Documentación Completa del Proyecto Microservicios

Índice

Introducción

Este documento tiene como objetivo proporcionar una guía completa sobre la implementación de un proyecto de microservicios, desde la arquitectura hasta la puesta en producción utilizando Kubernetes y Azure DevOps.

Arquitectura del Proyecto

El proyecto sigue una arquitectura de microservicios, donde cada servicio es independiente y se comunica con los demás a través de APIs REST. Los principales componentes son:

- **Frontend**: Aplicación web construida con Vue.js.
- **Auth API**: Servicio de autenticación construido con Go.
- **Users API**: Servicio de gestión de usuarios construido con Java.
- **Todos API**: Servicio de gestión de tareas construido con Node.js.
- **API Gateway**: Punto de entrada único para todas las APIs, construido con Nginx.
- **Redis**: Base de datos en memoria para la gestión de sesiones y caché.

Patrones Implementados

API Gateway

Implementación: El API Gateway se implementó utilizando Nginx. Se configuró como un punto de entrada único para todas las APIs, manejando el

enrutamiento de las solicitudes a los diferentes microservicios.

Ubicación: El archivo de configuración del API Gateway se encuentra en `frontend/nginx.conf`.

Código:

```
server {

listen 80;

server_name localhost;



location / {

root /usr/share/nginx/html;

index index.html;

try_files $uri $uri/ /index.html;

}



location /api {

proxy_pass http://api-gateway.dev.svc.cluster.local;

proxy_set_header Host $host;

proxy_set_header X-Real-IP $remote_addr;

}

}
```

Justificación: El API Gateway centraliza el acceso a los microservicios, lo que simplifica la gestión de las APIs y mejora la seguridad al ocultar la estructura interna del sistema. Además, permite implementar políticas de seguridad, autenticación y autorización en un solo lugar.

Service Discovery

Implementación: Kubernetes gestiona automáticamente la descubribilidad de los servicios a través de su sistema de DNS interno. Cada servicio se registra con un nombre único y puede ser accedido por otros servicios utilizando este nombre.

Ubicación: La configuración de los servicios se encuentra en los archivos YAML de Kubernetes, como `infra/k8s/auth-api-deployment.yaml`.

Código:

```
apiVersion: apps/v1

kind: Deployment

metadata:

name: auth-api

namespace: dev

spec:

replicas: 2

selector:

matchLabels:

app: auth-api

template:

metadata:

labels:

app: auth-api

spec:

containers:

- name: auth-api
```

```
image: $(ACR_NAME).azurecr.io/auth-api:$(Build.BuildId)

ports:

- containerPort: 8081
```

Justificación: Service Discovery es esencial en una arquitectura de microservicios para que los servicios puedan encontrarse y comunicarse entre sí sin necesidad de configuraciones manuales. Kubernetes facilita esto con su sistema de DNS interno.

Circuit Breaker

Implementación: El patrón Circuit Breaker se implementó en el API Gateway utilizando Nginx. Se configuró para manejar fallos en los servicios subyacentes, evitando que un fallo en un servicio afecte a todo el sistema.

Ubicación: La configuración del Circuit Breaker se encuentra en `frontend/nginx.conf`.

Código:

```
location /api {

proxy_pass http://api-gateway.dev.svc.cluster.local;

proxy_set_header Host $host;

proxy_set_header X-Real-IP $remote_addr;

proxy_next_upstream error timeout http_500 http_502 http_503 http_504;

proxy_next_upstream_tries 3;

}
```

Justificación: Circuit Breaker es crucial para mejorar la resiliencia del sistema. Cuando un servicio falla, el Circuit Breaker "abre" el circuito y redirige las solicitudes a un fallback o devuelve un error controlado, evitando la propagación del fallo.

CQRS (Command Query Responsibility Segregation)

Implementación: El patrón CQRS se implementó en el servicio de tareas (`Todos API`). Se separaron las operaciones de lectura y escritura en diferentes rutas y controladores.

Ubicación: La implementación de CQRS se encuentra en el código fuente del servicio `Todos API`, específicamente en los archivos de controladores y rutas.

