Teaching Learning Plan

Course Code:	**21CS62**	CIE Marks	50
Teaching Hours/Week	3:0:2:0	SEE Marks	50
Total No. of Hours	40 T + 20 P	Total Marks	100
Credits	04	Exam Hours	03

Course Objectives: The student should be made to:

- CLO 1. Explain the use of learning full stack web evelopment.
- CLO 2. Make use of rapid application development in the design of responsive web pages.
- CLO 3. Illustrate Models, Views and Templates with their connectivity in Django for full stack web development.
- CLO 4. Demonstrate the use of state management and admin interfaces automation in Django.
- CLO 5. Design and implement Django apps containing dynamic pages with SQL databases.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) need not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes criticalthinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop designthinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. Introduce Topics in manifold representations.
7. Show the different ways to solve the same problem with different logic and encourage thestudents to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

Course Outcomes (Course Skill Set) At the end of the course the student will be able to:

- CO 1. Understand the working of MVT based full stack web development with Django.
- CO 2. Designing of Models and Forms for rapid development of web pages.
- CO 3. Analyze the role of Template Inheritance and Generic views for developing full stack web applications.
- CO 4. Apply the Django framework libraries to render nonHTML contents like CSV and PDF.
- CO 5. Perform jQuery based AJAX integration to Django Apps to build responsive full stack web applications,

Module-1: MVC based Web Designing

Web framework, MVC Design Pattern, Django Evolution, Views, Mapping URL to Views, Working of Django URL Confs and Loose Coupling, Errors in Django, Wild Card patterns in URLS.

Textbook 1: Chapter 1 and Chapter 3

Laboratory Component: 1. Installation of Python, Django and Visual Studio code editors can be demonstrated. 2. Creation of virtual environment, Django project and App should be demonstrated 3. Develop a Django app that displays current date and time in server 4. Develop a Django app that displays date and time four hours ahead and four hours before as an offset of current date and time in server.

Teaching-Learning Process

1. Demonstration using Visual Studio Code
2. PPT/Prezi Presentation for Architecture and Design Patterns
3. Live coding of all concepts with simple examples

Module-2: Django Templates and Models

Template System Basics, Using Django Template System, Basic Template Tags and Filters, MVT Development Pattern, Template Loading, Template Inheritance, MVT Development Pattern. Configuring Databases, Defining and Implementing Models, Basic Data Access, Adding Model String Representations, Inserting/Updating data, Selecting and deleting objects, Schema Evolution

Textbook 1: Chapter 4 and Chapter 5

Laboratory Component: 1. Develop a simple Django app that displays an unordered list of fruits and ordered list of selected students for an event 2. Develop a layout.html with a suitable header (containing navigation menu) and footer with copyright and developer information. Inherit this layout.html and create 3 additional pages: contact us, About Us and Home page of any website. 3. Develop a Django app that performs student registration to a course. It should also display list of students registered for any selected course. Create students and course as models with enrolment as ManyToMany field.

Teaching-Learning Process 1. Demonstration using Visual Studio Code 2. PPT/Prezi Presentation for Architecture and Design Patterns 3. Live coding of all concepts with simple examples 4. Case Study: Apply concepts learnt for an Online Ticket Booking System

Module-3: Django Admin Interfaces and Model Forms

Activating Admin Interfaces, Using Admin Interfaces, Customizing Admin Interfaces, Reasons to use Admin Interfaces. Form Processing, Creating Feedback forms, Form submissions, custom validation, creating Model Forms, URLConf Ticks, Including Other URLConfs.

Textbook 1: Chapters 6, 7 and 8

Laboratory Component:

1. For student and course models created in Lab experiment for Module2, register admin interfaces, perform migrations and illustrate data entry through admin forms.

2. Develop a Model form for student that contains his topic chosen for project, languages used and duration with a model called project.

Teaching-Learning Process 1. Demonstration using Visual Studio Code 2. PPT/Prezi Presentation for Architecture and Design Patterns 3. Live coding of all concepts with simple examples

Module-4: Generic Views and Django State Persistence

Using Generic Views, Generic Views of Objects, Extending Generic Views of objects, Extending Generic Views. MIME Types, Generating Non-HTML contents like CSV and PDF, Syndication Feed Framework, Sitemap framework, Cookies, Sessions, Users and Authentication.

Textbook 1: Chapters 9, 11 and 12

Laboratory Component:

1. For students enrolment developed in Module 2, create a generic class view which displays list of students and detailview that displays student details for any selected student in the list.

2. Develop example Django app that performs CSV and PDF generation for any models created in previous laboratory component.

Teaching-Learning Process

1. Demonstration using Visual Studio Code
2. PPT/Prezi Presentation for Architecture and Design Patterns
3. Live coding of all concepts with simple examples
4. Project Work: Implement all concepts learnt for Student Admission Management.

Module-5: jQuery and AJAX Integration in Django

Ajax Solution, Java Script, XHTMLHttpRequest and Response, HTML, CSS, JSON, iFrames, Settings of Java Script in Django, jQuery and Basic AJAX, jQuery AJAX Facilities, Using jQuery UI Autocomplete in Django

Textbook 2: Chapters 1, 2 and 7.

Laboratory Component:

1. Develop a registration page for student enrolment as done in Module 2 but without page refresh using AJAX.
2. Develop a search application in Django using AJAX that displays courses enrolled by a student being searched.

Teaching-Learning Process

1. Demonstration using Visual Studio Code

2. PPT/Prezi Presentation for Architecture and Design Patterns
3. Live coding of all concepts with simple examples
4. Case Study: Apply the use of AJAX and jQuery for development of EMI calculator.

Suggested Learning Resources:

Textbooks

1. Adrian Holovaty, Jacob Kaplan Moss, The Definitive Guide to Django: Web Development Done Right, Second Edition, Springer-Verlag Berlin and Heidelberg GmbH & Co. KG Publishers, 2009
2. Jonathan Hayward, Django Java Script Integration: AJAX and jQuery, First Edition, Pack Publishing, 2011

Reference Books

1. Aidas Bendroraitis, Jake Kronika, Django 3 Web Development Cookbook, Fourth Edition, Packt Publishing, 2020
2. William Vincent, Django for Beginners: Build websites with Python and Django, First Edition, Amazon Digital Services, 2018
3. Antonio Mele, Django3 by Example, 3rd Edition, Pack Publishers, 2020
4. Arun Ravindran, Django Design Patterns and Best Practices, 2nd Edition, Pack Publishers, 2020.
5. Julia Elman, Mark Lavin, Light weight Django, David A. Bell, 1st Edition, Oreily Publications, 2014

Weblinks and Video Lectures (e-Resources):

1. MVT architecture with Django: https://freevideolectures.com/course/3700/django-tutorials
2. Using Python in Django: https://www.youtube.com/watch?v=2BqoLiMT3Ao
3. Model Forms with Django: https://www.youtube.com/watch?v=gMM1rtTwKxE
4. Real time Interactions in Django: https://www.youtube.com/watch?v=3gHmfoeZ45k
5. AJAX with Django for beginners: https://www.youtube.com/watch?v=3VaKNyjlxAU

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning 1. Real world problem solving - applying the Django framework concepts and its integration with AJAX to develop any shopping website with admin and user dashboards.

Short Preamble on Full Stack Web Development:

Website development is a way to make people aware of the services and/or products they are offering, understand why the products are relevant and even necessary for them to buy or use, and highlight the striking qualities that set it apart from competitors. Other than commercial reasons, a website is also needed for quick and dynamic information delivery for any domain. Development of a well-designed, informative, responsive and dynamic website is need of the hour from any computer science and related engineering graduates. Hence, they need to be augmented with skills to use technology and framework which can help them to develop elegant websites. Full Stack developers are in need by many companies, who knows and can develop all pieces of web application (Front End, Back End and business logic). MVT based development with Django is the cutting-edge framework for Full Stack Web Development. Python has become an easier language to use for many applications. Django based framework in Python helps a web developer to utilize framework and develop rapidly responsive and secure web applications.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation:

Three Unit Tests each of **20 Marks (duration 01 hour**)

1. First test at the end of 5th week of the semester
2. Second test at the end of the 10th week of the semester
3. Third test at the end of the 15th week of the semester

Two assignments each of **10 Marks**

4. First assignment at the end of 4th week of the semester
5. Second assignment at the end of 9th week of the semester

Practical Sessions need to be assessed by appropriate rubrics and viva-voce method. This will contribute to **20 marks**.

- Rubrics for each Experiment taken average for all Lab components – 15 Marks.
- Viva-Voce– 5 Marks (more emphasized on demonstration topics)

The sum of three tests, two assignments, and practical sessions will be out of 100 marks and will be **scaled down to 50 marks**
(to have a less stressed CIE, the portion of the syllabus should not be common /repeated for any of the methods of the CIE. Each method of CIE should have a different syllabus portion of the course).

CIE methods /question paper has to be designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the subject (**duration 03 hours)**

1. The question paper will have ten questions. Each question is set for 20 marks. Marks scored shall be proportionally reduced to 50 marks
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.

The students have to answer 5 full questions, selecting one full question from each module

Module-1: MVC based Web Designing

Web Frameworks:

A web framework is a software framework designed to aid in the development of web applications by providing reusable code, components, and tools that streamline the process. Web frameworks typically follow the model-view-controller (MVC) architecture or a similar pattern, which helps in organizing code and separating concerns.

Examples:

1. Django (Python): Django is a high-level web framework for Python that follows the MVC pattern. It provides a robust set of features for building web applications, including an ORM, URL routing, form handling, session management, and a built-in admin interface.
2. Flask (Python): Flask is a lightweight web framework for Python that is designed to be simple and easy to use. It provides essential features for building web applications, such as URL routing, template rendering, and support for extensions.
3. Ruby on Rails (Ruby): Ruby on Rails is a popular web framework for Ruby that follows the MVC pattern. It emphasizes convention over configuration and includes features like ActiveRecord (ORM), URL routing, form helpers, and built-in support for testing.
4. Express.js (JavaScript/Node.js): Express.js is a minimal and flexible web framework for Node.js that is used for building web applications and APIs. It provides a simple yet powerful API for defining routes, middleware, and handling HTTP requests and responses.
5. ASP.NET Core (C#): ASP.NET Core is a cross-platform web framework for building modern, cloud-based web applications using C#. It includes features like MVC pattern support, middleware pipeline, dependency injection, and built-in support for authentication and authorization.

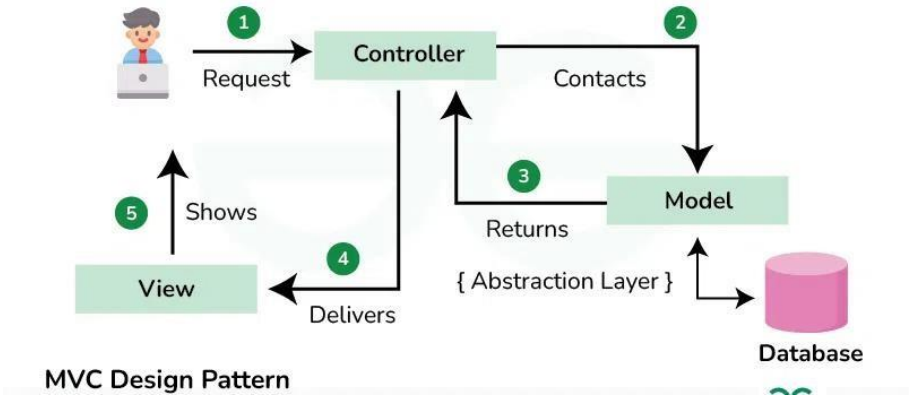
MVC Design Pattern

The Model View Controller (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

- The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application's functionality.

- This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to the other components.

Components of the MVC Design Pattern



1. Model

The Model component in the MVC (Model-View-Controller) design pattern represents the data and business logic of an application. It is responsible for managing the application's data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.

2. View

Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.

3. Controller

Controller acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

Communication between the components

This below communication flow ensures that each component is responsible for a specific aspect of the application's functionality, leading to a more maintainable and scalable architecture

User Interaction with View:

- The user interacts with the View, such as clicking a button or entering text into a form.

View Receives User Input:

- The View receives the user input and forwards it to the Controller.

Controller Processes User Input:

- The Controller receives the user input from the View.
- It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.

Controller Updates Model:

- The Controller updates the Model based on the user input or application logic.

Model Notifies View of Changes:

- If the Model changes, it notifies the View.

View Requests Data from Model:

- The View requests data from the Model to update its display.

Controller Updates View:

- The Controller updates the View based on the changes in the Model or in response to user input.

View Renders Updated UI:

- The View renders the updated UI based on the changes made by the Controller.

Django Evolution

The evolution of Django, a popular web framework for building web applications in Python, has been marked by significant milestones and improvements over the years. Here's a brief overview of its evolution:

1. Initial Release (2005): Django was originally developed by Adrian Holovaty and Simon Willison while working at the Lawrence Journal-World newspaper. It was first released as open-source software in July 2005, with the goal of enabling developers to build web applications quickly and efficiently.
2. Version 0.90 (2006): The first official release of Django (version 0.90) occurred in September 2006. This version introduced many of the core features that Django is known for, including its ORM (Object-Relational Mapping) system, its templating engine, and its admin interface.

3. Stable Releases and Growth (2007-2010): Over the next few years, Django continued to grow in popularity and maturity. The development team released several stable versions, adding new features, improving performance, and enhancing documentation.
4. Django 1.0 (2008): A significant milestone in Django's evolution was the release of version 1.0 in September 2008. This marked the stabilization of the framework's API and signaled Django's readiness for production use in a wide range of applications.
5. Django 1.x Series (2008-2015): The 1.x series of Django brought further refinements and enhancements to the framework. These included improvements to the ORM, support for more database backends, enhancements to the admin interface, and better support for internationalization and localization.
6. Django 1.11 LTS (2017): Django 1.11 was designated as a Long-Term Support (LTS) release, meaning it would receive security updates and bug fixes for an extended period. LTS releases are particularly important for organizations that require stability and long-term support for their Django applications.
7. Django 2.0 (2017): Django 2.0, released in December 2017, introduced several major changes, including support for Python 3.4 and higher as well as dropping support for Python 2.x. It also introduced asynchronous views and other improvements.
8. Django 3.x Series (2019-2022): The 3.x series of Django continued to build on the improvements introduced in Django 2.0. It further refined support for asynchronous views, introduced new features such as path converters, and continued to improve performance and security.
9. Django 4.0 (2022): Django 4.0, released in December 2022, brought significant changes and improvements, including support for Python 3.10 and higher as well as dropping support for older Python versions. It also introduced stricter settings, improved model inheritance, and other enhancements.

URLs

In Django, URLs are defined in the urls.py file. This file contains a list of URL patterns that map to views. A URL pattern is defined as a regular expression that matches a URL. When a user requests a URL, Django goes through the list of URL patterns defined in the urls.py file and finds the first pattern that matches the URL. If no pattern matches, Django returns a 404 error.

The urls.py file provides a way to map a URL to a view. A view is a Python callable that takes a request and returns an HTTP response. To map a URL to a view, we can create a urlpatterns list in the urls.py file. Each element of the list is a path() or re_path() function call that maps a URL pattern to a view.

Here is an example of a simple urls.py file:

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

```
urlpatterns = [
  path('', views.index, name='index'),
  path('about/', views.about, name='about'),
  path('contact/', views.contact, name='contact'),
]
```

In this example, we have three URL patterns. The first pattern ('') matches the home page, and maps to the index view. The second pattern ('about/') matches the about page, and maps to the about view. The third pattern ('contact/') matches the contact page, and maps to the contact view.

Django also allows us to capture parts of the URL and pass them as arguments to the view function. We can capture parts of the URL by placing them inside parentheses in the URL pattern. For example, the following URL pattern captures an integer id from the URL:

path('post/<int:id>/', views.post_detail, name='post_detail'),

In this example, the view function views.post_detail() takes an integer id as an argument.

Views

Once Django has found a matching URL pattern, it calls the associated view function. A view function takes a request as its argument and returns an HTTP response. The response can be a simple text message, an HTML page, or a JSON object.

Here is an example of a simple view function:

```
from django.http import HttpResponse

def index(request):
  return HttpResponse('Hello, world!')
```

In this example, the index view function takes a request object as its argument and returns an HTTP response with the message "Hello, world!".

Views can also render HTML templates. Templates allow developers to separate the presentation logic from the business logic. Here is an example of a view that renders an

Django URL Confs:

URLconfs in Django are Python modules that define the mapping between URLs and view functions. They serve as a central mechanism for routing incoming HTTP requests to the appropriate view functions or class-based views. Here's how they work:

1. URL Patterns: URLconfs consist of a collection of URL patterns, each of which associates a URL pattern (expressed as a regular expression or a simple string) with a view function or class.

2. Regular Expression Matchers: Django's URL dispatcher uses regular expression matching to determine which view function should handle an incoming request based on the requested URL. When a request comes in, Django compares the URL against each URL pattern in the URLconf until it finds a match.
3. Modular Structure: URLconfs can be organized hierarchically, allowing you to include other URLconfs using the include() function. This modular structure helps in breaking down URL configurations into smaller, more manageable components.
4. Named URL Patterns: URL patterns can be given names, making it easier to reference them in templates or view functions using the {% url %} template tag or the reverse() function.
5. Namespacing: URLconfs support namespacing, which allows you to differentiate between URLs with the same name in different parts of your project. This is particularly useful in large projects with multiple apps.

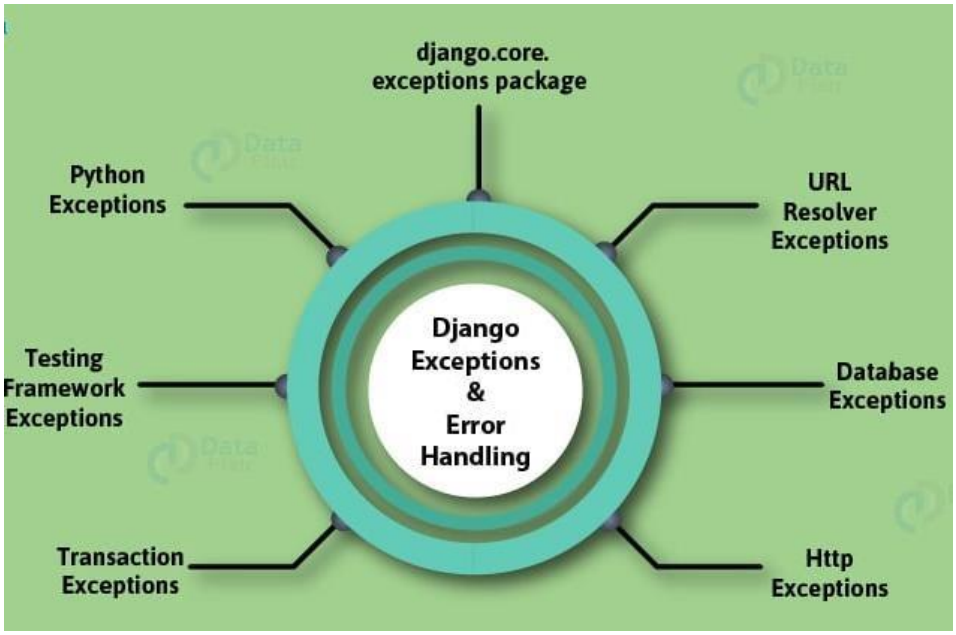
Loose Coupling:

Loose coupling is a software design principle that promotes independence and modularity by minimizing the dependencies between different components of a system. In the context of Django, loose coupling is achieved through:

1. Separation of Concerns: Django encourages the separation of concerns by dividing an application into distinct components, such as models, views, templates, and URLconfs. Each component has a specific responsibility and interacts with other components through well-defined interfaces.
2. Decoupled URLs and Views: In Django's URL routing system, views are decoupled from URLs. Views are Python functions or class-based views that encapsulate the logic for processing requests and generating responses. URLconfs define the mapping between URLs and views but do not directly reference the view functions themselves.
3. Dependency Injection: Django's design promotes dependency injection, allowing components to be loosely coupled by injecting dependencies rather than directly instantiating them. For example, views can accept parameters representing dependencies such as models, forms, or services, making them easier to test and reuse.
4. Pluggable Applications: Django's app-based architecture encourages the development of pluggable, reusable applications that can be easily integrated into different projects. By providing well-defined APIs and extension points, Django promotes loose coupling between applications, enabling greater flexibility and scalability.

