Dev Ops Documentatie

Gemaakt door:

Ebram

Denzel Bendt

Mitchel Meskes

Datum: 21 juni 2024

Inhoud

Introductie

Welkom bij ons Dev Ops project!

Dit DevOps-pipelineproject draait om automatisering, integratie en voortdurende verbetering. Door een volledig geautomatiseerde pipeline met strenge tests en monitoring te implementeren, willen we efficiënte en betrouwbare software van hoge kwaliteit leveren. De gedetailleerde documentatie en het naleven van de beste praktijken zorgen ervoor dat deze pipeline gemakkelijk kan worden overgenomen en opgeschaald door ontwikkelings- en operationele teams, wat uiteindelijk leidt tot een cultuur van samenwerking en continue levering.

Plan van aanpak

1. Projectkeuze + Toepassing principes

Projectkeuze Wij zijn van plan een Docker-image te maken voor onze software language 2B applicatie. Aangezien wij bij DevOps de applicatie Docker leren te gebruiken, leek het ons handig om dit voort te zetten bij dit project zodat we onze ervaring met Docker kunnen verbeteren.

Toepassing principes Door het gebruik van Docker willen we een standaard werkomgeving creëren, waardoor het samenwerken eenvoudiger wordt. Docker biedt de mogelijkheid om voor ons een consistente ontwikkelomgeving te gebruiken, ongeacht het besturingssysteem dat het teamlid gebruikt. Docker zorgt er ook voor dat we bezig kunnen gaan met automatisering met behulp van Dockerfile en Docker-compose, waardoor het maken van menselijke fouten verminderd wordt.

2. Praktijken en technieken

Wat wordt er gebruikt in de praktijk:

- **Code editor**: Een softwaretool die wordt gebruikt voor het schrijven, bewerken en organiseren van code.
    - Visual Studio Code
- **Database manager**: Een tool die het beheer, de ontwikkeling en de administratie van databases vergemakkelijkt.
    - TablePlus
- **Continuous Integration (CI) Tools**: Deze tools automatiseren het proces van het integreren van code-wijzigingen in een gedeelde repository en het uitvoeren van geautomatiseerde tests op de geïntegreerde code.
    - GitLab CI
- **Continuous Deployment/Continuous Delivery (CD) Tools**: CD-tools automatiseren het proces van het implementeren van code-wijzigingen naar productieomgevingen na succesvolle testen en validatie.
    - GitLab CI/CD
    - Azure DevOps Pipelines

- **Configuration Management Tools**: Deze tools automatiseren het beheer en de configuratie van infrastructuur- en serverresources, waarbij consistentie wordt gegarandeerd over verschillende omgevingen.
    - Git (GitHub, GitLab)
- **Containerization Tools**: Deze tools maken het verpakken van applicaties en hun afhankelijkheden in lichtgewicht, draagbare containers mogelijk, waardoor consistente implementatie over verschillende omgevingen mogelijk is.
    - Docker
- **Monitoring and Logging Tools**: Deze tools bieden inzicht in de prestaties, beschikbaarheid en het gedrag van applicaties en infrastructuur, waardoor problemen snel kunnen worden geïdentificeerd en opgelost.
    - Prometheus
- **Collaboration Tools**: Samenwerkingstools vergemakkelijken communicatie en samenwerking tussen DevOps-teamleden, inclusief projectbeheer, issue-tracking en communicatietools.
    - Azure DevOps
- **Version Control Systems (VCS)**: VCS-tools stellen teams in staat om wijzigingen in de broncode te beheren en bij te houden, waarbij versiebeheer en samenwerking tussen ontwikkelaars worden gegarandeerd.
    - Git (GitHub, GitLab)
- **Security Tools**: Beveiligingstools helpen bij het identificeren en verminderen van beveiligingskwetsbaarheden in de ontwikkeling van software.
    - Spectral

3. Reden voor gebruik volgende tools:

- **Docker**: We hebben voor Docker gekozen omdat we hier als team al eerder ervaring mee hebben van voorgaande opleidingen en stages, en hier een goede ervaring aan hebben overgehouden.
- **Git (GitHub, GitLab)**: We hebben sinds het begin van onze ontwikkeling met deze platformen gewerkt en hebben hier dus al ervaring mee. We hebben tot heden geen alternatief gevonden dat beter bevalt. Deze platformen zijn handig voor het gebruik van versiebeheer en voor onze samenwerking binnen het team.
- **Azure DevOps**: We hebben voor Azure gekozen omdat je hier een goed overzicht kunt maken voor stories en meetings, en ook de mogelijkheid hebt om de Pipelines te gebruiken. Ook hiermee hebben we al eerdere ervaring.

