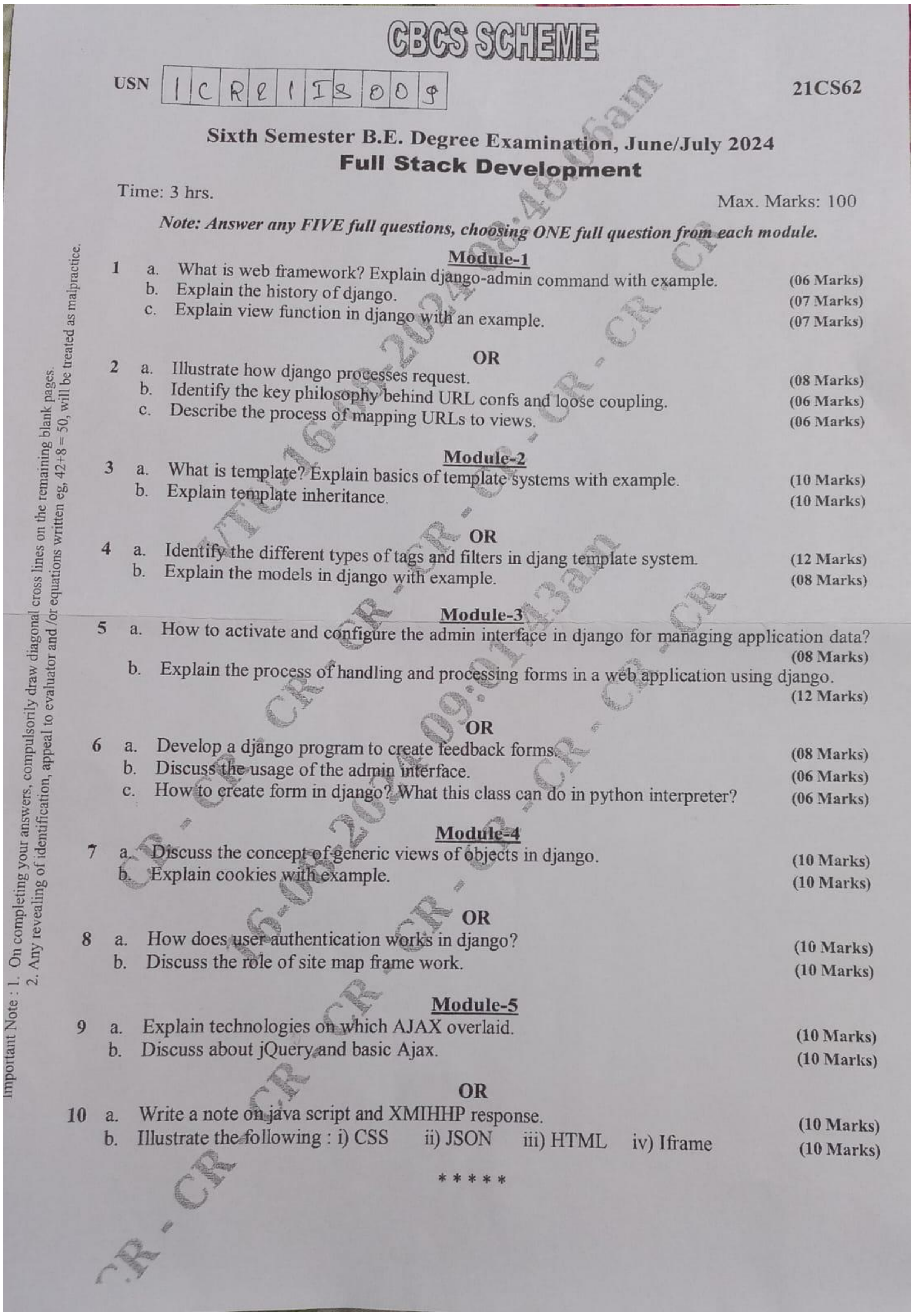

CBCS SCHEME

USN 1CR21IS009

21CS62

Sixth Semester B.E. Degree Examination, June/July 2024
Full Stack Development

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

Module-1

1 a. What is web framework? Explain django-admin command with example. **(06 Marks)**
b. Explain the history of django. **(07 Marks)**
c. Explain view function in django with an example. **(07 Marks)**

OR

2 a. Illustrate how django processes request. **(08 Marks)**
b. Identify the key philosophy behind URL confs and loose coupling. **(06 Marks)**
c. Describe the process of mapping URLs to views. **(06 Marks)**

Module-2

3 a. What is template? Explain basics of template systems with example. **(10 Marks)**
b. Explain template inheritance. **(10 Marks)**

OR

4 a. Identify the different types of tags and filters in djang template system. **(12 Marks)**
b. Explain the models in django with example. **(08 Marks)**

Module-3

5 a. How to activate and configure the admin interface in django for managing application data? **(08 Marks)**
b. Explain the process of handling and processing forms in a web application using django. **(12 Marks)**

OR

6 a. Develop a django program to create feedback forms. **(08 Marks)**
b. Discuss the usage of the admin interface. **(06 Marks)**
c. How to create form in django? What this class can do in python interpreter? **(06 Marks)**

Module-4

7 a. Discuss the concept of generic views of objects in django. **(10 Marks)**
b. Explain cookies with example. **(10 Marks)**

OR

8 a. How does user authentication works in django? **(10 Marks)**
b. Discuss the role of site map frame work. **(10 Marks)**

Module-5

9 a. Explain technologies on which AJAX overlaid. **(10 Marks)**
b. Discuss about jQuery and basic Ajax. **(10 Marks)**

OR

10 a. Write a note on java script and XMIHHP response. **(10 Marks)**
b. Illustrate the following : i) CSS ii) JSON iii) HTML iv) Iframe **(10 Marks)**

* * * * *

Module-1

1. a. What is a web framework? Explain django-admin command with an example.
   b. Explain the history of Django.
   c. Explain the view function in Django with an example.
   (06 Marks, 07 Marks, 07 Marks)

Solution
1. a) What is a web framework? Explain the django-admin command with an example.

- **Web Framework:** A web framework is a software framework that provides a standard way to build and deploy web applications. Frameworks simplify the development process by offering pre-built modules, libraries, and tools that allow developers to create web applications without reinventing the wheel. Web frameworks often provide solutions for routing (handling different web addresses), database management, templating (rendering HTML with dynamic data), session management, and other common functionalities. Examples of popular web frameworks include Django (Python), Ruby on Rails (Ruby), Laravel (PHP), and Spring (Java).
- **django-admin command:** Django includes a command-line utility called django-admin to help manage and streamline project creation and management tasks. Some common uses of django-admin are:
  - django-admin startproject <projectname>: Creates a new Django project directory structure, initializing a basic project with essential files.
  - django-admin startapp <appname>: Creates a new application within a Django project, organizing code for a specific feature or functionality (e.g., blog, authentication).
  - django-admin runserver: Runs a local development server to test the application.

Example: To create a new Django project, you can use the command:

```
django-admin startproject mysite
```

This command creates a project named "mysite" with a directory structure including settings, URLs, and configuration files.

---

1. b) Explain the history of Django.

- **History of Django:** Django was created in 2003 by Adrian Holovaty and Simon Willison while working at the Lawrence Journal-World, a newspaper company in Kansas, USA. They needed a web development tool to meet the tight deadlines and handle high-demand web applications for the newspaper. Recognizing the potential of their tool, they open-sourced Django in 2005 to allow other developers to use and improve it. Named after jazz guitarist Django Reinhardt, the framework was designed to support "rapid development" and focus on the "DRY" (Don't Repeat Yourself) principle. Django has since become one of the most widely used web frameworks for building robust, secure, and scalable web applications.

---

1. c) Explain view function in Django with an example.

- **View Function:** In Django, a view is a function or a class-based method that takes a web request and returns a web response. This response could be an HTML page, a redirect, or an error message. Views act as the middle layer between the database (model) and the presentation layer (templates). They process the logic and data needed by templates to create dynamic web pages.

Example of a view function:

```
from django.http import HttpResponse


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

In this example, hello_world is a simple view function that takes a request and returns a response saying "Hello, world!" The HttpResponse object is used to send the text back to the client.

OR

2. a. Illustrate how Django processes requests.
   b. Identify the key philosophy behind URL confs and loose coupling.
   c. Describe the process of mapping URLs to views.
   (08 Marks, 06 Marks, 06 Marks)

Solution

2. a) Illustrate how Django processes a request.

- **Django Request Processing:** Django processes an incoming web request in several steps:
    1. **URL Routing**: Django first checks the URL of the request against a list of URL patterns defined in the urls.py file. It matches the incoming URL with one of these patterns.
    2. **View Function**: Once a URL is matched, Django calls the corresponding view function for that pattern.
    3. **Middleware**: Middleware components process the request before and after it reaches the view. Middleware can handle tasks like authentication, session management, and request logging.
    4. **Response**: After processing the request, the view returns a response, typically rendered HTML or JSON data. The response goes back through the middleware and is finally sent to the client.

---

2. b) Identify the key philosophy behind URL confs and loose coupling.

- **URL Confs and Loose Coupling:** In Django, URLs are defined in a separate urls.py file and mapped to view functions. This separation, known as "loose coupling," ensures that the application's components are independent. URL configurations (URLConfs) can be changed without affecting view logic. This design makes Django applications more modular and maintainable since URLs and view logic are loosely coupled.

---

2. c) Describe the process of mapping URLs to views.

- **Mapping URLs to Views:** In Django, URL patterns are defined in urls.py, which maps each URL pattern to a specific view function. When a URL is accessed, Django looks for a matching pattern in urls.py and calls the corresponding view function.

Example of URL mapping:

```
from django.urls import path

from . import views


urlpatterns = [

    path('hello/', views.hello_world, name='hello'),

]
```

Here, the URL pattern hello/ is mapped to the hello_world view function in views.py. When /hello/ is accessed, Django invokes hello_world and returns its response.

Module-2

3. a. What is a template? Explain the basics of template systems with an example.
   b. Explain template inheritance.
   (10 Marks, 10 Marks)

Solution
3. a) What is a template? Explain the basics of template systems with an example.

- **Template:** In Django, templates are text files that define the structure of a web page. Templates use variables and logic (e.g., loops, conditionals) to render dynamic data. Django's template system separates presentation from business logic, making the code cleaner and easier to maintain.

Example of a template file:

```
<h1>Welcome, {{ user.name }}!</h1>
```

Here, {{ user.name }} is a variable that gets replaced with the actual data when the template is rendered.

---

3. b) Explain template inheritance.

- **Template Inheritance:** Template inheritance allows templates to extend a base layout, promoting code reuse and consistency. The {% block %} tag defines sections that child templates can override. {% extends %} is used in child templates to inherit the base template.

Example:

```
<!-- base.html -->

<html>

<body>

   <h1>Website Header</h1>

   {% block content %}{% endblock %}

</body>

</html>
```

html

Copy code

```
<!-- child_template.html -->

{% extends "base.html" %}

{% block content %}

   <p>This is child template content.</p>

{% endblock %}
```

OR

4. a. Identify the different types of tags and filters in Django template system.
   b. Explain the models in Django with an example.
   (12 Marks, 08 Marks)

Solution

4. a) Identify the different types of tags and filters in the Django template system.

- **Tags and Filters:**
  - **Tags** control logic, like {% for %}, {% if %}, and {% block %}.
  - **Filters** modify variable output. For example, {{ name|lower }} converts name to lowercase.

4. b) Explain models in Django with an example.

- **Django Models:** Models represent the structure of the database in Django. They are Python classes that inherit from models.Model and map to database tables.

Example:

```
from django.db import models


class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

Module-3

5. a. How to activate and configure the admin interface in Django for managing application data?
   b. Explain the process of handling and processing forms in a web application using Django.
   (08 Marks, 12 Marks)

Solution

5. a) How to activate and configure the admin interface in Django for managing application data?

- To activate the admin interface, add 'django.contrib.admin' to INSTALLED_APPS in settings.py. Then, run python manage.py createsuperuser to create a superuser account. Access the admin at /admin.

5. b) How to create custom admin views for models in Django?

- **Custom Admin Views:** In Django, the admin interface provides a default way to manage models, but you can customize how your models are displayed in the admin by creating an Admin class for each model and registering it with admin.site.register. Customization can include displaying specific fields, adding search functionality, filtering, and defining custom actions.

Example:

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


class StudentAdmin(admin.ModelAdmin):
    list_display = ('name', 'age')
    search_fields = ('name',)
    list_filter = ('age',)


admin.site.register(Student, StudentAdmin)
```

Here, StudentAdmin customizes the admin interface for the Student model, showing only the name and age fields, adding search functionality for name, and a filter for age.

---

6. a. Develop a Django program to create feedback forms.
   b. Discuss the usage of the admin interface.
   c. How to create forms in Django? What does this class do in Python interpreter?
   (08 Marks, 06 Marks, 06 Marks)

Solution

6(a) Develop a Django program to create feedback forms.

To create a feedback form in Django, we'll follow a few steps. A feedback form generally collects data such as user name, email, feedback text, and rating. In Django, forms can be created using Django's form library by defining a form class.

Step-by-Step Program

1. **Create a Django App**
   First, make sure you have a Django project and create an app within it.

```
django-admin startproject feedback_project

cd feedback_project

python manage.py startapp feedback_app
```

2. **Define the Model for Feedback** In the models.py file of feedback_app, define a Feedback model to store the feedback information in the database.

```
# feedback_app/models.py

from django.db import models


class Feedback(models.Model):

    name = models.CharField(max_length=100)

    email = models.EmailField()

    feedback = models.TextField()

    rating = models.IntegerField(choices=[(1, 'Poor'), (2, 'Fair'), (3, 'Good'), (4, 'Very Good'), (5, 'Excellent')])


    def __str__(self):

        return f"{self.name} - {self.rating}"
```

   - name and email fields store user information.

- feedback is a TextField that can hold longer text for the user’s feedback.
- rating is an integer field that stores feedback ratings, with predefined choices from 1 to 5.

3. **Create a Form for Feedback** In forms.py file in feedback_app, define a Django form based on the Feedback model.

```
# feedback_app/forms.py
from django import forms
from .models import Feedback


class FeedbackForm(forms.ModelForm):
    class Meta:
        model = Feedback
        fields = ['name', 'email', 'feedback', 'rating']
```

- FeedbackForm is a ModelForm that automatically creates form fields corresponding to the model fields.

4. **Create a View to Handle Feedback Submission** In views.py, define a view function to render the feedback form and handle form submissions.