Código:

```
// controllers/todosController.js

const getTodos = async (req, res) => {

// Lógica para obtener tareas

};


const createTodo = async (req, res) => {

// Lógica para crear una tarea

};


module.exports = {

getTodos,

createTodo

};


// routes/todosRoutes.js

const express = require('express');

const router = express.Router();

const todosController = require('../controllers/todosController');


router.get('/todos', todosController.getTodos);

router.post('/todos', todosController.createTodo);
```

```
module.exports = router;
```

Justificación: CQRS es útil cuando las operaciones de lectura y escritura tienen diferentes requisitos de rendimiento y escalabilidad. Al separar estas responsabilidades, se puede optimizar cada una de manera independiente.

Cómo Funcionan Juntos

Estos patrones trabajan juntos para crear una arquitectura de microservicios robusta, escalable y mantenible:

1. **API Gateway y Service Discovery**: El API Gateway utiliza Service Discovery para enrutar las solicitudes a los servicios correctos. Esto simplifica la gestión de las APIs y asegura que los servicios puedan encontrarse y comunicarse entre sí sin configuraciones manuales.
2. **API Gateway y Circuit Breaker**: El API Gateway implementa el patrón Circuit Breaker para manejar fallos en los servicios subyacentes. Esto mejora la resiliencia del sistema al evitar que un fallo en un servicio afecte a todo el sistema.
3. **CQRS y Service Discovery**: El patrón CQRS se beneficia de Service Discovery al permitir que las operaciones de lectura y escritura se escalen y optimicen de manera independiente. Service Discovery asegura que estas operaciones puedan encontrarse y comunicarse eficientemente.
4. **CQRS y Circuit Breaker**: Al separar las operaciones de lectura y escritura, CQRS permite que el Circuit Breaker maneje fallos de manera más efectiva. Por ejemplo, si una operación de escritura falla, las operaciones de lectura pueden continuar funcionando sin interrupciones.

La implementación de estos patrones en conjunto proporciona una arquitectura de microservicios que es robusta, escalable y fácil de mantener. Cada patrón aborda un aspecto específico de la arquitectura, y juntos crean un sistema que puede manejar eficientemente las demandas de una aplicación moderna.

Tecnologías Utilizadas

- **Kubernetes**: Orquestación de contenedores.
- **Docker**: Contenerización de aplicaciones.
- **Azure DevOps**: CI/CD pipelines.
- **Azure Container Registry (ACR)**: Almacenamiento de imágenes Docker.
- **Azure Kubernetes Service (AKS)**: Clúster de Kubernetes gestionado.
- **Redis**: Base de datos en memoria.

- **Nginx**: Servidor web y proxy inverso.

Configuración de Kubernetes

Namespaces

Creamos namespaces para separar los entornos de desarrollo y producción:

```
kubectl create namespace dev

kubectl create namespace prod
```

ConfigMap y Secrets

Configuramos ConfigMaps y Secrets para gestionar las configuraciones y credenciales:

```
kubectl apply -f infra/k8s/configmap.yaml -n dev

kubectl apply -f infra/k8s/secrets.yaml -n dev
```

Deployments y Services

Implementamos Deployments y Services para cada microservicio:

```
kubectl apply -f infra/k8s/auth-api-deployment.yaml -n dev

kubectl apply -f infra/k8s/users-api-deployment.yaml -n dev

kubectl apply -f infra/k8s/todos-api-deployment.yaml -n dev

kubectl apply -f infra/k8s/frontend-deployment.yaml -n dev
```

Ingress

Configuramos Ingress para manejar el tráfico externo:

```
kubectl apply -f infra/k8s/ingress.yaml -n dev
```

Creación de Pipelines

Azure DevOps

Configuramos pipelines en Azure DevOps para automatizar la construcción y despliegue de los microservicios.

Pipeline de Construcción

```
trigger:

- main


pool:

vmImage: 'ubuntu-latest'


steps:

- task: UseNode@1

inputs:

version: '14.x'

displayName: 'Use Node.js'


- script: |

npm install

npm run build

displayName: 'Build Frontend'


- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'frontend'

command: 'buildAndPush'
```

```
Dockerfile: 'frontend/Dockerfile'

tags: '$(Build.BuildId)'
```

Pipeline de Despliegue

```
trigger:

- main


pool:

vmImage: 'ubuntu-latest'


steps:

- task: KubernetesManifest@0

inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'infra/k8s/*.yaml'
```

Pipelines de Cada Módulo

Frontend

Ubicación: El pipeline de construcción y despliegue del frontend se encuentra en `frontend/pipeline.yaml`.

