13.- TIPS DE SEGURIDAD

Manual de Programación Segura en Django

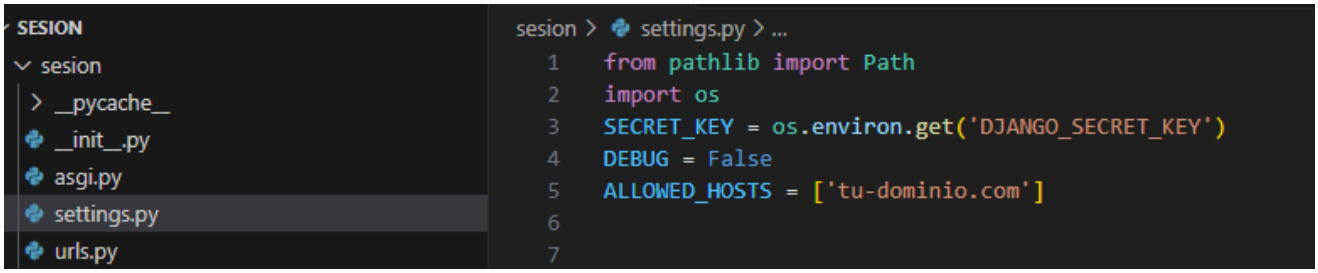
1. Actualiza Django y Dependencias Regularmente

Mantén tu entorno de desarrollo actualizado con las últimas versiones de Django y sus dependencias. Las actualizaciones suelen incluir correcciones de seguridad.

pip install --upgrade django

2. Configuración de Configuración Segura

Asegúrate de que la configuración de tu aplicación Django esté configurada de manera segura. Utiliza variables de entorno para configurar valores sensibles como claves secretas y credenciales de base de datos.

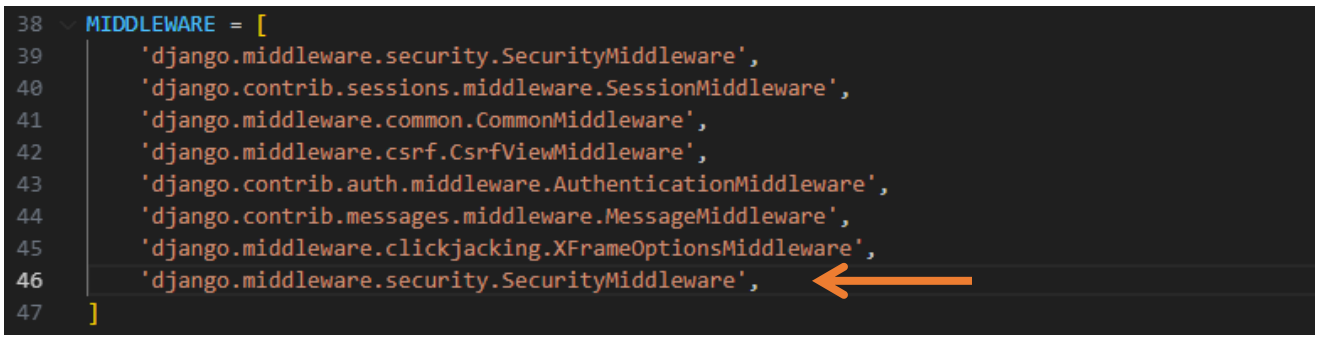


Python

```
import os
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY')
DEBUG = False
ALLOWED_HOSTS = ['tu-dominio.com']
```

3. Usa el Middleware de Seguridad de Django

Django proporciona middleware de seguridad para ayudar a proteger tu aplicación contra diversas amenazas. Asegúrate de tener activado el middleware de seguridad en tu configuración.

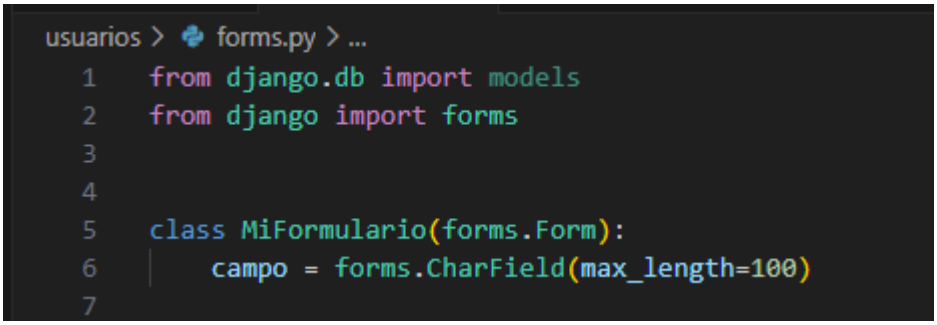


4. Configura HTTPS

Utiliza siempre HTTPS en producción para cifrar la comunicación entre el navegador y tu servidor. Configura tu servidor web para redirigir todas las solicitudes HTTP a HTTPS.

