - Each URL pattern can have a name for reverse URL lookup - Use `include()` to reference other URL configurations - Supports path converters: `<int:id>`, `<str:username>`, `<slug:slug>`

---

wsgi.py

Web Server Gateway Interface file for deploying Django with WSGI-compatible servers.

Purpose: Entry point for WSGI-compatible web servers to serve your project

When Used: Production deployment with servers like Gunicorn, uWSGI, or Apache with mod_wsgi

Example:

```
import os
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
        'myproject.settings')
application = get_wsgi_application()
```

WSGI Servers: - Gunicorn - uWSGI - Apache mod_wsgi - Waitress

---

asgi.py

Asynchronous Server Gateway Interface file for deploying Django with ASGI-compatible servers.

Purpose: Entry point for ASGI-compatible servers, supports async views and WebSockets

When Used: - Handling WebSockets - Asynchronous views - Long-polling connections - Modern async deployment

Example:

```
import os
from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
        'myproject.settings')
application = get_asgi_application()
```

ASGI Servers: - Daphne - Uvicorn - Hypercorn

Difference from WSGI: ASGI is the async successor to WSGI, supporting both synchronous and asynchronous applications.

---

6. Django Superuser and Admin Panel

What is a Superuser?

A superuser is a special user account in Django with all permissions enabled. This account has complete control over the Django admin interface and can manage all aspects of your application.

Superuser Privileges: - Full access to Django admin panel - Can create, edit, and delete any data - Can manage other users and their permissions - Has all permissions without explicitly assigning them

Creating a Superuser

```
# Create superuser
python manage.py createsuperuser

# You'll be prompted to enter:
# - Username
# - Email address
# - Password (entered twice for confirmation)
```

Example:

```
Username: admin
Email address: admin@example.com
```

```
Password: ********
Password (again): ********
Superuser created successfully.
```

Creating Superuser Programmatically:

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

User.objects.create_superuser(
    username='admin',
    email='admin@example.com',
    password='securepassword123'
)
```

Django Admin Panel

The Django admin panel is a powerful, automatically-generated interface for managing your application's data.

Key Features: - Automatically generated based on your models - CRUD operations (Create, Read, Update, Delete) - Search, filter, and sorting capabilities - User and permission management - Customizable interface

Accessing the Admin Panel:

1. Ensure admin is in INSTALLED_APPS (it is by default)
2. Run migrations: `python manage.py migrate`
3. Create a superuser (see above)
4. Start server: `python manage.py runserver`
5. Navigate to: `http://127.0.0.1:8000/admin`
6. Login with superuser credentials

---

7. Understanding Migrations and Django ORM

What are Migrations?

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're essentially version control for your database.

Why Migrations? - Track database schema changes over time - Synchronize models with database structure - Enable collaboration (team members can apply same changes) - Allow rollback to previous database states - Work across different database backends

Migration Workflow: 1. Create/modify models in `models.py` 2. Run `python manage.py makemigrations` (creates migration files) 3. Run `python manage.py migrate` (applies migrations to database)

Django ORM (Object-Relational Mapping)

The ORM allows you to interact with your database using Python code instead of writing SQL queries. Each model class represents a database table.

How ORM Works Behind the Scenes

Django ORM converts Python code into SQL queries automatically. You write Python, Django executes SQL.

Example: Python ORM vs SQL

```
# Python ORM Code
Article.objects.filter(status='published')

# Django converts this to SQL:
# SELECT * FROM blog_article WHERE status = 'published';

# Python ORM Code
Article.objects.create(title='My First Post',
        content='Hello World')

# Django converts this to SQL:
# INSERT INTO blog_article (title, content) VALUES ('My
        First Post', 'Hello World');
```

This is the power of ORM - you never have to write SQL manually!

Migration Commands

```
# Create migration files
python manage.py makemigrations

# View migration SQL without applying
python manage.py sqlmigrate app_name 0001

# Apply migrations to database
python manage.py migrate

# Show all migrations and their status
python manage.py showmigrations

# Rollback to specific migration
python manage.py migrate app_name 0001
```

Quick Start Checklist

1. ✅ Create virtual environment
2. ✅ Activate virtual environment
3. ✅ Install Django
4. ✅ Create Django project
5. ✅ Create Django app
6. ✅ Add app to INSTALLED_APPS
7. ✅ Run migrations
8. ✅ Create superuser
9. ✅ Start development server

```
# Complete workflow
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate on
        Windows
pip install django
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
# Edit settings.py to add 'myapp' to INSTALLED_APPS
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

Access your application: - Site: http://127.0.0.1:8000 - Admin: http://127.0.0.1:8000/admin