- **Prometheus**: We hebben hiervoor gekozen omdat dit wordt aangeraden.
- **Spectral**: Joshua had dit gevonden en zei dat dit hem beviel. Vervolgens hebben wij kennis opgedaan over dit programma en leek het ons ook een goede optie.

4. Gebruikte technieken

Communicatie We zullen volgens scrum principes samenwerken. We zullen wekelijks user stories afwerken en aan elkaar en de klant presenteren. Met de scrum methode die wij toepassen zullen we daily standups hebben en de code regelmatig pushen naar GitHub. We zullen agile werken en aan het einde van iedere sprint een review sessie houden, waar feedback ontstaat. Hier verwelkomen we nieuwe inzichten zodat we een applicatie kunnen opleveren die het beste past bij de eisen van de klant. Elke twee weken houden we als team een retrospective, waar we ons werk bekritiseren om efficiënter te worden. Buiten werktijden gebruiken we communicatie tools als Discord en WhatsApp om ideeën en plannen te bespreken of feedback te geven.

Codering We zullen werken volgens de coding conventions van Microsoft en zullen een zo schaalbaar mogelijke applicatie maken door gebruik te maken van OOP.

5. Aandachtspunten en verbeterstrategieën

Aandachtspunten

- Zorg voor een strakke planning met samen afgesproken richtlijnen.
- Het gehele team moet akkoord gaan met doelstellingen en verwachtingen.
- Clean code principes aanhouden zodat we op dezelfde manier werken.

Verbeterstrategieën

- Na het afmaken van een user story, pushen we de branch naar GitHub en maken we een pull request. Als de andere developers akkoord gaan, wordt deze gemerged met de main branch, zodat we altijd een werkende applicatie hebben.
- We zullen de applicatie zo klein mogelijk houden door geen onnodige methods en andere code te schrijven.
- Wanneer er dingen niet worden begrepen, worden deze besproken binnen de communicatie tools. Het probleem wordt dan besproken en de teamleden gaan samen kijken hoe ze het kunnen oplossen.

Build Stage

Build Stage: Beschrijf de gekozen build tools (bijv. Maven, Gradle) en de configuratie daarvan.

Gekozen Build Tool:

Docker:

Docker is gekozen als de tool om de applicatie te bouwen. Docker zorgt ervoor dat de omgeving waarin de applicatie wordt gebouwd altijd hetzelfde is, ongeacht waar het wordt uitgevoerd. Dit voorkomt problemen die kunnen ontstaan door verschillen in de omgeving.

Configuratie Details:

1. **Code Uitchecken**:
   De eerste stap in het build-proces is het uitchecken van de code uit de repository met *actions/checkout@v4*. Dit haalt de nieuwste code van de *master* branch op zodat de build met de laatste versie van de code werkt.

```
- name: Checkout
  uses: actions/checkout@v4
```

2. **Docker Image bouwen en Cachen:**

   - De Docker image wordt gebouwd met een Dockerfile die staat in ./Dierentuinn/Dockerfile. Het build-proces gebruikt de --target build optie, wat betekent dat het stopt bij de build-stap zoals gedefinieerd in de Dockerfile. Deze stap bevat meestal alle benodigdheden en stappen om de applicatie te compileren.
   - De optie --build-arg "INCLUDE_UNITTESTING=true" wordt gebruikt om een argument naar de Dockerfile te sturen. Hiermee kunnen we ervoor zorgen dat afhankelijkheden voor unit tests worden meegenomen tijdens het bouwen.
   - De optie --no-cache zorgt ervoor dat Docker de build opnieuw doet zonder gebruik te maken van eerder opgebouwde lagen. Dit is handig om zeker te zijn dat alle nieuwste wijzigingen worden meegenomen.

```
- name: Build and cache Docker image
  run: |
    docker build --no-cache -t ebram100/dierentuinn-name:${{ github.ref_name }} --target build --build-arg "INCLUDE_UNITTESTING=true" -f ./Dierentuinn/Dockerfile .
```

Test Stage

Test Stage: Leg uit welke test frameworks (bijv. JUnit, Selenium) zijn gebruikt en hoe de tests zijn opgezet.