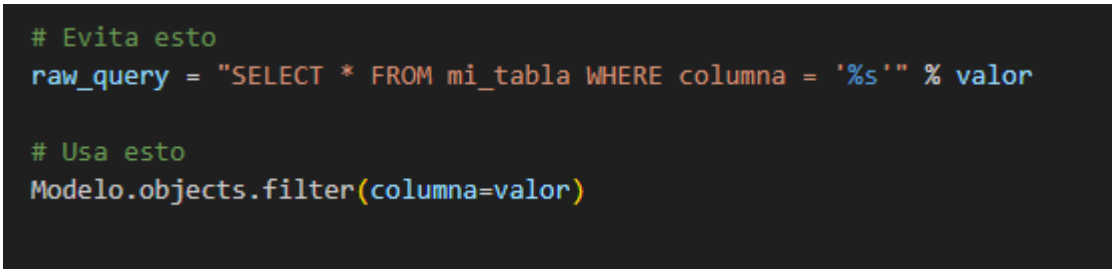
5. Validación de Formularios

Realiza una validación adecuada en el lado del servidor para todos los formularios. Usa el sistema de formularios de Django y las validaciones incorporadas para prevenir la introducción de datos maliciosos.



6. Evita la Inyección de SQL

Utiliza el **ORM de Django** y **consultas parametrizadas** para evitar la inyección de SQL



7. Protección contra CSRF

Django tiene protección **CSRF** habilitada de forma predeterminada. Asegúrate de que tus formularios están envueltos en el **token CSRF** y que las vistas requieren la validación del token.

8. Autenticación y Autorización

Utiliza el **sistema de autenticación y autorización de Django**. Implementa **roles y permisos** adecuados para limitar el acceso a las funciones según la autenticación del usuario.

9. Gestión de Sesiones Segura

Configura la gestión de sesiones de Django de manera segura. Usa configuraciones como **SESSION_COOKIE_SECURE** y **SESSION_COOKIE_HTTPONLY** para mejorar la seguridad.

10. Monitoreo de Seguridad

Implementa **monitoreo y registros de seguridad**. Utiliza herramientas como **Django Debug Toolbar** y **django-axes** para detectar y prevenir intentos de acceso no autorizado.

11. Pruebas de Seguridad

Realiza **pruebas de seguridad periódicas** utilizando herramientas como **OWASP ZAP** o **bandit** para analizar posibles vulnerabilidades en tu código.

12. Configuración de CORS

Si tu aplicación Django sirve recursos a través de diferentes dominios, configura **Cross-Origin Resource Sharing (CORS)** de manera adecuada para controlar qué dominios pueden acceder a tus recursos.

Instala la biblioteca **django-cors-headers**:

Python

```
pip install django-cors-headers
```

Agrega **'corsheaders.middleware.CorsMiddleware'** a la lista de **middleware** en tu archivo de configuración.

python Copy code

```python
# settings.py

INSTALLED_APPS = [
    # ...
    'corsheaders',
    # ...
]

MIDDLEWARE = [
    # ...
    'corsheaders.middleware.CorsMiddleware',
    # ...
]

CORS_ALLOWED_ORIGINS = [
    "https://tu-frontend.com",
    # Otros dominios permitidos
]
```

13. Configuración de HSTS

Habilita **HTTP Strict Transport Security (HSTS)** para forzar conexiones seguras a través de **HTTPS** durante un período de tiempo específico. Esto ayuda a prevenir ataques de intercepción de red.

sesion > settings.py > ...

```python
from pathlib import Path
import os

SECURE_HSTS_SECONDS = 31536000  # 1 año
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True

```

Python

SECURE_HSTS_SECONDS = 31536000 # 1 año
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True

14. Configuración de Cabeceras de Seguridad

Configura las **cabeceras de seguridad** para evitar ataques como el "**Clickjacking**" y la ejecución de scripts maliciosos. Usa el **middleware django-csp** para políticas de seguridad de contenido

bash Copy code

```
pip install django-csp
```

python Copy code

```
# settings.py

MIDDLEWARE = [
    # ...
    'csp.middleware.CSPMiddleware',
    # ...
]

# Configuración de políticas de seguridad de contenido (ejemplo)
CSP_DEFAULT_SRC = ("'self'",)
```

15. Actualizaciones Automáticas de Dependencias

Implementa herramientas como **safety** y **bandit** en tus procesos de construcción y despliegue para analizar y alertar sobre vulnerabilidades conocidas en tus dependencias.

