

Monitorización completa de un servidor de aplicaciones Java Wildfly



WildFly

Juan Manuel Durán Piñero

Motorización completa de un servidor de aplicaciones Java Wildfly

1. Objetivos que se quieren conseguir y se han conseguido.....	2
1.1. ¿Que quiero conseguir?.....	2
1.2. Objetivos.....	2
1.3. Objetivos cumplidos.....	3
2. Escenario necesario para la realización del proyecto.....	3
3. Descripción de los fundamentos teóricos y conceptos.....	5
3.1. WildFly.....	5
3.2. Grafana.....	6
3.3. Prometheus.....	7
3.4. Node Exporter.....	8
3.5. Loki.....	9
3.6. Promtail.....	10
3.7. Zipkin.....	11
3.8. Selenium.....	12
3.9. Jmeter.....	13
4. Ejemplos de lo que se ha realizado (instalaciones, configuraciones, pruebas, demos.....)	14
4.1. WildFly.....	14
4.2. Grafana.....	23
4.3. Prometheus.....	26
4.4. Node Exporter.....	28
4.5. Loki.....	30
4.6. Promtail.....	33
4.7. Zipkin.....	34
4.8. Selenium.....	38
4.9. Jmeter.....	41
5. Conclusiones y propuestas para continuar el desarrollo del proyecto.....	43
6. Dificultades encontradas.....	45
7. Bibliografía.....	47
7.1. WildFly.....	47
7.2. Grafana.....	47
7.3. Prometheus.....	47
7.4. Node Exporter.....	47
7.5. Loki.....	47

7.6. Promtail.....	47
7.7. Zipkin.....	47
7.8. Selenium.....	47
7.9. JMeter.....	47

1. Objetivos que se quieren conseguir y se han conseguido.

1.1. ¿Que quiero conseguir?.

El objetivo que me he propuesto para este proyecto es implementar un sistema de monitorización completo para un servidor WildFly utilizando herramientas como Grafana, Prometheus, Loki y Zipkin. Esto incluye la configuración de métricas, registros y seguimiento distribuido para obtener una visión integral del rendimiento y la salud del servidor.

1.2. Objetivos.

1. Configurar un entorno de monitorización para el servidor de aplicaciones WildFly que permita una supervisión sencilla y eficaz.
2. Visualizar métricas relevantes del servidor en tiempo real a través de paneles de control intuitivos, configurables y sencillos de entender en Grafana.
3. Recopilar y analizar registros de los servidores para identificar y resolver problemas de manera rápida y eficiente.
4. Implementar un seguimiento distribuido de solicitudes utilizando Zipkin para identificar cuellos de botella y mejorar el rendimiento.
5. Realizar pruebas de carga/estrés con Selenium y con jmeter o artillery.

1.3. Objetivos cumplidos.

Actualmente se han realizado completamente los tres primeros puntos y las pruebas de carga/estres con Selenium, se esta terminando de implementar el seguimiento distribuido de solicitudes utilizando Zipkin y realizando las primeras pruebas de carga con Jmeter y Artillery a fin de determinar el software que se utilizara en la demostración en vivo.

2. Escenario necesario para la realización del proyecto

Se utilizará mi propia maquina para implementar el entorno de monitorización, sin recurrir a un proveedor de nube externo, este entorno se encuentra implementado en contenedores docker usando docker compose, el servidor wildfly esta instalado en la maquina no sobre contenedores. Las tecnologías y programas que se utilizarán son:

- Servidor Java Wildfly: Para proveer la web que monitorizaremos, en este caso, usando el radio de un círculo insertado por pantalla calcula la circunferencia del mismo, su área, el volumen de la esfera y el área superficial de la esfera.
- Grafana: Para la visualización de métricas y registros en paneles de control personalizables.
- Prometheus: Para la recopilación y almacenamiento de métricas del servidor WildFly.
- Node Exporter:
- Loki: Para la recopilación y almacenamiento de registros de aplicación y sistema en un formato eficiente.
- Promtail: Para recopilar y enviar registros (logs) al servidor Loki para su almacenamiento y análisis.
- Zipkin: Para el seguimiento distribuido de solicitudes y la identificación de cuellos de botella en servicios distribuidos.
- Selenium: Herramienta para automatizar pruebas en mi aplicación web, con acceso a realizarlas desde una extensión del navegador o desde la terminal.
- Jmeter para la realización de pruebas de estrés y carga del servidor wildfly.

Los servicios necesarios incluyen:

- Servidor WildFly como la aplicación principal a monitorizar.

[Web de Wildfly](#)

- Servidor Grafana para visualizar datos de monitorización.

[Web de Grafana](#)

- Servidor Prometheus para recopilar y almacenar métricas.

[Web de Prometheus](#)

- Node Exporter

[Web de Node Exporter](#)

- Servidor Loki para recopilar y almacenar registros.

[Web de Loki](#)

- Servidor Zipkin para el seguimiento distribuido de solicitudes.

[Web de Zipkin](#)

- Servidor Jmeter para la realización del testeo del servidor wildfly.

[Web de Jmeter](#)

- Promtail para el envío de registros del sistema a Loki.

[Web de Promtail](#)

- Selenium para realizar pruebas a la web.

[Web de Selenium](#)

3. Descripción de los fundamentos teóricos y conceptos.

A continuación voy a explicar en detalle los fundamentos teóricos y los conceptos sobre las distintas tecnologías que voy a usar:

3.1. WildFly

WildFly es un servidor de aplicaciones Java EE que proporciona un entorno robusto y flexible para el desarrollo y despliegue de aplicaciones. Es el sucesor del JBoss Application Server, por lo cual WildFly se ha convertido en una opción popular entre los desarrolladores debido a su arquitectura modular, que permite arrancar solo los módulos necesarios, mejorando así el rendimiento y reduciendo el tiempo de inicio respecto a otros servidores de aplicaciones Java.

WildFly ofrece herramientas avanzadas de gestión y administración, incluyendo una interfaz de gestión web y una interfaz de línea de comandos (CLI), que facilitan la configuración, el despliegue y el monitoreo de aplicaciones. Su capacidad de clustering permite la alta disponibilidad y escalabilidad, asegurando que las aplicaciones pueden manejar grandes volúmenes de tráfico y seguir funcionando incluso en caso de fallos.

Conceptos:

- **Modularidad:** Permite un uso eficiente de recursos al cargar solo los componentes necesarios.
- **Rendimiento:** Su arquitectura modular y capacidades de clustering aseguran un alto rendimiento y escalabilidad.
- **Compatibilidad:** Totalmente compatible con las especificaciones Java EE, lo que facilita el desarrollo de aplicaciones empresariales robustas.
- **Herramientas de gestión:** Las interfaces de gestión web y CLI facilitan la administración del servidor.

3.2. Grafana

Grafana es una plataforma de análisis y visualización de datos de código abierto que se utiliza para monitorear y visualizar métricas en tiempo real. Permite crear dashboards personalizables que pueden combinar datos de múltiples fuentes en un solo lugar, proporcionando una vista unificada del rendimiento del sistema.

Grafana es compatible con una amplia gama de fuentes de datos, incluyendo Prometheus, Graphite, InfluxDB, Elasticsearch, y muchas más. Su sistema de alertas permite configurar notificaciones basadas en métricas específicas, lo que ayuda a identificar y responder rápidamente a problemas de rendimiento.

Conceptos:

Dashboards personalizables: Ofrece una gran flexibilidad para crear visualizaciones personalizadas según las necesidades del usuario.

- **Compatibilidad con múltiples fuentes de datos:** Permite integrar datos de diversas fuentes, proporcionando una visión completa del sistema.
- **Alertas:** Configuración de alertas proactiva que ayuda a mitigar problemas antes de que afecten a los usuarios.
- **Extensibilidad:** Amplio ecosistema de plugins que permite ampliar la funcionalidad de Grafana.

En los siguientes casos, creo que es muy necesario su uso:

- Cuando necesitemos una herramienta poderosa para visualizar y monitorear métricas en tiempo real.
- En entornos que requieran la integración de datos de múltiples fuentes en un solo dashboard.
- Para configurar alertas proactivas basadas en métricas específicas.

3.3. Prometheus

Prometheus es un sistema de monitoreo y alerta de código abierto diseñado para recolectar y almacenar métricas en forma de series temporales. Utiliza un modelo de datos basado en series temporales etiquetadas con pares clave-valor y emplea una arquitectura de scraping, donde Prometheus recoge métricas a intervalos regulares a través de HTTP.

Prometheus incluye su propio lenguaje de consultas, PromQL, que es muy potente y flexible para consultar y agregar métricas. Además, el componente Alertmanager de Prometheus maneja las alertas basadas en reglas definidas, enviándolas a diversos receptores.

Conceptos:

- **Modelo de datos robusto:** Su modelo de series temporales es ideal para el monitoreo continuo de métricas.
- **Lenguaje de consultas potente:** PromQL permite realizar consultas y agregaciones complejas para un análisis detallado.
- **Alertas integradas:** El Alertmanager facilita la gestión y el envío de alertas.
- **Escalabilidad:** Diseñado para escalar con grandes volúmenes de datos y múltiples instancias.

En los siguientes casos, creo que es muy necesario su uso:

- Cuando necesitemos un sistema de monitoreo escalable y robusto para recolectar y almacenar métricas.
- En entornos que requieran un lenguaje de consultas potente para analizar métricas.
- Para configurar alertas basadas en reglas específicas de métricas.

3.4. Node Exporter

Node Exporter es una herramienta de código abierto diseñada para recolectar métricas de hardware y del sistema operativo desde servidores Linux. Es parte del ecosistema de Prometheus y está optimizado para proporcionar información detallada sobre el rendimiento y el estado de los recursos del sistema, como la CPU, la memoria, el disco y la red.

Node Exporter expone estas métricas en un formato que puede ser fácilmente recolectado por Prometheus a través de HTTP, permitiendo la integración fluida y la visualización de estas métricas en dashboards de Grafana.

