David Steven Saldarriaga Pardo

Documento de diseño

MINI DAST MELI

1. Introducción.

Para el desarrollo de esta aplicación solicitada en la prueba técnica, se tomó un enfoque de distribución de carga para la optimización y la automatización combinada con la facilitación de ejecución de un proceso prestablecido de análisis de vulnerabilidades de un set de URLs halladas por medio de herramientas de descubrimiento de URLs (Spider y AJAX Spider) con una única URL base.

Con este documento se busca explicar cada decisión de diseño y arquitectura tomadas durante la elaboración de este aplicativo, para entenderse mejor al momento de su revisión.

Para esto se hará un análisis general de la aplicación y luego una vista punto por punto de los requerimientos del documento de la prueba técnica asociándolo a su respectiva solución.

1.1. Descripción de la aplicación.

La aplicación fue desarrollada usando Node.js 20, con ayuda del framework NestJS integrado con TypeORM y otras librerías qué ayudaron la construcción del programa con una arquitectura altamente modular.

La estructura de directorios y la arquitectura del código se diseñó tomando como base teórica la Arquitectura Hexagonal.

Para el diseño de la arquitectura del sistema se utilizaron técnicas de comunicación, desacoplamiento y alta disponibilidad de microservicios, además del API REST para la comunicación con el cliente. Estas decisiones se explicarán más a detalle en el contenido posterior del documento.

Acá de dará una vista general de la arquitectura y se explicarán sus componentes.

1.2. Componentes:

- **API:** Aplicación expuesta en la red que se encarga de recibir las peticiones del cliente, crear la información inicial del escaneo en base de datos y enviar el evento de ejecución del escaneo al broker de mensajería (Kafka).
  Nota: La aplicación de la API es exactamente la misma que la del Worker, con la diferencia de que, según la variable de entorno IS_WORKER, va a iniciar con la configuración adecuada para cada función dentro de la arquitectura.

- **Worker:** Esta encargada de recibir la carga del broker de mensajería y procesar el paso del escaneo, al terminar el paso, enviara el siguiente evento al broker para continuar el análisis.

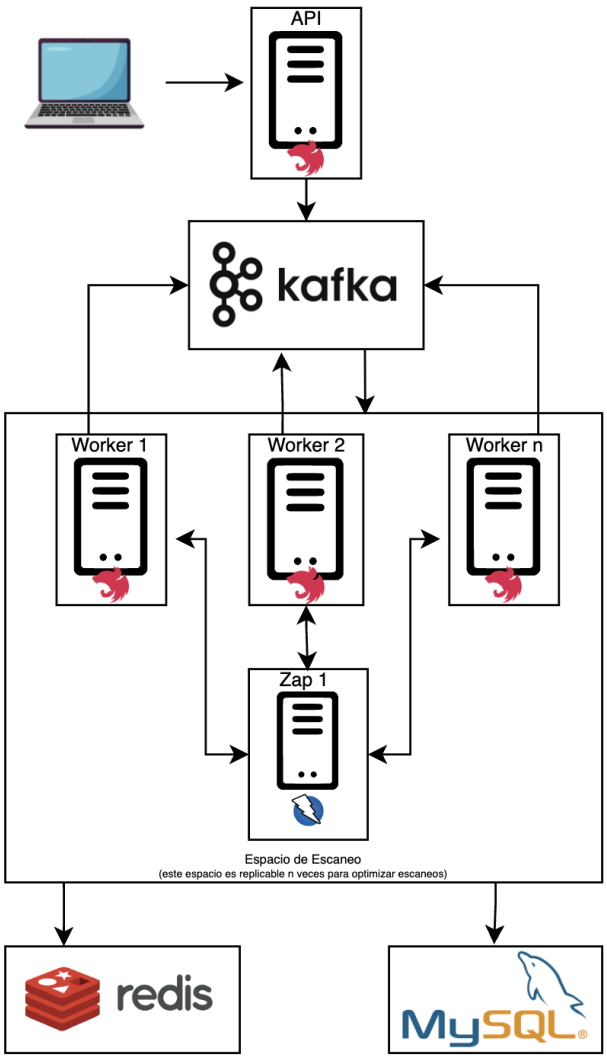
- **Kafka:** Broker encargado de la distribución de los eventos entre los consumidores disponibles para optimizar el procesamiento de estos y evitar las sobrecargas.

- **ZAP:** Contenedor o servidor que aloja la herramienta de análisis de vulnerabilidades. Este no debe necesariamente contener la herramienta ZAPROXY, sino que puede contener cualquier otra herramienta de análisis de vulnerabilidades.