- **Gebruikte Test Frameworks:**

  NET Test Framework: In deze pipeline gebruiken we het ingebouwde test framework van .NET om unit tests uit te voeren. Dit framework is goed geïntegreerd met .NET applicaties en maakt het eenvoudig om tests te schrijven en uit te voeren.

Configuratie Details:

1. Unit Tests Uitvoeren in Docker Container

   Nadat de Docker image is gebouwd, wordt deze gebruikt om unit tests uit te voeren. Dit gebeurt door de Docker container te starten en dotnet test uit te voeren binnen de container. Hiermee wordt ervoor gezorgd dat de tests worden uitgevoerd in dezelfde omgeving als waarin de applicatie is gebouwd.

```
- name: Run dotnet tests inside Docker container
  run: docker run --entrypoint "" ebram100/dierentuinn-name:master dotnet test
```

Deploy Stage

Deploy Stage: Bespreek de deploy methoden (bijv. Kubernetes, Docker Swarm) en hoe de deployment pipeline is ingericht.

Gebruikte methodes:

Kubernetes: We gebruiken Kubernetes om onze applicatie uit te rollen. Kubernetes helpt ons om onze applicatie automatisch te schalen, te balanceren en te herstellen als er problemen zijn.

Stappen uitgelegd:

- **Code checken**
  Eerst halen we de nieuwste versie van onze code uit de repository. Dit zorgt ervoor dat we altijd met de meest recente code werken.

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

- **KubeConfig instellen:**
  Daarna stellen we onze kubeconfig in. Dit bestand bevat informatie die nodig is om verbinding te maken met onze Kubernetes cluster.

```
- name: Deploy to Kubernetes host (old steps)
  run: echo "${{ secrets.KUBECONFIG }}" > ./config.yml
```

- **Uitrollen naar Kubernester**
  Tenslotte rollen we de applicatie uit naar Kubernetes. We gebruiken een bestand genaamd deploy.yml dat vertelt hoe Kubernetes onze applicatie moet draaien.

```
- name: Deploy to Kubernetes host
  run: kubectl apply -f deploy.yml
```

Environments

Environments: Een DevOps-omgeving verwijst naar de verschillende fasen waarin software door de levenscyclus van ontwikkeling tot productie gaat. Elk van deze omgevingen is bedoeld om specifieke taken uit te voeren en heeft unieke configuraties en instellingen. De belangrijkste omgevingen zijn:

Development (Ontwikkelomgeving):

Beschrijving: Dit is de omgeving waarin ontwikkelaars nieuwe functies en wijzigingen in de code implementeren. Hier worden ook de eerste unit tests uitgevoerd.

Kenmerken: Lokale ontwikkelomgevingen, vaak uitgevoerd op de persoonlijke machines van ontwikkelaars. Kan afhankelijk zijn van tools zoals Docker om consistentie te waarborgen.

Doel: Ondersteunen van actieve ontwikkeling en het snel kunnen itereren op de code.

Staging (Testomgeving):

Beschrijving: Dit is een pre-productieomgeving die de productieomgeving zo nauwkeurig mogelijk nabootst. Hier worden uitgebreidere tests uitgevoerd, zoals integratietests, acceptatietests en load tests.

Kenmerken: Nabootsing van de productieomgeving met vergelijkbare configuraties en datasets, vaak gehost op aparte servers of virtuele machines.

Doel: Validatie van de volledige applicatie in een omgeving die de productieomgeving nabootst om eventuele problemen op te sporen voordat de code naar productie gaat.

Production (Productieomgeving):

Beschrijving: Dit is de live omgeving waar de uiteindelijke versie van de software wordt geïmplementeerd en toegankelijk is voor eindgebruikers.

Kenmerken: Hoogste niveaus van beveiliging, prestaties en betrouwbaarheid. Continue monitoring en health checks om de beschikbaarheid en prestaties van de applicatie te garanderen.

Doel: Ondersteunen van de uiteindelijke gebruikers van de applicatie, met een focus op stabiliteit en uptime.