Conceptos:

- **Recolección de métricas del sistema:** Node Exporter recopila una amplia gama de métricas del sistema operativo y del hardware, lo que proporciona una visión detallada del estado del servidor.
- **Integración con Prometheus:** Diseñado para trabajar perfectamente con Prometheus, facilita la recolección y el almacenamiento de métricas.
- **Compatibilidad con Grafana:** Las métricas recolectadas pueden ser visualizadas en Grafana, permitiendo crear dashboards detallados y personalizados.
- **Monitoreo detallado de recursos:** Proporciona métricas detalladas de CPU, memoria, disco, red, entre otros, esenciales para el monitoreo del rendimiento del sistema.

En los siguientes casos, creo que es muy necesario su uso:

- Cuando necesitemos una herramienta para monitorear el estado y el rendimiento de los recursos de servidores Linux.
- En entornos donde ya se esté utilizando Prometheus para la recolección de métricas, y se necesite extender la visibilidad a nivel de hardware y sistema operativo.
- Para integrar las métricas del sistema operativo en dashboards de Grafana y obtener una vista unificada del rendimiento del sistema.

Node Exporter es fundamental para cualquier infraestructura de monitoreo que busque obtener datos detallados del estado del hardware y del sistema operativo, permitiendo una gestión más eficaz y proactiva del rendimiento y la disponibilidad del sistema.

3.5. Loki

Loki es un sistema de almacenamiento y búsqueda de logs desarrollado por Grafana Labs. Está diseñado para ser eficiente y escalable, siguiendo una filosofía similar a la de Prometheus pero aplicada a los logs. Loki no indexa el contenido completo de los logs, sino que utiliza etiquetas para organizar y buscar los logs, lo que reduce significativamente los requisitos de almacenamiento y mejora la eficiencia.

Loki se integra perfectamente con Grafana, lo que nos permite visualizar logs junto con métricas en los dashboards, proporcionando una vista unificada del estado y el rendimiento del sistema.

Conceptos:

- **Eficiencia en almacenamiento:** No indexar todo el contenido de los logs reduce los costos y la carga de almacenamiento.
- **Integración con Grafana:** Facilita la correlación entre métricas y logs, mejorando el análisis y la resolución de problemas.
- **Escalabilidad:** Diseñado para manejar grandes volúmenes de logs sin comprometer el rendimiento.
- **Simplicidad:** Menor complejidad en la gestión y búsqueda de logs gracias al uso de etiquetas.

En los siguientes casos, creo que es muy necesario su uso:

Cuando necesitemos una solución eficiente y escalable para el almacenamiento y búsqueda de logs.

En entornos que ya utilicen Grafana para la visualización de métricas.

Para correlacionar métricas y logs en una misma interfaz.

3.6. Promtail

Promtail es un agente de Loki utilizado para la recopilación y envío eficiente de logs. Desarrollado como parte del ecosistema de Loki por Grafana Labs, Promtail ofrece características clave que mejoran la gestión de logs en entornos de producción.

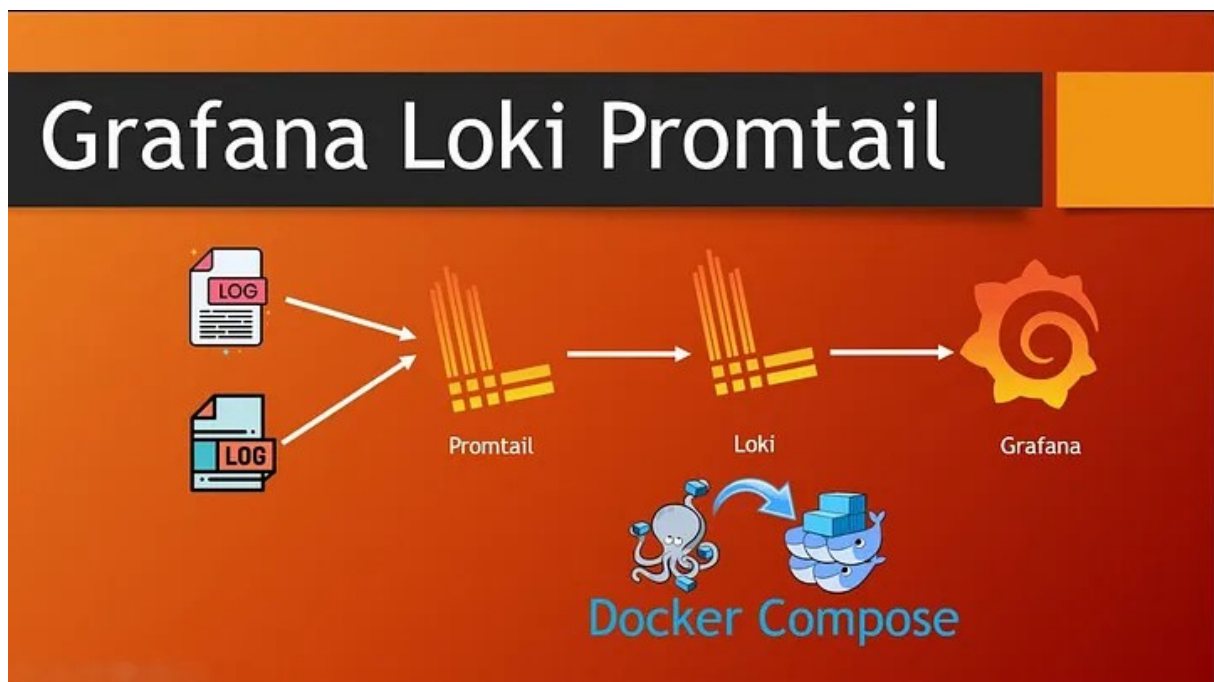
Promtail está diseñado para recopilar logs de manera eficiente, minimizando el impacto en los recursos del sistema. Esto asegura una recopilación rápida y fiable de los registros, incluso en entornos con grandes volúmenes de datos.

Nos permite etiquetar y filtrar logs antes de enviarlos a Loki para su almacenamiento. Esto ayuda a organizar y categorizar los registros, facilitando su posterior búsqueda y análisis.

Promtail se integra perfectamente con Grafana, lo que permite visualizar logs junto con métricas en los dashboards. Esta integración facilita la correlación entre logs y métricas, mejorando la capacidad de análisis y resolución de problemas.

En los siguientes casos, creo que es muy necesario su uso:

- En entornos de producción donde se necesita una recopilación eficiente de logs para el monitoreo y análisis del sistema.
- En entornos con grandes volúmenes de logs, ya que nos ayuda a gestionar la recopilación y almacenamiento de datos de manera eficiente, reduciendo la carga en los sistemas.



3.7. Zipkin

Zipkin es un sistema de trazado distribuido que ayuda a recopilar y visualizar datos sobre los tiempos de ejecución en una arquitectura de microservicios. Permite rastrear la ruta de las solicitudes a través de varios servicios, proporcionando una vista completa del tiempo de ejecución y ayudando a identificar cuellos de botella y problemas de latencia.

Para utilizar Zipkin, los servicios deben estar instrumentados para enviar datos de trazado. Zipkin almacena estos datos en una base de datos y ofrece herramientas para consultar y visualizar las trazas.

Conceptos:

- **Trazado distribuido:** Proporciona visibilidad completa sobre las interacciones entre microservicios, facilitando la identificación de problemas.
- **Instrumentación:** Compatible con varias bibliotecas y marcos, facilitando su adopción en diferentes entornos.
- **Análisis detallado:** Permite un análisis profundo del rendimiento y la latencia a nivel de solicitud.
- **Visualización:** Ofrece herramientas efectivas para visualizar y comprender las trazas.

En los siguientes casos, creo que es muy necesario su uso:

- Cuando trabajemos con arquitecturas de microservicios y necesitemos trazar el flujo de las solicitudes.
- Para identificar y solucionar problemas de latencia y rendimiento en sistemas distribuidos.
- En entornos que requieran una herramienta robusta para el trazado distribuido.

3.8. Selenium

Selenium es una suite de herramientas para la automatización de pruebas en aplicaciones web. Aunque no está directamente relacionado con Loki, Selenium desempeña un papel importante en la garantía de calidad y el monitoreo de aplicaciones web.

Selenium permite automatizar pruebas de funcionalidad, rendimiento y carga en aplicaciones web, lo que ayuda a identificar y corregir problemas de manera proactiva.

Selenium puede utilizarse para monitorear el rendimiento de aplicaciones web, identificando cuellos de botella, tiempos de carga lentos y otros problemas relacionados con el rendimiento.

Aunque no está directamente integrado con Loki, Selenium puede utilizarse en conjunto con herramientas de monitoreo como Prometheus y Grafana para recopilar y analizar logs relacionados con el rendimiento de aplicaciones web.

En los siguientes casos, creo que es muy necesario su uso:

- **Pruebas Automatizadas en Aplicaciones Web:** ya que es ideal para la automatización de pruebas en aplicaciones web, permitiendo realizar pruebas de regresión, funcionales y de rendimiento de manera eficiente y repetible.
- **Monitoreo de Rendimiento en Tiempo Real:** puesto que puede utilizarse para monitorear el rendimiento de aplicaciones web en tiempo real, identificando problemas de rendimiento y optimizando el rendimiento del sistema.
- **Análisis de Logs de Rendimiento:** aunque no está directamente relacionado con Loki, Selenium puede utilizarse en conjunto con herramientas de monitoreo de logs para analizar logs de rendimiento y mejorar el rendimiento de aplicaciones web.

3.9. Jmeter

Apache JMeter es una herramienta de prueba de carga de código abierto diseñada para probar aplicaciones web y medir su rendimiento. Proporciona una interfaz gráfica de usuario (GUI) que facilita el diseño y ejecución de pruebas de carga, soportando varios protocolos y tecnologías como HTTP, HTTPS, FTP, SOAP, REST, JDBC, entre otros.

JMeter ofrece capacidades de análisis y generación de informes, permitiendo visualizar y exportar resultados de las pruebas. Además, permite realizar pruebas de carga distribuidas, ejecutando múltiples instancias de JMeter en diferentes máquinas para simular una carga mayor.

