Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implantación de herramientas de observabilidad del código en una malla de microservicios

Autor: Carlos Martínez Lobato

Tutor: Óscar García Población

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implantación de herramientas de observabilidad del código en una malla de microservicios

Autor: Carlos Martínez Lobato

Tutor: Oscar García Población

TRIBUNAL:

Presidente: Antonio Da Silva Fariña

Vocal 1º: Juan Ignacio García Tejedor

Vocal 2º: Oscar García Población

FECHA: 02/06/2021

RESUMEN

Este proyecto tiene como objetivo la implantación de herramientas de observabilidad en una aplicación diseñada bajo el patrón de arquitectura de microservicios. Estas herramientas se apoyan en una tecnología llamada *service mesh* o malla de servicios que encapsulan los microservicios creando un ecosistema que hace más sencilla su observabilidad y análisis de comportamiento.

Durante el proceso de implantación se hará un estudio de las tecnologías actuales así como también comparaciones para elegir cuál es la mejor manera de implementar estas herramientas. Se realizará también una prueba de concepto con una aplicación de ejemplo ya creada donde se utilizará un software de pruebas de carga para simular tráfico y así obtener muestras analizables.

Palabras clave

Observabilidad. Malla de servicios. Microservicios. Trazabilidad distribuida. Istio

ABSTRACT

This project aims to implement observability tools in an application designed under the microservices architecture pattern. These tools are supported by a technology called service mesh that encapsulates these microservices creating an ecosystem that makes their observability and behavior analysis easier.

During the implementation process, a study of current technologies will be made, as well as comparisons to choose the best way to implement these tools. A proof of concept will also be carried out with an example application already created where load testing software will be used to simulate traffic and thus obtain analyzable samples.

Key words

Observability. Mesh service. Microservices. Distributed tracing. Istio.

ÍNDICE

RESUMEN **1**

ABSTRACT **2**

ÍNDICE **3**

ÍNDICE DE FIGURAS **5**

1. Introducción **7**

2. Objetivos de trabajo fin de grado **9**

3. Estado del arte **11**

3.1. Conceptos básicos relacionados con el tema 11

3.1.1. Arquitectura monolítica 11

3.1.2. Arquitectura de microservicios 12

3.1.3. Service mesh 14

3.1.3.1. Plano de datos y Plano de control 15

3.2. Tecnologías y herramientas para implementar el service mesh 17

3.2.1. Contenedores y Docker 17

3.2.2. Orquestación de contenedores y Kubernetes 17

3.2.2.1. Arquitectura de Kubernetes 19

3.2.3. Capa de malla de servicios (mesh service) 22

3.2.3.1. Istio 22

3.2.3.2. Linkerd 23

3.2.3.3. Consul 23

3.2.3.4. Conclusión 23

3.2.4. Arquitectura de Istio 24

3.2.5. Características de Istio 25

3.2.5.1. Gestión del tráfico 25

3.2.5.2. Gestión de la seguridad 26

3.2.5.3. Gestión de la observabilidad 27

3.2.6. Integraciones en Istio 33

3.2.6.1. Prometheus 33

3.2.6.2. Grafana 35
3.2.6.3. Jaeger 35
3.2.6.4. Kiali 38
3.2.7. Locust 40
4. Implementación de la malla de servicios 43
4.1. Diseño de la aplicación de microservicios. 43
4.1.1 Plantilla HTML 44
4.1.2 Servicio Productpage 46
4.1.3 Servicio Details 51
4.1.4 Servicio Reviews 53
4.1.5 Servicio Ratings 59
4.2. Instalación 60
4.2.1 Máquina virtual y sistema operativo 60
4.2.2. Docker 61
4.2.3. Kubernetes y minikube 62
4.2.4. Istio 64
4.2.5. Despliegue de la aplicación Bookinfo 65
4.2.6. Kiali 70
4.2.7 Trazas distribuidas con Jaeger 74
4.3. Desinstalación y limpieza 76
5. Conclusiones 79
6. Bibliografía y referencias 81

ÍNDICE DE FIGURAS

Fig. 1. *Arquitectura monolítica* 12

Fig. 2. *Arquitectura microservicios* 12

Fig. 3. *Plano de datos con los servicios y sus sidecars* 15

Fig. 4. *Plano de datos, pero sólo sidecars. Malla de servicios* 16

Fig. 5. *Plano de control en malla de servicios* 16

Fig. 6. *Arquitectura de Kubernetes* 19

Fig. 7. *Nodo master en Kubernetes* 20

Fig. 8. *Nodo Worker en Kubernetes* 21

Fig. 9. *Componentes de la arquitectura de Istio* 24

Fig. 10. *Componentes de la seguridad en Istio* 27

Fig. 11. *Panel de control de Grafana* 29

Fig. 12. *Seguimiento petición mediante trazabilidad distribuida* 31

Fig. 13. *Trazabilidad en Istio* 32

Fig. 14. *Componentes de la arquitectura de Prometheus* 34

Fig. 15. *Definición de trazas y tramos* 36

Fig. 16. *Componentes de la arquitectura de Jaeger.* 37

Fig. 17. *Ejemplo de trazas en Jaeger* 38

Fig. 18. *Arquitectura de los componentes de Kiali* 39

Fig. 19. *Ejemplo de test en Locust con 300 usuarios virtuales* 41

Fig. 20. *Aplicación web demo Bookinfo proporcionada por Istio* 43

Fig. 21. *Esquema de la aplicación Bookinfo* 44

Fig. 22. *Servicio reviews-v1* 54

Fig. 23. *Servicio reviews-v2* 54

Fig. 24. *Servicio reviews-v3* 54

Fig. 25. *Características máquina virtual en VirtualBox* 61

Fig. 26. *Resultado de comprobación versión de Docker* 62

Fig. 27. *Resultado comprobación versión minikube instalada* 63

Fig. 28. *Resultado de instalación de kubernetes y minikube en local* 63

Fig. 29. *Panel de control de Kubernetes desplegado en local* 64

Fig. 30. *Resultado instalación Istio* 65

Fig. 31. *Resultado asignación de etiqueta namespace* 65

Fig. 32. *Resultado despliegue aplicación Bookinfo* 66

Fig. 33. *Estado de los servicios* 66

Fig. 34. *Estado de los pods, parte 1* 67

Fig. 35. *Estado de los servicios, parte 2* 67

Fig. 36. *Ingress gateway asociada al despliegue* 67

Fig. 37. *Comprobación de que todo va bien* 68

Fig. 38. *Resultado establecimiento de puerto e ip del servicio* 68

Fig. 39. *Aplicación desplegada en la dirección mostrada por consola* 69

Fig. 40. *Panel de control Kubernetes con la aplicación desplegada* 69

Fig. 41. *Resultado de rollout de Kialli* 70

Fig. 42. *Grafo de servicios interconectados en Kiali* 71

Fig. 43. *Gestión del tráfico* 72

Fig. 44. *Tráfico según porcentajes de carga de trabajo* 73

Fig. 45. *Resultado despliegue Jaeger* 74

Fig. 46. *Buscador de trazas del panel de control de Jaeger* 75

Fig. 47. *Análisis de primera traza en Jaeger* 76

Fig. 48. *Análisis de una segunda traza en Jaeger* 76

Fig. 49. *Resultado limpieza del despliegue* 77

Fig. 50. *Resultado de consulta de servicios tras limpieza de del despliegue* 77

1. Introducción

En los últimos años se han estado produciendo cambios con respecto a las arquitecturas empleadas a la hora de diseñar nuevas aplicaciones. Atrás está quedando la arquitectura de una gran aplicación monolítica cuyos componentes están integrados en un mismo sistema y que realiza todas las funciones del negocio y un tipo de arquitectura basada en microservicios se está abriendo camino desde un tiempo a esta parte.

Se entiende como microservicios a las unidades funcionales e independientes que cooperan juntas para ofrecer la funcionalidad general de una aplicación. Esta división en unidades funcionales permiten que cada microservicio pueda ser actualizado o escalado de manera independiente sin que afecte a la disponibilidad del resto de unidades.

Una de las aplicaciones de ejemplo diseñadas con esta arquitectura es Spotify. La cual está compuesta por más de 800 microservicios distintos y sus más de 345 millones de usuarios activos hace que sus servidores reciban millones de peticiones por minuto.

Con la expansión y éxito de este tipo de arquitecturas basadas en microservicios se dan una serie de condiciones y desafíos tecnológicos que no eran necesarios de solventar en las arquitecturas clásicas.

Podemos tomar como un ejemplo de esos desafíos la trazabilidad de las peticiones: en las arquitecturas monolíticas clásicas, resolver una petición de una aplicación, implicaba que ésta fuese resuelta de manera completa dentro de la propia aplicación, exceptuando casos en los que se requiriese una interacción externa con la base de datos o un servicio de autenticaciòn.

Por otro lado, trabajar con arquitecturas de microservicios basados en el principio de responsabilidad única conlleva la utilización de múltiples servicios por los que va a pasar una petición antes de presentar una respuesta, lo cual es un inconveniente si se desea conocer el transcurso de esa petición a lo largo de todos los microservicios.

Por tanto se nos presenta la siguiente cuestión: ¿De qué manera podemos determinar el transcurso de nuestra petición? Con esta respuesta podríamos solventar otros problemas, como por ejemplo, solucionar algún bug y depurar nuestra aplicación. Este TFG, por tanto, tendría como objetivo analizar el comportamiento de los servicios con respecto a las

peticiones que se llevan a cabo dentro de la arquitectura de microservicios apoyándose en herramientas que facilitan este análisis.

Al final de este TFG se habrá comprendido acerca de los dos tipos de arquitecturas: monolíticas y basada en microservicios. Se habrá aprendido también sobre qué es la malla de servicios, como se implementa y las ventajas que ello supone así como también las herramientas que se pueden aplicar sobre la malla para sacar partido de esta funcionalidad. Para demostrar estos conocimientos se llevará a cabo un ejemplo práctico de instalación de malla de servicios y la utilización de las herramientas sobre una aplicación construida en arquitectura basada en microservicios.

2. Objetivos de trabajo fin de grado

El presente Trabajo de Fin de Grado tiene como objetivos:

- Creación de una prueba de concepto con una malla de servicios.

Se hará una investigación del estado actual de la tecnología y de las diferentes herramientas que proporcionan la implementación de la capa de malla de servicios. Se partirá de una aplicación creada con varios servicios y finalmente se llevará a cabo la implementación.

- Despliegue de herramientas sobre la malla de servicios que nos permitan analizar y/o modificar el comportamiento de los microservicios.

Apoyándose sobre la malla de servicios creada, se desplegarán una serie de herramientas que nos permitan analizar el comportamiento así como modificarlo. Entre esas herramientas se encontrará una herramienta que nos permita visualizar la malla de servicios en una interfaz gráfica y nos permita ver las interconexiones entre los servicios; otra herramienta que nos permita visualizar el tráfico e incluso modificarlo para liberar de carga de trabajo algunos de estos servicios; y finalmente, otra herramienta que nos permita analizar las trazas distribuidas para así obtener información útil entre las distintas llamadas que los microservicios hacen entre ellos. Esta información constará de los tiempos de ejecución o por dónde ha pasado una petición hasta su final.

- Utilización de Locust para la generación de pruebas de estrés. y así obtener diferentes cargas de usuario.

Se utilizará este software de pruebas de carga para simular tráfico real sobre la aplicación y así observar el comportamiento y obtener las diferentes trazas con las que documentar la memoria.

3. Estado del arte

Para la realización de este TFG se han conducido una serie de investigaciones sobre el tema para conocer el estado actual de las tecnologías de microservicios y hacia qué dirección se está innovando. Es por ello que utilizaré esta sección para hablar sobre los conceptos importantes relacionados con el tema a tratar.

3.1. Conceptos básicos relacionados con el tema

Primero, se va a analizar qué es la arquitectura monolítica y qué es la arquitectura basada en microservicios: sus características principales, ventajas e inconvenientes. Finalmente se explicará qué es la malla de servicios o *service mesh* y qué elementos la componen.

3.1.1. Arquitectura monolítica

Se considera al modelo tradicional unificado para el diseño de un programa de software. En el cual, todo el software está puesto en una sóla pieza autónoma que alberga todos los componentes del programa interconectados entre sí haciendo que todos los aspectos funcionales del mismo quedan acoplados y sujetos en un mismo programa. Todos los componentes deben de estar presentes para que el código se pueda ejecutar o compilar.

Si se quisiese actualizar algún componente en concreto sería necesario volver a escribir toda la aplicación y compilarla de manera completa para enviarla en su próxima versión. Por otro lado, este tipo de arquitectura presenta mejor rendimiento que los enfoques modulares como la arquitectura de microservicios y pueden ser más fáciles de depurar.

Por tanto, para mejorar aspectos como el desacoplamiento de los componentes y así solventar la problemática de modificar uno de los componentes sin afectar a toda la aplicación o corregir problemas de escalabilidad hace que nazca la arquitectura de microservicios

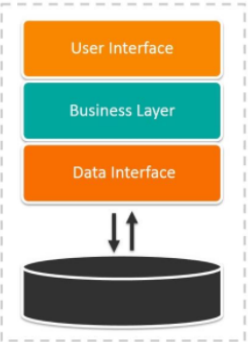


Figura 1. *Arquitectura monolítica.*[2]

3.1.2. Arquitectura de microservicios

Según la Wikipedia, “la arquitectura de microservicios o también conocido por las siglas MSA (Micro Services Architecture), es una aproximación para el desarrollo de software que consiste en construir una aplicación como un conjunto de pequeños servicios los cuales se ejecutan en su propio proceso y se comunican con mecanismos ligeros entre ellos.” Normalmente a través de APIs[1].

“Cada servicio se encarga de implementar una funcionalidad completa del negocio y es desplegado de forma independiente. Es posible que cada uno de estos servicios esté programado en distinto lenguaje o que usen diferentes tecnologías de almacenamiento de datos”[1].

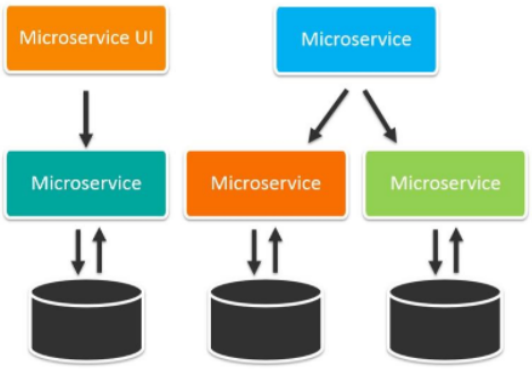


Figura 2. *Arquitectura microservicios.*[2]

Esta arquitectura nos permitiría estructurar nuestra aplicación con una colección de servicios cuyas características serían:

- Desplegables de manera independiente. Es decir, un cambio en un módulo no afectará al resto de componentes. Haciendo que sean mantenibles y comprobables

por si solas.

- Débilmente acopladas.

- Organizadas según la idea de negocio que cubran por lo que haría más fácil entender su estructura y su diseño.

- Entrega e implementación continua de aplicaciones grandes sin tener que desplegar la aplicación al completo

- Facilidad para escalar a nivel de software, es decir, si uno de los componentes de la aplicación necesita replicarse o realizar las tareas con más potencia sólo es necesario enfocarnos en ese microservicio

- Permite a los equipos de desarrollo trabajar de manera separada en los módulos independientes sin necesidad de estar en continua colaboración con otros módulos.

A simple vista parece que la arquitectura en microservicios es mucho mejor que la arquitectura monolítica pero en realidad nos enfrentaremos a otros problemas como:

- Incremento del consumo de memoria al tener cada microservicio sus propios recursos y bases de datos consumirá más memoria y CPU.

- Inversión de tiempo inicial en realizar el estudio de fragmentación y el desarrollo de la comunicación entre los distintos microservicios que compondrán el programa.

- Complejidad de gestión de un gran número de servicios. Será muy complicado controlar una gestión e integración de los mismos si hubiese gran cantidad de ellos. Por lo que sería necesario depender de una centralización de trazas o herramientas avanzadas de procesamiento de información que permitan tener una visión general de todos los microservicios. Así como una herramienta de despliegues automatizados.

- Necesidad de desarrolladores experimentados que tengan conocimientos para la solución de problemas como latencia en la red o balanceo de cargas

- Pruebas o testeos complicados tras un despliegue distribuido.

Para tener un control más exhaustivo de todo lo que está pasando en nuestros microservicios: métricas, seguridad, monitorización, trazabilidad distribuida de las peticiones… Se podría pensar en una solución en la que cada microservicio lance sus propios mensajes de estado pero eso haría incrementar la complejidad de todo el sistema por lo que se llegó a otra solución en la cual las tareas de monitorización sean llevadas a una capa diferente que encapsule a todos los microservicios. Esta capa es conocida como *service mesh* o malla de servicios.

3.1.3. Service mesh

Service mesh o malla de servicios es una funcionalidad al ecosistema de microservicios que puede ser considerado como una infraestructura de software encargada de gestionar la comunicación entre microservicios haciéndose responsable de que las solicitudes de los servicios sean entregadas de manera confiable y segura.[3]

Una correcta implementación del service mesh se considera que debe reunir las siguientes características:

- Descubrimiento de servicios: descubrimiento de los endpoints del servicios a través de un registro de servicio dedicado.
- Capacidades básicas que aseguren la disponibilidad en la comunicación entre servicios: patrones de Circuit-breaking, reintentos y tiempos de espera, gestión de errores y balanceo de cargas.
- Capacidad de enrutamiento: enrutamiento de peticiones a versiones diferentes de servicios.
- Observabilidad: métricas, monitorización, creación de registros y trazabilidad distribuida
- Seguridad: seguridad a nivel de transporte (TLS) y gestión de claves.
- Protocolos de comunicación entre servicios
- Despliegues en contenedores (Docker, Kubernetes…)

Para poder llevar a cabo la funcionalidad del service mesh se utilizan algunos de los patrones de diseño de aplicaciones distribuidas ya conocidos pero en vez de hacerlo

directamente sobre el propio código de la aplicación se realiza sobre la infraestructura, este patrón es conocido como **patrón sidecar.**.

Esto significa que la funcionalidad de un servicio principal se extiende o mejora mediante otro servicio paralelo sin apenas acoplamiento entre ambos. Este patrón es particularmente útil cuando se utiliza Kubernetes como plataforma de orquestación de contenedores ya que como veremos posteriormente en el desarrollo de este TFG, explicamos que en Kubernetes se utilizan pods para encapsular servicios; cada uno de los cuales está compuesto por uno o más contenedores de aplicaciones.

3.1.3.1. Plano de datos y Plano de control

A continuación se va a explicar los principios sobre los que se sustenta el *service mesh.* Para comenzar, vamos a explicar qué son el plano de datos y el plano de control.

En el plano de datos, cada proceso se comunica con su propio proxy-sidecar en lugar de comunicarse directamente entre ellos, son los sidecars los que terminan siendo los encargados de la entrega fiable de las peticiones.

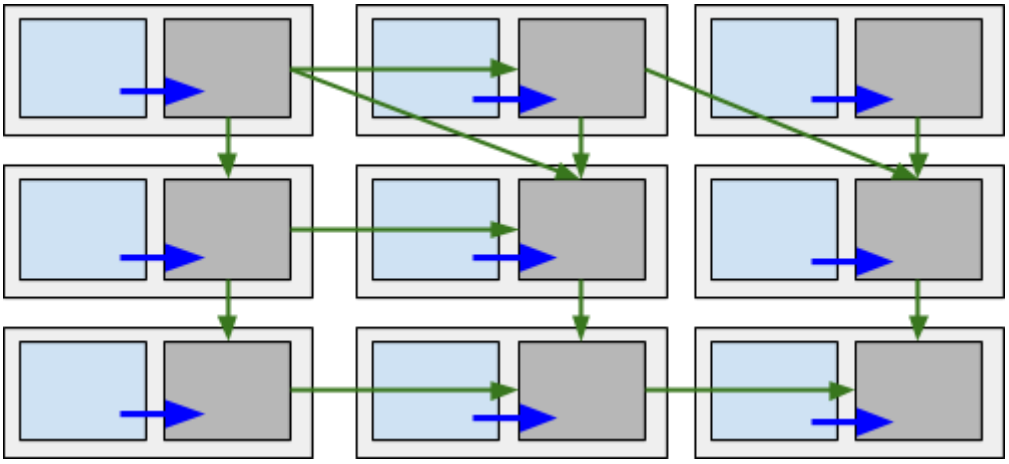


Figura 3. *Plano de datos con los servicios y sus sidecars.*[3]

Si eliminamos los propios servicios de negocio y dejamos los sidecar podemos ver la intercomunicación entre ellos y el diagrama resultante se nos muestra como la malla de conectividad:

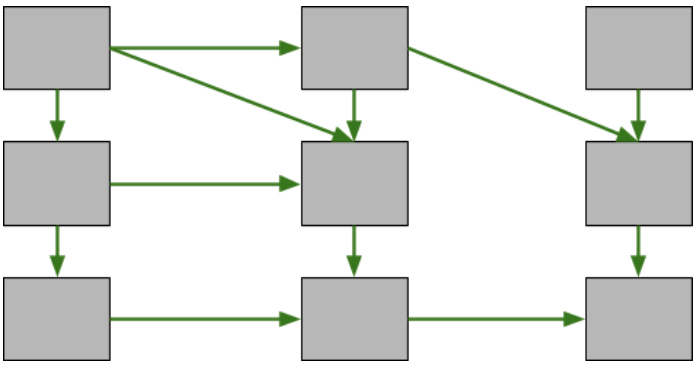


Figura 4. *Plano de datos, pero sólo sidecars. Malla de servicios.*[3]

A esta malla se la llama “Plano de datos”. La responsabilidad del plano de datos es asegurarse de que las peticiones del microservicio A sean transportadas al microservicio B de una manera confiable y segura siendo el encargado de proporcionar las siguientes funcionalidades:

- Descubrimiento de servicios (Service discovery)
- Comprobación de disponibilidad (Health check)
- Enrutamiento y balanceo de carga.
- Securización y control de acceso.
- Observabilidad

El **plano de control** se encarga de gestionar, configurar y controlar todas las instancias de proxy-sidecars. Convirtiéndose en el elemento que implementa las políticas de control, monitorización, recolección de métricas, registros de acceso.

