UNIDAD 3.

Desarrollo de aplicaciones web (Back end).

(El estudiante demostrará el proceso para el desarrollo de una aplicación web.)

Tema II. Manejo de peticiones web.

Evidencia de aprendizaje Tema II –RESUMEN

Temas	Saber	Saber hacer	Resultado de Aprendizaje
Manejo de peticiones web	Identificar los métodos de comunicación entre cliente y servidor: get y post	Desarrollar Aplicaciones Web que implementen Manejo de peticiones web.	Los estudiantes identifican los métodos de intercambio de información hacia el servidor y la base de datos.
Evidencia de aprendizaje			
A partir de un caso práctico realizar una aplicación web que contenga lo siguiente: Páginas web, hojas de estilos, gestión de contenido de base de datos desde formularios, implementación de seguridad por medio de sesiones.			

Información sobre el profesor

Profesor	Correo	Días de Clase
Bernardo Prado Díaz	Tenor_prado@yahoo.com.mx	Lunes y Martes

SABER: Fundamentos teóricos del manejo de peticiones web

¿Qué es una petición web?

Una petición web es una solicitud que el cliente (navegador) envía al servidor para obtener o enviar información. Django maneja estas peticiones a través del objeto request.

Métodos HTTP principales

Método	Propósito	Características
GET	Solicita datos	Se usa para obtener información. Los datos se envían en la URL.
POST	Envía datos	Se usa para enviar información al servidor. Los datos se envían en el cuerpo de la petición.

Diferencias clave entre GET y POST

Aspecto	GET	POST
Visibilidad	Parámetros visibles en la URL	Parámetros ocultos en el cuerpo
Seguridad	Menos seguro	Más seguro para datos sensibles
Uso común	Consultas, navegación	Formularios, envío de datos
Tamaño de datos	Limitado	Sin límite práctico

¿Cómo maneja Django las peticiones?

Django recibe cada petición en una vista, que es una función o clase que procesa la solicitud y devuelve una respuesta (HttpResponse o render). El tipo de petición se identifica con:

python

```
request.method  # Devuelve 'GET' o 'POST'
```

SABER HACER: Aplicación práctica en Django

Crear una vista que maneje GET y POST

python

```
# views.py
from django.shortcuts import render

def contacto(request):
    if request.method == 'POST':  # Si el usuario envió el formulario
        nombre = request.POST.get('nombre')  # Extrae el dato del formulario
        mensaje = request.POST.get('mensaje')  # Extrae otro dato
        return render(request, 'respuesta.html', {'nombre': nombre, 'mensaje': mensaje})  # Muestra respuesta
    return render(request, 'formulario.html')  # Si es GET, muestra el formulario vacío
```

Formulario HTML para enviar datos (POST)

```
<!-- formulario.html -->
<h2>Formulario de contacto</h2>
<form method="POST">
  {% csrf_token %} <!-- Protección contra ataques CSRF -->
  <input type="text" name="nombre" placeholder="Tu nombre">
  <textarea name="mensaje" placeholder="Tu mensaje"></textarea>
  <button type="submit">Enviar</button>
</form>
```

Mostrar los datos recibidos

```
<!-- respuesta.html -->
<h2>Gracias por tu mensaje</h2>
<p>Nombre: {{ nombre }}</p>
<p>Mensaje: {{ mensaje }}</p>
```

Ejemplo de uso de GET para filtrar datos

python

```
# views.py
def buscar_producto(request):
    nombre = request.GET.get('nombre')  # Obtiene parámetro desde la URL
    productos = Producto.objects.filter(nombre__icontains=nombre)  # Filtra
productos
    return render(request, 'resultados.html', {'productos': productos})
```

html

```
<!-- formulario de búsqueda -->
<form method="GET">
  <input type="text" name="nombre" placeholder="Buscar producto">
  <button type="submit">Buscar</button>
</form>
```

Resultado de aprendizaje

Al finalizar este tema, se podrá:

- Identificar los métodos GET y POST como formas de comunicación entre cliente y servidor.
- Comprender cómo se usan en Django para recibir y procesar datos.
- Desarrollar aplicaciones web que implementen formularios, filtros y respuestas dinámicas.
- Aplicar buenas prácticas de seguridad como el uso de csrf_token.

Este contenido es opcional para complementar el tema de manejo de peticiones web en Django: validación de formularios, manejo de errores y conexión con base de datos.

Validación de formularios en Django

Django facilita la validación de datos con sus formularios integrados. Puedes validar campos automáticamente y mostrar errores al usuario.

Ejemplo con validación automática

python

```
# forms.py
from django import forms

class ContactoForm(forms.Form):
    nombre = forms.CharField(max_length=100)
    mensaje = forms.CharField(widget=forms.Textarea)

    def clean_nombre(self):
        nombre = self.cleaned_data['nombre']
        if len(nombre) < 3:
            raise forms.ValidationError("El nombre debe tener al menos 3 caracteres.")
        return nombre
```

Vista que maneja el formulario

python

```
# views.py
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
```

```
        form = ContactoForm(request.POST)
        if form.is_valid():
            nombre = form.cleaned_data['nombre']
            mensaje = form.cleaned_data['mensaje']
            # Aquí podrías guardar en la base de datos
            return render(request, 'respuesta.html', {'nombre': nombre,
'mensaje': mensaje})
    else:
        form = ContactoForm()
    return render(request, 'formulario.html', {'form': form})
```

Plantilla con errores

html

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

{% if form.errors %}
  <ul>
    {% for field in form %}
      {% for error in field.errors %}
        <li>{{ error }}</li>
      {% endfor %}
    {% endfor %}
  </ul>
{% endif %}
```

Manejo de errores en vistas

Puedes capturar errores con bloques try/except para evitar que tu aplicación se caiga.

python

```
def buscar_producto(request):
    try:
        nombre = request.GET.get('nombre')
        productos = Producto.objects.filter(nombre__icontains=nombre)
    except Exception as e:
        productos = []
        print("Error:", e)
    return render(request, 'resultados.html', {'productos': productos})
```

Conexión con base de datos (integración con peticiones)

Cuando usas formularios para enviar datos (POST), puedes guardarlos directamente en la base de datos usando modelos.

python

```
# models.py
class Mensaje(models.Model):
    nombre = models.CharField(max_length=100)
    contenido = models.TextField()
```

python

```
# views.py
from .models import Mensaje

def contacto(request):
    if request.method == 'POST':
        form = ContactoForm(request.POST)
        if form.is_valid():
            Mensaje.objects.create(
                nombre=form.cleaned_data['nombre'],
                contenido=form.cleaned_data['mensaje']
```

```
            )
            return redirect('mensaje_enviado')
    else:
        form = ContactoForm()
    return render(request, 'formulario.html', {'form': form})
```

Resultado de aprendizaje ampliado

- Validar entradas del usuario de forma segura y clara
- Manejar errores sin afectar la experiencia del usuario
- Integrar peticiones web con almacenamiento en base de datos
- Diseñar aplicaciones robustas y funcionales con Django

Desarrollando una app en Flutter y conectarla con un backend en Django, lo ideal es usar el Django REST Framework (DRF) para crear APIs RESTful.

Paso 1: Crear tu API REST con Django

Instala Django y DRF

bash

```
pip install django djangorestframework
```

Agrega 'rest_framework' a tu archivo settings.py en INSTALLED_APPS.

Define tu modelo

python

```
# models.py
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
```

Crea el serializer

python

```
# serializers.py
from rest_framework import serializers
from .models import Producto

class ProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Producto
        fields = '__all__'
```

Define las vistas

python

```
# views.py
from rest_framework import viewsets
```

```
from .models import Producto
from .serializers import ProductoSerializer

class ProductoViewSet(viewsets.ModelViewSet):
    queryset = Producto.objects.all()
    serializer_class = ProductoSerializer
```

Configura las URLs

python

```
# urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import ProductoViewSet

router = DefaultRouter()
router.register(r'productos', ProductoViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

Ahora tu API estará disponible en http://localhost:8000/api/productos/.

