



# **Estudio y evaluación de la resiliencia de Aplicaciones JBoss en clúster de kubernetes**

**Gema Cabanillas Morales**

Grado de Ingeniería Informática

Área: GNU/Linux

**Tutor: Joaquin López Sanchez-Montañes**

Junio 2024

## FICHA DEL TRABAJO FINAL

<b>Título del trabajo:</b>	Estudio y evaluación de la resiliencia de Aplicaciones JBoss en clúster de kubernetes
<b>Nombre del autor:</b>	<i>Gema Cabanillas Morales</i>
<b>Nombre del consultor/a:</b>	Joaquin López Sanchez-Montañes
<b>Nombre del PRA:</b>	
<b>Fecha de entrega (mm/aaaa):</b>	06/2024
<b>Titulación:</b>	Grado de Ingeniería Informática
<b>Área del Trabajo Final:</b>	<i>GNU/Linux</i>
<b>Idioma del trabajo:</b>	<i>Castellano</i>
<b>Palabras clave</b>	Kubernetes, Jboss, WildFly
<p><b>Resumen del Trabajo (máximo 250 palabras):</b> <i>Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.</i></p>	
<p>El Trabajo de Fin de Grado tiene como objetivo desplegar una aplicación Java contenerizada en un entorno estable utilizando la tecnología Docker, para posteriormente publicarla en alta disponibilidad mediante un clúster Kubernetes.</p> <p>El trabajo se estructura en dos partes. En la primera, se llevará a cabo un estudio de la infraestructura necesaria para el despliegue, instalación y configuración del entorno. Esta fase incluirá la configuración de la herramienta Jboss, que proporcionará una comprensión sólida de cómo funciona un clúster de este gestor de aplicaciones. Además de la configuración para su uso en entornos contenerizados, estudiando las tecnologías Docker y Kubernetes.</p> <p>La segunda parte del trabajo se centrará en la realización de pruebas y tests para evaluar la estabilidad del clúster desplegado. Se llevarán a cabo diversas pruebas para validar la capacidad del clúster para mantener una alta disponibilidad y manejar cargas de trabajo variables.</p> <p>El objetivo final del trabajo es demostrar la viabilidad y eficacia de utilizar kubernetes para configurar un clúster de JBoss de forma que siempre se mantenga una alta disponibilidad, usando un clúster dinámico, capaz de desplegar y mantener aplicaciones Java en entornos empresariales.</p>	

**Abstract (in English, 250 words or less):**

The Final Degree Project aims to deploy a containerized Java application in a stable environment using Docker technology, and subsequently publish it with high availability through a Kubernetes cluster.

The work is structured into two parts. In the first part, a study will be conducted on the necessary infrastructure for deployment, installation, and configuration of the environment. This phase will include the configuration of the Jboss tool, which will provide a solid understanding of how a cluster of this application manager works. In addition to the configuration for use in containerized environments, studying Docker and Kubernetes technologies.

The second part of the work will focus on conducting tests and tests to evaluate the stability of the deployed cluster. Various tests will be carried out to validate the cluster's ability to maintain high availability and handle variable workloads.

The ultimate goal of the work is to demonstrate the viability and effectiveness of using Kubernetes to configure a JBoss cluster in such a way that high availability is always maintained, using a dynamic cluster capable of deploying and maintaining Java applications in enterprise environments.

# Índice

<b>Lista de figuras</b> .....	<b>6</b>
1. Introducción.....	7
1.1 Contexto y justificación del Trabajo.....	7
1.2 Objetivos del Trabajo.....	8
1.3 Enfoque y método seguido.....	8
1.4 Planificación del Trabajo.....	8
1.5 Breve resumen de productos obtenidos.....	10
1.6 Breve descripción de los otros capítulos de la memoria.....	11
2. Investigación.....	13
2.1 Contenedores.....	13
2.2 Orquestadores.....	14
2.3 Orquestadores conocidos.....	15
2.4. Elementos básicos kubernetes.....	16
3. Diseño.....	17
3.1 Criterio selección de herramientas.....	17
4. Implementación y pruebas.....	17
4.1 Instalación y configuración de kubernetes.....	17
4.2 Instalación de Docker.....	18
4.3 Instalación de Minikube.....	19
4.3.1 Dashboard de Minikube.....	20
4.4 Configuración docker como driver de Minikube.....	21
4.5 Arquitectura de WildFly.....	23
4.6 Software necesario para Jboss.....	23
4.6.1 Diferencia entre Jboss y WildFly.....	24
4.7 Configuración instancias WildFly desde YAML.....	24
4.8 Configuración almacenamiento persistente: PV y PVC.....	26
4.9 Creación de usuarios para los nodos que conforman el clúster.....	28
4.10 Configuración de Service en Kubernetes.....	30
4.11 Diferencias WildFly en modo standalone y en modo domain.....	31
4.12 Dockerfile para Jboss en modo standalone-full-ha.xml.....	32
4.13 Construir nueva imagen.....	34
4.13.1 Docker Hub.....	35
4.13.2 Arrancar contenedor local.....	35
4.13.3 Arrancar imagen en kubernetes.....	36
4.13.4 Hello World en WildFly standalone.....	38
4.15 Dockerfile Domain.....	40
4.15.1 Domain.xml.....	41
4.15.2 Host.xml.....	42
4.15.3 domain-service.yaml.....	47

4.16 Acceso consola entorno gráfico.....	48
4.17 Hello World en WildFly Domain.....	49
4.18 Dockerfile Host.....	51
4.18.1 Host.xml.....	52
4.18.2 jboss_host.yaml.....	53
4.18.3 host-service.yaml.....	55
4.19 Server-group en WildFly, Host2.....	57
4.19.1 Ingress Controller.....	59
4.19.2 hello.mydomain.com.....	62
4.20 HaProxy.....	63
4.20.1 Balanceo de carga.....	66
4.20.2 Pruebas balanceo.....	67
5. Materiales.....	68
5.1 Hardware utilizado.....	68
5.2 Software y herramientas.....	68
6. Resultados.....	69
7. Conclusión.....	69
8. Glosario.....	71
9. Bibliografía.....	73

## Lista de figuras

- Figura 1. Logo Docker
- Figura 2. Arquitectura Diseñada
- Figura 3. Instalación repositorio Docker
- Figura 4: Instalación Docker
- Figura 5. Inicio servicio Docker
- Figura 6. Inicio entorno gráfico minikube
- Figura 7. Conectamos Minikube a Docker
- Figura 8. Inicio dashboard minikube
- Figura 9. Consola de Kubernetes
- Figura 10. Configuración driver docker para Minikube
- Figura 11. Agregar usuario al grupo docker
- Figura 12. Estado servicio Minikube
- Figura 13. Nodos ejecución
- Figura 14. Arquitectura a utilizar
- Figura 15. Deployment jboss-instance1
- Figura 16. Listado pods
- Figura 17. Yaml Persistent Volume
- Figura 18. Yaml Persisten Volume Claim
- Figura 19. Configurar almacenamiento
- Figura 20. Listado discos creados

Figura 21. Borrado deployments  
Figura 22. Modificación Deployments  
Figura 23. Consola bash al pod  
Figura 24. Servicio Wildfly ejecutándose en pod  
Figura 25. Creación usuario administrador en WildFly  
Figura 26. Creación usuario host en WildFly  
Figura 27. Creación servicio jboss-instance1  
Figura 28. Aplicar configuración servicio jboss-instance1  
Figura 29. Comprobar IP del nodo  
Figura 30. Acceso gráfico a WildFly  
Figura 31. Dockerfile WildFly standalone  
Figura 32. Copia ficheros desde el contenedor a local  
Figura 33. Detalle comando CMD en Dockerfile  
Figura 34. Esquema imagen contenedores  
Figura 35. Creación imagen WildFly  
Figura 36. Acceso a Docker Hub  
Figura 37. Subida a Docker Hub de imagen  
Figura 38. Yaml Deployment jboss-instance1 con imagen personalizada  
Figura 39. Listado de pods con dos réplicas  
Figura 40. Consola administración Wildfly  
Figura 41. Yaml servicio acceso jboss-instance1  
Figura 42. Acceso a aplicación de prueba "Hello World"  
Figura 43. Listado pods aumento de réplicas a 4  
Figura 44. Configuración por defecto domain controller  
Figura 45. Configuración domain controller  
Figura 46. Configuración por defecto de conexión a base de datos  
Figura 47. Configuración nombre de nodo máster en host.xml  
Figura 48. Configuración por defecto de interfaces en host.xml  
Figura 49. Listado de servicios creados en consola de kubernetes  
Figura 50. Configuración de interfaces en host.xml  
Figura 51. Yaml Deployment jboss-domain  
Figura 52. Dockerfile usado para crear Domain Controller  
Figura 53. Construcción imagen para Domain Controller  
Figura 54. Subimos a Docker Hub imagen Domain Controller  
Figura 55. Lanzamos Deployment jboss\_master  
Figura 56. Configuración servicio Domain Controller  
Figura 57. Consola gráfica WildFly con Domain Controller  
Figura 58. Despliegue aplicación pruebas en consola de administración  
Figura 59. Despliegue correcto de helloworld  
Figura 60. Nodo máster en consola de administración  
Figura 61. Acceso a aplicación de prueba desde servicio  
Figura 62. Dockerfile Host Controller  
Figura 63. Configuración nombre de nodo host1  
Figura 64. Configuración usuario Wildfly en host1  
Figura 65. Configuración remote Domain Controller en host.xml  
Figura 66. Yaml Deployment Host1  
Figura 67. Host1 en consola de WildFly  
Figura 68. Yaml del Servicio jboss-host  
Figura 69. Log WildFly máster reconociendo a nodo host  
Figura 70. Consola de administración de WildFly con dos nodos

Figura 71. Configuración server-groups en consola Domain de Wildfly  
Figura 72. Ingress  
Figura 73. Consola de WildFly mostrando los servicios.  
Figura 74. Instalación ingress en kubernetes  
Figura 75. Configuración Ingress  
Figura 76. Yaml de servicio host2  
Figura 77. Configuración nombre dominio DNS en fichero hosts local  
Figura 78. Acceso aplicación test desde DNS  
Figura 79. Log de ingress  
Figura 80. configuración ingress con dos servicios  
Figura 81. Yaml de deployment para haproxy  
Figura 82. Configmap Haproxy  
Figura 83. Servicio Haproxy  
Figura 84. Ingress de haproxy  
Figura 85. Log Haproxy  
Figura 86. Servicio server-new parado  
Figura 87. Log de haproxy con server-new parado  
Figura 88. Log de haproxy con server-new levantado

# 1. Introducción

## 1.1 Contexto y justificación del Trabajo

La necesidad a cubrir radica en proporcionar una disponibilidad del 100% a una aplicación web alojada en un entorno de servidor JBoss. Actualmente, muchas empresas confían en JBoss para desplegar sus aplicaciones, sin embargo, enfrentan el desafío de la indisponibilidad si el servidor JBoss en el que se ejecuta la aplicación experimenta una caída.

Para resolver este problema, se han implementado soluciones como el uso de varios nodos de JBoss para balancear la carga. Sin embargo, esta solución no garantiza una disponibilidad total y puede resultar en interrupciones del servicio en caso de fallos.

La solución propuesta implica la implementación de un clúster dinámico de WildFly en un entorno Kubernetes. Se pretende desplegar WildFly con 2 o más Pods/Réplicas capaces de formar un clúster de manera automatizada y dinámica. Para ello necesitamos implementar un mecanismo de descubrimiento de nodos automatizado que permita la formación del clúster de manera eficiente.

La principal meta es lograr una disponibilidad del 100% para la aplicación web, incluso en el caso de fallos en uno o más nodos del clúster.

El tema es relevante debido a la amplia adopción de JBoss en el entorno empresarial para el despliegue de aplicaciones web. La indisponibilidad de estas aplicaciones puede resultar en pérdidas económicas y de reputación para las empresas. La implementación de un clúster dinámico de WildFly en Kubernetes aborda directamente este problema al garantizar una alta disponibilidad y continuidad del servicio, incluso en situaciones de fallo del servidor.

La implementación de un clúster dinámico en un entorno Kubernetes, proporcionará una solución más robusta y escalable para las empresas que dependen de estas aplicaciones críticas.



## 1.2 Objetivos del Trabajo

Los objetivos de este Trabajo de Fin de Grado son los siguientes:

- Conocer mejor la tecnología Kubernetes.
- Profundizar en la herramienta de WildFly.
- Conseguir desplegar un clúster de WildFly en Kubernetes formado por varios nodos.

## 1.3 Enfoque y método seguido

Este trabajo consta de dos partes.

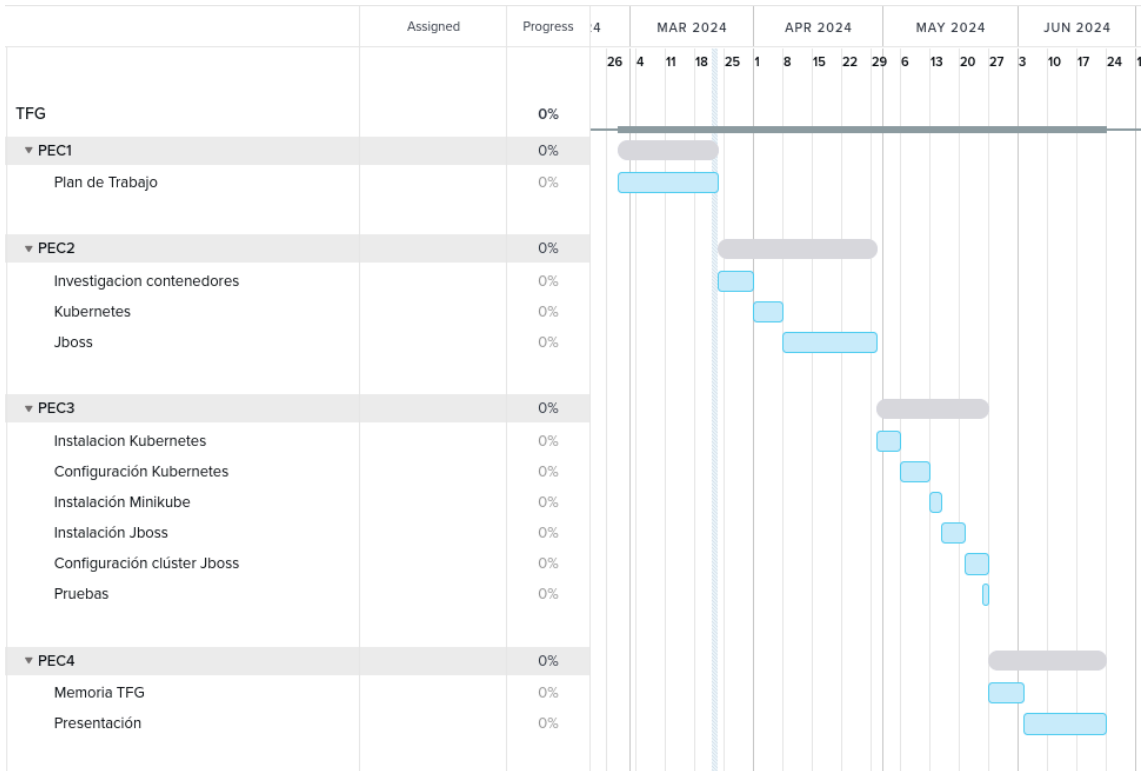
En la primera, vamos a estudiar y configurar la infraestructura, tanto de kubernetes como de WildFly.

Para ello, instalaremos minikube y configuraremos un Deployment de WildFly, con almacenamiento externo, dónde realizaremos todo lo necesario para la configuración como clúster.

La segunda parte del trabajo se centrará en la realización de pruebas y tests para evaluar la estabilidad del clúster desplegado.

El fin es adaptar un entorno ya utilizado en las empresas, como es un clúster de WildFly, pero intentando dar un poco de inteligencia al mismo, de modo que sea capaz de montar un clúster dinámicamente implementando un descubrimiento de nodos de forma automática.

## 1.4 Planificación del Trabajo



		Fecha Inicio	Fecha Fin
1ª Entrega	Plan de Trabajo	28/02/24	21/03/2024
2ª Entrega	Análisis y estudio tecnológico	13/03/2024	29/04/2024
3ª Entrega	Implementación	30/04/2024	26/05/2024
4ª Entrega	Memoria y presentación	27/05/2024	22/06/2024

## 1.5 Breve resumen de productos obtenidos

### WildFly:

Servidor de aplicaciones Java EE de código abierto.

#### Características:

Sucesor de JBoss AS Community Edition.

Compatible con las especificaciones Java EE.

Ofrece características avanzadas para el desarrollo, implementación y gestión de aplicaciones empresariales Java.

Uso Principal: Ideal para entornos de desarrollo y producción que requieren un servidor de aplicaciones robusto y flexible para aplicaciones Java EE.

### Kubernetes:

Plataforma de código abierto para la automatización de despliegues, escalado y operaciones de contenedores de aplicaciones.

#### Características:

Orquestación y gestión de contenedores en clústeres.

Facilita la alta disponibilidad, el equilibrio de carga y el escalado automático.

Uso Principal: Utilizado para gestionar aplicaciones contenerizadas.

### Docker:

Plataforma de código abierto diseñada para desarrollar, enviar y ejecutar aplicaciones dentro de contenedores.

#### Características:

Facilita la creación y gestión de contenedores de forma rápida y eficiente.

Ofrece portabilidad y simplifica el empaquetado de aplicaciones y sus dependencias.

Uso Principal: Utilizado para contenerizar aplicaciones, permitiendo un desarrollo ágil y aislado de otros entornos.

## **1.6 Breve descripción de los otros capítulos de la memoria**

Esta memoria consta de 9 capítulos.

### **1. Introducción**

Este capítulo trata de dar una explicación del entorno y la importancia del proyecto, destacando la relevancia de tecnologías como Docker, Kubernetes y WildFly.

### **2. Investigación**

Continuamos con una explicación de los conceptos básicos y la importancia de los contenedores en el desarrollo moderno de aplicaciones y de kubernetes.

### **3. Diseño**

En este capítulo tratamos de dar una explicación de los criterios utilizados para seleccionar las herramientas y tecnologías empleadas en el proyecto.

### **4. Implementación y pruebas**

Este capítulo es el más importante, se detalla la instalación y configuración de Kubernetes, WildFly y todos los componentes que hemos necesitado para implementar nuestro clúster.

Desde los pasos para instalar Minikube, hasta la arquitectura de JBoss. Desplegamos una aplicación de pruebas y detallamos la configuración del clúster.

### **6. Resultados**

En el capítulo 6 se presentan los resultados del trabajo, con un resumen de todo lo que hemos trabajado.

### **7. Conclusión**

El capítulo 7, trata este trabajo desde un punto de vista reflexivo, explicando aquello que podría ser investigado en más profundidad y tratando los puntos clave que más trabajo han costado.

## 8. Glosario

El capítulo 8, presenta un glosario que recoge los términos más relevantes utilizados.

## 9. Bibliografía

Y por último, el capítulo 9 lista la bibliografía, con los enlaces visitados y aplicados en este trabajo.

## 2. Investigación

### 2.1 Contenedores

Un contenedor representa una entidad de software compacta y autónoma que engloba todos los elementos necesarios para poner en marcha una aplicación, incluyendo sus dependencias, bibliotecas y configuraciones. Su propósito es encapsular las aplicaciones y los elementos que lo forman, facilitando así su despliegue, tanto en entornos locales de desarrollo hasta servidores en la nube y centros de datos.

Una forma de comprender el concepto de contenedor es establecer una analogía con los contenedores convencionales utilizados en el transporte de mercancías: estos contienen un conjunto modular que puede integrarse con otros contenedores. El logotipo de Docker, una de las tecnologías de contenedores más difundidas, refleja precisamente esta similitud.



Figura 1. Logo Docker

Gracias a que los contenedores incorporan todo lo necesario para la aplicación, resulta sencillo trasladarla entre distintos entornos o sistemas operativos, incluso en la nube. Esta característica dota a los contenedores de una notable ventaja en términos de portabilidad.

Los contenedores necesitan un entorno para ejecutarse y se destacan por su ligereza, ya que se basan en el sistema operativo del host donde se despliegan. A diferencia de las máquinas virtuales, no albergan un sistema operativo completo.

Las principales ventajas que ofrecen en comparación con otras arquitecturas de despliegue son:

- **Ligereza:** Al compartir el kernel del sistema operativo del host, no requieren una instancia completa del sistema por cada aplicación. Esto se traduce en contenedores de reducido tamaño y uso eficiente de recursos. Su tamaño compacto permite un arranque rápido cuando se necesita escalar horizontalmente.
- **Portabilidad e independencia de plataforma:** Al llevar consigo todas sus dependencias, el software puede desarrollarse una vez y ejecutarse sin necesidad de reconfiguraciones en distintos entornos.
- **Aislamiento:** Los contenedores ofrecen un entorno aislado para la aplicación que albergan, lo que impide que los procesos internos interactúen con el sistema operativo del host o con otros contenedores, aunque sí pueden comunicarse entre sí.
- **Escalabilidad:** Son capaces de expandirse fácilmente para atender las necesidades de los usuarios de la aplicación. Se pueden generar múltiples instancias del mismo contenedor para distribuir la carga de trabajo de manera eficiente.
- **Flexibilidad:** Permiten a los desarrolladores manejar distintas versiones de bibliotecas y dependencias de una aplicación sin interferir en otras aplicaciones o en el sistema operativo principal.
- **Inmutabilidad:** Si el proceso del contenedor se detiene repentinamente o por solicitud del usuario, cualquier modificación realizada directamente en el contenedor se perderá. Considerar esta característica contribuye significativamente a la estabilidad del servicio ofrecido.

## 2.2 Orquestadores

Un orquestador de contenedores es una herramienta de software diseñada para facilitar la gestión, implementación y escalabilidad de aplicaciones basadas en contenedores. Su función principal es automatizar y coordinar las operaciones necesarias para administrar múltiples contenedores.

La orquestación de contenedores significa automatización de las operaciones necesarias para gestionar y ejecutar aplicaciones y servicios alojados en contenedores. En entornos donde se emplea un gran número de contenedores, la gestión manual resulta ineficiente y complicada, se necesitan herramientas más avanzadas para automatizar tareas de mantenimiento como la sustitución de contenedores fallidos, la recuperación automática, los ajustes de configuración, actualizaciones periódicas y la gestión de la escalabilidad, tanto horizontal como vertical, en respuesta a la demanda. En este escenario, los orquestadores de contenedores se vuelven esenciales para simplificar la administración y ejecución de estas aplicaciones.

En la mayoría de los orquestadores un desarrollador crea un archivo de configuración (normalmente en formato YAML o JSON) que describe el estado deseado de la configuración. La herramienta de orquestación interpreta este archivo y utiliza su lógica interna para materializar la definición realizada.

El orquestador se encarga de desplegar los contenedores y sus réplicas para garantizar la resiliencia del entorno en un host, seleccionando el más adecuado según la capacidad de CPU disponible, la memoria y otros requisitos o restricciones definidos en el archivo de configuración.

## 2.3 Orquestadores conocidos

Los orquestadores más conocidos a día de hoy son:

- Kubernetes

También llamado k8s (el 8 representa la cantidad de letras entre la “K” y la “s”), es una plataforma de código abierto desarrollado por Google.

Kubernetes automatiza las tareas operativas de la administración de contenedores e incluye comandos integrados para implementar aplicaciones, actualizarlas, escalarlas y todo lo necesario para la administración de las aplicaciones.

Kubernetes VS Docker:

La diferencia entre Docker y Kubernetes está en la función que cada uno cumple en la creación de contenedores y la ejecución de las aplicaciones.

Docker es un estándar de la industria para el empaquetado y la distribución de aplicaciones en contenedores. Kubernetes usa Docker para implementar, administrar y escalar aplicaciones en contenedores. Podemos usar Kubernetes sin Docker.

- OpenShift

OpenShift está basado en Kubernetes pero mantenida por Red Hat, lo que puede ser una ventaja para las organizaciones que buscan un nivel de soporte y servicio más alto.

OpenShift, además de incluir todas las características de Kubernetes, agrega herramientas adicionales para facilitar la implementación, gestión y escalabilidad de aplicaciones en contenedores. Estas herramientas incluyen servicios de desarrollador, automatización de pipelines, gestión de contenedores de almacenamiento, entre otros.

Además, OpenShift se limita a sistemas operativos Linux, Fedora y CentOS. Dispone de un enfoque más integrado en términos de seguridad, cosa que Kubernetes no tiene.



Proporciona una experiencia más completa para el ciclo de vida de las aplicaciones, desde el desarrollo hasta la producción, ofreciendo capacidades como la integración continua, entrega continua (CI/CD), monitoreo integrado, entre otros, que están integrados de manera nativa en la plataforma.

#### 2.4. Elementos básicos kubernetes

- Pod: Es la unidad más pequeña de trabajo de Kubernetes. Cada pod contiene uno o más contenedores. Todos los contenedores en un pod tienen la misma IP y puerto, pueden comunicarse utilizando un proceso interno de comunicación. Los contenedores de un pod pueden acceder al almacenamiento local del nodo que aloja al pod.
- Node/Worker: Un nodo es un único host, virtual o físico. Su trabajo es ejecutar pods. Cada nodo ejecuta varios componentes como kubelet y kube proxy. Cada clúster tiene mínimo un nodo worker.
- Master: Es el plano de control de Kubernetes. Está formado por varios elementos como un API server, un scheduler y un controller manager. Se encarga de tomar las decisiones globales sobre el clúster, de la planificación de los pods y del manejo de eventos.
- Label: Las etiquetas (labels) son pares clave-valor asociados a recursos como pods, servicios, replicaset, entre otros. Estas etiquetas son metadatos que se pueden adjuntar de manera flexible y personalizada a los objetos de Kubernetes. Sirven para identificar y organizar los recursos de manera lógica y significativa. Usan selectors para seleccionar los objetos según su label.
- Replica sets: Los controladores de replicación y los conjuntos de réplica administran un grupo de pods identificados por un selector. Aseguran que cierto número de elementos estén siempre en ejecución.
- Services: Se utilizan para exponer aplicaciones a los usuarios o a otros servicios. Facilitan la comunicación interna entre los componentes de una aplicación dentro del clúster, usando una IP virtual. Actúa como un balanceador de carga interno que distribuye el tráfico entre los pods del conjunto
- Ingress: Se utiliza para exponer servicios de aplicaciones a usuarios externos o a otras aplicaciones fuera del clúster de Kubernetes. Es especialmente útil para implementar reglas de enrutamiento avanzadas a objetos HTTP, terminación SSL, y balanceo de carga para aplicaciones web expuestas al tráfico externo.
- Volumen: El almacenamiento local de un pod es efímero y se pierden con el pod. Cuando la información debe preservarse o compartirse se usa un volumen.

- StatefulSet: Es un controlador de Kubernetes que se utiliza cuando es necesario manejar aplicaciones de forma persistente. Asegura que hay un número concreto de elementos ejecutándose en todo momento con un identificador único. Se usa por ejemplo en bases de datos, que debe haber un pod en ejecución siempre.
- Deployment: Es un objeto de kubernetes que se usa para declarar y administrar la implementación de una aplicación. Se declara en un fichero con formato JSON o YAML y en él se configura el número de réplicas que queremos de una aplicación. El deployment se encargará automáticamente de crear el número deseado de réplicas para que esté siempre en funcionamiento.
- Secrets: Los secretos son objetos que contienen información sensible, como las credenciales y tokens. Pueden montarse como ficheros dentro de los pods que necesiten acceder a ellos. Es posible montar un mismo secreto en varios pods.
- Namespaces: Es un espacio de nombres, en kubernetes es la forma de definir un dominio virtual aislado. Sirve para separar de forma lógica los recursos de un proyecto de otro dentro del clúster.

## 3. Diseño

### 3.1 Criterio selección de herramientas

Este trabajo lo hemos diseñado para usar herramientas open source en todo momento, con el objetivo de que cualquier empresa lo pueda implementar como solución a sus aplicaciones.

La arquitectura propuesta es la siguiente:

- Nodo físico o virtual con sistema operativo Linux
- Docker
- Kubernetes
- HaProxy
- WildFly

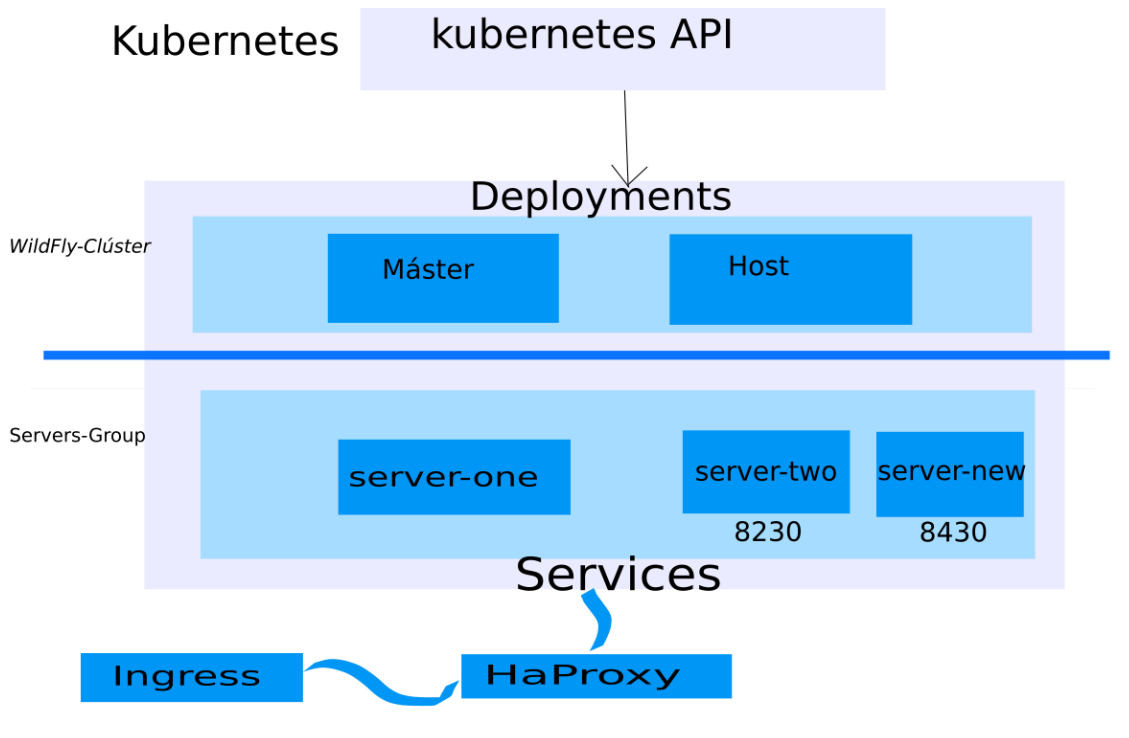


Figura 2. Arquitectura Diseñada

## 4. Implementación y pruebas

### 4.1 Instalación y configuración de kubernetes

Vamos a utilizar un sistema operativo linux, concretamente Fedora release 38.

Puesto que los requisitos de hardware para la instalación de Kubernetes son bastante elevados, vamos a utilizar una versión más ligera llamada Minikube.

La instalación de Minikube tiene como requisito instalar Docker. Docker es sinónimo de contenedor, es el software de gestión de contenedores más usado mundialmente, para el aislamiento a nivel de sistema operativo.

Lo atractivo de esta tecnología es que se puede hacer funcionar una aplicación, independientemente del entorno donde se ejecute el contenedor, de forma portátil y autosuficiente, ya que existe un aislamiento de los recursos que tenga nuestro host.

Kubernetes es un orquestador de contenedores, cuenta con una serie de procesos que mantienen el estado del sistema estable. Permite balancear la carga de los contenedores, orquestar el almacenamiento de estos incluso una auto regeneración en caso de caída. Administra los nodos o workers, donde corren las aplicaciones que queremos desplegar.

Por todo esto, Kubernetes se utilizará en el despliegue del clúster que vamos a configurar en este TFG.

## 4.2 Instalación de Docker

Instalamos docker, configurando previamente el repositorio para descargar los paquetes:

```
sudo dnf config-manager --add-repo  
https://download.docker.com/linux/fedora/docker-ce.repo
```

```
[Gema@local VirtualBox VMs]$ sudo dnf config-manager --add-repo https://download.docker.com/linux/fedora/docker-ce.repo  
Agregando repositorio de: https://download.docker.com/linux/fedora/docker-ce.repo
```

Figura 3: Instalación repositorio Docker

Después, actualizamos el repositorio de paquetes y procedemos a la instalación de Docker:

```
sudo dnf update
```

```
sudo dnf install docker-ce docker-ce-cli containerd.io --skip-broken
```

```
[Gema@local VirtualBox VMs]$ sudo dnf install docker-ce docker-ce-cli containerd.io --skip-broken  
Última comprobación de caducidad de metadatos hecha hace 0:00:33, el sáb 02 mar 2024 20:50:24.  
Dependencias resueltas.  
  
Problema: problema con el paquete instalado moby-engine-24.0.5-1.fc38.x86_64  
- package moby-engine-24.0.5-1.fc38.x86_64 from @System conflicts with docker-ce provided by docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable  
- package docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable conflicts with docker provided by moby-engine-24.0.5-1.fc38.x86_64 from @System  
- package docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable conflicts with docker provided by moby-engine-20.10.23-1.fc38.x86_64 from fedora  
- package moby-engine-20.10.23-1.fc38.x86_64 from fedora conflicts with docker-ce provided by docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable  
- package docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable conflicts with docker provided by moby-engine-24.0.5-1.fc38.x86_64 from updates  
- package moby-engine-24.0.5-1.fc38.x86_64 from updates conflicts with docker-ce provided by docker-ce-3:25.0.3-1.fc38.x86_64 from docker-ce-stable  
- cannot install the best candidate for the job  
=====
```

Paquete	Arquitectura	Versión	Repositorio	Tam.
Instalando:				
containerd.io	x86_64	1.6.28-3.1.fc38	docker-ce-stable	34 M
se sustituye containerd.x86_64	1.6.19-1.fc38			
se sustituye runc.x86_64	2:1.1.12-1.fc38			
Descartando paquetes con conflictos:				
(añada '--best --allowrasing' a la línea de comandos para forzar su actualización):				
docker-ce	x86_64	3:25.0.3-1.fc38	docker-ce-stable	26 M
moby-engine	x86_64	20.10.23-1.fc38	fedora	29 M

Figura 4. Instalación Docker

Iniciamos el servicio de docker y comprobamos que está activo:

```
[Gema@local VirtualBox VMs]$ sudo systemctl start docker  
[Gema@local VirtualBox VMs]$ systemctl status docker  
● docker.service - Docker Application Container Engine  
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; preset: disabled)  
   Drop-In: /usr/lib/systemd/system/service.d  
     Lto: /etc/systemd/system/service.d  
   Active: active (running) since Sat 2024-03-02 20:51:57 CET; 9s ago  
 TriggeredBy: ● docker.socket  
     Docs: https://docs.docker.com  
   Main PID: 76214 (dockerd)  
     Tasks: 21 (limit: 14188)  
    Memory: 152,6M  
       CPU: 673ms  
   CGroup: /system.slice/docker.service  
           └─76214 /usr/bin/dockerd --host=fd:// --exec-opt native.cgroupdriver=systemd --selinux-enabled --log-driver=journald --live-restore --default-ulimit nofile:  
             └─76221 containerd --config /var/run/docker/containerd/containerd.toml
```

Figura 5. Inicio servicio Docker

### 4.3 Instalación de Minikube

Para la instalación de Minikube hemos seguido estos pasos. Primero hemos descargado el ejecutable 'kubectl', que es la herramienta de línea de comandos de kubernetes:

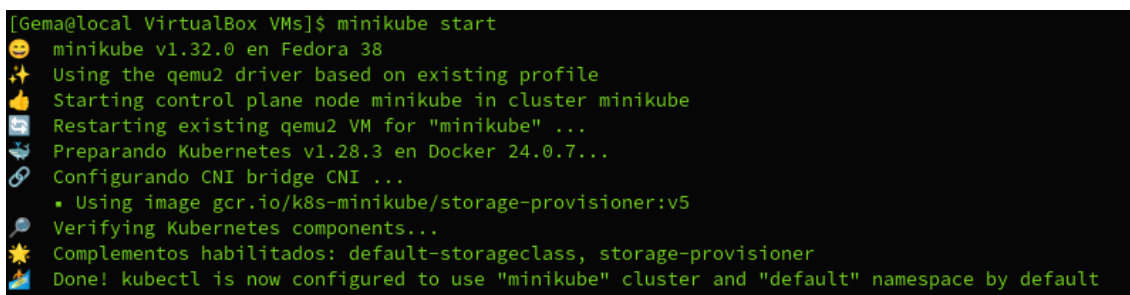
```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
```

Después de enviar una solicitud HTTP GET al enlace, para descargar kubectl, le damos permisos de ejecución y lo movemos al directorio de binarios de nuestra máquina.

```
chmod +x ./kubectl
```

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

```
minikube start
```



```
[Gema@local VirtualBox VMs]$ minikube start
🐾 minikube v1.32.0 en Fedora 38
🌟 Using the qemu2 driver based on existing profile
👍 Starting control plane node minikube in cluster minikube
🔄 Restarting existing qemu2 VM for "minikube" ...
🐳 Preparando Kubernetes v1.28.3 en Docker 24.0.7...
🔗 Configurando CNI bridge CNI ...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
🔍 Verifying Kubernetes components...
🌟 Complementos habilitados: default-storageclass, storage-provisioner
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Figura 6. Inicio entorno gráfico minikube

De esta forma, estamos usando Minikube con qemu2 como driver. Lo vamos a cambiar por docker como driver, porque es más rápido y ligero en cuanto a recursos, ya que utiliza los contenedores de Docker para ejecutar nodos.

Para ello, es necesario borrar cualquier cluster que existiera previamente usando el driver qemu2, con 'minikube delete'.

También será necesario que el entorno Docker esté conectado al clúster de Minikube. Para ello, hay que usar el siguiente comando para configurarlo:

```
[Gema@local jbossns]$ minikube docker-env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.49.2:2376"
export DOCKER_CERT_PATH="/home/gema/.minikube/certs"
export MINIKUBE_ACTIVE_DOCKERD="minikube"

# To point your shell to minikube's docker-daemon, run:
# eval $(minikube -p minikube docker-env)
[Gema@local jbossns]$
```

Figura 7. Conectamos Minikube a Docker

### 4.3.1 Dashboard de Minikube

Puede ser de utilidad activar el entorno gráfico de MiniKube.

```
[Gema@local jbossns]$ minikube dashboard
Habilitando dashboard
• Using image docker.io/kubernetes/dashboard:v2.7.0
• Using image docker.io/kubernetes/metrics-scraper:v1.0.8
Some dashboard features require the metrics-server addon. To enable all features please run:

  minikube addons enable metrics-server

🐛 Verifying dashboard health ...
🚀 Launching proxy ...
🐛 Verifying proxy health ...
🔗 Opening http://127.0.0.1:39387/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser...
Se está abriendo en una sesión de navegador existente.
```

Figura 8. Inicio dashboard minikube

Automáticamente levanta en el navegador:

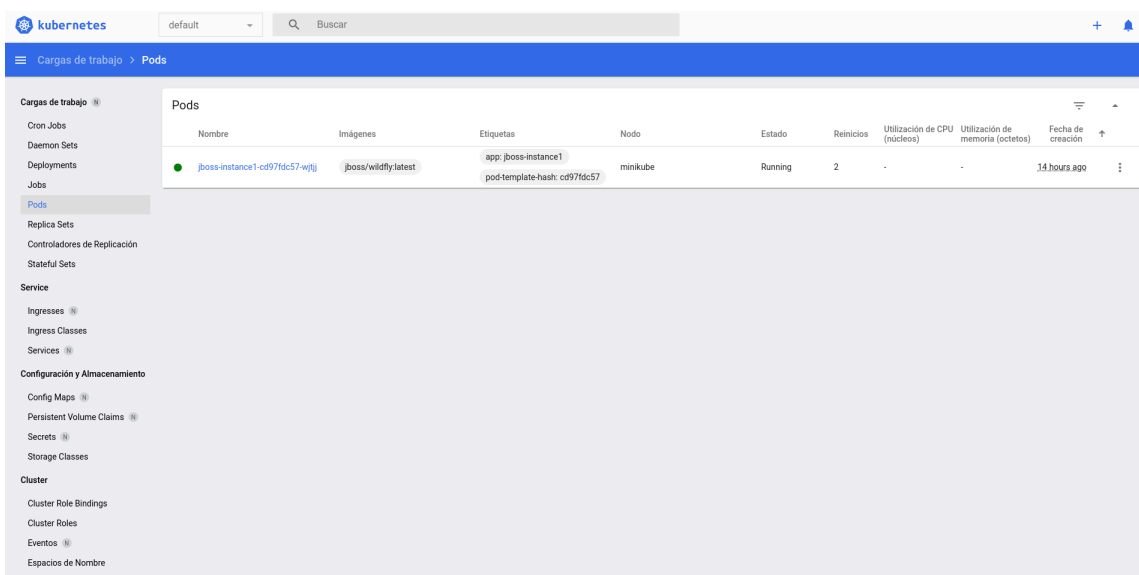


Figura 9. Consola de Kubernetes

## 4.4 Configuración docker como driver de Minikube

Para usar Docker como driver, ejecutamos:

```
minikube start --driver=docker
```

```
Gema@local ~]$ minikube start --driver=docker
minikube v1.32.0 en Fedora 38
Using the docker driver based on user configuration

Saliendo por un error PROVIDER_DOCKER_NEWGRP: "docker version --format <no value>--<no value>:<no value>" exit status 1: permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/version": dial unix /var/run/docker.sock: connect: permission denied
Suggestion: Add your user to the 'docker' group: 'sudo usermod -aG docker $USER && newgrp docker'
Documentación: https://docs.docker.com/engine/install/linux-postinstall/
```

Figura 10. Configuración driver docker para Minikube

Este error aparece porque el usuario no tiene permisos para acceder al socket de Docker, por eso se impide que Minikube lo use. Se soluciona agregando mi usuario al grupo 'docker':

```
[Gema@local ~]$ sudo usermod -aG docker $USER && newgrp docker
[sudo] contraseña para Gema:
[Gema@local ~]$
(reverse-i-search)`mini': ^Cminikube status
[Gema@local ~]$ minikube start --driver=docker
minikube v1.32.0 en Fedora 38
Using the docker driver based on user configuration
Using Docker driver with root privileges
Starting control plane node minikube in cluster minikube
Pulling base image ...
> gcr.io/k8s-minikube/kicbase...: 453.90 MiB / 453.90 MiB 100.00% 23.65 M
```

Figura 11. Agregar usuario al grupo docker

Ahora comprobamos que Minikube está ejecutándose correctamente.

```
[Gema@local ~]$ minikube start --driver=docker
minikube v1.32.0 en Fedora 38
Using the docker driver based on user configuration
Using Docker driver with root privileges
Starting control plane node minikube in cluster minikube
Pulling base image ...
> gcr.io/k8s-minikube/kicbase...: 453.90 MiB / 453.90 MiB 100.00% 23.65 M
Creating docker container (CPUs=2, Memory=2900MB) ...
Preparando Kubernetes v1.28.3 en Docker 24.0.7...
  • Generando certificados y llaves
  • Iniciando plano de control
  • Configurando reglas RBAC...
Configurando CNI bridge CNI ...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
Verifying Kubernetes components...
Complementos habilitados: storage-provisioner, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
[Gema@local ~]$ minikube status
minikube
type: Control Plane
host: Running
kubenet: Running
apiserver: Running
kubeproxy: Running
kubedns: Running
kubestorage: Running
kubecfg: Configured
```

Figura 12. Estado servicio Minikube

En este momento podemos mostrar los servidores que forman el clúster, siendo ahora mismo un único nodo.

```
[Gema@local ~]$ kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
minikube     Ready    control-plane  21d   v1.28.3
[Gema@local ~]$
```

Figura 13. Nodos ejecución

Lo recomendable para un clúster sería configurar distintas máquinas virtuales, de modo que una de ellas fuera el servidor máster y el resto de máquinas fueran los workers. Pero estoy limitada en almacenamiento, por lo que se usará sólo uno.

Una breve explicación del procedimiento recomendado:

Clúster compuesto por un servidor que hace de máster y mínimo dos servidores con funciones de worker y un servidor adicional con funciones de almacenamiento persistente.

Después habría que configurar el clúster con las máquinas que lo forman, para que se vean entre ellos. Para esto se suele usar un token.

En mi caso, como no estoy usando máquinas virtuales para configurarlas como nodos, los pods correrán sobre el mismo nodo. Para configurar el clúster de WildFly usaremos dos ficheros yaml diferentes, uno para cada nodo WildFly que conforme el clúster.

Estos ficheros definen los pods.

Un archivo YAML se utiliza para configurar y definir los recursos de Kubernetes. Está compuesto por pares de clave-valor, esto hace que los archivos YAML sean fáciles de leer.

#### 4.5 Arquitectura de WildFly

Hemos de decir, que además de la infraestructura de kubernetes, un clúster de WildFly también consta de varios nodos.



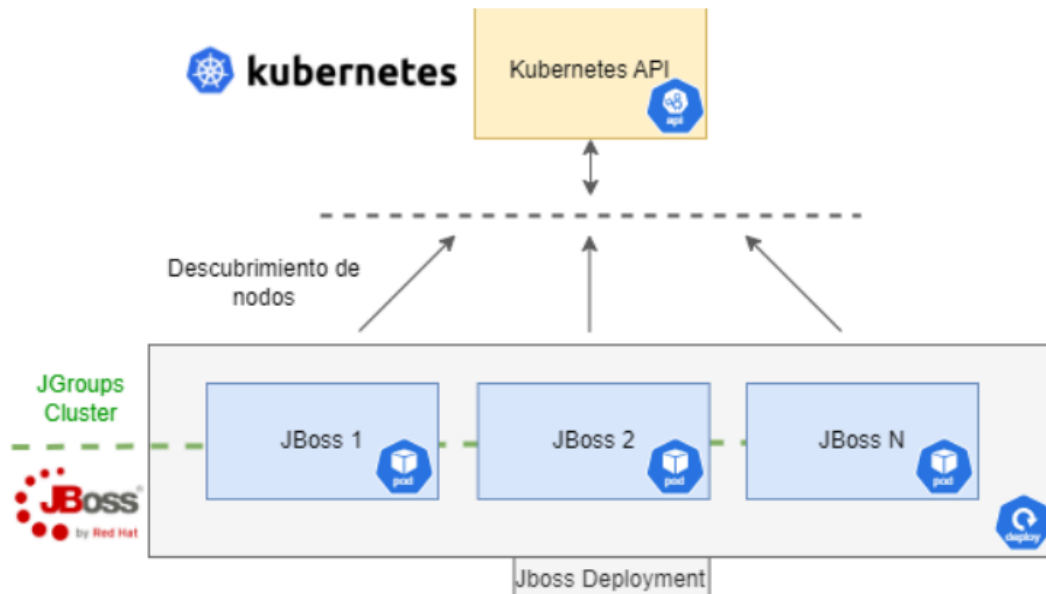


Figura 14. Arquitectura a utilizar

WildFly tiene un nodo máster o Domain Controller y uno o varios nodos llamados Host Controller.

#### 4.6 Software necesario para Jboss

Para la configuración de Jboss se necesita:

- Paquete JBoss, es el conjunto de archivos y configuraciones necesarios para ejecutar el servidor de aplicaciones. La imagen de Docker para JBoss Enterprise Application Platform (EAP) 7.2, por ejemplo, se encuentra en el catálogo de contenedores de Red Hat. Está disponible en el Portal del cliente de Red Hat. Pero necesitamos tener suscripción a Red Hat. En su lugar usaremos Wildfly, que es el proyecto de código abierto sobre el que se basa JBoss EAP.
  - Java JDK, JBoss requiere Java Development Kit (JDK) para ejecutarse.
  - Driver jdbc para la conexión con la base de datos, se necesita el controlador JDBC adecuado para la base de datos específica.
  - Base de datos SQL u Oracle
- Todo esto está incluido en la imagen de Jboss que vamos a usar para montar nuestro contenedor. Una imagen es un binario que incluye todos los requisitos para ejecutar un único contenedor.
- Configurar las variables de entorno: `JBOSS_HOME` Y `JAVA_HOME`

#### **4.6.1 Diferencia entre Jboss y WildFly**

**JBoss:** JBoss Application Server fue creado por JBoss, Inc., que luego fue adquirida por Red Hat en 2006. JBoss AS fue la versión comercial del servidor de aplicaciones, mientras que JBoss Community, también conocido como JBoss AS Community, era la versión de código abierto.

**WildFly:** WildFly es el sucesor de JBoss AS Community. Fue renombrado en la versión 8 para evitar confusiones con la marca registrada "JBoss", ya que Red Hat deseaba mantener la marca para sus productos empresariales. WildFly sigue siendo un servidor de aplicaciones de código abierto y es desarrollado por Red Hat.

En términos de funcionalidad y características, ambos servidores de aplicaciones son bastante similares, ya que WildFly es esencialmente la continuación del desarrollo de JBoss AS Community. Ambos son compatibles con las especificaciones Java EE y ofrecen una amplia gama de funcionalidades para el desarrollo y la implementación de aplicaciones empresariales Java. Sin embargo, WildFly tiende a tener actualizaciones más frecuentes y puede incluir nuevas características y mejoras en comparación con las versiones anteriores de JBoss AS Community.

#### **4.7 Configuración instancias WildFly desde YAML**

Antes de nada, hemos creado un namespace llamado `jbossns`, donde vamos a definir todos los deployments para nuestro proyecto. Los despliegues los haremos desde ficheros `yaml`.

El comando para ello es: `kubectl create namespace jbossns`

En el siguiente archivo `yaml` se define un objeto `Deployment`, con los detalles de la instancia, incluyendo la imagen del contenedor que vamos a usar, la cantidad de réplicas y los pods que pertenecen a este despliegue. Empezaremos configurando una instancia de WildFly usando un fichero `yaml` con la siguiente configuración:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jboss-instance1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jboss-instance1
  template:
    metadata:
      labels:
        app: jboss-instance1
    spec:
      serviceAccountName: jboss-serviceaccount
      containers:
      - name: jboss-container
        image: quay.io/wildfly/wildfly:27.0.0.Final-jdk11
        ports:
        - name: jgroups
          protocol: TCP
          containerPort: 7600
        - name: http-port
          protocol: TCP
          containerPort: 8080
        - name: admin-port
          protocol: TCP
          containerPort: 9990
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: jboss1-volume
          mountPath: /app/data
      volumes:
      - name: jboss1-volume
        persistentVolumeClaim:
          claimName: jboss1-pvc

```

---

Figura 15. Deployment jboss-instance1

Donde jboss-instance1 será el nodo máster de WildFly. Y para que esta instancia sea accesible desde fuera del contenedor, se expone el puerto 8080 y el 9990 para acceder al panel de administración.

Hemos especificado la imagen que vamos a usar, evitando usar la etiqueta “latest”, esto no es recomendable puesto que sino no controlaremos la versión que se está ejecutando. Para asegurarnos de que el pod siempre usa la misma versión de una imagen de contenedor debemos especificar la versión.

```
[Gema@local Deployments]$ kubectl apply -f jboss1.yaml
deployment.apps/jboss-instance1 created
[Gema@local Deployments]$
[Gema@local Deployments]$
[Gema@local Deployments]$ kubectl get pods -A
NAMESPACE      NAME                                                    READY   STATUS    RESTARTS   AGE
default        jboss-instance1-778d7c66d5-kkxpn                      1/1     Running   0          26s
kube-system    coredns-5dd5756b68-chvlz                              1/1     Running   0          21d
kube-system    etcd-minikube                                          1/1     Running   0          21d
kube-system    kube-apiserver-minikube                                1/1     Running   0          21d
kube-system    kube-controller-manager-minikube                      1/1     Running   0          21d
kube-system    kube-proxy-m48t2                                       1/1     Running   0          21d
kube-system    kube-scheduler-minikube                               1/1     Running   0          21d
kube-system    storage-provisioner                                    1/1     Running   2 (21d ago) 21d
```

Figura 16. Listado pods

Para aplicar la configuración del fichero yaml, se ejecuta “kubectl apply -f jboss1.yaml”, que creará una instancia con la configuración dispuesta en ese yaml.

Podemos ver con “kubectl get pods” que hay un nuevo pod ejecutándose desde hace 26 segundos.

Si dejamos el pod configurado de esta forma, la información que tengamos en el pod será efímera, si queremos guardar los logs, es necesario configurar un almacenamiento persistente.