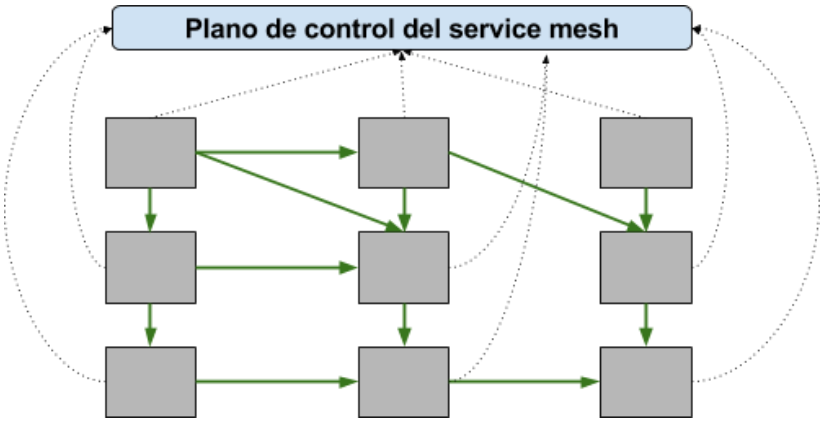


Figura 5. *Plano de control en malla de servicios.*[3]

3.2. Tecnologías y herramientas para implementar el *service mesh*

Para este trabajo se ha hecho una investigación sobre las tecnologías y las distintas herramientas que nos ayudarán tanto a implementar la capa de *service mesh* así como para desplegar los propios microservicios. En este apartado se describirán todas aquellas herramientas utilizadas y un estudio de las diferentes alternativas que nos ayudaron a tomar la decisión final.

3.2.1. Contenedores y Docker

Los contenedores son una virtualización a nivel de sistema que permite la existencia de múltiples instancias aisladas entre sí que pueden resultar ordenadores reales desde la visión de los programas que se lanzan en ellos. Estos contenedores nos permiten desplegar aplicaciones independientes entre ellas mediante generación de entornos virtuales replicables pero a diferencia de las máquinas virtuales, los contenedores pueden compartir los recursos del propio sistema operativo host sobre el que se ejecutan lo que los hacen mucho más ligeros ya que nos ahorra la instalación de un sistema operativo por cada aplicación que queramos desplegar por contenedor.

Actualmente Docker, se ha posicionado en el mercado como líder tecnológico en la virtualización basada en contenedores. Existen otras herramientas que permiten la virtualización en contenedores como LXC (Linux containers) y RKT Rocket. Pero debido a la popularidad de Docker y la facilidad de encontrar soluciones a un posible problema se ha elegido esta tecnología para la realización de este TFG.

3.2.2. Orquestación de contenedores y Kubernetes

Si el número de servicios creciese en nuestra aplicación se convertiría en un problema complicado de gestionar. En este caso, Docker no sería suficiente ya que necesitaríamos una coordinación para hacer el despliegue, la supervisión de servicios, reemplazo o escalado automático. Ante esta problemática surge la necesidad de una herramienta para orquestar los contenedores.

Por tanto las funcionalidades principales que debería de tener un orquestador de contenedores serían:

- Subida y despliegue automático de servicios basados en contenedores.
- Repartición de carga.
- Configuración automática.
- Auto-escalado de contenedores.
- Control del estado de salud de cada contenedor.
- Intercambio de información.

Google comprendió que necesitaba una mejor forma de administrar e implementar sus componentes software para escalar a nivel mundial y desarrolló Kubernetes, un sistema de código libre para la automatización de los despliegues y el más popular que existe actualmente en el mercado. Para solventar su necesidad, comenzó a desarrollar este proyecto de manera interna pero finalmente liberó el código.

Docker Swarm es la herramienta que propone Docker para solventar los problemas que surgen a la hora de orquestar y planificar contenedores.

Para orquestar nuestros contenedores necesitaremos un clúster donde poder desplegarlos. **Minikube** es una herramienta para desarrolladores que nos permite la capacidad de tener un clúster local usando *VirtualBox* y poder desplegar nuestros contenedores en nuestro ordenador o máquina virtual y así evitarnos contratar una plataforma en la nube. Es por ello que nos apoyaremos en las tecnología de Kubernetes y minikube para realizar este trabajo.

3.2.2.1. Arquitectura de Kubernetes

Para empezar a conocer Kubernetes es necesario tener claros algunos conceptos clave[4]:

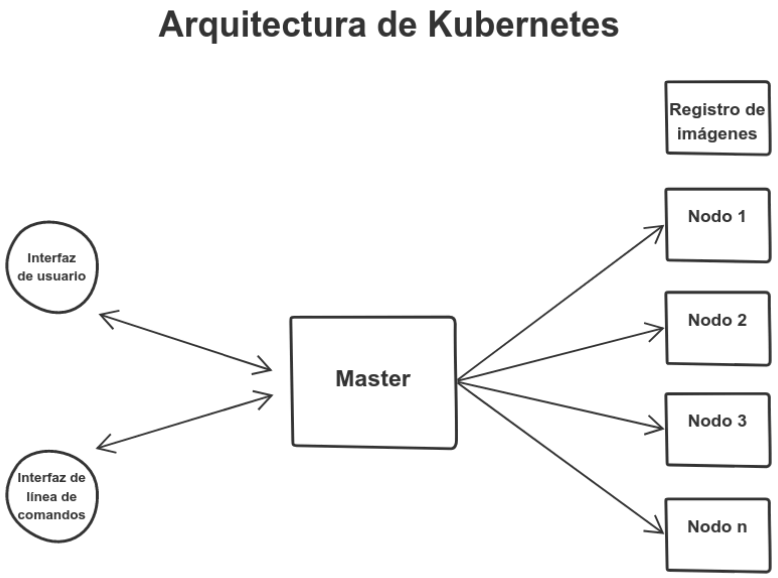
- Pods: se llama pod a una colección lógica de contenedores y recursos compartidos por esos contenedores que pertenecen a una aplicación. Un pod contiene uno o varios contenedores.

- Labels: las etiquetas o labels son pares claves/valor que se asignan a los objetos, por ejemplo, a los pods para especificar atributos de identificación de objetos que sean significativos.

- Servicios: son los end-points que definen cómo acceder a las aplicaciones.

- kubectl es el comando que se utiliza para configurar nuestro clúster o enviar peticiones para realizar los cambios que ordenemos.

- Nodos: son las máquinas que componen el clúster Kubernetes.Estas máquinas pueden ser virtuales o físicas, y estar desplegadas en la nube o en local utilizando minikube. Pueden ser nodos worker o nodos master.[5]



-

Figura 6. *Arquitectura de Kubernetes.*[5]

Los nodos master son los encargados de gestionar el clúster de Kubernetes. Estos nodos toman decisiones globales sobre el clúster como el reparto de trabajo entre los nodos worker, y detectan y dan respuesta a distintos eventos del clúster como puede ser el inicio de nuevos pods cuando el número de réplicas es inferior al configurado.

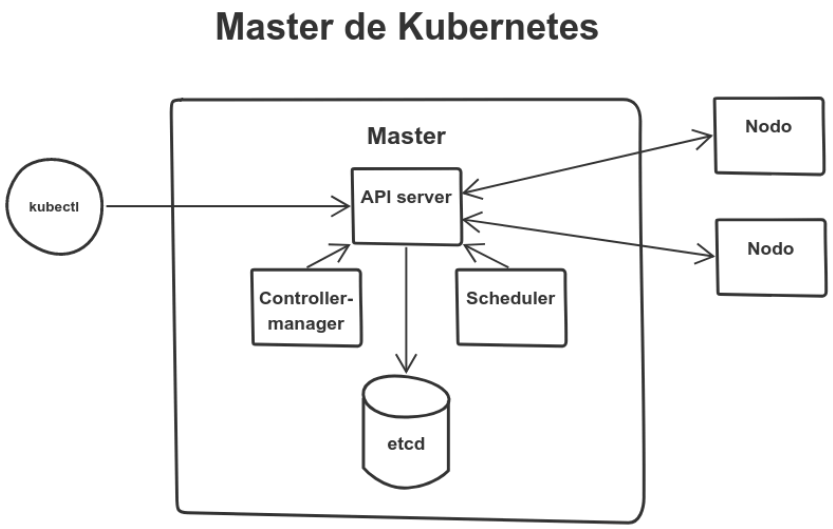


Figura 7. *Nodo master en Kubernetes.*[5]

- API server es el componente que expone la API de Kubernetes y el punto de entrada para todos los comandos REST utilizados para controlar el clúster. Procesa las solicitudes REST, las valida y ejecuta.

- Scheduler es el componente de Kubernetes encargado de observar los pods recién creados que no tienen nodo asignado y decide en qué nodo se ejecuta un determinado contenedor.

- Controller-manager este componente ejecuta los distintos controladores para observar los estados compartidos del clúster y realiza los cambios correctivos en el estado actual para cambiarlo al estado deseado.

- etcd es un almacén de datos de tipo clave-valor que se utiliza para mantener la configuración global del clúster.

Los nodos worker, o simplemente nodos, son los responsables de ejecutar las aplicaciones en pods. Cada nodo worker puede ejecutar múltiples pods. Es muy recomendable tener varios nodos worker si queremos asegurar la disponibilidad de las aplicaciones desplegadas en el clúster Kubernetes.

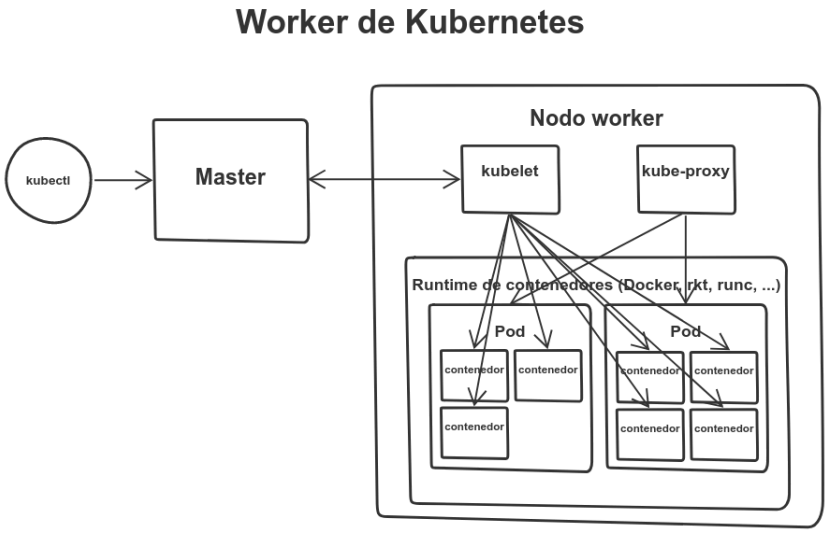


Figura 8. *Nodo Worker en Kubernetes.*[5]

- kubelet es el servicio, dentro de cada nodo worker, responsable de comunicarse con el nodo master. Obtiene la configuración de los pods del API server y garantiza que los contenedores descritos en dicha configuración estén levantados y funcionando correctamente.

- kube-proxy actúa como proxy de red y balanceador de carga enrutando el tráfico hacia el contenedor correcto en función de la dirección IP y el número de puerto indicados en cada petición.

- Runtime de contenedores es el software responsable de ejecutar los contenedores de los pods. Para ello, se encarga de descargar las imágenes virtuales necesarias y de arrancar los contenedores.

3.2.3. Capa de malla de servicios (*mesh service*)

Como previamente se ha contado, un *service mesh* o malla de servicios se puede considerar como una capa de infraestructura que envuelve a todo el conjunto de microservicios que permite que la comunicación entre ellos sea segura, rápida y confiable con capacidades adicionales que nos permiten observar el comportamiento de los microservicios y obtener información de su uso entre otras funcionalidades ya mencionadas.

Actualmente en el mercado existen diferentes tipos de tecnologías que nos facilitan la implementación de la capa *service mesh* en la arquitectura de microservicios.

3.2.3.1. Istio

Istio es una plataforma de código abierto que implementa el *service mesh*. Lanzado en 2017 y escrito en Go. Proporciona una forma de controlar cómo los microservicios comparten datos entre sí. Incluye APIs que permiten que Istio se integre con herramientas que generan datos de telemetría, recolección de métricas y administrar sistema de políticas que gestionan el flujo del tráfico a través de los microservicios de una aplicación.

Por tanto, y según la documentación que Istio ofrece en su página web, esta herramienta proporciona las siguientes funcionalidades para la implementación de un *service mesh*[6]:

- Equilibrio automático de la carga de trabajo de las peticiones HTTP, gRPC, WebSocket y TCP.

- Control exhaustivo del comportamiento del tráfico con reglas de enrutamiento enriquecidas, reintentos, inyección de errores y tolerancia a fallos.

- Una API de configuración junto con una capa de políticas conectable que permiten controles de acceso, cuotas y límites de tasa.

- Creación de registros, métricas automatizadas y trazas para todo el tráfico dentro de un clúster, incluida la entrada y salida de los datos a través del clúster.

- Traspaso de información de manera segura de servicio a servicio en un clúster basadas en identidades sólidas recogidas mediante autenticación y autorización.

3.2.3.2. Linkerd

Lanzado en febrero de 2016. Es un proyecto opensource, fue el primer producto en popularizar el concepto *service mesh*. Está escrito en Scala y corre en una máquina virtual de Java. Al igual que Istio, permite administrar el flujo del tráfico, equilibrio de carga de trabajo, enrutamiento de las comunicaciones entre servicios, manejo de fallas e instrumentalización de datos de telemetría.

Más tarde y tras una serie de críticas, un sector de los creadores de Linkerd hicieron un proyecto aparte llamado Conduit. Una versión más ligera que implementa un *service mesh*. Finalmente Conduit se integró en la versión v2 de Linkerd.

A diferencia de Istio, Linkerd, a la hora de implementar los proxys utiliza Finagle con Jetty en vez de Envoy. Tiene dos tipos de circuit breaker: orientado a sesión y orientado a petición.

3.2.3.3. Consul

Lanzado en 2017. Es una solución de malla de servicios que proporciona un plano de control completo con funciones de detección, configuración y segmentación de servicios. El principal problema de Consul es que no dispone de tantas herramientas de telemetría como las dos anteriores.

3.2.3.4. Conclusión

Después de realizar distintas pruebas y leer documentación de las dos primeras herramientas que implementan el service mesh así como herramientas adicionales para obtener datos de observabilidad, se ha podido llegar a la conclusión de que ambas pueden ser válidas para una solución pero como nuestras decisiones hasta ahora han sido basarnos en la popularidad de la herramienta más utilizada debido a la facilidad de encontrar soluciones en caso de problemas se ha decidido escoger Istio para la realización de este Trabajo de Fin de Grado.

3.2.4. Arquitectura de Istio

El siguiente diagrama muestra los diferentes componentes de la arquitectura de Istio:

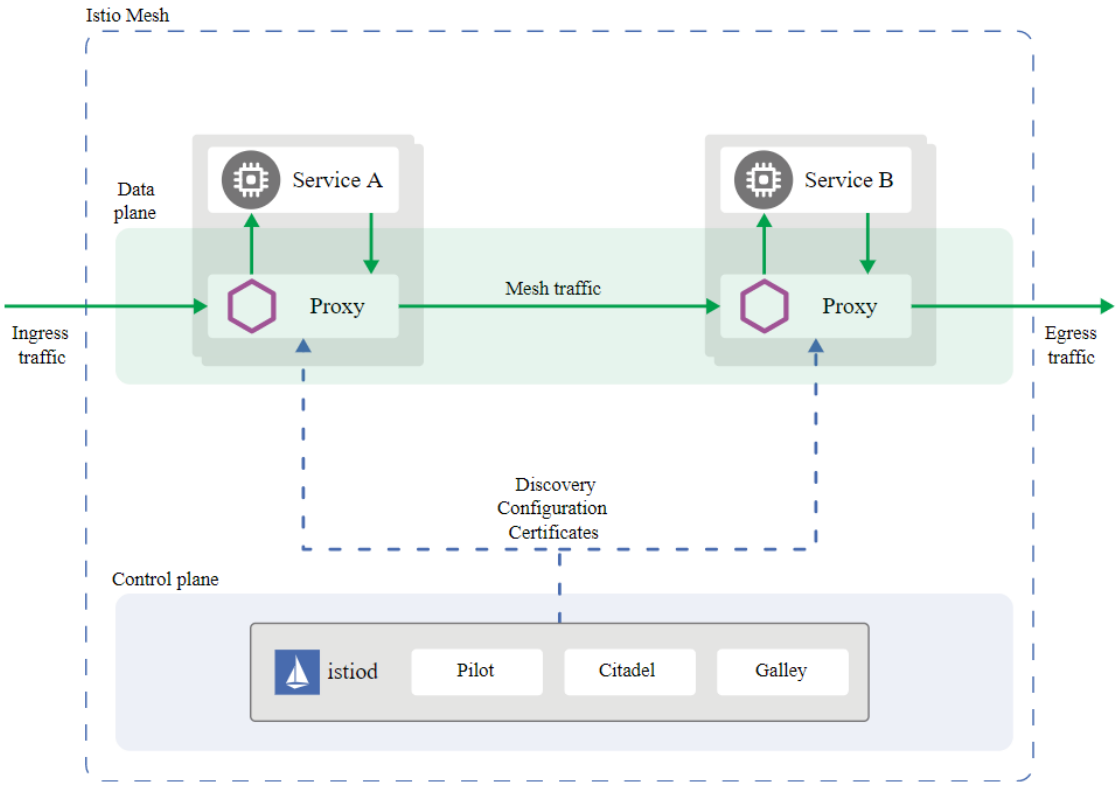


Figura 9. *Componentes de la arquitectura de Istio.*[7]

La malla de servicio de Istio está compuesta por dos planos diferentes, el plano de datos y el plano de control.

El plano de datos está compuesto por un conjunto de proxies inteligentes (Envoy) desplegados como sidecars de los servicios que componen la aplicación mediante el patrón sideca. Estos proxies hacen de intermediarios para controlar toda la comunicación entre los microservicios. Al mismo tiempo, recogen y reportan los datos de telemetría por todo el tráfico de la red.

Istio utiliza Envoy para la implementación de estos proxies. Envoy es un *edge service* o servicio de borde. Es un componente que actúa como punto de acceso para otros servicios. Es un proyecto de código abierto enfocado en las aplicaciones nativas en la nube. Es de alto rendimiento y está escrito en C++.

El plano de control es la parte que se encarga de controlar las reglas de enrutado y configurar los proxies para las políticas de seguridad. Istiod proporciona el descubrimiento de servicios, la configuración y la administración de certificados.

Istiod obtiene las reglas de enrutamiento de alto nivel configuradas en el plano de control y las transforma en configuraciones concretas que Envoy puede interpretar de manera que así sabe cómo actuar frente a las peticiones que atraviesan los sidecars. Estas configuraciones son propagadas a cada uno de los proxies en tiempo de ejecución.

Pilot extrae los mecanismos que permiten descubrir los servicios específicos de la aplicación y los empaqueta en un formato estándar para que cualquier sidecar que conforma el servicio pueda recibir las configuraciones. Es el componente que se encarga de gestionar los registros de servicios, enrutado y resiliencia.

Citadel es la pieza que gestiona la generación y rotación de los certificados. Permite una sólida autenticación de servicio a servicio y de usuario final con identidad y gestión de credenciales incorporadas. Esto hace que Istiod actúe como una Autoridad de Certificación (CA) y que genere certificados que permitan la securización de la comunicación mTLS en el plano de datos.

Galley valida, procesa y distribuye la configuración de Istio. Su responsabilidad es la de obtener la configuración del usuario desde la plataforma y propagar esta información detallada a los demás componentes de Istio..

3.2.5. Características de Istio

Como ya se ha mencionado antes, Istio proporciona una serie de capacidades clave a través de sus servicios[6]. En los siguientes apartados describimos cada uno de estos servicios de forma más detallada

3.2.5.1. Gestión del tráfico

Istio mediante sus reglas de configuración de enrutado del tráfico permite controlar el flujo y las llamadas API entre los servicios que engloba su malla. Esta configuración tiene en cuenta distintas propiedades como los tiempos de espera, los reintentos y los disyuntores y

facilita la disposición de tareas importantes como pruebas A/B[1] y lanzamientos con tráfico controlado basado en porcentajes que permitan la división del flujo.

3.2.5.2. Gestión de la seguridad

Como comentamos al inicio, tras haber fragmentado una aplicación monolítica en pequeños servicios, nos ha proporcionado una serie de beneficios que incluyen una mejor escalabilidad de la aplicación, mayor agilidad en el desarrollo y mantenimiento y la capacidad de reutilizar servicios. Sin embargo se nos presentan una serie de problemas en cuanto a la seguridad:[8]

- Que sea capaz de defenderse contra los ataques de intermediarios. (*Man-in-the-middle-attacks*)

- Proporcionar servicios flexibles de control de acceso y políticas de acceso detalladas.

- Determinar quién ha hecho qué y cuándo mediante herramientas de auditoría.

Por tanto se nos presenta que las capacidades de seguridad de Istio liberan a los desarrolladores de estas tareas de la seguridad en la aplicación que han desarrollado ya que Istio proporciona una comunicación segura por defecto y es capaz de controlar la autenticación, autorización y la encriptación de los servicios. Al mismo tiempo Istio permite reforzar las políticas ya existentes a través de diversos protocolos sin apenas cambios en la aplicación desarrollada.