Paso 2: Conectar desde Flutter

Usa el paquete http para hacer peticiones a tu API.

Ejemplo básico en Flutter

dart

```
import 'package:http/http.dart' as http;
import 'dart:convert';

Future<void> obtenerProductos() async {
  final response = await
http.get(Uri.parse('http://localhost:8000/api/productos/'));
```

```
  if (response.statusCode == 200) {
    List productos = json.decode(response.body);
    print(productos);
  } else {
    throw Exception('Error al cargar productos');
  }
}
```

Manejo de Peticiones Web con Python y Django

1 SABER – Dimensión Conceptual

Objetivo: Comprender los métodos de comunicación entre cliente y servidor (GET y POST) y su uso en el desarrollo de aplicaciones web con Django.

A) Conceptos Clave

• Cliente:
Definición: Aplicación (generalmente un navegador) que solicita recursos o servicios al servidor.
Ejemplo: Ejemplo: Google Chrome solicitando una página web.

• Servidor:
Definición: Equipo o programa que recibe solicitudes y envía respuestas.
Ejemplo: Ejemplo: Servidor Django respondiendo con una página HTML.

• HTTP:
Definición: Protocolo de comunicación entre cliente y servidor.
Ejemplo: Ejemplo: Navegador usando HTTP para solicitar un documento.

• Método GET:
Definición: Método HTTP para solicitar datos del servidor. Los parámetros se envían en la URL.
Ejemplo: Ejemplo: Buscar en Google: https://google.com/search?q=python

• Método POST:
Definición: Método HTTP para enviar datos al servidor. Los parámetros se envían en el cuerpo de la solicitud.
Ejemplo: Ejemplo: Enviar un formulario de registro en Django.

2 SABER HACER – Dimensión Actuacional

Objetivo: Implementar aplicaciones en Django que utilicen métodos GET y POST.

A) Ejemplo con Método GET

```
# archivo views.py
from django.http import HttpResponse

def saludo(request):
    nombre = request.GET.get('nombre', 'Invitado')
    return HttpResponse(f"Hola {nombre}")


# URL: http://localhost:8000/saludo?nombre=Juan
```

B) Ejemplo con Método POST

```
# archivo views.py
from django.shortcuts import render

def formulario(request):
    if request.method == 'POST':
        nombre = request.POST['nombre']
        return render(request, 'respuesta.html', {'nombre': nombre})
    return render(request, 'formulario.html')



<!-- archivo templates/formulario.html -->
<form method="POST">
    {% csrf_token %}
    <input type="text" name="nombre" placeholder="Escribe tu nombre">
    <button type="submit">Enviar</button>
</form>



<!-- archivo templates/respuesta.html -->
<p>Hola {{ nombre }}, has enviado el formulario correctamente.</p>
```

3 SER Y CONVIVIR – Dimensión Socioafectiva

Objetivo: Fomentar el trabajo en equipo en el desarrollo de aplicaciones web y la aplicación del razonamiento lógico para la comunicación cliente-servidor.

- Definir en equipo la estructura de datos a enviar y recibir.

- Probar en conjunto las funciones GET y POST.

- Usar control de versiones para integrar cambios.

- Mantener comunicación constante entre los roles de back-end y front-end.

4 Glosario

- Cliente → Aplicación que solicita recursos al servidor.

- Servidor → Programa que procesa las solicitudes del cliente.

- HTTP → Protocolo de transferencia de hipertexto.

- GET → Método HTTP para solicitar datos.

- POST → Método HTTP para enviar datos.

- CSRF Token → Medida de seguridad en Django para formularios POST.