```
# feedback_app/views.py
from django.shortcuts import render, redirect
from .forms import FeedbackForm


def feedback_view(request):
    if request.method == 'POST':
        form = FeedbackForm(request.POST)
        if form.is_valid():
            form.save()  # Save feedback to database
            return redirect('feedback_success')
    else:
        form = FeedbackForm()
    return render(request, 'feedback_app/feedback_form.html', {'form': form})
```

- If the request method is POST, it means the form has been submitted, so we check if the form data is valid. If valid, we save the data to the database.
- If the request method is GET, we simply display an empty form.

5. **Create a Template for the Feedback Form** Create an HTML template to render the form and display it to users.

```
<!-- feedback_app/templates/feedback_app/feedback_form.html -->

<h2>Feedback Form</h2>

<form method="post">

  {% csrf_token %}

  {{ form.as_p }}

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

</form>
```

   - {{ form.as_p }} renders the form fields with basic styling.
   - {% csrf_token %} is a Django template tag that adds a CSRF token for security.

6. **Define URL Patterns** In urls.py, configure URLs to link to the feedback form view.

```
# feedback_app/urls.py

from django.urls import path

from . import views


urlpatterns = [

  path('feedback/', views.feedback_view, name='feedback_form'),

]
```

7. **Add Feedback URL to Project's URL Configuration** Link the app's URLs to the main project URL configuration.

```
# feedback_project/urls.py

from django.contrib import admin

from django.urls import include, path


urlpatterns = [

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

  path('', include('feedback_app.urls')),

]
```

This completes the basic setup for a feedback form in Django.

6(b) Discuss the Usage of the Admin Interface.

The Django Admin interface is an automatic admin backend that Django generates based on your app's models. It allows authenticated users (typically site admins) to manage the app's data directly from the web interface without writing additional code for data management.

Key Features and Usage of Django Admin Interface

1. **Model Management**
   Admin users can view, add, edit, and delete data records directly from the interface. For example, an administrator can manage all entries in the Feedback model.
2. **Customizable Interface**
   Django allows you to customize how models appear in the admin interface using the ModelAdmin class in admin.py. You can specify how fields are displayed, apply filters, and define search functionality for improved data management.

```
# feedback_app/admin.py

from django.contrib import admin

from .models import Feedback


class FeedbackAdmin(admin.ModelAdmin):

    list_display = ('name', 'email', 'rating')

    search_fields = ('name', 'email')


admin.site.register(Feedback, FeedbackAdmin)
```

   - list_display shows the specified fields on the list view in the admin.
   - search_fields adds a search box to look up data in specified fields.

3. **User and Permission Management**
   The admin interface has tools to create and manage users and groups, assign permissions, and restrict access to specific parts of the interface. This enables role-based access control.
4. **Data Validation and Consistency**
   The Django admin performs basic data validation based on model constraints, ensuring that entered data is consistent. For example, if the rating field accepts only values from 1 to 5, the admin will enforce this constraint.
5. **Inline Editing for Related Models**
   If a model has a relationship with another model, Django allows inline editing of related model data directly within the main model's admin page. This is useful when you have models with foreign key relationships.

The Django Admin interface simplifies data management for developers and administrators, reducing the need to create custom data management interfaces from scratch.

6(c) How to Create Forms in Django? What Does This Class Do in Python Interpreter?

Creating Forms in Django

In Django, forms are created using the forms module in two primary ways:

1. **Django Forms (Simple Forms)**
   Django provides a Form class for creating simple forms where each form field is manually defined. For example:

```
# feedback_app/forms.py
from django import forms


class SimpleFeedbackForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    feedback = forms.CharField(widget=forms.Textarea)
    rating = forms.ChoiceField(choices=[(1, 'Poor'), (2, 'Fair'), (3, 'Good'), (4, 'Very Good'), (5, 'Excellent')])
```

2. **Model Forms**
   Django's ModelForm class is used to create forms that are directly tied to a model. This automatically generates form fields based on model fields, making it easier to create forms for database records.

```
# feedback_app/forms.py
from django import forms
from .models import Feedback


class FeedbackForm(forms.ModelForm):
    class Meta:
        model = Feedback
        fields = ['name', 'email', 'feedback', 'rating']
```

   - FeedbackForm is linked to the Feedback model, so it will have fields corresponding to each field in the model.

What Does This Class Do in Python Interpreter?

In Python, a class is a blueprint for creating objects. When you create a form class in Django, Python's interpreter processes the class definition and creates a new type. For example, when defining FeedbackForm, Python:

1. Registers FeedbackForm as a type.
2. Creates attributes (name, email, etc.) according to the form's fields.

3. Defines methods that the form class can use, like .is_valid() to check for data validation and .save() to save data if it's a ModelForm.

When you instantiate this form class (e.g., form = FeedbackForm()), Python creates an object of this class type, allowing you to interact with it (e.g., display form fields, validate data). In a Django context, the form class thus encapsulates all functionality needed for form data handling, validation, and rendering.

Module-4

7. a. Discuss the concept of generic views of objects in Django.
   b. Explain cookies with an example.
   (10 Marks, 10 Marks)

SOLUTION

7(a) Discuss the Concept of Generic Views of Objects in Django

In Django, **Generic Views** are a type of view provided by Django's class-based views system that streamline the process of creating common views for performing tasks like displaying a list of items, retrieving details of a specific item, creating, updating, or deleting objects. These views are called "generic" because they are designed to handle common patterns, reducing the amount of code developers need to write and making Django applications easier to develop and maintain.

Why Use Generic Views?

Using generic views has several advantages:

1. **Reduces Code Duplication**: Generic views eliminate the need to write boilerplate code for repetitive tasks.
2. **Consistency**: They provide a consistent approach to handling standard actions.
3. **Efficiency**: They streamline development by providing built-in views for common actions like displaying a list of objects or a detailed view of an object.

Commonly Used Generic Views in Django

1. **ListView**: Displays a list of objects from a database.
   - Example: Displaying a list of blog posts.