Testing (Testomgeving):

Beschrijving: Een omgeving die specifiek is bedoeld voor het uitvoeren van tests. Dit kan overlappen met de staging-omgeving, maar wordt soms afzonderlijk gehouden voor specifieke testdoeleinden.

Kenmerken: Flexibele configuraties voor verschillende soorten tests zoals unit tests, integratietests, regressietests, enz.

Doel: Testen van verschillende onderdelen van de applicatie in isolatie of geïntegreerd.

Scheiding van Omgevingen

Het is cruciaal om een duidelijke scheiding tussen deze omgevingen te handhaven om verschillende redenen:

Isolatie van Problemen: Problemen in de ontwikkelings- of testfase mogen geen invloed hebben op de productieomgeving.

Consistente Omstandigheden: Door gebruik te maken van tools zoals Docker kunnen we zorgen voor consistente omgevingen, waardoor "it works on my machine" problemen worden verminderd.

Beveiliging: De productieomgeving vereist striktere beveiligingsmaatregelen dan ontwikkelings- en testomgevingen.

Betrouwbaarheid: Door een staging-omgeving te gebruiken die de productieomgeving nabootst, kunnen we potentiële problemen opsporen voordat de software live gaat.

Best Practices voor Omgevingsbeheer

Infrastructure as Code (IaC): Gebruik tools zoals Terraform of Ansible om de infrastructuur als code te beheren en te automatiseren, waardoor consistentie en herhaalbaarheid worden gewaarborgd.

Geautomatiseerde Deployments: Gebruik CI/CD-tools zoals Jenkins, GitLab CI/CD of Azure DevOps voor het automatisch implementeren van applicaties naar verschillende omgevingen.

Monitoring en Logging: Implementeer uitgebreide monitoring en logging in alle omgevingen om vroegtijdig problemen te detecteren en te diagnosticeren.

Rollback Procedures: Zorg voor robuuste rollback-procedures om snel terug te keren naar een vorige stabiele versie als er problemen optreden in de productieomgeving.

Configuratiebeheer: Gebruik configuratiebeheer tools om verschillen in omgevingsspecifieke instellingen en geheimen (bijv. API-sleutels, wachtwoorden) veilig en efficiënt te beheren.

Docker

Beschrijving

Base Image

In het eerste stadium maakt het dockerfile gebruik van de .NET 8.0 ASP.NET run-time als een base image en zet de werkende directory naar '/app'. De ports 8080 en 8081 worden ook weergegeven en gebruikt bij de applicatie.

Build Image

In het tweede stadium wordt .NET 8.0 SDK image gebruikt om de applicatie te bouwen. De environment variabele wordt van 'Build_Configuration' naar 'Release' omgezet, en de werkende directory wordt naar '/src' gezet.

Het Dockerbestand kopieert vervolgens de projectbestanden (dierentuinn.csproj en TestProject2.csproj) naar de map /src en herstelt de afhankelijkheden voor elk project met behulp van .NETt-herstel.

Het Dockerbestand bevat twee debug-stappen om de directory-inhoud van /src/Dierentuinn en /src/TestProject2 weer te geven om te verifiëren dat de bestanden correct zijn gekopieerd.

Vervolgens worden alle bestanden uit de mappen gekopieer Dierentuinn en TestProject2 naar de map /src gekopieerd en bouwt het project dierentuinn.csproj met behulp van dotnet build. De build-uitvoer wordt opgeslagen in de map /app/build.

Test Runner

In het derde stadium wordt de build-fase als basisimage gebruikt en de werkmap ingesteld op /src/TestProject2. Het voert de unit-tests uit met behulp van de .NET-test met de trx-logger.

Final Image

In het laatste stadium wordt de basisimage als basisimage gebruikt en wordt de werkmap ingesteld op /app. Het kopieert de build-uitvoer van de build-fase naar de

/app-map en stelt het toegangspunt in om de Dierentuinn.dll-assembly uit te voeren met behulp van .Net.

Kort samengevat bouwt dit Dockerfile een .NET 8.0 ASP.NET-applicatie, voert het unit-tests uit en creëert een uiteindelijke image die kan worden gebruikt om de applicatie uit te voeren.

Waarom Docker?

Docker wordt veel gebruikt volgens het internet wegens de draagbaarheid, consistentie en schaalbaarheid, wat wordt geboden voor het implementeren van applicaties in verschillende omgevingen. Docker-containers zijn een populaire keuze voor de ontwikkeling en implementatie van moderne applicaties, wegens hun licht gewicht, isolatie en implementatie. Wij kozen voor Docker om de eenvoudige werkwijze, aangezien wij voor het eerst bezig zijn virtualisatie en container technologie.

Unit Tests & Infrastructure Tests

Unit Tests & Infrastructure Tests: Geef details over de opzet en uitvoering van de tests.

Opzet: Unit tests zijn ontworpen om individuele eenheden van de code (meestal functies of methoden) te testen om te verifiëren dat ze correct werken. Voor ons project maken we gebruik van de volgende technologieën en bibliotheken:

1. **xUnit**: Een gratis, open-source, unit testing tool voor .NET.
2. **Moq**: Een populaire mocking bibliotheek voor .NET die ons helpt bij het creëren van mock objecten.
3. **Microsoft.EntityFrameworkCore.InMemory**: Een in-memory database provider die handig is voor het testen van Entity Framework Core applicaties zonder een echte database.

Test Setup: De unit tests zijn ondergebracht in een aparte testproject binnen de oplossing. Het testproject bevat referenties naar de benodigde NuGet-pakketten en het hoofdproject. De belangrijkste pakketten zijn:

- xunit
- xunit.runner.visualstudio
- Moq
- Microsoft.EntityFrameworkCore.InMemory

Toelichting

1. **Mocking DbContext**: Een Mock DbContext wordt ingesteld om de database-interacties te simuleren zonder een echte database te gebruiken.
2. **Sample Data**: Voorbeeldgegevens worden gemaakt om te worden gebruikt in de tests.
3. **Mock DbSet**: Een Mock DbSet wordt ingesteld om te functioneren als de DbSet in de DbContext.
4. **Index Test**: Controleert of de Index-actie een ViewResult teruggeeft met een lijst van Dieren.
5. **Details Test**: Controleert of de Details-actie een ViewResult teruggeeft met het juiste dier, en een NotFoundResult als het dier niet bestaat.
6. **Create Test**: Controleert of de Create-actie een ViewResult teruggeeft.
7. **Edit Test**: Controleert of de Edit-actie een ViewResult teruggeeft met het juiste dier, en een NotFoundResult als het dier niet bestaat.
8. **Delete Test**: Controleert of de Delete-actie een ViewResult teruggeeft met het juiste dier, en een NotFoundResult als het dier niet bestaat.
9. **DeleteConfirmed Test**: Controleert of de DeleteConfirmed-actie een redirect naar de Index-actie doet.

Uitvoering: Unit tests worden uitgevoerd met behulp van het dotnet test-commando. Dit commando bouwt het project en voert de tests uit, waarbij een samenvatting van de resultaten wordt weergegeven.

```
dotnet test
```

Infrastructure Tests

Opzet: Infrastructure tests verifiëren of de infrastructuur en configuraties correct zijn opgezet en werken zoals verwacht. Deze tests kunnen onder andere de integratie met databases, API's, en andere externe systemen omvatten.

Voor ons project gebruiken we meestal integratietests voor deze doeleinden. Deze tests worden uitgevoerd in een realistische omgeving waarin echte gegevensbronnen worden gebruikt, in tegenstelling tot de mocks die in unit tests worden gebruikt.

Test Setup: De infrastructuurtests zijn meestal opgenomen in dezelfde testproject als de unit tests, maar ze vereisen een andere configuratie van de database en andere externe bronnen.

Toelichting

```
public class DatabaseTests
{
    private readonly DierentuinDbContext _context;

    public DatabaseTests()
    {
        var options = new DbContextOptionsBuilder<DierentuinDbContext>()
            .UseSqlServer("YourConnectionString")
            .Options;
        _context = new DierentuinDbContext(options);
    }

    [Fact]
    public async Task CanConnectToDatabase()
    {
        var canConnect = await _context.Database.CanConnectAsync();
        Assert.True(canConnect);
    }
}
```

Uitvoering: Infrastructure tests worden uitgevoerd op dezelfde manier als unit tests, maar kunnen specifieke configuraties of opstartscript vereisen om de benodigde omgevingen op te zetten.

dotnet test

Pipeline Badges

Pipeline Badges: Beschrijf de badges en hoe ze bijdragen aan de inzichtelijkheid van de pipeline status.