#### 4.8 Configuración almacenamiento persistente: PV y PVC

Primero se debe definir un Persistent Volume (PV), es el almacenamiento persistente.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: jboss-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /run/media/Gema/Disco_Backup/UNI/TFG/Gema/Deployments/
```

Figura 17. Yaml Persistent Volume

Después de definir el PV, creamos un PVC que solicite el almacenamiento proporcionado por el PV.

La diferencia entre PV y PVC, es que un PV es un recurso de almacenamiento que representa un almacenamiento físico, como un disco duro, un sistema de archivos NFS, almacenamiento en la nube, etc.

Y un PVC es una solicitud de almacenamiento persistente por parte de una aplicación en Kubernetes. Representa la demanda de recursos de almacenamiento.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jboss1-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Figura 18. Yaml Persisten Volume Claim

```
[Gema@local Deployments]$ kubectl apply -f pv.yaml
persistentvolume/jboss-pv created
[Gema@local Deployments]$ kubectl apply -f pvc.yaml
persistentvolumeclaim/jboss1-pvc created
```

Figura 19. Configurar almacenamiento

```
[Gema@local Deployments]$ kubectl get pv
NAME          CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM          STORAGECLASS
EASON  AGE
jboss-pv      5Gi       RW0           Retain          Available
jboss-pv      48s
pvc-bfb49b61-809b-4698-a760-d7a4658128ae  1Gi       RW0           Delete          Bound        default/jboss1-pvc  standard
pvc-bfb49b61-809b-4698-a760-d7a4658128ae  31s
[Gema@local Deployments]$ kubectl get pvc
NAME          STATUS  VOLUME          CAPACITY  ACCESS MODES  STORAGECLASS  AGE
jboss1-pvc    Bound   pvc-bfb49b61-809b-4698-a760-d7a4658128ae  1Gi       RW0           standard      44s
```

Figura 20. Listado discos creados

Cuando ya existe el PVC, lo configuramos para que el pod pueda montarlo como volumen persistente. Para ello modificamos el fichero yaml del deployment de WildFly y aplicaremos los cambios.

La forma de hacerlo es eliminando el deployment creado, jboss-instance1, y volviendo a crearlo.

NOTA: Se ha eliminado el deployment, porque si eliminamos el pod, este se volverá a crear con la misma configuración que había antes y no actualizará los cambios definidos.

```
[Gema@local Deployments]$ kubectl delete deployments.apps jboss-instance1
deployment.apps "jboss-instance1" deleted
[Gema@local Deployments]$ kubectl get pods
No resources found in default namespace.
```

Figura 21. Borrado deployments

El Deployment monta el PersistentVolumeClaim (PVC) jboss1-pvc en el directorio /app/data del contenedor, que es el que hemos definido en mountPath.

```
[Gema@local Deployments]$ kubectl apply -f jboss1.yaml
deployment.apps/jboss-instance1 created
[Gema@local Deployments]$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
jboss-instance1-cd97fdc57-8xr2v    0/1     ContainerCreating   0           2s
[Gema@local Deployments]$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
jboss-instance1-cd97fdc57-8xr2v    1/1     Running             0           32s
```

Figura 22. Modificación Deployments

Ahora podemos entrar al pod creado.

```
[Gema@local Deployments]$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
jboss-instance1-cd97fdc57-8xr2v    1/1     Running             0           5m51s
[Gema@local Deployments]$ kubectl exec -it jboss-instance1-cd97fdc57-8xr2v -- bash
jboss@jboss-instance1-cd97fdc57-8xr2v ~]$
```

Figura 23. Consola bash al pod

Este nodo, tendrá el pvc montado y una instancia de WildFly ejecutándose, puesto que en el fichero yaml que usamos para crear este pod, indicamos que para generar este pod, usamos una imagen de WildFly completamente configurado.

```
jboss@jboss-instance1-cd97fdc57-8xr2v ~]$ df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay         98G   54G   44G  56% /
tmpfs           64M    0   64M   0% /dev
/dev/sda1       98G   54G   44G  56% /app/data
shm            64M    0   64M   0% /dev/shm
tmpfs          12G   12K   12G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs          5.8G    0   5.8G   0% /proc/asound
tmpfs          5.8G    0   5.8G   0% /proc/acpi
tmpfs          5.8G    0   5.8G   0% /proc/scsi
tmpfs          5.8G    0   5.8G   0% /sys/firmware
jboss@jboss-instance1-cd97fdc57-8xr2v ~]$ ps -ef | grep java
jboss      95      1  5 22:17 ?        00:00:26 /usr/lib/jvm/java/bin/java -D[Standalone] -server -Xms64m -Xmx512m -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m -Djava.net.preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byteman -Djava.awt.headless=true --add-exports=java.desktop/sun.awt=ALL-UNNAMED --add-exports=java.naming/com.sun.jndi.ldap=ALL-UNNAMED --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=java.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.security=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --add-opens=java.base/java.util.concurrent=ALL-UNNAMED --add-opens=java.management/javax.management=ALL-UNNAMED --add-opens=java.naming/javax.naming=ALL-UNNAMED -Dorg.jboss.boot.log.file=/opt/jboss/wildfly/standalone/log/server.log -Dlogging.configuration=file:/opt/jboss/wildfly/standalone/configuration/logging.properties -jar /opt/jboss/wildfly/jboss-modules.jar -mp /opt/jboss/wildfly/modules org.jboss.as.standalone -Djboss.home.dir=/opt/jboss/wildfly -Djboss.server.base.dir=/opt/jboss/wildfly/standalone -b 0.0.0.0
jboss     275    255  0 22:25 pts/0    00:00:00 grep --color=auto java
```

Figura 24. Servicio Wildfly ejecutándose en pod

## 4.9 Creación de usuarios para los nodos que conforman el clúster

Vamos a necesitar crear un usuario para acceder a la consola de administración de WildFly. WildFly tiene un script para crear usuarios, haremos uso de él.

- Creamos el usuario administrador con la herramienta: `JBOSS_HOME/bin/add-user.sh`

```
[jboss@jboss-instance1-cd97fdc57-d5dbj data]# /opt/jboss/wildfly/bin/add-user.sh
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a):
Enter the details of the new user to add.
Using realm 'ManagementRealm' as discovered from the existing property files.
Username : administrador
Password recommendations are listed below. To modify these restrictions edit the add-user.properties configuration file.
- The password should be different from the username
- The password should not be one of the following restricted values {root, admin, administrator}
- The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
Password :
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[ ]:
About to add user 'administrador' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'administrador' to file '/opt/jboss/wildfly/standalone/configuration/mgmt-users.properties'
Added user 'administrador' to file '/opt/jboss/wildfly/domain/configuration/mgmt-users.properties'
Added user 'administrador' with groups to file '/opt/jboss/wildfly/standalone/configuration/mgmt-groups.properties'
Added user 'administrador' with groups to file '/opt/jboss/wildfly/domain/configuration/mgmt-groups.properties'
Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server Jakarta Enterprise Beans calls.
yes/no? yes
To represent the user add the following to the server-identities definition <secret value="SmJvc3NAMDE=" />
```

Figura 25. Creación usuario administrador en WildFly

Los datos configurados son:

usuario: administrador

contraseña : Jboss@01

`<secret value="SmJvc3NAMDE=" />`

- Vamos a necesitar también crear un usuario, no administrador, para cada host controller, los hosts que actúan de workers en el clúster de WildFly. Este usuario lo usará cada nodo para autenticarse en el clúster. Como vamos a usar dos nodos, necesitaremos dos usuarios, a los que llamaremos hc1 y hc2, por ejemplo.

Crear los usuarios necesarios para que los hosts puedan conectarse con el domain controller (hc1, hc2):

```

What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a):
Enter the details of the new user to add.
Using realm 'ManagementRealm' as discovered from the existing property files.
Username : hc1
Password recommendations are listed below. To modify these restrictions edit the add-user.properties configuration file.
- The password should be different from the username
- The password should not be one of the following restricted values (root, admin, administrator)
- The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
Password :
WFLYDM0099: Password should have at least 8 characters!
Are you sure you want to use the password entered yes/no? yes
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[ ]:
About to add user 'hc1' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'hc1' to file '/opt/jboss/wildfly/standalone/configuration/mgmt-users.properties'
Added user 'hc1' to file '/opt/jboss/wildfly/domain/configuration/mgmt-users.properties'
Added user 'hc1' with groups to file '/opt/jboss/wildfly/standalone/configuration/mgmt-groups.properties'
Added user 'hc1' with groups to file '/opt/jboss/wildfly/domain/configuration/mgmt-groups.properties'
Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server Jakarta Enterprise Beans calls.
yes/no? yes
To represent the user add the following to the server-identities definition <secret value="aGMxQDax" />

```

Figura 26. Creación usuario host en WildFly

Los datos configurados son:

usuario: hc1

contraseña : hc1@01

<secret value="aGMxQDax" />

Los datos configurados son:

usuario: hc2

contraseña : hc2@01

<secret value="aGMyQDax" />

#### 4.10 Configuración de Service en Kubernetes

Para poder acceder al WildFly que corre en este contenedor, debemos exponer el puerto 8080 para que sea accesible desde fuera, esto se puede hacer creando una ruta <<route>>

Para crear una ruta, también se puede utilizar un recurso de tipo Service con un tipo de servicio NodePort.



```
[Gema@local Deployments]$ kubectl apply -f service.yaml
service/jboss-service created
[Gema@local Deployments]$ cat service.yaml
apiVersion: v1
kind: Service
metadata:
  name: jboss-service
spec:
  selector:
    app: jboss-instance1 # Etiqueta del pod al que apuntará el servicio
  ports:
    - name: app-port
      protocol: TCP
      port: 8080 # Puerto del servicio para la aplicación
      targetPort: 8080 # Puerto en el contenedor para la aplicación
    - name: admin-port
      protocol: TCP
      port: 9990 # Puerto del servicio para la administración de JBoss
      targetPort: 9990 # Puerto en el contenedor para la administración de JBoss
  type: NodePort # Tipo de servicio NodePort
```

Figura 27. Creación servicio jboss-instance1

Configuramos dos puertos, el de la aplicación, 8080, y el puerto que WildFly levanta para acceder a la consola de administración, 9990. Esta configuración está definida en el fichero standalone.xml

Creamos el servicio:

```
[Gema@local Deployments]$ kubectl apply -f service.yaml
service/jboss-service created
[Gema@local Deployments]$ kubectl get services
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
jboss-service       NodePort    10.109.254.65   <none>           8080:32725/TCP   24s
```

Figura 28. Aplicar configuración servicio jboss-instance1

Para acceder al contenedor JBoss a través del servicio NodePort, necesitamos la dirección IP del nodo del clúster Kubernetes y el puerto asignado por NodePort.

```
[Gema@local Deployments]$ kubectl get nodes -o wide
NAME                STATUS    ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
minikube            Ready    control-plane  23d   v1.28.3   192.168.49.2   <none>        Ubuntu 22.04.3 LTS   6.8.4-200.fc39.x86_64 docker://24.0.7
```

Figura 29. Comprobar IP del nodo

Abrimos un navegador:



- Es más fácil de configurar y administrar, ya que cada instancia es independiente y no hay necesidad de sincronización con otras instancias.

Modo Domain:

- Varias instancias de WildFly se ejecutan como miembros de un dominio centralizado y están coordinadas por un controlador de dominio.
- El controlador de dominio es responsable de la administración centralizada de las configuraciones y despliegues de las instancias de WildFly en el dominio.
- Las configuraciones comunes, como los perfiles de seguridad, los despliegues compartidos y la configuración de red, se pueden gestionar de forma centralizada desde el controlador de dominio.
- Es adecuado para entornos donde se requiere la administración centralizada y la coordinación de múltiples instancias de WildFly para aplicaciones distribuidas o de gran escala.

La solución podría ser descargar una imagen de WildFly diferente, usar “image: [registry.redhat.io/jboss-eap-7/eap72](https://registry.redhat.io/jboss-eap-7/eap72)” pero para descargar esta imagen necesitamos estar registrados en redhat.

Otra opción, que será la que usemos, es editar la imagen WildFly descargada, utilizando un Dockerfile para crear una nueva configuración personalizada con las modificaciones que necesitamos y de esa forma generar una imagen personalizada de WildFly.

#### **4.12 Dockerfile para WildFly en modo standalone-full-ha.xml**

Vamos a probar primero la opción de utilizar el archivo [standalone-full-ha.xml](#) en lugar de arrancar en modo domain. El archivo [standalone-full-ha.xml](#) proporciona una configuración completa para habilitar alta disponibilidad (HA) en un entorno de instancia única.

Para ello, vamos a usar la imagen que ya teníamos previamente, [quay.io/wildfly/wildfly:27.0.0.Final-jdk11](https://quay.io/wildfly/wildfly:27.0.0.Final-jdk11), pero modificada.

Usaremos este fichero Dockerfile, que es un archivo de texto utilizado para definir las instrucciones necesarias para construir una imagen de Docker. El Dockerfile describe paso a paso cómo se debe crear esta imagen.

```

1 # La imagen base de Wildfly 27 con JDK 11
2 FROM quay.io/wildfly/wildfly:27.0.0.Final-jdk11
3
4 # Los puertos necesarios para acceder al contenedor
5 EXPOSE 8080 9990 7600
6
7
8 COPY standalone-full-ha.xml /opt/jboss/wildfly/standalone/configuration
9 COPY mgmt-users.properties /opt/jboss/wildfly/standalone/configuration/mgmt-users.properties
10 COPY helloworld.war /opt/jboss/wildfly/standalone/deployments
11
12 CMD ["/bin/bash", "/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0", "-bmanagement", "0.0.0.0", "-c", "standalone-full-ha.xml"]
13
14

```

Figura 31. Dockerfile WildFly standalone

En el Dockerfile, vemos que hay varias instrucciones, entre ellas la instrucción “COPY” que sirve para copiar ficheros locales al contenedor.

Hemos descargado previamente, los ficheros standalone-full-ha.xml y mgmt-users.properties de la imagen anterior, para modificarlos localmente y después usarlos en nuestra imagen personalizada.

Descargar ficheros del contenedor a local:

```
kubectl cp jboss-instance1-cd97fdc57-wjtjj:/app/data/standalone-full-ha.xml
./standalone-full-ha.xml
```

```
[Gema@local jbossns]$ kubectl cp jboss-instance1-cd97fdc57-wjtjj:/app/data/standalone-full-ha.xml ./standalone-full-ha.xml
```

Figura 32. Copia ficheros desde el contenedor a local

El fichero mgmt-users.properties contiene los usuarios creados en WildFly. Hemos creado para este caso un usuario administrador. (Esta parte se detalla en el punto 4.9)

```
administrador=dfc7d6ea250e433ce816dae3e16f3fe7
```

También hemos descargado una aplicación de prueba, un helloworld, para desplegar en WildFly. La subimos al directorio correspondiente de la imagen que queremos crear, con el comando COPY.

La última parte, CMD:

```
CMD ["/bin/bash", "/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0", "-bmanagement", "0.0.0.0", "-c", "standalone-full-ha.xml"]
```

Figura 33. Detalle comando CMD en Dockerfile

Indica el comando que queremos ejecutar cuando se inicie el contenedor. Se especifica que debe hacerse en una shell Bash, seguido del comando principal que se ejecutará al iniciar el contenedor, y los elementos restantes son las opciones que queremos especificar.

"-b 0.0.0.0":

Indica que JBoss/Wildfly debería escuchar en todas las interfaces de red (0.0.0.0). Esto permite que el servidor sea accesible desde fuera del contenedor.

"-bmanagement 0.0.0.0":

Indica que la consola de administración también debería escuchar en todas las interfaces. Esto permite el acceso a la consola de administración desde fuera del contenedor.

"-c standalone-full-ha.xml":

Indica que JBoss/Wildfly debería usar este archivo de configuración específico.

#### 4.13 Construir nueva imagen

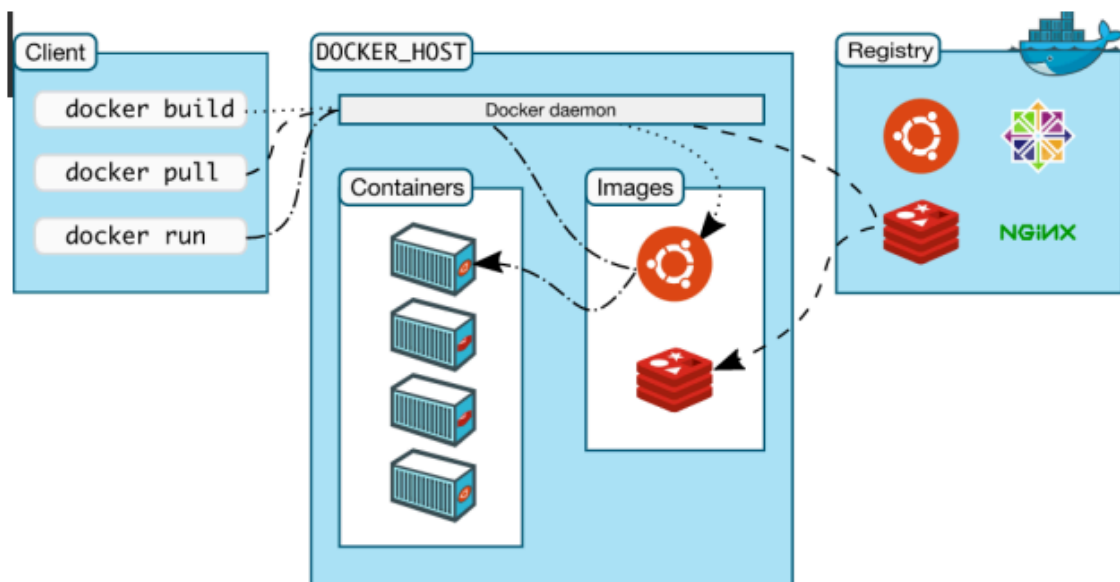


Figura 34. Esquema imagen contenedores

Los pasos para crear ahora la nueva imagen usando el fichero Dockerfile son los siguientes:

Primero, debemos situarnos en el mismo directorio donde esté ubicado el fichero Dockerfile, así como los distintos ficheros que queremos usar para generar la imagen.

Ejecutamos la orden docker build, para crearla.

