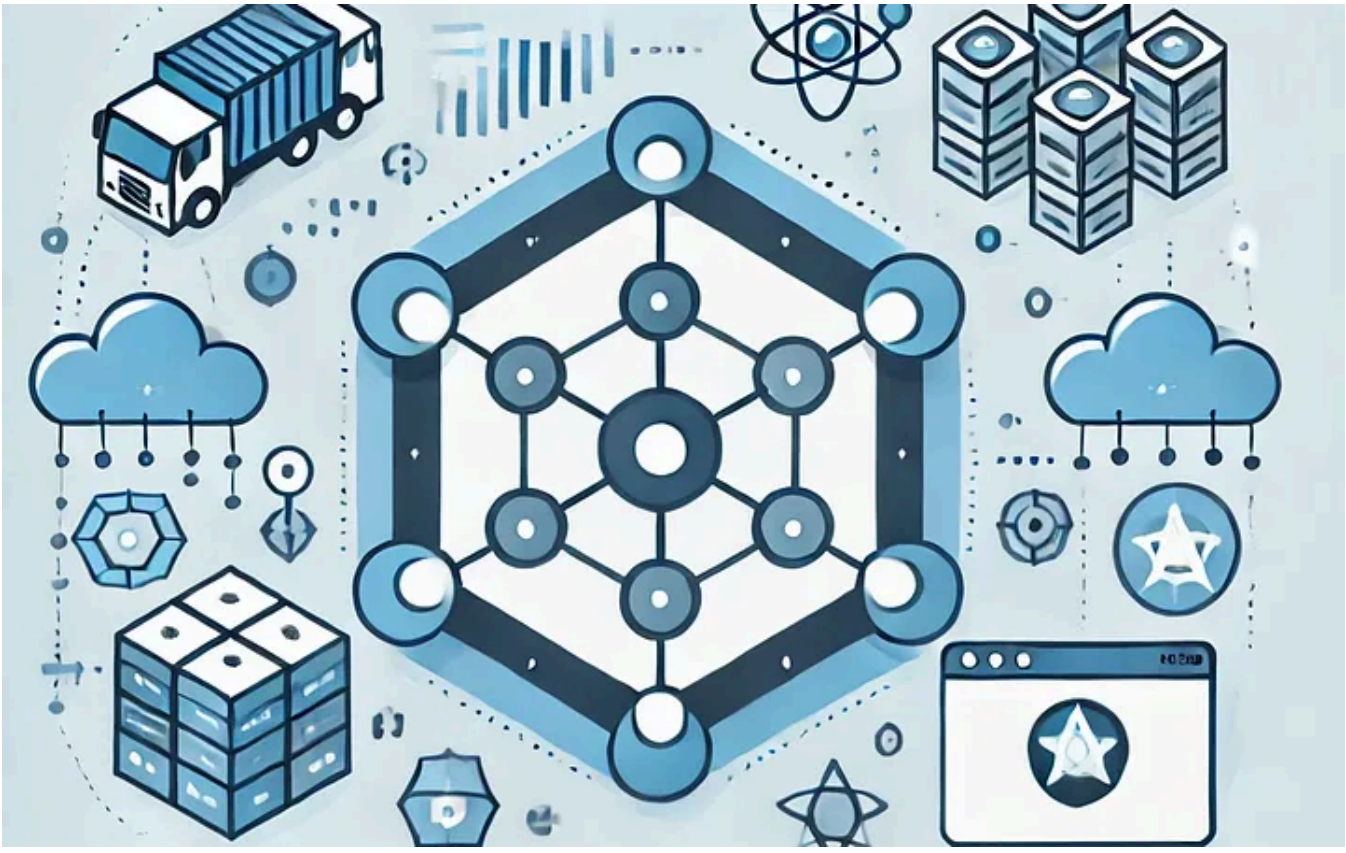
Microservices Architecture with .NET Core and Kubernetes

Paulo Torres · Follow
4 min read · Jul 23, 2024

Listen Share More



Microservices architecture has become a cornerstone of modern software development, enabling organizations to build scalable, resilient, and maintainable applications. Combining .NET Core with Kubernetes provides a powerful platform for developing, deploying, and managing microservices. This article delves into the advanced aspects of implementing microservices architecture using .NET Core and Kubernetes, illustrated with real-world examples.

Understanding Microservices

Principle Overview

Microservices architecture decomposes a large application into smaller, independent services that communicate over well-defined APIs. Each microservice focuses on a specific business capability, facilitating better scalability and development agility.