Conceptos:

- **Interfaz gráfica intuitiva:** Facilita el diseño y ejecución de pruebas sin necesidad de conocimientos avanzados en programación.
- **Soporte para múltiples protocolos:** Versatilidad para probar diferentes tipos de aplicaciones y servicios.
- **Análisis y reporting:** Capacidades avanzadas de análisis y generación de informes para evaluar el rendimiento.
- **Escalabilidad:** Permite realizar pruebas de carga a gran escala mediante pruebas distribuidas.

En los siguientes casos, creo que es muy necesario su uso:

- Cuando necesitemos una herramienta flexible y poderosa para pruebas de carga de aplicaciones web.
- Para probar aplicaciones que utilicen múltiples protocolos y tecnologías.
- En entornos de desarrollo y producción para asegurar que las aplicaciones pueden manejar el tráfico esperado.

4. Ejemplos de lo que se ha realizado (instalaciones, configuraciones, pruebas, demos...)

4.1. WildFly

Para realizar la instalación del servidor WildFly primero debemos instalar los paquetes de los que depende el servidor, en primer lugar Java, en este caso optamos por OpenJDK.

```
sudo apt update
sudo apt -y install default-jdk
```

Una vez ejecutado comprobamos la versión de java que ha instalado.

```
java --version
Openjdk 17.0.11 2024-04-16
OpenJDK Runtime Environment (build 17.0.11+9-Debian-1deb12u1)
OpenJDK 64-Bit Server VM (build 17.0.11+9-Debian-1deb12u1, mixed mode, sharing)
```

Descargamos el ejecutable de WildFly para después descomprimirlo.

```
WILDFLY_RELEASE=$(curl -s
https://api.github.com/repos/wildfly/wildfly/releases/latest
|grep tag_name|cut -d '"' -f 4)
wget https://github.com/wildfly/wildfly/releases/download/${WILDFLY_RELEASE}/wildfly-${WILDFLY_RELEASE}.tar.gz
tar xvf wildfly-${WILDFLY_RELEASE}.tar.gz
```

La carpeta que acabamos de generar al descomprimir debemos moverla al directorio /opt

```
sudo mv wildfly-${WILDFLY_RELEASE} /opt/wildfly
```


Para hacer mas sencillo su mantenimiento he creado un archivo de unidad de sistema.

```
root@debian:/home/debian# systemctl status wildfly
● wildfly.service - The WildFly Application Server
   Loaded: loaded (/etc/systemd/system/wildfly.service; enabled; preset:
enabled)
   Active: active (running) since Mon 2024-05-27 12:47:23 CEST; 1min 59s ago
 Main PID: 531 (launch.sh)
    Tasks: 47 (limit: 8268)
  Memory: 364.4M
     CPU: 12.449s
   CGroup: /system.slice/wildfly.service
           └─531 /bin/bash /opt/wildfly/bin/launch.sh standalone
standalone.xml 0.0.0.0
           └─538 /bin/sh /opt/wildfly/bin/standalone.sh -c standalone.xml -b
0.0.0.0
               └─756 java "-D[Standalone]" "-
Djdk.serialFilter=maxbytes=10485760;maxdepth=128;maxarray=100000;maxrefs=30000
0" -Xms6>

may 27 12:47:23 debian systemd[1]: Started wildfly.service - The WildFly
Application Server.
```

Para empezar a utilizar nuestro servidor de aplicaciones Java debemos crear un usuario de administración para poder acceder a la consola.

```
debian@debian:~$ sudo /opt/wildfly/bin/add-user.sh

What type of user do you wish to add?
  a) Management User (mgmt-users.properties)
  b) Application User (application-users.properties)
(a): a

Enter the details of the new user to add.
Using realm 'ManagementRealm' as discovered from the existing property
files.
Username : jdurán
Password recommendations are listed below. To modify these restrictions
edit the add-user.properties configuration file.
  - The password should be different from the username
  - The password should not be one of the following restricted values
{root, admin, administrator}
  - The password should contain at least 8 characters, 1 alphabetic
character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
Password :
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma
separated list, or leave blank for none)[ ]:
About to add user 'jduran' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'jduran' to file '/opt/wildfly/standalone/configuration/mgmt-
users.properties'
Added user 'jduran' to file '/opt/wildfly/domain/configuration/mgmt-
users.properties'
Added user 'jduran' with groups  to file
'/opt/wildfly/standalone/configuration/mgmt-groups.properties'
Added user 'jduran' with groups  to file
'/opt/wildfly/domain/configuration/mgmt-groups.properties'
```

Para finalizar, para poder acceder a la consola de administración desde la terminal debemos añadir a nuestro bashrc las siguientes variables de entorno:

```
cat >> ~/.bashrc <<EOF
export WildFly_BIN="/opt/wildfly/bin/"
export PATH=$PATH:$WildFly_BIN

EOF
```

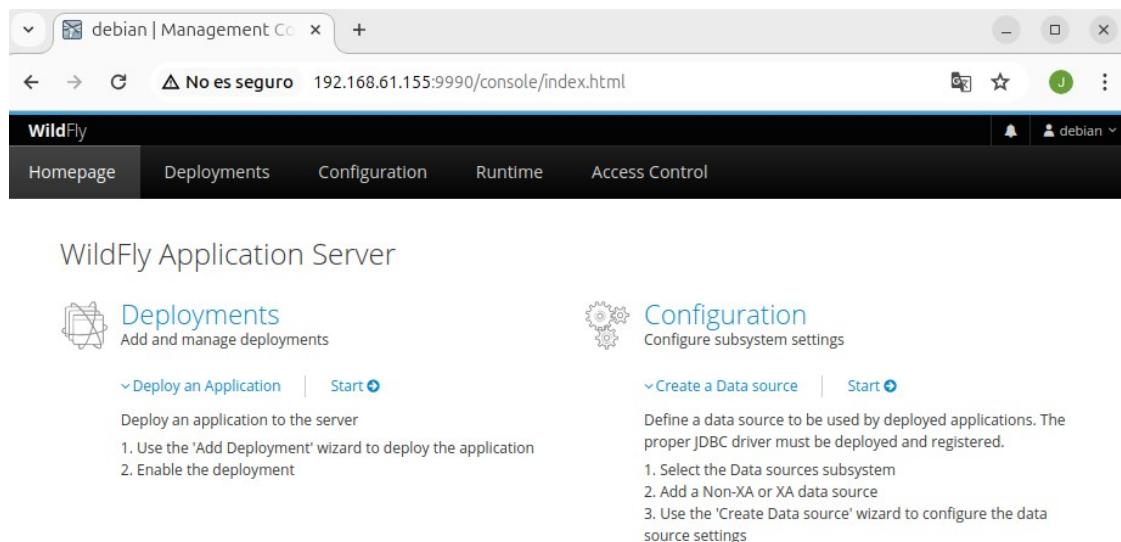
Y cargamos las modificaciones.

```
source ~/.bashrc
```

En este momento ya podemos acceder tanto a través de la terminal como desde el navegador:

```
jboss-cli.sh --connect
Authenticating against security realm: ManagementRealm
Username: jduran
Password:
[standalone@localhost:9990 /] version
JBoss Admin Command-line Interface
JBOSS_HOME: /opt/wildfly
Release: 20.0.2.Final
Product: WildFly Full 28.0.1.Final
JAVA_HOME: null
java.version: 17.0.11
java.vm.vendor: Debian
java.vm.version: 17.0.9+11-Debian-1deb12u1
os.name: Linux
os.version: 6.1.0-21-amd64
[standalone@localhost:9990 /] exit
```

Vía web podemos acceder por el puerto 9990.



Despliegue:

Para poder realizar el despliegue de nuestra aplicación tenemos que realizar varios pasos.

1. Debemos crear tres documentos principales, el primero un documento pom.xml, donde definimos las dependencias de nuestro proyecto:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ejemplo</groupId>
  <artifactId>tu-proyecto</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>17</source>
          <target>17</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

También necesitamos dentro de la ruta, src/main/java/com/ejemplo/PiInJava.java, el código de la app:

```
public class PiInJava {  
    public static double circumferenceOfCircle(int radius) {  
        return Math.PI * (2 * radius);  
    }  
  
    public static double areaOfCircle(int radius) {  
        return Math.PI * Math.pow(radius, 2);  
    }  
  
    public static double volumeOfSphere(int radius) {  
        return (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);  
    }  
  
    public static double surfaceAreaOfSphere(int radius) {  
        return 4 * Math.PI * Math.pow(radius, 2);  
    }  
  
    public static void main(String[] args) {  
        int radius = 5;  
        System.out.println("Circumference of the Circle = " +  
circumferenceOfCircle(radius));  
        System.out.println("Area of the Circle = " + areaOfCircle(radius));  
        System.out.println("Volume of the Sphere = " +  
volumeOfSphere(radius));  
        System.out.println("Surface Area of the Sphere = " +  
surfaceAreaOfSphere(radius));  
    }  
}
```

Y por ultimo un pequeño html donde indicaremos la configuración de la visión de la web:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Calculadora de Pi</title>
</head>
<body>
  <h1>Calculadora de Pi</h1>
  <p>Este es un servlet Java que calcula una aproximación de Pi
utilizando el método de Leibniz.</p>
  <p>Para calcular Pi, presiona el siguiente botón:</p>
  <form action="/pi-calculator-1.0/pi" method="get"> <
    <button type="submit">Calcular Pi</button>
  </form>
</body>
</html>
```

Una vez tenemos estos ficheros podemos realizar el despliegue para ello utilizamos maven, debemos instalarlo:

```
sudo apt install maven
```

Una vez preparados vamos a realizar la compilación del fichero java:

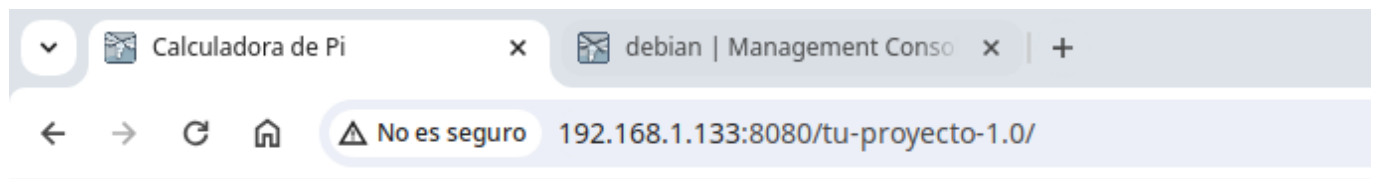
```
sudo apt install maven
sudo mvn -e clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ejemplo:tu-proyecto
>-----
[INFO] Building tu-proyecto 1.0
[INFO] -----
[ war ]-----
```

Podemos despegarla desde el entorno web, quedándonos el despliegue listo y la app funcionando:

The screenshot shows the WildFly Management Console interface. The browser address bar indicates the URL: 192.168.1.133:9990/console/index.html#deployments;path=deployment-dply-tu-proyecto-10war. The console has a navigation bar with tabs: Homepage, Deployments (selected), Configuration, Runtime, and Access Control. On the left, under 'Deployment (1)', there is a filter by name or deployment status and a list item 'tu-proyect...' with a 'View' button. The main content area displays the details for 'tu-proyecto-1.0.war'. A green status bar at the top indicates the deployment is enabled and active, with a 'Disable' link. Below this, the 'Main Attributes' table provides details about the deployment.

Main Attributes	
Name:	tu-proyecto-1.0.war
Runtime Name:	tu-proyecto-1.0.war
Context Root:	/tu-proyecto-1.0
Hash:	dd0958340b577ce94fac41092b270c7e8e968a46
Enabled, Managed, Exploded:	✓ ✓ ✗
Status:	OK
Last enabled at:	5/24/24, 9:06 AM
Last disabled at:	n/a

Vista de la app:



Calculadora de Pi

Radio del círculo: Calcular

Resultados:

Circunferencia del círculo: 6.283185307179586

Área del círculo: 3.141592653589793

Volumen de la esfera: 4.1887902047863905

Área superficial de la esfera: 12.566370614359172

4.2. Grafana

Para la implantación de estos componentes de la practica he elegido usar docker, en concreto he creado un docker compose con la información de cada uno:

```
grafana:
  environment:
    - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
    - GF_AUTH_ANONYMOUS_ENABLED=true
    - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
  entrypoint:
    - sh
    - -euc
    - |
      mkdir -p /etc/grafana/provisioning/datasources
      cat <<EOF >
/etc/grafana/provisioning/datasources/ds.yaml
  apiVersion: 1
  datasources:
    - name: Loki
      type: loki
      access: proxy
      orgId: 1
      url: http://192.168.1.138:3100
      basicAuth: false
      isDefault: true
      version: 1
      editable: false
  EOF
  /run.sh
  image: grafana/grafana:latest
  ports:
    - "3000:3000"
```

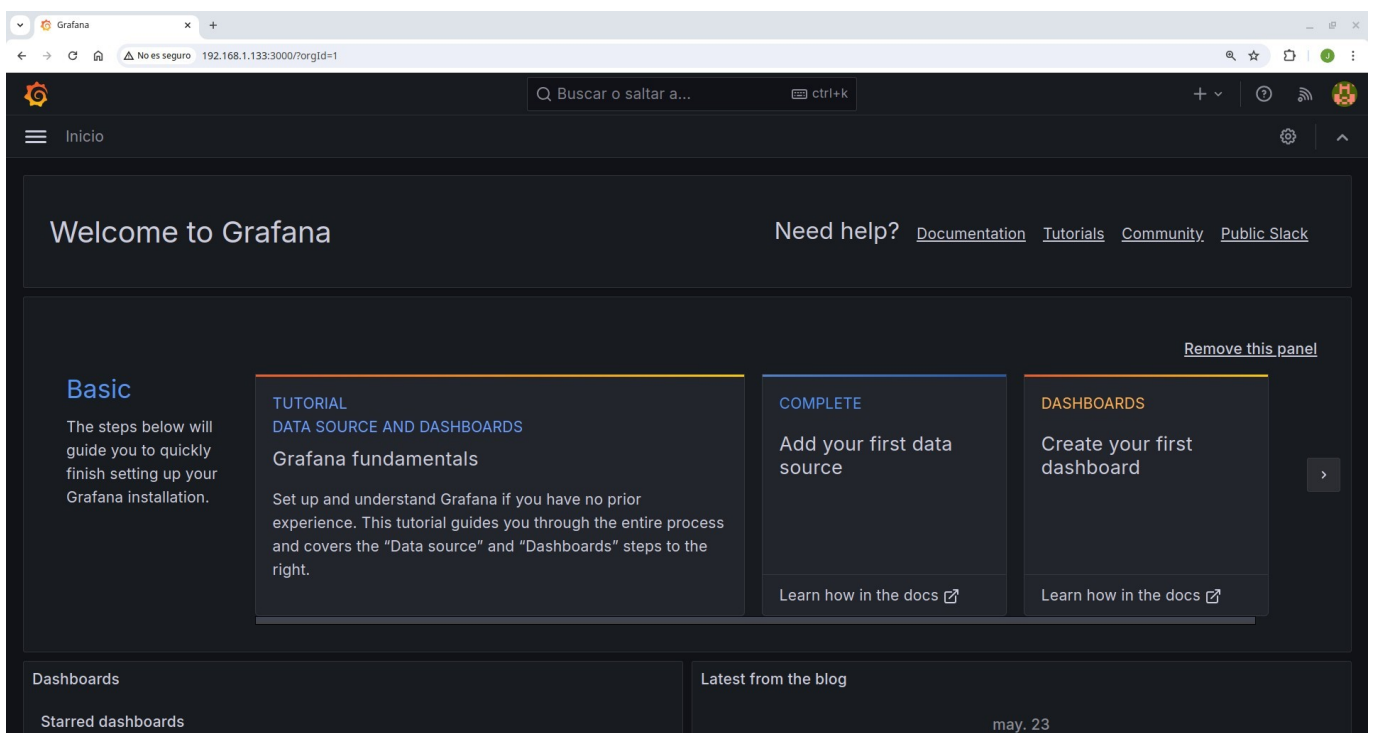
Podemos lanzar el docker compose con el siguiente comando:

```
docker compose up -d
WARN[0000] /home/debian/docker/docker-compose.yaml:
`version` is obsolete
[+] Running 5/5
 Container docker-promtail-1 Started 1.4s
 Container docker-grafana-1 Started 1.4s
 Container zipkin Running 0.0s
 Container prometheus Started 2.2s
 Container docker-loki-1 Started 1.8s
```

Comprobamos el estado del contenedor de grafana:

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
919a66998c5f      grafana/grafana:latest "sh -euc 'mkdir -p /..." 2 days ago         Up 2 minutes       0.0.0.0:3000->3000/tcp, :::3000->3000/tcp   docker-grafana-1
```

Ya podemos acceder al cliente web de grafana:



4.3. Prometheus

Comprobamos al igual que antes que el contenedor esta funcionando.

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
Bd15180bc80a	prom/prometheus:v2.11.1	"/bin/prometheus --c..."	2 days ago	Up 9 minutes

PORTS	NAMES
0.0.0.0:9090->9090/tcp, :::9090->9090/tcp	prometheus

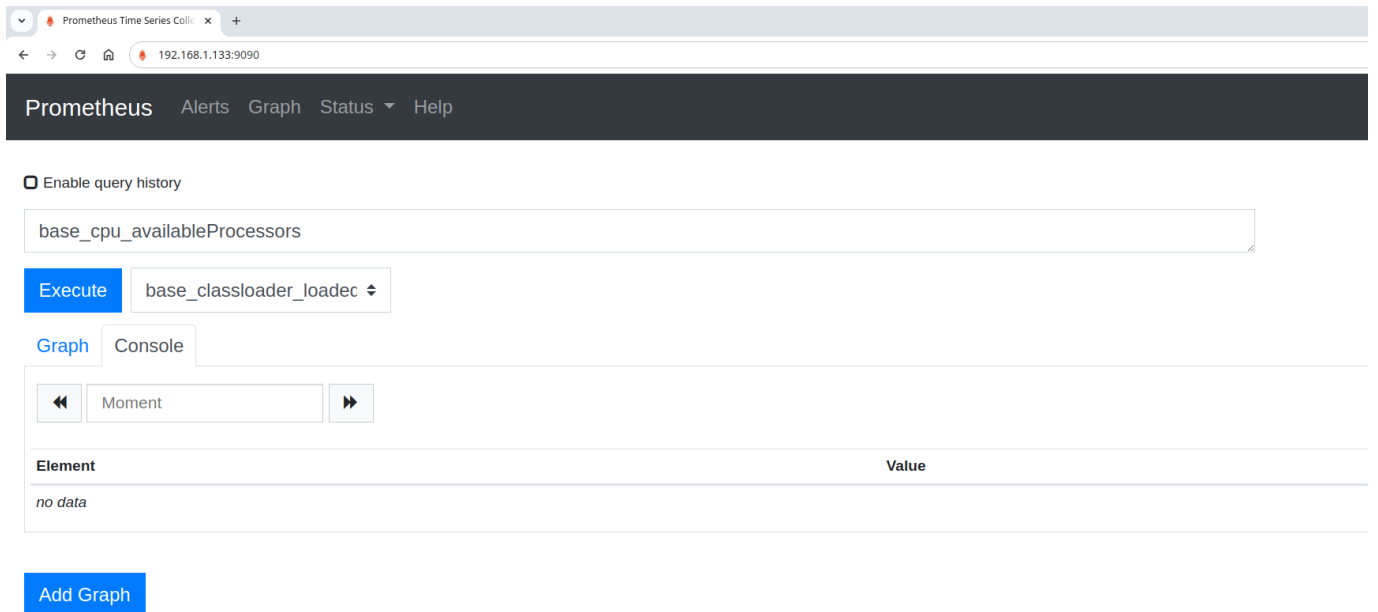
Tenemos configurado el contenedor con lo siguientes parámetros:

```
prometheus:
  image: prom/prometheus:v2.11.1
  container_name: prometheus
  volumes:
    - prometheus:/prometheus
    - ./config/prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - 9090:9090
  restart: on-failure
  networks:
    - ProyectoAsir
```

Y tenemos un fichero de configuración con la información de la conexión con el registro de las métricas

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
rule_files:
scrape_configs:
  - job_name: 'wildfly'
    metrics_path: /metrics
    static_configs:
      - targets: ['192.168.1.138:9990']
```

Aquí tenemos el acceso a la web donde nos muestra el acceso a Prometheus.



The screenshot shows the Prometheus web interface. At the top, there's a navigation bar with 'Prometheus', 'Alerts', 'Graph', 'Status', and 'Help'. Below this, there's a checkbox for 'Enable query history'. The main query input field contains 'base_cpu_availableProcessors'. To the right of the input is an 'Execute' button and a dropdown menu showing 'base_classloader_loaddec'. Below the input field, there are two tabs: 'Graph' (selected) and 'Console'. Under the 'Graph' tab, there's a time range selector set to 'Moment' with left and right arrow buttons. Below this is a table with two columns: 'Element' and 'Value'. The table currently shows 'no data'. At the bottom left, there is an 'Add Graph' button.

Element	Value
no data	

4.4. Node Exporter

Vamos a crear el contenedor de node exporter, para ello lo primero es generar su docker compose.

