Module-01

MVC based Web Designing

Web Framework

Django: High-level Python web framework for rapid development. Features include an ORM, URL routing, template engine, admin interface, and strong security. Best for large, database-driven applications.

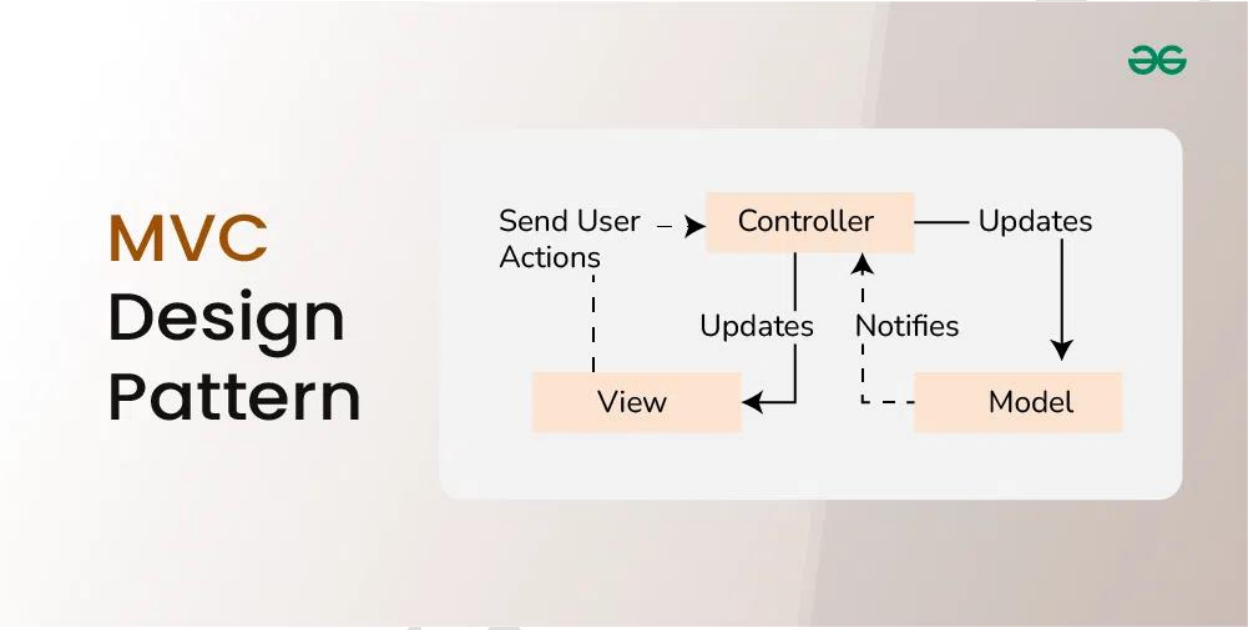
Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the model-template-view (MTV) architectural pattern, which is similar to the model-view-controller (MVC) pattern used in other frameworks.

Key features of Django include:

1. **ORM (Object-Relational Mapping):** Django provides a powerful ORM that allows you to interact with databases using Python code instead of raw SQL.
2. **URL Routing:** It uses a URL configuration file to map URLs to views, making it easy to create clean, SEO-friendly URLs.
3. **Template Engine:** Django has a template language that allows you to separate the design from Python code.
4. **Forms:** It offers a form framework that handles rendering forms, validating user input, and processing submitted data.
5. **Authentication and Authorization:** Django comes with a robust user authentication system out of the box.
6. **Admin Interface:** It automatically generates an admin interface for your models, which is great for managing your site's content.
7. **Security Features:** Django provides protection against common web vulnerabilities like XSS, CSRF, SQL injection, etc.

MVC Design Pattern

- The **MVC** design pattern is a software architecture pattern that separates an application into three main components: **Model, View, and Controller,** making it easier to manage and maintain the codebase.
- It also allows for the reusability of components and promotes a more modular approach to software development.



What is the 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.

How the MVC pattern works in general and how it can be related to Django in scope.

The MVC pattern is a software architecture pattern that separates data presentation from the logic of handling user interactions (in other words, saves you stress:), it has been around as a concept for a while, and has invariably seen an exponential growth in use since its inception. It has also been described as one of the best ways to create client-server applications, all of the best frameworks for web are all built around the MVC concept.

Here's a general overview of the MVC Concept;

Model: This handles your data representation, it serves as an interface to the data stored in the database itself, and also allows you to interact with your data without having to get perturbed with all the complexities of the underlying database.