```
# views.py

from django.views.generic import ListView

from .models import Post


class PostListView(ListView):

   model = Post

   template_name = 'blog/post_list.html'  # Specify your template name here
```

```
    context_object_name = 'posts'  # Name of the context variable to use in the template
```

2. **DetailView**: Displays details of a single object.
   - Example: Displaying details of a specific blog post.

```
# views.py
from django.views.generic import DetailView
from .models import Post


class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/post_detail.html'
    context_object_name = 'post'
```

3. **CreateView**: Handles the creation of a new object.
   - Example: Creating a new blog post.

```
# views.py
from django.views.generic import CreateView
from .models import Post
from .forms import PostForm


class PostCreateView(CreateView):
    model = Post
    form_class = PostForm
    template_name = 'blog/post_form.html'
    success_url = '/posts/'  # Redirect after successful creation
```

4. **UpdateView**: Handles updating an existing object.
   - Example: Editing an existing blog post.

```
# views.py
from django.views.generic import UpdateView
from .models import Post
from .forms import PostForm


class PostUpdateView(UpdateView):
    model = Post
    form_class = PostForm
    template_name = 'blog/post_form.html'
    success_url = '/posts/'
```

5. **DeleteView**: Handles deleting an existing object.
    - Example: Deleting a blog post.

```
# views.py
from django.views.generic import DeleteView
from .models import Post


class PostDeleteView(DeleteView):
    model = Post
    template_name = 'blog/post_confirm_delete.html'
    success_url = '/posts/'
```

How Generic Views Work in Django

Generic views are implemented as classes that inherit from Django's View base class or TemplateView class. Each generic view provides specific methods for handling HTTP requests, such as get, post, and put. The generic views automatically map these methods to perform actions like retrieving, creating, updating, or deleting database records.

For example:

- **ListView** internally calls get_queryset() to retrieve a list of objects from the database.
- **CreateView** and **UpdateView** automatically handle form validation and saving the data to the database.

Using generic views can significantly reduce development time, especially for common tasks. By simply subclassing these views and providing minimal configuration (like specifying the model, template, and sometimes form class), developers can create powerful views with minimal code.

7(b) Explain Cookies with an Example

Cookies are small pieces of data stored on the client's browser that can be used by web servers to track and remember information about the user's session or preferences. In Django, cookies are primarily used to store small bits of information on the client side, allowing the server to "remember" users across requests.

How Cookies Work

1. **Setting a Cookie**: When a user visits a website, the server can send a cookie to the user's browser. This cookie is stored on the client's device.
2. **Retrieving a Cookie**: On subsequent requests, the browser sends the stored cookies back to the server, allowing the server to retrieve stored information and personalize the user experience.
3. **Expiration**: Cookies can have expiration dates, after which they are automatically deleted. They can also be session-based, meaning they are deleted when the browser is closed.

Setting and Retrieving Cookies in Django

Django provides a straightforward way to work with cookies through the HttpResponse object.

1. **Setting a Cookie**
   Here's an example of setting a cookie in a Django view:

```
# views.py

from django.http import HttpResponse


def set_cookie_view(request):

   response = HttpResponse("Cookie Set")

   response.set_cookie('username', 'JohnDoe', max_age=3600)  # Expires after 1 hour

   return response
```

   - set_cookie() is used to set a cookie named username with a value of JohnDoe.
   - max_age=3600 specifies the cookie expiration time in seconds (3600 seconds = 1 hour).

2. **Retrieving a Cookie**
   Once a cookie has been set, it can be accessed in subsequent requests:

```
# views.py

from django.http import HttpResponse


def get_cookie_view(request):

   username = request.COOKIES.get('username')

   if username:

      return HttpResponse(f"Welcome back, {username}!")

   else:

      return HttpResponse("Hello, new user!")
```

   - request.COOKIES.get('username') is used to retrieve the value of the cookie named username.

    - If the cookie exists, it welcomes the user back. If not, it treats the user as a new visitor.

3. **Deleting a Cookie**
   To delete a cookie, set its expiry date to a past time or use the delete_cookie() method.