Python

pip install safety bandit

16. Protección contra Inyecciones de Código

Evita evaluar código dinámicamente y utiliza funciones seguras de la biblioteca estándar de Python. Por ejemplo, si estás trabajando con consultas dinámicas, utiliza el método filter() de Django en lugar de construir cadenas SQL manualmente.

15. Actualizaciones Automáticas de Dependencias

Implementa herramientas como `safety` y `bandit` en tus procesos de construcción y despliegue para analizar y alertar sobre vulnerabilidades conocidas en tus dependencias.

bash Copy code

```
pip install safety bandit
```

16. Protección contra Inyecciones de Código

Evita evaluar código dinámicamente y utiliza funciones seguras de la biblioteca estándar de Python. Por ejemplo, si estás trabajando con consultas dinámicas, utiliza el método `filter()` de Django en lugar de construir cadenas SQL manualmente.

17. Configuración de Tiempo de Sesión

Ajusta la configuración de tiempo de sesión según tus necesidades de seguridad. Define períodos de expiración para las sesiones y los tokens de autenticación.

python Copy code

```python
# settings.py

SESSION_COOKIE_AGE = 1209600  # 2 semanas en segundos
```

18. Protección contra Ataques de Fuerza Bruta

Usa `**django-axes**` para protegerte contra ataques de fuerza bruta. Esto bloquea automáticamente las direcciones IP que realizan intentos repetidos de inicio de sesión incorrectos.

bash Copy code

```
pip install django-axes
```

python Copy code

```
# settings.py

INSTALLED_APPS = [
    # ...
    'axes',
    # ...
]
```

19. Configuración de Seguridad de la Base de Datos

Asegúrate de que tu base de datos esté configurada de manera segura. Utiliza cuentas con permisos mínimos necesarios y cifrado si es posible.

20. Auditoría de Seguridad

Implementa auditorías de seguridad para registrar eventos importantes. Esto puede ayudarte a rastrear y analizar actividades sospechosas.

Otras configuraciones de programación

1. Configuración de Seguridad Básica

- **Modo de Depuración**:
  - Es importante asegurarse de que DEBUG = False en el entorno de producción para evitar que se muestren mensajes de error detallados que puedan revelar información sensible.
  - Configurar ALLOWED_HOSTS para permitir solo los dominios legítimos de la aplicación.
- **Variables de Entorno**:
  - Se recomienda almacenar información sensible, como claves secretas, credenciales de base de datos y claves de API, en variables de entorno en lugar de incluirlas en el código fuente.
  - Utilizar la biblioteca django-environ o un archivo .env para gestionar estas variables de manera segura.

2. Gestión de la Autenticación

- **Contraseñas**:
  - Configurar **AUTH_PASSWORD_VALIDATORS** para imponer reglas de complejidad en las contraseñas, incluyendo longitud mínima y caracteres especiales.
  - Emplear el sistema de autenticación de Django para gestionar las contraseñas, ya que utiliza un sistema de hashing seguro (PBKDF2) por defecto.
- **Limitación de Intentos de Inicio de Sesión**:
  - Implementar una limitación de intentos de inicio de sesión mediante la biblioteca **django-axes** o un sistema propio para bloquear temporalmente las cuentas después de varios intentos fallidos.
- **Autenticación de Dos Factores (2FA)**:
  - Considerar la implementación de autenticación de dos factores utilizando django-otp o django-two-factor-auth para mayor seguridad en el inicio de sesión.

3. Protección Contra Inyección de Código

- **Validación y Escapado de Entradas**:
    - Aprovechar el sistema de plantillas de Django, que escapa automáticamente las variables en las plantillas, para prevenir inyecciones de HTML y JavaScript (XSS).
    - Escapar cualquier dato que se muestre en las plantillas y provenga de usuarios o de fuentes externas.
- **Uso de ORM**:
    - Usar el ORM de Django en lugar de escribir consultas SQL crudas, lo cual ayuda a evitar inyecciones SQL.
    - En caso de requerir consultas personalizadas, utilizar params en raw para protegerse contra inyecciones SQL.

4. Protección Contra Ataques de Cross-Site Scripting (XSS)

- **Escapar Contenido HTML**:
    - Utilizar el filtro **{{ variable|safe }}** únicamente cuando se esté seguro de que el contenido es seguro. Evitar usarlo con datos proporcionados por el usuario.
- **CSRF (Cross-Site Request Forgery)**:
    - Verificar que el middleware **CsrfViewMiddleware** esté habilitado (por defecto en Django) para proteger las solicitudes POST contra CSRF.
    - Incluir **{% csrf_token %}** en todos los formularios HTML en las plantillas para generar tokens únicos de protección CSRF.

5. Control de Acceso y Autorización

- **Permisos de Usuario**:
    - Definir roles y permisos específicos en la aplicación mediante los modelos Group y Permission de Django.
    - Usar decoradores como @login_required y @permission_required para restringir el acceso a las vistas de acuerdo con el rol del usuario.

- **Protección de Vistas**:
    - Aplicar @login_required en todas las vistas que deban ser accesibles solo para usuarios autenticados.
    - En vistas basadas en clases, aplicar LoginRequiredMixin y PermissionRequiredMixin.

6. Configuración de Seguridad en HTTPS

- **HTTPS**:
    - Usar HTTPS para cifrar la comunicación entre el servidor y los usuarios, especialmente si se manejan datos sensibles.
    - Configurar **SECURE_SSL_REDIRECT** = True para redirigir automáticamente a HTTPS.
- **Seguridad de Cookies**:
    - Configurar SESSION_COOKIE_SECURE = True y CSRF_COOKIE_SECURE = True para que las cookies solo se envíen a través de HTTPS.
    - Activar SESSION_COOKIE_HTTPONLY = True para que las cookies de sesión no sean accesibles desde JavaScript.
- **HSTS (HTTP Strict Transport Security)**:
    - Configurar SECURE_HSTS_SECONDS para habilitar HSTS y forzar HTTPS en los navegadores. Por ejemplo, SECURE_HSTS_SECONDS = 31536000 habilita HSTS por un año.

7. Protección Contra Clickjacking

- **X-Frame-Options**:
    - Configurar X_FRAME_OPTIONS = 'DENY' en settings.py para evitar que la aplicación se muestre en un iframe, previniendo así ataques de clickjacking.

8. Registro y Monitoreo

- **Logging**:
    - Configurar el sistema de registro (logging) en settings.py para registrar errores y eventos de seguridad importantes.
    - Evitar registrar datos sensibles, como contraseñas y tokens de acceso.

- **Monitoreo de Seguridad**:
    - Utilizar herramientas de monitoreo como Sentry para rastrear errores en tiempo real y responder rápidamente a incidentes de seguridad.
    - Configurar alertas para detectar actividades inusuales o errores críticos en la aplicación.

9. Escaneo de Vulnerabilidades

- **Herramientas de Análisis de Seguridad**:
    - Emplear herramientas de escaneo de seguridad como Bandit para identificar vulnerabilidades en el código Python.
    - Considerar la integración de análisis estático de código en el proceso de CI/CD para detectar problemas de seguridad en el código.
- **Actualización de Dependencias**:
    - Mantener las dependencias de Django y otros paquetes actualizadas, ya que las versiones antiguas pueden contener vulnerabilidades.

10. Seguridad de la API

- **Protección con Token**:
    - Para las APIs, usar autenticación por tokens (como JWT o TokenAuthentication de Django REST Framework).
    - Evitar pasar tokens de autenticación en URLs; en su lugar, enviar tokens a través de encabezados HTTP.
- **Permisos en API**:
    - Definir permisos específicos para las API, como IsAuthenticated, IsAdminUser, o permisos personalizados en Django REST Framework para asegurar que solo usuarios autorizados puedan acceder.
- **Límites de Tasa de Peticiones (Rate Limiting)**:
    - Aplicar limitación de tasa para evitar abuso de la API, especialmente en endpoints críticos. Django REST Framework permite configurar rate limiting en endpoints específicos.

11. Sanitización y Validación de Datos

- **Validación de Formularios y Datos de Entrada**:
    - Utilizar ModelForm y validación de modelos para asegurar que los datos ingresados por los usuarios cumplan con los requisitos del modelo.
    - Implementar validaciones personalizadas para datos que necesiten reglas adicionales, como restricciones específicas en ciertos campos.
- **Sanitización de HTML**:
    - Si es necesario permitir entradas HTML, utilizar una biblioteca de sanitización como bleach para limpiar cualquier contenido potencialmente peligroso.