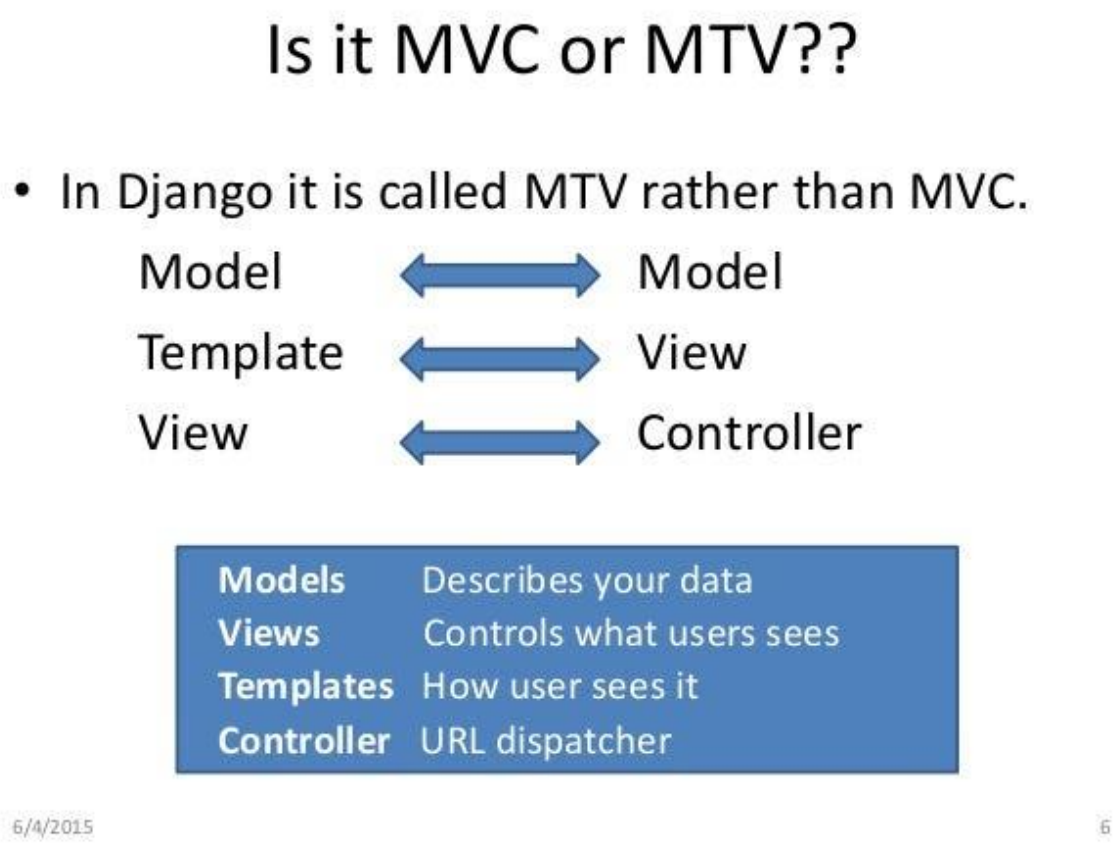
View: As the name implies, it represents what you see while on your browser for a web application or In the UI for a desktop application.

Controller: provides the logic to either handle presentation flow in the view or update the model's data i.e. it uses programmed logic to figure out what is pulled from the database through the model and passed to the view, also gets information from the user through the view and implements the given logic by either changing the view or updating the data via the model, To make it more simpler, see it as the engine room.

Now that we understand the general concept of the MVC, understanding how it is implemented in different frameworks can be another task as some frameworks (Django inclusive) like to implement this same functionality in another way making it a bit difficult understanding what actually happens at each layer.

How do we relate it to Our scope in Django?

Even with Django following the MVC pattern, it prefers to use it's own logic in the implementation, the framework considers handling the Controller part of the MVC itself and letting most of the good stuff happen in the Model-Template-View, this is why Django is mostly referred to as the MTV framework.



In the MTV pattern:

1. models.py (the database tables):

- Defines the database model for a book using Django's models. Model class.
- Contains a Book class with two fields: name (CharField) and pub_date (DateField).
- Represents the data layer of the application

```
from django.db import models
class Book(models.Model):
    name = models.CharField(max_length=50)
    pub_date = models.DateField()
```

2. views.py (the business logic):

- Implements the latest_books() function as the view.
- Retrieves the latest 10 books from the database and passes them to the template for rendering.
- Handles user requests and prepares data for display.

```
from django.shortcuts import render_to_response
from .models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})
```

3. urls.py (the URL configuration):

- Defines URL patterns and maps them to corresponding views.
- Specifies that the URL /latest/ should be handled by the latest_books() view.