```
# views.py

from django.http import HttpResponse


def delete_cookie_view(request):

    response = HttpResponse("Cookie Deleted")

    response.delete_cookie('username')

    return response
```

    - delete_cookie('username') removes the username cookie from the client's browser.

Example: Using Cookies to Track a User's Visit Count

A common example is tracking the number of visits a user has made to the site. Here's how you could implement this in Django:

```
# views.py

from django.http import HttpResponse


def visit_count_view(request):

    visit_count = int(request.COOKIES.get('visit_count', 0)) + 1

    response = HttpResponse(f"Welcome back! You've visited {visit_count} times.")

    response.set_cookie('visit_count', visit_count, max_age=86400)  # Expires in 1 day

    return response
```

- request.COOKIES.get('visit_count', 0) retrieves the visit_count cookie. If it doesn't exist, it defaults to 0.
- visit_count + 1 increments the visit count each time the user accesses this view.
- set_cookie('visit_count', visit_count, max_age=86400) updates the cookie with the new visit count and sets it to expire in 24 hours (86400 seconds).

Each time the user visits, they'll see an updated count. This information can be useful for customizing the user experience, like showing a welcome message based on the number of visits.

Important Notes on Cookies

1. **Security**: Cookies are stored on the client's device and can be accessed or tampered with by the user. For sensitive data, avoid storing it directly in cookies or use a secure cookie setting (HttpOnly and Secure flags).

2. **Size Limitations**: Cookies have a size limit of about 4 KB, so they should be used only for small pieces of data.
3. **Session Cookies**: Django also has a session framework that uses cookies to store session IDs while storing the session data on the server. This is a secure way to manage user sessions as only the session ID is stored on the client side.

8. a. How does user authentication work in Django?
   b. Discuss the role of site map framework.
   (10 Marks, 10 Marks)

SOLUTION

8(a) How Does User Authentication Work in Django?

In Django, **user authentication** is the process of verifying the identity of a user who is attempting to access a system. Django provides a built-in authentication system that manages user accounts, passwords, permissions, and groups. It handles user login, logout, and registration in a secure manner.

The Django authentication system mainly revolves around the following components:

1. **User Model**
   - Django comes with a built-in User model (located in django.contrib.auth.models) which includes fields like username, password, email, first_name, and last_name.
   - The User model supports creating, updating, and deleting users, as well as setting permissions and managing groups.
   - Developers can either use Django's built-in User model or create a custom user model if they need additional fields or custom behavior.
2. **Password Management**
   - Django stores passwords securely by hashing them before storing them in the database, ensuring that plaintext passwords are never saved.
   - Django uses a secure hashing algorithm (PBKDF2 by default) to encrypt passwords.
   - The set_password and check_password methods in Django's User model are used to hash passwords and verify them during login.
3. **Authentication System Functions**
   - Django provides several helper functions to simplify authentication:
     - **authenticate()**: This function takes a username and password as input, checks if a user with these credentials exists, and returns the user object if the credentials are valid.
     - **login()**: After authentication, this function is used to log the user into the session, allowing them to access protected resources.

        - **logout()**: This function logs the user out of the current session, ending their authenticated session.

4. **Views for Authentication**
    - Django provides built-in views to handle common user authentication workflows, such as:
        - **LoginView**: Renders a login form and processes the user’s credentials.
        - **LogoutView**: Logs the user out and redirects them to a specified page.
        - **PasswordChangeView** and **PasswordResetView**: Allow users to change or reset their passwords.
    - These views can be customized by specifying different templates or redirect URLs.
5. **Using the Django Authentication Middleware**
    - Django's authentication system uses middleware (django.contrib.auth.middleware.AuthenticationMiddleware) to manage user sessions.
    - This middleware retrieves the currently logged-in user from the session and assigns it to request.user in each view, so the application can know if a user is authenticated or not.
6. **Permissions and Groups**
    - Django’s authentication system includes a permission framework that allows developers to assign permissions to users and groups.
    - Permissions allow specific actions like "add," "change," and "delete" on certain models.
    - Groups are used to manage sets of permissions, making it easier to assign permissions to multiple users.
7. **Example Workflow for User Authentication**

Here’s a simplified workflow of how user authentication works in Django:

- **User Login**:

1. A user submits a login form with a username and password.
2. The authenticate() function verifies the credentials.
3. If credentials are valid, the login() function logs the user in, creating a session for them.
4. The user can then access protected resources since request.user is now set to the authenticated user.

- **User Logout**:

1. When a user clicks "logout," the logout() function ends the session.
2. request.user is reset to AnonymousUser, which indicates that no user is logged in.

Example Code

Here’s an example of how to handle user login in a Django view:

```
# views.py
from django.contrib.auth import authenticate, login, logout
from django.shortcuts import render, redirect
from django.http import HttpResponse