- **Redis:** Instancia compartida de Redis para manejar Cache de manera global en el sistema planteado (usado para lista de espera explicado en el punto 3.3 y para control de progreso).

- **MySQL:** Base de datos relacional compartida en todo el sistema. Usado para persistir los resultados hallados en cada paso del análisis de vulnerabilidades.

1.3. Modelo de arquitectura:

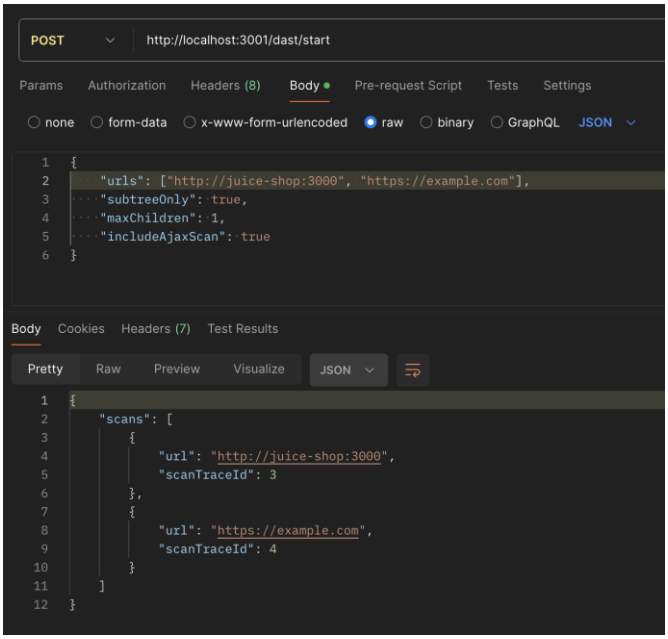


Modelo de arquitectura de la aplicación

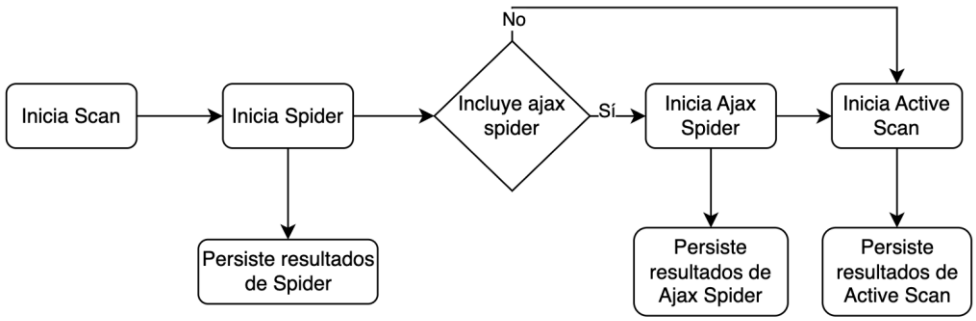
2. Análisis y solución de requerimientos:

2.1. *Realizar un escaneo de vulnerabilidades sobre un set de urls usando Active Scan de Zaproxy.*

Para esto se puso a disposición un endpoint que recibe una lista de urls y devuelve un id por cada url de la lista, esto permite dar seguimiento al estado de cada escaneo. Este endpoint cubre todos los pasos que puede tener el escaneo (incluyendo el escaneo ajax spider condicional).



Este endpoint ejecuta el siguiente flujo de manera asincrónica:

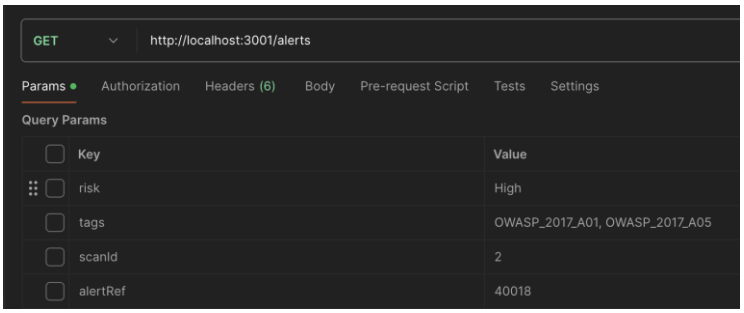


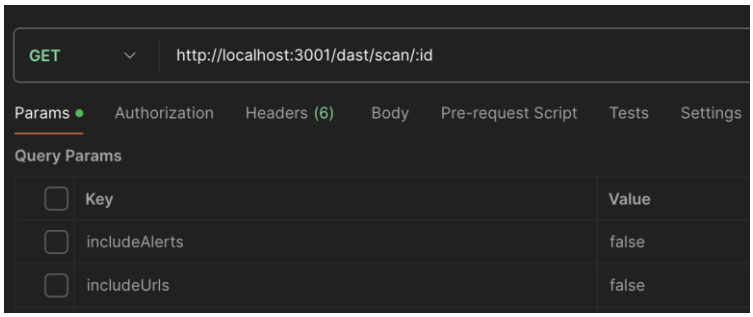
Además, cada paso comprueba haciendo la petición de comprobación de estado durante un tiempo hasta que termine. Para tener un mayor control sobre esto se configuraron variables de entorno que corresponde al tiempo de refresco de la petición y al tiempo máximo que puede esperar a la finalización.

```
# REFRESH TIMES
SPIDER_REFRESH_MS = 2000
MAX_SPIDER_TIME_MS = 3600000
AJAX_SPIDER_REFRESH_MS = 2000
MAX_AJAX_SPIDER_TIME_MS = 3600000
ACTIVE_SCAN_REFRESH_MS = 2000
MAX_ACTIVE_SCAN_TIME_MS = 3600000
```

2.2. *Poder consultar el estado de un escaneo y los resultados una vez terminado el mismo.*

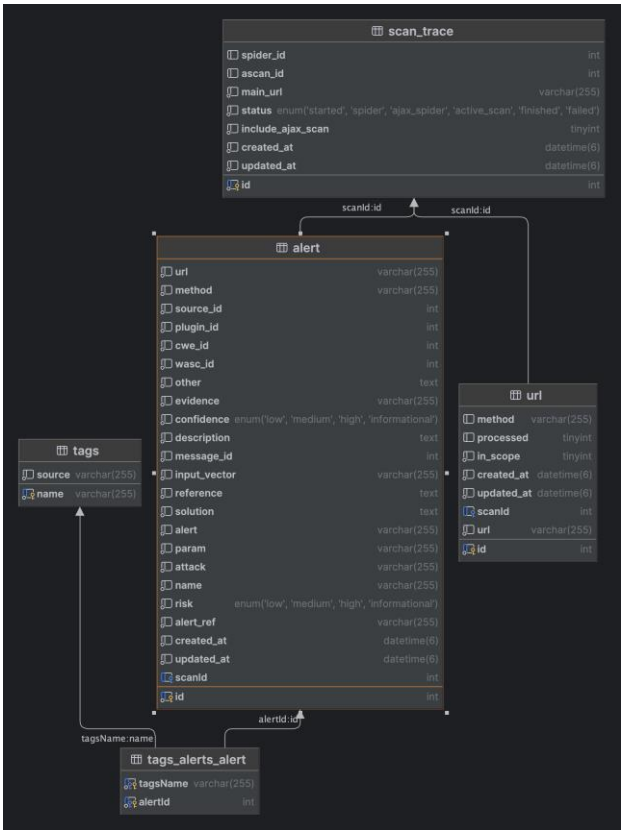
Para esto se habilitaron 2 endpoints que permiten consultar el estado de uno o varios escaneos y, en caso de haber terminado, las alertas y urls halladas en el análisis.





2.3. *Tienen que persistir tanto los escaneos realizados como sus resultados más allá de lo que puede guardar Zaproxy.*

Para esto se diseñó el siguiente modelo de datos, el cual guarda todos los datos que, durante el análisis del funcionamiento de la herramienta, parecía relevante e informativo:



2.4. *Tiene que ser independiente del motor de escaneo que utilice.*

Para cubrir este punto, se aprovechó el diseño modular del código y se realizó una estrategia de importación dinámica que utiliza el parámetro enviado en la importación para decidir el motor de escaneo a utilizar:

```
@Module({})
export class ScannerModule {
  static async register(options: {
    type: ScannersEnum;
  }): Promise<DynamicModule> {
    const { ScannerImplementation } = await import(
      `./services/${options.type}.service`
    );
    return {
      module: ScannerModule,
      providers: [{ provide: 'SCANNER', useClass: ScannerImplementation }],
      exports: ['SCANNER'],
    };
  }
}
```

```
@Module({
  imports: [
    KafkaModule,
    CacheModule,
    ScannerModule.register({
      type: ScannersEnum[process.env.SCANNER_TYPE] || ScannersEnum.ZAPROXY,
    }),
```

Aquí vemos como en el parámetro “type” recibe una variable de entorno que decide el escáner a utilizar, en caso de no ser especificada utiliza “ZAPROXY” por defecto.