```
[Gema@local jbossns]$ docker build -t lavic/wildfly-personalizada .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
Install the buildx component to build images with BuildKit:
https://docs.docker.com/go/buildx/

Sending build context to Docker daemon 336.9kB
Step 1/6 : FROM quay.io/wildfly/wildfly:27.0.0.Final-jdk11
--> 714ae6f8cab5
Step 2/6 : EXPOSE 8080 9990 7600
--> Using cache
--> 8a328498bf23
Step 3/6 : COPY standalone-full-ha.xml /opt/jboss/wildfly/standalone/configuration
--> Using cache
--> d03858d8a62a
Step 4/6 : COPY mgmt-users.properties /opt/jboss/wildfly/standalone/configuration/mgmt-users.properties
--> Using cache
--> e0784d6ea524
Step 5/6 : COPY helloworld.war /opt/jboss/wildfly/standalone/deployments
--> Using cache
--> 55d673830d1f
Step 6/6 : CMD ["/bin/bash", "/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0", "-bmanagement", "0.0.0.0", "-c", "standalone-full-ha.xml"]
--> Running in 3c4409785fba
Removing intermediate container 3c4409785fba
--> cde9794fb819
Successfully built cde9794fb819
Successfully tagged lavic/wildfly-personalizada:latest
[Gema@local jbossns]$
[Gema@local jbossns]$
[Gema@local jbossns]$
[Gema@local jbossns]$ docker push lavic/wildfly-personalizada
Using default tag: latest
The push refers to repository [docker.io/lavic/wildfly-personalizada]
5c276d78fdc6: Layer already exists
7a8173a0fff4: Layer already exists
8843884a25e5: Layer already exists
f430b215e001: Layer already exists
31f865f5be5da: Layer already exists
732e148cac5c: Layer already exists
7c7e3466a282: Layer already exists
265bf44e0b41: Layer already exists
9269874c166b: Layer already exists
174f56854903: Layer already exists
latest: digest: sha256:01e533cccf7f8ef4fc9aae548987c5d0124918e6b18bafbc8124720468ff4235 size: 2414
```

Figura 35. Creación imagen WildFly

#### 4.13.1 Docker Hub

Hemos subido la imagen a nuestro repositorio de Docker Hub. Para ello nos logamos desde la consola, ya teníamos cuenta registrada, sino sería necesario crearla.

```
[Gema@local jbossns]$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: lavic
Password:
WARNING! Your password will be stored unencrypted in /home/gema/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

Figura 36. Acceso a Docker Hub

Subimos la imagen:

```
[Gema@local jbossns]$ docker push lavic/wildfly-personalizada
Using default tag: latest
The push refers to repository [docker.io/lavic/wildfly-personalizada]
5c276d78fdc6: Layer already exists
7a8173a0fff4: Layer already exists
8843884a25e5: Layer already exists
f430b215e001: Layer already exists
31f865f5be5da: Layer already exists
732e148cac5c: Layer already exists
7c7e3466a282: Layer already exists
265bf44e0b41: Layer already exists
9269874c166b: Layer already exists
174f56854903: Layer already exists
latest: digest: sha256:01e533cccf7f8ef4fc9aae548987c5d0124918e6b18bafbc8124720468ff4235 size: 2414
```

Figura 37. Subida a Docker Hub de imagen

### 4.13.2 Arrancar contenedor local

Para probar nuestro contenedor local lanzamos el siguiente comando:

```
[Gema@local jbossns]$ docker run -it -p 8080:8080 -p 9990:9990 lavic/wildfly-personalizada /bin/bash
```

Una vez hemos comprobado que esta imagen nos sirve, que ejecuta WildFly en modo standalone y podemos acceder desde nuestro PC a la consola de administración ( puerto 9090), vamos a usarla en kubernetes.

### 4.13.3 Arrancar imagen en kubernetes

Para ello, modificamos el fichero .YAML que habíamos usado para crear el deployment modificando la imagen a utilizar y cambiando el número de réplicas a usar:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: jboss-instance1
5 spec:
6   replicas: 2
7   selector:
8     matchLabels:
9       app: jboss-instance1
10  template:
11    metadata:
12      labels:
13        app: jboss-instance1
14    spec:
15      serviceAccountName: jboss-serviceaccount
16      containers:
17      - name: jboss-container
18        image: lavic/wildfly-personalizada
19        ports:
20        - name: jgroups
21          protocol: TCP
22          containerPort: 7600
23        - name: http-port
24          protocol: TCP
25          containerPort: 8080
26        - name: admin-port
27          protocol: TCP
28          containerPort: 9990
29        env:
30        - name: POD_NAMESPACE
31          valueFrom:
32            fieldRef:
33              fieldPath: metadata.namespace
34        volumeMounts:
35        - name: config-volume
36          mountPath: /app/data
```

Figura 38. Yaml Deployment jboss-instance1 con imagen personalizada

Lanzamos el deployment: `kubectl apply -f jboss1_v4.yaml`

Vemos que ha creado 2 pods y que el deployment se ha generado

```
[Gema@local jbossns]$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
jboss-instance1-6845f45b9d-8tm25    1/1    Running   0          133m
jboss-instance1-6845f45b9d-c4d9f    1/1    Running   0          97m
[Gema@local jbossns]$ kubectl get deployments.apps
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
jboss-instance1 2/2     2            2           133m
```

Figura 39. Listado de pods con dos réplicas

Comprobamos que WildFly es accesible desde fuera y , para ello, usamos la IP que usa nuestro contenedor y el puerto asignado por el servicio (más info en el punto 4.13) :

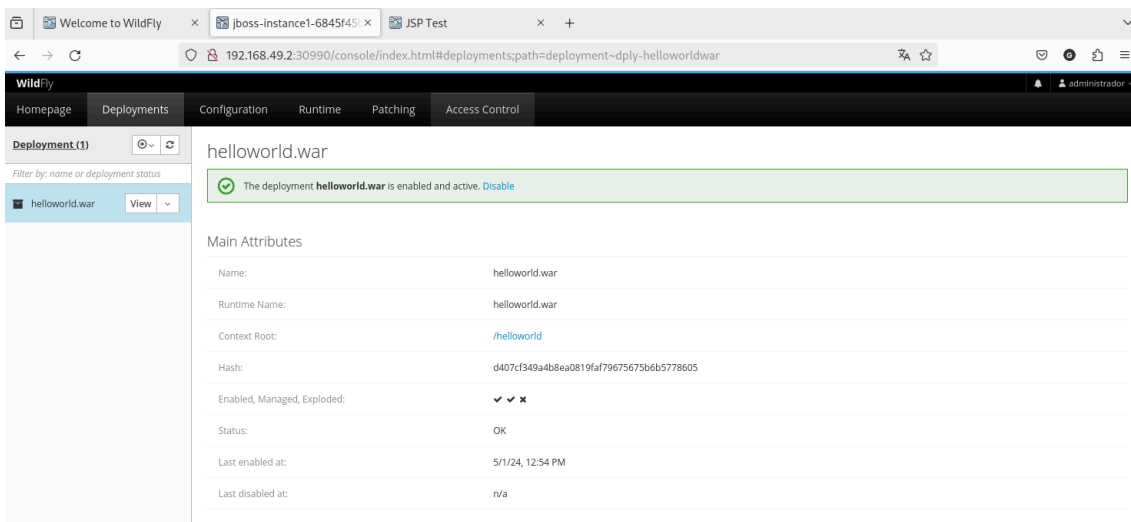


Figura 40. Consola administración Wildfly

El YAML del servicio para acceder al contenedor es este:

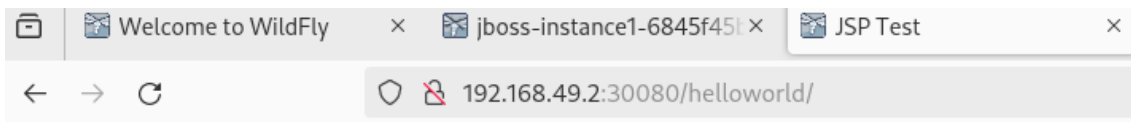


```
[Gema@local jbossns]$ cat jboss1-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: jboss-service
  namespace: jbossns
spec:
  type: NodePort # Tipo de servicio NodePort para exponer puertos fuera del clúster
  selector:
    app: jboss-instance1
  ports:
    - name: http # Nombre del primer puerto
      protocol: TCP
      port: 80 # Puerto que será accesible desde fuera
      targetPort: 8080 # Puerto dentro del contenedor
      nodePort: 30080 # Puerto NodePort para acceder al nodo
    - name: admin # Nombre del segundo puerto
      protocol: TCP
      port: 9990
      targetPort: 9990
      nodePort: 30990 # Puerto NodePort para acceso externo
    - name: jgroups # Nombre del tercer puerto
      protocol: TCP
      port: 7600
      targetPort: 7600
```

Figura 41. Yaml servicio acceso jboss-instance1

#### 4.13.4 Hello World en WildFly standalone

La aplicación de pruebas es esta:



## Hello World

If you see this, the example war-file was correctly deployed! Congrats!

Wed May 01 11:30:17 UTC 2024

You are from 10.244.0.1



Figura 42. Acceso a aplicación de prueba “Hello World”

Esta aplicación será accesible siempre que alguno de los pods que levanta el deployment esté en estado running. Podemos escalar el número de pods según nuestras necesidades, y kubernetes podrá balancear las peticiones a los distintos pods.

Para aumentar el número de pods de este deployment:

```
kubectl scale deployment/jboss-instance1 --replicas=4
```

```
[Gema@local jbossns]$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
jboss-instance1-6845f45b9d-8tm25    1/1     Running   0           25m
jboss-instance1-6845f45b9d-chglp    1/1     Running   0           19m
jboss-instance1-6845f45b9d-ddtq5    1/1     Running   0           13m
jboss-instance1-6845f45b9d-znvjr    1/1     Running   0            4s
```

Figura 43. Listado pods aumento de réplicas a 4

## 4.15 Dockerfile Domain

La idea inicial era usar el modo Domain, donde habría un nodo máster que centralice y coordine nuestro clúster, así que vamos a ello.

Configurar el nodo master.

Los ficheros de configuración que necesitamos modificar son dos, nos los descargamos a local desde el pod que teníamos previamente ejecutándose en el deployment "jboss-instance1". Vamos a configurarlos para que este nodo realice la función de máster.

- Descargamos los ficheros host-master.xml y domain.xml, que están en:

```
/opt/jboss/wildfly/domain/configuration/host-master.xml  
/opt/jboss/wildfly/domain/configuration/domain.xml
```

- En en caso de servidores físicos con Ip's estáticas, debemos modificar la IP del interface de management y el nombre del host en el host-master.xml:

Por ejemplo:

Cambiamos:

```
<domain-controller>  
  <local/>  
</domain-controller>  
<interfaces>  
  <interface name="management">  
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>  
  </interface>  
</interfaces>  
<jvms>
```

Figura 44. Configuración por defecto domain controller

Por:

```
<domain-controller>  
  <local/>  
</domain-controller>  
<interfaces>  
  <interface name="management">  
    <inet-address value="${jboss.bind.address.management:10.244.0.7}"/>  
  </interface>  
</interfaces>
```

Figura 45. Configuración domain controller

- Si quisiéramos configurar los perfiles y los server-groups, tendríamos que hacerlo en el fichero domain.xml.
- En un entorno de producción, tenemos que configurar la conexión a la bbdd. Esto se hace en el fichero domain.xml. En nuestro caso nos sirve con la configuración por defecto.

```

<datasources>
  <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" use-java-context="true"
  statistics-enabled="${wildfly.datasources.statistics-enabled:${wildfly.statistics-enabled:false}}">
    <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
    <driver>h2</driver>
    <security>
      <user-name>sa</user-name>
      <password>sa</password>
    </security>
  </datasource>

```

Figura 46. Configuración por defecto de conexión a base de datos

Pero en Kubernetes, asignar direcciones IP estáticas directamente a un Deployment no es una práctica común ni recomendada, principalmente porque los Pods en un Deployment son efímeros y pueden ser recreados o movidos a diferentes nodos según las necesidades de escalabilidad y resiliencia del clúster.

Kubernetes está diseñado para ser escalable y resiliente. La asignación de direcciones IP estáticas podría interferir con esta flexibilidad. Por lo tanto no vamos a modificar la IP.

#### 4.15.1 Domain.xml

Los cambios que vamos a realizar sobre el fichero domain.xml son nulos, porque no vamos a cambiar los <server-groups> ni los <profiles>, dejamos los que aparecen por defecto:

```

<profile name="default">
<profile name="ha">
<profile name="full">
<profile name="full-ha">
<profile name="load-balancer">

```

```

<server-group name="main-server-group" profile="full">
<server-group name="other-server-group" profile="full-ha">

```

<profile name="default">: Define un perfil de configuración predeterminado para el entorno de la aplicación. Este perfil tiene configuraciones estándar.

<profile name="ha">: Un perfil de alta disponibilidad que incluye configuraciones específicas para garantizar la redundancia y la resistencia ante fallos.

<profile name="full">: Un perfil que incluye todas las características disponibles en el entorno de la aplicación.

<profile name="full-ha">: Un perfil que combina las características de alta disponibilidad y la funcionalidad completa.

<profile name="load-balancer">: Un perfil que contiene configuraciones relacionadas con el equilibrio de carga para distribuir las solicitudes entre los servidores.

<server-group name="main-server-group" profile="full">: Define un grupo de servidores llamado "main-server-group" que utiliza el perfil "full". Esto implica que los servidores en este grupo se configuran con todas las características disponibles en el perfil "full".

<server-group name="other-server-group" profile="full-ha">: Define otro grupo de servidores llamado "other-server-group" que utiliza el perfil "full-ha". Esto significa que los servidores en este grupo se configuran con las características de alta disponibilidad y la funcionalidad completa del perfil "full-ha".

#### 4.15.2 Host.xml

En este fichero sí que vamos a realizar cambios

- Definimos el nombre que tendrá este nodo:

```
<?xml version="1.0" ?>

<host xmlns="urn:jboss:domain:20.0" name="master">
  <extensions>
    <extension module="org.jboss.as.jmx"/>
    <extension module="org.wildfly.extension.core-management"/>
    <extension module="org.wildfly.extension.elytron"/>
  </extensions>
  <management>
    <audit-log>
      <formatters>
        <json-formatter name="json-formatter"/>
      </formatters>
    </audit-log>
  </management>
</host>
```

Figura 47. Configuración nombre de nodo máster en host.xml

- Modificamos las interfaces

La configuración predeterminada para este fichero es la siguiente:

```

<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
  <interface name="unsecured">
    <inet-address value="127.0.0.1" />
  </interface>
</interfaces>

```

Figura 48. Configuración por defecto de interfaces en host.xml

Necesitamos cambiar la dirección de la interfaz de administración para que desde otros nodos, el controlador de host, pueda conectarse al controlador de host principal, que es el que estamos configurando desde el fichero host.xml del nodo que hará de máster.

La interfaz pública permite acceder a la aplicación mediante HTTP no local y la interfaz no segura permite el acceso RMI remoto.

Hemos configurado la IP del servicio jboss-domain que habíamos configurado previamente.

Nombre	Etiquetas	Tipo	IP cluster
jboss-host	-	ClusterIP	10.104.224.207
jboss-domain	-	NodePort	10.102.237.60

Figura 49. Listado de servicios creados en consola de kubernetes

```

<domain-controller>
  <local/>
</domain-controller>
<interfaces>
  <interface name="management">
    <inet-address value="{jboss.bind.address.management:10.102.237.60}"/>
  </interface>
  <interface name="public">
    <inet-address value="{jboss.bind.address:10.102.237.60}"/>
  </interface>
  <interface name="unsecured">
    <inet-address value="10.102.237.60" />
  </interface>
</interfaces>

```

Figura 50. Configuración de interfaces en host.xml

Podríamos haber configurado también el nombre del servicio “jboss-domain” ya que el servidor DNS de Kubernetes lo resuelve.

- Para levantar el servicio de jboss-domain, arrancamos desde el fichero de tipo Deployment siguiente:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jboss-domain
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jboss-domain
  template:
    metadata:
      labels:
        app: jboss-domain
    spec:
      serviceAccountName: jboss-serviceaccount
      containers:
        - name: jboss-container
          image: lavic/wildfly-domain
          ports:
            - name: jgroups
              protocol: TCP
              containerPort: 7600
            - name: http-port
              protocol: TCP
              containerPort: 8080
            - name: admin-port
              protocol: TCP
              containerPort: 9990

```

Figura 51. Yaml Deployment jboss-domain

- El fichero Dockerfile que hemos usado para este deployment es:

```
# La imagen base de Wildfly 27 con JDK 11
FROM quay.io/wildfly/wildfly:27.0.0.Final-jdk11

# Los puertos necesarios para acceder al contenedor
EXPOSE 8080 9990 7600

COPY domain.xml /opt/jboss/wildfly/domain/configuration/
COPY host.xml /opt/jboss/wildfly/domain/configuration/
COPY mgmt-users.properties /opt/jboss/wildfly/domain/configuration/
COPY mgmt-groups.properties /opt/jboss/wildfly/domain/configuration/

CMD ["/bin/bash", "/opt/jboss/wildfly/bin/domain.sh", "-b", "0.0.0.0", "-bmanagement", "0.0.0.0", "--domain-config=domain.xml", "--host-config=host.xml"]
```

Figura 52. Dockerfile usado para crear Domain Controller

- El comando que arranca esta instancia es:

```
/opt/jboss/wildfly/bin/domain.sh -b 0.0.0.0 -bmanagement 0.0.0.0
--domain-config=domain.xml --host-config=host.xml
```

Nótese, que en este caso no estamos configurando ningún despliegue de ejemplo, puesto que WildFly en modo domain no funciona igual. Más adelante se explicará cómo desplegar aplicaciones.