Django Exceptions & Errors

There are multiple packages for developer's use. These packages may/ may not come in your use-case. For the matter, the 1st package we are going to discuss is for those developers who are writing their own Python scripts. These developers customize Django at a low level.



1. django.core.exceptions Package

This package provides us with low-level exceptions. It enables us to define new rules for our Django project. We can load models, views in our own defined ways.

This module has use-cases like:

- When you are working on a custom middleware.
- When making some changes to Django ORM.

For that, we will have to understand some very basic concepts.

In the screenshot below, we can see the list of exceptions provided by this package.

1.1. AppRegistryNotReady

This error occurs when the application model is imported before the app-loading process.

When we start our Django server, it first looks for settings.py file.

Django then installs all the applications in the list INSTALLED_APPS. Here, is a part of that process.

App registry in brief:

When Django project starts, it generates an application registry. It contains the information in settings.py and some more custom settings. This registry will keep record and install all the important components.

This registry contains settings-config of all apps inside INSTALLED_APPS. It is the first kernel of your Django project. To use any app or submodule in your project, it has to load-in here first.

It is useful in exception catching when developing your own scripts. It may not occur when using default Django files.

1.2. ObjectDoesNotExist

This exception occurs when we request an object which does not exist. It is the base class for all the DoesNotExist Errors.

ObjectDoesNotExist emerges mainly from get() in Django.

Brief on get():

get() is an important method used to return data from the server. This method returns the object when found. It searches the object on the basis of arguments passed in the get(). If the get() does not find any object, it raises this error.

1.3. EmptyResultSet

This error is rare in Django. When we generate a query for objects and if the query doesn't return any results, it raises this error.

The error is rare because most of the queries return something. It can be a false flag, but for custom lookups this exception is useful.

1.4. FieldDoesNotExist

This one is clear from its name. When a requested field does not exist in a model, this meta.get_field() method raises this exception.

Meta.get_field() method mediates between views and models. When the model objects are in view functions, Django uses this method for searching the fields. This method looks for the requested field in super_models as well. It comes in handy when we have made some changes to models.

1.5. MultipleObjectsReturned

When we expect a query to return a single response/ object but it returns more than one object, then Django raises this exception.

The MultipleObjectsReturned is also an attribute of models. It is necessary for some models to return multiple objects but for others, it cannot be more than one. This exception helps us to get more control over model data.

1.6. SuspiciousOperation

It is one of the security classes of Django. Django raises this error when it has detected any malicious activity from the user. This exception has many subclasses which are helpful in taking better security measures.

When anyone modifies the session_data, Django raises this exception. Even when there is a modification in csrf_token, we get this exception.

There are many subclasses, dealing with very particular problems. These subclasses are:

- DisallowedHost
- DisallowedModelAdminLookup
- DisallowedModelAdminToField
- DisallowedRedirect
- InvalidSessionKey
- RequestDataTooBig
- SuspiciousFileOperation
- SuspiciousMultipartForm
- SuspiciousSession
- TooManyFieldsSent

As we can see, all of them are very particular in some kind of data defect. These exceptions can detect various types of activities on the server. They have helped a lot of developers to build reliable and secure applications.

1.7. PermissionDenied

This exception is the clearest of all. You must have dealt with this exception while working on static files. Django raises this error when we store our static files in a directory which is not accessible.

You can raise this using the try/ except block but it will be more fun, the static files way. To raise it, change static folder settings to hidden or protected.

1.8. ViewDoesNotExist

We all have experienced this one. Websites have a very evolving frontend design and there are frequent modifications which can lead to some faulty urls.

Django checks for all urls and view function by default. If there is something wrong, the server will show an error.

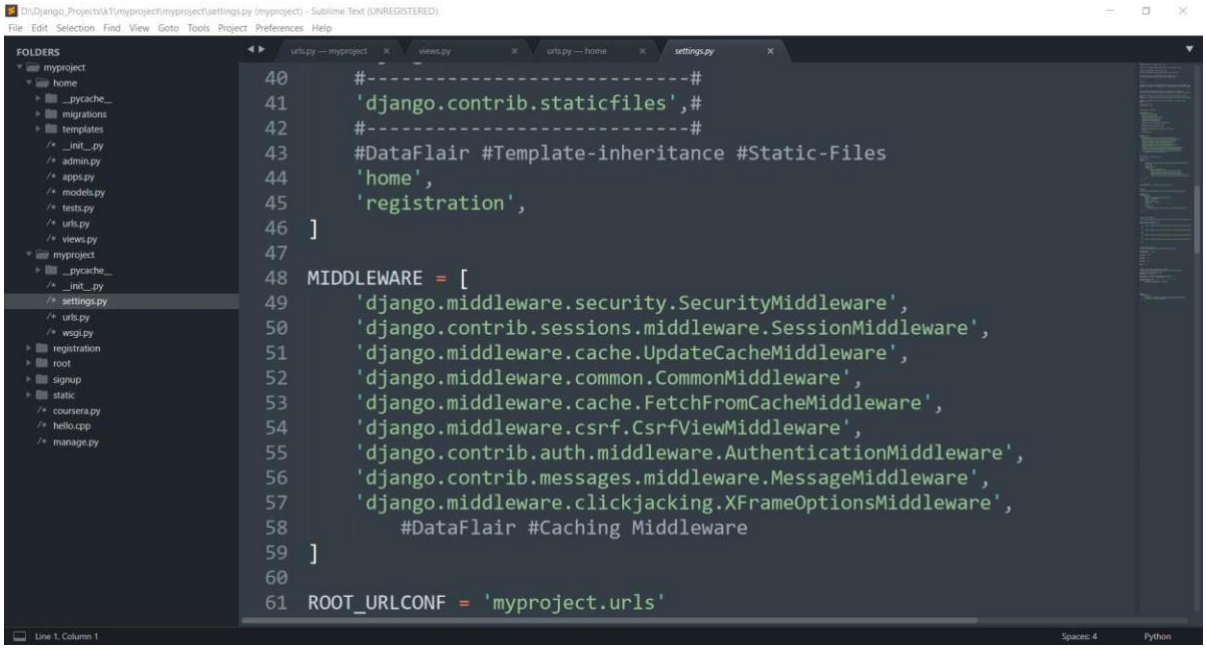
But we can also raise this error when urls-config can’t get the view to load. It is a problem which occurs while using relative URL addressing.

You can implement this from Django Static Files Tutorial.

1.9. MiddlewareNotUsed

Django is very useful at times. It raises this exception when an unused middleware is present in MIDDLEWARES list. It's like the caching middleware. Whenever we have not implemented caching on our website, it will throw this exception.

All the middlewares present in our settings.py are utilized. Most of them are there for the admin app. You can see in the below image that Django comes with pre-installed middlewares that we may require in our application.



1.10. ImproperlyConfigured

You must have encountered this exception when configuring your first project. This exception is for the main settings.py file. If there are some incorrect settings in the main settings then this error will raise. It can also come up if the middlewares or modules do not load properly.

1.11. FieldError

We raise field errors when models have some errors. For example: using the same name in a class is correct syntactically but it can be a problem for Django Models. Therefore, the exceptions will check for these kinds of things.

Cases like:

- Fields in a model defined with the same name
- Infinite loops caused by wrong ordering
- Invalid use of join, drop methods
- Fields name may not exist, etc.

1.12. ValidationError

We used the validation error in validating the form data. This class is a sub-class of django.core.exceptions class. It is extensively used to check data correctness. It will match the data with the model field or forms field. And, it ensures that no security threat is there due to illegal characters in user-input.

You can understand more about them in our Django Forms article.

There can be validation errors which do not belong to any particular field in Django. They are NON_FIELD_ERRORS.

2. URL Resolver Exceptions

This class is a major part of urls.py for defining urls. We import our path function from urls class.

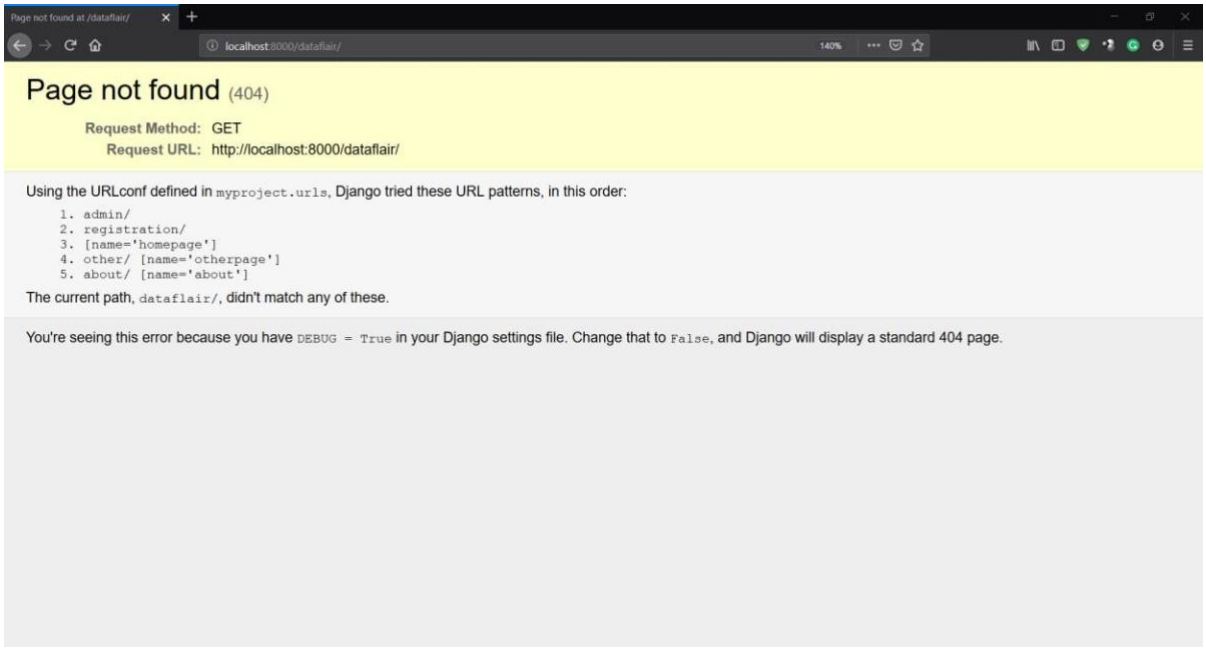
django.urls is one of the core classes of Django without which it might not function. This class also provides some exceptions:

2.1. Resolver404

We raise this exception if the path() doesn't have a valid view to map. The Resolver404 shows us the error page. It is django.http.Http404 module's subclass.

2.2. NoReverseMatch

It is a common occurrence. When we request for a URL which is not defined in our urls-config, we get this error. We all have seen that one page.



This exception is also raised by django.urls module. It applies to regular expressions as well.

3. Database Exceptions

We can import these exceptions from django.db module. The idea behind this implementation is:

Django wraps the standard database exceptions. And, when it happens, our Python code can correspond with the database exception. That makes it easy to deal with database exceptions in Python at a certain level.

The exception wrappers provided by Django work in the same way as Python database API.

The errors are:

- InterfaceError
- DatabaseError
- DataError
- IntegrityError
- InternalError
- ProgrammingError
- NotSupportedError

We raise these errors when:

- Database is not found
- Interface is not there
- Database is not connected
- Input data is not valid etc.

4. Http Exceptions

We used this class in our first views. The HttpResponse is a sub-class of django.http class. The module provides some exceptions and special responses.

UnreadablePostError - This exception occurs when a user uploads a corrupt file or cancels an upload. In both cases, the file received by the server becomes unusable.

5. Transaction Exceptions

A transaction is a set of database queries. It contains the smallest queries or atomic queries. When an error occurs at this atomic level, we resolve them by the django.db.transaction module.

There are errors like TransactionManagementError for all transaction-related problems with the database.

6. Python Exceptions

Django is a Python framework. Of course, we get all the pre-defined exceptions of Python as well. Python has a very extensive library of exceptions. And, you can also add more modules according to use-case.

Django

Django Tutorial provides basic and advanced concepts of Django. Our Django Tutorial is designed for beginners and professionals both.

Django is a Web Application Framework which is used to develop web applications.

Our Django Tutorial includes all topics of Django such as introduction, features, installation, environment setup, admin interface, cookie, form validation, Model, Template Engine, Migration, MVT etc. All the topics are explained in detail so that reader can get enought knowledge of Django

Need some processing to be done on back-end side to interact with servers, we have

1. Servlets
2. PHP
3. Javascript
4. ASP
5. Python - **Django - Python Web Framework**

MVC - Model View Controller

Helps to build good web application
Model - for data
View - HTML form
Controller - To control on operations

In Django - it is called as **MVT** - Model View Template

Features of Django

Rapid Development

Django was designed with the intention to make a framework which takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

Secure

Django takes security seriously and helps developers to avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery etc. Its user authentication system provides a secure way to manage user accounts and passwords.

Scalable

Django is scalable in nature and has ability to quickly and flexibly switch from small to large scale application project.

Fully loaded

Django includes various helping task modules and libraries which can be used to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds etc.

Versatile

Django is versatile in nature which allows it to build applications for different-different domains. Now a days, Companies are using Django to build various types of applications like: content management systems, social networks sites or scientific computing platforms etc.

Open Source

Django is an open source web application framework. It is publicly available without cost. It can be downloaded with source code from the public repository. Open source reduces the total cost of the application development.

Vast and Supported Community

Django is an one of the most popular web framework. It has widely supportive community and channels to share and connect.

Django Simplifies many of the common tasks

1. Database Access - it consist of ORM (Object Relational Manager - abstract layer - easy to interact with database) that simplifies database access
2. Form Validation - easy way to validate data submitted via a forms, check for correct format also
3. User Authentication - Inbuilt authentication system
4. Data Serialization - allow developers to easily convert data into formats like JSON to easily transfer it to the web .

Web application - some software running on the web browser over the internet.

Restful Web API - system that will provide data on proper request from the client.

PYTHON SETUP

As first step we need to

1. install Python (latest Version)
2. Set up a virtual environment

```
C:\Users\VISHNU>pip install virtualenvwrapper-win
Collecting virtualenvwrapper-win
  Downloading virtualenvwrapper_win-1.2.7-py3-none-any.whl.metadata (10 kB)
Collecting virtualenv (from virtualenvwrapper-win)
  Downloading virtualenv-20.25.1-py3-none-any.whl.metadata (4.4 kB)
Collecting distlib<1,>=0.3.7 (from virtualenv->virtualenvwrapper-win)
  Downloading distlib-0.3.8-py2.py3-none-any.whl.metadata (5.1 kB)
Collecting filelock<4,>=3.12.2 (from virtualenv->virtualenvwrapper-win)
  Downloading filelock-3.13.3-py3-none-any.whl.metadata (2.8 kB)
Requirement already satisfied: platformdirs<5,>=3.9.1 in c:\users\vishnu\appdata\local\programs\python\python38\lib\site
-packages (from virtualenv->virtualenvwrapper-win) (3.11.0)
Downloading virtualenvwrapper_win-1.2.7-py3-none-any.whl (18 kB)
Downloading virtualenv-20.25.1-py3-none-any.whl (3.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.8/3.8 MB 302.1 kB/s eta 0:00:04
```

3. Create and Environment

```
C:\Users\VISHNU>mkvirtualenv first
 C:\Users\VISHNU\Envs is not a directory, creating
created virtual environment CPython3.8.0.final.0-64 in 7191ms
  creator CPython3Windows(dest=C:\Users\VISHNU\Envs\first, clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=C:\Users\VISHNU
\AppData\Local\pypa\virtualenv)
    added seed packages: pip==24.0, setuptools==69.1.0, wheel==0.42.0
  activators BashActivator,BatchActivator,FishActivator,NushellActivator,PowerShellActivator,PythonActivator

(first) C:\Users\VISHNU>
```

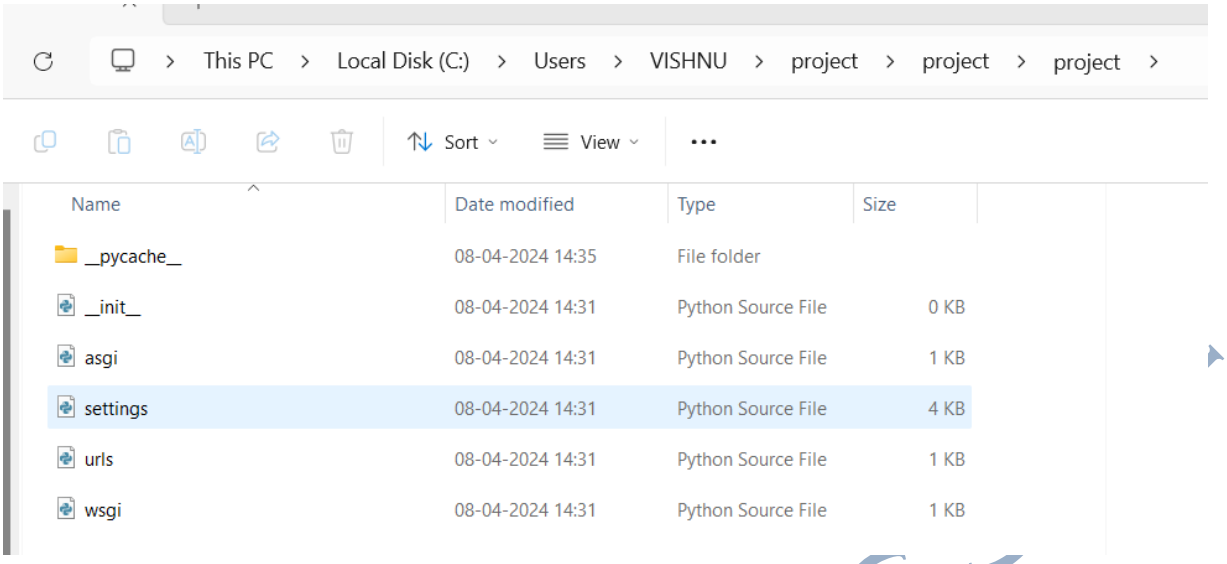
4. Install Django

```
(first) C:\Users\VISHNU>pip install django
WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None, status=None)) after connection broken by 'Read
TimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)")': /simple/django/
WARNING: Retrying (Retry(total=3, connect=None, read=None, redirect=None, status=None)) after connection broken by 'Read
TimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)")': /simple/django/
WARNING: Retrying (Retry(total=2, connect=None, read=None, redirect=None, status=None)) after connection broken by 'Read
TimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)")': /simple/django/
WARNING: Retrying (Retry(total=1, connect=None, read=None, redirect=None, status=None)) after connection broken by 'Read
TimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)")': /simple/django/
Collecting django
  Downloading Django-4.2.11-py3-none-any.whl.metadata (4.2 kB)
Collecting asgiref<4,>=3.6.0 (from django)
  Downloading asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse>=0.3.1 (from django)
  Downloading sqlparse-0.4.4-py3-none-any.whl.metadata (4.0 kB)
Collecting backports.zoneinfo (from django)
  Downloading backports.zoneinfo-0.2.1-cp38-cp38-win_amd64.whl.metadata (4.7 kB)
Collecting tzdata (from django)
  Downloading tzdata-2024.1-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting typing-extensions>=4 (from asgiref<4,>=3.6.0->django)
  Downloading typing_extensions-4.11.0-py3-none-any.whl.metadata (3.0 kB)
Downloading Django-4.2.11-py3-none-any.whl (8.0 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.0/8.0 MB 10.0 MB/s eta 0:00:00
Downloading asgiref-3.8.1-py3-none-any.whl (23 kB)
Downloading sqlparse-0.4.4-py3-none-any.whl (41 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 41.2/41.2 kB ? eta 0:00:00
```

```
(first) C:\Users\VISHNU>django-admin --version
4.2.11

(first) C:\Users\VISHNU>
```

```
(first) C:\Users\VISHNU\project>django-admin startproject project

(first) C:\Users\VISHNU\project>
```

This above command will create a project folder. With some default file and another folder with the same name.