Para poder implementar un nuevo escáner de manera satisfactoria, se debe hacer una especie de implementación del patrón adapter para que los recursos del nuevo escáner coincidan con los de la interfaz diseñada (con base en zaproxy) para estos escaneres. Esta la podemos ver a continuación:

```
export interface ScannerService {
  machineId: string;
  startSpiderScan(data: StartSpiderScanDto): Promise<number>;
  getSpiderScanStatus(scanId: string): Promise<number>;
  getSpiderFoundUrls(scanId: number): Promise<GetSpiderFoundUrlsResponse>;
  startAjaxSpiderScan(data: StartSpiderScanDto): Promise<void>;
  getAjaxSpiderScanStatus(): Promise<AjaxSpiderScanStatus>;
  getAjaxSpiderFoundUrls(): Promise<GetAjaxSpiderFoundUrlsResponse>;
  startActiveScan(url: string): Promise<number>;
  getActiveScanStatus(scanId: number): Promise<number>;
  getAlertsResult(url: string): Promise<AlertResponse[]>;
}
```

2.5. *Es parte del desafío el diseño de la API REST (los recursos, paths y métodos utilizados para cada acción).*

Dentro de la entrega se adjunta un archivo importable de Postman con todos los endpoints creados para esta solución y algunas descripciones de los parámetros de consumo para mayor facilidad de prueba.

2.6. *Utilizar docker-compose para levantar toda la app, incluyendo base de datos a utilizar y la herramienta Zaproxy.*

Adicional a los servicios iniciales dados al inicio de la prueba, se agregaron las herramientas necesarias para el correcto funcionamiento de la aplicación y dos servicios que levantan tanto el API como el worker para aumentar la facilidad de uso. El archivo docker-compose.yml se encuentra dentro del código de la solución. A continuación, la lista completa de los servicios:

- Zaproxy
- Juice-shop (aplicación vulnerable)
- MySQL
- Redis
- Kafka
- Zookeeper
- DAST-API (Aplicación desarrollada)
- DAST-WORKER

2.7. *Código funcionando en repositorio privado de GitHub.*

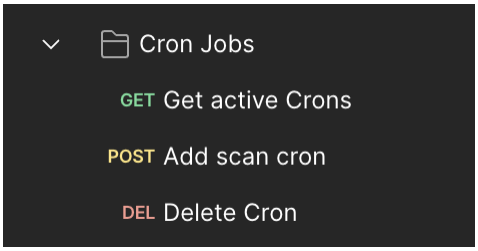
El código se encuentra en un repositorio privado por cómo se especificó en la prueba, por lo tanto, es necesario pedir acceso.

DAST-APP-MELI

3. Extras:

3.1. *Mecanismo para realizar escaneos programados.*

Se creo todo un apartado de la API para el manejo de los escaneos programados, tanto para agregarlos, eliminarlos o ver los eventos activos.



Como los Cronjobs son manejados internamente en la aplicación, suponiendo el caso de que se haga el levantamiento de varios API, cada uno manejara su propio sistema de Crons.

3.2. *Diseño adaptable a distintos scanners, ejemplo Burp o Acunetix.*

Este punto se trata en el punto 2.4.

3.3. *Propuesta de diseño para una aplicación escalable que pueda realizar N escaneos en paralelo, teniendo en cuenta cuestiones de performance y resiliencia.*

La herramienta zaproxy nos da la opción de hacer n escaneos paralelamente (a excepción del escaneo con Ajax spider), sin embargo, al ver que al hacer varios escaneos usando un único scanner, se decidió llevar más lejos y crear una arquitectura que permitiera el escalamiento horizontal de los servicios. Con el diagrama visto al inicio del documento podemos ver que,

usando esta arquitectura, podemos tener varias instancias de Zaproxy o cualquier otro escáner debidamente adaptado a la aplicación funcionando en perfecta sincronía con la aplicación siempre y cuando se respete el modelo de escalamiento y se haga una configuración adecuada del Broker para evitar entrecruzamiento de datos.

Adicionalmente, para controlar el uso del escáner con Ajax spider ya que no tiene la funcionalidad de escaneos en paralelo, se implementó un sistema de lista de espera (una especie de cola de eventos) el cual ayuda a controlar el orden de uso de esta funcionalidad sin causar cruce de información.