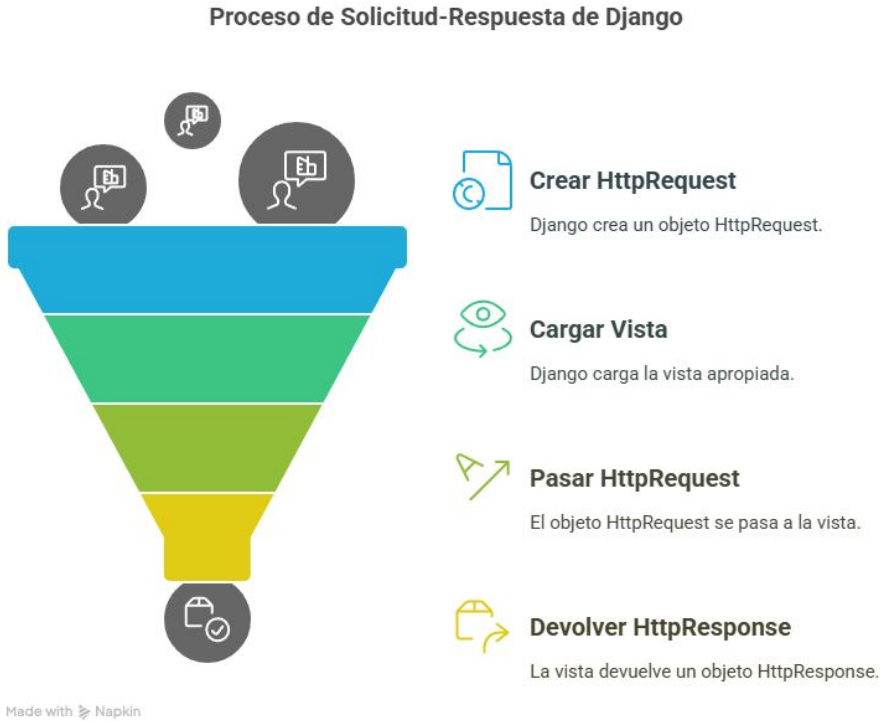
Django

Desencadenado

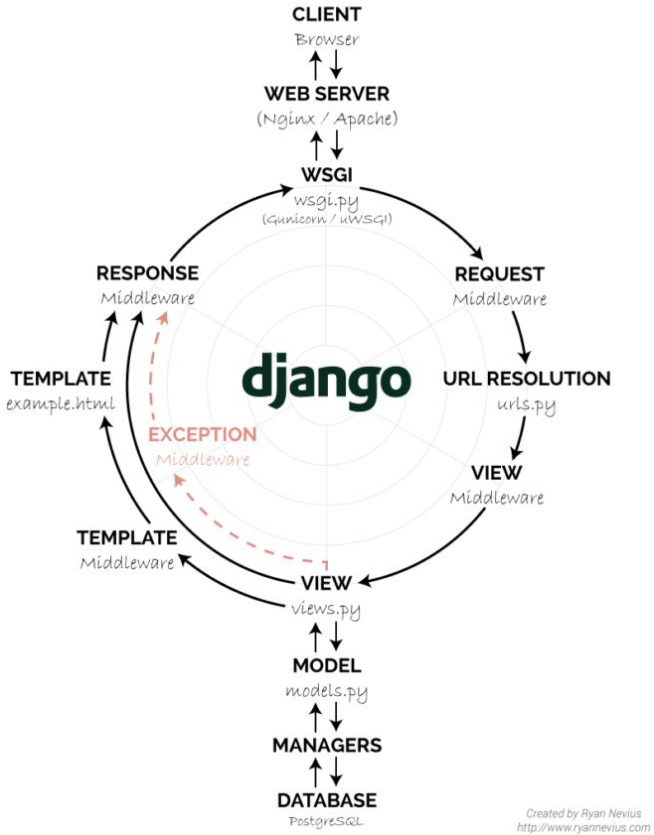
Request and response objects

Cuando se solicita una página, Django crea un objeto `HttpRequest` que contiene metadatos sobre la solicitud. Luego, Django carga la vista apropiada, pasando el `HttpRequest` como el primer argumento a la función de la vista. Cada vista es responsable de devolver un objeto `HttpResponse`.

https://django.readthedocs.io/en/stable/ref/request-response.html



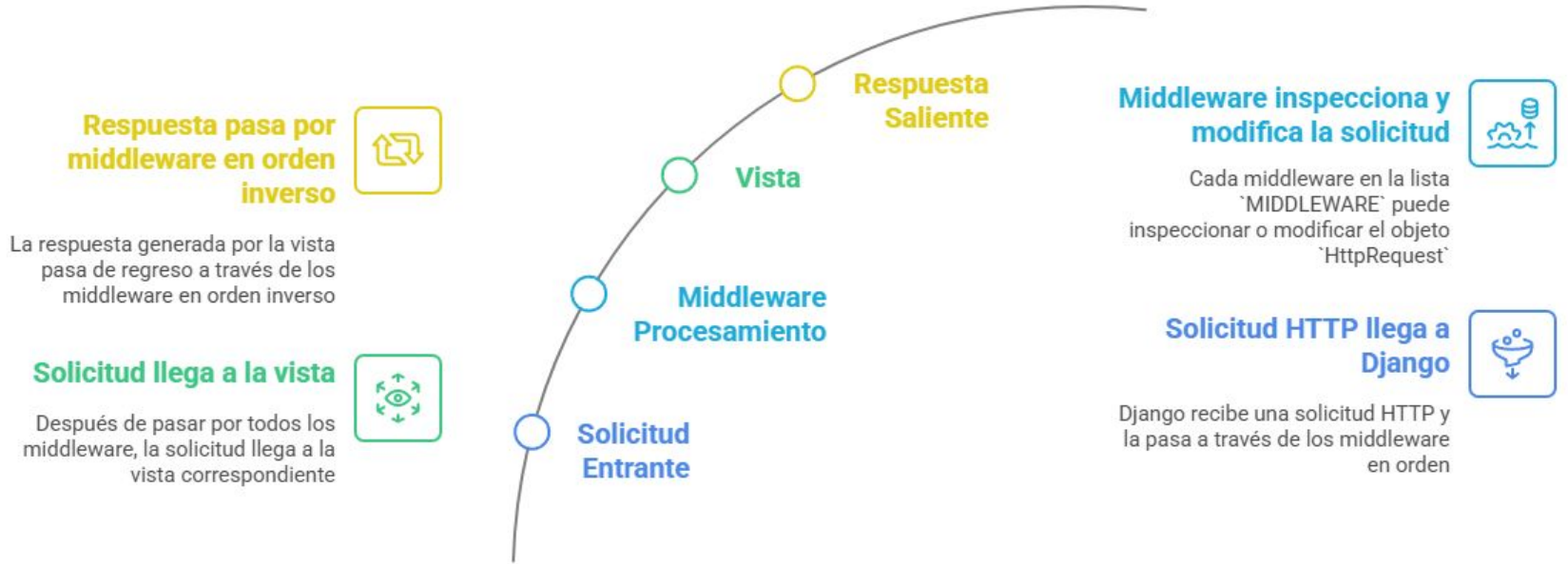
django
Request-Response Cycle



Created by Ryan Nevius
http://www.ryannevius.com

Los Middleware en Django

Un **middleware** en Django es una capa intermedia que se ejecuta entre la solicitud (request) del cliente y la respuesta (response) del servidor. Es una forma de procesar las peticiones entrantes y las respuestas salientes de manera centralizada, antes o después de que lleguen a la vista o salgan de ella.



Algunos de los atributos y métodos de los objetos `HttpRequest` son:

- **`HttpRequest.path`** (`request.path` en nuestra función de vista), que nos da la **ruta** solicitada, por ejemplo: `/profile/`.

- **`HttpRequest.method`** (`request.method` en nuestra función de vista), que nos da el **método HTTP en mayúsculas**, como: `GET`, `POST`, `HEAD`.

- **`HttpRequest.GET`** (`request.GET` en nuestra función de vista), que nos da **todos los parámetros enviados por GET** en un objeto similar a un diccionario.

- **`HttpRequest.POST`** (`request.POST` en nuestra función de vista), que nos da **todos los parámetros enviados por POST** en un objeto similar a un diccionario.

- **`HttpRequest.FILES`** (`request.FILES` en nuestra función de vista), que nos da **todos los archivos subidos** desde un formulario HTML en un objeto similar a un diccionario.

- **`HttpRequest.headers`** (`request.headers` en nuestra función de vista), que nos da **todas las cabeceras HTTP** como un diccionario.

- **`HttpRequest.user`** (`request.user` en nuestra función de vista), que nos da el **usuario actualmente autenticado**. Este atributo es asignado por el middleware. Veremos más sobre esto en una sección posterior de esta serie.

Más adelante lo usaremos en el ejercicio…

También incluye algunos métodos útiles como:

- `HttpRequest.get_host()`
- `HttpRequest.get_port()`
- `HttpRequest.is_secure()` (para saber si la conexión es por HTTPS o no), que podés probar y experimentar por tu cuenta.

Más adelante lo usaremos en el ejercicio…

El objeto `HttpResponse`

Aprendimos sobre el objeto `HttpRequest`, que es **creado automáticamente por Django** y se pasa a la función de vista. Pero, por otro lado, **es nuestra responsabilidad como programadores devolver un objeto `HttpResponse` desde la función de vista**.

Un protocolo de red

Es un conjunto de reglas y estándares que permiten la comunicación entre dispositivos en una red. Define cómo deben enviarse, recibir e interpretar los datos. Ejemplos comunes son **TCP/IP**, **HTTP**, **FTP**, **DNS**, etc.

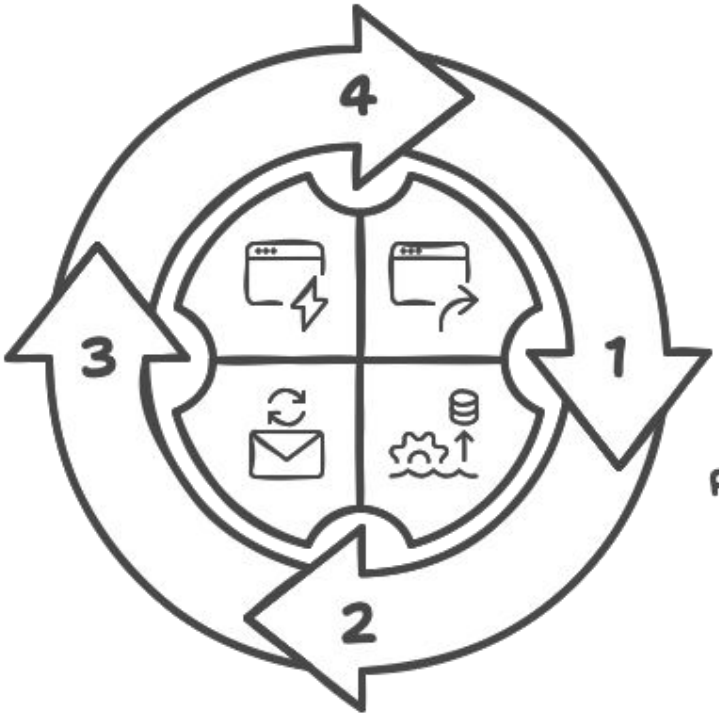
¿Qué es HTTP?

HTTP (HyperText Transfer Protocol) es un protocolo de red utilizado para la transferencia de información en la web. Permite que los navegadores soliciten páginas web y que los servidores las entreguen.

¿Qué es HTTPS?

HTTPS (HyperText Transfer Protocol Secure) es la versión segura de HTTP. Utiliza **cifrado TLS/SSL** para proteger la información que se transmite entre el navegador y el servidor.

Ciclo de Request-Response HTTP



Petición

El navegador inicia una petición al servidor.

Procesamiento

El servidor procesa la petición.

Respuesta

El servidor envía una respuesta al navegador.

Visualización

El navegador muestra o usa la respuesta.

Mensajes de solicitud (HTTP Request)

Son enviados **desde el cliente al servidor** (por ejemplo, desde tu navegador al sitio web). Incluyen:

Los Método HTTP más comunes son:

Método	Descripción
GET	**Solicita datos del servidor (como una página o imagen).**
POST	**Envía datos al servidor (como un formulario).**
PUT	**Envía datos para actualizar o reemplazar recursos.**
PATCH	**Modifica parcialmente un recurso existente.**
DELETE	**Elimina un recurso del servidor.**
HEAD	**Como GET, pero solo devuelve los encabezados.**
OPTIONS	**Consulta los métodos soportados por el servidor.**

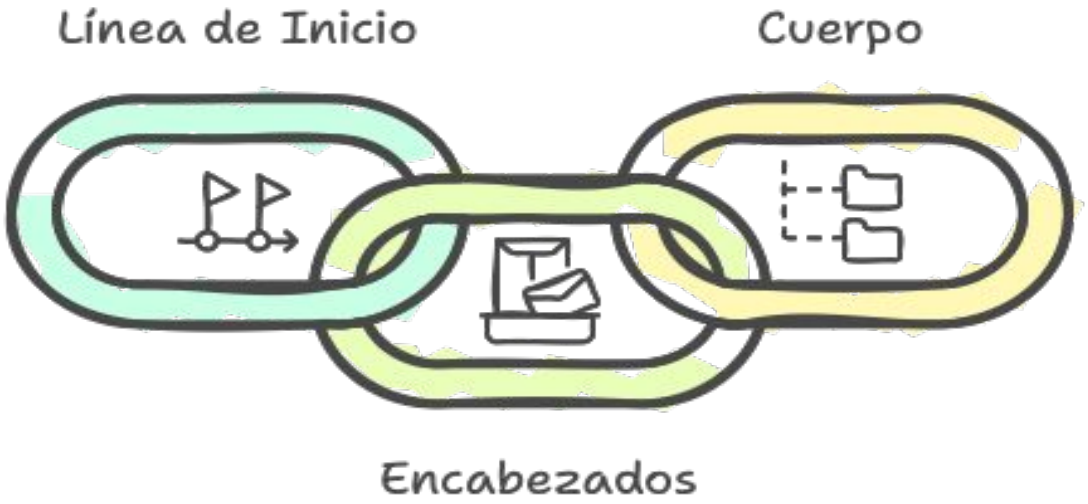
Un recurso en la red

En el contexto de redes (y especialmente en HTTP), un **recurso** es **cualquier cosa que puede ser identificada y accedida mediante una URL**. Es la **unidad de información** que un cliente (como un navegador) solicita al servidor. Ejemplos de recursos en la web:

https://ejemplo.com/index.html	**Una página**
https://ejemplo.com/logo.png	**Una imagen**
https://api.ejemplo.com/usuarios/1	**Un objeto de tipo usuario en una API**
https://ejemplo.com/videos/video.mp4	**Un archivo de video**
https://ejemplo.com/style.css	**Una hoja de estilos (CSS)**

En HTTP, tanto las solicitudes (**request**) como las respuestas (**response**) están compuestas por **tres partes principales**:

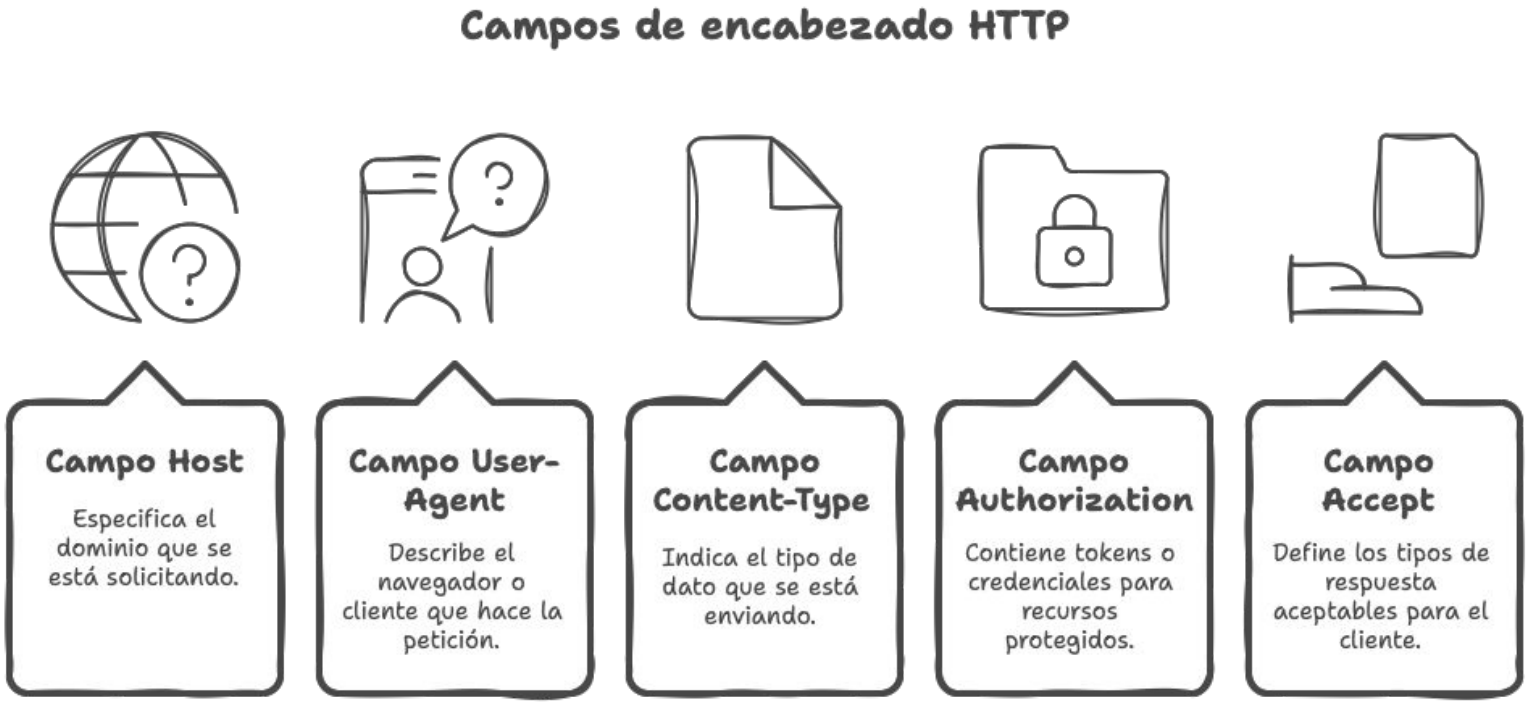
1. **Línea de inicio** (como el método o el código de estado)
2. **Encabezados (headers)**
3. **Cuerpo (body)**



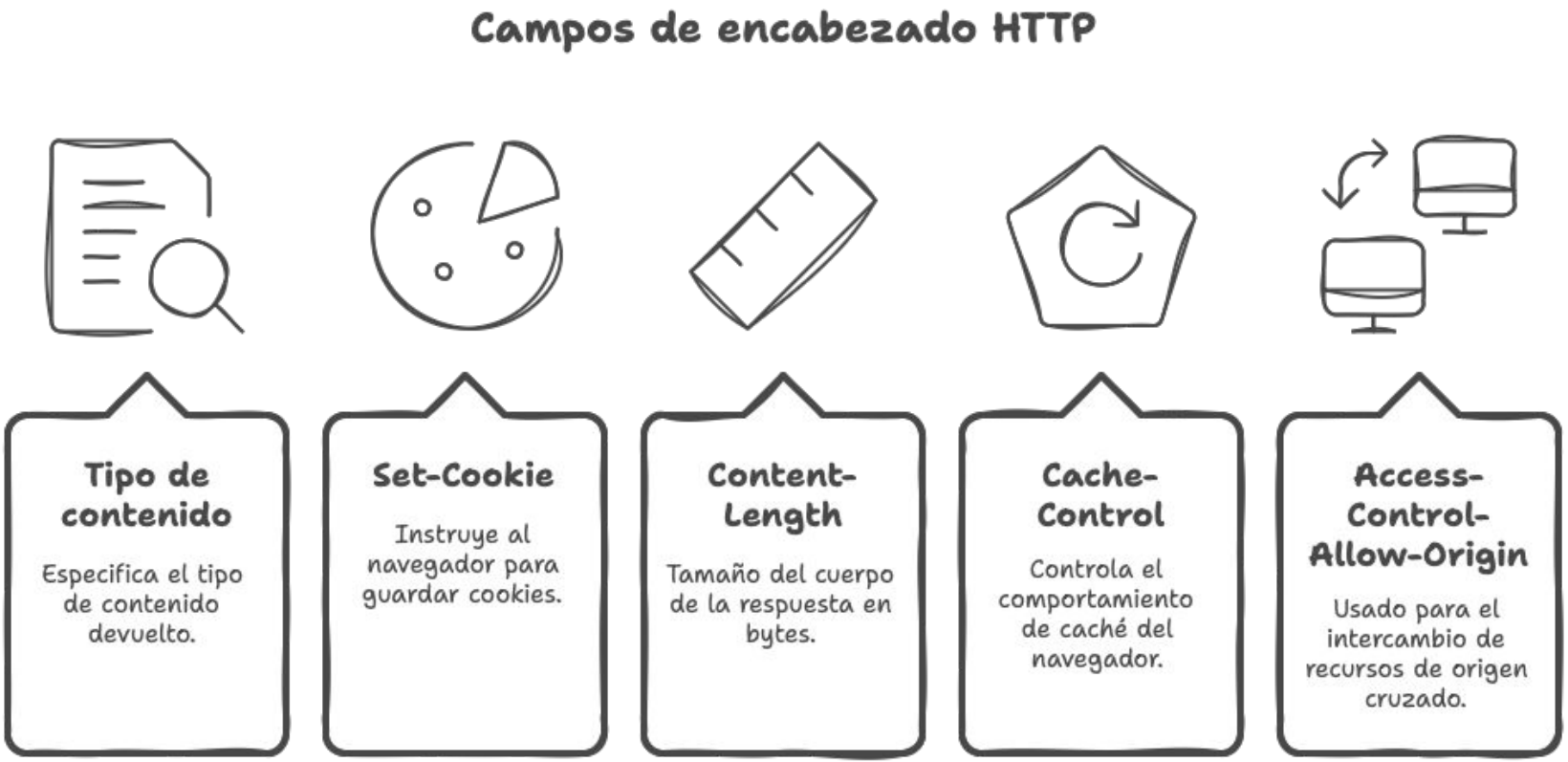
¿Qué son los encabezados (headers) HTTP?

Los **headers** son líneas de texto con información adicional sobre la solicitud o la respuesta. Se escriben como pares `Clave: Valor`.

Ejemplos comunes en una solicitud:



Ejemplos en una respuesta de header:



¿Qué es el cuerpo (body) HTTP?

El **body** es la parte del mensaje que **contiene los datos reales** que se envían o reciben. Solo se utiliza cuando es requerido.

En solicitudes:

- Se usa en métodos como `POST`, `PUT` o `PATCH`.
- Contiene datos como formularios, archivos o JSON.

¿Qué son los códigos de estado HTTP?

Los códigos de estado son unos números emitidos por el servidor en respuesta a una solicitud del cliente. Los códigos de estado se dividen en cinco grupos, de la siguiente manera, donde el primer dígito se refiere a la categoría de la respuesta:

- **1XX** (respuesta informativa)
- **2XX** (éxito)
- **3XX** (redirección)
- **4XX** (error del cliente)
- **5XX** (error del servidor)

Los Json

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos, fácil de leer y escribir para los humanos, y fácil de analizar y generar para las máquinas. Se utiliza ampliamente para enviar datos entre un servidor y una aplicación web, aunque también se usa en muchas otras aplicaciones.

Características principales:

- Basado en texto: Es simplemente texto con una estructura específica.
- Formato de clave-valor: Los datos se presentan en pares como "clave": "valor".
- Estructuras comunes: Soporta objetos ({}), listas ([]), cadenas de texto, números, valores booleanos (true, false) y null.