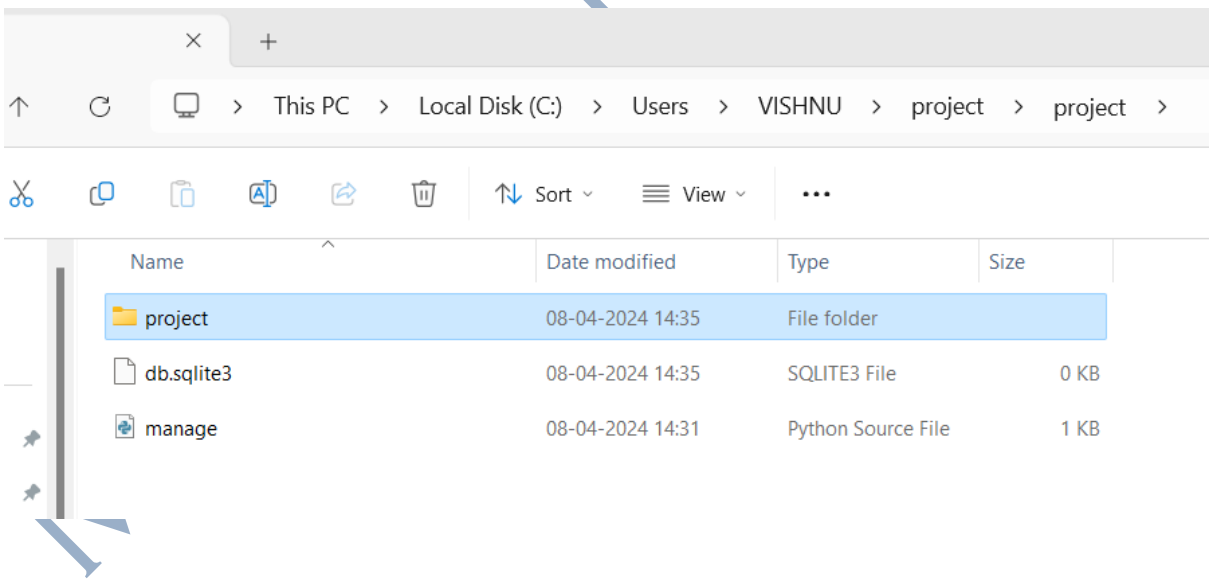


init.py - used to treat our folder (project) as a package

wsgi.py - web server gateway interface - to establish connection between b/w web application and the web server

url - used to handle the urls.

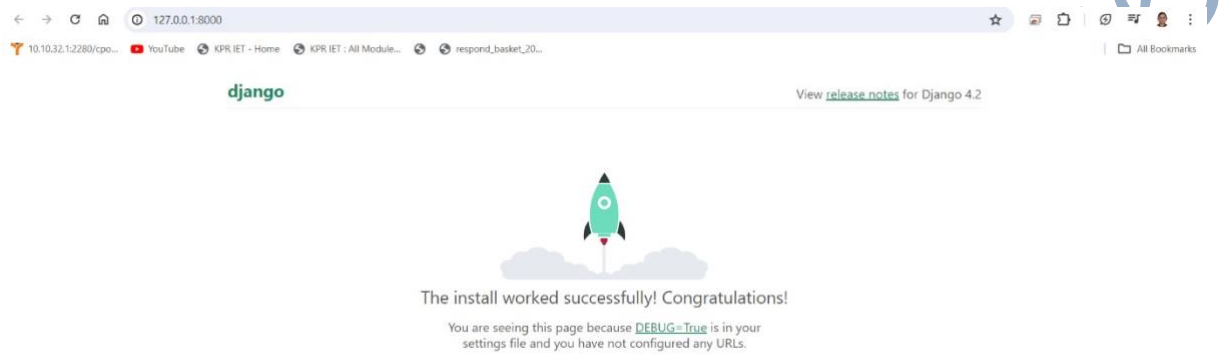
manage.py - to run the server



Use the below command to run the manage.py file

```
(first) C:\Users\VISHNU\project\project>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 08, 2024 - 14:35:16
Django version 4.2.11, using settings 'project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Click on the url and it will open the below page

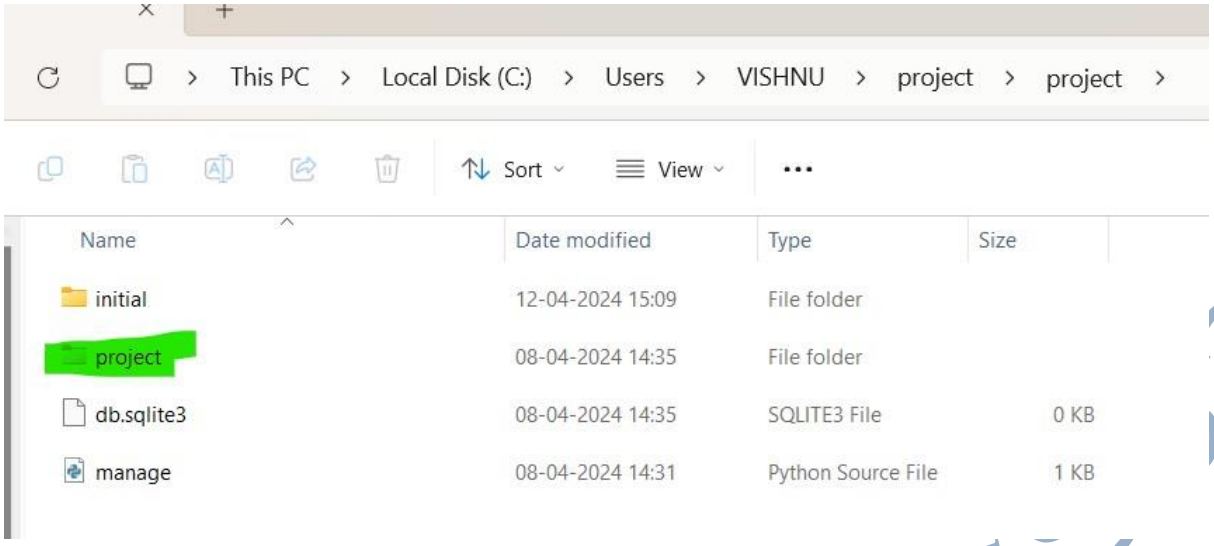


Building First App

Creating new app in the name “initial”

```
PS C:\Users\VISHNU\project\project> python .\manage.py startapp initial
PS C:\Users\VISHNU\project\project>
```

A new folder in the name of “initial” will be created



Now we need to create urls.py and include the below code

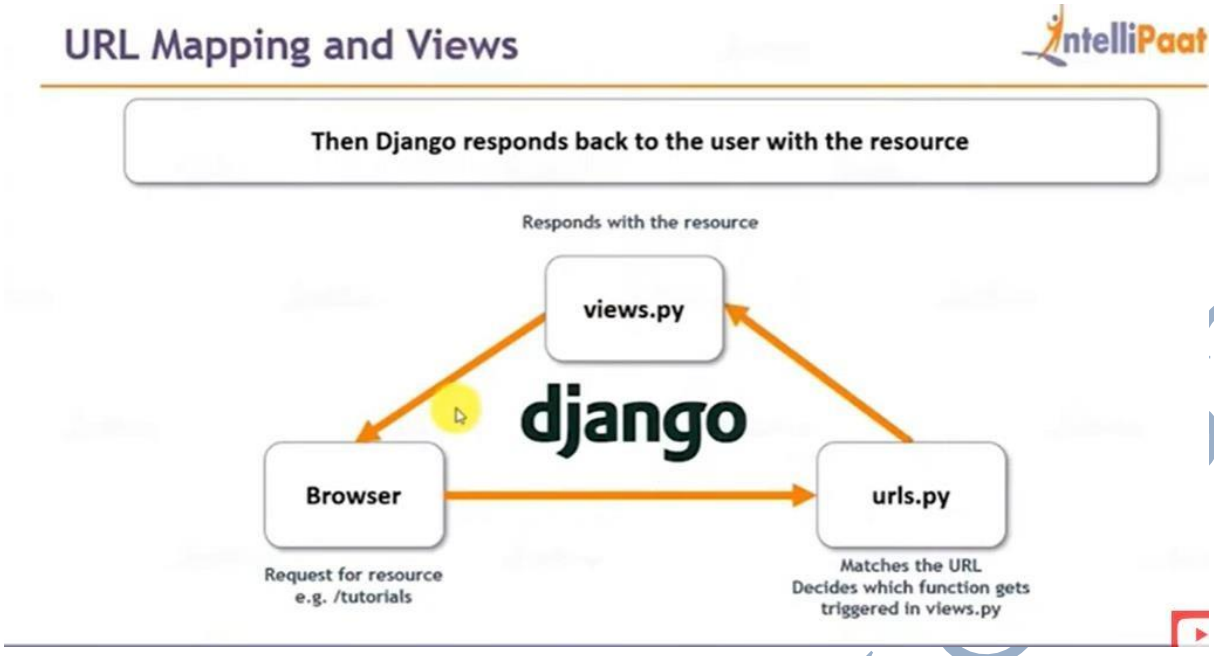
urls.py

```
from django.urls import path

from . import views


urlpatterns = [
    path(' ',views.home, name="home")

]
```

Name = “home” → “home” is a function which is created in views.py. As views.py will accept the user request and send the response

URL Mapping and views

Once the requested URL is mapped, Django will move to views.py file and call a view (Function), which takes a web request and returns a web response. And the response back to the user



Now navigate to views.py and create a home function.

```
from django.http import HttpResponse

# Create your views here.
L

def home(request):
    return HttpResponse("Welcome to my homepage")
```

Now, to include the new app to our main project, do the change in the main project's urls.py file as, include the below line

Using path() function will call the "initial" app urls

```
from django.urls import path, include

path('',include('initial.urls')),
```

Save and run the app using the "**python manage.py runserver**" command, then click on the link or reload the browser. You will the message that need to displayed

Program 1

Develop a Django app that displays current date and time in server

views.py

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def dis_datetime(request):
    import datetime
    x = datetime.datetime.today()
    return HttpResponse(x)
```

Open urls.py in main project folder

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

from datetimeapp import views as vd
from initial import views as vi


urlpatterns = [
    #path('',include('initial.urls')),
    path('admin/', admin.site.urls),
    path('wl/',vi.home),
    path('dt/',vd.dis_datetime), #an url to open datatime views.py
```

```
]
```

PROGRAM 2

Develop a Django app that displays date and time four hours ahead and four hours before as an offset of current date and time in server.

views.py

```
from django.shortcuts import render
from django.http import HttpResponse
from datetime import datetime, timedelta
# Create your views here.
def datetime_display(request):
    # Get current date and time
    current_datetime = datetime.now()

    # Calculate offsets
    four_hours_before = current_datetime - timedelta(hours=4)
    four_hours_ahead = current_datetime + timedelta(hours=4)

    # Prepare the response content
    response_content = (
        f"Current Date and Time: {current_datetime}\n"
        f"Four Hours Before: {four_hours_before}\n"
        f"Four Hours Ahead: {four_hours_ahead}\n"
    )

    # Return the response
    return HttpResponse(response_content, content_type='text/plain')
```

urls.py

```
from django.contrib import admin
```

```
from django.urls import path, include
from datetimeapp import views as vd
from initial import views as vi
urlpatterns = [
    #path('',include('initial.urls')),
    #path('admin/', admin.site.urls),
    #path('wl/',vi.home),
    #path('dt/',vd.dis_datetime),
    path('dt2/',vd.datetime_display),
]
```

Module-2: Django Templates and Models

MVT Structure has the following three parts –

Model: The model is going to act as the interface of your data. It is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database (generally relational databases such as MySql, Postgres).

View: The View is the user interface — what you see in your browser when you render a website. It is represented by HTML/CSS/Javascript and Jinja files.

Template: A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

TEMPLATES

Django provides a convenient way to generate dynamic HTML pages by using its template system.

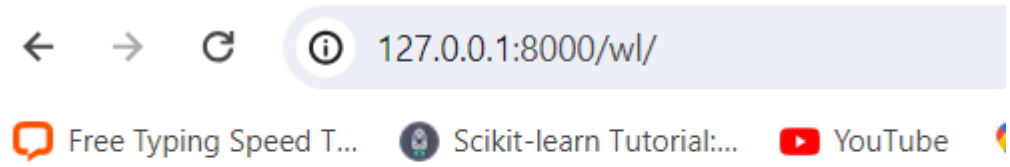
A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

Why we need separate urls.py for multiple apps, because it will be isolated and the changes made in one will not affect the other.

Django's template language (DTL) is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML.

It is possible to added additional styles to my webpage like heading, image, and so on. Which can be done as below. For example

```
def home(request):

    return HttpResponse("<h1>Welcome to my homepage</h1>")
```



Welcome to my homepage

Writing tags directly in the HTTPresponse will not be an efficient way when it comes to a big page or project.

In an HTML file, we can't write python code because the code is only interpreted by the python interpreter not the browser.

We know that **HTML is a static markup language,** while **Python is a dynamic programming language.**

Django template engine is used to separate the design from the python code and allows us to build dynamic web pages.

Now create a new folder “Webpages” and create a new HTML file.

EXAMPLE 1

Index1.html

```
<h1> Using Django Templates </h1>

<p> Django’s template language is designed to strike a balance between power and ease.

   It’s designed to feel comfortable to those used to working with HTML. </p>
```

Now open settings.py file and go to templates and change

```
'DIRS': [os.path.join(BASE_DIR,'Webpages')],
```

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'Webpages')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Next step to call the HTML page in views.py in main app

```
from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.



def home(request):

    return render(request,'index1.html') #render is used to combaine static and dynmaic webpage, index1.html is a html file name
```

urls.py

```
from django.urls import path

from . import views

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

The Render Function

This function takes three parameters –

Request – The initial request.

The path to the template − This is the path relative to the TEMPLATE_DIRS option in the project settings.py variables.

Dictionary of parameters − A dictionary that contains all variables needed in the template. This variable can be created or you can use locals() to pass all local variable declared in the view.

EXAMPLE 2

A variable looks like this: **{{variable}}**. The template replaces the variable by the variable sent by the view in the third parameter of the render function.

To change and display other name rather than Django

Index1.html

```
<h1> Using {{subject}} Templates </h1>

<p> Django's template language is designed to strike a balance between power and ease.

   It's designed to feel comfortable to those used to working with HTML. </p>
```

views.py

```
from django.shortcuts import render

from django.http import HttpResponse

def home(request):

   return render(request,'index1.html',{'subject':'PYTHON'})
```

We are passing **{'subject':'PYTHON'}** as JSON format

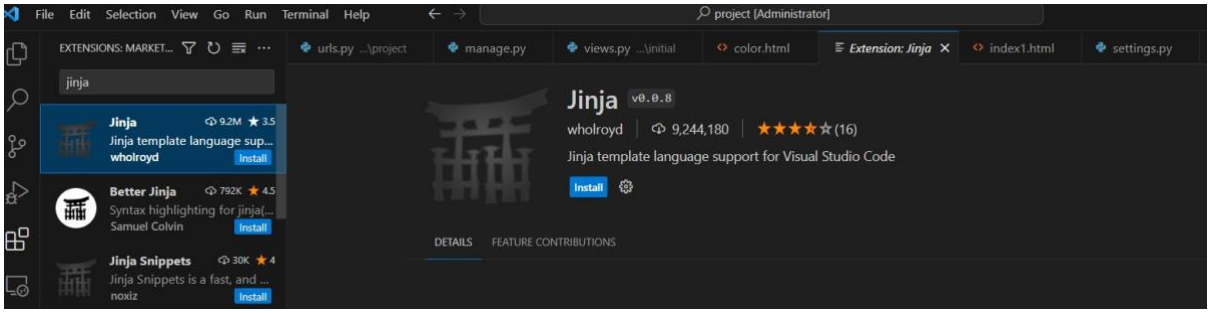
In views.py **{{subject}},** the double brackets means that the code is dynamic, similar to PHP.

EXAMPLE 3

To change the background color

Before this we need to install Jinja

A wild Jinja has appeared, Jinja2 specifically. Jinja is a Python templating engine, aimed at helping you to do dynamic things with your HTML like passing variables, running simple logic, and more! With Jinja, you will notice we are using {% %}, this denotes logic. For variables, you will see {%{% }}. The use of block content and endlock is one of the two major ways of incorporating templates within templates. This method is generally used for header/template combinations, but there is another you could use that I will show later. Next, let's create another HTML file that will serve as whatever will fill this block content.



Color.html

```
<!DOCTYPE html>

<body bgcolor="Yellow">
```

```
    {% block content%}

    {% endblock %}

</body>
```

Index1.html

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

{% block content %}

<h1> Using {{subject}} Templates </h1>

<p> Django's template language is designed to strike a balance between power and ease.