```
node-exporter:
  image: prom/node-exporter:latest
  container_name: node-exporter
  restart: unless-stopped
  volumes:
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /:/rootfs:ro
    - /etc/hostname:/etc/nodename:ro
    - /mnt/docker-cluster:/mnt/docker-cluster:ro
    - /etc/timezone:/etc/TZ:ro
    - /etc/localtime:/etc/localtime:ro
  command:
    - '--path.procfs=/host/proc'
    - '--path.rootfs=/rootfs'
    - '--path.sysfs=/host/sys'
    - '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($|/)'
  ports:
    - 9100:9100
  networks:
    - ProyectoAsir
```

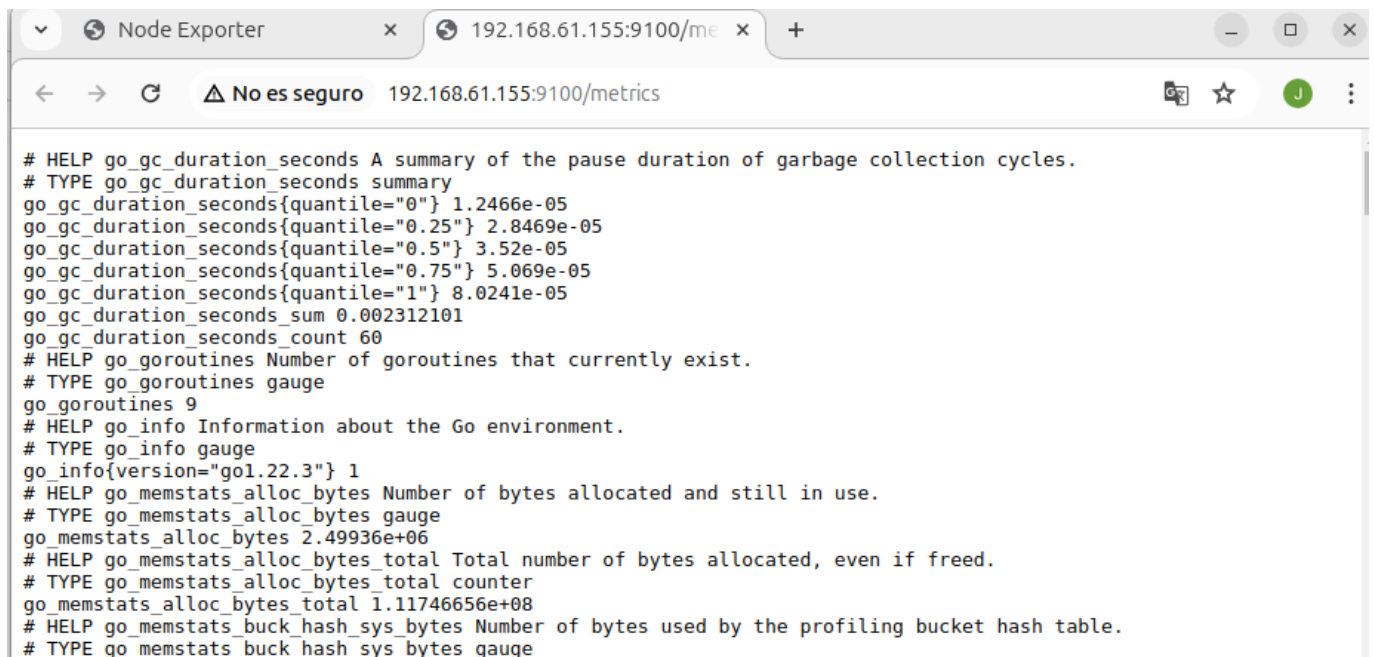
Comprobamos como el contenedor esta funcionando.

```
docker ps -a
CONTAINER ID   IMAGE                                COMMAND
4417546563b7   prom/node-exporter:latest           "/bin/node_exporter ..."
STATUS        PORTS
2 hours ago    Up 41 seconds                        0.0.0.0:9100->9100/tcp, :::9100->9100/tcp
NAMES
node-exporter
```

Para finalizar, debemos añadir en la configuración de prometheus, la información referente a la conexión con node exporter.

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
rule_files:
scrape_configs:
  - job_name: 'wildfly'
    metrics_path: /metrics
    static_configs:
      - targets: ['localhost:9990']
  - job_name: 'node-exporter'
    metrics_path: /metrics
    static_configs:
      - targets: ['localhost:9100']
```

Ya estaría configurado, podemos ver como podemos acceder al navegador al puerto 9100 para ver los datos o directamente desde prometheus o grafana en los cual hemos integrado al realizar la configuración nueva en el fichero de configuración de prometheus.



4.5. Loki

Esta es la configuración del docker compose del contenedor de Loki:

```
loki:
  image: grafana/loki:2.9.2
  ports:
    - "3100:3100"
  command: -config.file=/etc/loki/local-config.yaml
  networks:
    - ProyectoAsir
```

Al igual que antes aquí tenemos el contenedor arrancado y funcionando.

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                               NAMES
183db20c6848      grafana/loki:2.9.2  "/usr/bin/loki -conf...  3 days ago         Up 9 minutes        0.0.0.0:3100->3100/tcp, :::3100->3100/tcp  docker-loki-1
```

La integración de Loki con Grafana se realiza de forma automática ya que esta definida en su fichero de configuración:

```
auth_enabled: false

server:
  http_listen_port: 3100

ingester:
  lifecycler:
    ring:
      kvstore:
        store: inmemory
      replication_factor: 1
    final_sleep: 0s

schema_config:
  configs:
    - from: 2022-01-01
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      index:
        prefix: index_
        period: 24h

storage_config:
  boltdb_shipper:
    active_index_directory: /loki/index
    cache_location: /loki/cache
    shared_store: filesystem
  filesystem:
    directory: /loki/chunks

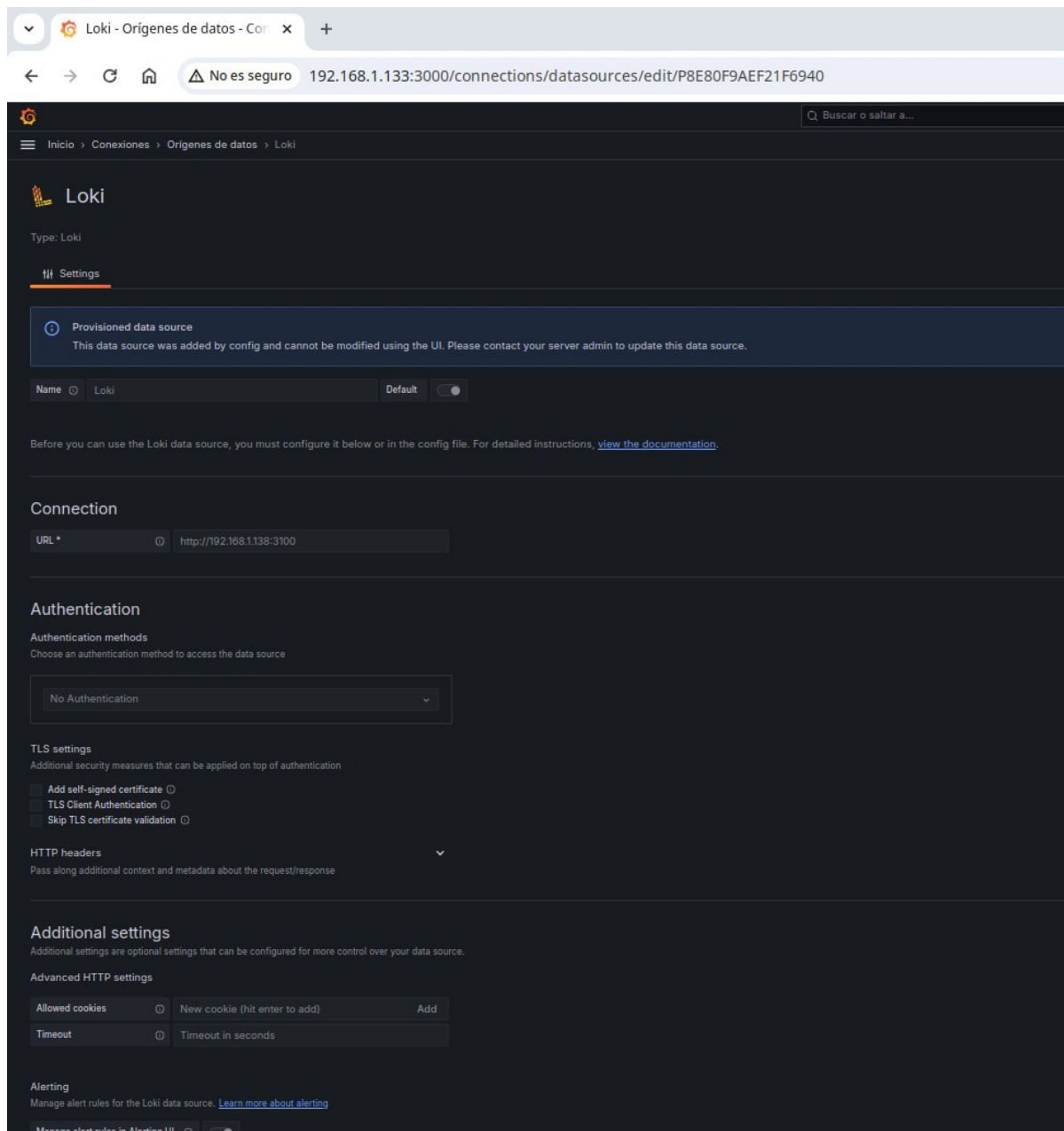
compactor:
  working_directory: /loki/compactor
  shared_store: filesystem

limits_config:
  enforce_metric_name: false
  reject_old_samples: true
  reject_old_samples_max_age: 168h

chunk_store_config:
  max_look_back_period: 0s

table_manager:
  retention_deletes_enabled: true
  retention_period: 168h
```


Aquí podemos ver como se ha configurado de forma automática Loki en Grafana según la configuración que habíamos decidido.



4.6. Promtail

Esta es la configuración del contenedor docker de promtail.