Los pasos que hemos seguido para construir la imagen son los mismos que en modo standalone, pero cambiando el nombre, para distinguir las imágenes que usamos:

```
docker build -t lavic/wildfly-domain .
docker push lavic/wildfly-domain
kubectl apply -f jboss_master.yaml
```



```
[Gema@local master]$ docker build -t lavic/wildfly-domain .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
              Install the buildx component to build images with BuildKit:
              https://docs.docker.com/go/buildx/

Sending build context to Docker daemon 233.5kB
Step 1/6 : FROM quay.io/wildfly/wildfly:27.0.0.Final-jdk11
--> 714ae6f8cab5
Step 2/6 : EXPOSE 8080 9990 7600
--> Using cache
--> 8a328498bf23
Step 3/6 : COPY domain.xml /opt/jboss/wildfly/domain/configuration/
--> Using cache
--> eb970a27eb04
Step 4/6 : COPY host.xml /opt/jboss/wildfly/domain/configuration/
--> Using cache
--> 25a822977f09
Step 5/6 : COPY mgmt-users.properties /opt/jboss/wildfly/domain/configuration/
--> Using cache
--> c84b5f509908
Step 6/6 : CMD ["/bin/bash", "/opt/jboss/wildfly/bin/domain.sh", "--b", "0.0.0.0", "--bmanagement", "0.0.0.0", "--domain-config=domain.xml", "--host-config=host.xml"]
--> Running in 0c8e68a7dd95
Removing intermediate container 0c8e68a7dd95
--> 335d0ad0b43a
Successfully built 335d0ad0b43a
Successfully tagged lavic/wildfly-domain:latest
```

Figura 53. Construcción imagen para Domain Controller

```
[Gema@local master]$ docker push lavic/wildfly-domain
Using default tag: latest
The push refers to repository [docker.io/lavic/wildfly-domain]
63c14c2cc5ef: Layer already exists
3f578978fe38: Layer already exists
eeba7e3ac51d: Layer already exists
f430b215e001: Layer already exists
31f865f8e5da: Layer already exists
732e148cac5c: Layer already exists
7c7e3466a282: Layer already exists
265bf44e0b41: Layer already exists
9269874c166b: Layer already exists
174f56854903: Layer already exists
latest: digest: sha256:e54a79a9a2f1095f686a6568acc312106f7fed284048f46c1b705b30ce41de0c size: 2413
```

Figura 54. Subimos a Docker Hub imagen Domain Controller

```
[Gema@local master]$ kubectl apply -f jboss_master.yaml
deployment.apps/jboss-domain created
```

Figura 55. Lanzamos Deployment jboss\_master

### 4.15.3 domain-service.yaml

Para que este WildFly sea accesible desde fuera, debemos configurar un servicio, con puertos diferentes a los ya utilizados en el deployment standalone:

```

apiVersion: v1
kind: Service
metadata:
  name: jboss-domain
  namespace: jbossns
spec:
  type: NodePort # Tipo de servicio NodePort
  selector:
    app: jboss-domain
  ports:
    - name: http-8080
      protocol: TCP
      port: 80 # Puerto que será accesible desde fuera
      targetPort: 8080 # Puerto dentro del contenedor
      nodePort: 30002 # Puerto NodePort para acceder al nodo
    - name: http-8330 # Nombre del puerto 8330
      protocol: TCP
      port: 8330
      targetPort: 8330
      nodePort: 30003 # Puerto NodePort para acceder desde fuera
    - name: http-8430 # Nombre del puerto 8430
      protocol: TCP
      port: 8430
      targetPort: 8430
      nodePort: 30004 # Puerto NodePort para acceder desde fuera
    - name: admin # Nombre puerto administración
      protocol: TCP
      port: 9990
      targetPort: 9990 #Puerto dentro del contenedor
      nodePort: 30001 # Puerto NodePort para acceso externo
    - name: jgroups
      protocol: TCP
      port: 7600
      targetPort: 7600
    - name: conect
      protocol: TCP
      port: 9999
      targetPort: 9999

```

Figura 56. Configuración servicio Domain Controller

#### 4.16 Acceso consola entorno gráfico

Comprobamos que es accesible desde fuera, esto lo haremos con la IP asignada automáticamente por minikube y el puerto en el que el servicio específico dentro de Kubernetes está expuesto, que sería el puerto 9990, que exponemos desde su servicio como puerto 30001.

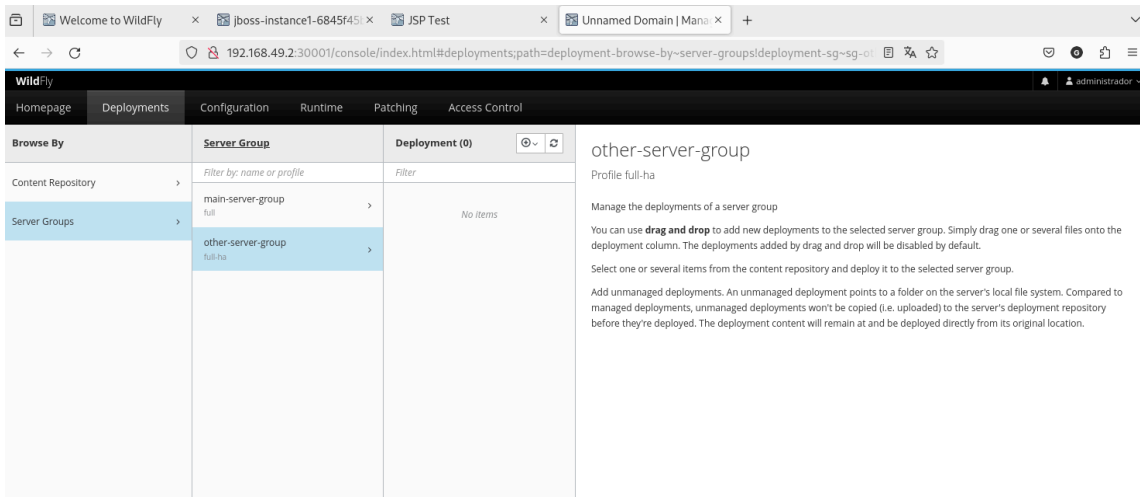


Figura 57. Consola gráfica WildFly con Domain Controller

#### 4.17 Hello World en WildFly Domain

Desplegamos la aplicación de pruebas de forma manual, en uno de los dos servidores que vienen configurados por defecto, en nuestro caso, hemos elegido el “full-ha”

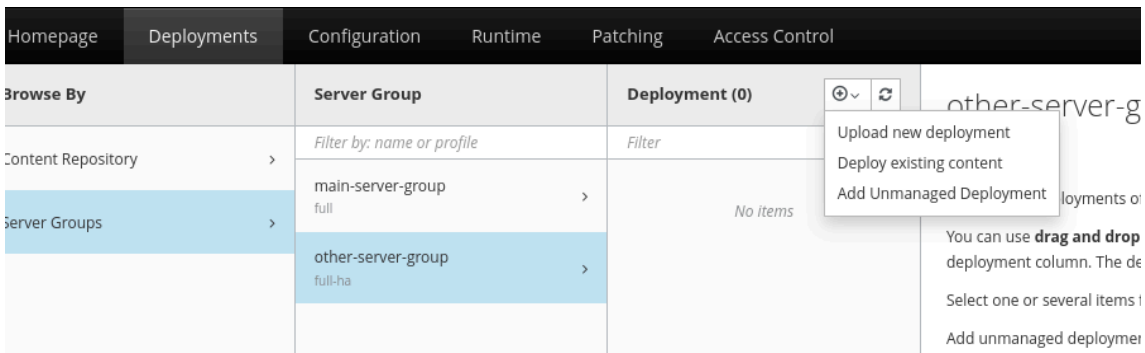


Figura 58. Despliegue aplicación pruebas en consola de administración

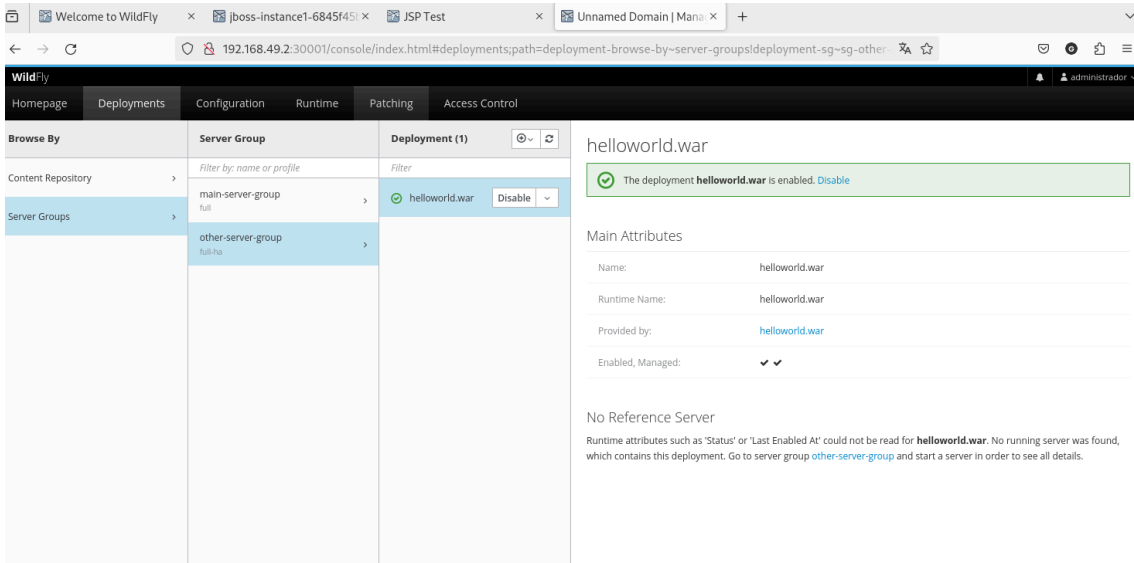


Figura 59. Despliegue correcto de helloworld

Si vamos a la pestaña de Runtime, veremos los nodos que forman parte de este clúster. En este caso sólo uno, el nodo master.

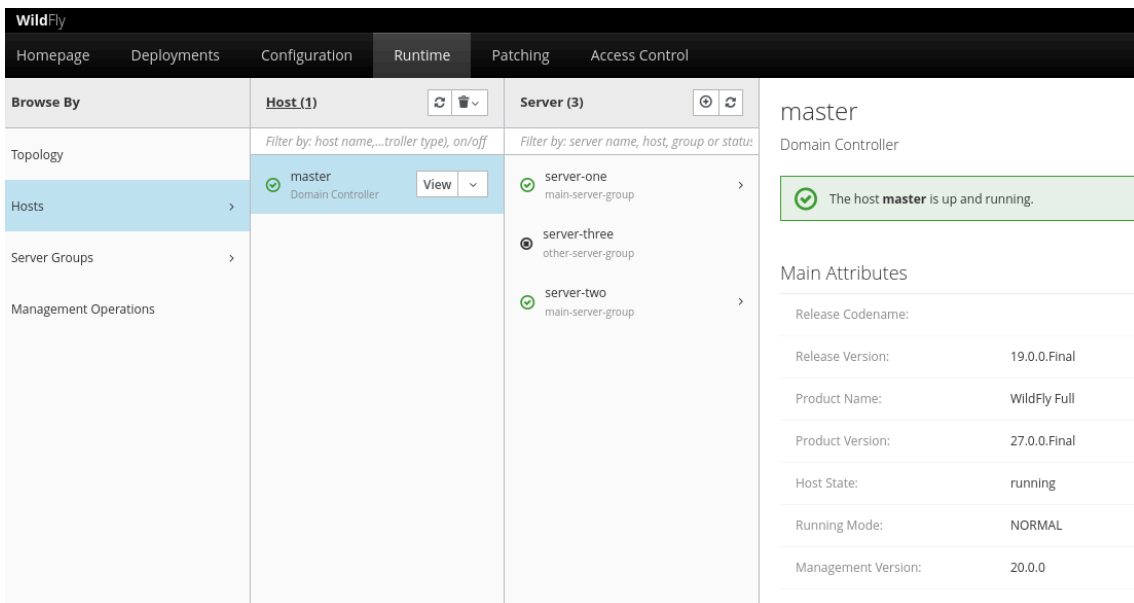


Figura 60. Nodo máster en consola de administración

Comprobamos que la aplicación también es accesible desde fuera, usando los puertos configurados en el servicio para jboss en modo domain, esto es, el configurado en el fichero domain-service.yaml.



## Hello World

If you see this, the example war-file was correctly deployed! Congrats!

Sun May 19 18:55:02 UTC 2024

You are from 10.244.0.1



Figura 61. Acceso a aplicación de prueba desde servicio

### 4.18 Dockerfile Host

Vamos a configurar un segundo nodo, un host controller, que se conectará al nodo maestro para así poder ser centralizado desde este.

Usaremos este fichero Dockerfile para su despliegue.

```
# La imagen base de Wildfly 27 con JDK 11
FROM quay.io/wildfly/wildfly:27.0.0.Final-jdk11

# Los puertos necesarios para acceder al contenedor
EXPOSE 8080 9990 7600

COPY host.xml /opt/jboss/wildfly/domain/configuration/

CMD ["/bin/bash", "/opt/jboss/wildfly/bin/domain.sh", "--host-config=host.xml"]
```

Figura 62. Dockerfile Host Controller

### 4.18.1 Host.xml

Para configurar el nodo host, necesitamos modificar un fichero de configuración, host.xml. Nos los descargamos a local desde el pod que teníamos previamente ejecutándose en el deployment “jboss-instance1”.

- Descargamos el fichero host.xml , que está en:

/opt/jboss/wildfly/domain/configuration/host.xml

- Lo editamos, para realizar varios cambios en él.

La configuración que usaremos en el host secundario es un poco diferente al usado en el host del nodo máster, porque debemos permitir que el controlador de host secundario se conecte al controlador de host principal.

Primero necesitamos configurar el nombre de host. Cambiamos la propiedad del nombre, y configuramos “host1” en nuestro caso:

```
<?xml version="1.0" ?>

<host xmlns="urn:jboss:domain:20.0" name="host1">
  <extensions>
    <extension module="org.jboss.as.jmx"/>
    <extension module="org.wildfly.extension.core-management"/>
  </extensions>
</host>
```

Figura 63. Configuración nombre de nodo host1

Debemos añadir la siguiente configuración en el subsistema elytron, que por defecto no está, con el fin de definir la identidad del controlador del host.

- Usuario de este host: hc1
- Contraseña de dicho usuario: hc1@01

```
<profile>
  <subsystem xmlns="urn:jboss:domain:core-management:1.0"/>
  <subsystem xmlns="urn:wildfly:elytron:16.0" final-providers="combined-providers" disallowed-providers="OracleUcrypto">
    <authentication-client>
      <authentication-configuration sasl-mechanism-selector="DIGEST-MD5"
        name="hostAuthConfig"
        authentication-name="hc1"
        realm="ManagementRealm">
        <credential-reference clear-text="hc1@01"/>
      </authentication-configuration>
    </authentication-client>
    <authentication-context name="hcAuthContext">
      <match-rule authentication-configuration="hostAuthConfig"/>
    </authentication-context>
  </subsystem>
</profile>
```

Figura 64. Configuración usuario Wildfly en host1

10.102.237.60 es la dirección IP del servicio jboss-domain, así que a nuestro host tenemos que indicarle que su domain controller no está en local, sino en dicha IP, la cual también responde con el nombre DNS “ jboss-domain”

Esta parte es importante, ya que sino el host secundario no se conectará al clúster.

```
</management>
<domain-controller>
  <remote protocol="http-remoting" host="jboss-domain" port="9990" authentication-context="hcAuthContext"/>
</domain-controller>
<interfaces>
  <interface name="management">
    <inet-address value="{jboss.bind.address.management:0.0.0.0}"/>
  </interface>
  <interface name="public">
    <inet-address value="{jboss.bind.address:0.0.0.0}"/>
  </interface>
  <interface name="unsecured">
    <inet-address value="0.0.0.0" />
  </interface>
</interfaces>
```

Figura 65. Configuración remote Domain Controller en host.xml

#### 4.18.2 jboss\_host.yaml

El fichero que usaremos para el despliegue del Deployment para este nodo será el siguiente:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jboss-host
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jboss-host
  template:
    metadata:
      labels:
        app: jboss-host
    spec:
      serviceAccountName: jboss-serviceaccount
      containers:
      - name: jboss-container
        image: lavic/wildfly-host
        ports:
        - name: jgroups
          protocol: TCP
          containerPort: 7600
        - name: http-port
          protocol: TCP
          containerPort: 8080
        - name: admin-port
          protocol: TCP
          containerPort: 9990
        env:
        - name: JBOSS_DOMAIN_MASTER_ADDRESS
          value: "jboss-domain.jbossns.svc.cluster.local"
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: config-volume
          mountPath: /app/data
      volumes:
      - name: config-volume
      - name: secret-volume
        secret:
          secretName: my-secret

```

Figura 66. Yaml Deployment Host1

Este Deployment, como podemos ver en el fichero yaml, se llama jboss-host. Se ha creado desde la imagen de wildfly-host, que hemos generado y guardado en nuestro gitlab, igual que hicimos con el deployment de jboss-domain.



### 4.18.3 host-service.yaml

Definimos un servicio para este host, igual que hicimos con el domain, para poder acceder a él desde fuera del contenedor.

Hemos configurado el puerto 8230, que es el ofrecido por WildFly en su consola de administración.

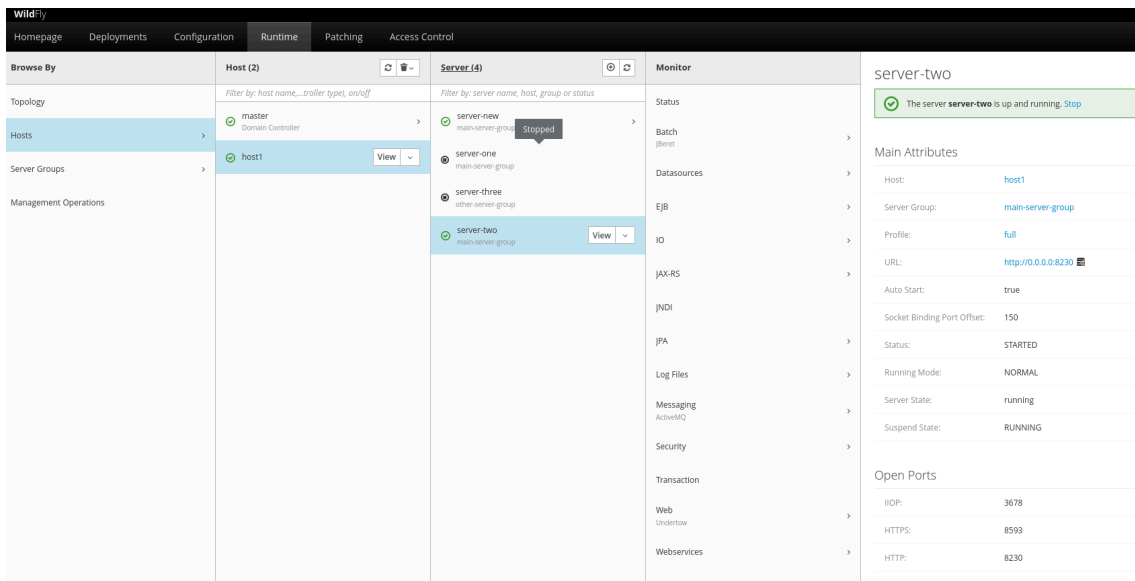


Figura 67. Host1 en consola de WildFly

```

apiVersion: v1
kind: Service
metadata:
  name: jboss-host
  namespace: jbossns
spec:
  type: NodePort
  selector:
    app: jboss-host
  ports:
    - name: http
      protocol: TCP
      port: 8080
      targetPort: 8080
    - name: http-8230
      protocol: TCP
      port: 8230
      targetPort: 8230
      nodePort: 30003
    - name: admin
      protocol: TCP
      port: 9990
      targetPort: 9990
    - name: jgroups
      protocol: TCP
      port: 7600
      targetPort: 7600

```

Figura 68. Yaml del Servicio jboss-host

Tras arrancar el deployment del host, el segundo nodo, podemos ver en el log del nodo máster como reconoce a host1, de forma que conforman el clúster, formado hasta ahora por un nodo máster y un nodo host.

```

2024-05-29 18:26:50,823 INFO [org.jboss.as.domain.controller] (Host Controller Service Threads - 73) WFLYHC0019: Registered remote secondary host "host1", JBoss WildFly Full 27.0.0.Final (WildFly 19.0.0.Final)
2024-05-29 18:28:28,428 INFO [org.jboss.as.domain.controller] (management task-2) WFLYHC0026: Unregistered remote secondary host "host1"
2024-05-29 18:39:15,793 INFO [org.jboss.as.domain.controller] (Host Controller Service Threads - 76) WFLYHC0019: Registered remote secondary host "host1", JBoss WildFly Full 27.0.0.Final (WildFly 19.0.0.Final)
2024-05-29 18:42:47,545 INFO [org.jboss.as.domain.controller] (management task-2) WFLYHC0026: Unregistered remote secondary host "host1"
2024-05-29 18:43:32,458 INFO [org.jboss.as.domain.controller] (Host Controller Service Threads - 78) WFLYHC0019: Registered remote secondary host "host1", JBoss WildFly Full 27.0.0.Final (WildFly 19.0.0.Final)

```

Figura 69. Log WildFly máster reconociendo a nodo host

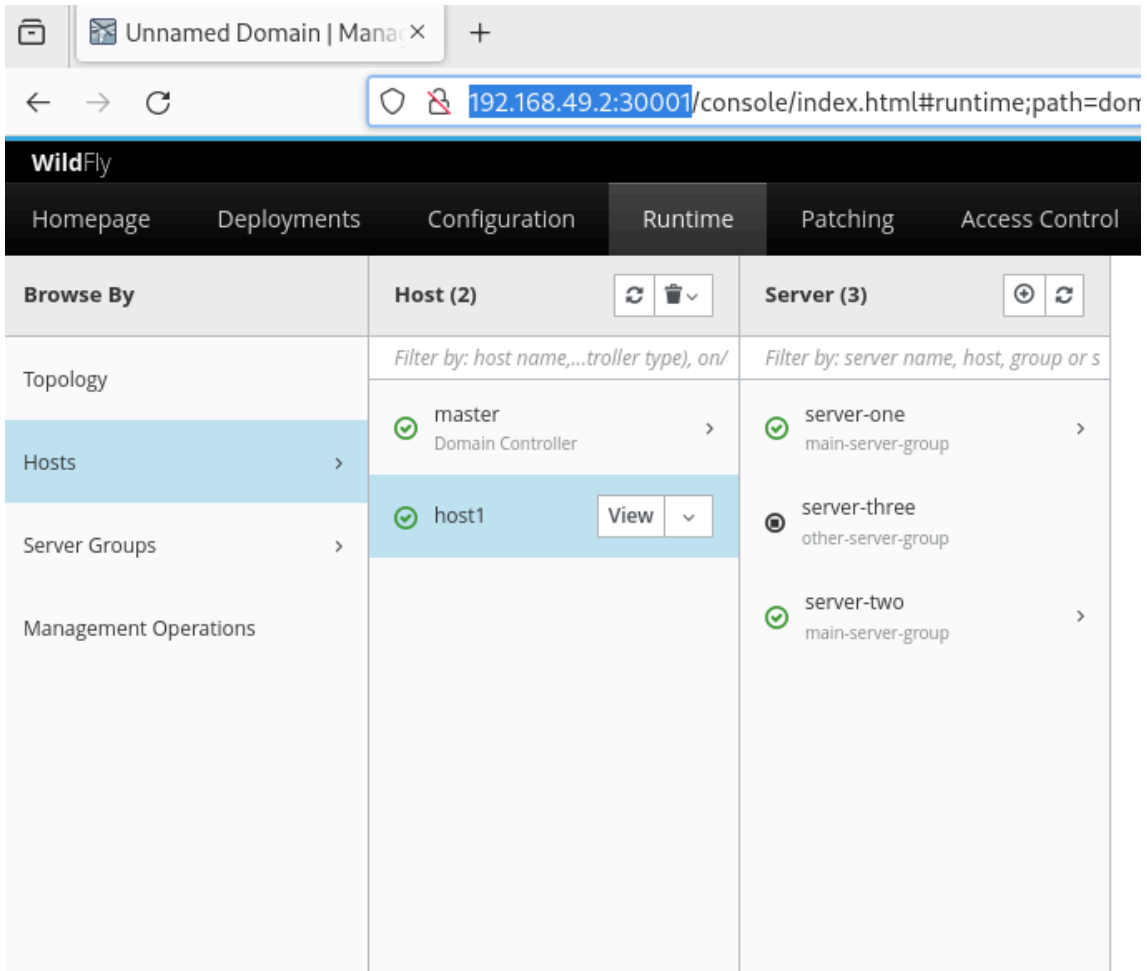


Figura 70. Consola de administración de WildFly con dos nodos

Podemos observar como el clúster se ha formado, desde la consola del Domain vemos dos nodos, el máster y el host1.

#### 4.19 Server-group en WildFly, Host2

Una vez configurado el deployment Host Controller, podemos configurar varios servidores virtuales, dentro de este nodo host.

Vamos a configurar un server group para que la carga se balancee entre dos host virtuales, que conviven físicamente en el nodo host.

Vamos a crear otro servicio en kubernetes para “jboss-host2” que exponga el servicio de otro host controller.

Así queda el server group:

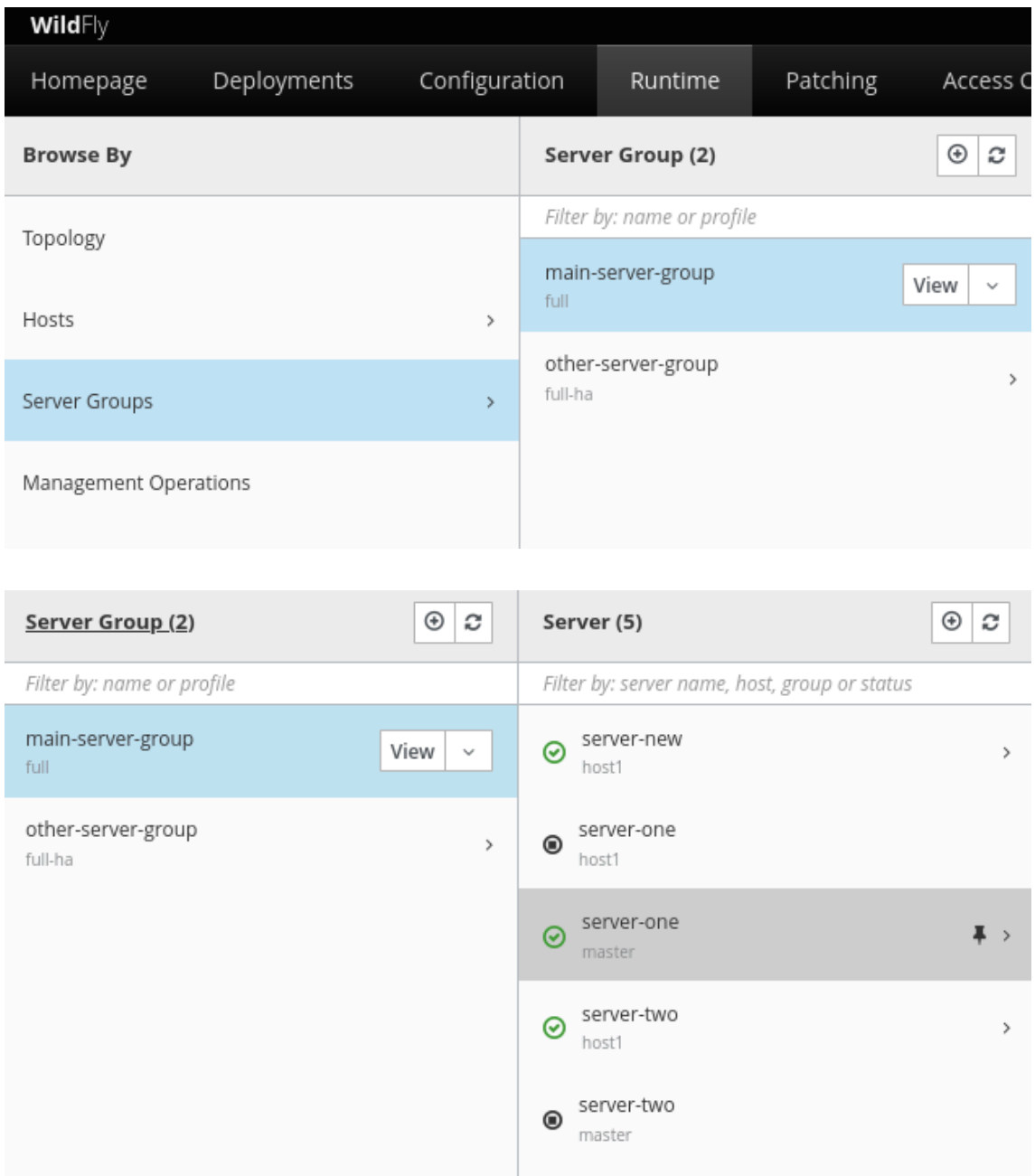


Figura 71. Configuración server-groups en consola Domain de Wildfly

Usaremos el server “server-two” que está en el nodo “host1” y el server “server-new” que también está en “host1”

Main-server-group:

    master:

        server-one: Listen 8080

    host1:

        server-two: Listen 8230 - servicio jboss-host

        server-new: Listen 8430 - servicio jboss-host2

Siempre podemos escalar horizontalmente nuestro clúster, añadiendo más nodos, desplegando más deployments de host controller. Podemos usar el deployment creado para jboss-host, modificando el nombre del host y creando y configurando otro usuario en el domain para que este segundo host se conecte al clúster con su propia identidad.

Pero en nuestro caso, vamos a trabajar sólo con nodo máster y nodo host.

#### 4.19.1 Ingress Controller

Para añadir una capa más en nuestra arquitectura, de forma que la carga pueda balancear entre varios nodos ( en nuestro caso, varios servidores virtuales), vamos a instalar NGINX Ingress Controller.

En nuestro caso, el ingress va a balancear la carga entre los dos servicios: host y host2

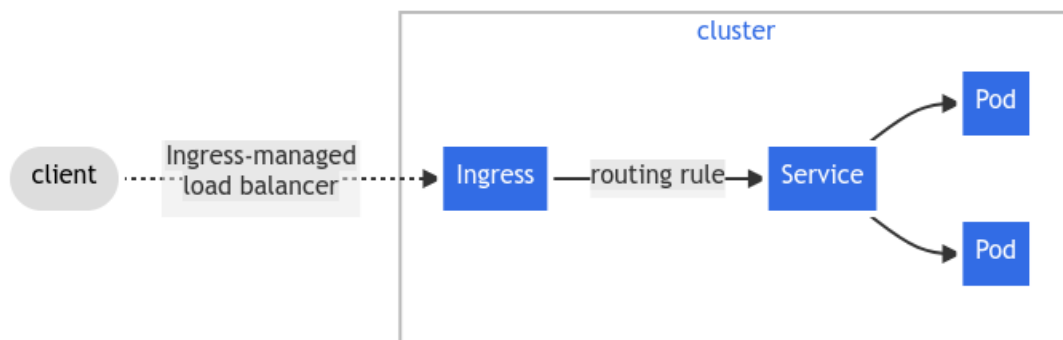


Figura 72. Ingress

Nombre	Etiquetas	Tipo	IP cluster	Endpoints Internos
● jboss-host2	-	NodePort	10.97.180.38	jboss-host2,jbossns:8430 TCP jboss-host2,jbossns:30004 TCP
● jboss-host	-	NodePort	10.104.224.207	jboss-host,jbossns:8080 TCP jboss-host,jbossns:31643 TCP jboss-host,jbossns:8230 TCP jboss-host,jbossns:30003 TCP jboss-host,jbossns:9990 TCP jboss-host,jbossns:30917 TCP jboss-host,jbossns:7600 TCP jboss-host,jbossns:32627 TCP
● jboss-domain	-	NodePort	10.102.237.60	jboss-domain,jbossns:8080 TCP jboss-domain,jbossns:30002 TCP jboss-domain,jbossns:9990 TCP jboss-domain,jbossns:30001 TCP jboss-domain,jbossns:7600 TCP jboss-domain,jbossns:32620 TCP jboss-domain,jbossns:9999 TCP jboss-domain,jbossns:31508 TCP

Figura 73. Consola de WildFly mostrando los servicios.

En el caso de Minikube, la instalación se realiza añadiendo el addon correspondiente.

Ejecutamos:

minikube addons enable ingress

```
[Gema@local master]$ minikube addons enable ingress
🔔 ingress is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
  • Using image registry.k8s.io/ingress-nginx/controller:v1.9.4
  • Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v20231011-8b53cabe0
  • Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v20231011-8b53cabe0
🌐 Verifying ingress addon...
🌟 The 'ingress' addon is enabled
[Gema@local master]$
```

Figura 74. Instalación ingress en kubernetes

El servicio de ingress balanceará las peticiones en los puertos expuestos por los servicios: 8430 y 8230.

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: jboss-ingress
  namespace: jbossns
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/load-balance: "true"
spec:
  rules:
  - host: hello.mydomain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: jboss-host2
            port:
              number: 8430
      - path: /
        pathType: Prefix
        backend:
          service:
            name: jboss-host
            port:
              number: 8230

```

Figura 75. Configuración Ingress

Previamente configurado en Wildfly.

```

apiVersion: v1
kind: Service
metadata:
  name: jboss-host2
  namespace: jbossns
spec:
  type: NodePort
  selector:
    app: jboss-host
  ports:
  - name: http-8430
    protocol: TCP
    port: 8430
    targetPort: 8430
    nodePort: 30004

```

Figura 76. Yaml de servicio host2

Y lo aplicamos con: `kubectl apply -f ingress.yaml`

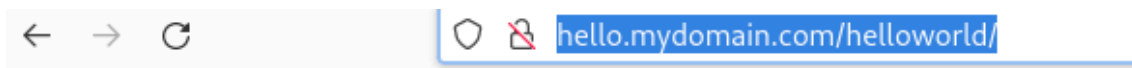
#### 4.19.2 hello.mydomain.com

Para simular el dominio de pruebas: `hello.mydomain.com`, vamos a configurar localmente ese dominio, editando el fichero “`/etc/hosts`”.

```
[Gema@local master]$ cat /etc/hosts
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
195.235.106.197 www.vexcan.es www.emcs.es
192.168.49.2 hello.mydomain.com
```

Figura 77. Configuración nombre dominio DNS en fichero hosts local

Ahora podemos acceder a la aplicación a través del Ingress accediendo a la URL: <http://hello.mydomain.com/helloworld/>



## Hello World

If you see this, the example war-file was correctly deployed! Congrats!

Fri May 31 19:21:12 UTC 2024

You are from 10.244.0.113





Figura 78. Acceso aplicación test desde DNS

Que balancea sobre estas dos URL's:

<http://hello.mydomain.com:30004/helloworld/>  
<http://hello.mydomain.com:30003/helloworld/>

En los logs de Ingress podemos ver las peticiones, aunque están llegando todas al primer de los servicios expuestos en el ingress:

```
kubectl logs -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx -f
```

Cada vez que entramos en <http://hello.mydomain.com/helloworld/> las peticiones llegan al servicio que escucha en el puerto 8430:

```
192.168.49.1 - - [05/Jun/2024:08:38:15 +0000] "GET /helloworld/ HTTP/1.1" 200 97707 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:124.0) Gecko/20100101 Firefox/124.0" 497 0.015 [jbossns-jboss-host2-8430] [] 10.244.0.115:8430 97570 0.015 200 a093ef1dacef62461d123f5e91ef6eff
192.168.49.1 - - [05/Jun/2024:08:38:15 +0000] "GET /favicon.ico HTTP/1.1" 200 1150 "http://hello.mydomain.com/helloworld/" "Mozilla/5.0 (X11; Linux x86_64; rv:124.0) Gecko/20100101 Firefox/124.0" 375 0.029 [jbossns-jboss-host2-8430] [] 10.244.0.115:8430 1150 0.029 200 a2a927b5f229dc3f21dc7a2be9b7500
192.168.49.1 - - [05/Jun/2024:08:38:21 +0000] "GET /helloworld/ HTTP/1.1" 200 97707 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:124.0) Gecko/20100101 Firefox/124.0" 454 0.010 [jbossns-jboss-host2-8430] [] 10.244.0.115:8430 97570 0.009 200 538226d4e44be8063039cb8b211aef0e
192.168.49.1 - - [05/Jun/2024:08:38:23 +0000] "GET /helloworld/ HTTP/1.1" 200 97707 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:124.0) Gecko/20100101 Firefox/124.0" 454 0.015 [jbossns-jboss-host2-8430] [] 10.244.0.115:8430 97570 0.016 200 cd3836c2378048ce61c83cd49a2ae72e
```

Figura 79. Log de ingress

```
[Gema@local master]$ kubectl describe ingress jboss-ingress
Name:          jboss-ingress
Labels:        <none>
Namespace:    jbossns
Address:       192.168.49.2
Ingress Class: nginx
Default backend: <default>
Rules:
  Host          Path  Backends
  ----          -
  hello.mydomain.com
                /    jboss-host2:8430 (10.244.0.115:8430)
                /    jboss-host:8230 (10.244.0.115:8230)
Annotations:   nginx.ingress.kubernetes.io/load-balance: true
                nginx.ingress.kubernetes.io/rewrite-target: /
Events:         <none>
```

Figura 80. configuración ingress con dos servicios

Necesitamos una capa más por encima que balancee sobre los dos puertos expuestos.