    It's designed to feel comfortable to those used to working with HTML. </p>

{% endblock %}
```

Filters

They help you modify variables at display time. Filters structure looks like the following: {{var|filters}}.

Some examples –

{{string|truncatewords:80}} – This filter will truncate the string, so you will see only the first 80 words.

{{string|lower}} – Converts the string to lowercase.

{{string|escape|linebreaks}} – Escapes string contents, then converts line breaks to tags.

views.py

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


def fliters_ex(request):
    current_date = datetime.now().date()
    current_time = datetime.now().time()
    username = "django"
    email = "jdjango@gmail.com"


    context = {
        'current_date': current_date,
        'current_time': current_time,
        'username': username,
        'email': email,
    }
    return render(request, 'filter_ex.html', context)
```

filter_ex.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Filters Template</title>
</head>
<body>
    <h1> Fitlers example</h1>
    <p>Today's date is: {{ current_date|date:"Y-m-d" }}</p>
    <p>Current time is: {{ current_time|time:"H:i:s" }}</p>
    <p>Your username is: {{ username|upper }}</p>
    <p>Your email is: {{ email|lower }}</p>
</body>
</html>
```

urls.py

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.fliters_ex, name='fliters_ex'),
]
```

urls.py - Main project

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('flt/', include('filters.urls')),

]
```

Tags

Tags lets you perform the following operations: if condition, for loop, template inheritance and more.

for

Loop over each item in an array.

if, **elif**, and **else**

Evaluates a variable, and if that variable is “true” the contents of the block are displayed:

PROGRAM 1

Develop a simple Django app that displays an unordered list of fruits and ordered list of selected students for an event

Student_tag.html

<!DOCTYPE html>

```
<html>
<head>
  <title>Event Details</title>
</head>
<body>
  <h1>Fruits</h1>
  <ul>
    {% for fruit in fruits %}
      <li>{{ fruit }}</li>
    {% endfor %}
  </ul>

  <h1>Selected Students</h1>
  <ol>
    {% for student in students %}
      <li>{% if student.score > 80 %}{{ student.name }}{% endif %}</li>
```

```
    {% endfor %}
  </ol>
</body>
</html>
```

Athlete_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Athletes List</title>
</head>
<body>
  <h1>List of Athletes</h1>
  <ul>
    {% for athlete in athlete_list %}
    <li>{{ athlete }}</li>
    {% endfor %}
  </ul>

  {% if athlete_list %}
    <p>Number of athletes: {{ athlete_list|length }}</p>
```

```
    {% elif athlete_in_locker_room_list %}
        <p>Athletes should be out of the locker room soon!</p>
    {% else %}
        <p>No athletes.</p>
    {% endif %}
</body>
</html>
```

views.py

```
from django.shortcuts import render
def event_details(request):
    fruits = ['Apple', 'Banana', 'Orange', 'Mango']

    # Sample student data with scores
    students = [
        {'name': 'Harish', 'score': 85},
        {'name': 'Briana', 'score': 70},
        {'name': 'John', 'score': 95},
        {'name': 'Arul', 'score': 60},
    ]
    return render(request, 'student_tag.html', {'fruits': fruits, 'students': students})
def athlete_l(request):
```

```
    context = {
        'athlete_list': ['Rajwinder Kaur', 'Chitra Soman', 'Manjeet Kaur '],
        'athlete_in_locker_room_list': []
    }
    return render(request, 'athlete_list.html', context)
```

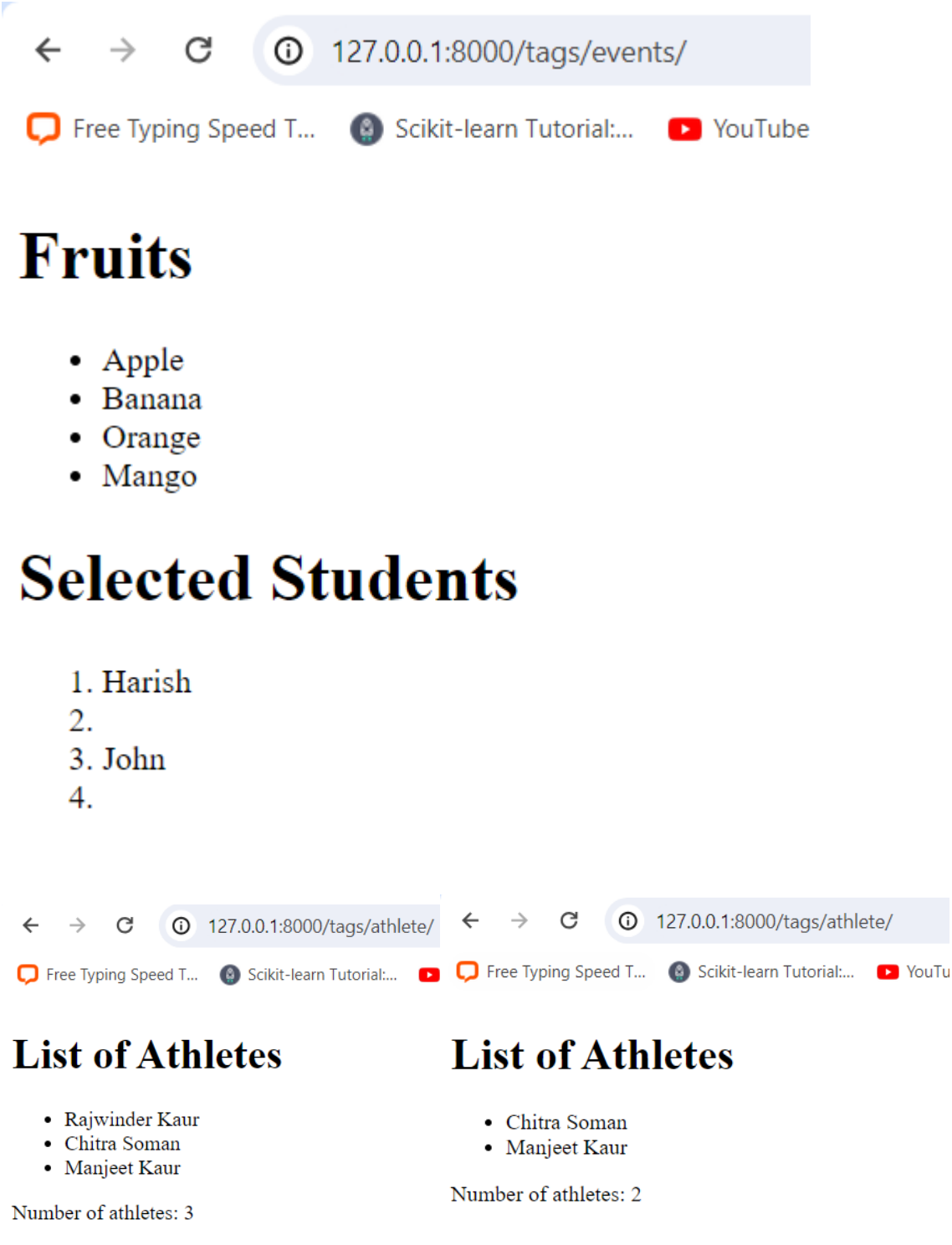
urls.py

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

urlpatterns = [
    path('events/', views.event_details, name='event_details'),
    path('athlete/', views.athlete_l, name='athlete_l'),
]
```

urls.py - Main Project

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('tags/', include('Lab_temp_1.urls')),
```

```
]
```



MVT Development Pattern

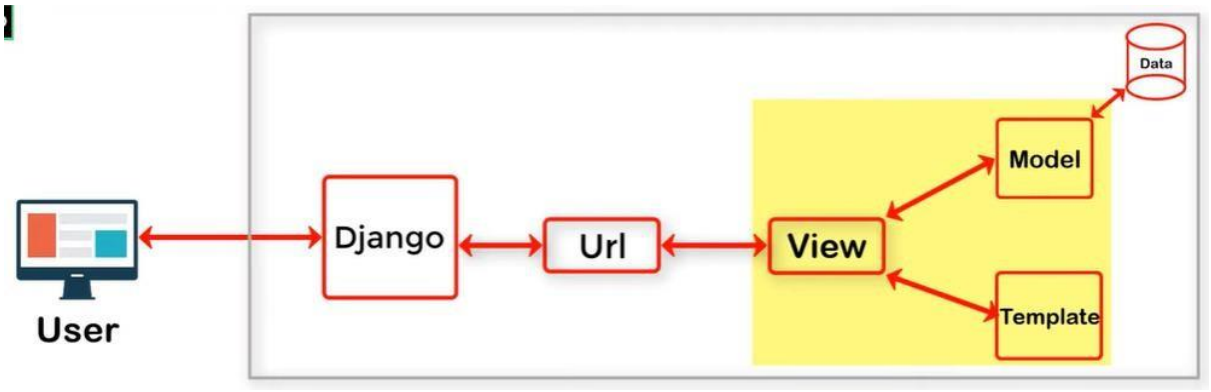
The MVT (Model View Template) is a software design pattern. It is a collection of three important

components: Model View and Template.

The **Model** helps to handle databases. It is a data access layer which handles the data.

The **View** is used to execute the business logic and interact with a model to carry data and render a template.

The **Template** is a presentation layer which handles the User Interface part completely.



Models - will work with Data

Views - will work with Logic

Template - Will work with Layouts

Template Inheritance

Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

EXAMPLE 1

views.py

```
from django.shortcuts import render

def news_article(request, article_id):

    # Add logic to fetch a specific news article by its ID

    article = {'title': 'News Article', 'content': 'Content of the article with ID {}'.format(article_id)}
```

```
    return render(request, 'inh_news_article.html', {'article': article})

def news(request):
    # Add logic to fetch and pass news articles to the template
    context = {
        'articles': [
            {'id': 1, 'title': 'News Article 1', 'content': 'Content of Article 1'},
            {'id': 2, 'title': 'News Article 2', 'content': 'Content of Article 2'},
        ]
    }
    return render(request, 'inh_news.html', context)
```

urls.py

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

urlpatterns = [
    path('news/', views.news, name='news'),
    path('news/<int:article_id>/', views.news_article, name='news_article'),
]
```

```
{% extends "inh_base.html" %}
```

Inh_news.html

```
{% block title %}News - {{ block.super }}{% endblock %}

{% block sidebar %}

  {{ block.super }}

  <ul>

    <li><a href="/news/">Latest News</a></li>

    <li><a href="/news/archive/">News Archive</a></li>

  </ul>

{% endblock %}

{% block content %}

{% for article in articles %}

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

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

{% endfor %}

{% endblock %}
```

Inh_news_article.html

```
{% extends "inh_base.html" %}

{% block title %}{{ article.title }} - {{ block.super }}{% endblock %}

{% block content %}

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

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

{% endblock %}
```

Inh_base.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <title>{% block title %}My amazing site{% endblock %}</title>

</head>

<body>

    <div id="sidebar">

        {% block sidebar %}

        <ul>

            <li><a href="/">Home</a></li>

            <li><a href="/blog/">Blog</a></li>

        </ul>

        {% endblock %}

    </div>

    <div id="content">

        {% block content %}{% endblock %}

    </div>

</body>
```

```
</htmL>
```

CONFIGURING DATABASE

Django officially supports the following databases:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

As first step we need to download a **MySQL** workbench through the below link

MySQL :: Download MySQL Installer

Once installed open the MySQL Command line Client,

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.02 sec)
```

The above command will show the existing or pre-defined database

Using **create database <database_name>** to create a database

```
mysql> create database first;
Query OK, 1 row affected (0.01 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| first              |
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
7 rows in set (0.00 sec)
```

To use the newly created database

```
mysql> use first;
Database changed
mysql> show tables;
Empty set (0.02 sec)
```

MySQL Database Setting in Django

After installing Django and setting up MySQL, you need to configure your Django project to use the MySQL database. Open your Django project's settings.py file, usually located inside the project folder. Look for the DATABASES setting and update it with the MySQL configuration:

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

We need to make migrations, type the below command

```
PS C:\Users\VISHNU\project\data_django> python .\manage.py makemigrations
No changes detected
```

Now type the below command, which will migrate all our fields and table to our mysql database.

```
PS C:\Users\VISHNU\project\data_django> python .\manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
```

Now all the django database has been migrated

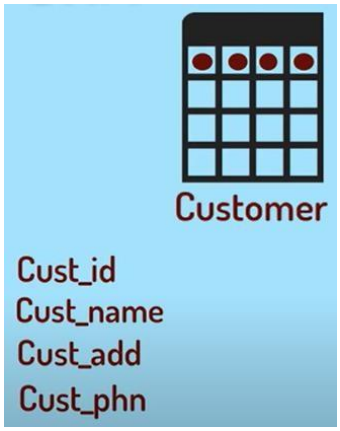
```
mysql> use first;
Database changed
mysql> show tables;
+----------------------------+
| Tables_in_first            |
+----------------------------+
| auth_group                 |
| auth_group_permissions     |
| auth_permission            |
| auth_user                  |
| auth_user_groups           |
| auth_user_user_permissions |
| django_admin_log           |
| django_content_type        |
| django_migrations          |
| django_session             |
+----------------------------+
10 rows in set (0.01 sec)
```

Module-3: Django Admin Interfaces and Model Forms

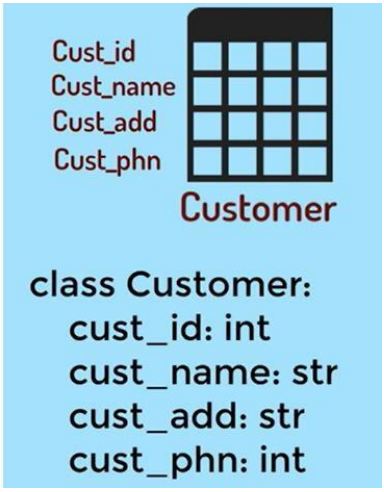
OBJECT RELATIONAL MAPPING (ORM)

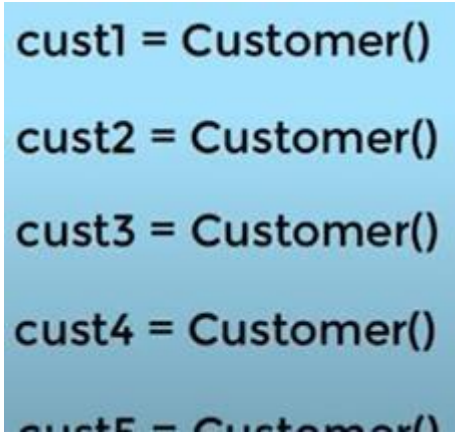
As users we need to interact with databases.

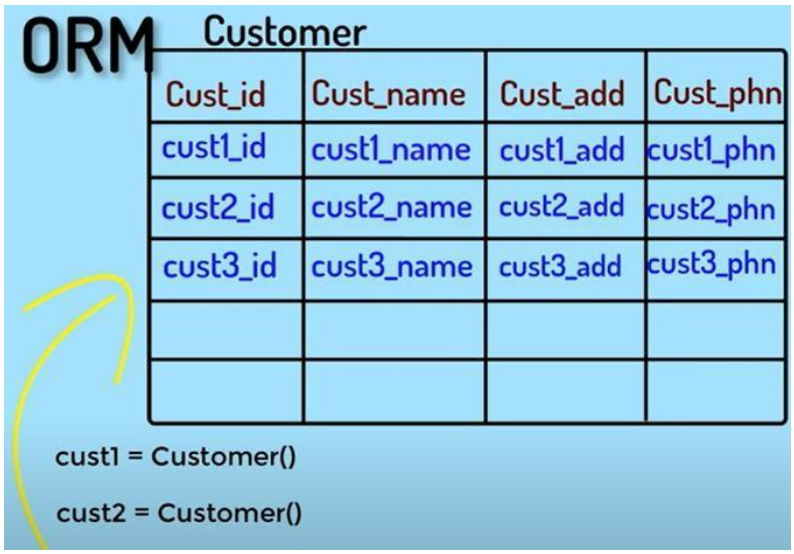
Consider the customer table



Customer class and customer objects







Django officially supports the following databases:

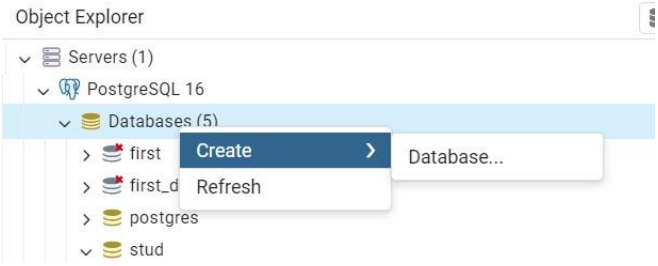
- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

Here we are going to use PostgreSQL, from the below link

https://www.postgresql.org/download/

Once installed, open **pgadmin**, then click on the server and give the password (to be set during the time of installation) to access.

Now create a new database



Once the database was created we need to configure Django

Django and Postgres begin a different softwares, to make it connected we need to install

```
PS C:\Users\VISHNU\project\data_django> pip install psycopg2
Collecting psycopg2
  Downloading psycopg2-2.9.9-cp38-cp38-win_amd64.whl.metadata (4.5 kB)
Downloading psycopg2-2.9.9-cp38-cp38-win_amd64.whl (1.2 MB)
   ---------------------------------------- 1.2/1.2 MB 9.2 MB/s eta 0:00:00
Installing collected packages: psycopg2
Successfully installed psycopg2-2.9.9

[notice] A new release of pip is available: 23.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Psycopg2 - Python - Postgresql Database Adapter

Migration of Models

We can check the different model in https://docs.djangoproject.com/en/5.0/ref/models/fields/

Add contents to views.py

```
from django.shortcuts import render

# Create your views here.

def index(request):

    return render(request,"index.html")
```

Create a Template folder in App and create a new file index.html

```
<!doctype html>

<html lang="en">

  <head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
    <!-- Bootstrap CSS -->

    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/css/bootstrap.min.css" integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">

    <title>Hello, world!</title>

  </head>

  <body>

    <h1>Club</h1>    <!-- Optional JavaScript -->

    <!-- jQuery first, then Popper.js, then Bootstrap JS -->

    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n" crossorigin="anonymous"></script>

    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/js/bootstrap.min.js" integrity="sha384-wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>

  </body>

</html>
```

Project **urls.py**

```
from django.contrib import admin

from django.urls import path, include

from prod.views import index

urlpatterns = [

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

    path('d_u/',index,name = 'index')
```

```
]
```

Now run the command

```
PS C:\Users\VISHNU\project\project> python .\manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 17, 2024 - 09:08:45
Django version 4.2.11, using settings 'project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Open the link http://127.0.0.1:8000/

You will get the “Club” as output

Now modify the **model.py**

```
from django.db import models

# Create your models here.

class Student(models.Model):

    name = models.CharField(max_length=70)

    des = models.TextField()

    def __str_(self):

        return self.name
```

Now modify **admin.py**

```
from django.contrib import admin

#from .models import Destination
```

```
from .models import Student

# Register your models here.

#admin.site.register(Destination)

admin.site.register(Student)
```

Create a link for database in **setting.py**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'django_db', #give the database name
        'USER': 'postgres',
        'PASSWORD': 'vishnu',
        'HOST': 'localhost',  # or the hostname where your MySQL server is running
    }
}
```

MIGRATION OF DATABASE

Run the below command

```
PS C:\Users\VISHNU\project\project> python .\manage.py makemigrations
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory 'C:\Users\VISHNU\project\project\sta
.
Migrations for 'prod':
  prod\migrations\0002_student_delete_destination.py
    - Create model Student
    - Delete model Destination
```

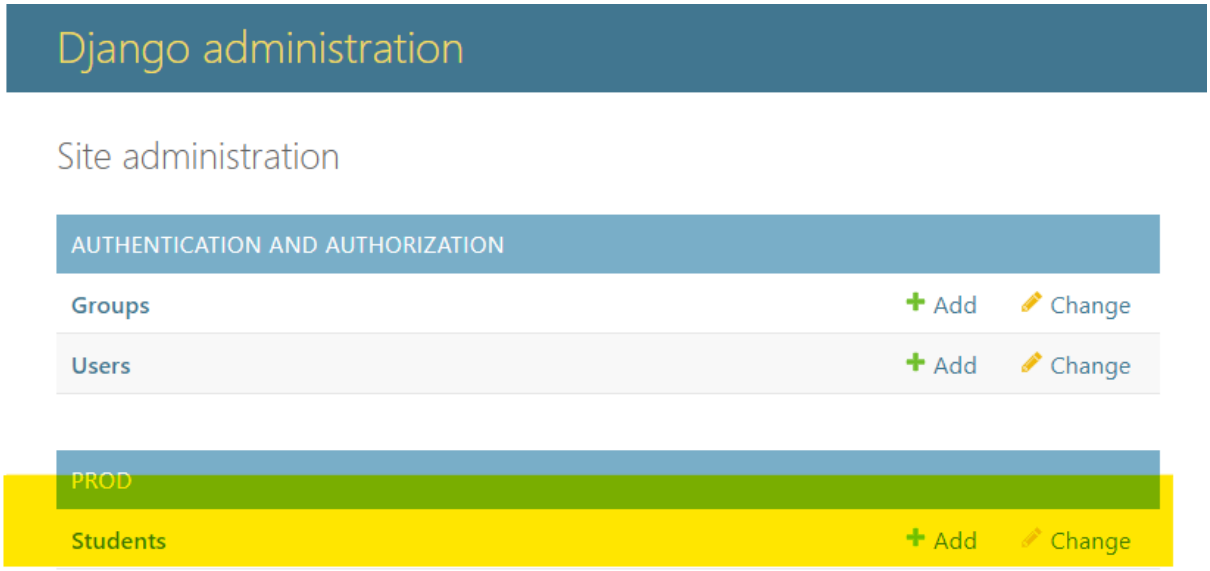
```
PS C:\Users\VISHNU\project\project> python .\manage.py migrate
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory 'C:\Users\VISHNU\project\projec
.
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, prod, sessions
Running migrations:
  Applying prod.0002_student_delete_destination... OK
```

One migration it will create a file in **migration** folder

Now to access the admin page we need to create a super user using the below command

