

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.

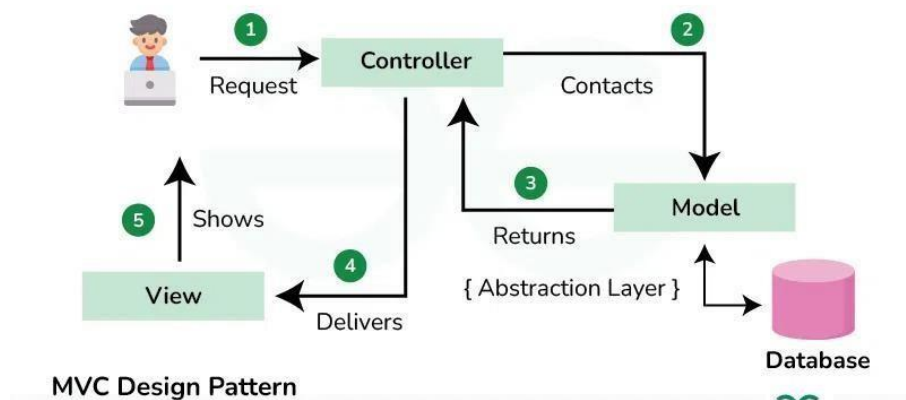
MVC Design Pattern

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.

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:

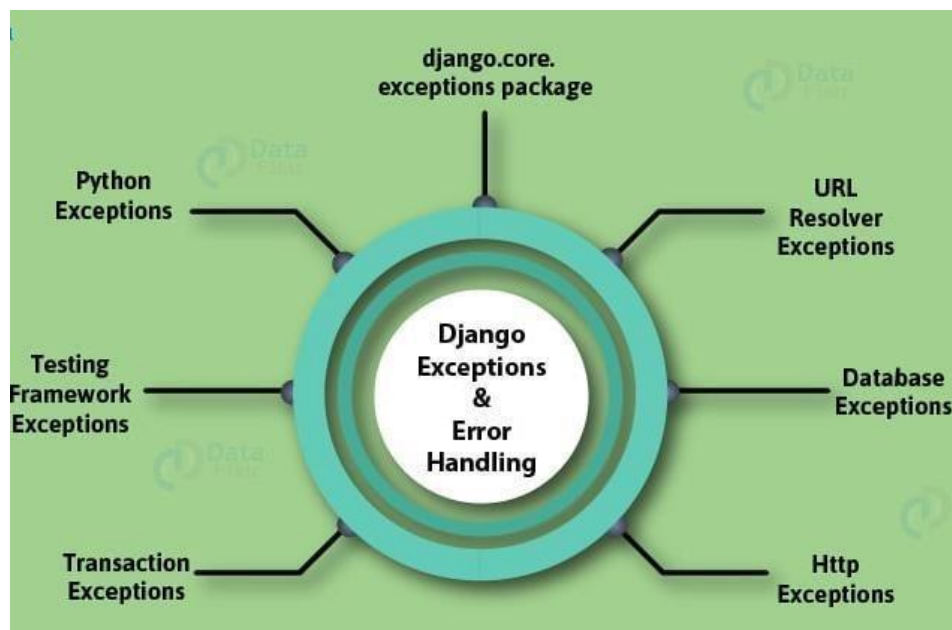
Loose coupling is a software design principle that promotes independence and modularity by minimizing the dependencies between different components of a system. In the context of Django, loose coupling is achieved through:

3. **Modular Structure:** URLconfs can be organized hierarchically, allowing you to include other
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.