De badgeafbeelding toont doorgaans de huidige status van de pijplijn, zoals:

Passeren (groen): De pijpleiding is succesvol voltooid.

Falend (rood): De pijplijn is mislukt.

In behandeling (geel): De pijplijn is momenteel actief.

Onbekend (grijs): De pijplijnstatus is onbekend of niet beschikbaar.

Door een pijplijnbadge in het proces op te nemen, wordt er een snelle visuele indicatie gegeven van de status van de pijplijn, waardoor het voor anderen, die ook werken aan het project, gemakkelijker wordt om de bouw- en implementatiestatus van het project te begrijpen.

Code Quality & Analysis

Code Quality & Analysis: Dit onderdeel richt zich op het waarborgen van de kwaliteit en veiligheid van de code door verschillende technieken en tools toe te passen. Hieronder worden de gebruikte tools en technieken beschreven:

Statische Code-analyse

Beschrijving: Een methode waarbij de broncode automatisch wordt geanalyseerd zonder deze uit te voeren, om potentiële fouten, kwetsbaarheden en code smells te identificeren.

Tools: SonarQube, ESLint, PMD.

Voordelen:

- Vroegtijdige opsporing van problemen in de ontwikkelfase.
- Verbetering van de algehele codekwaliteit.
- Verhoogde beveiliging door het detecteren van kwetsbaarheden.
- Bevordering van best practices en code consistentie.

Code Reviews

Beschrijving: Handmatige beoordelingen van de code door collega's (peer reviews) om fouten op te sporen, de leesbaarheid te verbeteren en best practices te waarborgen.

Tools: GitHub pull requests, GitLab merge requests.

Voordelen:

- Detectie van problemen die mogelijk niet door geautomatiseerde tools worden opgemerkt.
- Verbetering van de kennisdeling binnen het team.
- Verhoging van de codekwaliteit door feedback en samenwerking.
- Bevordering van teamstandaarden en best practices.

Continuous Integration (CI)

Beschrijving: Een ontwikkelpraktijk waarbij ontwikkelaars regelmatig code wijzigingen integreren in een gedeelde repository, gevolgd door geautomatiseerde builds en tests. Dit helpt om problemen snel te detecteren en op te lossen.

Tools: GitLab CI.

Voordelen:

- Snelle detectie van integratieproblemen.
- Vermindering van de tijd tussen het maken van wijzigingen en het ontdekken van fouten.
- Automatisering van het build- en testproces, wat de efficiëntie verhoogt.

Continuous Deployment/Continuous Delivery (CD)

Beschrijving: CD-tools automatiseren het proces van het implementeren van code-wijzigingen naar productieomgevingen na succesvolle testen en validatie.

Tools: GitLab CI/CD, Azure DevOps Pipelines.

Voordelen:

- Consistente en betrouwbare implementaties.
- Vermindering van handmatige fouten door automatisering.
- Snelle releasecycli en verbeterde time-to-market.

Unit Tests

Beschrijving: Geautomatiseerde tests die individuele eenheden van de broncode testen, zoals functies of methoden, om ervoor te zorgen dat ze correct werken.

Tools: Frameworks zoals JUnit (voor Java), NUnit (voor .NET), PyTest (voor Python).

Voordelen:

- Detectie van fouten in een vroeg stadium van ontwikkeling.
- Verbetering van de codebetrouwbaarheid en stabiliteit.
- Facilitering van refactoring door bestaande functionaliteit te waarborgen.

Security Tools

Beschrijving: Tools die helpen bij het identificeren en verminderen van beveiligingskwetsbaarheden in de ontwikkeling van software.

Tools: Spectral.

Voordelen:

- Verhoogde veiligheid door vroegtijdige detectie van kwetsbaarheden.
- Bescherming tegen aanvallen en misbruik.
- Naleving van beveiligingsstandaarden en best practices.

Monitoring and Logging Tools

Beschrijving: Tools die inzicht bieden in de prestaties, beschikbaarheid en het gedrag van applicaties en infrastructuur, waardoor problemen snel kunnen worden geïdentificeerd en opgelost.

Tools: Prometheus.

Voordelen:

Continu inzicht in de systeemprestaties.

Vroegtijdige detectie van problemen en afwijkingen.

Verbeterde beschikbaarheid en betrouwbaarheid van de applicaties.