```
promtail:
  image: grafana/promtail:2.9.2
  volumes:
    - /var/log:/var/log
  command: -config.file=/etc/promtail/config.yml
  networks:
    - ProyectoAsir
```

También disponemos de la configuración interna de promtail para que realice la conexión con Loki y le envíe las métricas:

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://192.168.1.138:3100/loki/api/v1/push

scrape_configs:
  - job_name: system
    static_configs:
      - targets:
          - localhost
        labels:
          job: varlogs
          __path__: /var/log/*log
```

4.7. Zipkin

Esta es la configuración del docker compose referente a Zipkin.

```
storage:
  image: openzipkin/zipkin-mysql
  container_name: mysql
  environment:
    - MYSQL_ROOT_PASSWORD=zipkin
    - MYSQL_DATABASE=zipkin
    - MYSQL_USER=zipkin
    - MYSQL_PASSWORD=zipkin
  networks:
    - ProyectoAsir
  # ports:
  #   - 3306:3306

zipkin:
  image: openzipkin/zipkin
  container_name: zipkin
  environment:
    - STORAGE_TYPE=mysql
    - MYSQL_HOST=mysql
    - MYSQL_USER=zipkin
    - MYSQL_PASS=zipkin
  ports:
    - 9411:9411
  depends_on:
    - storage
  networks:
    - ProyectoAsir

dependencies:
  image: openzipkin/zipkin-dependencies
  container_name: dependencies
  entrypoint: crond -f
  environment:
    - STORAGE_TYPE=mysql
    - MYSQL_HOST=mysql
    - MYSQL_USER=zipkin
    - MYSQL_PASS=zipkin
  depends_on:
    - storage
  networks:
    - ProyectoAsir
```

Podemos ver como los contenedores se encuentran arrancados de forma óptima.

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
23fce3a41bb1	openzipkin/zipkin	"start-zipkin"	About a minute ago	Up 57 seconds (healthy)
4fae2198b3d1	openzipkin/zipkin-dependencies	"crond -f"	About a minute ago	Up 57 seconds
1230580b1dfb	openzipkin/zipkin-mysql	"start-mysql"	About a minute ago	Up 58 seconds (healthy)

Para configurar zipkin necesitamos aparte de crear el contenedor con la configuración anterior ,
descargarnos opentelemetry en nuestro servidor para poder exportar las métricas.

```
wget
https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/download/v1.33.3/opentelemetry-javaagent.jar
Grabando a: «opentelemetry-javaagent.jar»

opentelemetry-javaagen 100%[=====>] 19,84M 9,52MB/s
en 2,1s

2024-06-11 10:53:19 (9,52 MB/s) - «opentelemetry-javaagent.jar» guardado
[20804256/20804256]
```

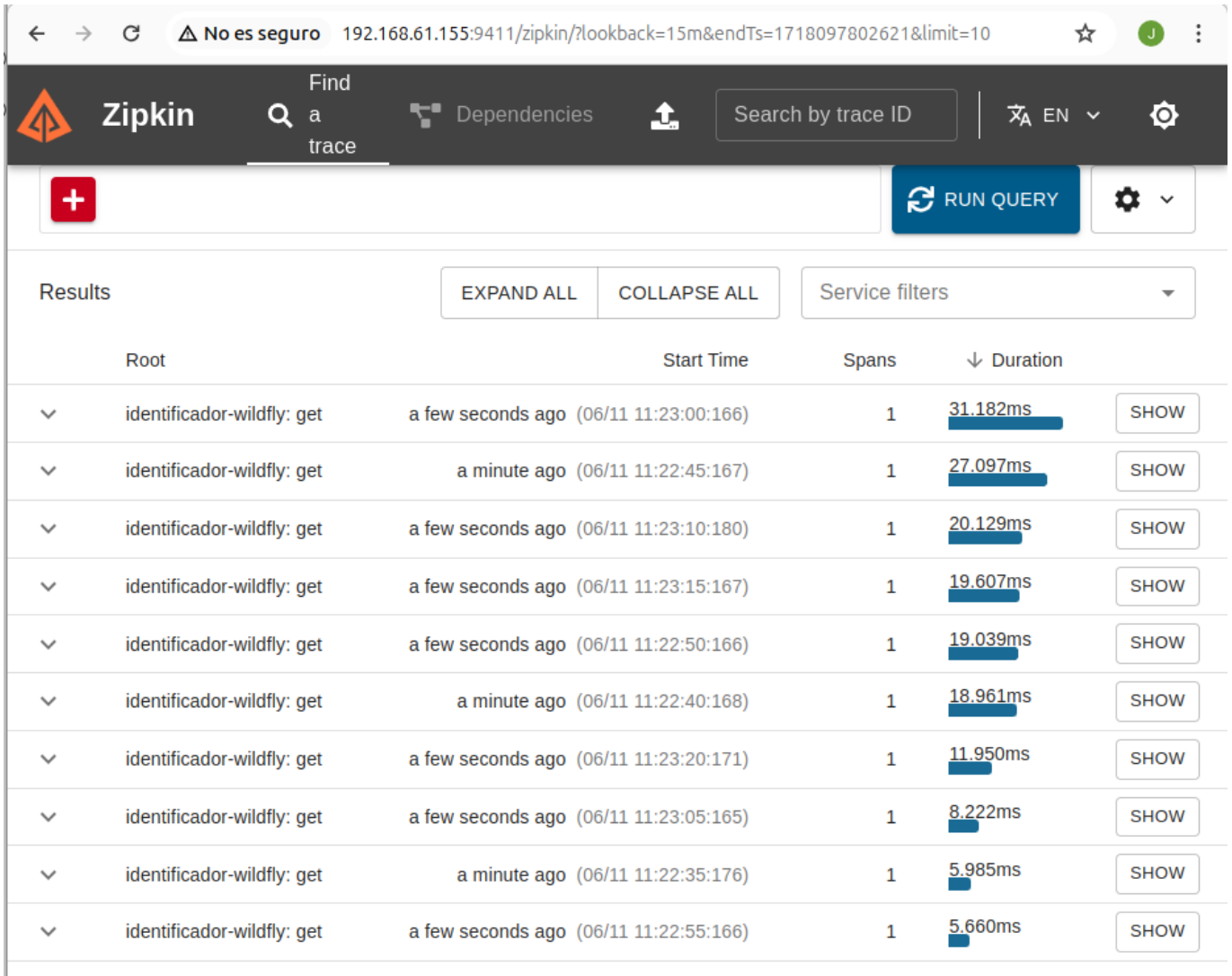
Copiamos el jar en la siguiente ruta:

```
sudo cp opentelemetry-javaagent.jar /opt/wildfly/bin/
chown wildfly: /opt/wildfly/bin/opentelemetry-javaagent.jar
```

Añadimos en la ruta /opt/wildfly/bin/standalone.conf la siguiente linea:

```
JAVA_OPTS="$JAVA_OPTS -javaagent:RUTA_DEL_WILDFLY/bin/opentelemetry-
javaagent.jar \
-Dotel.resource.attributes=service.name=identificador-wildfly \
-Dotel.traces.exporter=zipkin \
-Dotel.exporter.zipkin.endpoint=http://ip del servidor
zipkin:9411/api/v2/spans -Dotel.metrics.exporter=none -
Dotel.logs.exporter=none"
```

Aquí podemos ver ya funcionando, obteniendo las métricas de las peticiones a la web a Zipkin.



The screenshot displays the Zipkin web interface. The browser address bar shows the URL: `192.168.61.155:9411/zipkin/?lookback=15m&endTs=1718097802621&limit=10`. The interface includes a search bar with the text "Find a trace", a "Dependencies" link, and a "Search by trace ID" input field. A "RUN QUERY" button is visible on the right. Below the search bar, there are "EXPAND ALL" and "COLLAPSE ALL" buttons, and a "Service filters" dropdown menu. The main table lists traces with columns: Root, Start Time, Spans, and Duration. Each row includes a "SHOW" button.