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

urlpatterns = patterns('',
    (r'^latest/$', views.latest_books),
)
```

3. latest_books.html (the template):

- HTML template for rendering the latest books.
- Uses Django's template language to iterate over the book_list passed from the view and display book names in a list.

```
<html>
<head>
  <title>Books</title>
</head>
<body>
  <h1>Books</h1>
  <ul>
    {% for book in book_list %}
    <li>{{ book.name }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

This Django program follows the Model-View-Controller (MVC) design pattern:

- Model (models.py): Defines the database model.
- View (views.py): Handles business logic and prepares data for display.
- Controller (urls.py): Routes URLs to corresponding views.
- Template (latest_books.html): Defines the presentation layer and renders data to the user.

The components are loosely coupled, allowing for independent changes and enhancing maintainability and scalability of the application.

Django Evolution

1. **Origins:** Django was created by Adrian Holovaty and Simon Willison in the fall of 2003. They were web programmers at the Lawrence Journal-World newspaper in Lawrence, Kansas, USA.

2. **Development Environment:** The team worked in a fast-paced environment with tight journalism deadlines. They needed to build and maintain web applications quickly and efficiently.

3. **Organic Growth:** Django grew organically from the real-world needs of the development team. They initially built applications from scratch but soon realized the need for a framework to streamline their process.

4. **Framework Creation:** Following the typical path of many web developers, the team started refactoring their code to share common functionalities across applications. This led to the creation of a framework.

5. **Open-Source Release:** In the summer of 2005, after developing the framework to a usable state, the team, including Jacob Kaplan-Moss, decided to release it as open source software in July 2005.

6. **Name Origin:** The framework was named Django after the jazz guitarist Django Reinhardt.

7. **Current Status:** Today, Django is a well-established open-source project with a large community of users and contributors worldwide. Adrian Holovaty and Jacob Kaplan-Moss, two of the original developers, still provide guidance for its growth, but it's now a collaborative effort.

8. **Sweet Spot:** Django's origins in a news environment influenced its design, making it particularly well-suited for content sites with dynamic, database-driven information. However, it's effective for building a wide range of dynamic websites.

Version	Date	Description
0.90	16 Nov 2005	
0.91	11 Jan 2006	magic removal
0.96	23 Mar 2007	newforms, testing tools
1.0	3 Sep 2008	API stability, decoupled admin, unicode
1.1	29 Jul 2009	Aggregates, transaction based tests
1.2	17 May 2010	Multiple db connections, CSRF, model validation
1.3	23 Mar 2011	Timezones, in browser testing, app templates.
1.5	26 Feb 2013	Python 3 Support, configurable user model
1.6	6 Nov 2013	Dedicated to Malcolm Tredinnick, db transaction management, connection pooling.
1.7	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	1 Apr 2015	Native support for multiple template engines.*Supported until at least April 2018*
1.9	1 Dec 2015	Automatic password validation. New styling for admin interface.
1.10	1 Aug 2016	Full text search for PostgreSQL. New-style middleware.
1.11 LTS	1.11 LTS	Last version to support Python 2.7.*Supported until at least April 2020*

2.0	Dec 2017	First Python 3-only release, Simplified URL routing syntax, Mobile friendly admin.

Views, Mapping URL to Views

View Creation (views.py):

- In Django, views are Python functions that handle web requests and return web responses.
- The views.py file is a convention for organizing view functions within a Django project.
- To create a view, you define a Python function that takes an HttpRequest object as its first parameter and returns an HttpResponse object.
- In the provided example, the hello view function simply returns an HttpResponse object with the text "Hello world".

```
# views.py
from django.http import HttpResponse
def hello(request):
    return HttpResponse("Hello world")
```

Explanation:

- Import the HttpResponse class from django. http module.
- Define a function named hello which will be the view.
- The view function takes a request parameter (an instance of HttpRequest), although it doesn't use it here.
- The view returns an HttpResponse object containing the text "Hello world".

URL Configuration (urls.py):

- Django uses a URLconf (URL configuration) to map URLs to view functions.
- The urls.py file is where you define the URL patterns for your Django project.

