Django MVT (Model-View-Template) Architecture

Django follows the **MVT (Model-View-Template)** architectural pattern, which is a slight variation of the traditional **MVC (Model-View-Controller)** pattern. Below are the detailed notes on each component and how they work together in Django.

1. Model

- **Definition**: The Model is the data access layer. It handles everything related to the database.
- **Responsibilities**:
  - Defines the structure of the database (tables, fields, relationships).
  - Handles CRUD (Create, Read, Update, Delete) operations.
  - Contains the business logic and rules for data storage.
- **Key Features**:
  - Django provides an **ORM (Object-Relational Mapper)** to interact with the database using Python code instead of SQL.
  - Models are defined as Python classes that inherit from `django.db.models.Model`.
  - Each attribute in the model class represents a database field.
- **Example**:

```
from django.db import models

class Book(models.Model):
        title = models.CharField(max_length=100)
        author = models.CharField(max_length=100)
        published_date = models.DateField()
```

- **Database Migrations**:
  - Django automatically generates database schema from models using migrations.
  - Commands:
    - `python manage.py makemigrations`: Creates migration files.
    - `python manage.py migrate`: Applies migrations to the database.

2. View

- **Definition**: The View is the business logic layer. It processes user requests and returns responses.
- **Responsibilities**:
  - Fetches data from the Model.
  - Processes the data (e.g., filtering, sorting).
  - Passes data to the Template for rendering.
  - Handles user input (e.g., form submissions).
- **Types of Views**:
  - **Function-Based Views (FBVs)**: Simple views defined as Python functions.

```python
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()
    return render(request, 'books/book_list.html', {'books': books})
```

  - **Class-Based Views (CBVs)**: More complex views defined as Python classes.

```python
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
    model = Book
    template_name = 'books/book_list.html'
    context_object_name = 'books'
```

- **Key Features**:
  - Views are mapped to URLs via the urls.py file.
  - Use HttpRequest and HttpResponse objects to handle requests and responses.
  - Can return HTML, JSON, or other types of responses.

3. Template

- **Definition**: The Template is the presentation layer. It defines how data is displayed to the user.
- **Responsibilities**:
  - Renders HTML dynamically using data passed from the View.
  - Separates design (HTML/CSS) from logic (Python).
- **Key Features**:
  - Uses Django's **template language** to insert dynamic content.
  - Supports template inheritance for reusable layouts.
  - Includes tags, filters, and variables for dynamic rendering.
- **Example**:

```html
<!-- base.html -->
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
    <div id="content">
    {% block content %}{% endblock %}
    </div>
</body>
</html>
```

```
<!-- book_list.html -->
{% extends "base.html" %}

{% block title %}Book List{% endblock %}

{% block content %}
<h1>Books</h1>
<ul>
    {% for book in books %}
    <li>{{ book.title }} by {{ book.author }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

Run HTML

- **Template Tags and Filters**:
  - Tags: {% for %}, {% if %}, {% block %}, {% extends %}, etc.
  - Filters: {{ value|lower }}, {{ value|date:"Y-m-d" }}, etc.

4. URL Dispatcher

- **Definition**: Maps URLs to Views.
- **Responsibilities**:
  - Routes incoming requests to the appropriate View based on the URL.
  - Uses regular expressions or path converters to define URL patterns.
- **Example**:

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

urlpatterns = [
    path('books/', views.book_list, name='book_list'),
    path('books/<int:pk>/', views.book_detail, name='book_detail'),
]
```

- **Key Features**:
  - URLs are defined in the urls.py file.
  - Supports named URLs for reverse resolution.
  - Can include other URL configurations using include().

5. How MVT Works Together

1. **User Request**: A user sends a request to a specific URL.
2. **URL Dispatcher**: Django matches the URL to a View.
3. **View**: The View fetches data from the Model and processes it.
4. **Template**: The View passes the data to the Template for rendering.

5. **Response**: The rendered HTML is sent back to the user as a response.

6. Key Differences Between MVT and MVC

- **MVC**:
  - **Controller**: Handles user input and updates the Model and View.
- **MVT**:
  - **View**: Acts as the Controller in MVC, handling user input and business logic.
  - **Template**: Acts as the View in MVC, handling presentation logic.

7. Advantages of Django MVT

- **Separation of Concerns**: Clear division between data, logic, and presentation.
- **Reusability**: Templates and Views can be reused across applications.
- **Scalability**: Easy to scale and maintain large applications.
- **Built-in Features**: Django provides built-in tools for authentication, admin interface, and more.

8. Common Commands

- Start a new project: `django-admin startproject projectname`
- Start a new app: `python manage.py startapp appname`
- Run the development server: `python manage.py runserver`
- Create migrations: `python manage.py makemigrations`
- Apply migrations: `python manage.py migrate`
- Create a superuser: `python manage.py createsuperuser`

From <https://chat.deepseek.com/a/chat/s/d2e31f17-78ea-4e6b-a5e9-ab67fee903a8>

Django Forms: Comprehensive Notes

Django provides a powerful form-handling mechanism to manage user input, validate data, and render HTML forms. Below are detailed notes on Django forms, including predefined form classes and their usage.

1. Introduction to Django Forms

- **Purpose**: Handle user input, validate data, and render HTML forms.
- **Key Features**:
  - Automatically generate HTML form elements.
  - Validate user input and display error messages.

    - Handle form submission and data processing.
- **Types of Forms**:
    - **Regular Forms**: Defined using django.forms.Form.
    - **Model Forms**: Linked to a Django model for CRUD operations.

2. Creating a Basic Form

- Define a form class by inheriting from django.forms.Form.
- Use form fields to define input types (e.g., CharField, EmailField, DateField).
- Example:

```
from django import forms

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

3. Rendering Forms in Templates

- Pass the form instance to the template context.
- Use Django's template language to render the form.
- Example:

```
<form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
</form>
```

Run HTML

- **Form Rendering Options**:
    - {{ form.as_p }}: Renders form fields as paragraphs.
    - {{ form.as_table }}: Renders form fields as table rows.
    - {{ form.as_ul }}: Renders form fields as list items.
    - Custom rendering: Manually render each field using {{ form.field_name }}.

4. Handling Form Submission

- In the View, handle GET and POST requests.
- Validate the form using form.is_valid().
- Access cleaned data using form.cleaned_data.
- Example:

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

def contact_view(request):
```

```
if request.method == 'POST':
form = ContactForm(request.POST)
if form.is_valid():
name = form.cleaned_data['name']
email = form.cleaned_data['email']
message = form.cleaned_data['message']
# Process the data (e.g., save to database, send email)
return redirect('success_url')
else:
form = ContactForm()
return render(request, 'contact.html', {'form': form})
```

5. Model Forms

- **Purpose**: Create forms linked to Django models for CRUD operations.
- Inherit from django.forms.ModelForm.
- Automatically generate form fields based on the model.
- Example:

```
from django.forms import ModelForm
from .models import Book

class BookForm(ModelForm):
    class Meta:
    model = Book
    fields = ['title', 'author', 'published_date']
```

- **Usage**:
  - Create a new record: form = BookForm(request.POST)
  - Update an existing record: form = BookForm(request.POST, instance=book)

6. Predefined Form Classes

Django provides several predefined form classes for common use cases:

a. Authentication Forms

- **Login Form** (django.contrib.auth.forms.AuthenticationForm):

```
from django.contrib.auth.forms import AuthenticationForm

def login_view(request):
    if request.method == 'POST':
    form = AuthenticationForm(request, data=request.POST)
    if form.is_valid():
    # Log the user in
    return redirect('home')
    else:
    form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

- **User Creation Form** (django.contrib.auth.forms.UserCreationForm):

```
from django.contrib.auth.forms import UserCreationForm

def register_view(request):
    if request.method == 'POST':
    form = UserCreationForm(request.POST)
    if form.is_valid():
    form.save()
    return redirect('login')
    else:
    form = UserCreationForm()
    return render(request, 'register.html', {'form': form})
```

b. Password Reset Forms

- **PasswordResetForm**: For requesting a password reset.
- **SetPasswordForm**: For setting a new password after reset.

c. Admin Forms

- **AdminAuthenticationForm**: Used in the Django admin login page.

7. Form Fields

Django provides a wide range of form fields to handle different types of input:

- **CharField**: Text input.
- **EmailField**: Email input with validation.
- **DateField**: Date input.
- **ChoiceField**: Dropdown or radio buttons.
- **FileField**: File upload.
- **BooleanField**: Checkbox.
- Example:

```
class SurveyForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    age = forms.IntegerField()
    gender = forms.ChoiceField(choices=[('M', 'Male'), ('F', 'Female')])
    agree = forms.BooleanField(label="I agree to the terms")
```

8. Form Widgets

- **Purpose**: Control the HTML rendering of form fields.
- Common Widgets:
  - TextInput: Renders as <input type="text">.
  - Textarea: Renders as <textarea>.
  - Select: Renders as <select>.
  - CheckboxInput: Renders as <input type="checkbox">.
- Example:

```
class CommentForm(forms.Form):
        comment = forms.CharField(widget=forms.Textarea(attrs={'rows': 4,
'cols': 40}))
```

9. Form Validation

- **Built-in Validation**:
  - Django automatically validates form fields based on their type (e.g., EmailField validates email format).
- **Custom Validation**:
  - Override the clean() method or add clean_<fieldname>() methods.
  - Example:

```
class ContactForm(forms.Form):
        name = forms.CharField(max_length=100)
        email = forms.EmailField()

def clean_name(self):
        name = self.cleaned_data['name']
        if len(name) < 3:
        raise forms.ValidationError("Name must be at least 3 characters
long.")
        return name
```

10. Formset

- **Purpose**: Handle multiple instances of a form on a single page.
- **Usage**:
  - Create a formset using formset_factory.
  - Example:

```
from django.forms import formset_factory
from .forms import ContactForm

ContactFormSet = formset_factory(ContactForm, extra=2)

def contact_formset_view(request):
        if request.method == 'POST':
        formset = ContactFormSet(request.POST)
        if formset.is_valid():
        for form in formset:
                # Process each form
                pass
        else:
        formset = ContactFormSet()
        return render(request, 'contact_formset.html', {'formset':
formset})
```

11. Best Practices

- Use ModelForms for CRUD operations.
- Validate user input both on the client and server sides.
- Use CSRF protection ({% csrf_token %}) to prevent cross-site request forgery.
- Keep forms modular and reusable.

12. Common Commands

- Create a form: Define a class inheriting from forms.Form or forms.ModelForm.
- Render a form: Use {{ form }} in templates.
- Validate a form: Call form.is_valid() in the view.
- Save data: Use form.save() for ModelForms.

From <https://chat.deepseek.com/a/chat/s/d2e31f17-78ea-4e6b-a5e9-ab67fee903a8>

Django Authentication: Login, Registration, and Predefined Classes

Django provides a robust built-in authentication system to handle user login, registration, and other user management tasks. Below are detailed notes on how to implement login and registration in Django, along with predefined classes and methods.

1. Django Authentication System Overview

- **Purpose**: Manage user authentication (login, logout, registration, password reset).
- **Key Features**:
    - User model for storing user information.
    - Built-in views and forms for authentication.
    - Middleware for session management.
    - Password hashing and validation.

2. User Model

- Django provides a built-in User model (django.contrib.auth.models.User) with fields like:
    - username
    - password
    - email
    - first_name
    - last_name
    - is_active
    - is_staff
    - is_superuser

- **Custom User Model**:
    - You can create a custom user model by extending AbstractUser or AbstractBaseUser.
    - Example:

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

class CustomUser(AbstractUser):
        phone_number = models.CharField(max_length=15)
```

3. Registration (User Creation)

a. Using UserCreationForm

- Django provides a predefined form for user registration: django.contrib.auth.forms.UserCreationForm.
- Example:

```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render, redirect

def register_view(request):
        if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
        form.save()
        return redirect('login')  # Redirect to login page after registration
        else:
        form = UserCreationForm()
        return render(request, 'register.html', {'form': form})
```

b. Custom Registration Form

- Extend UserCreationForm to add additional fields (e.g., email, phone number).
- Example:

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

class CustomUserCreationForm(UserCreationForm):
        email = forms.EmailField(required=True)

class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

4. Login

a. Using AuthenticationForm

- Django provides a predefined form for user login: `django.contrib.auth.forms.AuthenticationForm`.
- Example:
  python
  Copy
  ```
  from django.contrib.auth import login, authenticate
  from django.contrib.auth.forms import AuthenticationForm
  from django.shortcuts import render, redirect

  def login_view(request):
      if request.method == 'POST':
      form = AuthenticationForm(request, data=request.POST)
      if form.is_valid():
      username = form.cleaned_data.get('username')
      password = form.cleaned_data.get('password')
      user = authenticate(username=username, password=password)
      if user is not None:
          login(request, user)
          return redirect('home')  # Redirect to home page after login
      else:
      form = AuthenticationForm()
      return render(request, 'login.html', {'form': form})
  ```

b. Using `LoginView` (Class-Based View)