A screenshot of a Windows PowerShell window titled "Select Windows PowerShell". The window has a black background with white text. The text shows a Python script being executed in a Django environment. The script imports exceptions from django.core, lists them, and prints them. The output shows a list of Django exceptions, including AppRegistryNotReady, DisallowedHost, DisallowedRedirect, EmptyResultSet, FieldDoesNotExist, FieldError, ImproperlyConfigured, MiddlewareNotUsed, MultipleObjectsReturned, NON_FIELD_ERRORS, ObjectDoesNotExist, PermissionDenied, RequestDataTooBig, SuspiciousFileOperation, SuspiciousMultipartForm, SuspiciousOperation, TooManyFieldsSent, ValidationError, and ViewDoesNotExist. Below these, there are several attributes like __builtins__, __cached__, __doc__, __file__, __loader__, __name__, __package__, and __spec__.

```
>>> from django.core import exceptions
>>> ls = list(dir(exceptions))
>>> for i in ls:
...     print(i)
...
AppRegistryNotReady
DisallowedHost
DisallowedRedirect
EmptyResultSet
FieldDoesNotExist
FieldError
ImproperlyConfigured
MiddlewareNotUsed
MultipleObjectsReturned
NON_FIELD_ERRORS
ObjectDoesNotExist
PermissionDenied
RequestDataTooBig
SuspiciousFileOperation
SuspiciousMultipartForm
SuspiciousOperation
TooManyFieldsSent
ValidationError
ViewDoesNotExist
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
__package__
__spec__
>>>
```

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.

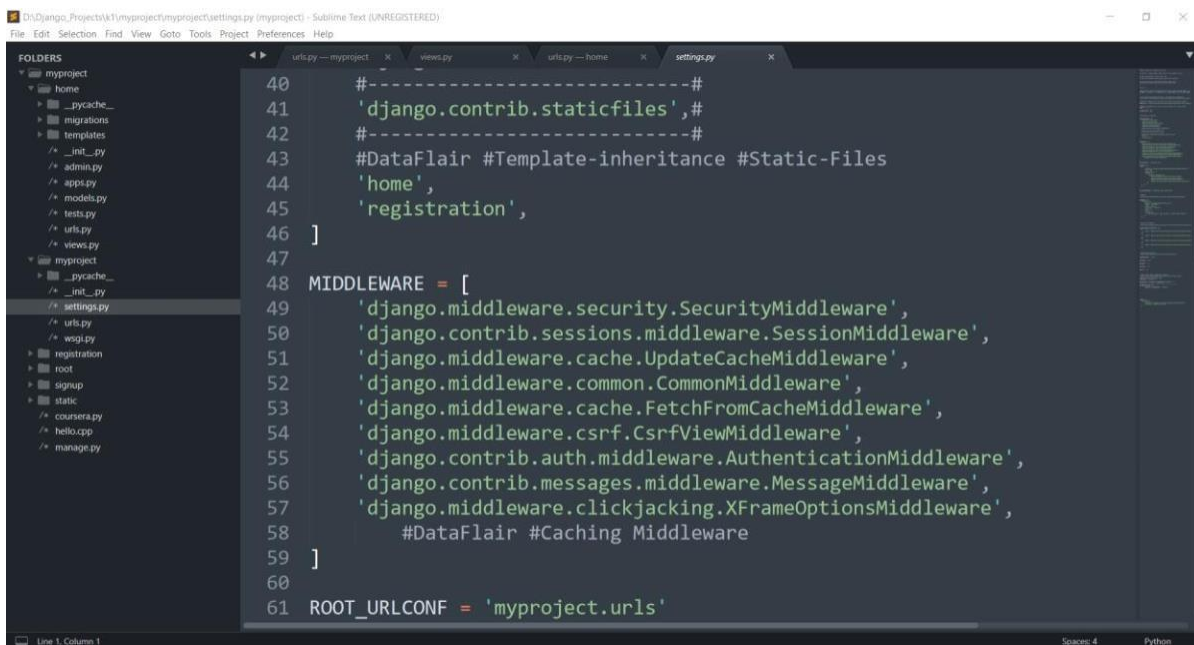
But we can also raise this error when urls-config can't get the view to load. It is a problem which occurs while using relative URL addressing.

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.



```
40 #-----#
41 'django.contrib.staticfiles',#
42 #-----#
43 #DataFlair #Template-Inheritance #Static-Files
44 'home',
45 'registration',
46 ]
47
48 MIDDLEWARE = [
49     'django.middleware.security.SecurityMiddleware',
50     'django.contrib.sessions.middleware.SessionMiddleware',
51     'django.middleware.cache.UpdateCacheMiddleware',
52     'django.middleware.common.CommonMiddleware',
53     'django.middleware.cache.FetchFromCacheMiddleware',
54     'django.middleware.csrf.CsrfViewMiddleware',
55     'django.contrib.auth.middleware.AuthenticationMiddleware',
56     'django.contrib.messages.middleware.MessageMiddleware',
57     'django.middleware.clickjacking.XFrameOptionsMiddleware',
58     #DataFlair #Caching Middleware
59 ]
60
61 ROOT_URLCONF = 'myproject.urls'
```

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:

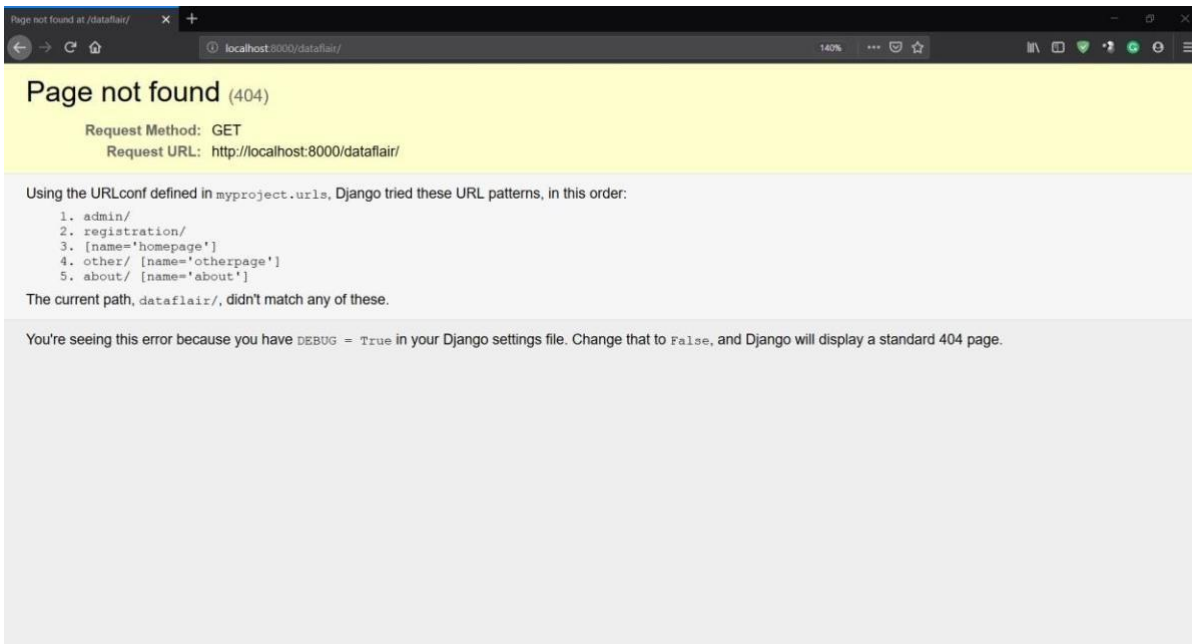
2.1. Resolver404

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 enough 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 CPythonWindows(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\pip\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("HTTPConnectionPool(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("HTTPConnectionPool(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("HTTPConnectionPool(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("HTTPConnectionPool(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
```

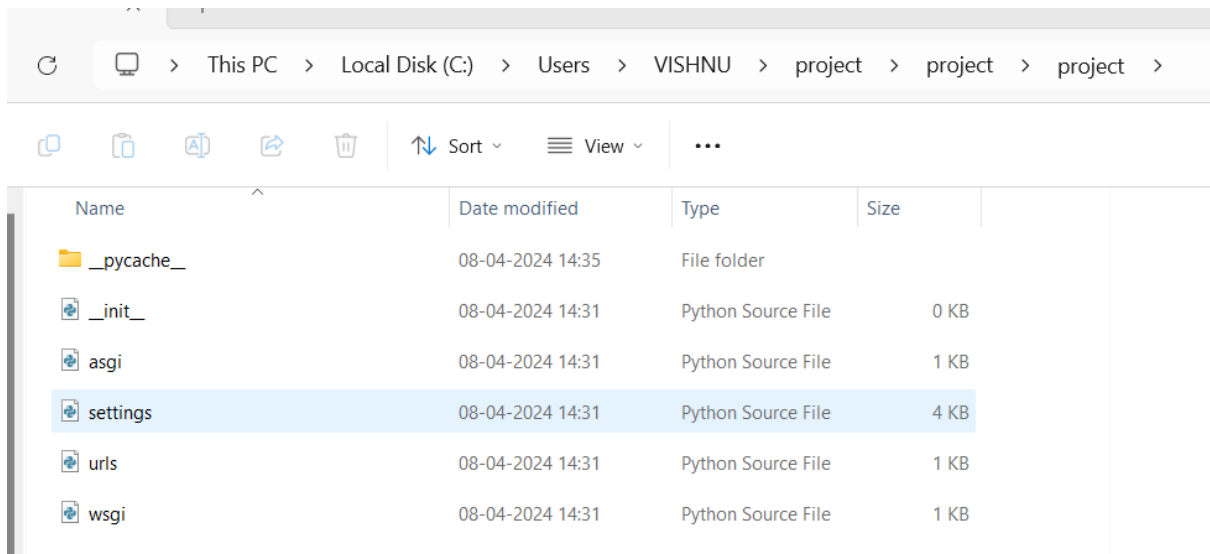
```
(first) C:\Users\VISHNU>django-admin --version
4.2.11
```