Código:

```
trigger:

- main
```

```
pool:

vmImage: 'ubuntu-latest'


steps:

- task: UseNode@1

inputs:

version: '14.x'

displayName: 'Use Node.js'


- script: |

npm install

npm run build

displayName: 'Build Frontend'


- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'frontend'

command: 'buildAndPush'

Dockerfile: 'frontend/Dockerfile'

tags: '$(Build.BuildId)'
```

```
- task: KubernetesManifest@0

inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'frontend/k8s/dev/*.yaml'
```

Auth API

Ubicación: El pipeline de construcción y despliegue del Auth API se encuentra en `auth-api/pipeline.yaml`.

Código:

```
trigger:

- main


pool:

vmImage: 'ubuntu-latest'


steps:

- task: UseGo@0

inputs:

version: '1.16'

displayName: 'Use Go'


- script: |

go mod download
```

```
go build -o auth-api .

displayName: 'Build Auth API'


- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'auth-api'

command: 'buildAndPush'

Dockerfile: 'auth-api/Dockerfile'

tags: '$(Build.BuildId)'


- task: KubernetesManifest@0

inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'auth-api/k8s/dev/*.yaml'
```

Users API

Ubicación: El pipeline de construcción y despliegue del Users API se encuentra en `users-api/pipeline.yaml`.

Código:

```
trigger:

- main
```

```yaml
pool:

vmImage: 'ubuntu-latest'


steps:

- task: UseJava@1

inputs:

version: '11'

displayName: 'Use Java'


- script: |

mvn clean install

displayName: 'Build Users API'


- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'users-api'

command: 'buildAndPush'

Dockerfile: 'users-api/Dockerfile'

tags: '$(Build.BuildId)'


- task: KubernetesManifest@0
```

```
inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'users-api/k8s/dev/*.yaml'
```

Todos API

Ubicación: El pipeline de construcción y despliegue del Todos API se encuentra en `todos-api/pipeline.yaml`.

Código:

```
trigger:

- main


pool:

vmImage: 'ubuntu-latest'


steps:

- task: UseNode@1

inputs:

version: '14.x'

displayName: 'Use Node.js'


- script: |

npm install

npm run build
```

```
displayName: 'Build Todos API'


- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'todos-api'

command: 'buildAndPush'

Dockerfile: 'todos-api/Dockerfile'

tags: '$(Build.BuildId)'


- task: KubernetesManifest@0

inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'todos-api/k8s/dev/*.yaml'
```

API Gateway

Ubicación: El pipeline de construcción y despliegue del API Gateway se encuentra en `api-gateway/pipeline.yaml`.

Código:

```
trigger:

- main
```

```
pool:

vmImage: 'ubuntu-latest'


steps:

- task: Docker@2

inputs:

containerRegistry: 'acr-connection'

repository: 'api-gateway'

command: 'buildAndPush'

Dockerfile: 'api-gateway/Dockerfile'

tags: '$(Build.BuildId)'


- task: KubernetesManifest@0

inputs:

action: 'deploy'

kubernetesServiceConnection: 'k8s-dev-connection'

namespace: 'dev'

manifests: 'api-gateway/k8s/dev/*.yaml'
```

Los pipelines de cada módulo están configurados para automatizar la construcción y despliegue de los microservicios. Cada pipeline sigue un flujo similar: instalar dependencias, construir la aplicación, crear y subir la imagen Docker, y desplegar en Kubernetes. Esto asegura que los microservicios se construyan y desplieguen de manera consistente y eficiente.