Release Management

Release Management: Release management richt zich op het plannen, beheren en controleren van de software build door de verschillende stadia van ontwikkeling, testing en uiteindelijk productie. Dit proces zorgt ervoor dat nieuwe softwareversies betrouwbaar en efficiënt naar de eindgebruikers worden gebracht. Hieronder wordt beschreven hoe releases worden beheerd en gedistribueerd via een registry:

Beheer van Releases via een Registry

Versiebeheer en Tagging:

Beschrijving: Bij elke nieuwe release wordt de code in de versiebeheertool (zoals Git) getagd met een specifieke versie (bijv. v1.0.0). Dit zorgt voor een duidelijke identificatie van de exacte code die in de release zit.

Tools: Git (GitHub, GitLab).

Voordelen:

- Eenvoudige terugverwijzing naar specifieke versies.
- Betere traceerbaarheid en controle over de verschillende releases.
- Mogelijkheid om snel terug te keren naar een vorige versie indien nodig.

Build en Packaging:

Beschrijving: De broncode wordt gecompileerd en verpakt in een distributieformaat zoals Docker images. Dit gebeurt meestal in de CI/CD-pijplijn.

Tools: Docker, GitLab CI/CD, Azure DevOps Pipelines.

Voordelen:

- Consistente en herhaalbare builds.
- Gemakkelijke distributie en implementatie over verschillende omgevingen.
- Verpakking van alle afhankelijkheden binnen de container zorgt voor een consistente runtime-omgeving.

Publicatie naar een Registry:

Beschrijving: De verpakte applicatie (bijv. Docker image) wordt gepusht naar een container registry zoals Docker Hub, GitHub Packages of een privé registry.

Tools: Docker Hub, GitHub Packages, Azure Container Registry.

Voordelen:

- Centrale locatie voor het opslaan en beheren van builds en artefacten.
- Eenvoudige toegang en distributie naar verschillende omgevingen.
- Ondersteuning voor versiebeheer en rollbacks.

Distributie en Implementatie:

Beschrijving: Vanaf de registry kunnen de images worden gedownload en gedeployed naar verschillende omgevingen zoals development, staging en productie.

Tools: Kubernetes, Docker Swarm, Azure Kubernetes Service (AKS).

Voordelen:

- Gestandaardiseerde implementaties door het gebruik van containers.
- Schaalbaarheid en flexibiliteit bij het implementeren van applicaties.
- Eenvoudig beheer en monitoring van gecontaineriseerde applicaties.

Stappen in het Releaseproces via een Registry

Code Tagging:

Ontwikkelaars taggen de release versie in de code repository (bijv. git tag v1.0.0).

Build Process:

- De CI/CD-pijplijn triggert een build van de code.
- De code wordt gecompileerd, getest en verpakt als een Docker image.

Push naar Registry:

De Docker image wordt gepusht naar een container registry met een specifieke tag (bijv. docker push myregistry/myapp:v1.0.0).

Deployment:

De gecontaineriseerde applicatie wordt vanaf de registry gedownload en geïmplementeerd in de doelomgeving (bijv. staging of productie) met behulp van orkestratietools zoals Kubernetes.

Monitoring en Validatie:

- Na implementatie wordt de applicatie gemonitord om de prestaties en stabiliteit te waarborgen.
- Eventuele problemen worden snel geïdentificeerd en opgelost.

HealthCheck & Monitoring:

HealthCheck & Monitoring: Dit onderdeel richt zich op het waarborgen van de beschikbaarheid, prestaties en betrouwbaarheid van applicaties door middel van health checks en monitoring dashboards. Voor ons project gebruiken we tools zoals Prometheus en Azure DevOps voor monitoring en logging, en we implementeren health checks binnen onze Docker-gebaseerde omgeving.

Implementatie van Health Checks

Health checks zijn geautomatiseerde tests die periodiek worden uitgevoerd om de status en gezondheid van een applicatie te controleren. Ze helpen om snel problemen te identificeren en te reageren voordat deze de gebruikerservaring beïnvloeden.

Health Check Endpoints:

Beschrijving: Voeg specifieke endpoints toe aan de applicatie die informatie geven over de gezondheid van verschillende onderdelen, zoals de database, externe API's, en andere afhankelijkheden.