[1] Según la Wikipedia, “el término prueba A/B (comúnmente en inglés conocido como A/B testing) se utiliza en el ámbito del Marketing Digital y la Analítica web para describir experimentos aleatorios con dos variantes, A y B, siendo una la de control y la otra la variante. Es decir, enrutar un cierto número de peticiones a una versión y el otro restante a otra versión, hacer un estudio para ver cuál es la mejor versión para los usuarios”.[20]

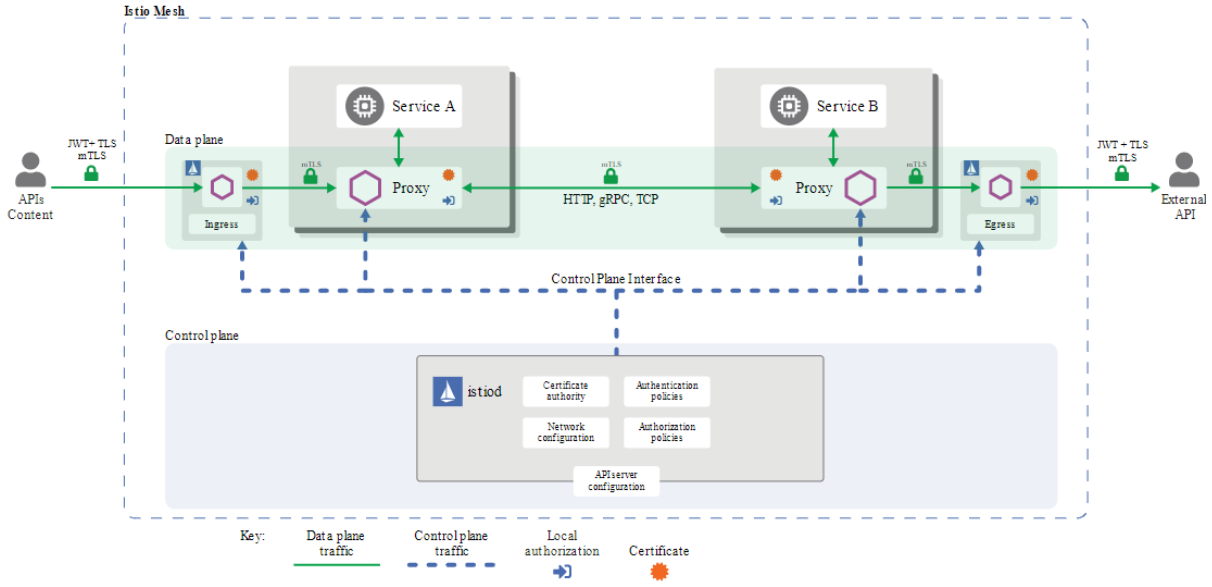


Figura 10. *Componentes de la seguridad en Istio.*[8]

La seguridad en Istio involucra a varios componentes:

- Certificado de Autoridad (CA) para la gestión de claves y certificados

- Una API de configuración del servidor que lo distribuye a los diferentes proxies. Políticas de autenticación, de autorización y securización de información de nombres.

- Los sidecar asociados a los servicios actúan como puntos de aplicación de políticas (*PEP, policy enforcement points*) que hacen de puertas de entrada de las peticiones y securizan la comunicación entre los clientes y servidores.

- El conjunto de proxies Envoy gestionan la información de la auditación de las peticiones.

3.2.5.3. Gestión de la observabilidad

Gracias a que Istio genera una detallada información de telemetría para cada una de las comunicaciones de los servicios que se hacen dentro de la malla, Istio es capaz de proporcionar sólidas funciones de seguimiento, monitoreo y registro de peticiones que

brindan una visión profunda de la implementación de la aplicación para más tarde poder optimizar el comportamiento de ésta debido al conocimiento que se obtiene de entender cómo interactúan nuestros servicios entre ellos. Así como también detectar y corregir posibles errores de una manera rápida y eficiente.[9]

Istio genera los siguientes tipos de datos telemétricos:

Métricas

Istio genera un conjunto de métricas de servicio basado en las cuatro “señales de oro” de la monitorización (latencia, tráfico, errores y saturación) lo que nos proporciona detalles métricos para la malla del plano de datos así como también información detallada para la malla del plano de control. Toda esta información se nos presenta en un panel de control gráfico ya construido de manera predeterminada sobre estas métricas.

Para monitorear el comportamiento de los servicios, Istio genera métricas para todo el tráfico de entrada y salida y también dentro de la malla de servicios. Estas métricas consisten en proporcionar el volumen general del tráfico, las tasas de error dentro del tráfico y los tiempos de respuesta de las solicitudes. Así como una visión global del estado actual de los servicios.

- *Métricas a nivel de proxy:*

La recolección de las métricas de Istio comienzan por los proxies sidecar (Envoy). Cada proxy genera un rico conjunto de métricas de todo el tráfico que pasa a través del proxy (tanto entrada como salida). Los proxies también proporcionan estadísticas detalladas sobre el estado actual del proxy como su configuración y su estado.

Istio permite la activación de operadores para seleccionar cuáles métricas del Envoy son generadas y recogidas por cada instancia de trabajo. Por defecto, Istio permite un pequeño subconjunto de estadísticas generadas para evitar métricas abrumadoras y así reducir la sobrecarga al CPU asociada a la recopilación de datos. Estos operadores permiten expandir al gusto el conjunto de métricas de proxy cuando fuese necesario.

Ejemplo de métricas a nivel de proxy:

```
envoy_cluster_internal_upstream_rq{response_code_class="2xx",cluster_name="xds-grpc"} 7163
envoy_cluster_upstream_rq_completed{cluster_name="xds-grpc"} 7164
envoy_cluster_ssl_connection_error{cluster_name="xds-grpc"} 0
envoy_cluster_lb_subsets_removed{cluster_name="xds-grpc"} 0
envoy_cluster_internal_upstream_rq{response_code="503",cluster_name="xds-grpc"} 1
```

- *Métricas a nivel de servicio*

Istio también proporciona un conjunto de métricas orientadas al servicio para monitorear las comunicaciones de los microservicios. Estas métricas cubren las cuatro necesidades básicas de control: latencia, tráfico, errores y saturación. Istio viene integrado de manera predeterminada con un conjunto de paneles que sirven para observar el resultado de las métricas obtenidas

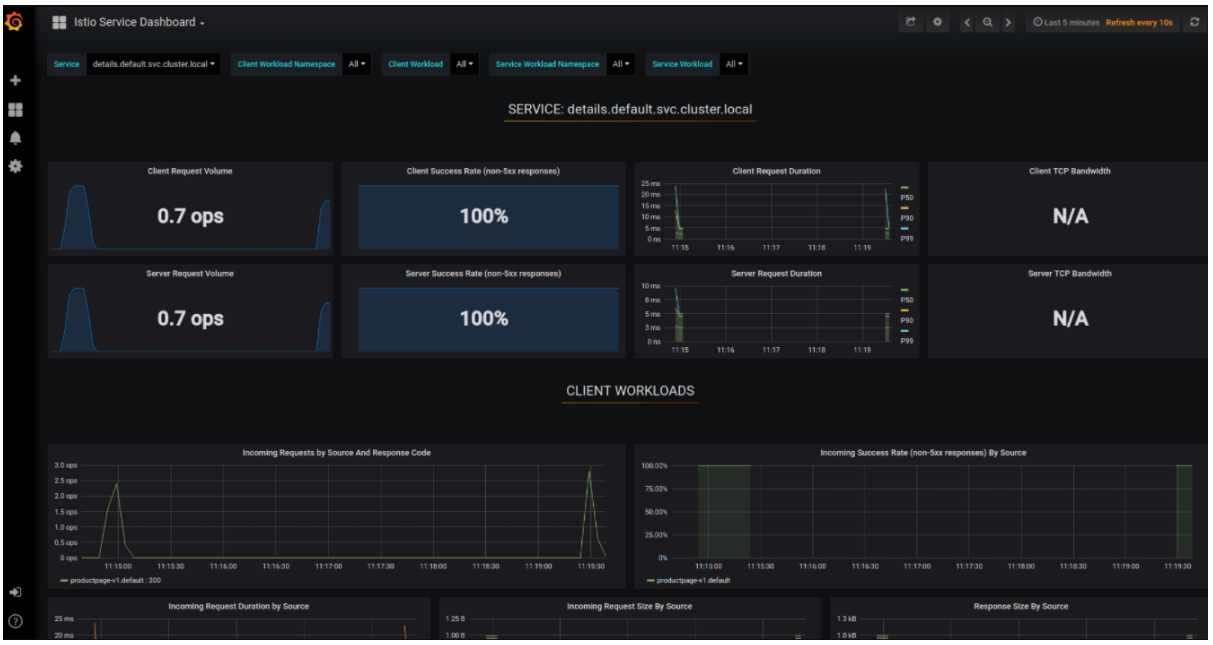


Figura 11. *Panel de control de Grafana.*[9]

Las métricas de Istio están estandarizadas y son exportadas a Prometheus por defecto. Prometheus es un sistema de monitoreo de información open source. Más adelante incluyo más información sobre esta herramienta así como la herramienta que permite la visualización de esta información llamada Grafana.

Ejemplo de métricas a nivel de servicio:

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_app="details",
  destination_canonical_service="details",
  destination_canonical_revision="v1",
  destination_principal="cluster.local/ns/default/sa/default",
  destination_service="details.default.svc.cluster.local",
  destination_service_name="details",
  destination_service_namespace="default",
  destination_version="v1",
  destination_workload="details-v1",
  destination_workload_namespace="default",
  reporter="destination",
  request_protocol="http",
  response_code="200",
  response_flags="-",
  source_app="productpage",
  source_canonical_service="productpage",
  source_canonical_revision="v1",
  source_principal="cluster.local/ns/default/sa/default",
  source_version="v1",
  source_workload="productpage-v1",
  source_workload_namespace="default"
} 214
```

- *Métricas del plano de control*

El plano de control de Istio también proporciona un conjunto de auto-monitoreo de métricas. Estas métricas permiten conocer el propio comportamiento de los componentes de Istio. Entre ellos se incluyen marcas de tiempo, número de errores de autenticación, número de eventos que han sido procesados, etc…

Más información en la documentación de métricas exportadas de Istio.
https://istio.io/latest/docs/reference/commands/pilot-discovery/#metrics

Trazas distribuidas

Las trazas distribuidas permiten una manera de seguir y entender el comportamiento de las peticiones de manera individual al tiempo que atraviesan nuestra malla de servicios a través de nuestros sidecars. Estos seguimientos nos permiten comprender las dependencias de los servicios, localizar errores, detectar posibles latencias durante las peticiones y mejorar el rendimiento. En definitiva, las trazas distribuidas nos permiten estudiar una petición desde su punto origen hasta su destino final al tiempo que es observada por cada uno de los servicios con los que interactúa.[10]

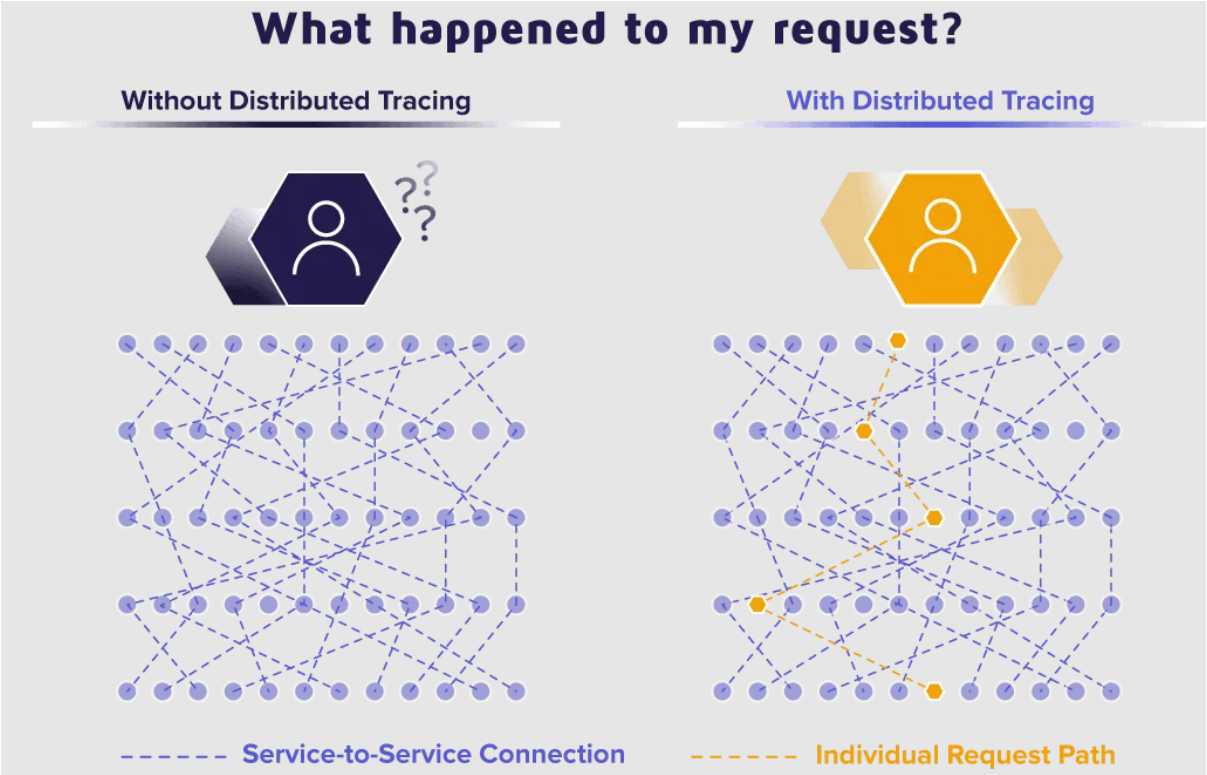


Figura 12. *Seguimiento petición mediante trazabilidad distribuida.*[10]

La trazabilidad distribuida de Istio se apoya en los proxies Envoy. Estos proxies generan de manera automática los tramos de seguimiento en nombre de las aplicaciones sobre las que se apoyan y solo requieren que las aplicaciones reenvíen el contexto de solicitud de manera apropiada. Es decir, a pesar de que los proxies de Istio son capaces de mandar los mensajes de manera automática, de alguna manera la aplicación tiene que propagar las cabeceras correctas de HTTP para que los diferentes tramos de la traza puedan ser correlacionados y unidos bajo una misma traza a través de estas cabeceras que se compondrán con identificadores únicos.

Istio se apoya en las especificaciones de cabeceras "*B3*"[11]. Estas cabeceras son utilizadas para transportar el contexto a través de peticiones y así componer una traza distribuida de manera completa. Las cabeceras son las siguientes:

```
    -   x-request-id
    -   x-b3-traceid
    -   x-b3-spanid
    -   x-b3-parentspanid
    -   x-b3-sampled
    -   x-b3-flags
    -   x-ot-span-context
```

Istio soporta varios sistemas de trazabilidad entre ellos se encuentran Zipkin, Jaeger, Lighstep y Datadog. Los operadores que se habían mencionado en el punto de métricas a nivel de proxy permiten controlar la tasa de muestreo a la hora de generar trazas, es decir, la cantidad de datos de traza generados por cada petición realizada a la aplicación de microservicios.

Ejemplo de una traza distribuida generada por Istio tras una petición:

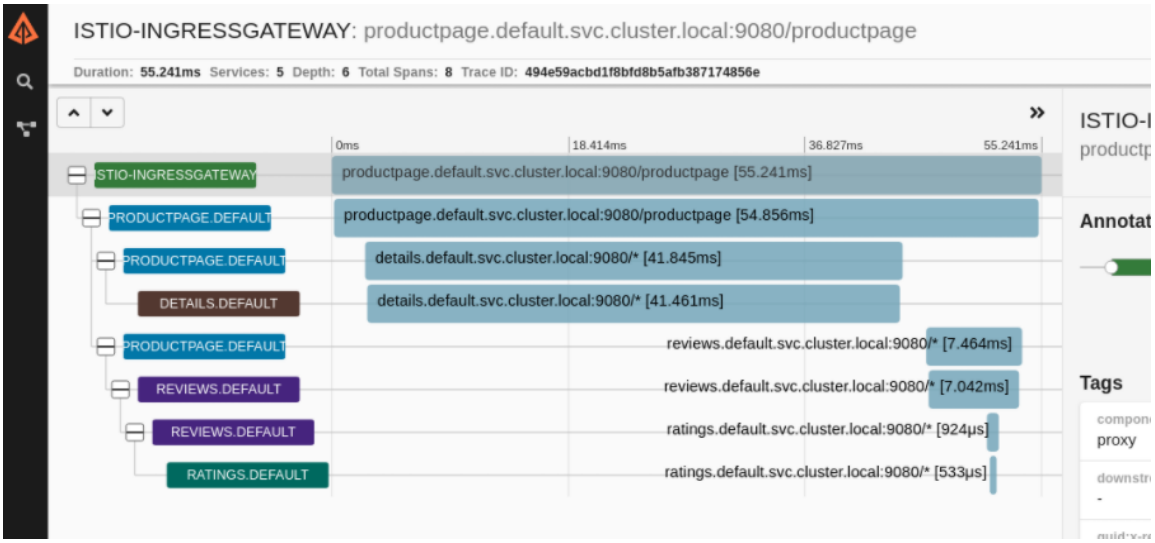


Figura 13. *Trazabilidad en Istio.*[12]

En esta imagen se puede observar la traza de una petición desglosada por los microservicios que la componen. Se apoya en una estructura de árbol siendo la raíz la petición completa donde aparece la cantidad de tiempo total que ha tardado desde su inicio hasta su final. Los microservicios que se han ido ejecutando se muestran desglosados en hojas indicando el orden de ejecución, desde su inicio a lo largo de la petición global, la cantidad de tiempo que han tardado hasta su final.

Logs de acceso

Los registros de acceso proporcionan una manera de monitorizar y comprender el comportamiento de una aplicación desde la perspectiva individual de una instancia de carga de trabajo.

Istio permite generar registros de acceso según el tráfico de los servicios y mediante un conjunto configurable de formatos, proporcionando un control total del cómo, qué, cuándo y dónde registrar.

El formato por defecto del mensaje es[9]:

```
[%START_TIME%] "%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"
%RESPONSE_CODE% %RESPONSE_FLAGS% %BYTES_RECEIVED% %BYTES_SENT% %DURATION%
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% "%REQ(X-FORWARDED-FOR)%" "%REQ(USER-AGENT)%"
"%REQ(X-REQUEST-ID)%" "%REQ(:AUTHORITY)%" "%UPSTREAM_HOST%"\n
```

Ejemplo de mensaje del log de acceso:

```
[2019-03-06T09:31:27.360Z] "GET /status/418 HTTP/1.1" 418 - "-" 0 135 5 2 "-" "curl/7.60.0"
"d209e46f-9ed5-9b61-bbdd-43e22662702a"              "httpbin:8000"              "127.0.0.1:80"
inbound|8000|http|httpbin.default.svc.cluster.local  -  172.30.146.73:80  172.30.146.82:38618
outbound_.8000_._.httpbin.default.svc.cluster.local
```

3.2.6. Integraciones en Istio

Antes se ha mencionado una serie de herramientas e integraciones externas que utiliza Istio para complementarse y proporcionar todos los servicios que se han mencionado con anterioridad. En este apartado se va a describir cada una de ellas y para qué sirven.

3.2.6.1. Prometheus

Prometheus es un sistema de monitorización y de alerta open source basado en métricas. Recoge datos de los servicios mediante el envío de solicitudes HTTP en puntos finales y luego guarda los resultados en una base de datos de series de tiempo y los pone a disposición para su posterior análisis.[13]

Para extraer los datos métricos, Prometheus envía una solicitud HTTP llamada “scrape” o raspado. Es decir, envía raspaduras a los objetivos de los que recoge información según la configuración establecida. Cada objetivo se raspa en un intervalo regular de tiempo y lee el punto final HTTP para obtener el estado actual de las métricas del cliente conservando los valores en la base de datos de series de tiempo (TDSB) de Prometheus.

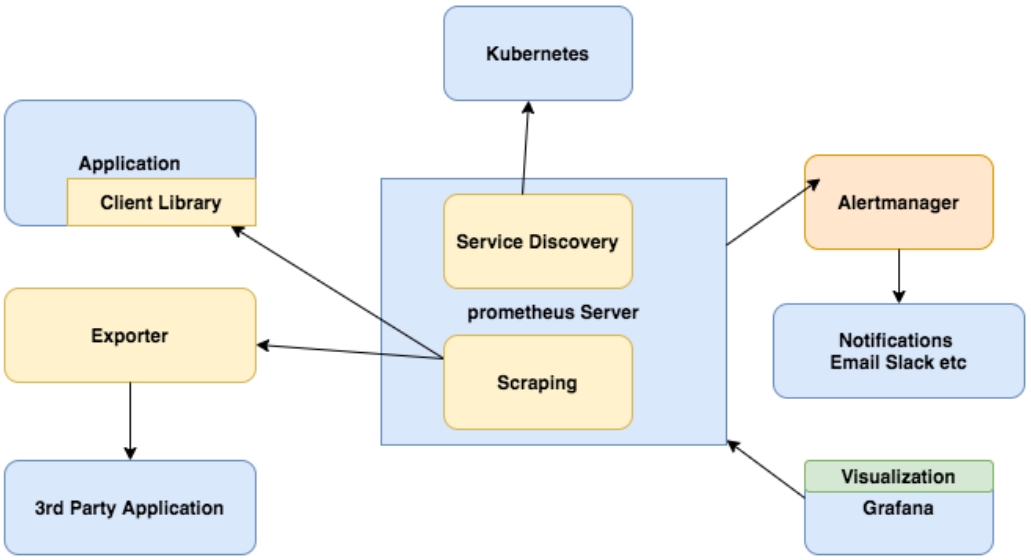


Figura 14. *Componentes de la arquitectura de Prometheus.*[14]

Prometheus está compuesto por los siguientes elementos:

- Servidor de Prometheus: recopila las métricas y las almacena poniéndolas a disposición para futuras consultas. Envía alertas basadas en las métricas recopiladas.

- Bibliotecas de clientes: para monitorear un servicio se necesita agregar la instrumentación al código para comenzar a emitir métricas. En nuestro caso Istio tiene integrada estas bibliotecas por defecto.

- Exportador: algunas aplicaciones exponen métricas en un formato que noes de Prometheus por lo que este componente actúa como proxy entre la aplicación y Prometheus para recibir solicitudes donde recopilar los datos los transformar al formato correcto para finalmente regresarlos.

- Gestión de alertas: las reglas de alerta envían alertas al gestor para enviar notificaciones utilizando varias integraciones listas por defecto como correo electrónico o Slack.

- Visualización con paneles: tiene una serie de APIs que utilizan las consultas de PromQL para producir datos sin procesar para su posterior visualización. La mejor herramienta para visualizar estos datos es Grafana ya que esta es capaz de integrarse completamente con Prometheus.

3.2.6.2. Grafana

Grafana es un software de código abierto para el análisis y supervisión de datos métricos. Con esta herramienta se podrá visualizar, consultar, establecer alertas y obtener información de estos datos. En nuestro caso, nos permite analizar los datos recopilados en Istio y monitorizar el estado de salud de los diferentes servicios que componen la malla.

Esta herramienta permite construir tus propios paneles de control pero Istio viene con un conjunto preconfigurado de paneles para las métricas más importantes y para el plano de control.

3.2.6.3. Jaeger

Jaeger es un sistema open source de rastreo distribuido de extremo a extremo que permite monitorizar y solucionar problemas de transacciones en sistemas con arquitectura de microservicios. Es decir, es la herramienta que nos va a permitir seguir el camino que toma una petición a través de los diferentes microservicios obteniendo una representación visual de la llamada.

Para comprender cómo funciona la arquitectura de Jaeger es necesario introducir el concepto del "*OpenTracing Semantic Specification*"[15], es decir, el convenio de especificación semántica o terminología.

Span, o tramo, representa la unidad lógica de trabajo en Jaeger que tiene un nombre de operación, hora de inicio y duración. Los tramos pueden estar anidados y ordenados para modelar relaciones causales.

Trace, o traza, es una ruta o ejecución de datos a través del sistema y se puede considerar como un gráfico acíclico dirigido de tramos.

Baggage, o equipaje, son metadatos definidos por el propio usuario de manera personalizada que pueden ser adjuntados en el contexto para ser propagados pero que no constituyen en el convenio de del OpenTracing.

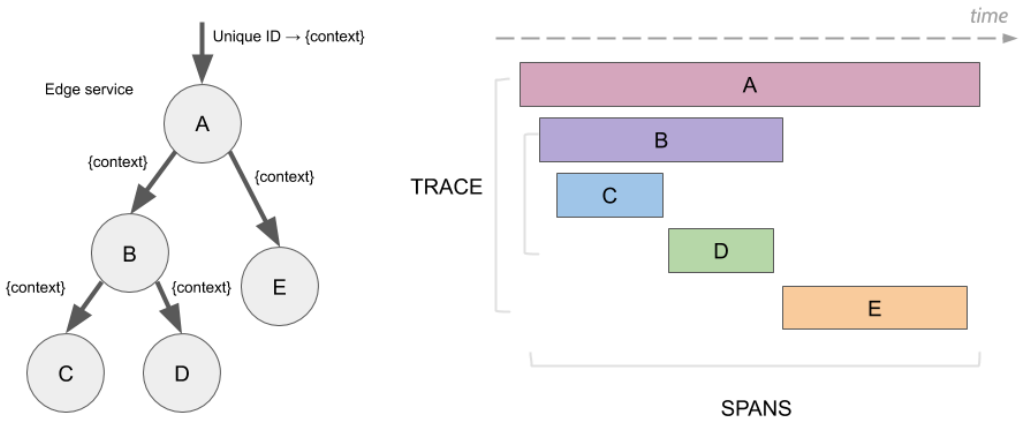


Figura 15. *Definición de trazas y tramos.*[16]

A continuación se detallarán los elementos que componen Jaeger y cómo se relacionan entre ellos ordenados en la que los tramos interactúan con ellos.

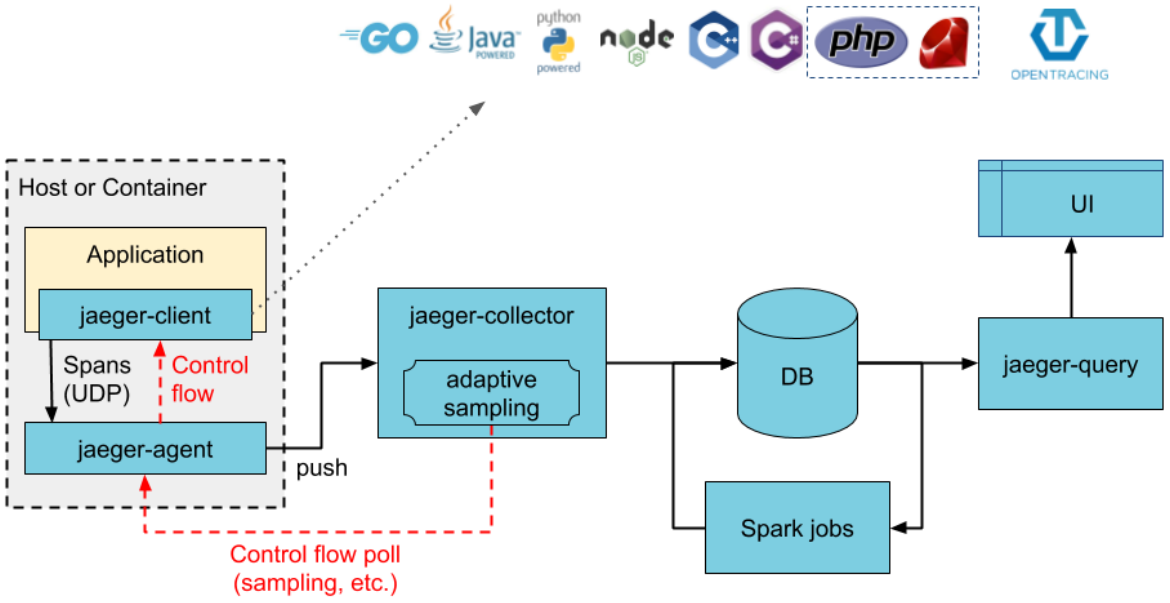


Figura 16. *Componentes de la arquitectura de Jaeger.*[16]

Desde la aplicación, se tiene que implementar la librería Jaeger. Cada cliente tiene su propia implementación según el lenguaje de programación. Istio ya viene preparado para una integración sencilla con Jaeger.

Un servicio genera tramos cada vez que se reciben peticiones y adjunta en ellas la información de contexto (trace id, span id y equipaje). Solo las ids y el equipaje son transmitidos durante las peticiones, todos los demás datos de creación de perfiles, como el nombre de la operación, el tiempo, las etiquetas y los registros no se propagan. En cambio, se transmite fuera del proceso al backend de Jaeger de forma asíncrona.

El *jaeger-agent* es un demonio[2] de red que se mantiene a la escucha de trazas enviadas a través del UDP las cuales las recoge y las envía al jaeger-colector. Está diseñado para implementarse en todos los hosts como un componente de infraestructura. El agente abstrae el enrutamiento y el descubrimiento de los recopiladores del cliente. En Istio se encuentra implementado a través del sidecar.

[2] daemon, o demonio es un tipo especial de proceso informático no interactivo, es decir, que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario.

El *jaeger-collector* recibe las trazas de los agentes y los ejecuta a través de una pipeline[3] que valida los seguimientos, los indexa y realiza cualquier transformación que fuese necesaria antes de almacenarlas.

Jaeger-query recupera las trazas del almacenamiento y los aloja en una interfaz de usuario para mostrarlos.

Figura 17. *Ejemplo de trazas en Jaeger.*[12]

3.2.6.4. Kiali

Kiali es una consola de observabilidad construida para Istio con capacidades de validación y configuración de malla de servicios así como observar las conexiones de los microservicios. También permite comprender la estructura y el estado de la malla de servicios al monitorizar el flujo del tráfico. Proporciona métricas detalladas y una integración básica de Grafana hace que se pueda utilizar para consultas avanzadas. También permite analizar las trazas distribuidas mediante una integración con Jaeger.[17]

A continuación se detallarán los elementos que componen la arquitectura de Kiali.

[3] pipline, o red de procesos de datos conectados en serie.

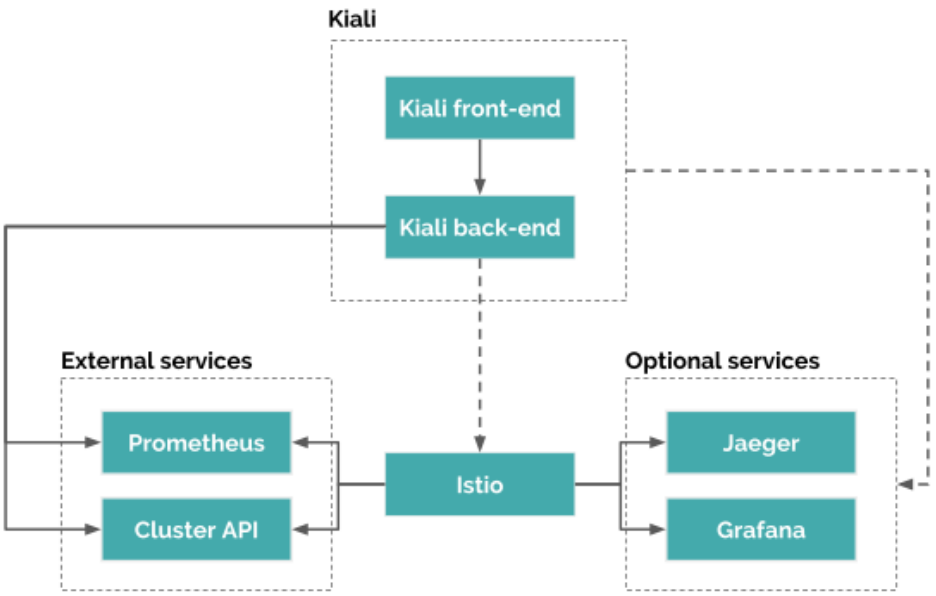


Figura 18. *Arquitectura de los componentes de Kiali.*[18]

Kiali está compuesto de dos componentes, una parte de *back-end* ejecutándose en los contenedores de la aplicación encargada de comunicarse con las distintas partes de Istio. Recoge y procesa los datos para finalmente transportarlos a la parte parte *front-end* donde visualizar los resultados del análisis. Como se puede ver en el diagrama, Kiali depende de servicios externos y otros componentes proporcionados por Istio.

Para obtener los datos de Istio, Kiali se apoya en la integración de Prometheus que hemos mencionado anteriormente de esta manera Kiali es capaz de recolectar los datos relacionados con las métricas, los estados de salud y mostrar posibles problemas si hubiese en nuestra arquitectura de microservicios.

Kiali utiliza la API de la plataforma de aplicaciones de contenedores para buscar y resolver configuraciones de la malla de servicios. En nuestro caso, Kubernetes.

Jaeger y Grafana son herramientas opcionales pero si queremos obtener información sobre la trazabilidad distribuida necesitaremos integrar Jaeger. Grafana nos permitiría personalizar vistas y consultas de los diferentes paneles de control.

3.2.7. Locust

Locust es una herramienta escrita en Python que nos permite hacer pruebas de carga haciendo simulaciones de tráfico real sobre un servicio para observar el comportamiento de este.

La principal característica de esta herramienta es que permite modificar su comportamiento mediante código Python haciendo que cualquier usuario con algún conocimiento en dicho lenguaje de programación pueda flexibilizar o escalar las pruebas al gusto sin tener que utilizar una complicada interfaz gráfica.

El código fuente que he generado para mis pruebas de carga de trabajo sería el siguiente:

```
import time
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
       wait_time = between(7, 19.5)

       @task
       def hellow_world(self):
             self.client.get("/productpage")
```

Este código será el encargado de realizar peticiones a esa dirección (*.../productpage*) y tendrá un tiempo de espera aleatorio entre 7 y 19,5 segundos. Al mismo tiempo y una vez iniciada la herramienta se puede configurar cuántos usuarios virtuales realizan las peticiones, cuánto se incrementan a lo largo del tiempo y a qué dirección realizar las peticiones. Esto se hace desde una sencilla aplicación web con una interfaz simple e intuitiva.

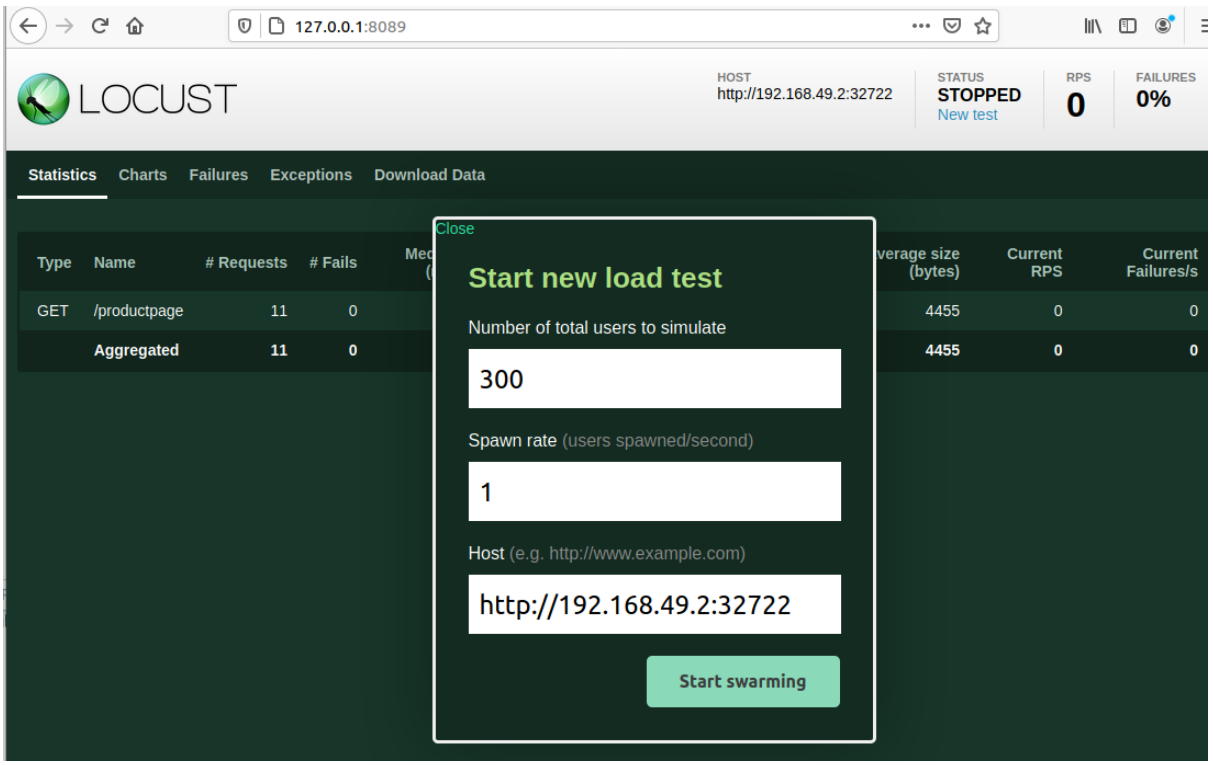


Figura 19. *Ejemplo de test en Locust con 300 usuarios virtuales.*

Con estos parámetros introducidos, esta configuración crearía 300 usuarios a una velocidad de 1 usuario por segundo. Estos usuarios lanzarían peticiones a la dirección: *http://192.168.49.2:32722/productpage* de manera continua con el tiempo de espera aleatorio que se introdujo en el código. La dirección del host viene determinada por el lugar donde se desplegará nuestra aplicación con arquitectura de microservicios que posteriormente explicaré en los siguientes apartados.

4. Implementación de la malla de servicios

Tal y como hemos planteado en los objetivos al inicio del Trabajo de Fin de Grado, la idea es la creación de una prueba de concepto implantando Istio como malla de servicios dada una aplicación con arquitectura de microservicios para posteriormente utilizar las herramientas que proporciona Istio y así observar el comportamiento de nuestra aplicación. Bien a través de gestores del tráfico y bien a través de la aplicación del concepto de trazabilidad distribuida.

4.1. Diseño de la aplicación de microservicios.

La documentación de Istio proporciona una aplicación demo con arquitectura de microservicios que nos permitirá instalar la malla de servicios y jugar con las herramientas que proporciona. Esta aplicación es una aplicación web que consta de cuatro microservicios que componen una página web. El conjunto de todos los microservicios que conforman la aplicación se encuentran alojados en el repositorio de GitHub que Istio utiliza para su propio proyecto[21]. La página en cuestión es la siguiente:

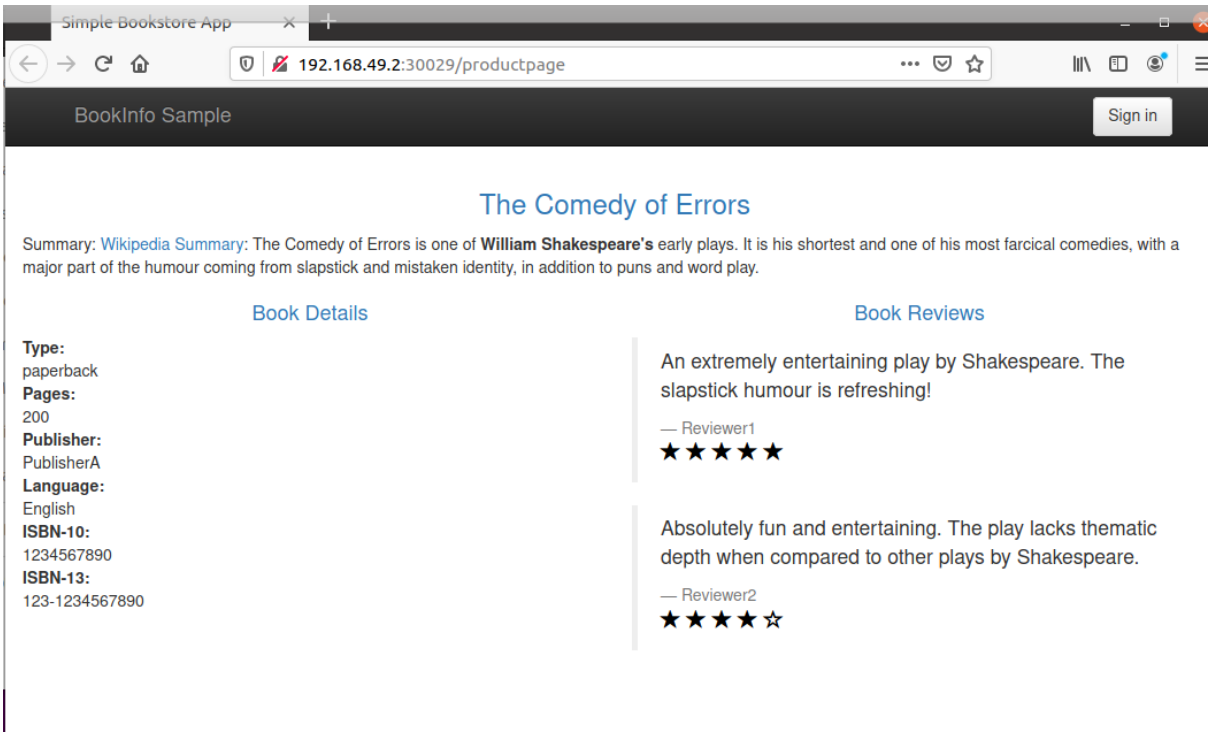


Figura 20. *Aplicación web demo Bookinfo proporcionada por Istio.*

Esta página nos muestra información sobre un libro similar a lo que mostraría un registro de un catálogo de una librería online. Este registro nos indica el título y la descripción del libro, información detallada como el ISBN, número de páginas, etc. y un par de reseñas.

Esta aplicación está fragmentada en cuatro microservicios separados escritos en cuatro lenguajes diferentes:

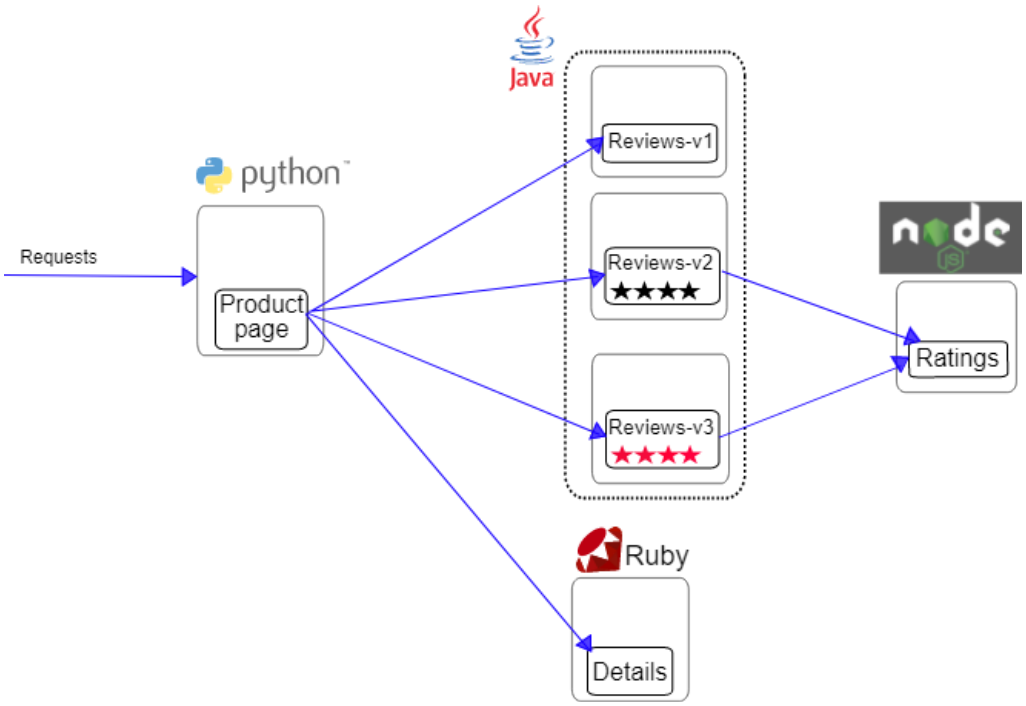


Figura 21. *Esquema de la aplicación Bookinfo.*[19]

A continuación se va a explicar al detalle cada uno de los elementos que componen la aplicación en arquitectura de multiservicios.

4.1.1 Plantilla HTML

A nivel más específico, se puede decir que la página principal está compuesta por una plantilla escrita en HTML que es rellenada por los resultados de cada uno de los microservicios que componen la aplicación. Estos resultados se encapsulan en un objeto JSON que el HTML recibe y posteriormente utiliza para pintar por pantalla la información. La plantilla HTML se encuentra en el repositorio de GitHub que Istio tiene para albergar la

aplicación[22]. Seguidamente vamos a desglosar cada una de las partes de la plantilla que corresponden a cada uno de los microservicios y después se profundizará en estos.

Quitando los fragmentos que se encargan de la cabecera, los estilos y el panel de login, nos quedarían tres elementos importantes en la página HTML. La parte del título y la descripción, la parte de los detalles y la parte de las reseñas.

