Chapter 14 - Django

2.1 Django Overview

What is Django? Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the **Model-View-Template (MVT)** architectural pattern and is known for its simplicity, scalability, and security. Django handles much of the hassle of web development, allowing developers to focus on writing applications without reinventing the wheel.

Key Features of Django:

- **Built-in Admin Interface:** A powerful interface for managing data.
- **Security:** Protection against common attacks like SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).
- **URL Routing:** Flexible and readable URL routing.
- **ORM (Object-Relational Mapping):** Allows for easy interaction with databases.
- **Template Engine:** Facilitates separating the HTML code from the business logic.
- **Forms handling:** Easy handling of form input and validation.

2.2 Installing Django

To install Django, you need Python installed on your machine. You can install Django using `pip`, the Python package installer.

Steps:

1. Install Django via pip:

   ```
   pip install django
   ```

2. Verify the installation:

   ```
   python -m django --version
   ```

Note: Ensure you are using Python 3.6 or newer as Django is fully compatible with these versions.

2.3 Initializing a Django Project

A Django project is a collection of settings for an instance of Django. It contains configurations for settings, URLs, and the database.

Steps to initialize a project:

1. Open a terminal and run:

   ```
   django-admin startproject myproject
   ```

   This will create a directory called `myproject` with the following structure:

   ```
   myproject/
       manage.py
       myproject/
           __init__.py
           settings.py
           urls.py
           wsgi.py
           asgi.py
   ```

2. To run the development server:

   ```
   cd myproject
   python manage.py runserver
   ```

   You should see output indicating the server is running at `http://127.0.0.1:8000/`.

2.4 Creating a Django Application

A Django project can contain multiple applications. Each app is responsible for a specific functionality, such as managing a blog, handling user authentication, or managing a store.

Steps to create an app:

1. Inside the project folder, run:

   ```
   python manage.py startapp myapp
   ```

2. This creates a new folder `myapp/` with the following structure:

   ```
   myapp/
       __init__.py
       admin.py
       apps.py
       models.py
       tests.py
       views.py
   ```

3. You must add the app to the `INSTALLED_APPS` list in the `settings.py` file of the project:

   ```
   INSTALLED_APPS = [
       # other apps,
       'myapp',
   ]
   ```

2.5 Django ORM (Object-Relational Mapping)

Django's ORM allows you to interact with the database using Python code instead of writing raw SQL queries. This makes it easier to work with databases in an object-oriented way.

Key ORM Features:

- **Models:** Classes that define the structure of your database tables.
- **QuerySets:** Used to retrieve data from the database.
- **Migrations:** Track changes in the database schema over time.

2.6 Django Login

Django provides a built-in authentication system that handles user login, logout, and session management. To enable login functionality, you can use Django's `django.contrib.auth` module.

Basic login setup:

1. In your app's `views.py`, create a login view:

```python
from django.contrib.auth import authenticate, login
from django.shortcuts import render, redirect

def user_login(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')  # Redirect to home page
        else:
            return render(request, 'login.html', {'error': 'Invalid
credentials'})
    return render(request, 'login.html')
```

2. You also need to create a URL pattern for this view in your `urls.py`.

2.7 Models Layer

2.7.1 Models: A model in Django is a Python class that defines the fields and behaviors of the data you're storing in the database. Django models are mapped to database tables.

Example of a simple model for storing `Book` details:

```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()

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

2.7.2 QuerySets: A QuerySet is a collection of database queries. You can filter and retrieve data in a flexible way using the `filter()`, `exclude()`, and `get()` methods.

Example:

```
books = Book.objects.filter(author="J.K. Rowling")
```

2.7.3 Instance methods and accessing related objects: You can define methods inside your models. These methods can help encapsulate logic related to the instance.

Example:

```
class Book(models.Model):
    title = models.CharField(max_length=200)

    def is_published(self):
        return self.published_date <= timezone.now()
```

2.7.4 Existing databases and supported databases: Django supports several databases like PostgreSQL, MySQL, SQLite, and Oracle. You can also use an existing database by configuring the `DATABASES` setting in `settings.py`.

2.8 Views Layer

2.8.1 URLConfs: URL configurations map URL patterns to views. In Django, URLs are defined in the `urls.py` file.

2.8.2 View functions: Views are Python functions that accept web requests and return web responses. Views can be written as simple functions or class-based views.

Example of a view function:

```
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse("Hello, world!")
```

2.8.3 Shortcuts: Django provides shortcut functions to simplify view logic, such as `render()` for rendering templates and `redirect()` for URL redirection.

2.8.4 Included views: Django includes built-in generic views for common tasks like displaying a list of objects, creating new objects, or handling forms.

2.8.5 Request-response objects and TemplateResponse objects

Request-Response Cycle: In Django, every interaction between the client (browser) and the server (Django app) follows the **request-response** cycle. When a client makes an HTTP request, Django processes it and returns an HTTP response. The process involves receiving the request, executing a view function, generating a response, and sending it back to the client.

- **Request object**: This encapsulates all the details of the incoming request. You can access various data like HTTP method, GET and POST data, session information, and more. Example:

```
def my_view(request):
    user_agent = request.META['HTTP_USER_AGENT']
    return HttpResponse(f"Your user agent is: {user_agent}")
```

- **Response object**: The `HttpResponse` object represents the server's response to the client. It can contain text, HTML, JSON, etc. A `TemplateResponse` is a special kind of response that is used when rendering a template. Example of returning a simple HTTP response:

```
from django.http import HttpResponse

def my_view(request):
    return HttpResponse("Hello, world!")
```

  TemplateResponse: A `TemplateResponse` is an object that combines a response with a template rendering. It allows for the lazy rendering of templates, which can be useful when the view context is determined dynamically. Example:

```
from django.template.response import TemplateResponse

def my_view(request):
    return TemplateResponse(request, 'my_template.html', {'name':
'Django'})
```

2.8.6 Generating PDFs and CSVs

Django doesn't natively provide functionality for generating PDFs or CSVs, but there are third-party libraries that can help.

- **Generating PDFs:** A popular library for generating PDFs in Django is `ReportLab`. You can use it to create dynamic PDFs based on the data in your application. Example:

```python
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def generate_pdf(request):
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'inline; filename="mypdf.pdf"'

    p = canvas.Canvas(response, pagesize=letter)
    p.drawString(100, 750, "Hello, world!")
    p.showPage()
    p.save()

    return response
```

- **Generating CSVs:** Python's built-in `csv` module can be used to generate CSV files dynamically in Django views. Example:

```python
import csv
from django.http import HttpResponse

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

    writer = csv.writer(response)
    writer.writerow(['Name', 'Age'])
    writer.writerow(['Alice', 30])
    writer.writerow(['Bob', 25])

    return response
```

2.8.7 Included Middleware Classes

Middleware is a way to process requests globally before they reach the view or after the view has processed the response. Middleware is a lightweight, low-level plugin system for globally altering Django's input or output.

Some common middleware classes included in Django:

- **SecurityMiddleware**: Adds various security headers to HTTP responses (e.g., preventing clickjacking, XSS protection).
- **SessionMiddleware**: Manages session data for users.
- **AuthenticationMiddleware**: Manages user authentication and assigns a user to the request object.

You can add custom middleware or modify the existing ones by including them in the `MIDDLEWARE` setting in `settings.py`.

2.9 Templates Layer

Templates in Django are used to dynamically generate HTML (or any text-based format) that is returned to the client. Templates are typically written in HTML with special Django template tags and filters to add dynamic behavior.

2.9.1 Included Tags and Filters

Django templates include several built-in tags and filters to enhance the functionality of templates.

- **Tags** are used to control the logic in templates. Examples include `{% for %}`, `{% if %}`, `{% block %}`, etc.
- **Filters** modify the display of variables. For instance, `{{ name|lower }}` would convert `name` to lowercase.

Examples of common tags:

```
{% for item in items %}
  <p>{{ item }}</p>
{% endfor %}
```

Examples of common filters:

```
<p>{{ user.name|default:"Anonymous" }}</p>
<p>{{ price|floatformat:2 }}</p>
```

2.9.2 Custom Tags and Filters

You can define your own custom template tags and filters in Django by creating a `templatetags` module in one of your apps.

Example of a custom filter:

```
# myapp/templatetags/custom_filters.py
from django import template

register = template.Library()

@register.filter
def multiply(value, arg):
    return value * arg
```

Then, in your template:

```
{% load custom_filters %}
<p>{{ 5|multiply:2 }}</p>
```

Example of a custom tag:

```
# myapp/templatetags/custom_tags.py
from django import template

register = template.Library()

@register.simple_tag
def greet(name):
    return f"Hello, {name}!"
```

Usage in a template:

```
{% load custom_tags %}
<p>{% greet "Alice" %}</p>
```

2.9.3 Template API

Django's template system exposes an API for dynamically rendering templates. It provides a method to render templates from the view and pass data to the templates.

Example of rendering a template from a view:

```
from django.shortcuts import render

def my_view(request):
    context = {'name': 'Django'}
    return render(request, 'my_template.html', context)
```

2.9.4 Custom Template Engine

If the built-in Django template engine doesn't meet your needs, you can integrate other template engines like Jinja2. To use Jinja2, modify the `TEMPLATES` setting in `settings.py`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
    },
]
```

2.10 Forms

2.10.1 Forms API

Django provides a powerful form handling system for user input and validation. The `django.forms` module provides classes to create forms easily.

Example of a simple form:

```
from django import forms

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

2.10.2 Included Fields / Included Components

Django forms include many useful fields, such as `CharField`, `EmailField`, `IntegerField`, and `DateField`. You can also use widgets to render the fields.

2.10.3 Model Forms

Model forms are forms tied to Django models, making it easy to create, update, and validate model instances using forms.

Example:

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

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

2.10.4 Customizing Validation

You can add custom validation to form fields using `clean_<fieldname>` methods.

Example:

```
def clean_name(self):
    name = self.cleaned_data['name']
    if len(name) < 2:
        raise forms.ValidationError("Name must be at least 2 characters long")
    return name
```

2.11 Django Admin Interface

Django provides a built-in admin interface to manage your models.

Key Features:

- **Auto-generated CRUD interface** for your models.
- Customization to define how models are displayed.

Example:

```
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

2.12 Security

2.12.1 Published Security Advisories Django actively publishes security advisories to help developers stay informed about potential vulnerabilities. Regularly check the Django security page for updates.

2.12.2 Protection against Clickjacking

Django includes a middleware (`XFrameOptionsMiddleware`) to protect against clickjacking. You can enable this feature by adding the middleware to `settings.py` .

2.12.3 Protection against Cross-Site Request Forgery (CSRF) Token

Django has built-in protection against CSRF attacks. Ensure that the `{% csrf_token %}` tag is included in your forms to protect against these attacks.

Example:

```
<form method="post">
    {% csrf_token %}
    <input type="text" name="username" />
    <button type="submit">Submit</button>
</form>
```

Exercises:

1. Implement a view to generate a PDF or CSV from a list of items in your database.
2. Create a custom filter and custom tag in Django templates.
3. Build a form using Django's `ModelForm` for a `Product` model and validate the form input.
4. Set up Django admin for managing `Book` and `Author` models, and customize the display in the admin interface.
5. Implement CSRF protection in a custom form in your Django app.