	Root	Start Time	Spans	Duration	
▼	identificador-wildfly: get	a few seconds ago (06/11 11:23:00:166)	1	31.182ms	SHOW
▼	identificador-wildfly: get	a minute ago (06/11 11:22:45:167)	1	27.097ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:23:10:180)	1	20.129ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:23:15:167)	1	19.607ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:22:50:166)	1	19.039ms	SHOW
▼	identificador-wildfly: get	a minute ago (06/11 11:22:40:168)	1	18.961ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:23:20:171)	1	11.950ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:23:05:165)	1	8.222ms	SHOW
▼	identificador-wildfly: get	a minute ago (06/11 11:22:35:176)	1	5.985ms	SHOW
▼	identificador-wildfly: get	a few seconds ago (06/11 11:22:55:166)	1	5.660ms	SHOW

4.8. Selenium

Para utilizar Selenium tenemos varias posibilidades como son:

1. Usar el IDE web, para ello debemos instalar en nuestro equipo los siguientes paquetes:

```
sudo apt install nodejs
sudo apt install npm
sudo npm install -g selenium-side-runner
sudo npm install -g chromedriver
sudo apt-get install fonts-liberation libu2f-udev
https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
sudo dpkg -i google-chrome-stable_current_amd64.deb
sudo systemctl daemon-reload
sudo apt update
sudo chmod +x /usr/bin/google-chrome
sudo chmod +x /usr/local/bin/chromedriver
export PATH=$PATH:/usr/local/bin
```

Una vez instalados, podemos añadir desde la tienda de chrome la extensión de Selenium y empezar a realizar pruebas:

[Extensión Selenium para GoogleChrome](#)

Con la extensión de Selenium podemos empezar a realizar pruebas en nuestra web que podremos guardar como un fichero .side para llamarlo cuando necesitemos desde la terminal, con el siguiente comando:

```
selenium-side-runner --capabilities "goog:chromeOptions.args=[--headless,--
disable-gpu,--no-sandbox]" Descargas/prueba.side
```

2. La que voy a utilizar en la demostración, utilizar un script para realizar las pruebas de carga, para ello necesitamos instalar los siguientes paquetes:

```
apt install python3
apt install pip
pip install selenium webdriver-manager
```

Este es el código del script:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
import os
import time
import random
from concurrent.futures import ThreadPoolExecutor

def iniciar_driver():
    options = webdriver.ChromeOptions()
    options.add_argument("--use-fake-ui-for-media-stream")
    options.add_argument("--window-size=1920,1080")
    #Con esta opción sin comentar no se apertura abre ventana del navegador
    #options.add_argument("--headless")
    options.add_argument("--disable-gpu")
    options.add_argument("--disable-software-rasterizer")
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    driver =
webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()),
options=options)
    driver.maximize_window()
    return driver

def hacer_test(numero_instancias, interacciones):
    driver = iniciar_driver()
    #Se puede introducir la url de la aplicación java
    driver.get("http://192.168.61.115:8080/tu-proyecto-1.0/")
    wait = WebDriverWait(driver, 10)
    for i in range(interacciones):
        try:
            # Busca el campo de radio e ingresa un número aleatorio
            radius_field = wait.until(EC.element_to_be_clickable((By.XPATH,
'//*[@id="radius"]'))))
            radius = random.randint(1, 100) # Genera un número aleatorio
entre 1 y 100
            radius_field.clear() # Limpia el campo antes de ingresar un
nuevo valor
            radius_field.send_keys(str(radius))
            # Busca y hace clic en el botón de calcular
            calculate_button = driver.find_element(By.XPATH,
'//*[@id="piForm"]/button')
            calculate_button.click()
            # Breve retardo para evitar sobrecargar el servidor en exceso
            time.sleep(0.1)
            print(f"Instancia {numero_instancias}, Iteración {i+1}: Radio =
{radius}")
```

```
        except Exception as e:
            print(f"Instancia {numero_instancias}, Iteración {i+1}: Fallo - {e}")
        driver.quit()
        print(f"Prueba de cálculo completada para la instancia {numero_instancias}.")
        #Numero de navegadores en simultaneo
        def test_a_la_vez(num_usuarios, interacciones_usuario):
            with ThreadPoolExecutor(max_workers=num_usuarios) as executor:
                futures = [executor.submit(hacer_test, i, interacciones_usuario)
                           for i in range(1, num_usuarios + 1)]
                for future in futures:
                    future.result()
        if __name__ == "__main__":
            # Desde aqui puedo modificar el número de instancias y el número de iteraciones según sea necesario para realizar la prueba de carga
            num_usuarios = 10 # Número de usuarios concurrentes
            interacciones_usuario = 1000 # Número de iteraciones por usuario
            test_a_la_vez(num_usuarios, interacciones_usuario)
```


4.9. Jmeter

Para realizar la instalación de Jmeter podemos descargar su binario, actualmente necesita una versión de Java superior a la 8, que ya tenemos instalada gracias a la instalación previa de nuestro servidor de aplicaciones, en caso de usar jmeter desde otro equipo debemos instalar java.

[Enlace de descarga binario de Apache JMeter 5.6.3](#)

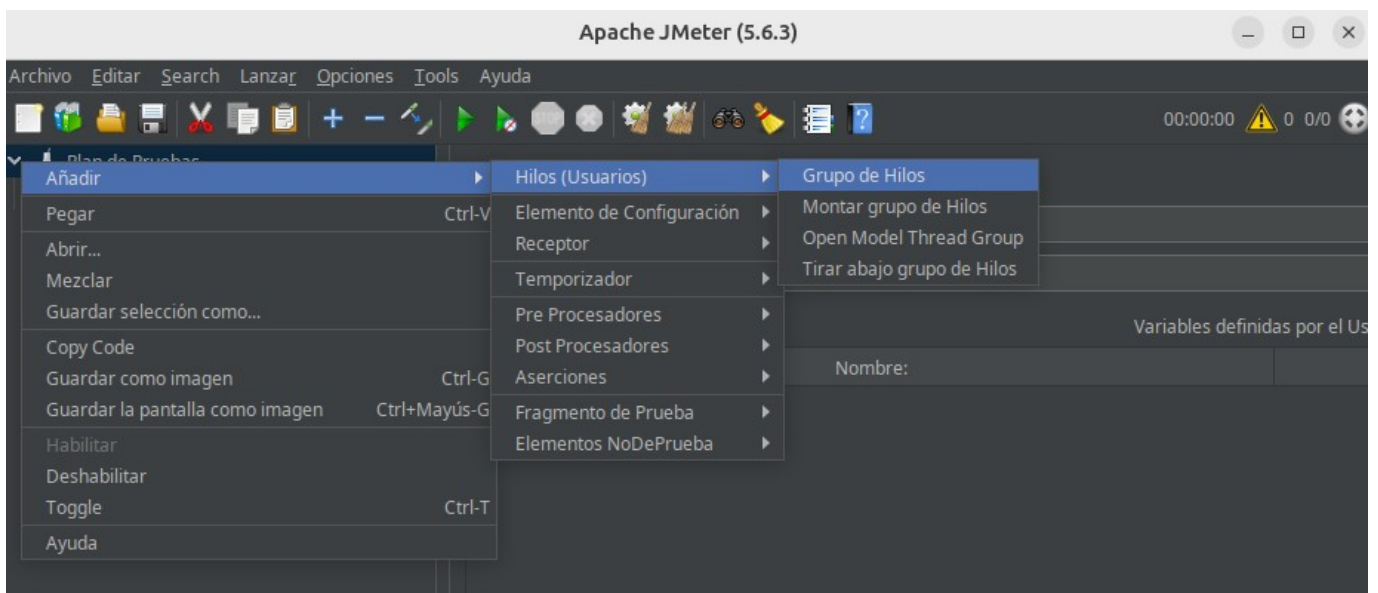
Una vez en la ubicación del directorio donde hemos descargado Jmeter procedemos a descomprimir el fichero tgz:

```
tar -xzf apache-jmeter-5.6.3.tgz
```

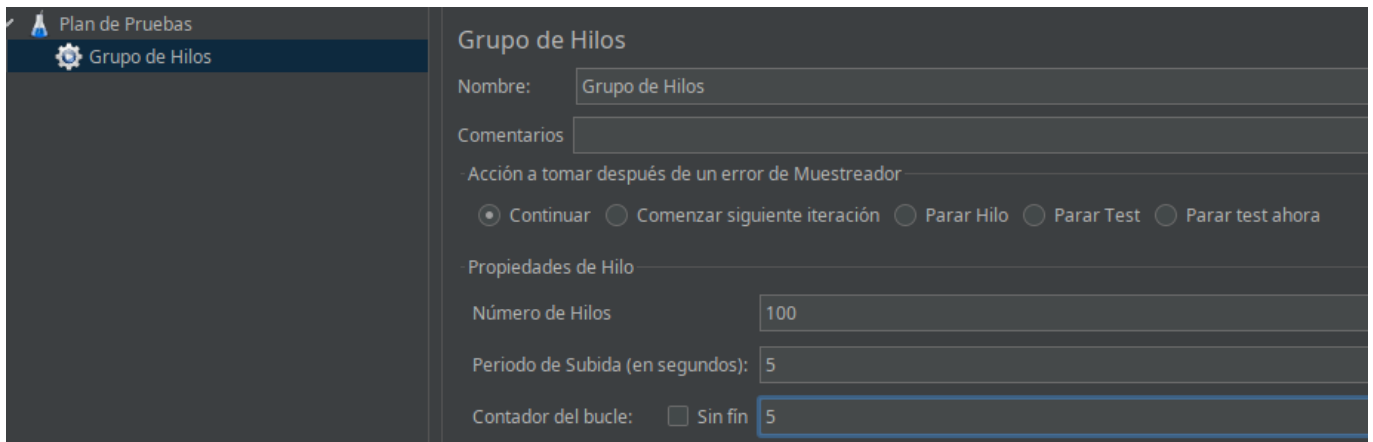
Para ejecutar Jmeter procedemos a lanzarlo desde la carpeta de los binarios del paquete:

```
~/Descargas/apache-jmeter-5.6.3/bin$ ./jmeter.sh
```

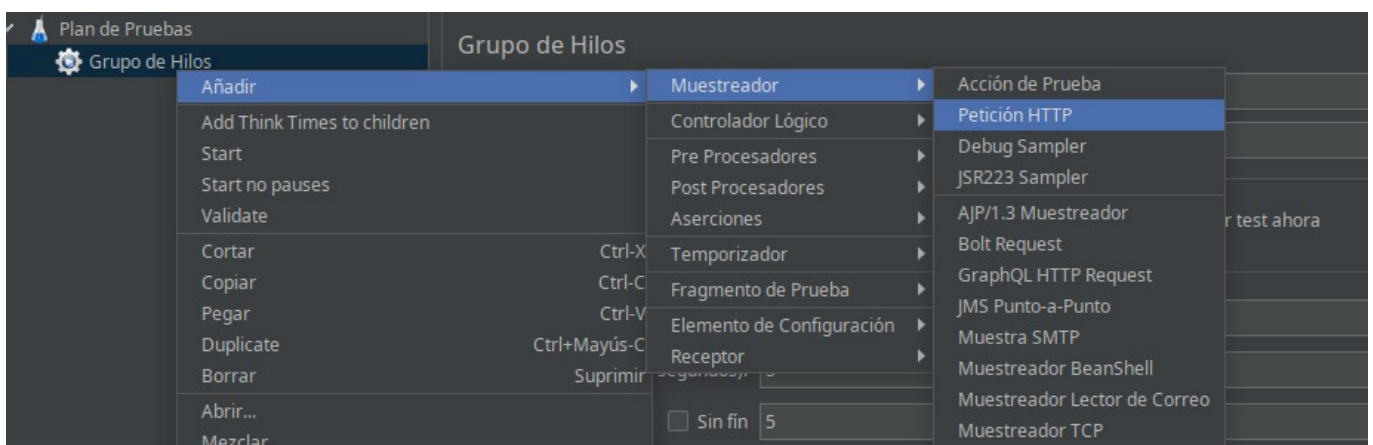
Nada mas entrar en jmeter debemos crear la configuración de la prueba para ello primero creamos el grupo de hilos.



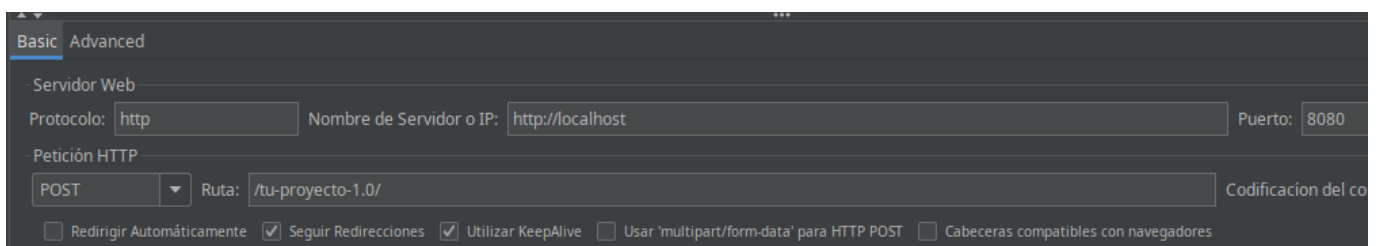
Aquí seleccionaremos el Número de hilos, numero de usuarios que vamos a simular, el periodo de subida, en cuanto tiempo se enviaran los accesos de los usuarios que hemos elegido anteriormente y el contador del bucle, cuantas veces se ejecutará nuestro grupo de hilos.



Ahora desde el Grupo de Hilos, seleccionamos el tipo de muestra que vamos a realizarla, en este caso sera mediante petición http al servidor.



Dentro de la petición debemos especificar los campos que usaremos para poder realizar las peticiones al servidor:



En este apartado es donde añadiríamos el script en javascript para realizar acciones en la web.

Aparte de estas configuraciones con la que ya estamos realizando las consultas a la web, podemos añadir un temporizador, para hacer que estas consultas sean mas lentas y simulen a un ser humano, una aserción de respuesta, para asegurarnos de obtener la respuesta que queremos y también podríamos añadir un listener para visualizar los resultados que hemos obtenido directamente en jmeter.

5. Conclusiones y propuestas para continuar el desarrollo del proyecto.

Con el proyecto he logrado establecer un sistema de monitorización robusto y efectivo para el servidor WildFly. Las herramientas seleccionadas han demostrado ser eficaces para la visualización de métricas, gestión de registros y seguimiento distribuido. La implementación de Grafana, Prometheus, Loki y Zipkin proporcionan una vista integral del rendimiento y la salud del servidor, permitiendo identificar y resolver problemas de manera eficiente y mejorando la capacidad de respuesta y estabilidad del servidor.

Propuestas para futuros trabajos:

1. Integrar alertas automáticas:

- **Objetivo:** Configurar alertas automáticas basadas en las métricas y registros recopilados.
- **Ventajas:** Permitiría una respuesta proactiva a problemas potenciales antes de que afecten gravemente el rendimiento del servidor.
- **Herramientas:** Utilizar las capacidades de alertas de Prometheus y Grafana para enviar notificaciones a través de correo electrónico, Slack o SMS, por ejemplo, cuando se detecten los valores especificados.

2. Explorar el uso de otros frameworks de pruebas de carga y estrés:

- **Objetivo:** Comparar la eficacia de diferentes herramientas de prueba de carga como JMeter, Artillery, Gatling y Locust.
- **Ventajas:** Determinar la herramienta que ofrece la mejor cobertura y resultados más precisos para las necesidades específicas del proyecto.
- **Método:** Realizar pruebas comparativas en escenarios idénticos y analizar los resultados en términos de facilidad de uso, profundidad de análisis y precisión.

3. Evaluar la migración a un proveedor de nube:

- **Objetivo:** Considerar la posibilidad de migrar el entorno de monitorización a un proveedor de nube (AWS, GCP, Azure).
- **Ventajas:** Mejorar la escalabilidad, flexibilidad y disponibilidad del sistema de monitorización. Reducir la carga sobre la infraestructura local y aprovechar los servicios gestionados en la nube.
- **Pasos a seguir:** Evaluar los costos, beneficios y posibles desafíos de la migración. Realizar una prueba piloto en la nube y comparar el rendimiento y la estabilidad con la configuración actual.

4. Implementar una mayor automatización en la configuración y despliegue:

- **Objetivo:** Utilizar herramientas como Ansible, Terraform o Kubernetes para automatizar la configuración y despliegue del entorno de monitorización.
- **Ventajas:** Reducir el tiempo y esfuerzo manual requerido para la configuración inicial y las actualizaciones. Garantizar la consistencia y reproducibilidad del entorno de monitorización.
- **Método:** Desarrollar scripts de automatización y plantillas de configuración. Probar la automatización en un entorno de desarrollo antes de desplegar en producción.

5. Optimizar la recolección y almacenamiento de datos:

- **Objetivo:** Mejorar la eficiencia y la velocidad de recolección y almacenamiento de datos para reducir la carga sobre el sistema.
- **Ventajas:** Aumentar la capacidad de respuesta y reducir el tiempo de inactividad del servidor.
- **Método:** Revisar y optimizar las configuraciones actuales de Prometheus, Loki y Zipkin para un rendimiento óptimo. Considerar el uso de bases de datos de series temporales más eficientes si es necesario.

6. Expandir el alcance de la monitorización:

- **Objetivo:** Incluir otros servicios y aplicaciones en el sistema de monitorización para obtener una vista más completa del entorno.
- **Ventajas:** Proporcionar una monitorización más integral y detallada del ecosistema de aplicaciones y servicios.
- **Método:** Integrar más servicios en la configuración de Prometheus y Grafana. Asegurarse de que todas las aplicaciones críticas estén cubiertas por el sistema de monitorización.

6. Dificultades encontradas.

Durante el desarrollo del proyecto, se encontraron varias dificultades que requirieron soluciones creativas y persistencia:

1. Configuración inicial y compatibilidad de versiones:

- **Problema:** Asegurar que todas las herramientas (Grafana, Prometheus, Loki, Zipkin) fueran compatibles entre sí y con el servidor WildFly.
- **Solución:** He realizado múltiples pruebas y consultas a la documentación oficial para encontrar versiones compatibles y configurar adecuadamente los archivos de configuración.

2. Problemas de rendimiento:

- **Problema:** La carga simultánea de múltiples servicios en una sola máquina afectó el rendimiento general de mi sistema.
- **Solución:** Optimizar la configuración de los contenedores Docker y ajustar los recursos asignados a cada servicio. Realizar un seguimiento constante del uso de recursos y ajustar según sea necesario.

3. Ajustes en la configuración de red y permisos:

- **Problema:** Configurar correctamente la red y los permisos para permitir la comunicación adecuada entre los contenedores Docker y el servidor WildFly instalado directamente en la máquina.
- **Solución:** Ajustar la configuración de Docker Compose y los permisos del sistema operativo. Revisar documentación de Wildfly para poder enviar las métricas de forma correcta a los contenedores.

4. Curva de aprendizaje:

- **Problema:** Integrar y personalizar herramientas como Grafana, Prometheus y Loki requirió bastante tiempo y esfuerzo debido a la falta de experiencia previa.
- **Solución:** Invertir tiempo en el visionado y en la lectura de tutoriales y documentación oficial. Experimentar con diferentes configuraciones y ejemplos hasta lograr una integración exitosa y eficiente. Participar en comunidades y foros en línea para obtener consejos y resolver dudas. El uso de Java también requirió mucho tiempo de practica.

Estos desafíos fueron superados mediante una combinación de investigación, pruebas exhaustivas y ajustes continuos, a parte de la prueba error lo que finalmente llevó a la creación de un sistema de monitorización eficaz y robusto, del que estoy bastante orgulloso.

7. Bibliografía.

7.1. WildFly

- [Instalar WildFly/JBoss en Debian 12](#)
- [Habilitar acceso remoto en WildFly](#)
- [Aplicaciones de ejemplo para realizar tests en servidores WildFly/JBoss EAP](#)

7.2. Grafana

- [Cómo configurar Grafana y Prometheus usando Docker](#)

7.3. Prometheus

- [Configuración de Prometheus y Docker Compose en Linux](#)

7.4. Node Exporter

- [Node Exporter para Prometheus](#)

7.5. Loki

- [Instalación de Loki usando Docker](#)

7.6. Promtail

- [Configuración de Promtail para enviar logs a Loki](#)

7.7. Zipkin

- [Repositorio Docker de Zipkin](#)

7.8. Selenium

- [Selenium y python](#)
- [Selenium documentación oficial](#)

7.9. JMeter

- [Artillery y pruebas de carga con Docker](#)
- [Jmeter ejemplos](#)
- [Jmeter documentación oficial](#)