## 4.20 HaProxy

Necesitamos montar HAProxy en el clúster de Kubernetes para balancear las peticiones entre los dos servicios que sirven a los host controller. Sólo con un ingress no conseguimos tal balanceo, porque el Ingress por sí solo no puede

manejar adecuadamente los fallos y la distribución del tráfico entre los servicios. HAProxy proporcionará un nivel adicional, asegurando que las peticiones se redirijan correctamente incluso si uno de los servicios o pods falla.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: haproxy
  namespace: jbossns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: haproxy
  template:
    metadata:
      labels:
        app: haproxy
    spec:
      containers:
        - name: haproxy
          image: haproxy:latest
          ports:
            - containerPort: 80
          volumeMounts:
            - name: haproxy-config
              mountPath: /usr/local/etc/haproxy/haproxy.cfg
              subPath: haproxy.cfg
      volumes:
        - name: haproxy-config
          configMap:
            name: haproxy-config
```

Figura 81. Yaml de deployment para haproxy

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: haproxy-config
  namespace: jbossns
data:
  haproxy.cfg: |
    global
      log stdout format raw local0

    defaults
      log      global
      mode     http
      option   httplog
      option   dontlognull
      timeout  connect 5000ms
      timeout  client  50000ms
      timeout  server  50000ms

    frontend http_front
      bind *:80
      default_backend http_back

    backend http_back
      balance roundrobin
      server jboss-host 10.104.224.207:8230 check
      server jboss-host2 10.97.180.38:8430 check

```

Figura 82. Configmap Haproxy

```

apiVersion: v1
kind: Service
metadata:
  name: haproxy
  namespace: jbossns
spec:
  selector:
    app: haproxy
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: NodePort

```

Figura 83. Servicio Haproxy

Además, necesitaremos modificar el ingress para utilizar HaProxy:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: jboss-ingress
  namespace: jbossns
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: hello.mydomain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: haproxy
            port:
              number: 80
```

---

Figura 84. Ingress de haproxy

#### 4.20.1 Balanceo de carga

Ahora que hemos configurado un clúster de WildFly con dos nodos en Kubernetes, nuestra aplicación desplegada se beneficia de una mayor resiliencia.

La configuración de alta disponibilidad (HA) de WildFly garantiza que las instancias de la aplicación pueden manejar cargas de trabajo en paralelo y compartir la carga de manera eficiente.

Tener dos servidores asegura que si uno de ellos falla, el otro aún puede continuar proporcionando servicio a los usuarios. Esto se logra porque ambos nodos están configurados para manejar la misma aplicación y pueden responder a las mismas solicitudes.

Kubernetes monitoriza continuamente el estado de los pods. Si detecta que uno de los pods de WildFly ha fallado, Kubernetes intentará reiniciar el pod automáticamente.

Ahora, HAProxy está configurado para balancear las peticiones entre los dos servicios que sirven en los puertos 8230 y 8430.

Cuando vayamos a `http://hello.mydomain.com/helloworld/`, HAProxy gestionará el balanceo de carga entre los dos servicios.

Para comprobar si funciona correctamente, podemos detener uno de los servicios y verificar si HAProxy redirige las peticiones al servicio que sigue funcionando.

Podemos revisar los logs de HAProxy para más detalles:

```
kubectl logs -l app=haproxy -n jbossns
```

```
10.244.0.113:41786 [05/Jun/2024:09:02:01.379] http_front http_back/jboss-host 0/0/0/8/10 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:41786 [05/Jun/2024:09:02:04.223] http_front http_back/jboss-host2 0/0/1/9/13 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:41786 [05/Jun/2024:09:02:05.510] http_front http_back/jboss-host 0/0/0/14/18 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:41786 [05/Jun/2024:09:02:06.276] http_front http_back/jboss-host2 0/0/0/13/15 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
```

Figura 85. Log Haproxy

#### 4.20.2 Pruebas balanceo

Si paramos el servicio de uno de los servidores desde la consola de administración de WildFly, vemos en el log de haproxy que se detecta ese servicio como caído y la carga se balancea al servicio funcionando:

Server Group (2)	Server (5)
Filter by: name or profile	Filter by: server name, host, group or status
main-server-group full <span style="float: right;">View ▾</span>	<input type="radio"/> server-new host1
other-server-group full-ha <span style="float: right;">&gt;</span>	<input type="radio"/> server-one host1
	<input checked="" type="radio"/> server-one master <span style="float: right;">&gt;</span>
	<input checked="" type="radio"/> server-two host1 <span style="float: right;">&gt;</span>
	<input type="radio"/> server-two master

Figura 86. Servicio server-new parado

```
[WARNING] (8) : Server http_back/jboss-host2 is DOWN, reason: Layer4 connection problem, info: "Connection refused", check duration: 0ms. 1 active and 0 backup servers left. 0 sessions active, 0 requested, 0 remaining in queue.
Server http_back/jboss-host2 is DOWN, reason: Layer4 connection problem, info: "Connection refused", check duration: 0ms. 1 active and 0 backup servers left. 0 sessions active, 0 requested, 0 remaining in queue.
10.244.0.113:50996 [05/Jun/2024:09:06:05.938] http_front http_back/jboss-host 0/0/0/5/7 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:50996 [05/Jun/2024:09:06:07.975] http_front http_back/jboss-host 0/0/0/6/7 200 97668 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:50996 [05/Jun/2024:09:06:10.018] http_front http_back/jboss-host 0/0/0/5/6 200 97668 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
```

Figura 87. Log de haproxy con server-new parado

Volvemos a levantar el servicio jboss-host2 y comprobamos que el haproxy vuelve a enviarle peticiones:

```
Server http_back/jboss-host2 is UP, reason: Layer4 check passed, check duration: 0ms. 2 active and 0 backup servers online. 0 sessions requested, 0 total in queue.
[WARNING] (8) : Server http_back/jboss-host2 is UP, reason: Layer4 check passed, check duration: 0ms. 2 active and 0 backup servers online. 0 sessions requested, 0 total in queue.
10.244.0.113:44102 [05/Jun/2024:09:11:55.752] http_front http_back/jboss-host 0/0/0/15/16 200 97668 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
10.244.0.113:44102 [05/Jun/2024:09:11:57.366] http_front http_back/jboss-host2 0/0/1/1404/1418 200 97777 - - ---- 1/1/0/0/0 0/0 "GET /helloworld/ HTTP/1.1"
```

Figura 88. Log de haproxy con server-new levantado

## 5. Materiales

### 5.1 Hardware utilizado

Este trabajo se llevó a cabo en un portátil Samsung con un procesador Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz y 12 GB de RAM.

El sistema operativo de dicho portátil es Linux, así que no hizo falta hacer uso de ninguna máquina virtual.

## 5.2 Software y herramientas

### Docker

Es un proyecto de código abierto, que permite empaquetar aplicaciones y dependencias de esta, dentro de un contenedor.

### Minikube

Es una distribución minimizada de kubernetes, que nos permite crear un clúster de forma funcional pero más ligero.

### GitHub

Es un servicio en la nube que nos permite almacenar nuestros proyectos y acceder a otros que sean públicos.

### Jboss/WildFly

Es una plataforma de aplicaciones Java, un gestor de aplicaciones, que proporciona un entorno de ejecución de aplicaciones.

### HaProxy

Es un servicio que ofrece un proxy TCP y HTTP de alta disponibilidad.

## 6. Resultados

Hemos logrado desplegar manualmente un clúster de WildFly en modo domain sin utilizar ningún operador de Kubernetes.

El clúster está formado por dos instancias de WildFly, una instancia de domain controller y una instancia de host controller que se vincule al domain, para poder administrar el clúster desde la consola de administración.

Inicialmente, teníamos la intención de trabajar con la imagen de JBoss Enterprise Application Platform (EAP), la cual, al ser respaldada por Red Hat, ofrece mayor seguridad y soporte en entornos productivos. Sin embargo, decidimos optar por una versión de código abierto gratuita, como WildFly, desde el principio, ya que carece de soporte empresarial pero es asequible para cualquier empresa.

Aunque JBoss EAP y WildFly son similares en términos de arquitectura y funcionalidad, nos inclinamos hacia WildFly debido a la atractiva posibilidad de implementar esta solución con un presupuesto reducido.

En entornos productivos, sería recomendable instalar la versión de Red Hat, JBoss EAP, aunque no es necesario.

Nuestro objetivo era configurar un clúster de Jboss/WildFly en alta disponibilidad, por ello, hemos configurado un HaProxy para que balancee las peticiones a los host controller que exponamos.

Hemos personalizado las imágenes de WildFly con los cambios necesarios y los hemos subido a un repositorio de GitHub propio para que estén accesibles desde la nube.

## 7. Conclusión

La elección entre JBoss Enterprise Application Platform (EAP) y WildFly se fundamenta en la necesidad de balancear costos y soporte. Si bien JBoss EAP ofrece ventajas significativas en términos de seguridad y soporte empresarial, WildFly permite un enfoque más económico y accesible, especialmente útil en fases iniciales y de prueba. No obstante, se verifica que WildFly presenta ciertas limitaciones en cuanto a documentación y soporte técnico, lo cual podría representar un desafío en entornos productivos.

La implementación de WildFly en modo domain dentro de un clúster de Kubernetes ha demostrado ser factible pero compleja. El despliegue inicial sin el uso de un operador de Kubernetes requiere configuraciones precisas y ajustes manuales, evidenciando la necesidad de un conocimiento profundo tanto de Kubernetes como de WildFly.

Uno de los desafíos más significativos es asegurar la correcta conectividad entre los distintos pods, especialmente entre el host controller y el domain controller. El uso de servicios y resoluciones DNS internas de Kubernetes es crucial para permitir la comunicación entre estos componentes.

La configuración de servicios NodePort y ClusterIP es fundamental para exponer los puertos necesarios y asegurar la accesibilidad tanto interna como externa al clúster.



Los archivos de configuración, especialmente host.xml, requieren múltiples ajustes para alinearse con la arquitectura de Kubernetes. He de decir, que cuando se trata de nodos independientes dentro de la misma VLAN, con Ip's estáticas, la configuración entre domain controller y host controller es más sencilla, porque no es necesario exponer los puertos para acceder desde fuera ni crear servicios, como ocurre en kubernetes. Esto ha dado más trabajo, puesto que la configuración oficial de WildFly no contempla esto.

A pesar de los desafíos, el despliegue de JBoss/WildFly en Kubernetes ofrece ventajas significativas, incluyendo escalabilidad, gestión simplificada de recursos y alta disponibilidad. La capacidad de Kubernetes para manejar la orquestación de contenedores proporciona una base robusta para implementar aplicaciones empresariales complejas.

Para futuras implementaciones en entornos productivos, probaría el uso de operadores de Kubernetes específicos para JBoss/WildFly, que pueden simplificar significativamente el proceso de despliegue y gestión.

## 8. Glosario

**Linux:** Sistema operativo de código abierto basado en Unix. Es utilizado ampliamente en servidores, supercomputadoras, dispositivos integrados y en el desarrollo de software.

**Docker:** Una plataforma de software que permite empaquetar aplicaciones y sus dependencias en contenedores, lo que facilita el despliegue y la ejecución en cualquier entorno compatible con Docker.

**Kubernetes:** Sistema de orquestación de contenedores de código abierto para automatizar el despliegue, escalado y gestión de aplicaciones basadas en contenedores como Docker.

**Imagen:** Una plantilla que contiene el sistema operativo, la aplicación y todas sus dependencias necesarias para ejecutar un contenedor.

**Jboss:** Una familia de productos de software de middleware empresarial, principalmente el servidor de aplicaciones JBoss Enterprise Application Platform (EAP), utilizado para desplegar aplicaciones Java EE.

**Dockerfile:** Un archivo de texto que contiene una serie de instrucciones para construir una imagen de Docker, especificando el entorno base, la instalación de dependencias y otros pasos necesarios para configurar una aplicación.

**Instancia:** Un único ejemplar de un recurso o entidad dentro del clúster de Kubernetes.

**Deployment:** Es un recurso que describe cómo deben ser gestionadas y actualizadas las réplicas de un conjunto de pods, proporcionando mecanismos para el despliegue y el escalado automático.

**Pod:** El bloque básico de Kubernetes que representa una o más instancias de contenedores ejecutándose en un mismo contexto, con recursos compartidos como almacenamiento y red.

**Standalone:** En JBoss/WildFly, se refiere a un modo de ejecución donde el servidor de aplicaciones funciona como una única instancia independiente, sin gestión centralizada o agrupación en clúster.

**Domain:** En JBoss/WildFly, se refiere a un modo de ejecución donde múltiples servidores de aplicaciones están organizados bajo un controlador de dominio centralizado, permitiendo la gestión de clústeres y la configuración centralizada.

**Minikube:** Una herramienta que permite ejecutar Kubernetes localmente, proporcionando un entorno de desarrollo y pruebas simple para experimentar con Kubernetes en una sola máquina.

**kubectrl:** Es la herramienta para gestionar, por línea de comandos, kubernetes.

**Clúster:** Un conjunto de máquinas interconectadas que trabajan conjuntamente como un sistema unificado y que ejecutan aplicaciones y servicios en contenedores, gestionados por un controlador central.

## 9. Bibliografía

[1] Desplegar un clúster de JBoss en Kubernetes con alta disponibilidad y descubrimiento de nodos dinámicos, 2024.

<https://deividsdocs.wordpress.com/2024/02/21/desplegar-un-cluster-de-jboss-en-kubernetes-con-alta-disponibilidad-y-descubrimiento-de-nodos-dinamicos/> [En línea] [Último acceso: 31 Mayo 2024]

[2] How to Install Minikube on Fedora 36 Step by Step, 2022

<https://www.linuxbuzz.com/how-to-install-minikube-on-fedora/> [En línea] [Último acceso: 31 Mayo 2024]

[3] Guía de alta disponibilidad

[https://docs.wildfly.org/31/High\\_Availability\\_Guide.html#remote-standalone-clients](https://docs.wildfly.org/31/High_Availability_Guide.html#remote-standalone-clients) [En línea] [Último acceso: 31 Mayo 2024]

[4] Domain Setup

[https://docs.wildfly.org/31/Admin\\_Guide.html#Domain\\_Setup](https://docs.wildfly.org/31/Admin_Guide.html#Domain_Setup) [En línea] [Último acceso: 29 Mayo 2024]

[5] Access Cluster

[https://docs.wildfly.org/31/High\\_Availability\\_Guide.html#cluster-configuration](https://docs.wildfly.org/31/High_Availability_Guide.html#cluster-configuration) [En línea] [Último acceso: 31 Mayo 2024]

[6] Dockerfile

<https://gist.github.com/welshstew/dff93abd8e523ad282ad98629629b0ed> [En línea] [Último acceso: 31 Mayo 2024]

[7] Run the Container in Kubernetes

[https://www.eclipse.org/community/eclipse\\_newsletter/2020/november/3.php](https://www.eclipse.org/community/eclipse_newsletter/2020/november/3.php) [En línea] [Último acceso: 31 Mayo 2024]

[8] Comparison: JBoss EAP and JBoss EAP for OpenShift

[https://access.redhat.com/documentation/en-us/red\\_hat\\_jboss\\_enterprise\\_application\\_platform/7.2/html/getting\\_started\\_with\\_jboss\\_eap\\_for\\_openshift\\_container\\_platform/introduction](https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.2/html/getting_started_with_jboss_eap_for_openshift_container_platform/introduction) [En línea] [Último acceso: 31 Mayo 2024]

[9]

<https://ellin.com/2021/03/11/highly-available-wildfly-applications-on-kubernetes/> [En línea] [Último acceso: 1 Junio 2024]

[10] WildFly Administration Remote Access

<https://www.baeldung.com/wildfly-remote-access> [En línea] [Último acceso: 1 Junio 2024]

[11] Remote clients  
[https://docs.wildfly.org/31/High\\_Availability\\_Guide.html#remote-clients-on-another-instance](https://docs.wildfly.org/31/High_Availability_Guide.html#remote-clients-on-another-instance) [En línea] [Último acceso: 1 Junio 2024]

[12] Ingress  
<https://kubernetes.io/es/docs/concepts/services-networking/ingress/> [En línea] [Último acceso: 5 Junio 2024]

[13] Describe ingress  
<https://kubernetes.io/es/docs/concepts/services-networking/ingress/#load-balancing> [En línea] [Último acceso: 5 Junio 2024]

[14]  
<https://adictosaltrabajo.com/2018/11/21/configurar-nginx-ingress-en-k8s/>  
[En línea] [Último acceso: 5 Junio 2024]

[15]  
<https://www.redhat.com/es/topics/containers/what-is-docker> [En línea]  
[Último acceso: 31 Mayo 2024]

[16] HaProxy  
<https://patriciocerda.com/?p=1284> [En línea] [Último acceso: 5 Junio 2024]