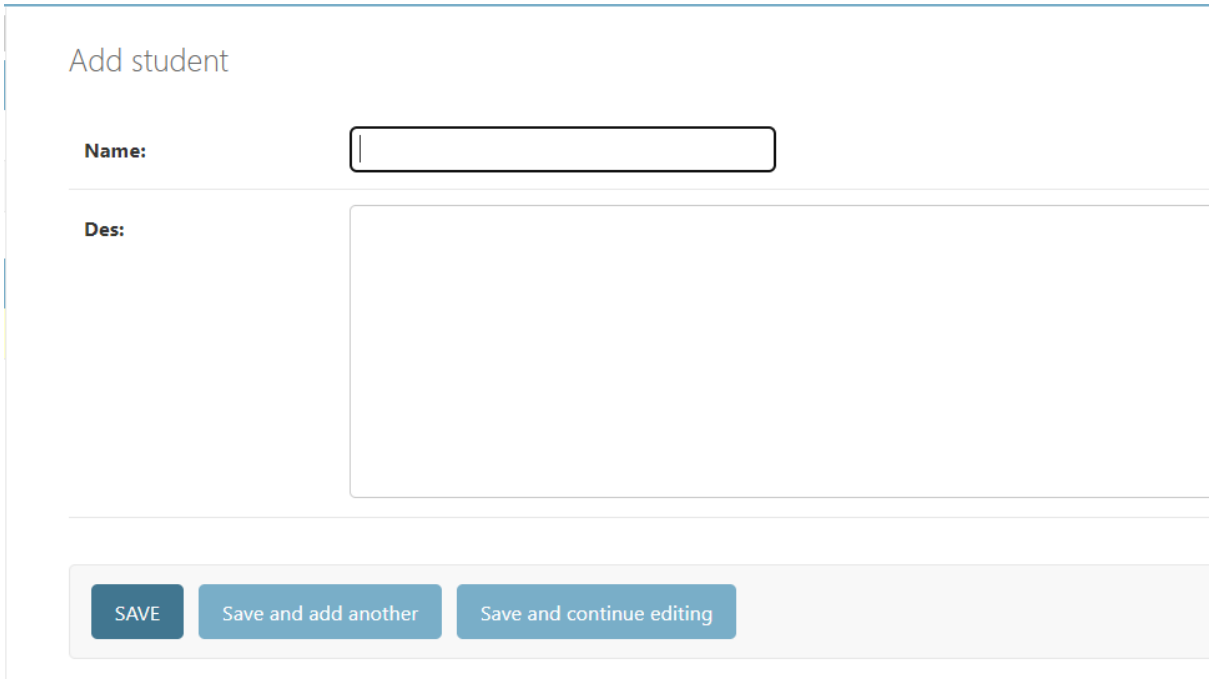
```
PS C:\Users\VISHNU\project\project> python .\manage.py createsuperuser
Username (leave blank to use 'vishnu'): vishnuad
Email address: vishnuad@gmail.com
Password:
Password (again):
The password is too similar to the username.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Now go to http://127.0.0.1:8000/admin and give the above created login credentials.



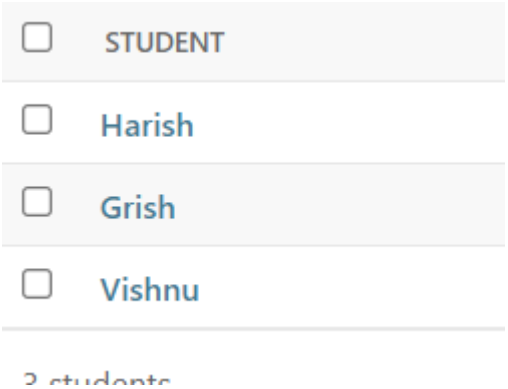
You could see the students table been created

Add Values to the Student table by clicking “Add”



Enter the details and click on “SAVE”

You could now able to see the data that is entered



Now again modify the **views.py** file as below

```
from django.shortcuts import render

from . models import Student

# Create your views here.

def index(request):

    obj = Student.objects.all()

    context={

        "obj":obj,

    }

    return render(request,"index.html",context)
```

Modify the **index.html** which will fetch the datas from the Student Table

```
<!doctype html>

<html lang="en">

  <head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/css/bootstrap.min.css" integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">

  <title>Hello, world!</title>
 </head>
 <body>
  <h1>Club</h1>
  <div class = "container">
     {% for x in obj %}

     <div class="row">
        <div class="com-lg-5 col-md-5 col-12">
           {{x.name}}
        </div>
        <div class="com-lg-5 col-md-5 col-12">
           {{x.des}}
        </div>
     </div>
     {% endfor %}
     </div>
  </div>
  <!-- Optional JavaScript -->
```

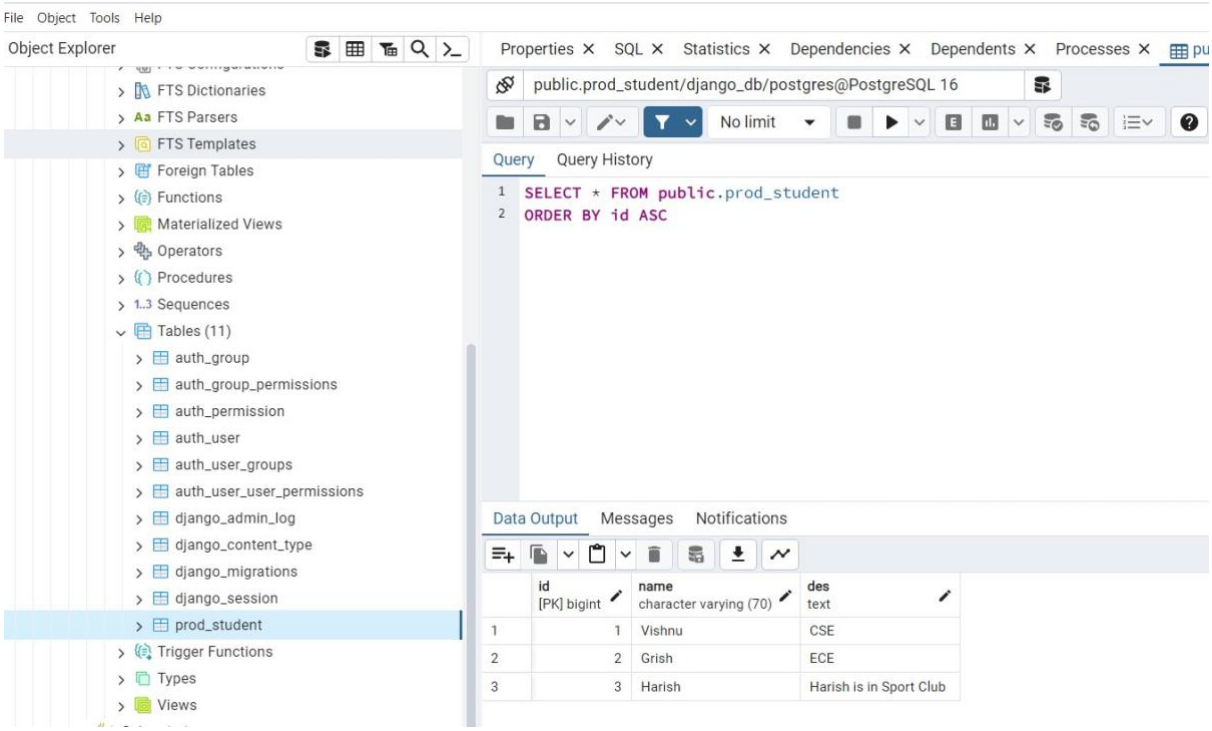
```
  <!-- jQuery first, then Popper.js, then Bootstrap JS -->

  <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
crossorigin="anonymous"></script>

  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>

  <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/js/bootstrap.min.js" integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>

 </body>

</html>
```

Again run the server

```
PS C:\Users\VISHNU\project\project> python .\manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified some issues:

WARNINGS:
```

Go to the url as below you could see the data been fetched from the table

The Same change can be seen in PGADMIN also



Now insert the value in Pgadmin as below

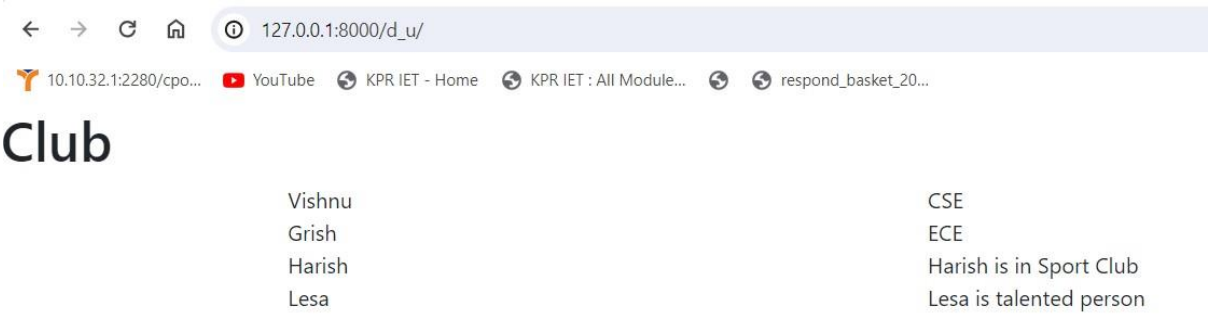
Query Query History

```
1  insert INTO public.prod_student(
2      id,name, des)
3      values (4,'Lesa','Lesa is talented person')
4
5  select * from prod_student
```

Data Output Messages Notifications

	id [PK] bigint	name character varying (70)	des text
1	1	Vishnu	CSE
2	2	Grish	ECE
3	3	Harish	Harish is in Sport Club
4	4	Lesa	Lesa is talented person

The is also been reflected in our webpage also



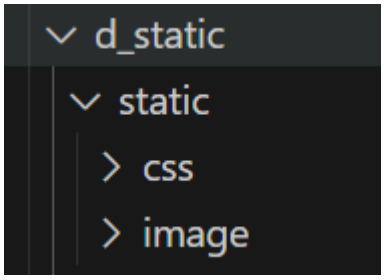
STATIC FILE CREATIONS

Go to the link https://docs.djangoproject.com/en/5.0/howto/static-files/

Add the **STATICFILES_DIRS** to **setting.py**

```
STATICFILES_DIRS = [

    BASE_DIR / "static",

]
```

Create a static and two sub folder image and css inside the project.



Give the below code in views.py

```
from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def home(request):

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

Add the below code in urls.py (Main Project)

```
from django.contrib import admin

from django.urls import path
```

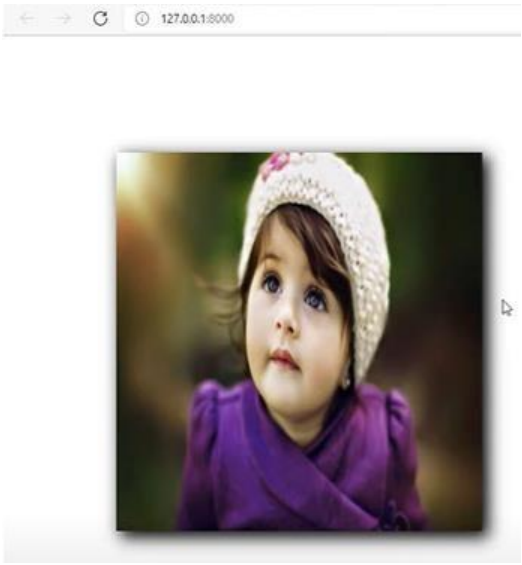
```
from d_static.s_app import views


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

Home.html

```
<html>
    <head>
        <title> Images </title>
        {% load static %}
        <link rel="stylesheet" href = "{% static 'css/style.css' %}">
    </head>
    <body>
        {% load static %}
        <img src="{% static 'Image/download.jpeg' %}" alt="My image">
    </body>
</html>
```

Style.css

```
img{
   height:350px ;
   width: 350px;
}
```

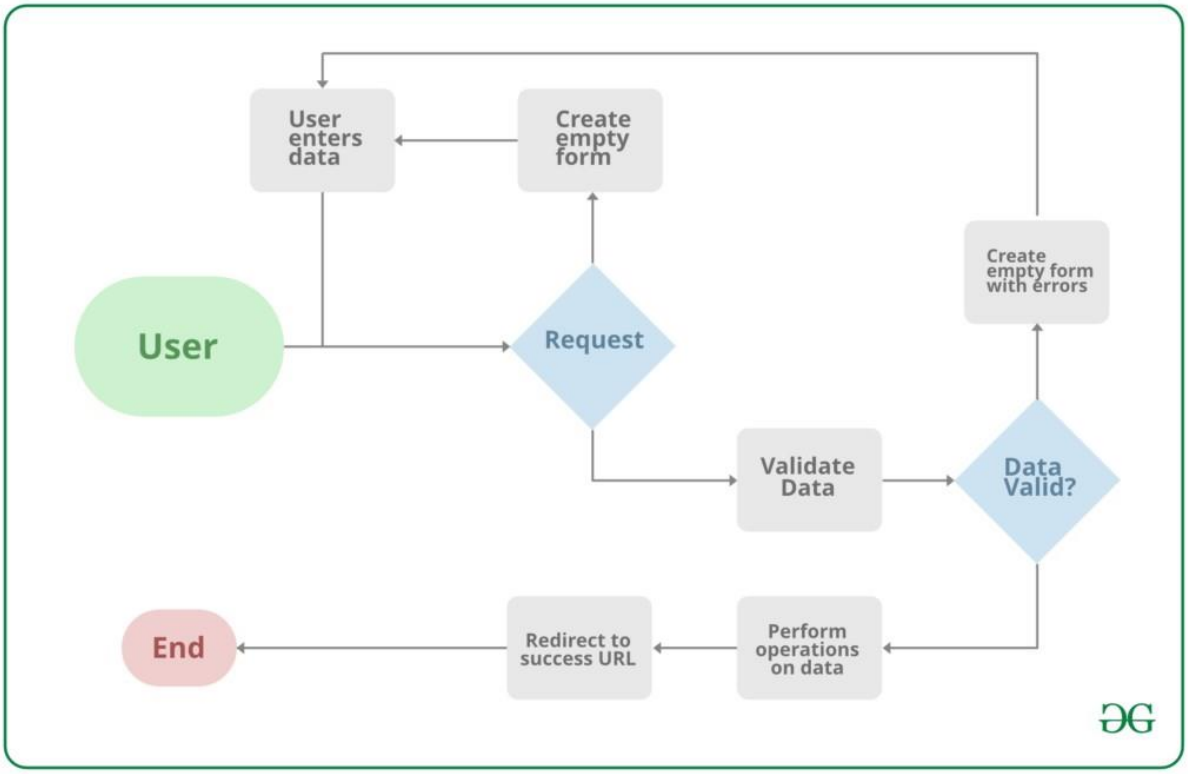


Django Forms

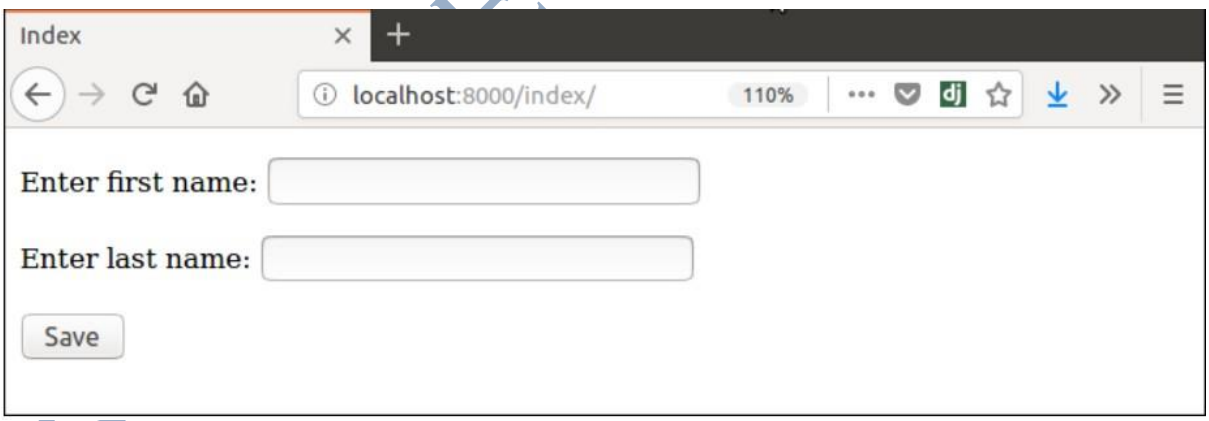
Forms are used for taking input from the user in some manner and using that information for logical operations on databases. For example, Registering a user by taking input such as his name, email, password, etc.

Django maps the fields defined in Django forms into HTML input fields. Django handles three distinct parts of the work involved in forms:

- Preparing and restructuring data to make it ready for rendering.
- Creating HTML forms for the data.
- Receiving and processing submitted forms and data from the client.



Note that all types of work done by forms in Django can be done with advanced HTML stuff, but Django makes it easier and efficient, especially the validation part. Once you get hold of forms in Django you will just forget about HTML forms.



DJANGO FORM SUBMISSION

Start by creating a new **forms.py** file in **myproject** folder.

To access various form-related functionalities, such as **validation** and **rendering**, create a **form** class that inherits Django's **built-in Form class**. This is where you can define the form fields you want to collect from the user. Here's the full code that will go in the file:

```
from django import forms


class ContactForm(forms.Form):
    name = forms.CharField(required=True)
    email = forms.EmailField(required=True)
    message = forms.CharField(widget=forms.Textarea)
```

Copy

This is just a basic example, but you can add additional fields and built-in validations to ensure that the user inputs correct data:

```
from django import forms
```

```
from django.core.validators import EmailValidator


class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.CharField(validators=[EmailValidator()])
    phone = forms.CharField(max_length=15)
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
```

Copy

Once the contact **form** class is created, create a new **views.py** file**,** and in it, add a **view** function that'll render the contact form template and handle the submission:

```
from django.shortcuts import render, redirect
from myproject.forms import ContactForm
from django.http import HttpResponse


def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process the form data
            pass
            return redirect('success')
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})


def success(request):
    return HttpResponse('Success!')
```

Copy

Here, we have defined a **view** function called **contact** that takes a web request and returns a rendered contact form template.

- The **view** function is responsible for:
    - processing the form data submitted by the user
    - checking if the request is a POST request
    - Creating an instance of the **ContactForm** class to validate the submitted data via **is_valid()** method

In this example, we simply **pass**, but you can also:

1. Send form data via email. We'll cover this in more detail with code examples in the next step below.
2. Save form data to a database:

```
import csv
from django.shortcuts import redirect

name = form.cleaned_data['name']
email = form.cleaned_data['email']
message = form.cleaned_data['message']

file = open('responses.csv', 'a')
writer = csv.writer(file)
writer.writerow([name,email,message])
file.close()

return redirect('success')
```

Copy

Note that this approach saves the form data to a CSV file instead of a database. To save form data to a database, you'll need to define a model to store the data. For more information on defining models in Django, check out the Django tutorial on setting up models.

To use the view function we created, you'll also need to map a URL to the view function to create a webpage or endpoint accessible via the URL.

This is done using the **path()** function that defines a URL pattern that points to the **contact()** function. Note that this code should go to the file called **myproject/urls.py**

```
from django.urls import path
from myproject.views import contact, success

urlpatterns = [
    path('contact/', contact, name='contact'),
    path('success/', success, name='success')
]
```

Copy

At this point, you'll need to write the HTML for the contact form from scratch or use a ready-made template file. There are several sources for HTML templates, and the one you choose depends on the specific design and functionality you need.

- Some of the sources can either be:
    - Django's built-in templates
    - Bootstrap or similar frameworks
    - Open-source projects on GitHub

Here are some simple steps you can take to set up a template that works with our guide:

- Add this to TEMPLATES setting in **settings.py**:

```
'DIRS': [
        os.path.join(BASE_DIR, "templates")
    ],
```

Copy

- Create **templates/contact.html** in our project with the following contents