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')
        else:
            return HttpResponse("Invalid credentials")
    return render(request, 'login.html')


def user_logout(request):
    logout(request)
    return redirect('login')
```

In this code:

- authenticate() checks the credentials.
- If the user is authenticated, login() creates a session.
- logout() clears the session, logging the user out.

8(b) Discuss the Role of the Sitemap Framework

A **sitemap** is an XML file that lists the URLs of a website along with additional metadata (e.g., when a URL was last updated, how frequently it changes, and its importance). Search engines like Google and Bing use sitemaps to index websites more effectively, improving the site’s search engine optimization (SEO).

Django’s **sitemap framework** (django.contrib.sitemaps) simplifies the process of creating and managing sitemaps for websites. It provides a way to automatically generate XML sitemaps that can be submitted to search engines.

Why Use Sitemaps?

1. **Improved SEO**: Sitemaps help search engines discover and index all pages on a website, especially those that might not be accessible via navigation.
2. **Enhanced Crawling**: Search engines can use metadata in the sitemap (e.g., update frequency and priority) to optimize the crawl rate.
3. **Better Visibility for Dynamic Sites**: Websites that frequently update content, like blogs or news sites, benefit from sitemaps, ensuring new content is indexed faster.

Creating Sitemaps in Django

The Django sitemap framework provides tools to create sitemaps easily. Here’s how it works:

1. **Defining a Sitemap Class**
    - The first step is to create a sitemap class for each set of URLs you want to include.
    - Each sitemap class must inherit from django.contrib.sitemaps.Sitemap and implement specific methods, such as items(), location(), lastmod(), changefreq, and priority.

Example of a sitemap for a blog model:

```
# sitemaps.py

from django.contrib.sitemaps import Sitemap

from .models import BlogPost


class BlogPostSitemap(Sitemap):

    changefreq = "weekly"

    priority = 0.8


    def items(self):

        return BlogPost.objects.all()


    def lastmod(self, obj):

        return obj.updated_at
```

- **changefreq**: Indicates how frequently the page is likely to change.
- **priority**: Specifies the importance of the page, relative to other pages on the site.

- **items()**: Returns a queryset of objects to include in the sitemap (e.g., all blog posts).
- **lastmod()**: Specifies the last modification date for each item.

2. **Configuring the Sitemap in URLs**
    - After defining the sitemap classes, the next step is to configure them in Django's URL configuration.
    - You use Django's sitemaps view to serve the sitemap XML.

```
# urls.py

from django.contrib.sitemaps.views import sitemap

from django.urls import path

from .sitemaps import BlogPostSitemap



sitemaps = {

    'blog': BlogPostSitemap,

}



urlpatterns = [

    # other URL patterns

    path('sitemap.xml', sitemap, {'sitemaps': sitemaps},
name='django.contrib.sitemaps.views.sitemap'),

]
```

- The sitemap view automatically generates an XML sitemap based on the specified sitemap classes.

3. **Additional Sitemap Customization**
    - You can create multiple sitemap classes for different models or types of content on your website.
    - You can add additional fields like priority to indicate the relative importance of pages.
4. **Submitting the Sitemap to Search Engines**
    - After creating the sitemap, you can submit it to search engines like Google Search Console and Bing Webmaster Tools.
    - This helps ensure that search engines crawl your website efficiently and keep their index up-to-date.

Example XML Sitemap Output

Here’s an example of what the generated XML sitemap might look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://example.com/blog/post1</loc>
    <lastmod>2024-10-25</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://example.com/blog/post2</loc>
    <lastmod>2024-10-24</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.8</priority>
  </url>
  <!-- Additional URLs -->
</urlset>
```

Benefits of Django’s Sitemap Framework

1. **Automated XML Generation**: With the Django sitemap framework, developers don’t need to manually create XML sitemaps.
2. **Dynamic Sitemaps**: The sitemap automatically updates whenever new content is added or modified.
3. **SEO-Friendly**: Sitemaps help improve SEO by allowing search engines to efficiently discover and index content.

Module-5

9. a. Explain technologies on which AJAX overlaid.
   b. Discuss about jQuery and basic Ajax.
   (10 Marks, 10 Marks)

SOLUTION

9a. Technologies on which AJAX Overlayed (10 Marks)

AJAX (Asynchronous JavaScript and XML) is a technology that allows web applications to send and receive data asynchronously without interfering with the display and behavior of the existing page. AJAX overlays several core technologies, which include:

1. **JavaScript**:
   - The backbone of AJAX, enabling asynchronous communication with the server. JavaScript provides the programming logic to send requests and handle responses.
   - It is used to manipulate the DOM (Document Object Model), allowing developers to update parts of a web page dynamically.
2. **XMLHttpRequest**:
   - A built-in browser object used to send HTTP requests and receive responses from a web server. It supports various formats, including XML, JSON, and plain text.
   - This object is critical for AJAX operations, as it enables asynchronous communication with the server without reloading the entire page.
3. **HTML/CSS**:
   - HTML structures the content of web pages, while CSS styles it. AJAX allows for partial updates of the HTML content on the page.
   - With AJAX, developers can load new content dynamically and style it without a full page refresh.
4. **Server-side Technologies**:
   - AJAX works with various server-side technologies like PHP, ASP.NET, Ruby on Rails, or Node.js. These technologies process the requests sent from the client and return appropriate responses.
   - The server can return data in various formats, such as JSON, XML, or HTML.
5. **JSON (JavaScript Object Notation)**:
   - Often used as an alternative to XML for data interchange in AJAX applications. JSON is lightweight and easy to parse in JavaScript, making it a popular choice for modern web applications.
6. **Web APIs**:
   - AJAX can be used to interact with RESTful APIs, allowing web applications to retrieve or send data to third-party services asynchronously.