Setting Up the Environment

Step 1: Create a New .NET Core Microservice

First, create a new .NET Core Web API project for your microservice:

```
dotnet new webapi -n ProductService
cd ProductService
```

Step 2: Define the Microservice

In this example, we will create a simple Product Service that handles product data.

Models/Product.cs

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Controllers/ProductController.cs

```
[ApiController]
[Route("api/[controller]")]
public class ProductController : ControllerBase
{
    private static readonly List<Product> Products = new List<Product>
    {
        new Product { Id = 1, Name = "Laptop", Price = 1200.00M },
        new Product { Id = 2, Name = "Smartphone", Price = 800.00M }
```

```
    };

    [HttpGet]
    public ActionResult<IEnumerable<Product>> Get() => Products;

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        var product = Products.FirstOrDefault(p => p.Id == id);
        if (product == null) return NotFound();
        return product;
    }

    [HttpPost]
    public ActionResult Post(Product product)
    {
        Products.Add(product);
        return CreatedAtAction(nameof(Get), new { id = product.Id }, product);
    }
}
```

Step 3: Dockerize the Microservice

Create a `Dockerfile` to containerize the Product Service.

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["ProductService/ProductService.csproj", "ProductService/"]
RUN dotnet restore "ProductService/ProductService.csproj"
COPY . .
WORKDIR "/src/ProductService"
RUN dotnet build "ProductService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "ProductService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ProductService.dll"]
```

Build and run the Docker image:

```
docker build -t productservice .
docker run -d -p 8080:80 --name productservice productservice
```

Deploying to Kubernetes

Step 4: Create Kubernetes Deployment and Service

Create Kubernetes manifests for deploying the Product Service.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: productservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: productservice
  template:
    metadata:
      labels:
        app: productservice
    spec:
      containers:
      - name: productservice
        image: productservice:latest
        ports:
        - containerPort: 80
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: productservice
spec:
  selector:
    app: productservice
  ports:
    - protocol: TCP
      port: 80
```

```
    targetPort: 80
  type: LoadBalancer
```

Deploy the service to Kubernetes:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Step 5: Set Up Ingress Controller
Set up an Ingress Controller to manage external access to the microservices.

ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: productservice-ingress
spec:
  rules:
  - host: productservice.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: productservice
            port:
              number: 80
```

Apply the ingress configuration:

```
kubectl apply -f ingress.yaml
```

Step 6: Configuring CI/CD Pipeline
Automate the build, test, and deployment process using CI/CD pipelines. Example

with GitHub Actions:

.github/workflows/ci-cd-pipeline.yaml

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up .NET
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: '5.0.x'

    - name: Build
      run: dotnet build --configuration Release

    - name: Test
      run: dotnet test --no-build --verbosity normal

    - name: Docker Build and Push
      run: |
        docker build -t productservice .
        docker tag productservice:latest yourdockerhub/productservice:latest
        echo "${{ secrets.DOCKER_HUB_PASSWORD }}" | docker login -u "${{ secret
        docker push yourdockerhub/productservice:latest

    - name: Deploy to Kubernetes
      uses: Azure/k8s-deploy@v1
      with:
        manifests: |
          ./deployment.yaml
          ./service.yaml
          ./ingress.yaml
        images: |
          yourdockerhub/productservice:latest
```

Monitoring and Scaling

Step 7: Set Up Monitoring with Prometheus and Grafana
Deploy Prometheus and Grafana to monitor the health and performance of your microservices.


```
w.githubusercontent.com/prometheus-operator/prometheus-operator/main/bundle.yaml
```

Set up Grafana for visualizing metrics:

```
kubectl apply -f https://raw.githubusercontent.com/grafana-operator/grafana-ope
```

Step 8: Auto-Scaling with Horizontal Pod Autoscaler
Enable auto-scaling for the Product Service:

```
kubectl autoscale deployment productservice --cpu-percent=50 --min=3 --max=10
```

Conclusion

Implementing microservices architecture with .NET Core and Kubernetes provides a robust framework for building scalable, resilient, and maintainable applications. By leveraging Docker for containerization, Kubernetes for orchestration, and CI/CD pipelines for automation, you can streamline the development and deployment processes. Monitoring tools like Prometheus and Grafana, along with auto-scaling capabilities, ensure that your microservices are always performing optimally. Dive into these advanced techniques to enhance your .NET Core applications and embrace the future of software development.

Microservices Dotnet Kubernetes Cloud Computing DevOps