```
<form action="/contact/" method="post">
  {% csrf_token %}
  {% for field in form %}
    <div class="fieldWrapper">
      {{ field.errors }}
```

```
        {{ field.label_tag }} {{ field }}
    </div>
  {% endfor %}
  <input type="submit" value="Submit">
</form>
```

Copy

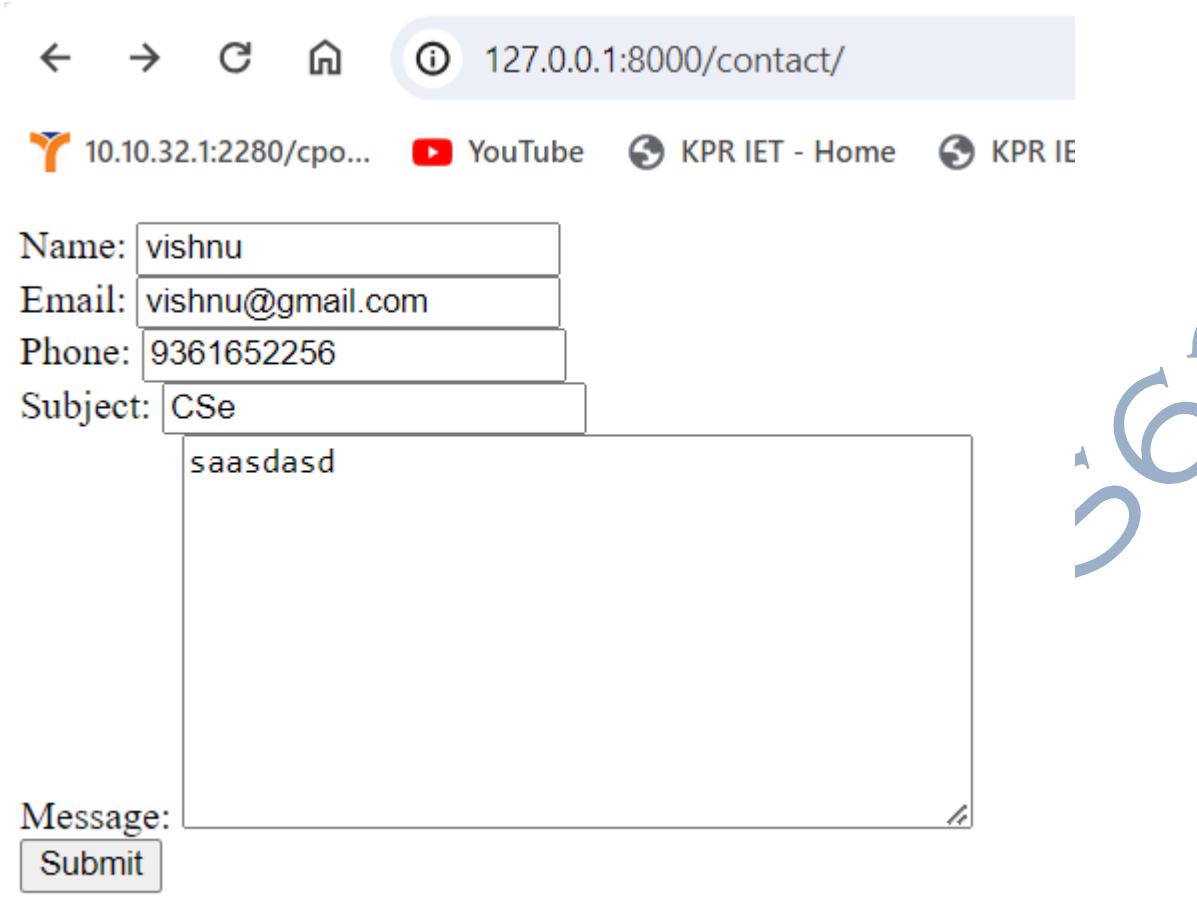
- As a last touch, add a success page to which the user is redirected to after hitting the submit button:

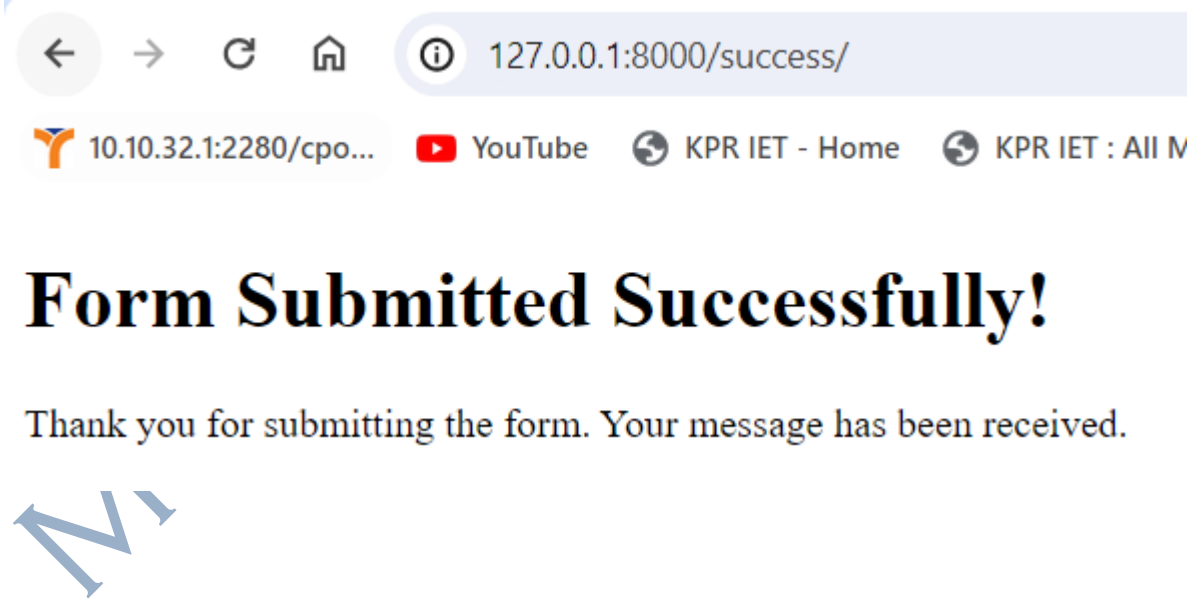
```
from django.shortcuts import render, redirect
from myproject.forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process the form data
            return redirect('success')
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})

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

Copy

To run all of the example codes from this tutorial, and to check the result, execute the following command in your terminal: python3 manage.py runserver

After that, open your web browser and enter this URL http://127.0.0.1:8000/contact/





CREATING MODEL FORMS

It is a class which is used to create an HTML form by using the Model. It is an efficient way to create a form without writing HTML code.

Django automatically does it for us to reduce the application development time. For example, suppose we

have a model containing various fields, we don't need to repeat the fields in the form file.

For this reason, Django provides a helper class which allows us to create a Form class from a Django model.

Let's see an example.

Django ModelForm Example

First, create a model that contains fields name and other metadata. It can be used to create a table in database and dynamic HTML form.

// model.py

```
1. from__future__import unicode_literals
2. from django.db import models
3.
4. class Student(models.Model):
5.     first_name = models.CharField(max_length=20)
6.     last_name  =  models.CharField(max_length=30)
7.     class Meta:
8.         db_table = "student"
```

This file contains a class that inherits ModelForm and mention the model name for which HTML form is created.

// form.py

```
1. from django import forms
2. from myapp.models import Student
3.
4. class EmpForm(forms.ModelForm):
5.     class Meta:
6.         model = Student
7.         fields = "__all__"
```

Write a view function to load the ModelForm from forms.py.

//views.py

```
1. from django.shortcuts import render
2. from myapp.form import StuForm
3.
4. def index(request):
5.     stu = EmpForm()
6.     return render(request,"index.html",{'form':stu})
```

//urls.py

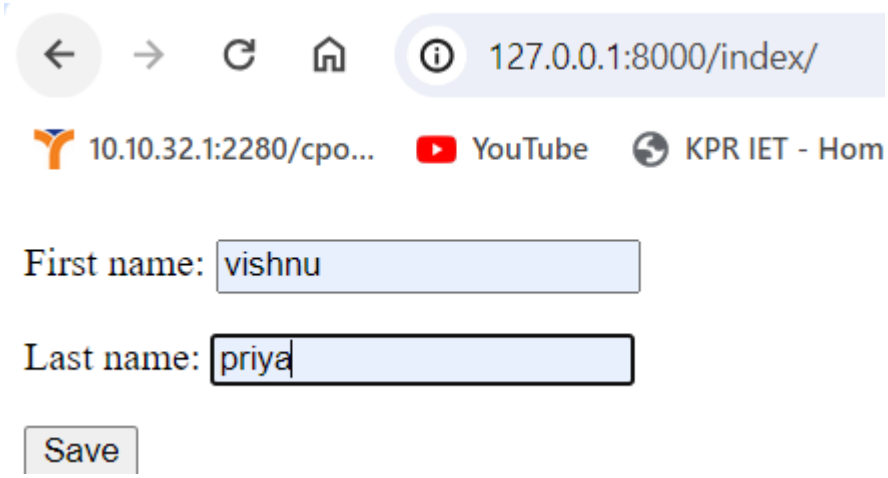
```
1. from django.contrib import admin
2. from django.urls import path
3. from myapp import views
4. urlpatterns = [
5.     path('admin/', admin.site.urls),
6.     path('index/', views.index),
7. ]
```

And finally, create a **index.html** file that contains the following code.

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>Index</title>
6. </head>
7. <body>
8.     <form method="POST" class="post-form">
9.         {% csrf_token %}
10.         {{ form.as_p }}
11.         <button type="submit" class="save btn btn-default">Save</button>
12.     </form>
13. </body>
14. </html>
```

Run Server

Run the server by using **python manage.py runserver** command.



Configuring URLs in Django

1. Understanding URLs in Django

URLs, or Uniform Resource Locators, are essential for web applications as they define the address of each resource within your application. In Django, URLs are used to map user requests to appropriate views, which in turn render the correct templates and data. Django's URL handling is designed to be flexible, easy to configure, and scalable.

2. Creating and Configuring URL Patterns

To configure URLs in Django, you need to create URL patterns or routes that define the mapping between URLs and views. Here are the steps to create and configure URL patterns:

Step 1: Define URL Patterns in your App

In your Django app, create a new Python module called urls.py if it doesn't already exist. Inside this file, you will define all the URL patterns for this specific app. To do this, you need to import the necessary modules, create a variable called urlpatterns, and add the URL patterns to it.

Here's a basic example:

```
from django.urls import path

from . import views
```

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

In this example, we import the path function from django.urls and the views module from the current directory. Then, we define two URL patterns: one for the home page and another for the about page. The first argument to the path function defines the URL pattern, the second argument specifies the view function that should be called when the pattern is matched, and the name parameter assigns a name to the URL pattern for easy referencing later on.

Project-Level URL Configuration (urls.py):

Next, you need to include the URL patterns of your app in the project's urls.py file. This is necessary for Django to know about the URLs defined in your app. To achieve this, use the include function from django.urls. Here's an example:

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

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

PythonCopy

In this example, we import the include function and add a new URL pattern that includes the URLs from the myapp.urls module. This means that all the URLs defined in myapp.urls will now be accessible with the prefix myapp/.

3. Using Views and URL Namespaces

When configuring URLs in Django, using views and URL namespaces can make your application more organized and maintainable. Here are some tips:

- Use Function-based Views or Class-based Views: Django supports both function-based views and class-based views. Function-based views are simple Python functions that take a request object and return a response object, while class-based views are more structured and provide greater flexibility and reusability. Choose the one that best suits your needs and coding style.
- Use URL Namespaces: URL namespaces allow you to group related URL patterns under a common prefix, making it easier to manage and reference them. To use URL namespaces, add an app_name variable in your app's urls.py file and use the include function with the namespace parameter in your project's urls.py file.

4. Best Practices for Configuring URLs in Django

Here are some best practices for configuring URLs in Django:

- Keep URL Patterns Simple and Readable: Use simple, descriptive, and readable URL patterns that convey the purpose of the resource. Avoid using complex patterns, numbers, or special characters that could confuse users and search engines.
- Use the reverse() Function: When referencing URLs in your views and templates, use the reverse() function instead of hardcoding URLs. This makes your application more maintainable and less prone to errors.
- Organize URL Patterns by App: Keep the URL patterns related to a specific app in its own urls.py file, and include them in the project's urls.py file. This keeps your URL patterns organized and easier to manage.

In conclusion, configuring URLs in Django is an essential skill for software developers working on web applications. By following the guidelines and best practices outlined in this tutorial, you can create a well-organized, maintainable, and user-friendly application. This knowledge will also be valuable for remote Python developers who are working on Django projects.

URL Dispatchers - Including other URL

In Django, the URL dispatcher is a core component responsible for matching incoming HTTP requests with the corresponding view functions. It helps Django determine which view function should be called based on the requested URL.

How it works:

1. Incoming Request: When a user sends a request to your Django application, Django's URL dispatcher examines the URL provided in the request.
2. URL Patterns: Django's URL dispatcher then compares the requested URL against a list of URL patterns defined in your URL configuration.
3. Match: If the requested URL matches any of the URL patterns, Django calls the associated view function.
4. Response: The view function processes the request and returns an HTTP response, which Django sends back to the user's browser.

Let's create a simple Django project called url_dispatcher_example with two apps: articles and blog. We'll demonstrate how to use the URL dispatcher to include URL configurations from both apps.

Step 1: Create the Django Project and Apps
First, create a Django project named url_dispatcher_example:

```
PS C:\Users\VISHNU\project> django-admin startproject url_dispatcher_example
PS C:\Users\VISHNU\project> cd .\url_dispatcher_example\
```

Then, create two Django apps named articles and blog:

```
PS C:\Users\VISHNU\project\url_dispatcher_example> python manage.py startapp articles
PS C:\Users\VISHNU\project\url_dispatcher_example> python manage.py startapp blog
PS C:\Users\VISHNU\project\url_dispatcher_example>
```

Define URL Patterns

Project-Level URL Configuration (url_dispatcher_example/urls.py):

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

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('articles/', include('articles.urls')),  # Include articles app URLs
    path('blog/', include('blog.urls')),  # Include blog app URLs
]
```

articles App-Level URL Configuration (articles/urls.py):

```
from django.urls import path

from . import views

urlpatterns = [

    path('', views.article_list, name='article_list'),  # Article list page

    path('<int:pk>/', views.article_detail, name='article_detail'),  # Article detail page

]
```

blog App-Level URL Configuration (blog/urls.py):

```
from django.urls import path

from . import views



urlpatterns = [

    path('', views.blog_home, name='blog_home'),  # Blog home page

    path('<int:post_id>/', views.blog_post, name='blog_post'),  # Individual blog post page

]
```

Step 3: Define Views

Views for articles App (articles/views.py):

```
from django.shortcuts import render

from django.http import HttpResponse



def article_list(request):

    return HttpResponse("This is the list of articles.")



def article_detail(request, pk):

    return HttpResponse(f"This is the detail view for article ID {pk}.")
```

Views for blog App (blog/views.py):

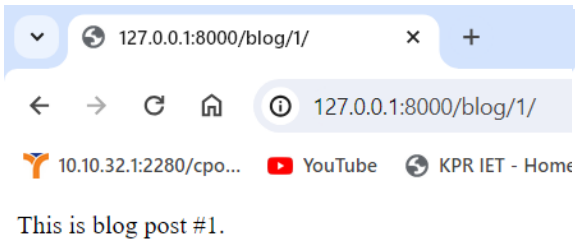
```
from django.shortcuts import render
from django.http import HttpResponse

def blog_home(request):
    return HttpResponse("Welcome to the blog!")

def blog_post(request, post_id):
    return HttpResponse(f"This is blog post #{post_id}.")
```

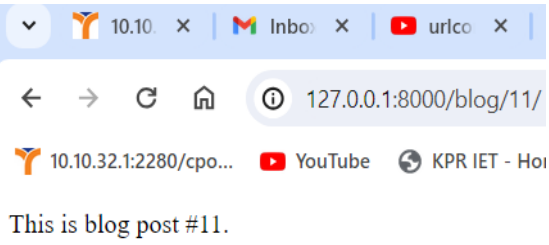
Step 4: Run the Server

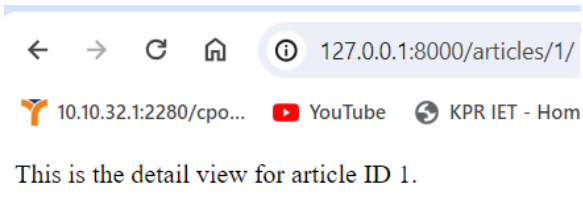
Finally, run the Django development server: **python .\manage.py runserver**

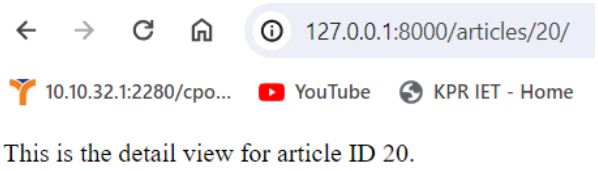
Now, you can navigate to URLs like /articles/ or /blog/ in your browser to see the respective pages served by

each app. For example, /articles/ will display the list of articles, and /blog/ will display the blog home page.









MODULE - 4 - Generic Views and Django State Persistence

Generic Views in Django Rest Framework (DRF)

Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django applications. One of its standout features is the use of Generic Views, which streamline the process of creating API endpoints for common operations like creating, reading, updating, and deleting objects.\ In this blog post, we'll take a deep dive into Generic Views in DRF and explore how they can simplify API development.

Key Benefits of Using Generic Views

1. **Code Reusability:** Generic Views encapsulate common API patterns, so you can reuse them across multiple endpoints and models. This reduces code duplication and makes your codebase more maintainable.
2. **Consistency:** DRF's Generic Views follow a consistent naming convention and structure, which improves the overall organization of your code and makes it easier for other developers to understand and work with.
3. **Saves Time:** Writing CRUD (Create, Read, Update, Delete) views for every model can be time-consuming. Generic Views save you time by providing pre-implemented views for these operations.
4. **Flexibility:** While Generic Views provide sensible defaults, they are highly customizable. You can easily override their behavior to suit your specific requirements.

Types of Generic Views:

1. ListView: Displays a list of objects from a queryset.
2. DetailView: Displays the details of a single object.
3. CreateView: Handles the creation of new objects.
4. UpdateView: Handles the updating of existing objects.
5. DeleteView: Handles the deletion of objects.
6. FormView: Handles displaying a form and processing form submissions.
7. TemplateView: Renders a template without any model data.

EXAMPLE - BOOK STORE

models.py

```
# models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    publication_year = models.IntegerField()

    def __str_(self):
        return self.title
```

urls.py

```
"""
URL configuration for g_views project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns: path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path
from form_temp_views.views import ContactView, AboutUsView
from type_views.views import BookListView, BookDetailView, BookCreateView, BookUpdateView, BookDeleteView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('books/', BookListView.as_view(), name='book_list'),
```

```
    path('books/<int:pk>/', BookDetailView.as_view(), name='book_detail'),
    path('books/create/', BookCreateView.as_view(), name='book_create'),
    path('books/<int:pk>/update/', BookUpdateView.as_view(), name='book_update'),
    path('books/<int:pk>/delete/', BookDeleteView.as_view(), name='book_delete'),
    path('contact/', ContactView.as_view(), name='contact'),
    path('about-us/', AboutUsView.as_view(), name='about_us'),
]
```

views.py

```
# views.py

from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView,FormView,TemplateView

from django.urls import reverse_lazy

from .forms import ContactForm

from type_views.models import Book



class BookListView(ListView):

    model = Book

    template_name = 'book_list.html'



class BookDetailView(DetailView):

    model = Book

    template_name = 'book_detail.html'
```

```
class BookCreateView(CreateView):
    model = Book
    template_name = 'book_form.html'
    fields = ['title', 'author', 'publication_year']
    success_url = reverse_lazy('book_list')

class BookUpdateView(UpdateView):
    model = Book
    template_name = 'book_form.html'
    fields = ['title', 'author', 'publication_year']
    success_url = reverse_lazy('book_list')

class BookDeleteView(DeleteView):
    model = Book
    template_name = 'book_confirm_delete.html'
    success_url = reverse_lazy('book_list')

class AboutUsView(TemplateView):
```

```
    template_name = 'about_us.html'



class ContactView(FormView):

    template_name = 'contact.html'

    form_class = ContactForm

    success_url = '/thank-you/'



    def form_valid(self, form):

        # Process the form data here (e.g., send an email)

        return super().form_valid(form)
```

Create Templates

Create templates for each view:

- book_list.html: Display a list of books.
- book_detail.html: Display details of a single book.
- book_form.html: Form for creating and updating books.
- book_confirm_delete.html: Confirmation page for deleting a book.