Configuración y Gestión de Clústeres de Kubernetes

Creación del Clúster

Para crear un clúster de Kubernetes en Azure, utilizamos Azure Kubernetes Service (AKS). Aquí están los pasos y comandos utilizados:

1. **Crear un Grupo de Recursos**:

```
az group create --name microservices-rg --location eastus
```

2. **Crear un Clúster de AKS**:

```
az aks create --resource-group microservices-rg --name microservices-aks --node-count 2 --en
```

3. **Obtener Credenciales del Clúster**:

```
az aks get-credentials --resource-group microservices-rg --name microservices-aks
```

Configuración de Namespaces

Los namespaces en Kubernetes permiten aislar recursos dentro de un clúster. Creamos namespaces para los entornos de desarrollo y producción:

1. **Crear Namespace de Desarrollo**:

```
kubectl create namespace dev
```

2. **Crear Namespace de Producción**:

```
kubectl create namespace prod
```

Configuración de ConfigMap y Secrets

Los ConfigMaps y Secrets se utilizan para gestionar configuraciones y credenciales:

1. **Crear ConfigMap**:

```
kubectl apply -f infra/k8s/configmap.yaml -n dev
```

2. **Crear Secrets**:

```
kubectl apply -f infra/k8s/secrets.yaml -n dev
```

Despliegue de Microservicios

Desplegamos los microservicios utilizando archivos YAML de Kubernetes:

1. **Desplegar Auth API**:

```
kubectl apply -f infra/k8s/auth-api-deployment.yaml -n dev
```

2. **Desplegar Users API**:

```
kubectl apply -f infra/k8s/users-api-deployment.yaml -n dev
```

3. **Desplegar Todos API**:

```
kubectl apply -f infra/k8s/todos-api-deployment.yaml -n dev
```

4. **Desplegar Frontend**:

```
kubectl apply -f infra/k8s/frontend-deployment.yaml -n dev
```

5. **Desplegar API Gateway**:

```
kubectl apply -f infra/k8s/api-gateway-deployment.yaml -n dev
```

Configuración de Ingress

El Ingress maneja el tráfico externo hacia los servicios:

1. **Crear Ingress**:

```
kubectl apply -f infra/k8s/ingress.yaml -n dev
```

Comandos Utilizados

Azure CLI

```
az login

az account set --subscription "Nombre de la suscripción"

az group create --name microservices-rg --location eastus

az acr create --resource-group microservices-rg --name tuacr --sku Basic

az aks create --resource-group microservices-rg --name microservices-aks --node-count 2 --er
```

```
az aks get-credentials --resource-group microservices-rg --name microservices-aks
```

Kubernetes

```
kubectl apply -f infra/k8s/configmap.yaml -n dev

kubectl apply -f infra/k8s/secrets.yaml -n dev

kubectl apply -f infra/k8s/auth-api-deployment.yaml -n dev

kubectl apply -f infra/k8s/ingress.yaml -n dev
```

Tutorial para Activar Todo

1. **Configuración del Entorno**:
   - Instala Docker, Kubernetes y Azure CLI.
   - Configura Azure DevOps y crea un proyecto.
2. **Construcción de Imágenes**:
   - Construye y sube las imágenes Docker a ACR.
3. **Despliegue en Kubernetes**:
   - Aplica los archivos de configuración de Kubernetes para desplegar los microservicios.
4. **Configuración de Pipelines**:
   - Crea y ejecuta los pipelines de construcción y despliegue en Azure DevOps.
5. **Verificación**:
   - Verifica que todos los servicios estén corriendo correctamente en Kubernetes.

Explicación de los Pods de Kubernetes

Un **Pod** es la unidad más pequeña y simple en el modelo de objetos de Kubernetes. Un Pod representa un conjunto de contenedores que comparten almacenamiento y red, y una especificación sobre cómo ejecutar los contenedores.

Características de los Pods

- **Contenedores**: Un Pod puede contener uno o más contenedores.
- **Almacenamiento**: Los contenedores en un Pod comparten volúmenes de almacenamiento.

- **Red**: Los contenedores en un Pod comparten la misma dirección IP y espacio de puertos.

Ejemplo de un Pod

```
apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: nginx
```

Gestión de Pods

- **Crear un Pod**:

```
kubectl apply -f pod.yaml
```

- **Listar Pods**:

```
kubectl get pods
```

- **Eliminar un Pod**:

```
kubectl delete pod mypod
```