- Django provides a predefined class-based view for login: `django.contrib.auth.views.LoginView`.
- Example:
  ```
  from django.contrib.auth.views import LoginView
  from django.urls import path

  urlpatterns = [
      path('login/', LoginView.as_view(template_name='login.html'),
  name='login'),
  ]
  ```

5. Logout

a. Using `logout()` Function

- Django provides a `logout()` function to log out the user.
- Example:
  ```
  from django.contrib.auth import logout
  from django.shortcuts import redirect

  def logout_view(request):
  ```

```
        logout(request)
        return redirect('home')  # Redirect to home page after logout
```

b. Using `LogoutView` (Class-Based View)

- Django provides a predefined class-based view for logout: `django.contrib.auth.views.LogoutView`.
- Example:

```
from django.contrib.auth.views import LogoutView
from django.urls import path

urlpatterns = [
    path('logout/', LogoutView.as_view(), name='logout'),
]
```

6. Password Reset

a. Using Predefined Views

- Django provides built-in views for password reset:
  - `PasswordResetView`: For requesting a password reset.
  - `PasswordResetDoneView`: Confirms the password reset request.
  - `PasswordResetConfirmView`: Allows the user to set a new password.
  - `PasswordResetCompleteView`: Confirms the password reset is complete.
- Example:

```
from django.contrib.auth.views import (
    PasswordResetView,
    PasswordResetDoneView,
    PasswordResetConfirmView,
    PasswordResetCompleteView,
)
from django.urls import path

urlpatterns = [
    path('password-reset/', PasswordResetView.as_view(),
name='password_reset'),
    path('password-reset/done/', PasswordResetDoneView.as_view(),
name='password_reset_done'),
    path('reset/<uidb64>/<token>/', PasswordResetConfirmView.as_view(),
name='password_reset_confirm'),
    path('reset/done/', PasswordResetCompleteView.as_view(),
name='password_reset_complete'),
]
```

b. Customizing Password Reset Emails

- Customize the email template by creating `registration/password_reset_email.html` and `registration/password_reset_subject.txt` in your templates directory.

7. Password Change

a. Using PasswordChangeForm

- Django provides a predefined form for changing passwords: django.contrib.auth.forms.PasswordChangeForm.
- Example:

```
 from django.contrib.auth.forms import PasswordChangeForm
 from django.contrib.auth import update_session_auth_hash
 from django.shortcuts import render, redirect

 def password_change_view(request):
         if request.method == 'POST':
         form = PasswordChangeForm(request.user, request.POST)
         if form.is_valid():
         user = form.save()
           update_session_auth_hash(request, user)  # Keep the user logged in
         return redirect('home')
         else:
         form = PasswordChangeForm(request.user)
         return render(request, 'password_change.html', {'form': form})
```

b. Using PasswordChangeView (Class-Based View)

- Django provides a predefined class-based view for password change: django.contrib.auth.views.PasswordChangeView.
- Example:

```
 from django.contrib.auth.views import PasswordChangeView
 from django.urls import path

 urlpatterns = [
         path('password-change/', PasswordChangeView.as_view(),
name='password_change'),
 ]
```

8. Predefined Classes and Methods

a. Forms

- UserCreationForm: For user registration.
- AuthenticationForm: For user login.
- PasswordResetForm: For password reset requests.
- SetPasswordForm: For setting a new password.
- PasswordChangeForm: For changing the password.

b. Views

- LoginView: For handling login.
- LogoutView: For handling logout.
- PasswordResetView: For handling password reset requests.
- PasswordResetConfirmView: For confirming password reset.

- PasswordChangeView: For handling password changes.

c. Methods

- authenticate(): Verifies user credentials.
- login(): Logs in a user.
- logout(): Logs out a user.
- update_session_auth_hash(): Updates the session after password change.

9. Best Practices

- Use Django’s built-in authentication system for security and efficiency.
- Customize templates for login, registration, and password reset.
- Use HTTPS to secure authentication data.
- Validate user input on both client and server sides.

10. Common Commands

- Create a superuser: python manage.py createsuperuser
- Run the development server: python manage.py runserver
- Apply migrations: python manage.py migrate

From <https://chat.deepseek.com/a/chat/s/d2e31f17-78ea-4e6b-a5e9-ab67fee903a8>