- In the URLconf, you import the view functions from views.py and map them to specific URLs using regular expressions.
- Each URL pattern is defined as a tuple containing a regular expression pattern and the corresponding view function.
- In the provided example, the URL pattern r'^hello/$' maps to the hello view function, indicating that the view should be invoked when the URL /hello/ is requested.

```
# urls.py
from django.conf.urls.defaults import *
from . import views
urlpatterns = patterns('',
   (r'^hello/$', views.hello),
)
```

Explanation:

- Import necessary modules and the hello view function from views.py.
- Define the urlpatterns variable, which is a mapping between URLs and the view functions.
- Map the URL /hello/ to the hello view function. When a user visits this URL, Django will call the hello function.

This setup tells Django to display "Hello world" when a user navigates to the /hello/ URL in the browser.

Request Handling:

- When a user makes a request to the Django server, Django's URL resolver examines the requested URL and determines which view function to call based on the URLconf.
- The view function receives an HttpRequest object as its first parameter, which contains metadata about the request (e.g., headers, user agent, request method).
- Although the hello view function doesn't utilize the HttpRequest object in this example, it's a standard parameter for all view functions in Django.

Response Generation:

- View functions in Django are responsible for generating web responses.
- In the provided example, the hello view function returns an HttpResponse object containing the string "Hello world".
- Django's HttpResponse class allows you to create HTTP responses with custom content (e.g., HTML, JSON) and status codes.

URL Mapping:

- URL patterns defined in the URLconf serve as a mapping between URLs and view functions.
- Django uses regular expressions to match incoming URLs to patterns defined in the URLconf.
- When a matching URL is found, Django calls the corresponding view function to handle the request and generate the response.

Working of Django URL Confs and Loose Coupling

In a dynamic web application, URLs often contain parameters that influence the output of the page. To demonstrate this, let's create a view in Django that displays the current date and time offset by a certain number of hours. Instead of creating separate views for each hour offset, which would clutter the URLconf, we can use a single view with a parameter in the URL to specify the hour offset dynamically.

View Function:

We'll create a single view function named offset_datetime, which takes an hour offset as a parameter and returns the current date and time offset by that number of hours.

```
# views.py
from django.http import HttpResponse
from datetime import datetime, timedelta


def offset_datetime(request, offset):
    try:
        offset_hours = int(offset)
        dt = datetime.now() + timedelta(hours=offset_hours)
        return HttpResponse(f"Current date/time offset by {offset_hours} hours: {dt}")
    except ValueError:
        return HttpResponse("Invalid offset")
```

2. URL Configuration:

- We'll define a dynamic URL pattern with a parameter to specify the hour offset.

```
# urls.py
from django.conf.urls.defaults import *
from . import views
urlpatterns = patterns('',
    (r'^time/plus/(?P<offset>\d+)/$', views.offset_datetime),
)
```

Explanation:

- The URL pattern r'^time/plus/(?P<offset>\d+)/$' matches URLs like /time/plus/1/, /time/plus/2/, etc., where the offset parameter specifies the number of hours to add to the current date/time.
- The (?P<offset>\d+) part of the pattern captures the value of the offset parameter as a string of digits and passes it to the offset_datetime view function.

Loose coupling is a fundamental principle in software development, and Django's URLconfs provide a clear example of its application within the framework. Here's a breakdown of how URLconfs and views demonstrate loose coupling in Django:

Definition Separation:

- In Django, URLconfs are used to map URLs to view functions, specifying which view function should be called for a given URL pattern.
- The URL definitions and the implementation of view functions are kept separate, residing in two distinct places within the Django project.

Interchangeability:

- Loose coupling ensures that changes made to one component have minimal impact on others.
- If two pieces of code are loosely coupled, modifications to one piece won't require changes to the other, promoting flexibility and maintainability.

Flexibility in Changes:

- With Django's URLconfs, you can modify URL patterns without affecting the view functions they map to, and vice versa.
- For example, changing the URL pattern for a view (e.g., moving from /time/ to /current-time/) can be accomplished without altering the view function itself.
- Similarly, modifications to the view function's logic can be made independently of URL changes.

Scalability:

- Loose coupling enables easy scaling and extension of functionality within a Django project.
- Adding new URLs or modifying existing ones can be done without disrupting other parts of the application.
- For instance, exposing a view function at multiple URLs can be achieved by editing the URLconf without touching the view code.