Book_list .html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Book List</title>
</head>
```

```
<body>
    <h1>Book List</h1>
    <ul>
        {% for book in object_list %}
            <li>{{ book.title }} by {{ book.author }} ({{ book.publication_year }})</li>
        {% empty %}
            <li>No books available</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Book_details.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Book Detail</title>

</head>

<body>

    <h1>Book Detail</h1>

    <p>Title: {{ object.title }}</p>

    <p>Author: {{ object.author }}</p>

    <p>Publication Year: {{ object.publication_year }}</p>

    <a href="{% url 'book_update' object.pk %}">Edit</a>

    <a href="{% url 'book_delete' object.pk %}">Delete</a>
```

```
</body>

</html>
```

Book_from.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>{% if form.instance.pk %}Edit Book{% else %}Create Book{% endif %}</title>

</head>

<body>

    <h1>{% if form.instance.pk %}Edit Book{% else %}Create Book{% endif %}</h1>

    <form method="post">

        {% csrf_token %}

        {{ form.as_p }}

        <button type="submit">Submit</button>

    </form>

</body>

</html>
```

Book_confirm_delete.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Delete Book</title>

</head>

<body>

    <h1>Delete Book</h1>

    <p>Are you sure you want to delete "{{ object.title }}"?</p>

    <form method="post">

        {% csrf_token %}

        <button type="submit">Confirm Delete</button>

    </form>

    <a href="{% url 'book_list' %}">Cancel</a>

</body>

</html>
```

Contact.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">
```

```
    <title>Contact Form</title>
</head>
<body>
    <h1>Contact Us</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

About_us.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>About Us</title>
</head>
<body>
    <h1>About Us</h1>
    <p>Welcome to our website! We provide...</p>
</body>
</html>
```

Now do the following changes in Project - **setting.py**

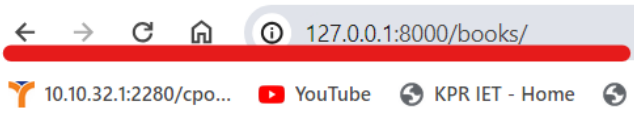
```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
         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

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'django_db', #give the database name
        'USER': 'postgres',
        'PASSWORD': 'vishnu',
        'HOST': 'localhost',  # or the hostname where your MySQL server is running
    }
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'type_views',
]
```

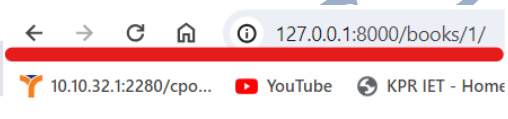
Migrate the model and then run the code follow the below commands

```
PS C:\Users\VISHNU\project\g_views> python .\manage.py makemigrations
Migrations for 'type_views':
  type_views\migrations\0001_initial.py
    - Create model Book
PS C:\Users\VISHNU\project\g_views> python .\manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, type_views
Running migrations:
  Applying type_views.0001_initial... OK
PS C:\Users\VISHNU\project\g_views> python .\manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
```



Book List

- C programming by Dennis (2008)
- Theory of Computation by Kamthane (2014)
- C programming Apporach by Dennis (2008)



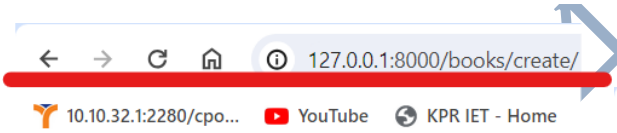
Book Detail

Title: C programming Apporach

Author: Dennis

Publication Year: 2008

Edit Delete



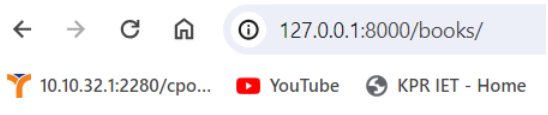
Create Book

Title: Machine Learning

Author: Dennis

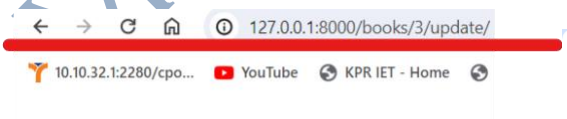
Publication year: 2020

Submit



Book List

- C programming by Dennis (2008)
- Theory of Computation by Kamthane (2014)
- C programming Apporach by Dennis (2008)
- Machine Learning by Dennis (2020)



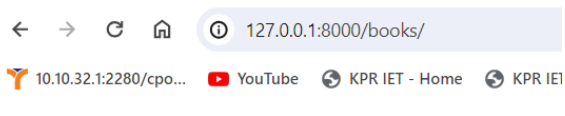
Edit Book

Title: ning A Complete Reference

Author: Dennis

Publication year: 2008

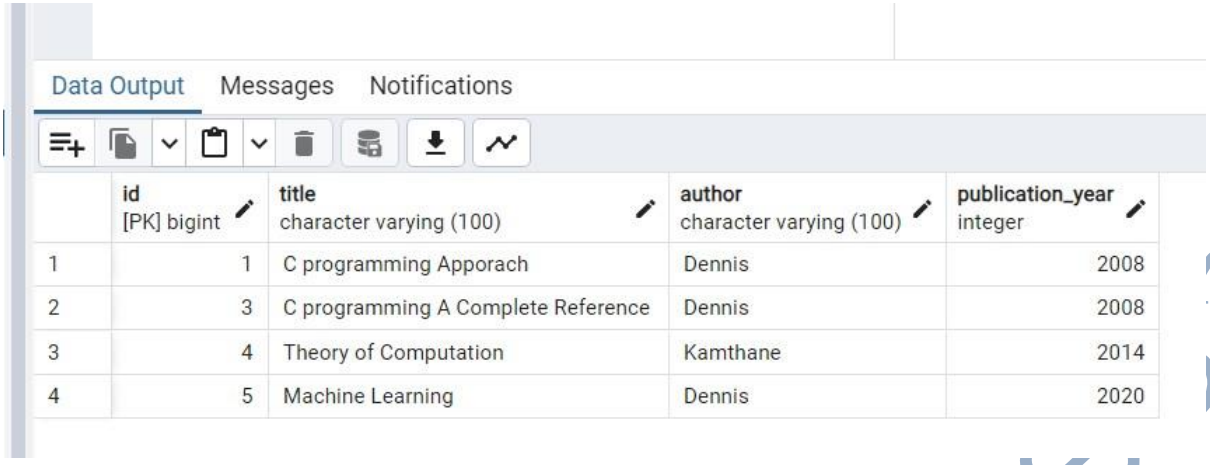
Submit



Book List

- Theory of Computation by Kamthane (2014)
- C programming Apporach by Dennis (2008)
- Machine Learning by Dennis (2020)
- C programming A Complete Reference by Dennis (2008)

All the changes will be reflected to the database (PostgreSQL)



Data Output | Messages | Notifications

	id [PK] bigint	title character varying (100)	author character varying (100)	publication_year integer
1	1	C programming Apporach	Dennis	2008
2	3	C programming A Complete Reference	Dennis	2008
3	4	Theory of Computation	Kamthane	2014
4	5	Machine Learning	Dennis	2020

MIME types (IANA media types)

A media type (also known as a Multipurpose Internet Mail Extensions or MIME type) indicates the nature and format of a document, file, or assortment of bytes. MIME types are defined and standardized in IETF's RFC 6838

The Internet Assigned Numbers Authority (IANA) is responsible for all official MIME types

Structure of a MIME type

A MIME type most commonly consists of just two parts: a *type* and a *subtype*, separated by a slash (/) — with no whitespace between:

type/subtype

The *type* represents the general category into which the data type falls, such as video or text.

The *subtype* identifies the exact kind of data of the specified type the MIME type represents. For example, for the MIME type text, the subtype might be plain (plain text), html (HTML source code), or calendar (for iCalendar/.ics) files.

Types

There are two classes of type:

1.discrete - iscrete types are types which represent a single file or medium, such as a single text or music file, or a single video.

The discrete types currently registered with the IANA are:

Application - application/pdf, application/pkcs8, and application/zip

Audio - Audio or music data. Examples include audio/mpeg, audio/vorbis

Font Font/typeface data. Common examples include font/woff, font/ttf, and font/otf.

Image Image or graphical data including both bitmap and vector still images as well as animated versions of still image formats such as animated GIF or APNG. Common examples are image/jpeg, image/png, and image/svg+xml.

Model Model data for a 3D object or scene. Examples include model/3mf and model/vrml

Text-only data including any human-readable content, source code, or textual data such as comma-separated value (CSV) formatted data. Examples include: text/plain, text/csv, and text/html.

Video Video data or files, such as MP4 movies (video/mp4)

2. Multipart - A multipart type represents a document that's comprised of multiple component parts, each of which may have its own individual MIME type; or, a multipart type may encapsulate multiple files being sent together in one transaction. For example, multipart MIME types are used when attaching multiple files to an email.

Message - message/rfc822 (for forwarded or replied-to message quoting) and message/partial to allow breaking a large message into smaller ones automatically to be reassembled by the recipient.

Multipart - Data that consists of multiple components which may individually have different MIME types. Examples include multipart/form-data (for data produced using the FormData API)

Producing CSV

CSV is a simple data format usually used by spreadsheet software. It's basically a series of table rows, with each cell in the row separated by a comma (CSV stands for comma-separated values). For example, here's some data on "unruly" airline passengers in CSV format:

views.py

```
import csv

from django.http import HttpResponse



# Number of unruly passengers each year 1995 - 2005. In a real application

# this would likely come from a database or some other back-end data store.

UNRULY_PASSENGERS = [146, 184, 235, 200, 226, 251, 299, 273, 281, 304, 203]
```

```
def unruly_passengers_csv(request):

    response = HttpResponse(content_type='text/csv')

    response['Content-Disposition'] = 'attachment; filename=unruly.csv'

    writer = csv.writer(response)

    writer.writerow(['Year', 'Unruly Airline Passengers'])

    for (year, num) in zip(range(1995, 2006), UNRULY_PASSENGERS):

        writer.writerow([year, num])

    return response
```

urls.py (Project File)

```
from django.contrib import admin

from django.urls import path

from app_csv import views as va

urlpatterns = [

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

    path('csv/',va.unruly_passengers_csv),

]
```

Run the project and check in http://127.0.0.1:8000/csv/ , your CSV file will be downloaded.

	A	B	C
	Year	Unruly Airline Passengers	
2	1995	146	
3	1996	184	
4	1997	235	
5	1998	200	
6	1999	226	
7	2000	251	
8	2001	299	
9	2002	273	
10	2003	281	
11	2004	304	
12	2005	203	
13			

Dynamic CSV using Database

views.py

```
# views.py

import csv
from django.http import HttpResponse
from .models import Student  # Import the Student model from your models.py file

def export_students_csv(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="students.csv"'

    # Retrieve data from the Student model
    students = Student.objects.all()

    # Write data to CSV file
    writer = csv.writer(response)
    writer.writerow(['ID', 'Name', 'Description']) # Column headers
    for student in students:
        writer.writerow([student.id, student.name, student.des]) # Assuming 'des' is the field for 'description' in your Student
```

```
model

    return response
```

models.py

```
# models.py



from django.db import models




class Student(models.Model):

    id = models.AutoField(primary_key=True)

    name = models.CharField(max_length=100)

    des = models.TextField()

    class Meta:

        db_table = 'prod_student'

    def __str__(self):

        return self.name
```

urls.py (Project File)

```
from django.contrib import admin

from django.urls import path

from app_csv import views as va

from db_app_csv import views as vb

urlpatterns = [

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

    path('csv/',va.unruly_passengers_csv),

    path('dy_csv/',vb.export_students_csv),

]
```

Run the project and check in http://127.0.0.1:8000/dy_csv/ , your CSV file will be downloaded.

	A	B	C	D	E
1	ID	Name	Description		
2	1	Vishnu	CSE		
3	2	Grish	ECE		
4	3	Harish	Harish is in Sport Club		
5	4	Lesa	Lesa is talented person		
6	5	Gargon	Gargon very nice place to		
7					
8					

GENERATING PDF

Portable Document Format (PDF) is a format developed by Adobe that's used to represent printable documents, complete with pixel-perfect formatting, embedded fonts, and 2D vector graphics. You can think of a PDF document as the digital equivalent of a printed document; indeed, PDFs are often used in distributing documents for the purpose of printing them. You can easily generate PDFs with Python and Django thanks to the excellent open source ReportLab library (http://www.reportlab.org/rl_toolkit.html).

The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes – say, for different users or different pieces of content.

For example, your humble authors used Django and ReportLab at KUSports.com to generate customized, printer-ready NCAA tournament brackets.

Installing ReportLab Before you do any PDF generation, however, you'll need to install ReportLab.

```
PS C:\Users\VISHNU\project\pr_pdf> pip install reportlab
```

views.py

```
from reportlab.pdfgen import canvas

from django.http import HttpResponse

from .models import Student

def getpdf(request):

    # Fetch all student objects from the database

    students = Student.objects.all()
```

```
response = HttpResponse(content_type='application/pdf')

response['Content-Disposition'] = 'attachment; filename="students.pdf"'

# Create a canvas object

p = canvas.Canvas(response)

# Set font and font size

p.setFont("Times-Roman", 12)

# Set initial y position

y_position = 700

# Write student information to the PDF

for student in students:

    # Write student ID, name, and description to the PDF

    p.drawString(100, y_position, f"ID: {student.id}")

    p.drawString(150, y_position, f"Name: {student.name}")

    p.drawString(300, y_position, f"Description: {student.des}")

    y_position -= 20

   # If the y_position is too low, create a new page

    if y_position < 50:
```

```
        p.showPage()

        p.setFont("Times-Roman", 12)

        y_position = 700

    # Save the PDF

    p.save()

    return response
```

model.py

```
# models.py

from django.db import models

class Student(models.Model):

    id = models.AutoField(primary_key=True)

    name = models.CharField(max_length=100)

    des = models.TextField()

    class Meta:

        db_table = 'prod_student'

    def __str_(self):
```

```
        return self.name
```

urls.py (Project File)

```
from django.contrib import admin

from django.urls import path

from db_app_pdf import views as vb

urlpatterns = [

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

    path('pdf',vb.getpdf),

]
```

Configure the app and database in the settings.py file as below.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'db_app_pdf',
]
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'django_db', #give the database name
        'USER': 'postgres',
        'PASSWORD': 'vishnu',
        'HOST': 'localhost',  # or the hostname where your MySQL server is running
    }
}
```

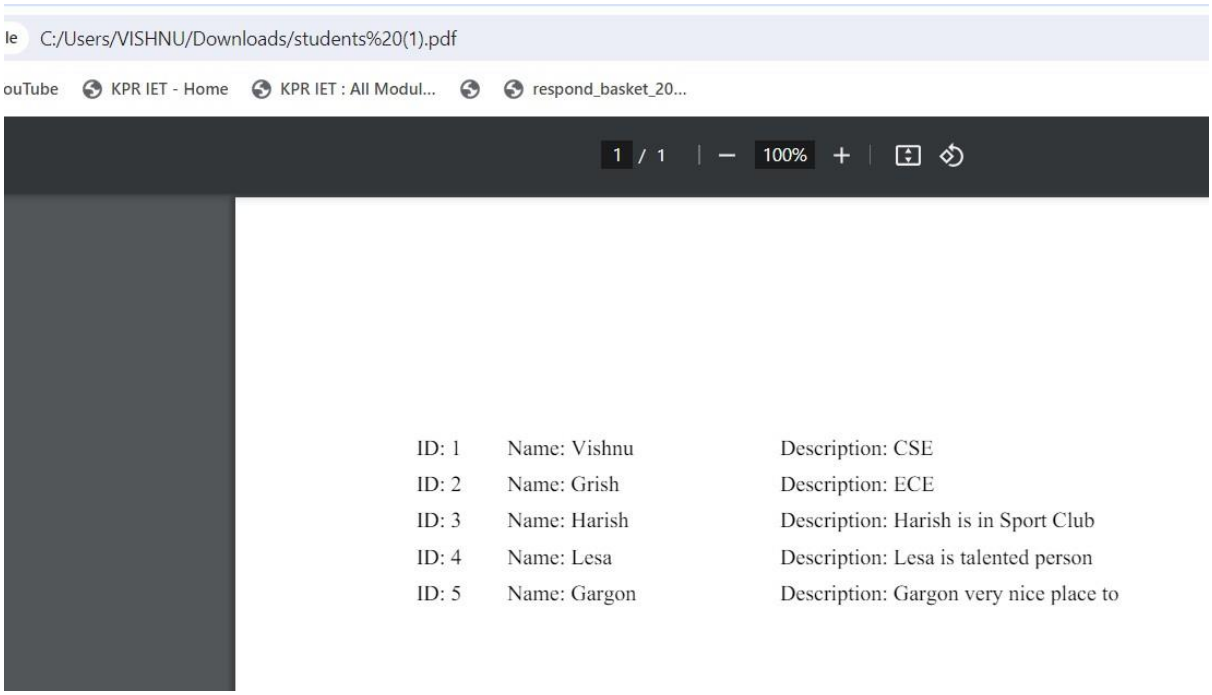
(module) backends

Run the below command

```
PS C:\Users\VISHNU\project\pr_pdf> python .\manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 23, 2024 - 15:56:00
Django version 4.2.11, using settings 'pr_pdf.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Now click on the above server address and give as "http://127.0.0.1:8000/pdf", automatically your PDf will be generated

It will generate the **student.pdf** file as below



Other Possibilities

There's a whole host of other types of content you can generate in Python. Here are a few more ideas and some

pointers to libraries you could use to implement them:

ZIP files: Python's standard library ships with the zipfile module, which can both read and write compressed ZIP files. You could use it to provide on-demand archives of a bunch of files, or perhaps compress large documents when requested. You could similarly produce TAR files using the standard library's tarfile module.

Dynamic images: The Python Imaging Library (PIL; http://www.pythonware.com/products/pil/) is a fantastic toolkit for producing images (PNG, JPEG, GIF, and a whole lot more). You could use it to automatically scale down images into thumbnails, composite multiple images into a single frame, or even do Web-based image processing.

Plots and charts: There are a number of powerful Python plotting and charting libraries you could use to produce on-demand maps, charts, plots, and graphs. We can't possibly list them all, so here are a couple of the highlights:

matplotlib (http://matplotlib.sourceforge.net/) can be used to produce the type of high-quality plots usually generated with MatLab or Mathematica.

pygraphviz (http://networkx.lanl.gov/pygraphviz/), an interface to the Graphviz graph layout toolkit (http://graphviz.org/), can be used for generating structured diagrams of graphs and networks.

In general, any Python library capable of writing to a file can be hooked into Django. The possibilities are immense.

Now that we've looked at the basics of generating non-HTML content, let's step up a level of abstraction. Django ships with some pretty nifty built-in tools for generating some common types of non-HTML content.

The Syndication Feed Framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy.