```
<div class="container-fluid">
  <div class="row">
    <div class="col-md-12">
      <h3 class="text-center text-primary">{{ product.title }}</h3>
      {% autoescape false %}
      <p>Summary: {{ product.descriptionHtml }}</p>
      {% endautoescape %}
    </div>
  </div>
```

Este fragmento de la plantilla corresponde a la parte del título y la descripción. En él podemos ver que se componen de un objeto JSON llamado "*product*" y que hace llamada a dos variables de esta estructura para rellenar el título y la descripción. Estas variables serán entregadas por el microservicio "*Productpage*" que explicaremos más adelante.

```
<div class="row">
    <div class="col-md-6">
      {% if detailsStatus == 200: %}
      <h4 class="text-center text-primary">Book Details</h4>
      <dl>
        <dt>Type:</dt>{{ details.type }}
        <dt>Pages:</dt>{{ details.pages }}
        <dt>Publisher:</dt>{{ details.publisher }}
        <dt>Language:</dt>{{ details.language }}
        <dt>ISBN-10:</dt>{{ details['ISBN-10'] }}
        <dt>ISBN-13:</dt>{{ details['ISBN-13'] }}
      </dl>
      {% else %}
      <h4 class="text-center text-primary">Error fetching product details!</h4>
      {% if details: %}
      <p>{{ details.error }}</p>
      {% endif %}
      {% endif %}
    </div>
```

Esta parte de la plantilla corresponde al cuerpo de los detalles. Al igual que "*product*", "*details*" también es un JSON compuesto de varios atributos que serán rellenados por su correspondiente microservicio. Con estas variables y con la maquetación correspondiente

se nos muestra la descripción del libro de manera estructurada en filas.

```
<div class="col-md-6">
      {% if reviewsStatus == 200: %}
      <h4 class="text-center text-primary">Book Reviews</h4>
      {% for review in reviews.reviews %}
      <blockquote>
        <p>{{ review.text }}</p>
        <small>{{ review.reviewer }}</small>
        {% if review.rating: %}
        {% if review.rating.stars: %}
        <font color="{{ review.rating.color }}">
          <!-- full stars: -->
          {% for n in range(review.rating.stars) %}
          <span class="glyphicon glyphicon-star"></span>
          {% endfor %}
          <!-- empty stars: -->
          {% for n in range(5 - review.rating.stars) %}
          <span class="glyphicon glyphicon-star-empty"></span>
          {% endfor %}
        </font>
        {% elif review.rating.error: %}
        <p><i>{{ review.rating.error }}</i></p>
        {% endif %}
        {% endif %}
      </blockquote>
      {% endfor %}
      {% else %}
      <h4 class="text-center text-primary">Error fetching product reviews!</h4>
      {% if reviews: %}
      <p>{{ reviews.error }}</p>
      {% endif %}
      {% endif %}
    </div>
  </div>
```

Este pedazo de la plantilla corresponde a las reseñas. También es completado por un objeto JSON previamente establecido por su correspondiente microservicio: *Reviews.* Este microservicio tiene la peculiaridad de que puede ser rellenado o no por estrellas en función de la respuesta del microservicio. Más adelante profundizaremos en este tema cuando expliquemos dicho microservicio.

4.1.2 Servicio Productpage

Productpage es el microservicio principal, está escrito en Python y es el encargado de rellenar la plantilla HTML con los objetos JSON que recibe de cada uno de los microservicios que componen la aplicación, incluido el JSON que él mismo compone para rellenar el título y la descripción. Su código fuente se encuentra en el repositorio[23]. Se

apoya en el microframework Flask para el diseño de la aplicación web y así mezclar HTML y lenguaje Python.

A continuación se va a analizar las partes más importantes que componen este microservicio:

```
#[...]imports etc

app = Flask(__name__)
Bootstrap(app)

servicesDomain = "" if (os.environ.get("SERVICES_DOMAIN") is None) else "." +
os.environ.get("SERVICES_DOMAIN")
detailsHostname = "details" if (os.environ.get("DETAILS_HOSTNAME") is None) else
os.environ.get("DETAILS_HOSTNAME")
ratingsHostname = "ratings" if (os.environ.get("RATINGS_HOSTNAME") is None) else
os.environ.get("RATINGS_HOSTNAME")
reviewsHostname = "reviews" if (os.environ.get("REVIEWS_HOSTNAME") is None) else
os.environ.get("REVIEWS_HOSTNAME")

details = {
    "name": "http://{0}{1}:9080".format(detailsHostname, servicesDomain),
    "endpoint": "details",
    "children": []
}

ratings = {
    "name": "http://{0}{1}:9080".format(ratingsHostname, servicesDomain),
    "endpoint": "ratings",
    "children": []
}

reviews = {
    "name": "http://{0}{1}:9080".format(reviewsHostname, servicesDomain),
    "endpoint": "reviews",
    "children": [ratings]
}

productpage = {
    "name": "http://{0}{1}:9080".format(detailsHostname, servicesDomain),
    "endpoint": "details",
    "children": [details, reviews]
}

service_dict = {
    "productpage": productpage,
    "details": details,
    "reviews": reviews,
}
```

Esta es la primera parte, es donde se declaran los objetos que componen los microservicios. También se hace la llamada a Flask.

```
@app.route('/productpage')
@trace()
def front():
    product_id = 0  # TODO: replace default value
    headers = getForwardHeaders(request)
    user = session.get('user', '')
    product = getProduct(product_id)
    detailsStatus, details = getProductDetails(product_id, headers)

    reviewsStatus, reviews = getProductReviews(product_id, headers)
    return render_template(
        'productpage.html',
        detailsStatus=detailsStatus,
        reviewsStatus=reviewsStatus,
        product=product,
        details=details,
        reviews=reviews,
        user=user)
```

Este fragmento se puede considerar el “main” del *Productpage* ya que hace referencia a cuando se llama a la vista */productpage* a través de esa dirección en la aplicación web.

Por un lado se encarga de recoger las cabeceras de las peticiones según el formato B3 como se mencionó en el apartado de trazabilidad distribuida, esto se produce con la llamada *getForwardHeaders* que explicaré más tarde.

Por el otro, se rellenan los objetos JSON que más tarde la plantilla HTML utiliza para mostrar la información. Entre estos objetos JSON se encuentran *product* que con el método *getProduct* obtendrá su título y descripción. *details* se rellena con el método *getProductDetails* que posteriormente veremos que se obtiene a través de la llamada al correspondiente microservicio. Con *reviews* ocurre exactamente lo mismo con su llamada a *getProductReviews.* Una vez rellenados los JSON. Se llama al método que pertenece al framework Flask: “*render_template*” utilizando la plantilla HTML y los JSON completos para componer la página principal.

```
def getForwardHeaders(request):
    headers = {}

    span = get_current_span()
    carrier = {}
    tracer.inject(
        span_context=span.context,
        format=Format.HTTP_HEADERS,
        carrier=carrier)

    headers.update(carrier)

    if 'user' in session:
        headers['end-user'] = session['user']

    incoming_headers = [
        'x-request-id',
        'x-ot-span-context',
        'x-datadog-trace-id',
        'x-datadog-parent-id',
        'x-datadog-sampling-priority',
        'traceparent',
        'tracestate',
        'x-cloud-trace-context',
        'grpc-trace-bin',
        'user-agent',
    ]

    for ihdr in incoming_headers:
        val = request.headers.get(ihdr)
        if val is not None:
            headers[ihdr] = val

    return headers
```

Este es el método que se encarga de componer las cabeceras. Este objeto se va a propagar por cada uno de los servicios de modo que de esta manera las trazas puedan ser asociadas por la herramienta Jaeger gracias a que tienen un identificador único. Esto, como comprobaremos más adelante, se tiene que encontrar en cada uno de los microservicios para no perder la referencia.

```
def getProducts():
    return [
        {
            'id': 0,
            'title': 'The Comedy of Errors',
            'descriptionHtml': '<a
href="https://en.wikipedia.org/wiki/The_Comedy_of_Errors">Wikipedia Summary</a>:
The Comedy of Errors is one of <b>William Shakespeare\'s</b> early plays. It is
his shortest and one of his most farcical comedies, with a major part of the
humour coming from slapstick and mistaken identity, in addition to puns and word
play.'
        }
    ]


def getProduct(product_id):
    products = getProducts()
    if product_id + 1 > len(products):
        return None
    else:
        return products[product_id]
```

Este es el método que compone el producto consultado. En principio sólo tiene definido un producto y termina devolviendo en formato JSON con los atributos siguientes: identificador a 0, el título “The Comedy of Errors” y su descripción.

```
def getProductDetails(product_id, headers):
        try:
            url = details['name'] + "/" + details['endpoint'] + "/" + str(product_id)
            res = requests.get(url, headers=headers, timeout=3.0)
        except BaseException:
            res = None
        if res and res.status_code == 200:
            return 200, res.json()
    status = res.status_code if res is not None and res.status_code else 500
    return status, {'error': 'Sorry, product details are currently unavailable for this
book.'}
```

Este es el método que hace la llamada al microservicio que compone los detalles: *Details*. Como se puede ver, este método genera una url con el objeto declarado *details* para hacer la petición al microservicio. Como se puede comprobar, se hace la petición propagando las cabeceras (*headers*). Después el microservicio actúa y genera una respuesta para posteriormente ser transformada en un JSON y devolverlo a la función principal. Más adelante definiremos que hace el microservicio *Details* por dentro

```
def getProductReviews(product_id, headers):
    for _ in range(2):
        try:
            url = reviews['name'] + "/" + reviews['endpoint'] + "/" + str(product_id)
            res = requests.get(url, headers=headers, timeout=3.0)
        except BaseException:
            res = None
        if res and res.status_code == 200:
            return 200, res.json()
    status = res.status_code if res is not None and res.status_code else 500
    return status, {'error': 'Sorry, product reviews are currently unavailable for this
book.'}
```

Este es el método que hace la llamada al microservicio que compone las reseñas: *Reviews*. Al igual que la llamada a los detalles, también se genera una url con el objeto declarado *review* para hacer la petición al microservicio. También se propagan las cabeceras a través de la petición. Finalmente, el microservicio genera una respuesta para posteriormente ser transformada en un JSON y devolverlo a la función principal.

4.1.3 Servicio Details

Details es el microservicio llamado desde *ProductPage*, está escrito en Ruby y es el encargado de generar un JSON que componga el objeto *details* con los atributos: *author*, *year, type*, *pages*, *publisher*, *language*, *ISBN-10, ISBN-13.* Es decir, el autor, el año de publicación, el tipo de documento, el número de páginas, el publicador, el idioma y los códigos ISBN. Su código fuente se encuentra en el repositorio[24].

A continuación voy a analizar las partes más destacadas del código fuente:

```
server = WEBrick::HTTPServer.new :BindAddress => '*', :Port => port

server.mount_proc '/details' do |req, res|
    pathParts = req.path.split('/')
    headers = get_forward_headers(req)

    begin
        begin
          id = Integer(pathParts[-1])
        rescue
          raise 'please provide numeric product id'
        end
        details = get_book_details(id, headers)
        res.body = details.to_json
        res['Content-Type'] = 'application/json'
    rescue => error
        res.body = {'error' => error}.to_json
        res['Content-Type'] = 'application/json'
        res.status = 400
    end
end

server.start
```

Este fragmento se puede considerar el “main” de *Details* ya que es el que recoge la llamada */details* con los parámetros de la petición. Descompone la url en partes y agarra el identificador del libro consultado para posteriormente obtener los detalles del libro llamando al método *get_book_details* para finalmente obtener una respuesta que devolver al microservicio principal. También añadir que se hace uso del método *get_forward_headers* que es el encargado de seguir propagando las cabeceras y así no perder la traza.

```
def get_book_details(id, headers)
    return {
        'id' => id,
        'author': 'William Shakespeare',
        'year': 1595,
        'type' => 'paperback',
        'pages' => 200,
        'publisher' => 'PublisherA',
        'language' => 'English',
        'ISBN-10' => '1234567890',
        'ISBN-13' => '123-1234567890'
    }
end
```

Este es el método que compone el JSON de los detalles del libro.

```
def get_forward_headers(request)
  headers = {}

  incoming_headers = [
      'x-request-id',
      'x-ot-span-context',
      'x-datadog-trace-id',
      'x-datadog-parent-id',
      'x-datadog-sampling-priority',
      'traceparent',
      'tracestate',
      'x-cloud-trace-context',
      'grpc-trace-bin',
      'x-b3-traceid',
      'x-b3-spanid',
      'x-b3-parentspanid',
      'x-b3-sampled',
      'x-b3-flags',
      'end-user',
      'user-agent',
  ]

  request.each do |header, value|
    if incoming_headers.include? header then
      headers[header] = value
    end
  end

  return headers
end
```

Este es el método que compone las cabeceras para ser propagadas a lo largo de la petición.

4.1.4 Servicio Reviews

Reviews es otro microservicio llamado desde *ProductPage*, está escrito en Java y es el encargado de generar un JSON que componga el objeto *reviews* con un array de las reseñas que contenga el libro consultado. La peculiaridad de este microservicio es que se despliega con tres versiones diferentes las cuales por defecto se reparten el flujo del tráfico de manera equilibrada. Esta repartición se podrá cambiar mediante las herramientas que proporciona Istio y que se estudiará más adelante.

De estas tres versiones desplegadas, dos de sus versiones hacen una llamada al microservicio *Ratings* que se encarga de obtener las puntuaciones del 1 al 5. El código fuente de este microservicio también se encuentra en el repositorio.[25]

A continuación explicaré la diferencia visual de las tres versiones, cómo se hace posible en el código y analizaré otras partes del código como la petición a *Ratings*:.

- *reviews-v1*: esta versión del microservicio no hace la llamada al servicio *ratings* y sólo nos muestra la reseña y el autor de la reseña.

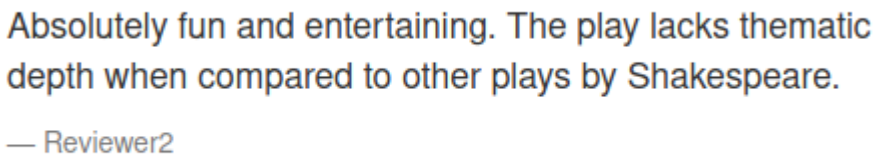


Figura 22. *Servicio reviews-v1.*

reviews-v2: esta versión hace una llamada a *ratings* y muestra la puntuación de la reseña con estrellas de color negro además de la información anterior descrita.

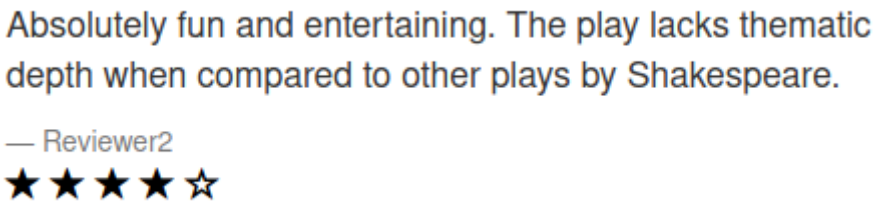


Figura 23. *Servicio reviews-v2.*

- *reviews-v3:* esta versión también hace una llamada a *ratings* pero esta vez muestra la puntuación de color rojo.

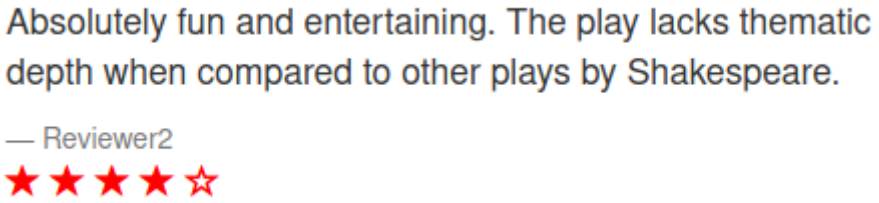


Figura 24. *Servicio reviews-v3.*

A la hora de desplegar la aplicación Java en contenedores mediante Docker, este necesita un fichero llamado Dockerfile. Este es el fichero que Istio tiene configurado para tal cometido:

```
FROM websphere-liberty:20.0.0.6-full-java8-ibmjava

ENV SERVERDIRNAME reviews

COPY ./servers/LibertyProjectServer /opt/ibm/wlp/usr/servers/defaultServer/

RUN /opt/ibm/wlp/bin/installUtility install  --acceptLicense
/opt/ibm/wlp/usr/servers/defaultServer/server.xml && \
    chmod -R g=rwx /opt/ibm/wlp/output/defaultServer/

ARG service_version
ARG enable_ratings
ARG star_color
ENV SERVICE_VERSION ${service_version}
ENV ENABLE_RATINGS ${enable_ratingse}
ENV STAR_COLOR ${star_color}

CMD ["/opt/ibm/wlp/bin/server", "run", "defaultServer"]
```

Como se puede observar, se reciben tres argumentos: versión del servicio, permitir puntuación y color de las estrellas. Estos argumentos se almacenan en una variable de entorno que posteriormente se utilizará en el código fuente del microservicio. Esto permite que cuando se vaya a desplegar las tres versiones se haga con las siguientes configuraciones:

- v1, false, *null*
- v2, true, black
- v3, true, red

El código fuente de la parte principal del microservicio sería el siguiente:

```
public class LibertyRestEndpoint extends Application {

    private final static Boolean ratings_enabled = Boolean.valueOf(System.getenv("ENABLE_RATINGS"));
    private final static String star_color = System.getenv("STAR_COLOR");
    private final static String services_domain = System.getenv("SERVICES_DOMAIN");
    private final static String ratings_hostname = System.getenv("RATINGS_HOSTNAME");
    private final static String ratings_service = "http://" + ratings_hostname + services_domain +
":9080/ratings";


    @GET
    @Path("/reviews/{productId}")
    public Response bookReviewsById(@PathParam("productId") int productId, @Context HttpHeaders
requestHeaders) {
      int starsReviewer1 = -1;
      int starsReviewer2 = -1;

      if (ratings_enabled) {
        JsonObject ratingsResponse = getRatings(Integer.toString(productId), requestHeaders);
        if (ratingsResponse != null) {
          if (ratingsResponse.containsKey("ratings")) {
            JsonObject ratings = ratingsResponse.getJsonObject("ratings");
            if (ratings.containsKey("Reviewer1")){
                    starsReviewer1 = ratings.getInt("Reviewer1");
            }
            if (ratings.containsKey("Reviewer2")){
              starsReviewer2 = ratings.getInt("Reviewer2");
            }
          }
        }
      }

      String jsonResStr = getJsonResponse(Integer.toString(productId), starsReviewer1, starsReviewer2);
      return Response.ok().type(MediaType.APPLICATION_JSON).entity(jsonResStr).build();
    }
}
```

Como mencionamos antes, a la hora de desplegar el microservicio se establecen unas variables de entorno en función de la versión y luego se recogen en el programa principal como variables estáticas globales. Esto va a determinar el comportamiento de nuestro microservicio.

Al tener la variable ratings_enabled a true, se hará la llamada al método *getRatings* que posteriormente analizaremos pero podemos adelantar que nos devolverá la cantidad de estrellas que el autor de la reseña ha dado al libro.

Si no está activada, salta directamente al método *getJsonResponse* que será el encargado de componer el JSON de las reseñas (con puntuación o sin ella).