Por ejemplo:

```
{
  "nombre": "Ana",
  "edad": 30,
  "activo": true,
  "hobbies": ["leer", "viajar",
"música"]
}
```

ASGI

ASGI nació como una evolución de WSGI (Web Server Gateway Interface), la interfaz estándar para aplicaciones web síncronas en Python, ASGI es la interfaz moderna y asíncrona que permite construir aplicaciones web de Python de alto rendimiento y en tiempo real, utilizando servidores web y frameworks que aprovechan las capacidades de la programación asíncrona.

¿Qué lado del puente ASGI es más crucial para el desarrollo web asíncrono?

Servidores Web Asíncronos

Manejan múltiples peticiones simultáneamente

Aplicaciones Web Asíncronas

Aprovechan las capacidades asíncronas de Python

ASGI son las siglas de Asynchronous Server Gateway Interface, en un lado del puente están los servidores web asíncronos (**como Uvicorn, Daphne, Hypercorn**). Estos servidores son capaces de manejar múltiples peticiones simultáneamente de manera eficiente, sin bloquearse mientras esperan operaciones de entrada/salida (E/S). En el otro lado del puente están las aplicaciones y frameworks web asíncronos de Python (como FastAPI, Starlette, Django Channels). Estas aplicaciones están diseñadas para aprovechar las capacidades asíncronas de Python (con async y await).

ASGI:

- Es una especificación de interfaz para aplicaciones web asíncronas en Python.
- Define cómo un servidor web asíncrono se comunica con una aplicación o framework web asíncrono.
- Su principal objetivo es permitir la construcción de aplicaciones web de alto rendimiento y en tiempo real en Python, soportando protocolos como HTTP y WebSockets de manera eficiente.
- Se centra en la comunicación entre el servidor y la aplicación dentro de un único proceso o conjunto de procesos.

¿Qué podrían hacer los desarrolladores para mitigar ataques de DOS en Django (Uvicorn)?

- Limitación de tamaño de las peticiones: Configurar límites en el tamaño de los archivos que se pueden cargar o enviar en las peticiones.
- Timeouts configurados: Establecer timeouts razonables para las peticiones, pero no tan largos como para agotar los recursos del servidor si una petición maliciosa es muy lenta.
- Implementación de Rate Limiting: Limitar el número de peticiones que un cliente puede hacer en un período de tiempo determinado. Esto puede ayudar a prevenir ataques DoS que envían una gran cantidad de peticiones.
- Uso de Web Application Firewalls (WAFs): Los WAFs pueden inspeccionar el tráfico HTTP y bloquear peticiones maliciosas basadas en reglas predefinidas o comportamientos anómalos.
- Monitorización y Alertas: Implementar sistemas de monitorización para detectar patrones de tráfico inusuales que puedan indicar un ataque.
- Infraestructura escalable: Utilizar una infraestructura que pueda escalar dinámicamente para manejar picos de tráfico, aunque esto no siempre es suficiente contra ataques DoS sofisticados.

Ejercicio

Realizar una app en django, donde se modifican las urls de la app y la view, donde se pueda realizar las siguientes solicitudes:

http://127.0.0.1:8000/ # Landing Page
http://127.0.0.1:8000/request/ # Responder por HttpRequest
http://127.0.0.1:8000/request/app-attributes/ # Responder con un atributo que fue seteado en la vista.
http://127.0.0.1:8000/request/middleware/ # Responder con un atributo que fue seteado en el Middleware
http://127.0.0.1:8000/request/querydict/ # Responder con un objetos QueryDict
http://127.0.0.1:8000/request/is-secure/ # Utilizar HttpRequest.is_secure()
http://127.0.0.1:8000/home/ #Esta página demuestra cómo manejar GET y POST en una sola vista.
http://127.0.0.1:8000/response/ #Esto es un HttpResponse básico.
http://127.0.0.1:8000/response/subclasses/ #Este es un HttpResponse con encabezados
http://127.0.0.1:8000/response/json/ # responder con json
http://127.0.0.1:8000/response/streaming/ #responder con una secuencia de streaming
http://127.0.0.1:8000/response/file/ # Responder con algún recurso al servidor (archivo)
http://127.0.0.1:8000/response/base/ # Responder utilizado HttpResponseBase