Implementatie:

- Voor een webapplicatie kan een endpoint zoals /health of /status worden geconfigureerd.
- De endpoint moet controleren of alle cruciale componenten van de applicatie correct werken en een statusrapport retourneren.

Tools: Voor onze .NET-applicatie gebruiken we ASP.NET Core Health Checks.

Periodieke Health Checks:

Beschrijving: Configureer tools die periodiek de health check endpoints aanroepen om de status van de applicatie te controleren.

Implementatie:

- Configureer Kubernetes liveness en readiness probes die de health check endpoints periodiek aanroepen.

Tools: Kubernetes, Docker Swarm.

Alerting:

Beschrijving: Stel waarschuwingen in om meldingen te ontvangen wanneer een health check faalt.

Implementatie:

- Integreer Prometheus Alertmanager met health check monitoring om alerts te configureren en verzenden.

Tools: Prometheus Alertmanager.

Implementatie van Monitoring Dashboards

Monitoring dashboards geven realtime inzicht in de prestaties, beschikbaarheid en het gedrag van applicaties en infrastructuur. Ze helpen bij het visualiseren van belangrijke metrics en trends om proactief problemen te identificeren en op te lossen.

Verzamelen van Metrics:

Beschrijving: Verzamelen van belangrijke prestatie- en gebruiksmetrics van de applicatie en infrastructuur.

Implementatie:

- Integreer Prometheus om metrics te verzamelen zoals CPU-gebruik, geheugenverbruik, responstijden en foutpercentages.
- Configureer applicatie logging om specifieke gebeurtenissen en fouten te volgen.

Tools: Prometheus.

Configureren van Dashboards:

Beschrijving: Maak visuele dashboards die de verzamelde metrics presenteren in een bruikbaar formaat.

Implementatie:

- Gebruik Grafana om aangepaste visualisaties te maken zoals grafieken, meters en tabellen.
- Configureer dashboards om belangrijke KPI's en trends te tonen.

Tools: Grafana.

Voorbeeld:

- **Stap 1:** Verbind Grafana met de Prometheus-database waar de metrics worden opgeslagen.
- **Stap 2:** Maak een nieuw dashboard en voeg panelen toe voor verschillende metrics zoals CPU-gebruik, geheugenverbruik en responstijden.
- **Stap 3:** Configureer alerts in Grafana om meldingen te ontvangen wanneer bepaalde drempels worden overschreden.

Alerting en Incident Management:

Beschrijving: Stel waarschuwingen en meldingen in op basis van de verzamelde metrics om snel te kunnen reageren op problemen.

Implementatie:

- Configureer alerts op dashboards om meldingen te verzenden bij overschrijding van bepaalde drempels.
- Integreer met incident management tools om waarschuwingen door te sturen naar verantwoordelijke teams.

Tools: Grafana Alerts, Prometheus Alertmanager, Azure DevOps

Peerreview Feedback

Peerreview Feedback: Geeft voorbeelden van ontvangen feedback en hoe deze zijn verwerkt.

Feedback

Index Test Feedback:

Positief: De test controleert effectief of de Index-actie een lijst van dieren retourneert.

Verbetering: Voeg meer dieren toe aan de testdata om de robuustheid van de test te verbeteren.

Details Test Feedback:

Positief: De test valideert correct de terugkeer van het juiste dier en de afhandeling van niet-bestaande dieren.

Verbetering: Test met verschillende geldige en ongeldige ID's om de betrouwbaarheid te vergroten.

Create Test Feedback:

Positief: De test zorgt ervoor dat de Create-actie een ViewResult retourneert.

Verbetering: Voeg tests toe voor het POST-gedeelte van de Create-actie om volledigheid te garanderen.

Edit Test Feedback:

Positief: De test valideert de terugkeer van het juiste dier en de afhandeling van niet-bestaande dieren.

Verbetering: Controleer ook de aanpassing van diergegevens na het bewerken.

Delete Test Feedback:

Positief: De test valideert de terugkeer van het juiste dier en de afhandeling van niet-bestaande dieren.

Verbetering: Voeg tests toe voor het daadwerkelijk verwijderen van dieren uit de database.

DeleteConfirmed Test Feedback:

Positief: De test controleert effectief de redirect naar de Index-actie na verwijdering.

Verbetering: Test ook of het dier daadwerkelijk uit de database is verwijderd na bevestiging.