9b. jQuery and Basic AJAX (10 Marks)

jQuery:

- jQuery is a fast, lightweight, and feature-rich JavaScript library that simplifies HTML document traversal and manipulation, event handling, and animation. One of its primary benefits is that it abstracts the complexity of JavaScript, making it easier to work with AJAX.

Basic AJAX with jQuery: jQuery provides several methods to make AJAX calls easily. The $.ajax() method is the most flexible and powerful, but there are also shorthand methods like $.get(), $.post(), and $.getJSON() for simpler use cases.

Example of Basic AJAX with jQuery

1. **GET Request**:

```
$.ajax({
  url: 'https://api.example.com/data',
  type: 'GET',
  success: function(response) {
    console.log(response);
    // Update the DOM with the response data
  },
  error: function(xhr, status, error) {
    console.error('Error:', error);
  }
});
```

2. **POST Request**:

```
$.ajax({
  url: 'https://api.example.com/data',
  type: 'POST',
  data: { name: 'John', age: 30 },
  success: function(response) {
    console.log('Data saved successfully:', response);
  },
  error: function(xhr, status, error) {
    console.error('Error:', error);
  }
});
```

10. a. Write a note on JavaScript and XMLHttp response.
    b. Illustrate the following:
    i) CSS
    ii) JSON

iii) HTML
iv) Iframe
(10 Marks, 10 Marks)

SOLUTION

JavaScript: JavaScript is a high-level, dynamic, and interpreted programming language that is a core technology of the World Wide Web, alongside HTML and CSS. It allows developers to create interactive and dynamic web applications. Key features of JavaScript include:

- **Client-Side Scripting**: JavaScript runs in the user's browser, enabling dynamic content updates without requiring page reloads.
- **Event Handling**: It allows developers to respond to user actions like clicks, form submissions, and keyboard input.
- **DOM Manipulation**: JavaScript can manipulate the Document Object Model (DOM), enabling changes to the content and structure of web pages.

XMLHttpRequest (XHR): XMLHttpRequest is a built-in JavaScript object that allows web applications to send HTTP requests and receive responses asynchronously without reloading the page. It is a fundamental component of AJAX (Asynchronous JavaScript and XML) and plays a crucial role in modern web development. Key features include:

- **Asynchronous Communication**: XHR enables web pages to send requests to the server and receive responses without blocking the user interface, enhancing user experience.
- **Support for Various Data Formats**: It can handle different response formats, including JSON, XML, HTML, and plain text.
- **Cross-Origin Requests**: XHR can be used to make requests to servers other than the one that served the web page, although this is subject to the same-origin policy for security reasons.

XMLHttpResponse: The response received from an XMLHttpRequest is accessible through the response property of the XHR object. Depending on the response type specified, response can be of different types:

- **responseText**: Contains the response data as a string.
- **responseXML**: Contains the response data as an XML Document if the response was XML.
- **responseJSON**: If the response is in JSON format, it can be parsed into a JavaScript object using JSON.parse(). This property is not directly available; instead, you convert responseText to JSON.

10b. Illustrations

i) CSS (Cascading Style Sheets)

CSS is a stylesheet language used to describe the presentation of a document written in HTML or XML. It controls the layout, colors, fonts, and overall appearance of web pages.

Example:

```
body {
  background-color: #f0f0f0; /* Light grey background */
  font-family: Arial, sans-serif; /* Font type */
}

h1 {
  color: #333; /* Dark grey text */
  text-align: center; /* Centered heading */
}

.button {
  background-color: #4CAF50; /* Green background */
  color: white; /* White text */
  padding: 10px 20px; /* Padding */
  border: none; /* No border */
  border-radius: 5px; /* Rounded corners */
}
```

ii) JSON (JavaScript Object Notation)

JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is often used for data exchange between a server and a web application.

Example:

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "courses": ["Mathematics", "Physics", "Computer Science"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "zipcode": "12345"
  }
}
```

iii) HTML (Hypertext Markup Language)

HTML is the standard markup language used to create web pages. It structures content on the web and is the foundation of any web application.

Example:

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Sample Page</title>

  <link rel="stylesheet" href="styles.css">

</head>

<body>

  <h1>Welcome to My Website</h1>

  <p>This is a sample web page.</p>

  <button class="button">Click Me</button>

</body>

</html>
```

iv) Iframe (Inline Frame)

An iframe is an HTML element that allows you to embed another HTML page within the current page. It is commonly used to display content from another website or to include external resources.

Example:

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>Iframe Example</title>

</head>

<body>

  <h1>Embedded Content</h1>

  <iframe src="https://www.example.com" width="600" height="400" title="Example
Website"></iframe>
```

```
</body>

</html>
```