```
private String getJsonResponse (String productId, int starsReviewer1, int starsReviewer2) {
         String result = "{";
         result += "\"id\": \"" + productId + "\",";
         result += "\"reviews\": [";

         // reviewer 1:
         result += "{";
         result += "  \"reviewer\": \"Reviewer1\",";
         result += "  \"text\": \"An extremely entertaining play by Shakespeare. The slapstick humour is
refreshing!\"";
      if (ratings_enabled) {
        if (starsReviewer1 != -1) {
          result += ", \"rating\": {\"stars\": " + starsReviewer1 + ", \"color\": \"" + star_color + "\"}";
        }
        else {
          result += ", \"rating\": {\"error\": \"Ratings service is currently unavailable\"}";
        }
      }
         result += "},";

         // reviewer 2:
         result += "{";
         result += "  \"reviewer\": \"Reviewer2\",";
         result += "  \"text\": \"Absolutely fun and entertaining. The play lacks thematic depth when
compared to other plays by Shakespeare.\"";
      if (ratings_enabled) {
        if (starsReviewer2 != -1) {
          result += ", \"rating\": {\"stars\": " + starsReviewer2 + ", \"color\": \"" + star_color + "\"}";
        }
        else {
          result += ", \"rating\": {\"error\": \"Ratings service is currently unavailable\"}";
        }
      }
         result += "}";
         result += "]";
         result += "}";
         return result;
    }
```

Como se puede ver, este método se encarga de componer el JSON a base de concatenar las cadenas de texto, las llaves ({,}) y los corchetes ([,]). Genera un objeto cuyo id es el identificador del libro, un array de una estructura llamada *reviews* (reseñas), compuesta por un *reviewer* (autor de la reseña) y *text* (descripción de su comentario).

Si las puntuaciones están habilitadas, se compone otra estructura más llamada *rating* que a su vez contiene los atributos: *stars* (cantidad de estrellas, ya recogido de la llamada del servicio y pasado por parámetro a esta función) y *color,* que se apoyará sobre la variable global que hemos recogido al despliegue del contenedor.

Finalmente, tras componer el JSON se devuelve al programa principal para rellenar la plantilla HTML con los datos correspondientes.

```
  private JsonObject getRatings(String productId, HttpHeaders requestHeaders) {
      ClientBuilder cb = ClientBuilder.newBuilder();
      Client client = cb.build();
      WebTarget ratingsTarget = client.target(ratings_service + "/" + productId);
      Invocation.Builder builder = ratingsTarget.request(MediaType.APPLICATION_JSON);
      for (String header : headers_to_propagate) {
        String value = requestHeaders.getHeaderString(header);
        if (value != null) {
          builder.header(header,value);
        }
      }
      try {
        Response r = builder.get();

        int statusCode = r.getStatusInfo().getStatusCode();
        if (statusCode == Response.Status.OK.getStatusCode()) {
          try (StringReader stringReader = new StringReader(r.readEntity(String.class));
               JsonReader jsonReader = Json.createReader(stringReader)) {
            return jsonReader.readObject();
          }
        } else {
          System.out.println("Error: unable to contact " + ratings_service + " got status of " +
statusCode);
          return null;
        }
      } catch (ProcessingException e) {
        System.err.println("Error: unable to contact " + ratings_service + " got exception " + e);
        return null;
      }
    }

private final static String[] headers_to_propagate = {
        "x-request-id",        "x-ot-span-context",         "x-datadog-trace-id",
        "x-datadog-parent-id",         "x-datadog-sampling-priority",
        "traceparent",        "tracestate",        "x-cloud-trace-context",
        "grpc-trace-bin",        "x-b3-traceid",        "x-b3-spanid",
        "x-b3-parentspanid",        "x-b3-sampled",        "x-b3-flags",
        "end-user",        "user-agent",    };
```

Este es el método que hace la llamada al microservicio *Ratings.* Al igual que las otras llamadas a los servicios, también se hace una llamada al método *getHeader* para establecer las cabeceras que hemos ido propagando a lo largo de nuestras peticiones y mandarla con nuestra petición a *Ratings* para así no perder la traza y poder analizarla con Jaeger.

Tras recibir respuesta del servicio, recoger el objeto y componer el JSON que haya recibido para mandarlo como respuesta a la función principal del microservicio.

4.1.5 Servicio Ratings

Ratings es el último microservicio, como hemos explicado anteriormente, es llamado desde *Reviews*, pero sólamente desde la versión 2 y 3. Está escrito en Node JS y es el encargado de obtener el número de estrellas que ha puntuado el autor de la reseña. Su código fuente se encuentra en el repositorio[26].

A continuación analizaré la parte más destacada del código:

```
var http = require('http')
var dispatcher = require('httpdispatcher')
var server = http.createServer(handleRequest)


function handleRequest (request, response) {
  try {
    console.log(request.method + ' ' + request.url)
    dispatcher.dispatch(request, response)
  } catch (err) {
    console.log(err)
  }
}

server.listen(port, function () {
  console.log('Server listening on: http://0.0.0.0:%s', port)
})

dispatcher.onPost(/^\/ratings\/[0-9]*/, function (req, res) {
  var productIdStr = req.url.split('/').pop()
  var productId = parseInt(productIdStr)
  var ratings = {}
  try {
    ratings = JSON.parse(req.body)
  } catch (error) {
    res.writeHead(400, {'Content-type': 'application/json'})
    res.end(JSON.stringify({error: 'please provide valid ratings JSON'}))
    return
  }
})

dispatcher.onGet(/^\/ratings\/[0-9]*/, function (req, res) {
  var productIdStr = req.url.split('/').pop()
  var productId = parseInt(productIdStr)
  var firstRating = 5
  var secondRating = 4
  var result = {
                id: productId,
                ratings: {
                  Reviewer1: firstRating,
                  Reviewer2: secondRating
                }
              }
  res.writeHead(200, {'Content-type': 'application/json'})
  res.end(JSON.stringify(result))
})
```

Estos métodos del código señalan cómo recoge la petición desde el microservicio *Reviews* para luego componer el JSON de las puntuaciones de las reseñas con 5 y 4 estrellas respectivamente y enviarlo al microservicio que lo llamó.

En el código fuente original del repositorio hay más métodos que se han omitido ya que estos métodos sirven para los diferentes casos de prueba cuando se ejecutan las herramientas de observabilidad y gestión de tolerancia a fallos que incluye Istio.

4.2. Instalación

A continuación se van a detallar las instalaciones necesarias y los pasos que se han ido tomando para la completa implementación de una malla de servicios. Empezando por la máquina virtual y la aplicación web, continuando por la malla de servicios y finalizando con las herramientas de análisis que Istio proporciona.

4.2.1 Máquina virtual y sistema operativo

Para la instalación de la aplicación y el posterior despliegue de Istio se ha creado una máquina virtual con capacidad suficiente para albergar los servicios necesarios (Docker, Kubernetes, Istio…).

Para crear la máquina virtual nos hemos apoyado en VirtualBox. La máquina se creó con las siguientes características:

- 20GB de memoria RAM
- 4 núcleos de procesadores
- 50 GB de espacio físico en el disco espacio físico.
- Acceso a la red

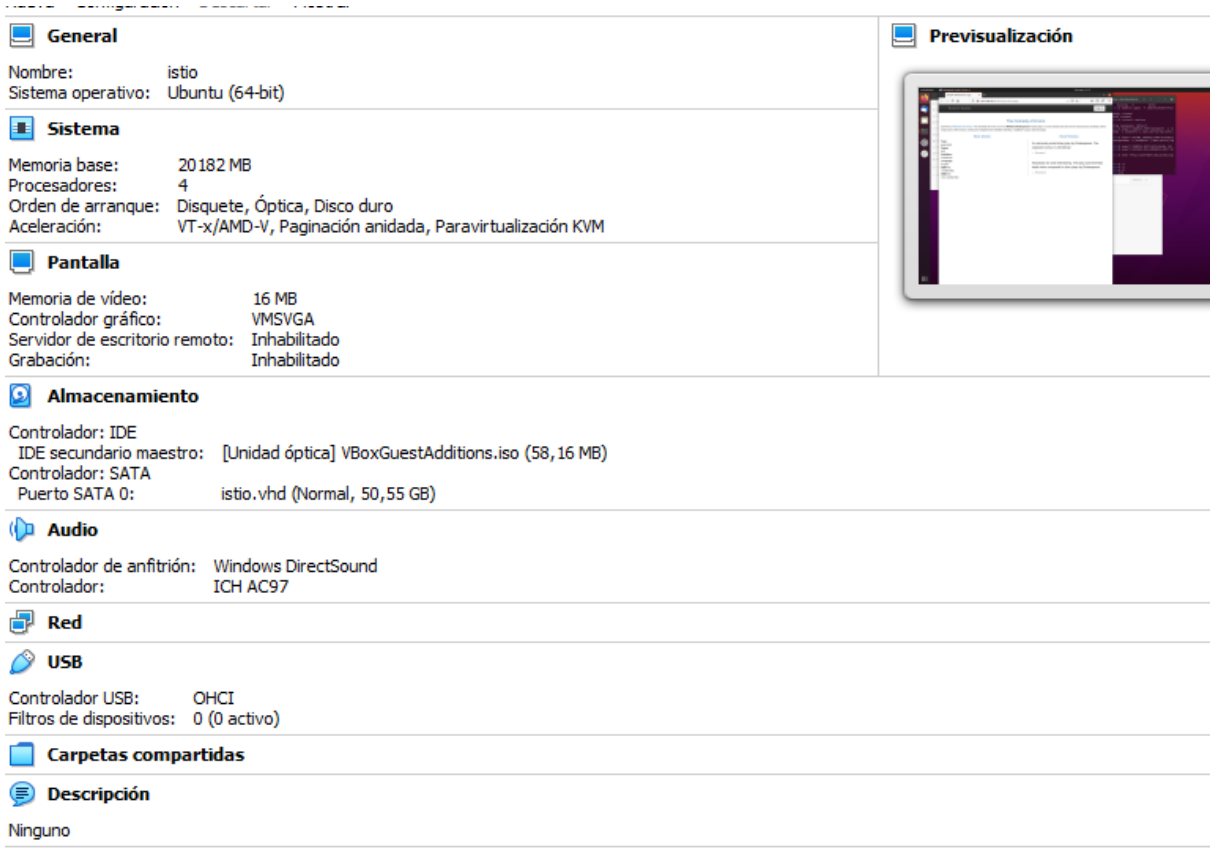


Figura 25. *Características máquina virtual en VirtualBox.*

El sistema operativo instalado ha sido Ubuntu 64 bits ya que en los manuales de instalación del resto de servicios se presentan más facilidades si se utiliza una distribución de Linux en vez de Windows.

4.2.2. Docker

Como mencionamos en el apartado anterior, Docker será la tecnología seleccionada para la creación de contenedores. Para la instalación de Docker ejecutamos la línea de comando:

```
$ sudo apt install docker.io
```

Tras unos minutos, el gestor de paquetes descarga Docker y todas las dependencias necesarias para su completa instalación. Para iniciar Docker y activarlo cada vez que se arranque la máquina virtual introducimos el siguiente comando:

```
$ sudo systemctl start docker
$ sudo systemctl enable --now docker
```

Si quisiéramos desactivar el inicio tras el arranque sólo tendríamos que introducir la misma línea de comando cambiando el enable por disable. Después establecemos los privilegios de usuario para no tener problemas de ejecución con el usuario actual.

```
$ sudo usermod -aG docker carlos && newgrp docker
```

Podemos comprobar la versión instalada de Docker si ejecutamos el comando:

```
$ docker --version
```

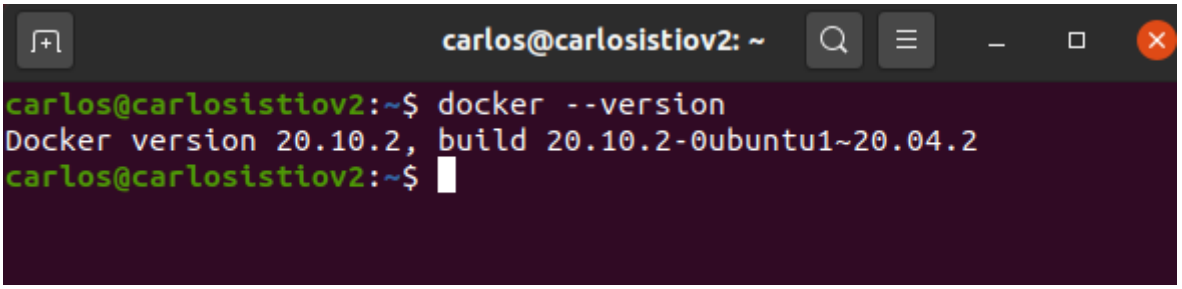


Figura 26. *Resultado de comprobación versión de Docker.*

4.2.3. Kubernetes y minikube

Una vez tenemos instalado Docker continuamos con la instalación del orquestador de contenedores para nuestra aplicación de microservicios. La tecnología seleccionada fue Kubernetes y minikube para poder desplegarlo en una máquina local sin necesidad de contratar un servicio en la nube.

Para comenzar la instalación se ha tenido que seguir el tutorial que hay en la documentación que la página web de *minikube* proporciona, para ello se ejecuta la línea de comandos:

```
$ curl https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Podemos comprobar si se ha instalado correctamente si lanzamos el comando:

```
$ minikube version
```

```
carlos@carlosistiov2: ~
carlos@carlosistiov2:~$ minikube version
minikube version: v1.19.0
commit: 15cede53bdc5fe242228853e737333b09d4336b5
carlos@carlosistiov2:~$
```

Figura 27. *Resultado comprobación versión minikube instalada.*

Para arrancar *minikube* primero deberíamos de establecer Docker como driver principal. Para ello ejecutamos las siguientes línea de comandos:

```
$ minikube config set driver docker
$ minikube start --driver=docker
```

```
carlos@carlosistiov2: ~
carlos@carlosistiov2:~$ minikube start --driver=docker
😄  minikube v1.19.0 en Ubuntu 20.04 (vbox/amd64)
✨  Using the docker driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
💾  Descargando Kubernetes v1.20.2 ...
    > preloaded-images-k8s-v10-v1...: 491.71 MiB / 491.71 MiB  100.00% 19.74 Mi
    > gcr.io/k8s-minikube/kicbase...: 357.67 MiB / 357.67 MiB  100.00% 5.94 MiB
🔥  Creando docker container (CPUs=2, Memory=5000MB) ...
🐳  Preparando Kubernetes v1.20.2 en Docker 20.10.5...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Complementos habilitados: storage-provisioner, default-storageclass
💡  kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" name
space by default
carlos@carlosistiov2:~$
```

Figura 28. *Resultado de instalación de kubernetes y minikube en local.*

Finalmente y para comprobar que todo ha ido correctamente, podemos introducir el comando que nos permitiría acceder al panel de control de nuestros servicios en Kubernetes:

```
$ minikube dashboard
```

Este comando nos levantaría un servicio en local donde podemos comprobar mediante el navegador de una página web el estado de nuestros servicios:

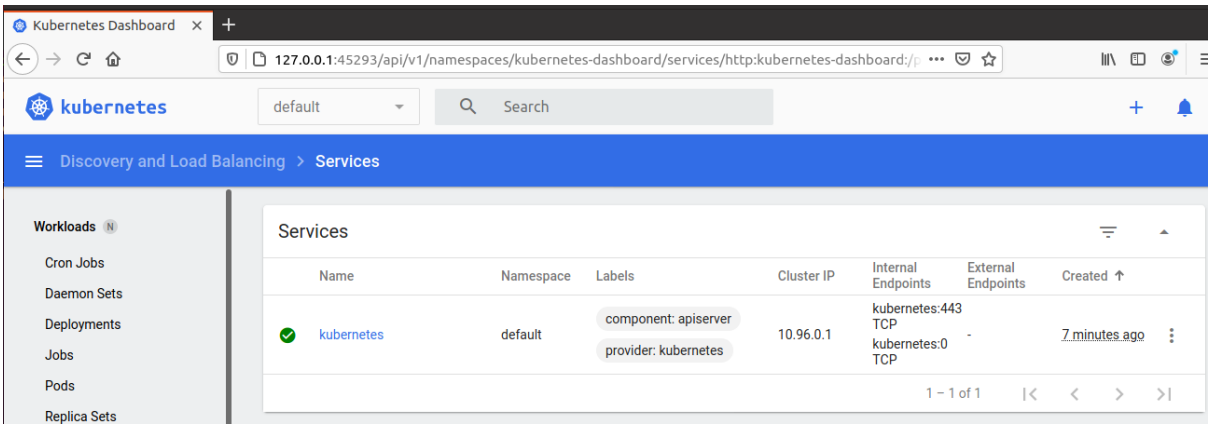


Figura 29. *Panel de control de Kubernetes desplegado en local.*

4.2.4. Istio

A continuación, se procede con la instalación de Istio siguiendo las indicaciones que la documentación oficial de Istio proporciona en su página web. Esta instalación incluye la aplicación de demostración de la que se ha hablado con anterioridad. Para obtener Istio se ejecutaría la siguiente línea de comando:

```
$ curl -L https://istio.io/downloadIstio |sh -
```

Esto nos descarga todos los ficheros en la ruta */home/carlos/istio-1.9.4*. Por comodidad, se ha preferido trasladar este fichero al escritorio y renombrarlo como *istio* para poder acceder a él con mayor facilidad. Incluimos en nuestras variables de entorno la nueva ruta para poder reconocer el comando "*istioctl*"

```
$ export PATH=$PATH:/home/carlos/Escritorio/istio/bin
```

De esta manera ya podemos ejecutar el siguiente comando para continuar con la instalación. Es importante establecer los parámetros indicados ya que nos permitirá instalar de manera completa la configuración predeterminada para la demostración:

```
$ istioctl install --set profile=demo -y
```

```
carlos@carlosistiov2:~$ istioctl install --set profile=demo -y
✔ Istio core installed
✔ Istiod installed
✔ Egress gateways installed
✔ Ingress gateways installed
✔ Installation complete
carlos@carlosistiov2:~$
```

Figura 30. *Resultado instalación Istio.*

Luego, se establece la etiqueta del *namespace* para ordenar a Istio que automáticamente inyecte los proxies sidecar de Envoy una vez desplegada nuestra aplicación en microservicios. Esto se realiza con el siguiente comando:

```
$ kubectl -label namespace default istio-injection=enabled
```

```
carlos@carlosistiov2:~$ kubectl label namespace default istio-injection=enabled
namespace/default labeled
carlos@carlosistiov2:~$
```

Figura 31. *Resultado asignación de etiqueta namespace.*

4.2.5. Despliegue de la aplicación Bookinfo

A continuación, en este apartado contaré los pasos que hay que seguir para desplegar nuestra aplicación y abrir su tráfico al público. Para más tarde ejecutar nuestras herramientas de análisis. Para ello desde nuestro directorio istio, introducimos el comando:

```
$ kubectl -apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```

```
carlos@carlosistiov2: ~/Escritorio/istio
carlos@carlosistiov2:~/Escritorio/istio$ kubectl apply -f samples/bookinfo/platf
orm/kube/bookinfo.yaml
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

Figura 32. *Resultado despliegue aplicación Bookinfo.*

La aplicación por tanto comenzará a desplegarse. Durante el despliegue y a medida que los pods están listos, el sidecar de Istio se irá desplegando junto a ellos. Podemos ir comprobando el estado de los servicios y los pods para ver cuándo están listos mediante los comandos:

```
$ kubectl get services
```

```
carlos@carlosistiov2: ~
carlos@carlosistiov2:~$ kubectl get services
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
details       ClusterIP   10.103.158.181   <none>        9080/TCP   72s
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP    3h8m
productpage   ClusterIP   10.108.222.94    <none>        9080/TCP   72s
ratings       ClusterIP   10.97.111.54     <none>        9080/TCP   72s
reviews       ClusterIP   10.106.21.236    <none>        9080/TCP   72s
```

Figura 33. *Estado de los servicios*

```
$ kubectl get pods
```

```
carlos@carlosistiov2:~$ kubectl get pods
NAME                                READY   STATUS            RESTARTS   AGE
details-v1-79f774bdb9-qx2d7         0/2     PodInitializing   0          81s
productpage-v1-6b746f74dc-wglqp     0/2     PodInitializing   0          82s
ratings-v1-b6994bb9-jx6b5           0/2     PodInitializing   0          81s
reviews-v1-545db77b95-x42v9         0/2     PodInitializing   0          81s
reviews-v2-7bf8c9648f-j5jf2         2/2     Running           0          82s
reviews-v3-84779c7bbc-hjw54         2/2     Running           0          82s
carlos@carlosistiov2:~$
```

Figura 34. *Estado de los pods, parte 1. Iniciando*

```
carlos@carlosistiov2:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
details-v1-79f774bdb9-qx2d7         2/2     Running   0          5m56s
productpage-v1-6b746f74dc-wglqp     2/2     Running   0          5m57s
ratings-v1-b6994bb9-jx6b5           2/2     Running   0          5m56s
reviews-v1-545db77b95-x42v9         2/2     Running   0          5m56s
reviews-v2-7bf8c9648f-j5jf2         2/2     Running   0          5m57s
reviews-v3-84779c7bbc-hjw54         2/2     Running   0          5m57s
```

Figura 35. *Estado de los servicios, parte 2. Listos y ejecutándose*

Tras unos minutos, todos los servicios deberían de estar con la indicación de READY en 2/2 y con el STATUS running antes de continuar. La manera de comprobarlo es ejecutando el anterior comando tantas veces como sea necesario hasta que lo veamos.