What's RSS? What's Atom?

RSS and Atom are both XML-based formats you can use to provide automatically updating "feeds" of your site's content. Read more about RSS at http://www.whatisrss.com/, and get information on Atom at http://www.atomenabled.org/.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

The high-level feed-generating framework is a view that's hooked to /feeds/ by convention. Django uses the remainder of the URL (everything after /feeds/) to determine which feed to return.To create a feed, you'll write a Feed class and point to it in your URLconf.

Initialization

To activate syndication feeds on your Django site, add this URLconf:

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
    {'feed_dict': feeds}
),
```

This line tells Django to use the RSS framework to handle all URLs starting with "feeds/". (You can change that "feeds/" prefix to fit your own needs.)

This URLconf line has an extra argument: {'feed_dict': feeds}. Use this extra argument to pass the syndication framework the feeds that should be published under that URL.

Django comes with a high-level syndication-feed-generating framework for creating RSS and **Atom** feeds.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a web context, or in some other lower-level way.

The high-level framework¶

Overview¶

The high-level feed-generating framework is supplied by the **Feed** class. To create a feed, write a **Feed** class and point to an instance of it in your URLconf.

Feed classes

A **Feed** class is a Python class that represents a syndication feed. A feed can be simple (e.g., a "site news" feed, or a basic feed displaying the latest entries of a blog) or more complex (e.g., a feed displaying all the blog entries in a particular category, where the category is variable).

Feed classes subclass **django.contrib.syndication.views.Feed**. They can live anywhere in your codebase.

Instances of **Feed** classes are views which can be used in your URLconf.

A simple example

This simple example, taken from a hypothetical police beat news site describes a feed of the latest five news items:

Create a new file feeds.py in you app

```
from django.contrib.syndication.views import Feed

from django.urls import reverse

from datetime import datetime



class LatestEntriesFeed(Feed):

    title = "Police beat site news"

    link = "/sitenews/"

    description = "Updates on changes and additions to police beat central."



    def items(self):
```

```
    # Define your feed items here
    return [
        {
            'title': 'News Item 1',
            'link': 'http://127.0.0.1:8000/news/News-Item-1/',
            'description': 'Description of News Item 1',
            'pub_date': 'Tue, 23 Apr 2024 00:00:00 +0000',
        },
        {
            'title': 'News Item 2',
            'link': 'http://127.0.0.1:8000/news/News-Item-2/',
            'description': 'Description of News Item 2',
            'pub_date': 'Mon, 22 Apr 2024 00:00:00 +0000',
        }
    ]



def item_title(self, item):
    return item['title']


def item_description(self, item):
    return item['description']


def item_pubdate(self, item):
```

```
        return item['pub_date']



    def item_link(self, item):
        # Since you're using dictionary-based items, you need to define the link for each item
        # For example, you can return a static URL or create dynamic URLs based on the item data
        return '/news/{}/'.format(item['title'].replace(' ', '-'))
```

urls.py (Main Project File)

```
from django.urls import path
from sync_feed_1.feeds import LatestEntriesFeed
urlpatterns = [
    # other URL patterns
    path('latest/feed/', LatestEntriesFeed(), name='latest_feed'),
]
```

```
PS C:\Users\VISHNU\project\sync_feed> python .\manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

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

And go the browser and give the link http://127.0.0.1:8000/latest/feed/

- The Feed class subclasses **django.contrib.syndication.views.Feed**.
- **title**, **link** and **description** correspond to the standard RSS **<title>**, **<link>** and **<description>** elements, respectively.
- **items()** is, a method that returns a list of objects that should be included in the feed as **<item>** elements. Although this example returns **NewsItem** objects using Django's object-relational mapper, **items()** doesn't have to return model instances. Although you get a few bits of functionality "for free" by using Django models, **items()** can return any type of object you want.

- If you're creating an Atom feed, rather than an RSS feed, set the **subtitle** attribute instead of the **description** attribute. See Publishing Atom and RSS feeds in tandem, later, for an example.

One thing is left to do. In an RSS feed, each **<item>** has a **<title>**, **<link>** and **<description>**. We need to tell the framework what data to put into those elements.

- For the contents of **<title>** and **<description>**, Django tries calling the methods **item_title()** and **item_description()** on the **Feed** class. They are passed a single parameter, **item**, which is the object itself. These are optional; by default, the string representation of the object is used for both.
  If you want to do any special formatting for either the title or description, Django templates can be used instead. Their paths can be specified with the **title_template** and **description_template** attributes on the **Feed** class. The templates are rendered for each item and are passed two template context variables:
  - **{{ obj }}** – The current object (one of whichever objects you returned in **items()**).
  - **{{ site }}** – A **django.contrib.sites.models.Site** object representing the current site. This is useful for **{{ site.domain }}** or **{{ site.name }}**. If you do *not* have the Django sites framework installed, this will be set to a **RequestSite** object. See the RequestSite section of the sites framework documentation for more.

See a complex example below that uses a description template.

Feed.get_context_data(*kwargs*)**¶

There is also a way to pass additional information to title and description templates, if you need to supply more than the two variables mentioned before. You can provide your implementation of **get_context_data** method in your **Feed** subclass. For example:

```
from mysite.models import Article

from django.contrib.syndication.views import Feed

class ArticlesFeed(Feed):

    title = "My articles"

    description_template = "feeds/articles.html"



    def items(self):

        return Article.objects.order_by("-pub_date")[:5]



    def get_context_data(self, **kwargs):

        context = super().get_context_data(**kwargs)
```

```
    context["foo"] = "bar"

    return context
```

And the template:

```
Something about {{ foo }}: {{ obj.description }}
```

- This method will be called once per each item in the list returned by **items()** with the following keyword arguments:
  - **item**: the current item. For backward compatibility reasons, the name of this context variable is **{{ obj }}**.
  - **obj**: the object returned by **get_object()**. By default this is not exposed to the templates to avoid confusion with **{{ obj }}** (see above), but you can use it in your implementation of **get_context_data()**.
  - **site**: current site as described above.
  - **request**: current request.
- The behavior of **get_context_data()** mimics that of generic views - you're supposed to call **super()** to retrieve context data from parent class, add your data and return the modified dictionary.
- To specify the contents of **<link>**, you have two options. For each item in **items()**, Django first tries calling the **item_link()** method on the **Feed** class. In a similar way to the title and description, it is passed it a single parameter, **item**. If that method doesn't exist, Django tries executing a **get_absolute_url()** method on that object. Both **get_absolute_url()** and **item_link()** should return the item's URL as a normal Python string. As with **get_absolute_url()**, the result of **item_link()** will be included directly in the URL, so you are responsible for doing all necessary URL quoting and conversion to ASCII inside the method itself.

A More Complex Feed

The framework also supports more complex feeds, via parameters.

For example, say your blog offers an RSS feed for every distinct "tag" you've used to categorize your entries. It would be silly to create a separate Feed class for each tag; that would violate the Don't Repeat Yourself (DRY) principle and would couple data to programming logic.

Instead, the syndication framework lets you make generic feeds that return items based on information in the feed's URL.

Your tag-specific feeds could use URLs like this:

http://example.com/feeds/tags/python/: Returns recent entries tagged with "python"

http://example.com/feeds/tags/cats/: Returns recent entries tagged with "cats"

The slug here is "tags". The syndication framework sees the extra URL bits after the slug – 'python' and 'cats'

– and gives you a hook to tell it what those URL bits mean and how they should influence which items get published in the feed.

An example makes this clear. Here's the code for these tag-specific feeds:

```
from django.core.exceptions import ObjectDoesNotExist from mysite.blog.models import Entry, Tag

 class TagFeed(Feed):

def get_object(self, bits):

# In case of "/feeds/tags/cats/dogs/mice/", or other such

# clutter, check that bits has only one member. if len(bits) != 1:

raise ObjectDoesNotExist

return Tag.objects.get(tag=bits[0])


def title(self, obj):

return "My Blog: Entries tagged with %s" % obj.tag


def link(self, obj):

return obj.get_absolute_url()


def description(self, obj):

return "Entries tagged with %s" % obj.tag def items(self, obj):

entries = Entry.objects.filter(tags  id  exact=obj.id)

return entries.order_by('-pub_date')[:30]
```

Here's the basic algorithm of the RSS framework, given this class and a request to the URL

/feeds/tags/python/:

1. The framework gets the URL /feeds/tags/python/ and notices there's an extra bit of URL after the slug. It splits that remaining string by the slash character ("/") and calls the Feed class's get_object() method, passing it the bits.

In this case, bits is ['python']. For a request to /feeds/tags/python/django/, bits would be

['python', 'django'].

2. get_object() is responsible for retrieving the given Tag object, from the given bits.

In this case, it uses the Django database API to retrieve the Tag. Note that get_object() should raise django.core.exceptions.ObjectDoesNotExist if given invalid parameters. There's no try/except around the Tag.objects.get() call, because it's not necessary. That function raises Tag.DoesNotExist on failure, and Tag.DoesNotExist is a subclass of ObjectDoesNotExist. Raising ObjectDoesNotExist in get_object() tells Django to produce a 404 error for that request.

3. To generate the feed's <title>, <link>, and <description>, Django uses the title(), link(), and description() methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings or methods. For each of title, link, and description, Django follows this algorithm:

1. It tries to call a method, passing the obj argument, where obj is the object returned by get_object().

2. Failing that, it tries to call a method with no arguments.

3. Failing that, it uses the class attribute.

4. Finally, note that items() in this example also takes the obj argument. The algorithm for items is the same as described in the previous step – first, it tries items(obj), then items(), and then finally an items class attribute (which should be a list).

Full documentation of all the methods and attributes of the Feed classes is always available from the official

Django documentation (http://docs.djangoproject.com/en/dev/ref/contrib/syndication/).

Specifying the Type of Feed

By default, the syndication framework produces RSS 2.0. To change that, add a feed_type attribute to your Feed

class:

from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):

feed_type = Atom1Feed

Note that you set feed_type to a class object, not an instance. Currently available feed types are shown in Table 11-1.

Table 11-1. Feed Types

Feed Class	Format
django.utils.feedgenerator.Rss201rev2Feed	RSS 2.01 (default)
django.utils.feedgenerator.RssUserland091Feed	RSS 0.91
django.utils.feedgenerator.Atom1Feed	Atom 1.0

Enclosures

To specify enclosures (i.e., media resources associated with a feed item such as MP3 podcast feeds), use the item_enclosure_url, item_enclosure_length, and item_enclosure_mime_type hooks, for example:

from myproject.models import Song class MyFeedWithEnclosures(Feed):

title = "Example feed with enclosures"

link = "/feeds/example-with-enclosures/"

```
def items(self):

return Song.objects.all()[:30]


def item_enclosure_url(self, item):

return item.song_url


def item_enclosure_length(self, item):

return item.song_length item_enclosure_mime_type = "audio/mpeg"
```

This assumes, of course, that you've created a Song object with song_url and song_length (i.e., the size in bytes) fields.

Language

Feeds created by the syndication framework automatically include the appropriate <language> tag (RSS 2.0) or

xml:lang attribute (Atom). This comes directly from your LANGUAGE_CODE setting.

URLs

The link method/attribute can return either an absolute URL (e.g., "/blog/") or a URL with the fully qualified domain and protocol (e.g., "http://www.example.com/blog/"). If link doesn't return the domain, the syndication framework will insert the domain of the current site, according to your SITE_ID setting. (See Chapter 16 for more on SITE_ID and the sites framework.)

Atom feeds require a <link rel="self"> that defines the feed's current location. The syndication framework populates this automatically.

Publishing Atom and RSS Feeds in Tandem

Some developers like to make available both Atom and RSS versions of their feeds. That's easy to do with Django: just create a subclass of your feed class and set the feed_type to something different. Then update

your URLconf to add the extra versions. Here's a full example:

```
from django.contrib.syndication.feeds import Feed from django.utils.feedgenerator import Atom1Feed from mysite.blog.models import Entry

class RssLatestEntries(Feed):

title = "My Blog" link = "/archive/"

description = "The latest news about stuff."

def items(self):

return Entry.objects.order_by('-pub_date')[:5]

class AtomLatestEntries(RssLatestEntries):

feed_type = Atom1Feed
```

And here's the accompanying URLconf:

```
from django.conf.urls.defaults import *

from myproject.feeds import RssLatestEntries, AtomLatestEntries

feeds = {

'rss': RssLatestEntries,

'atom': AtomLatestEntries,

}

urlpatterns = patterns('',

# ...

(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',

  {'feed_dict': feeds}),
```

```
# ...

)
```

The Sitemap Framework

A sitemap is an XML file on your Web site that tells search engine indexers how frequently your pages change and how “important” certain pages are in relation to other pages on your site. This information helps search engines index your site.

For example, here’s a piece of the sitemap for Django’s Web site (http://www.djangoproject.com/sitemap.xml):

```
<?xml version="1.0" encoding="UTF-8"?>

<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">

<url>

<loc>http://www.djangoproject.com/documentation/</loc>

<changefreq>weekly</changefreq>

<priority>0.5</priority>

</url>

<url>

<loc>http://www.djangoproject.com/documentation/0_90/</loc>

<changefreq>never</changefreq>

<priority>0.1</priority>

</url>

...

</urlset>
```

For more on sitemaps, see http://www.sitemaps.org/.

The Django sitemap framework automates the creation of this XML file by letting you express this information in

Python code. To create a sitemap, you just need to write a Sitemap class and point to it in your URLconf.

Installation

To install the sitemap application, follow these steps:

1. Add 'django.contrib.sitemaps' to your INSTALLED_APPS setting.

2. Make sure 'django.template.loaders.app_directories.load_template_source' is in your TEMPLATE_LOADERS

setting. It's in there by default, so you'll need to change this only if you've changed that setting.

3. Make sure you've installed the sites framework (see Chapter 16).

Note

The sitemap application doesn't install any database tables. The only reason it needs to go into

INSTALLED_APPS is so the load_template_source template loader can find the default templates.

Initialization

To activate sitemap generation on your Django site, add this line to your URLconf:

(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})

This line tells Django to build a sitemap when a client accesses /sitemap.xml. Note that the dot character in sitemap.xml is escaped with a backslash, because dots have a special meaning in regular expressions.

The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if sitemap.xml lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at /content/sitemap.xml, it may only reference URLs that begin with /content/.

The sitemap view takes an extra, required argument: {'sitemaps': sitemaps}. sitemaps should be a dictionary that maps a short section label (e.g., blog or news) to its Sitemap class (e.g., BlogSitemap or NewsSitemap). It may also map to an instance of a Sitemap class (e.g., BlogSitemap(some_var)).

Sitemap Classes

A Sitemap class is a simple Python class that represents a “section” of entries in your sitemap. For example, one

Sitemap class could represent all the entries of your weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one sitemap.xml, but it’s also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section (as described shortly).

Sitemap classes must subclass django.contrib.sitemaps.Sitemap. They can live anywhere in your code tree.

For example, let’s assume you have a blog system, with an Entry model, and you want your sitemap to include all the links to your individual blog entries. Here’s how your Sitemap class might look:

from django.contrib.sitemaps import Sitemap from mysite.blog.models import Entry

class BlogSitemap(Sitemap): changefreq = "never" priority = 0.5

def items(self):

return Entry.objects.filter(is_draft=False)

def lastmod(self, obj):

return obj.pub_date

Declaring a Sitemap should look very similar to declaring a Feed. That’s by design.

Like Feed classes, Sitemap members can be either methods or attributes. See the steps in the earlier “A

Complex Example" section for more about how this works.

A Sitemap class can define the following methods/attributes:

items (required): Provides list of objects. The framework doesn't care what type of objects they are; all that matters is that these objects get passed to the location(), lastmod(), changefreq(), and priority()

methods.

location (optional): Gives the absolute URL for a given object. Here, "absolute URL" means a URL that doesn't include the protocol or domain. Here are some examples:

Good: '/foo/bar/'

Bad: 'example.com/foo/bar/'

Bad: 'http://example.com/foo/bar/'

If location isn't provided, the framework will call the get_absolute_url() method on each object as returned by items().

lastmod (optional): The object's "last modification" date, as a Python datetime object.

changefreq (optional): How often the object changes. Possible values (as given by the Sitemaps

specification) are as follows:

'always'

'hourly'

'daily'

'weekly'

'monthly'

'yearly'

'never'

priority (optional): A suggested indexing priority between 0.0 and 1.0. The default priority of a page is

0.5; see the http://sitemaps.org/ documentation for more about how priority works.

Shortcuts

The sitemap framework provides a couple convenience classes for common cases. These are described in the sections that follow.

FlatPageSitemap

The django.contrib.sitemaps.FlatPageSitemap class looks at all flat pages defined for the current site and creates an entry in the sitemap. These entries include only the location attribute – not lastmod, changefreq, or priority.

GenericSitemap

The GenericSitemap class works with any generic views (see Chapter 11) you already have.

To use it, create an instance, passing in the same info_dict you pass to the generic views. The only requirement is that the dictionary have a queryset entry. It may also have a date_field entry that specifies a date field for objects retrieved from the queryset. This will be used for the lastmod attribute in the generated sitemap. You may also pass priority and changefreq keyword arguments to the GenericSitemap constructor to specify these attributes for all URLs.

Here's an example of a URLconf using both FlatPageSitemap and GenericSiteMap (with the hypothetical Entry

object from earlier):

```
from django.conf.urls.defaults import *
```

```
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap from mysite.blog.models import Entry

 info_dict = {

'queryset': Entry.objects.all(),

'date_field': 'pub_date',

}

 sitemaps = {

'flatpages': FlatPageSitemap,

'blog': GenericSitemap(info_dict, priority=0.6),

}

urlpatterns = patterns('',

# some generic view using info_dict

# ...

 # the sitemap

(r'^sitemap\.xml$',

'django.contrib.sitemaps.views.sitemap',

{'sitemaps': sitemaps})

)
```

Creating a Sitemap Index

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your sitemaps dictionary. The only differences in usage are as follows:

You use two views in your URLconf: django.contrib.sitemaps.views.index and

django.contrib.sitemaps.views.sitemap.

The django.contrib.sitemaps.views.sitemap view should take a section keyword argument. Here is what the relevant URLconf lines would look like for the previous example:

(r'^sitemap.xml$',

'django.contrib.sitemaps.views.index',

{'sitemaps': sitemaps}),

(r'^sitemap-(?P<section>.+).xml$',

'django.contrib.sitemaps.views.sitemap',

{'sitemaps': sitemaps})

This will automatically generate a sitemap.xml file that references both sitemap-flatpages.xml and

sitemap-blog.xml. The Sitemap classes and the sitemaps dictionary don't change at all.

Pinging Google

You may want to "ping" Google when your sitemap changes, to let it know to reindex your site. The framework provides a function to do just that: django.contrib.sitemaps.ping_google().

ping_google() takes an optional argument, sitemap_url, which should be the absolute URL of your site's sitemap (e.g., '/sitemap.xml'). If this argument isn't provided, ping_google() will attempt to figure out your sitemap by performing a reverse lookup on your URLconf.

ping_google() raises the exception django.contrib.sitemaps.SitemapNotFound if it cannot determine your sitemap URL.

One useful way to call ping_google() is from a model's save() method:

from django.contrib.sitemaps import ping_google class Entry(models.Model):

#def save(self, *args, **kwargs):

```
super(Entry, self).save(*args, **kwargs)

try:

ping_google()

except Exception:

# Bare 'except' because we could get a variety

# of HTTP-related exceptions. pass
```

A more efficient solution, however, would be to call ping_google() from a cron script or some other scheduled task. The function makes an HTTP request to Google's servers, so you may not want to introduce that network

overhead each time you call save().

Finally, if 'django.contrib.sitemaps' is in your INSTALLED_APPS, then your manage.py will include a new command, ping_google. This is useful for command-line access to pinging. For example:

```
python manage.py ping_google /sitemap.xml
```