Maintainability:

- Separating URL definitions and view implementations promotes code clarity and makes it easier to understand and maintain the project.
- Developers can focus on specific tasks (e.g., URL routing or view logic) without needing extensive knowledge of other components.

By leveraging loose coupling, Django enables developers to build web applications that are flexible, modular, and easy to maintain.

This approach aligns with Django's philosophy of promoting rapid development and scalability while ensuring code robustness and maintainability.

Errors in Django

When you deliberately introduce a Python error into your Django project, such as commenting out critical lines of code, Django's error handling mechanisms come into play. Let's delve into how Django's error pages help you debug and understand the issue:

Error Introduction:

- By commenting out crucial lines of code in a view function, you intentionally introduce a Python error into your Django project.

Error Page Display:

- When you navigate to a URL that triggers the error, Django displays a detailed error page with significant information about the exception.
- At the top of the error page, Django presents key information about the exception, including the type of exception, parameters, file name, and line number where the exception occurred.
- Additionally, Django provides the full Python traceback for the exception, displaying each level of the stack trace along with the corresponding file, function/method name, and line number.

Interactive Traceback:

- Django's error page offers an interactive traceback, allowing you to click on any line of source code to view several lines before and after the erroneous line, providing context for better understanding.
- You can also click on "Local vars" under any frame in the stack to view a table of all local variables and their values at the exact point where the exception was raised, aiding in debugging.

Sharing and Debugging:

- Django offers options to easily share the traceback with others for technical support. You can switch to a copy-and-paste view or share the traceback on a public website like http://www.dpaste.com/.
- Additionally, the error page includes information about the incoming web request (GET and POST data, cookies, CGI headers) and Django settings, helping you identify the context of the error.

Debugging Aid:

- Developers accustomed to debugging with print statements can leverage Django's error page for debugging purposes by inserting an assert False statement at any point in the view function to trigger the error page and inspect the local variables and program state.

Security Considerations:

- It's essential to recognize that the information displayed on Django's error page is sensitive and should not be exposed on the public internet. Django only displays the error page when the project is in debug mode, which is automatically activated during development but should be deactivated in production to prevent potential security risks.

By utilizing Django's error pages effectively, developers can gain valuable insights into runtime errors, facilitate debugging, and ensure the robustness of their Django applications.

Wild Card patterns in URLS

Wildcard patterns, also known as catch-all patterns or wildcard segments, are a useful feature in Django's URL routing system. They allow you to capture any part of a URL and pass it as a parameter to a view function. Here's how wildcard patterns work in Django's URL configuration:

Basic URL Patterns:

- In Django, URL patterns are defined using regular expressions (regex) in the URLconf module (urls.py).
- Typically, URL patterns match specific URLs and route them to corresponding view functions.

Wildcard Patterns:

- Wildcard patterns allow you to capture variable parts of a URL and pass them as parameters to view functions.
- The most common wildcard pattern in Django's URL routing system is the (?P<parameter_name>pattern) syntax, where parameter_name is the name of the parameter and pattern is a regex pattern that matches the parameter value.
- This syntax captures the matched part of the URL and passes it as an argument to the associated view function.

Usage Example:

- Let's say you want to create a dynamic URL pattern for displaying details of a specific item. Instead of creating separate URL patterns for each item, you can use a wildcard pattern to capture the item's identifier from the URL.

Example:

```
# urls.py
from django.urls import path
from . import views
urlpatterns = [
   path('item/<int:item_id>/', views.item_detail),
]
```

- In this example, <int:item_id> is a wildcard pattern that captures an integer value from the URL and passes it as the item_id parameter to the item_detail view function.

Parameter Naming:

- When using wildcard patterns, it's important to give meaningful names to the captured parameters to improve code readability.
- Parameter names are enclosed in angle brackets (< >) and must be valid Python variable names.

Wildcard Segment:

- Django also supports wildcard segments in URL patterns, denoted by an asterisk (*), which captures the remainder of the URL as a single parameter.
- Wildcard segments are useful for creating catch-all patterns to handle dynamic routes.

Usage Example with Wildcard Segment:

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

urlpatterns = [
    path('blog/<slug:category>/', views.blog_category),
    path('blog/<slug:category>/<path:remainder>/', views.blog_post),
]
```

- In this example, <path:remainder> is a wildcard segment that captures the remaining part of the URL as a single parameter, allowing for dynamic handling of blog posts with variable URLs.

Wildcard patterns in Django's URL routing system provide flexibility and enable the creation of dynamic and expressive URL patterns, making it easier to build robust and scalable web applications