```
(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.

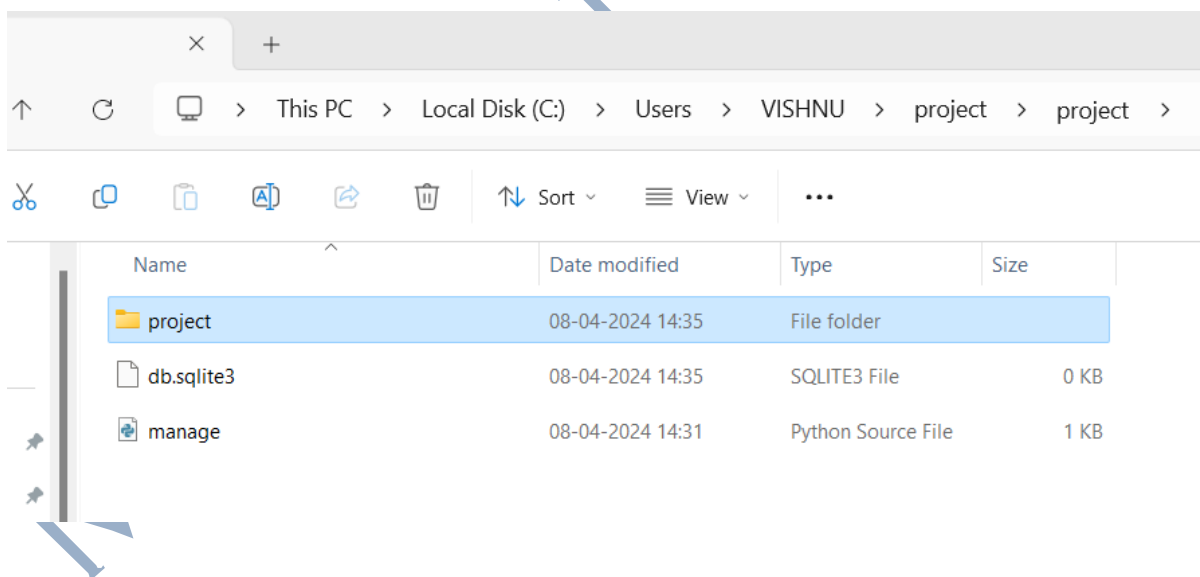


`__init__.py` - used to treat our folder (project) as a package

`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



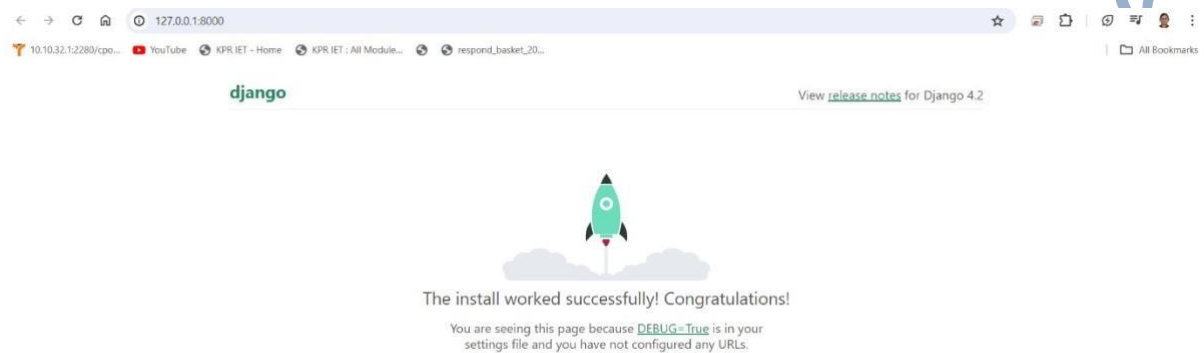
Use the below command to run the `manage.py` file

```
(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

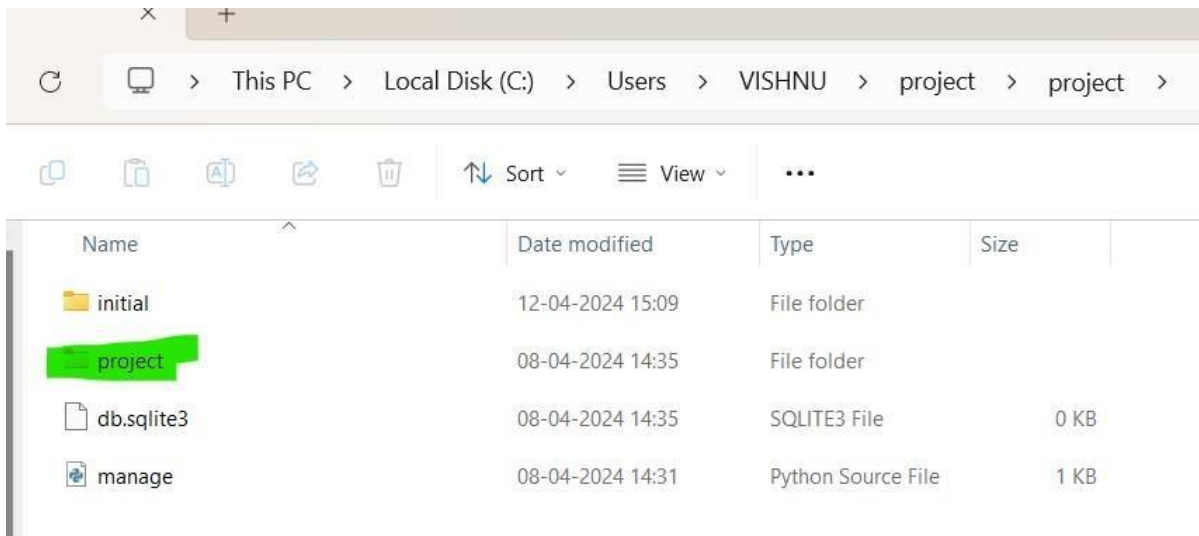


Building First App

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

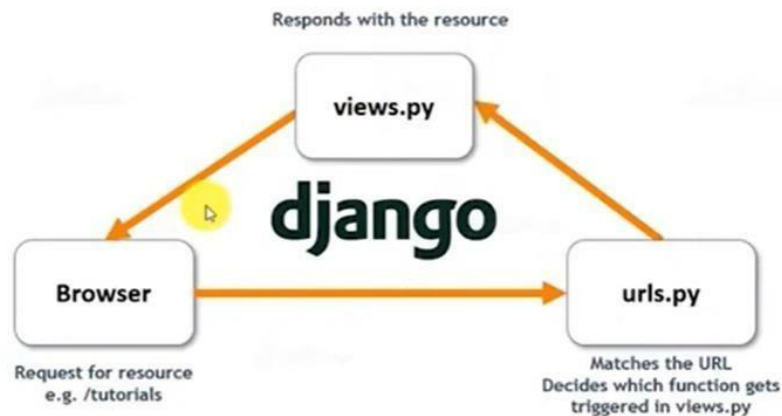
URL Mapping and views

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

URL Mapping and Views



Then Django responds back to the user with the resource



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

```

from django.http import HttpResponse

# Create your views here.

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 datetime 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 1 – 21CS62