Una vez desplegada la aplicación sólo queda abrirla al tráfico. Para hacerla accesible hay que crear un punto de acceso llamado *Istio Ingress Gateway* el cual permite gestionar la entrada y salida del tráfico dentro de la malla de servicios. Para asociar esta puerta de acceso con nuestra aplicación hay que ejecutar el siguiente comando:

```
$ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

```
carlos@carlosistiov2:~/Escritorio/istio$ kubectl apply -f samples/bookinfo/networking
/bookinfo-gateway.yaml
gateway.networking.istio.io/bookinfo-gateway created
virtualservice.networking.istio.io/bookinfo created
carlos@carlosistiov2:~/Escritorio/istio$
```

Figura 36. *Ingress gateway asociada al despliegue.*

Para comprobar que todo ha ido bien hasta ahora se puede ejecutar el siguiente comando:

```
$ istioctl analyze
```

```
carlos@carlosistiov2:~/Escritorio/istio$ istioctl analyze

✔ No validation issues found when analyzing namespace: default.
```

Figura 37. *Comprobación de que todo va bien.*

Seguidamente habría que determinar la IP y los puertos de ingreso para acceder a través del punto de acceso. Para ello vamos a recoger estos datos de los servicios y los vamos a concatenar en una variable de entorno que nos permita establecer la url de acceso. Utilizaremos los siguiente comandos:

```
$ export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
```

```
$ export INGRESS_HOST=$(minikube ip)

$ export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT

$ echo "http://$GATEWAY_URL/productpage"
```

El resultado final es una dirección web donde se encuentra nuestra aplicación para acceder a ella desde el navegador.

```
carlos@carlosistiov2: ~/Escritorio/istio
carlos@carlosistiov2:~/Escritorio/istio$ export INGRESS_PORT=$(kubectl -n istio-syste
m get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].node
Port}')
carlos@carlosistiov2:~/Escritorio/istio$ echo "$INGRESS_PORT"
32722
carlos@carlosistiov2:~/Escritorio/istio$ export INGRESS_HOST=$(minikube ip)
carlos@carlosistiov2:~/Escritorio/istio$  export GATEWAY_URL=$INGRESS_HOST:$INGRESS_P
ORT
carlos@carlosistiov2:~/Escritorio/istio$ echo "http://$GATEWAY_URL/productpage"
http://192.168.49.2:32722/productpage
carlos@carlosistiov2:~/Escritorio/istio$
```

Figura 38. *Resultado establecimiento de puerto e ip del servicio.*

En nuestro caso, la IP y puerto resultante son: 192.168.49.2:32722

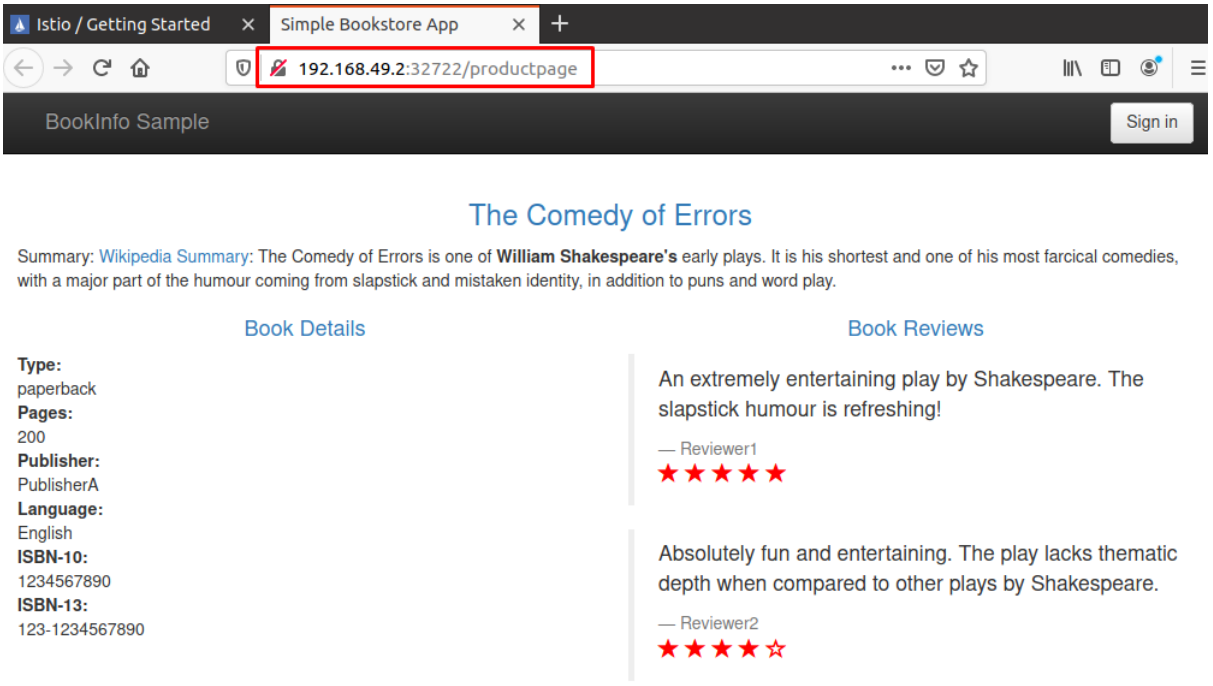


Figura 39. *Aplicación desplegada en la dirección mostrada por consola.*

Si consultamos nuestro panel de control de minikube también podemos ver los servicios desplegados y su estado actual:

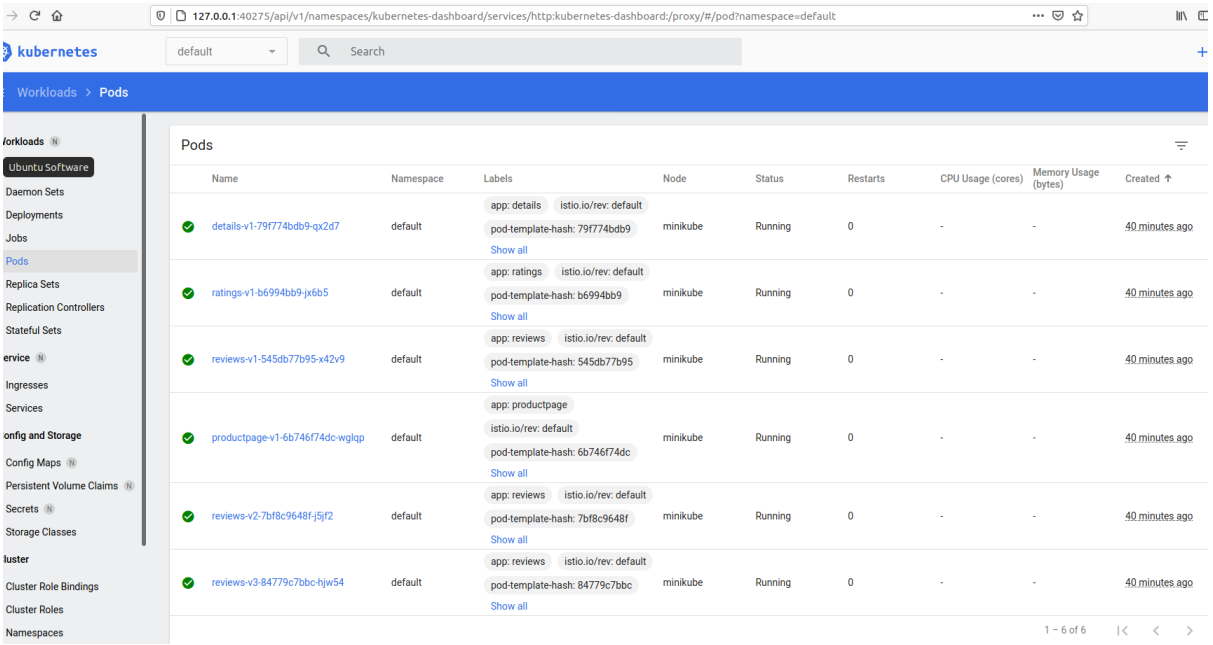


Figura 40. *Panel de control Kubernetes con la aplicación desplegada.*

Sin embargo, donde más información vamos a obtener va a ser utilizando las herramientas que integra Istio para la observabilidad y recogida de métricas.

4.2.6. Kiali

Comenzaremos por mostrar el despliegue en el panel de control de Kiali, de esta manera podemos obtener más información de la estructura de nuestra malla de servicios y analizar su estado de salud entre otras muchas funcionalidades. Para empezar, lo que tenemos que hacer es desplegar nuestra herramienta con el comando

```
$ kubectl apply -f samples/addons

$ kubectl rollout status deployment/kiali -n istio-system
```

```
carlos@carlosistiov2:~/Escritorio/istio$ kubectl rollout status deployment/kiali -n i
stio-system
deployment "kiali" successfully rolled out
```

Figura 41. *Resultado de rollout de Kialli.*

Y para acceder al panel de control de Kiali sería:

```
$ istioctl dashboard kiali
```

Esto levanta la aplicación en local con el puerto 20001 que nos permite consultar y acceder al panel de control de Kiali. En él podemos navegar hasta el menú de *Graph* (grafos) en el cual podemos consultar las relaciones existentes entre los microservicios. Para ello debemos seleccionar el *Namespace: default* y *Versioned app graph.*

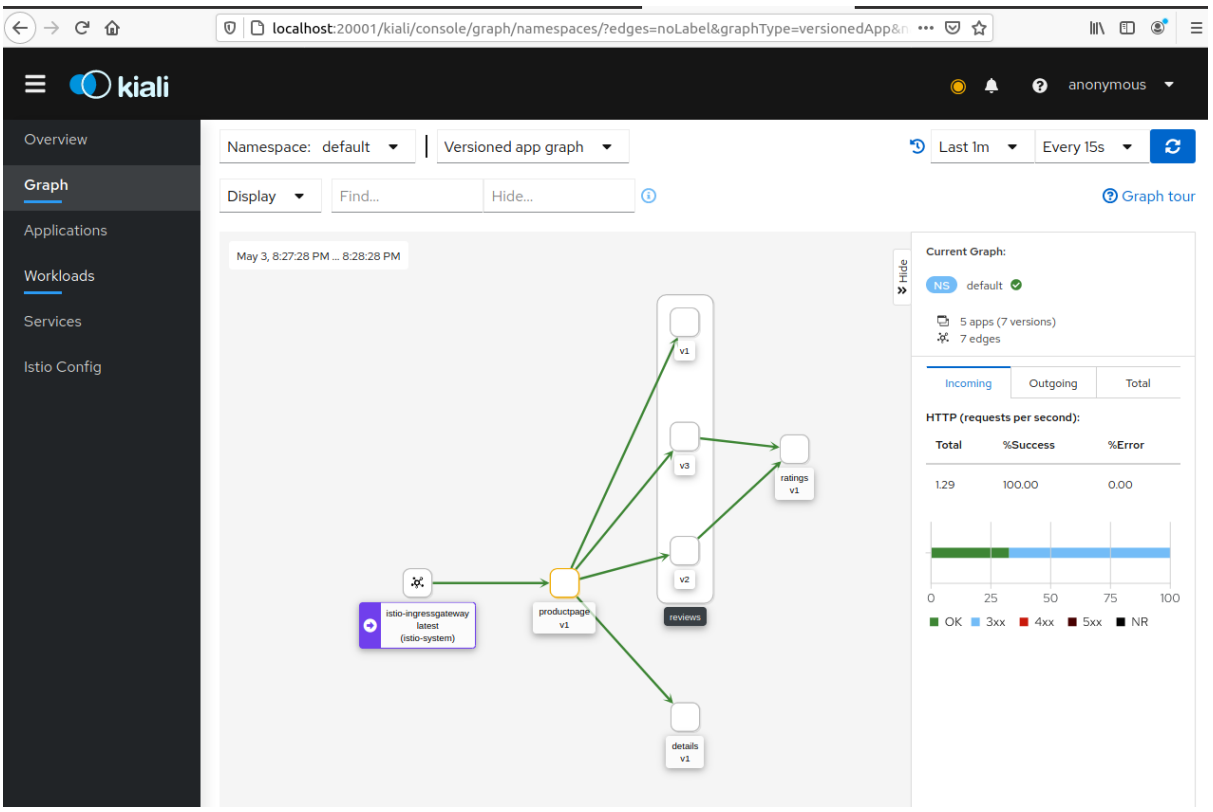


Figura 42. *Grafo de servicios interconectados en Kiali.*

Desde este panel de control tenemos otras funcionalidades, como por ejemplo gestionar el tráfico por donde queremos que pasen las futuras peticiones a la aplicación. Si vamos a *Services* y seleccionamos *reviews.* Nos posicionamos en el servicio *reviews* que contiene tres tipos de versiones del servicio. Por defecto, la distribución de las peticiones están a ⅓ de carga de trabajo. Sin embargo, si damos la opción "*Actions*" y seleccionamos "*Update Traffic Shifting"*, podemos establecer un % diferente de pesos para forzar a las peticiones ir a una sola versión.

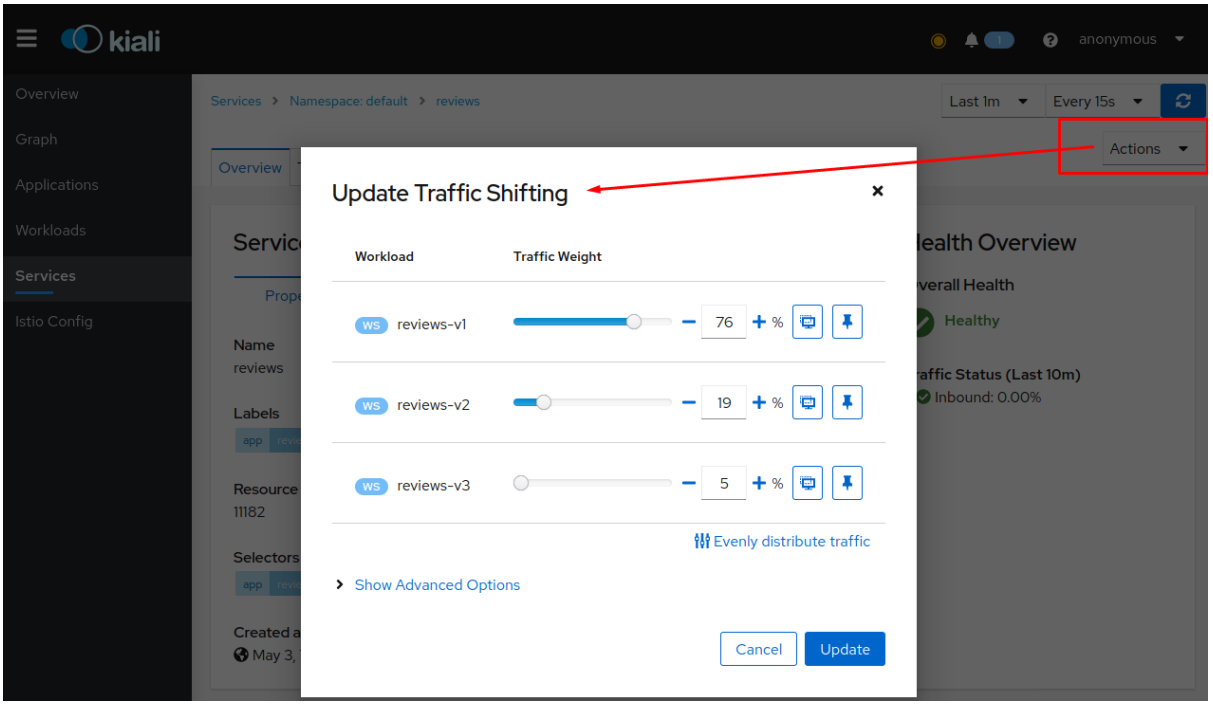


Figura 43. *Gestión del tráfico.*

Para continuar con la prueba nos hemos tenido que apoyar en la herramienta Locust previamente comentada y generar un tráfico considerable de personas virtuales y así obtener el análisis deseado. En nuestro caso hemos generado 300 usuarios tal y como expusimos con anterioridad.

El resultado obtenido en nuestra carga de trabajo del tráfico según los porcentajes ha quedado de la siguiente manera:

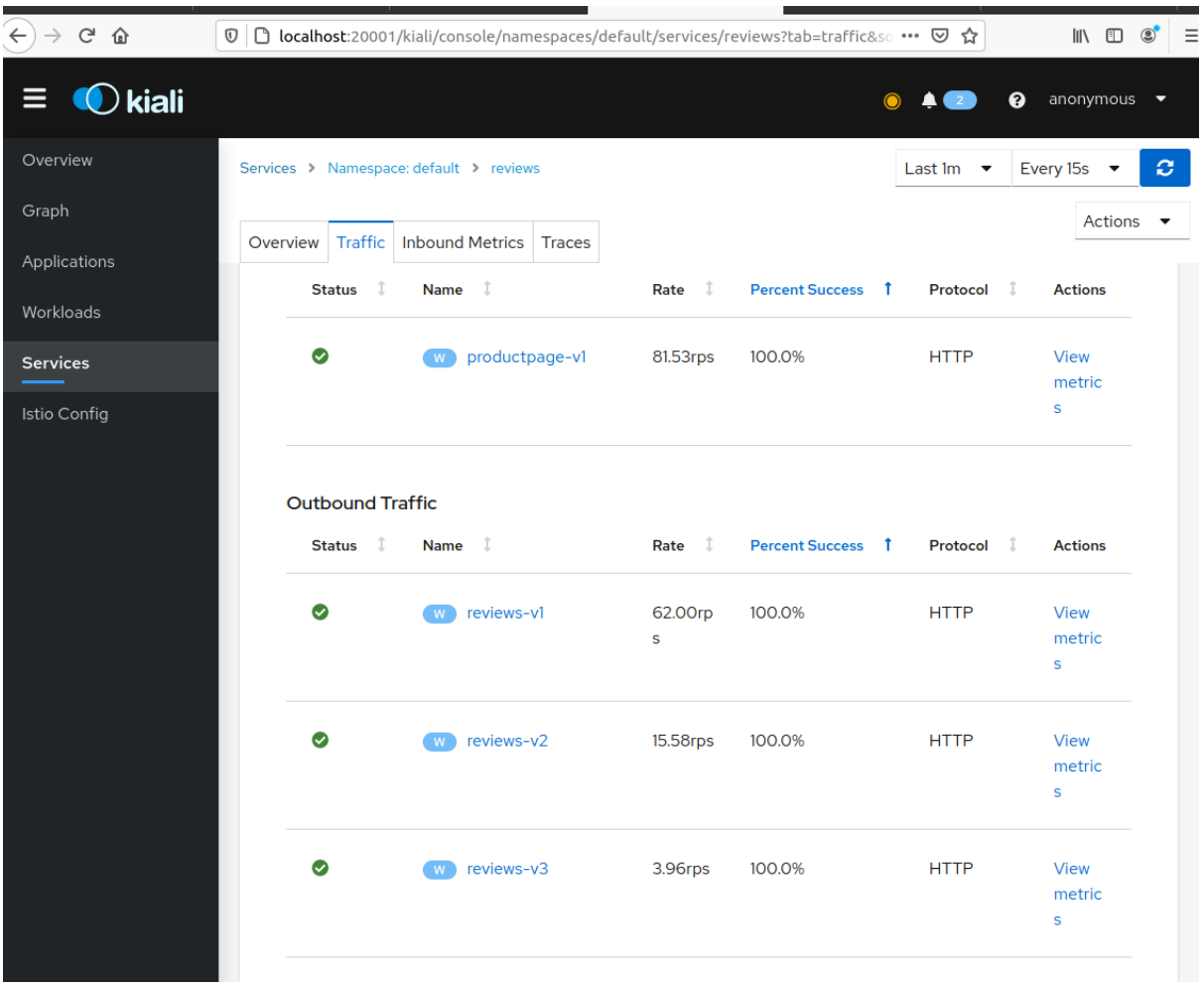


Figura 44. *Tráfico según porcentajes de carga de trabajo.*

Como se puede comprobar hay una media de 81,53 peticiones por segundo (RPS) a la aplicación *productpage* de las cuales se derivan 62,00 (76%) a la versión 1 de *reviews,* 15.58 (19%) a la versión 2 de *reviews* y 3,96 (5%) a la versión 3 de *reviews* haciendo efectiva la funcionalidad del desvío del tráfico así como también la capacidad de observar estas casuísticas.

Como punto adicional, Istio permite introducir “falsos erorres” para comprobar el porcentaje de éxito de una petición, entre otras diversas funcionalidades.

4.2.7 Trazas distribuidas con Jaeger

A continuación desplegamos Jaeger, la herramienta que nos proporciona la capacidad de rastrear las trazas de manera distribuida y así obtener información sobre el camino que toman. Para ello debemos ejecutar el siguiente comando:

```
$ kubectl apply -f
 https://raw.githubusercontent.com/istio/istio/release-1.9/samples/addons/jaeger.yaml
```

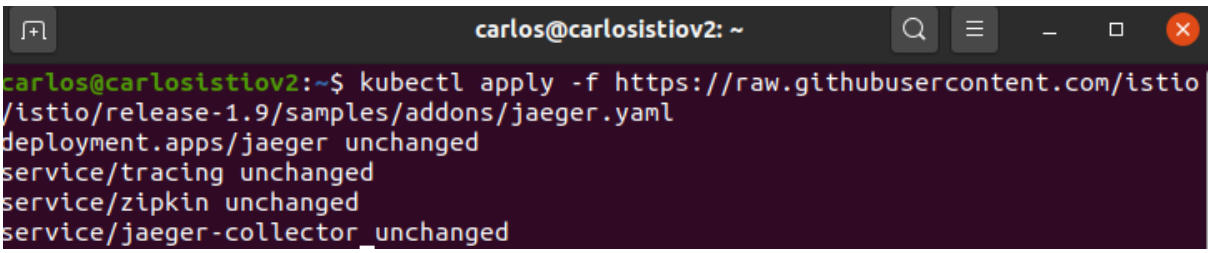


Figura 45. *Resultado despliegue Jaeger.*

Debido a que ya habíamos desplegado todas las herramientas externas este tendría que ser el mensaje que nos muestre. Como apunte, Jaeger es capaz de establecer la tasa de muestras a tener en cuenta. Por defecto viene configurado con 1%, es decir, que se muestra 1 traza por cada 100 peticiones que pasan por la aplicación. Esto se puede cambiar en un archivo de configuración llamado "*tracing.yaml*" y establecer cualquier porcentaje entre 0.0 y 100.0.

Una vez instalado, ejecutamos el siguiente comando para levantar la herramienta y acceder a su panel de control:

```
$ istioctl dashboard jaeger
```

Al igual que en el apartado anterior, nos apoyaremos en Locust para generar peticiones masivas para probar la herramienta y analizar su comportamiento. Usaremos la misma cantidad de usuarios con las mismas peticiones por segundo.

Tras iniciar el panel de control, nos encontraremos un panel en el lado izquierdo donde podremos filtrar la búsqueda de nuestras trazas según el servicio. En *Service* seleccionamos *productpage.default* y seleccionamos "*Find traces*" para así encontrar las últimas peticiones.

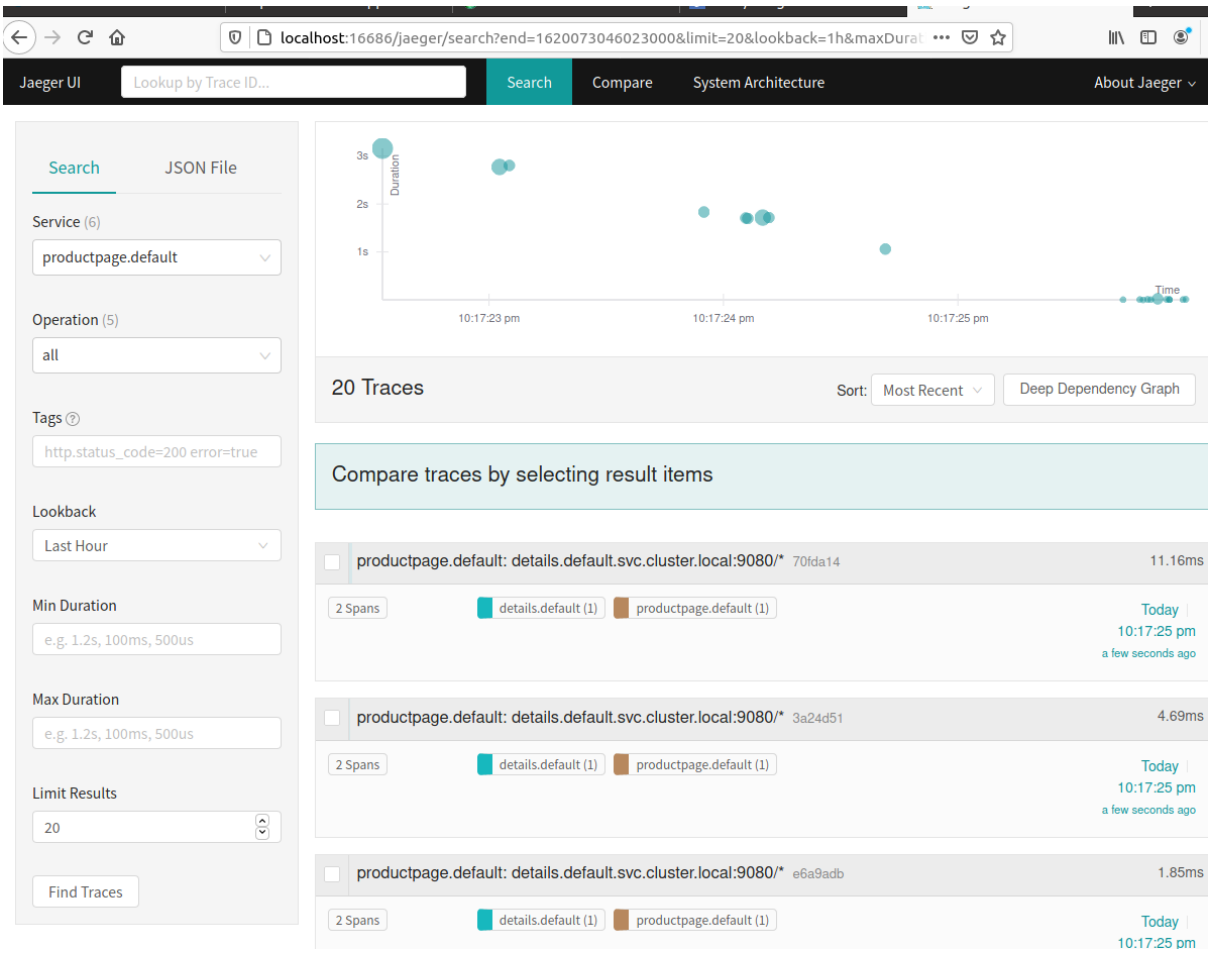


Figura 46. *Buscador de trazas del panel de control de Jaeger.*

Si seleccionamos una de las trazas nos mostraría el siguiente diagrama con los tiempos de los diferentes tramos entre los microservicios y el tiempo total de la petición según ha ido viajando por cada uno de ellos:

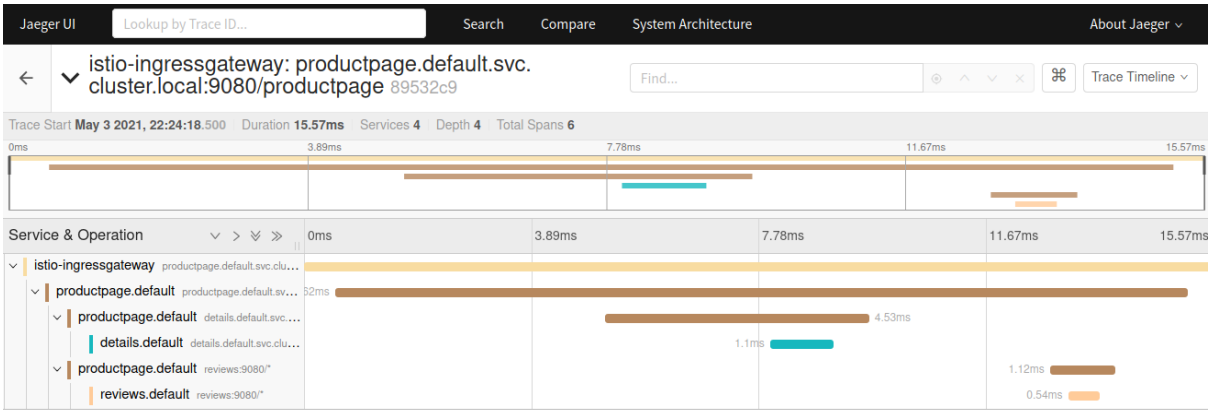


Figura 47. *Análisis de primera traza en Jaeger.*

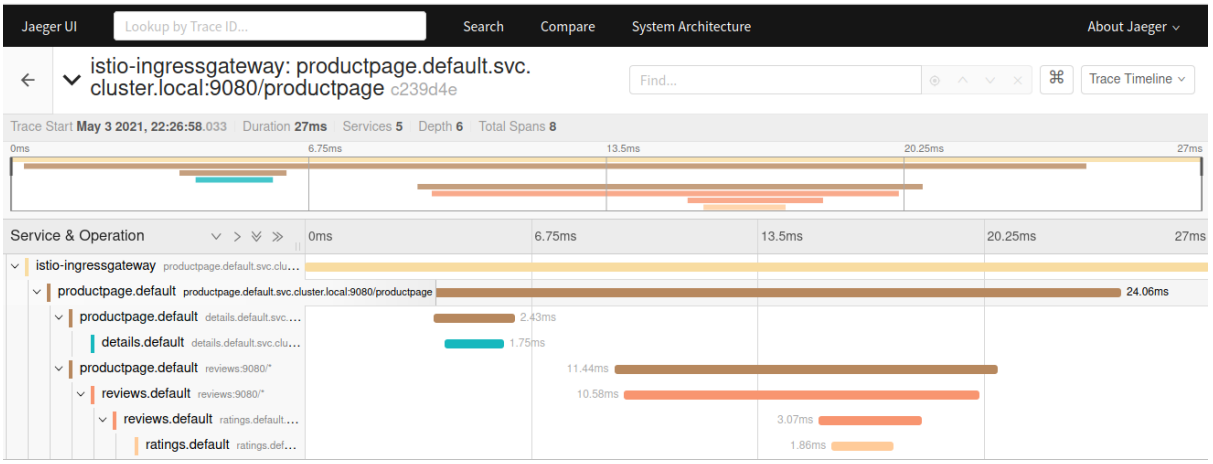


Figura 48. *Análisis de una segunda traza en Jaeger.*

Toda esta información por ejemplo nos permitiría analizar cuál microservicio es el que está creando latencia en caso de saturación de peticiones entre otras posibles utilidades que se le pueda encontrar tras analizar el comportamiento de nuestra traza distribuida.

4.3. Desinstalación y limpieza

Una vez acabadas las pruebas y cuando ya no se quiera trabajar con ninguna herramienta, sólo queda limpiar los despliegues. Primero cerramos cada una de las terminales donde hemos ido ejecutando los paneles de control (Kiali, Jaeger, Locust, etc…) Y después limpiaremos el despliegue de la aplicación. Para ello sólo tendríamos que ejecutar el archivo "*cleanup.sh*" que se encuentra en la carpeta de *samples/bookinfo/platform/kube/cleanup.sh.* Nos valdría con ejecutar el comando:

```
$ samples/bookinfo/platform/kube/cleanup.sh
```

```
carlos@carlosistiov2:~/Escritorio/istio$ samples/bookinfo/platform/kube/cleanup.sh
namespace ? [default]
using NAMESPACE=default
destinationrule.networking.istio.io "reviews" deleted
virtualservice.networking.istio.io "bookinfo" deleted
virtualservice.networking.istio.io "reviews" deleted
gateway.networking.istio.io "bookinfo-gateway" deleted
Application cleanup may take up to one minute
service "details" deleted
serviceaccount "bookinfo-details" deleted
deployment.apps "details-v1" deleted
service "ratings" deleted
serviceaccount "bookinfo-ratings" deleted
deployment.apps "ratings-v1" deleted
service "reviews" deleted
serviceaccount "bookinfo-reviews" deleted
deployment.apps "reviews-v1" deleted
deployment.apps "reviews-v2" deleted
deployment.apps "reviews-v3" deleted
service "productpage" deleted
serviceaccount "bookinfo-productpage" deleted
deployment.apps "productpage-v1" deleted
```

Figura 49. *Resultado limpieza del despliegue.*

Si hiciésemos llamadas de kubectl para comprobar el estado de los servicios los resultados serían los siguientes a diferencia de cuando aparecían los pods de la aplicación en estado "*running*":

```
carlos@carlosistiov2:~/Escritorio/istio$ kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   6h37m
carlos@carlosistiov2:~/Escritorio/istio$ kubectl get pods
No resources found in default namespace.
```

Figura 50. *Resultado de consulta de servicios tras limpieza de del despliegue.*

5. Conclusiones

En este Trabajo de Fin de Grado se ha buscado profundizar en el análisis de la arquitectura de microservicios y en la implementación de la capa *service mesh* así como herramientas que hacen uso de esta malla para comprender mejor el funcionamiento de esta arquitectura.

Hemos comenzado analizando la diferencia entre la arquitectura monolítica y la arquitectura distribuida en microservicios. Sus ventajas y desventajas. Una vez introducidos ambos conceptos hemos continuado explicando qué es el *service mesh* o malla de servicios, cómo se implementa y las ventajas que presenta hacer uso de ella en nuestra arquitectura de microservicios.

Durante el proceso de implementación de la malla de servicios ha sido necesario investigar sobre las tecnologías sobre las que se apoya para hacerla funcionar. Esto ha permitido conocer cómo se encuentra el mercado actualmente con respecto a estas tecnologías, aprender nuevos conceptos y conocer distintas herramientas con las que, yo personalmente, nunca había trabajado con anterioridad.

Istio, como tecnología que implementa el *service mesh,* nos ofrece muchas funcionalidades adicionales en nuestra arquitectura de microservicios. Funcionalidades que no existen de manera nativa en los orquestadores de contenedores como Kubernetes y que hacen de Istio una herramienta complementaria bastante potente.

Entre estas funcionalidades se encuentran las herramientas de observabilidad entre las que se incluye Jaeger como analizador de trazas distribuidas, Kiali para visualización de servicios, estado de salud, visualización o gestión del tráfico de las peticiones; Prometheus para recolectar datos métricos, y además, capas adicionales de seguridad para encriptar las comunicaciones, etc…

Todo esto puede parecer ventajoso a primera vista, sin embargo añadir esta capa a nuestra arquitectura de microservicios puede dotarla de cierta complejidad. Sobre todo, al principio cuesta bastante comprender cómo funcionan cada una de las herramientas, de qué manera se comunican y cómo instalarlas de manera satisfactoria, por lo que se necesita una buena cantidad de tiempo inicial para entrar en estas tecnologías. Esto, a nivel de conocimientos personales. A nivel técnico, añadir una capa a los microservicios también da complejidad a la aplicación al aumentar el número de instancias de ejecución y puede generar ciertas latencias entre las llamadas, que en principio en una aplicación como la que se ha estudiado en esta prueba con 4 microservicios no es significativa, pero quizás añadir esta

capa a una arquitectura compleja como *Netflix* puede suponer un aumento en la carga de trabajo.

Podemos concluir que la capa *service mesh* y las herramientas que tiene alrededor son elementos útiles y quizás complejos al principio pero resuelven algunos de los problemas que nos presenta la arquitectura en microservicios y con el tiempo se desarrollarán nuevas herramientas sobre esta capa que nos proporcionen un mejor conocimiento de lo que ocurre en nuestros microservicios.

6. Bibliografía y referencias

[1] Wikipedia, “Arquitectura de microservicios”. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Arquitectura_de_microservicios [Accedido: 17-abr-2021]

M. Fowler, “Microservices”, *MartinFowler.com*, 2014. [En línea]. Disponible en: https://martinfowler.com/articles/microservices.html [Accedido: 17-abr-2021]

“Arquitectura de microservicios: qué es, ventajas y desventajas”, *Decide Soluciones*, 2019. [En línea]. Disponible en: https://decidesoluciones.es/arquitectura-de-microservicios/ [Accedido: 17-abr-2021]

[2] M. Yerukala, “Microservices vs monolithic architectures”, 2019. [En línea]. https://rancher.com/blog/2019/microservices-vs-monolithic-architectures [Accedido: 17-abr-2021]

[3] J. Toribio, “Consolida tu arquitectura de microservicios con Service Mesh”, *Paradigma Digital*, 2018. [En línea]. Disponible en: https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/ [Accedido: 18-nov-2020]

J. Lupión, “Introducción a los service mesh”, *The Cocktail Engineering*, 2017. [En línea]. Disponible en: https://thecocktail.engineering/introducci%C3%B3n-a-los-service-mesh-a6505c29302 [Accedido: 18-nov-2020]

A. Rodríguez, “Arquitectura de microservicios: trazabilidad de peticiones”, *Paradigma Digital*, 2016. [En línea]. Disponible en: https://www.paradigmadigital.com/dev/arquitectura-de-microservicios-trazabilidad-de-peticiones/ [Accedido: 18-nov-2020]

T. Rodríguez, “De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo”, 2019. [En línea] https://www.xataka.com/otros/docker-a-kubernetes-entendiendo-que-contenedores-que-mayores-revoluciones-industria-desarrollo [Accedido: 17 de abril-2021]

[4] A. Fernández, “Introducción a Kubernetes”, *Aprender DevOps,* 2018. [En línea]. Disponible en: https://aprenderdevops.com/introduccion-a-kubernetes/ [Accedido: 17 de abril-2021]

[5] A. Fernández, “Arquitectura de Kubernetes”, *Aprender DevOps,* 2018. [En línea]. Disponible en: https://aprenderdevops.com/arquitectura-de-kubernetes/ [Accedido: 17 de abril-2021]

The Kubernetes Authors, “Kubernetes Documentation”, 2021. [En línea]. Disponible en: https://kubernetes.io/docs/tasks/tools/ [Accedido: 29 de abril-2021]

A. Rodríguez, “Jugando con Istio: arquitectura e instalación”, *Paradigma Digital*, 2019. [En línea]. Disponible en: https://www.paradigmadigital.com/dev/jugando-con-istio-the-next-big-thing-en-microservicios-1-2/ [Accedido: 18-nov-2020] [Accedido: 18-nov-2020]

S. Manpathak, “Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul”, *Platform9*, 2019. [En línea]. Disponible en: https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ [Accedido: 18-nov-2020]

[6] Istio Authors, “What is Istio?, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/concepts/what-is-istio/ [Accedido: 18 de abril-2021]

[7] Istio Authors, “Architecture”, *Istio,* 2020. [En línea]. Disponible en: https://istio.io/latest/docs/ops/deployment/architecture/ [Accedido: 18 de abril-2021]

[8] Istio Authors, “Security”, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/concepts/security/ [Accedido: 18 de abril-2021]

Istio Authors, “Observability”, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/tasks/observability/ [Accedido: 18 de abril-2021]

[9] Istio Authors, “Visualizing Metrics with Grafana”, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/tasks/observability/metrics/using-istio-dashboard/ [Accedido: 18 de abril-2021]

[10] N. Shapira, “Introduction to distributed tracing”, 2019. [En línea] https://epsagon.com/observability/introduction-to-distributed-tracing/ [Accedido: 18 de abril-2021]

[11] Z. Nagy *et al*, “b3-Propagation”, 2020. [En línea]. Disponible en: https://github.com/openzipkin/b3-propagation [Accedido: 24 de abril-2021]

[12] Istio Authors, “Distributed tracing: Zipkin”, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/tasks/observability/distributed-tracing/zipkin/ [Accedido: 18 de abril-2021]

[13] Prometheus Authors, “Documentation Prometheus”, *Prometheus*, 2014-2021. [En línea]. Disponible en: https://prometheus.io/ [Accedido: 24 de abril-2021]

[14] V. Khurana, “Introducción a Prometeus y Grafana”, *Geekflare*, 2020. [En línea]. Disponible en: https://geekflare.com/es/prometheus-grafana-intro/ [Accedido: 24 de abril-2021]

[15] B. Sigelman *et al*, “The OpenTracing Semantic Specification”, *GitHub,* 2020. [En línea]. Disponible en: https://github.com/opentracing/specification/blob/master/specification.md [Accedido: 24 de abril-2021]

[16] The Jaeger Authors, “Documentation Jaeger”, 2021. [En línea]. Disponible en: https://www.jaegertracing.io/docs/1.22/architecture/ [Accedido: 24 de abril-2021]

[17] Istio Authors, “Kiali”, *Istio*, 2020. [En línea]. Disponible en: https://istio.io/latest/docs/ops/integrations/kiali/ [Accedido: 25 de abril-2021]

[18] Kiali, “Documentation Kiali”, 2021. [En línea]. Disponible en: https://kiali.io/documentation/latest/architecture/ [Accedido: 25 de abril-2021]

J. Heyman *et al*, “What is Locust?”, Locust, 2021. [En línea]. Disponible en: https://docs.locust.io/en/stable/what-is-locust.html [Accedido: 25 de abril-2021]

Salvatorelab, “Locust: alternativa a JMeter para pruebas de carga”, *Testeando Software*, 2021. [En línea]. Disponible en: https://testeandosoftware.com/locust-alternativa-a-jmeter-para-pruebas-de-carga/ [Accedido: 25 de abril-2021]

[19] Istio Authors, “Bookinfo Application”, *Istio,* 2020. [En línea]. Disponible en: https://istio.io/latest/docs/examples/bookinfo/ [Accedido: 29 de abril-2021]

Y. Said, “How to Install and Configure Docker on Ubuntu 20.04”, *Linuxhint*, 2020. [En línea]. Disponible en: https://linuxhint.com/install_configure_docker_ubuntu/ [Accedido: 29 de abril-2021]

The Kubernetes Authors, “Minikube start”, Minukube, 2021. [En línea]. Disponible en: https://minikube.sigs.k8s.io/docs/start/ [Accedido: 29 de abril-2021]

[20] Wikipedia, “Prueba A/B”. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Prueba_A/B [Accedido: 12-may-2021]

[21] Istio Authors, Github Istio Bookinfo Sample, 2021. [En línea]- Disponible en: https://github.com/istio/istio/tree/master/samples/bookinfo [Accedido: 13-may-2021]

[21] Istio Authors, Github Istio Bookinfo Sample, 2021. [En línea]- Disponible en: https://github.com/istio/istio/tree/master/samples/bookinfo [Accedido: 13-may-2021]

[22] Istio Authors, Github Istio Bookinfo Sample, HTML template, “productpage.html”, 2021. [En línea] - Disponible en: https://github.com/istio/istio/blob/master/samples/bookinfo/src/productpage/templates/productpage.html [Accedido: 13-may-2021]

[23] Istio Authors, Github Istio Bookinfo Sample, Productpage service, “productpage.py” 2021. [En línea]- Disponible en: https://github.com/istio/istio/blob/master/samples/bookinfo/src/productpage/productpage.py [Accedido: 13-may-2021]

[24] Istio Authors, Github Istio Bookinfo Sample, Details service, “details.rb”, 2021. [En línea]- Disponible en: https://github.com/istio/istio/blob/master/samples/bookinfo/src/details/details.rb [Accedido: 13-may-2021]

[25] Istio Authors, Github Istio Bookinfo Sample, Review service, “LibertyRestEndpoint.java”, 2021. [En línea]- Disponible en: https://github.com/istio/istio/blob/master/samples/bookinfo/src/reviews/reviews-application/src/main/java/application/rest/LibertyRestEndpoint.java [Accedido: 14-may-2021]

[26] Istio Authors, Github Istio Bookinfo Sample, Ratings service, “ratings.js”, 2021. [En línea]- Disponible en: https://github.com/istio/istio/blob/master/samples/bookinfo/src/ratings/ratings.js [Accedido: 14-may-2021]

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá