

Stage CICD API Wilms NV

Realisatiedocument

Anthony Van Roy
Student Bachelor in de Elektronica-ICT – Cloud & Cyber Security

Inhoudsopgave

FIGURENLIJST	4
TABELLENLIJST	6
1. INLEIDING	7
2. ANALYSE	8
2.1. Waarom een CI/CD-tool	8
2.1.1. De huidige situatie en context	8
2.2. Doelstellingen	9
2.3. Onderzochte tooling/software	9
2.3.1. Kestra	10
2.3.2. Ansible & Ansible-runner	11
2.3.3. Puppet	11
2.3.4. FastAPI	12
2.3.5. Docker Swarm	12
2.3.6. Kubernetes	13
2.3.7. Docker Compose	13
2.3.8. Azure DevOps Pipelines	14
2.3.9. Infisical	14
2.3.10. OpenBao	15
2.4. Vergelijkingen tools	16
2.4.1. Vergelijking van container/orchestrator tools	16
2.4.2. Vergelijking van configuratiemanagement tools	17
2.4.3. Vergelijking van CI/CD Oplossingen	18
2.4.4. Vergelijking van Secret Management tools	19
2.5. Onderzochte architectuuropties	20
2.6. Verantwoording van de keuze	21
3. REALISATIE	22
3.1. Onsite Azure Devops pipelines	22
3.2. Centraal beheer van config met Openbao	24
3.2.1. Installatie	24
3.2.2. Secret structuur	28
3.3. Builden van containers via Azure Devops	29
3.3.1. Semantic versioning met GitVersion	30
3.3.2. Build trigger	32
3.3.3. Automatische version bump	35
3.4. Custom API	36
3.4.1. Het databasemodel	37
3.4.2. Pydantic modellen	38
3.4.3. Login	40
3.4.4. Token managers	42
3.4.5. Ansible playbooks	44
3.4.6. Logging	55

3.4.7. Metrics	56
3.4.8. Endpoints	58
3.5. Documentatie	60
4. BESLUIT	65
LITERATUURLIJST	66

Figurenlijst

Figuur 3-1 dockerfile azure pipeline agent	23
Figuur 3-2 dockerfile Openbao	24
Figuur 3-3 configuratiebestand Openbao	25
Figuur 3-4 docker-compose bestand Openbao	26
Figuur 3-5 Instellen secret engine Openbao	26
Figuur 3-6 Instellen ACL policy Openbao	27
Figuur 3-7 Gitversion.yml bestand	30
Figuur 3-8 Azure-pipelines.yml bestand	31
Figuur 3-9 Azure-pipelines.yml bestand	32
Figuur 3-10 Azure-pipelines.yml bestand	33
Figuur 3-11 Azure-pipelines.yml bestand	33
Figuur 3-12 Azure-pipelines.yml bestand	34
Figuur 3-13 Azure-pipelines.yml bestand	34
Figuur 3-14 Azure-pipelines.yml bestand	35
Figuur 3-15 ERD diagram database	37
Figuur 3-16 Pydantic model AddSecretRequest	38
Figuur 3-17 Pydantic AddSecretRequest voorbeeld	38
Figuur 3-18 Pydantic JobResponse model	39
Figuur 3-19 /jobs endpoint response body	39
Figuur 3-20 Unauthenticated request	40
Figuur 3-21 Login page swagger page API	41
Figuur 3-22 Authenticated request	41
Figuur 3-23 Inspectie JWT token	42
Figuur 3-24 auth.py azuretokenmanager	43
Figuur 3-25 auth.py azuretokenmanager	43
Figuur 3-26 Ansible playbook build_containers.yml	44
Figuur 3-27 Ansible playbook build_containers.yml	45
Figuur 3-28 Ansible playbook build_containers.yml	45
Figuur 3-29 Ansible playbook deploy_containers.yml	46
Figuur 3-30 Ansible playbook deploy_containers.yml	46
Figuur 3-31 Ansible playbook deploy_containers.yml	47
Figuur 3-32 Ansible playbook deploy_containers.yml	48
Figuur 3-33 Ansible playbook deploy_containers.yml	48
Figuur 3-34 Ansible playbook deploy_containers.yml	49
Figuur 3-35 Ansible playbook deploy_containers.yml	49
Figuur 3-36 Ansible playbook task_traefik_attach_and_change.yml	50
Figuur 3-37 Ansible playbook task_traefik_attach_and_change.yml	51
Figuur 3-38 Ansible playbook task_traefik_attach_and_change.yml	52
Figuur 3-39 Ansible playbook deploy_containers_dmz.yml	53

Figuur 3-40 Ansible playbook deploy_containers_dmz.yml	54
Figuur 3-41 API main.py	55
Figuur 3-42 Metrics job status gauge	56
Figuur 3-43 Metrics job duration bucket	56
Figuur 3-44 Metrics jobs started counter	56
Figuur 3-45 Metrics jobs in progress gauge	56
Figuur 3-46 Metrics endpoint	57
Figuur 3-47 Swagger user categorie	58
Figuur 3-48 Swagger runners categorie	58
Figuur 3-49 Swagger jobs categorie	58
Figuur 3-50 Swagger containers categorie	59
Figuur 3-51 Swagger apps categorie	59
Figuur 3-52 Swagger secrets categorie	59
Figuur 3-53 Swagger azure categorie	59
Figuur 3-54 Swagger metrics categorie	59
Figuur 3-55 Swagger utility categorie	60
Figuur 3-56 API readme	60
Figuur 3-57 API readme inhoudstafel	61
Figuur 3-58 API readme prerequisites	61
Figuur 3-59 API readme python packages	62
Figuur 3-60 API readme env variabelen	63
Figuur 3-61 API readme build flow	63
Figuur 3-62 API readme deploy flow	64

Tabellenlijst

Tabel 2-1 WRM container/orchistrator tools	16
Tabel 2-2 WRM configuratiemanagement tools	17
Tabel 2-3 CI/CD Oplossingen	18
Tabel 2-4 Secret Management tools	19

1. Inleiding

In het kader van mijn professionele bacheloropleiding Elektronica-ICT, met afstudeerrichting Cloud & Cybersecurity, heb ik stage gelopen bij het bedrijf Wilms NV.

Wilms NV is een producent van rolluiken, zonwering en, recentelijk, ook ventilatiesystemen. Het bedrijf beschikt over een uitgebreid intern IT-team, bestaande uit specialisten in IT-infrastructuur, support, development en externe developers.

Dit realisatiedocument is een uitwerking van het projectplan voor de ontwikkeling van een tool bij Wilms, waarmee software builds en deployments binnen de CI/CD-flow mogelijk worden gemaakt.

In het projectplan werd de huidige situatie beschreven, waarbij software grotendeels draait op bare metal en de deployment handmatig wordt uitgevoerd. Dit leidt tot miscommunicatie, fouten, tragere oplevering en inefficiëntie.

De doelstelling van dit project was om deze processen te moderniseren en automatiseren aan de hand van open source, schaalbare technologieën die passen binnen de evolutie naar containerization.

Dit document behandelt de uitvoering van het onderzoek en de technische realisatie van de oplossing. Er wordt toegelicht hoe een keuze is gemaakt tussen verschillende oplossingsrichtingen en tools, en welke realisaties en functionaliteiten zijn behaald.

De uiteindelijke oplossing bestaat uit een combinatie van een custom frontend en een backend-API, gebouwd in FastAPI, die Ansible Runner aanstuurt.

Volgende onderdelen komen aan bod in dit realisatiedocument:

- Een analyse van de huidige situatie en mogelijke architecturen en tooling (waaronder Kestra, Ansible en OpenBao), met verantwoording van de gemaakte keuzes.
- De realisatie van de infrastructuurautomatisatie-oplossing, inclusief:
 - de opbouw van de API en frontend,
 - de werking van build en deployprocessen,
 - en de integratie met monitoring en logging.
- Een evaluatie van de werking ten opzichte van de oorspronkelijke projectdoelstellingen.
- Een besluit met reflecties, aanbevelingen en suggesties voor toekomstige uitbreidingen.

Tot slot wordt verwezen naar ondersteunend materiaal in de bijlagen, zoals gebruikte schema's, screenshots en de geraadpleegde literatuur.

2. Analyse

Om een goede oplossing te kunnen bouwen, is een analyse van de huidige situatie noodzakelijk. Er moet onderzoek worden gedaan naar mogelijke oplossingsrichtingen en de tools die deze kunnen ondersteunen.

In dit hoofdstuk wordt de huidige situatie bij Wilms NV toegelicht en wordt besproken waarom een CI/CD-tool binnen Wilms aanzienlijke voordelen kan bieden. Ook worden de verschillende tools die ik onderzocht heb besproken, evenals de uiteindelijke keuze van de tools en hun rol binnen de oplossing.

Het doel van dit hoofdstuk is om op een overzichtelijke manier te tonen:

- waarom een CI/CD-tool nodig is,
- Welke tools onderzocht zijn,
- wat elke tool doet,
- welke functionaliteiten ze binnen het project bieden,
- en waarom deze keuzes geschikt zijn voor de beoogde oplossing.

2.1. Waarom een CI/CD-tool

Om een goede analyse te kunnen maken, moet eerst gekeken worden naar de huidige situatie, de bijbehorende problemen, en hoe een CI/CD-tool hierbij kan helpen.

De aanleiding voor deze opdracht ligt in de huidige werkwijze van het bedrijf. Wilms NV is een productiebedrijf met ongeveer 200 werknemers en een uitgebreid intern IT-team. Momenteel telt het bedrijf drie interne applicatiedevelopers, twee interne developers voor de productconfigurator en twee interne IT-medewerkers voor infrastructuur en support, aangevuld met externe ontwikkelaars.

Binnen dit team vinden gemiddeld drie applicatie-updates per week plaats. Deze updates vereisen telkens nauwe samenwerking tussen het ontwikkelteam en het infrastructuurteam.

De laatste jaren maakt Wilms NV een duidelijke shift richting containerisatie. Momenteel draaien er al zo'n 80 interne applicaties in containers. Deze evolutie naar een moderne infrastructuur brengt heel wat voordelen met zich mee, maar stelt ook extra eisen aan het releaseproces.

(Poberezhnyk, 2024)

2.1.1. De huidige situatie en context

Na het ontwikkelen van nieuwe functionaliteit moet de code worden overgedragen aan het infrastructuurteam. Deze overdracht gebeurt momenteel handmatig en vereist overleg tussen beide teams. De containerimages worden gebouwd met behulp van handmatige Docker-commando's, gepusht naar interne containerregistries en vervolgens gedeployed via SSH en command-line interfaces op de juiste servers.

Deze manier van werken brengt enkele problemen met zich mee:

- **Tijdverlies:** de handmatige processen zorgen ervoor dat een relatief eenvoudige wijziging toch veel tijd in beslag kan nemen.
- **Foutgevoeligheid:** handmatige tussenstappen verhogen het risico op menselijke fouten, zoals het pushen naar de verkeerde registry of het deployen van een foutieve image.

- **Frustraties binnen het team:** doordat het proces niet gestroomlijnd is, moeten developers en infrastructuurmedewerkers voortdurend heen en weer communiceren, wat leidt tot frustraties en inefficiënties.

2.2. Doelstellingen

Voor we met de tooling aan de slag kunnen, moeten eerst de doelstellingen duidelijk worden gedefinieerd. Dit is belangrijk, omdat deze doelstellingen in grote mate bepalen welke tools we kunnen en mogen gebruiken binnen het project.

Het hoofddoel is het automatiseren van het build- en deploymentproces van applicaties die binnen Wilms draaien.

De aanleiding hiervoor is de huidige situatie, waarin:

- ongeveer 90% van de software op bare metal draait;
- deployment handmatig gebeurt;
- miscommunicatie tussen teams regelmatig leidt tot fouten en downtime.

De oplossing moet bestaan uit een tool die:

- veilig is opgezet (*security by design*);
- modulair, uitbreidbaar, aanpasbaar en bij voorkeur eenvoudig is;
- integratie met API's en andere tools ondersteunt;
- gebaseerd is op open source technologieën, om vendor lock-in en onverwachte licentiekosten te vermijden;
- op een veilige manier kan omgaan met secrets zoals applicatiewachtwoorden en configuratiegegevens;
- bij voorkeur zoveel mogelijk gebruikmaakt van de lokale infrastructuur, en alleen cloud resources inzet wanneer dit noodzakelijk is.

Het feit dat de gebruikte tools bij voorkeur open source, modulair en aanpasbaar moeten zijn, betekent dat we gericht moeten zoeken naar geschikte, gespecialiseerde tooling.

2.3. Onderzochte tooling/software

Nu de doelstellingen zijn vastgesteld, kan er gericht gezocht worden naar geschikte tooling voor het project. In het kader van dit project zijn verschillende tools onderzocht. De volgende tools zullen in dit document specifiek worden besproken:

- **Kestra:** een declaratieve open-source orchestration-tool,
- **Ansible / Ansible Runner:** een open-source, agentless automatiserings- en configuratietool,
- **Puppet:** een open-source configuratiemanagementtool met een agent-based master-servermodel,
- **FastAPI:** een op Python gebaseerd API-framework,
- **Docker Swarm:** een open-source, cluster-native containerorkestratietool,
- **Kubernetes:** een open-source, cluster-native containerorkestratietool,
- **Docker Compose:** een uitbreidingsmodule bovenop Docker voor het definiëren van containerstacks,
- **Azure DevOps Pipelines:** het CI/CD-component binnen Azure DevOps,
- **Infisical:** een freemium open-source platform voor het beheren van secrets,
- **OpenBao:** een open-source fork van de HashiCorp Vault secret manager.

2.3.1. Kestra

Kestra is een open-source orkestratietool voor het creëren van modulaire automatiseringsflows, gericht op dataverwerking en het automatiseren van taken.

Kestra werd relatief vroeg in het project geïntroduceerd en was daarmee een van de eerste tools die getest werd. Kestra biedt verschillende voordelen: het is open-source, werkt met modules die in elke flow geplaatst kunnen worden, is modulair opgebouwd en beschikt over een gebruiksvriendelijke frontend waarin flows geprogrammeerd en beheerd kunnen worden.

De uitgebreide featureset van Kestra maakte het mogelijk om vrijwel het gehele project binnen Kestra op te bouwen. Kestra beschikt over een frontend, een uitgebreide API waarmee flows gestart kunnen worden, een modulaire aanpak voor het maken van automatiseringen, en native ondersteuning voor het bouwen van containers.

Toch zijn er een aantal nadelen die ertoe geleid hebben dat Kestra niet verder in het project is gebruikt:

- **Freemium model:**
Kestra hanteert een freemium model. Hoewel het open-source is, zijn bepaalde functionaliteiten achter een betaald model geplaatst, waaronder belangrijke beveiligingsfuncties zoals tweefactorauthenticatie (2FA), het scheiden van resources per project, een dashboard en frontend triggers.
- **Gebrek aan kennis binnen het team:**
Kestra is onbekend binnen het Wilms-team, waardoor extra inspanningen nodig zijn om het team te trainen en up-to-date te brengen met Kestra, zodat het in de toekomst ondersteund en gebruikt kan worden.
- **Complexiteit van de codebase:**
Hoewel Kestra het maken van automatiseringen vergemakkelijkt, is de onderliggende codebase omvangrijk en complex. Dit betekent dat bij problemen diepgaande kennis van Kestra vereist is om deze op te lossen, of dat er een recovery plan moet zijn om Kestra weer operationeel te krijgen.
- **Support en kosten:**
Vanwege de complexiteit en het gebrek aan interne kennis is ondersteuning noodzakelijk. Kestra biedt support via hun enterprise-subscriptiemodel, inclusief toegang tot de volledige feature set. Na overleg met het sales-team bleek deze optie voor Wilms financieel niet haalbaar en bood het geen duidelijke meerwaarde ten opzichte van de huidige werkwijze.

Conclusie

Kestra is een krachtig en modulair open-source product met veel mogelijkheden. Echter, de onderliggende complexiteit, het freemium model en de onbekendheid binnen het team maken het moeilijk om Kestra aan te raden of verder te gebruiken binnen Wilms.

Kestra heeft wel waardevolle inzichten gegeven in hoe een dergelijke tool eruit kan zien en welke features in andere tooling gewenst zijn.

2.3.2. Ansible & Ansible-runner

Ansible is een open-source automatiseringstool die voornamelijk wordt ingezet voor het configureren van systemen, het uitrollen van software en het beheren van infrastructuur op een schaalbare, herhaalbare en idempotente manier. Het werkt agentless via SSH en maakt gebruik van YAML-gebaseerde playbooks om taken declaratief te beschrijven. Hierdoor is het eenvoudig leesbaar en toegankelijk voor zowel ontwikkelaars als systeembeheerders. Ansible beschikt over een uitgebreide set modules voor uiteenlopende taken en maakt gebruik van een inventory-systeem om hosts te beheren en te groeperen in inventory bestanden.

Ansible Runner is een aanvullend component dat speciaal ontwikkeld is om Ansible-taken programmatisch aan te roepen vanuit externe applicaties of automatiseringspijplijnen. Het biedt een gestandaardiseerde interface, inclusief een command-line interface (CLI) en een Python SDK, om Ansible-instructies te activeren, monitoren en loggen. De goed gedocumenteerde Python SDK was een belangrijke reden om Ansible Runner te overwegen in dit project, omdat dit naadloos aansluit bij de bestaande Python-gebaseerde infrastructuur.

Tijdens het project werd Ansible geëvalueerd als een potentiële oplossing voor het automatisch beheren van infrastructuur en het provisioneren van virtuele omgevingen in test- en ontwikkelomgevingen. Dankzij de lage instapdrempel, duidelijke syntaxis, idempotentie en brede community-ondersteuning kon snel een proof-of-concept worden uitgewerkt. De eenvoudige integratie met Git-gebaseerde versiecontrole en de mogelijkheid om playbooks modulair op te bouwen, maakten het aantrekkelijk om configuratieversies snel op te stellen en iteratief te verbeteren.

(“Ansible Runner — Ansible Runner Documentation”, z.d.)

2.3.3. Puppet

Puppet is een open-source configuratiebeheertool die werkt volgens het principe van “infrastructure as code”. Het gebruikt een declaratieve taal (een eigen domeinspecifieke taal, DSL) waarmee systeemconfiguraties worden gedefinieerd. Puppet volgt een agent-based model waarbij een centrale server, de Puppet master, instructies verspreidt naar Puppet agents die op de beheerde machines zijn geïnstalleerd.

Puppet werd onderzocht als alternatief voor tools zoals Ansible vanwege de mogelijkheid tot geautomatiseerde, herhaalbare en consistente configuraties binnen een omgeving. Het systeem werkt op basis van zogenaamde manifests, bestanden waarin wordt aangegeven hoe systemen geconfigureerd moeten zijn. Puppet voert deze configuraties periodiek uit op de beheerde nodes, wat zorgt voor voortdurende naleving van de gewenste staat.

Voordelen van Puppet zijn onder andere de schaalbaarheid, het volwassen ecosysteem met uitgebreide modules en de sterke focus op compliance en rapportering. Het agent-based model zorgt ervoor dat de beheerde nodes actief contact onderhouden met de master, wat een robuuste configuratiehandhaving mogelijk maakt. Echter, het installeren en beheren van agents voegt een extra beheerlaag toe. Daarnaast is de leercurve iets steiler door het gebruik van de eigen DSL.

In de context van dit project werd Puppet minder interessant vanwege deze extra complexiteit en omdat het team al ervaring heeft met andere tools zoals Ansible, die een agentless model hanteren en daardoor eenvoudiger in beheer zijn.

(“Introduction To Puppet”, z.d.)

2.3.4. FastAPI

FastAPI is een modern, op Python gebaseerd webframework waarmee snel en efficiënt API's gebouwd kunnen worden. Het framework is ontworpen met het oog op snelheid, eenvoud en gebruiksvriendelijkheid. Het maakt intensief gebruik van Python type hints, wat automatische validatie, foutafhandeling en het genereren van OpenAPI-documentatie mogelijk maakt.

Binnen dit project werd FastAPI ingezet als orchestrator of tussenlaag die externe services en automatiseringstaken aanstuurt. Dankzij de ondersteuning voor asynchrone verwerking (via asyncio) kan het framework meerdere taken gelijktijdig afhandelen zonder de prestaties negatief te beïnvloeden.

Belangrijke voordelen van FastAPI in dit project zijn:

- **Automatische OpenAPI-documentatie:** Door het gebruik van type hints en Pydantic-modellen wordt automatisch gedetailleerde API-documentatie gegenereerd, wat de integratie met externe systemen vergemakkelijkt.
- **Asynchrone verwerking:** Complexe automatiseringsworkflows kunnen gelijktijdig en efficiënt worden uitgevoerd.
- **Python-integratie:** Omdat het team al ervaring had met Python, was de leercurve zeer laag.
- **Snelheid:** Door de onderliggende implementatie met Starlette (voor de webserver) en Pydantic (voor data-validatie) behoort FastAPI tot de snelste Python-gebaseerde API-frameworks.

("FastAPI", z.d.)

2.3.5. Docker Swarm

Docker Swarm is een native containerorkestratie-oplossing die ingebouwd is in Docker. Het stelt gebruikers in staat om meerdere Docker-hosts te clusteren en als één enkele virtuele Docker-host te beheren, waarbij containers automatisch over beschikbare nodes worden verspreid.

Binnen dit project werd Docker Swarm onderzocht als een relatief eenvoudige en snel te implementeren oplossing voor het beheren van container workloads, zonder de complexiteit van uitgebreidere systemen zoals Kubernetes.

Voordelen van Docker Swarm:

- **Eenvoudige setup en beheer:** Snelle installatie zonder nood aan extra tooling.
- **Naadloze integratie met Docker CLI:** Bekend en vertrouwd werkmodel voor teams die al met Docker werken.
- **Automatische load balancing:** Containers worden automatisch verdeeld over beschikbare nodes voor betere resourcebenutting.
- **Geschikt voor kleinere tot middelgrote clusters,** waar eenvoud en snelheid belangrijk zijn.

Beperkingen:

- **Minder uitgebreide functionaliteit:** In vergelijking met Kubernetes biedt Docker Swarm een beperkter scala aan features, zoals geavanceerde scheduling, zelfherstelmechanismen en uitgebreide networking-opties.
- **Kleinere community en minder actieve ontwikkeling:** Kubernetes heeft een veel grotere en actiever community-ondersteuning, waardoor het vaak de voorkeur krijgt voor productiesystemen.
- **Schaalbaarheid:** Docker Swarm is minder geschikt voor zeer grote, complexe omgevingen.

2.3.6. Kubernetes

Kubernetes is een zeer krachtige en uitgebreide open-source containerorkestratietool, ontworpen voor het draaien en beheren van grootschalige, gedistribueerde containerapplicaties.

Het biedt uitgebreide functionaliteiten zoals automatische schaalvergroting (autoscaling), zelfherstel van containers, rolling updates zonder downtime, geavanceerd secrets- en configuratiemanagement, en een declaratieve aanpak voor infrastructuurbeheer via YAML-manifesten.

Binnen het project werd Kubernetes onderzocht als een enterprise-grade oplossing, geschikt voor complexere workloads en omgevingen met hoge beschikbaarheid en schaalbaarheidseisen.

Voordelen van Kubernetes:

- **Schaalbaarheid en zelfherstel:** Kubernetes kan automatisch containers (pods) opnieuw opstarten of herplannen bij falen, en schaaft workloads dynamisch op basis van de belasting.
- **Rijk ecosysteem:** Het heeft een enorm ecosysteem van plugins, extensies en integraties, met sterke enterprise-ondersteuning van grote cloudproviders en softwareleveranciers.
- **Beheer van configuraties en secrets:** Kubernetes maakt gebruik van declaratieve YAML-configuraties, waardoor infrastructuur als code wordt beheerd. Het biedt bovendien ingebouwde mechanismen voor het veilig beheren van gevoelige gegevens.
- **Rollende updates en rollback:** Zonder downtime kunnen applicaties geüpdatet worden, met de mogelijkheid om snel terug te keren naar een vorige versie.

Beperkingen:

- **Complexiteit:** Kubernetes heeft een steile leercurve en vereist aanzienlijke expertise om goed te kunnen beheren en optimaliseren.
- **Resource footprint:** Kubernetes-clusters hebben een hogere infrastructuur- en resourcebelasting in vergelijking met lichtere orkestratietools zoals Docker Swarm.

2.3.7. Docker Compose

Docker Compose is een tool waarmee multi-container Docker-applicaties kunnen worden gedefinieerd en beheerd via eenvoudige YAML-bestanden. Het biedt een praktische en toegankelijke manier om applicatiestacks lokaal op te zetten, te testen en te beheren.

Binnen het project werd Docker Compose ingezet voor het lokaal uitrollen van testomgevingen en het snel provisioneren van volledige stackconfiguraties tijdens de ontwikkelfase.

Voordelen van Docker Compose:

- **Snelheid:** Maakt het mogelijk om complete applicatiestacks snel op te starten met één commando.
- **Eenvoudig beheer:** YAML-configuraties zijn makkelijk leesbaar, versiebeheerbaar en aanpasbaar.
- **Veelzijdigheid:** Breed toepasbaar voor zowel lokale ontwikkeling als integratie in CI/CD-pijplijnen.

Beperkingen:

- **Niet geschikt voor productie:** Docker Compose mist ingebouwde functies voor schaalbaarheid, load balancing en hoge beschikbaarheid die essentieel zijn voor productieomgevingen. Waardoor het niet altijd de beste keuze is.
- **Beperkte orkestratiefuncties:** Het is niet ontworpen voor het beheren van geclusterde of gedistribueerde omgevingen zoals Kubernetes of Docker Swarm dat kunnen.

2.3.8. Azure DevOps Pipelines

Azure DevOps Pipelines is het CI/CD-component van Microsoft Azure DevOps, waarmee geautomatiseerde build- en releasepijplijnen kunnen worden ontworpen, gebouwd en beheerd.

Binnen dit project werd Azure DevOps Pipelines ingezet voor het opzetten van automatisatiepijplijnen die verantwoordelijk zijn voor code deployments, infrastructuurprovisioning (via Ansible) en het uitvoeren van geautomatiseerde tests.

Voordelen:

- **Integratie met Azure:** Naadloze koppeling met andere Azure-services, wat zorgt voor een consistente workflow in de Azure-cloudomgeving.
- **Breed ecosysteem:** Beschikbaarheid van talrijke extensies en plugins om functionaliteit uit te breiden en te integreren met verschillende tools en platforms.
- **Flexibele pipelines:** YAML-gebaseerde pipelines zorgen voor overzichtelijke, versiebeheerbare configuraties. De mogelijkheid om verschillende jobs en stages te definiëren binnen één pipeline maakt het modulair en makkelijk aan te passen aan complexe workflows.

2.3.9. Infisical

Infisical is een freemium open-source secrets management oplossing waarmee gevoelige gegevens zoals wachtwoorden, API-sleutels en tokens veilig kunnen worden beheerd.

Binnen het project werd Infisical onderzocht als alternatief voor commerciële secret managers.

Voordelen:

- **Open-source:** Vrij beschikbaar met een actieve en groeiende community.
- **Integraties:** Ondersteuning voor CI/CD-pijplijnen, Kubernetes en diverse cloudproviders.
- **Gebruiksvriendelijk:** Beheer van secrets via zowel een webinterface als een command-line interface (CLI).

Beperkingen:

- **Freemium model:** Sommige geavanceerde enterprise features vereisen een betaald abonnement.
- **Jongere community:** Minder volwassen ecosysteem in vergelijking met gevestigde oplossingen zoals HashiCorp Vault.

2.3.10. OpenBao

OpenBao is een open-source fork van HashiCorp Vault, ontstaan als reactie op commerciële licentiewijzigingen bij Vault.

Binnen het project werd OpenBao onderzocht als alternatief voor Vault, vanwege de open-source licentie, het behoud van compatibiliteit met bestaande Vault API's en het vermijden van commerciële restricties.

Voordelen:

- **Open-source licentie:** Vrij gebruik zonder commerciële beperkingen.
- **Compatibiliteit:** Ondersteunt grotendeels dezelfde API's als Vault, waardoor bestaande integraties hergebruikt kunnen worden. Grote delen van de documentatie van vault kunnen tevens ook gebruikt worden
- **Bewezen security model:** Gebaseerd op het solide fundament van HashiCorp Vault.
- **Sterk bestaan:** Openbao zit onder de Linux Foundation waardoor het een veilige en vaste positie heeft voor de toekomst

Beperkingen:

- **Community support:** Kleinere en minder actieve community dan bij HashiCorp Vault.

2.4. Vergelijkingen tools

In dit deel worden verschillende tools voor specifieke oplossingen met elkaar vergeleken. Dit gebeurt aan de hand van een Weighted Ranking Method (WRM), waarmee op een gestructureerde manier een gefundeerde keuze kan worden gemaakt. De weegfactoren worden afgestemd op de specifieke behoeften van Wilms NV.

2.4.1. Vergelijking van container/orchestrator tools

Hieronder is het WRM-diagram voor build- en deploydoeleinden weergegeven. Deze WRM vergelijkt de volgende tools:

- Docker Compose,
- Docker Swarm,
- Kubernetes

Evaluatiecriteria:

1. Maturiteit & community support (x3)
2. Beheercomplexiteit (x2)
3. Integratie met CI/CD & Vaults (x2)
4. Schaalbaarheid (x3)
5. Open source / vendor lock-in (x2)
6. Beschikbaarheid van visuele interface (x1)
7. Inhouse kennis binnen Wilms NV (x3)

De factoren *inhouse kennis*, *beheercomplexiteit* en *community support* zijn extra zwaar meegewogen, omdat het team momenteel alleen ervaring heeft met Docker Compose en overdraagbaarheid van het project cruciaal is.

Evaluatiefactor	Docker Compose	Docker Swarm	Kubernetes
Beheercomplexiteit (x3)	9	6	3
Maturiteit & Community support (x3)	8	7	10
Schaalbaarheid (x1)	6	8	10
Integratie met CI/CD pipelines (x2)	6	6	9
Inhouse kennis binnen Wilms NV (x3)	9	6	3
Vendor lock-in / Open source (x2)	6	6	6
Totaalscore	108	89	88

Tabel 2-1 WRM container/orchistrator tools

2.4.2. Vergelijking van configuratiemanagement tools

Hieronder kan je het WRM diagram voor De configuratiemanagementtools terugvinden. Deze WRM vergelijkt de volgende tools:

- Ansible
- Puppet

Evaluatiecriteria:

1. Beheercomplexiteit & onderhoud (x3)
2. Inhouse kennis en ervaring (x3)
3. Schaalbaarheid en stabiliteit (x2)
4. Flexibiliteit & uitbreidbaarheid (x2)
5. Integratie met bestaande automatisatie (x3)
6. Community support & maturiteit (x2)

De criteria rond inhouse kennis en beheercomplexiteit werden zwaar gewogen aangezien de oplossing door het interne team beheerd zal worden, en de huidige kennis rond Ansible reeds sterk aanwezig is binnen Wilms NV.

Evaluatiefactor	Ansible	Puppet
1. Beheercomplexiteit & onderhoud (x3)	9	5
2. Inhouse kennis en ervaring (x3)	9	3
3. Schaalbaarheid en stabiliteit (x2)	6	8
4. Flexibiliteit & uitbreidbaarheid (x2)	6	6
5. Integratie met bestaande automatisatie (x3)	9	5
6. Community support & maturiteit (x2)	8	9
Totaal	121	85

Tabel 2-2 WRM configuratiemanagement tools

2.4.3. Vergelijking van CI/CD Oplossingen

Om de optimale CI/CD-oplossing te selecteren voor het build- en deploymentproces, werden drie tools onderzocht: Azure DevOps Pipelines, Jenkins, en Kestra (in CI/CD configuratie). Hiervoor werd een WRM opgemaakt waarbij de focus sterk ligt op integratie, flexibiliteit en de inhouse kennis/leercurve.

Evaluatiecriteria:

1. Integratie met bestaande omgeving (x3)
2. Beheerbaarheid & onderhoud (x2)
3. Uitbreidbaarheid & flexibiliteit pipelines (x3)
4. Security & compliance features (x3)
5. Inhouse kennis & leercurve (x3)

De factor integratie met de bestaande omgeving en inhouse kennis kregen extra gewicht, gezien de nauwe koppeling met Azure DevOps en de bestaande competenties binnen het team.

Evaluatiefactor	Azure DevOps Pipelines	Jenkins	Kestra (CI/CD)
Integratie met bestaande omgeving (x3)	6	4	5
Beheerbaarheid & onderhoud (x2)	8	6	6
Uitbreidbaarheid & flexibiliteit pipelines (x3)	6	9	8
Security & compliance features (x3)	8	7	7
Inhouse kennis & leercurve (x3)	7	4	4
Totaalscore	97	84	84

Tabel 2-3 CI/CD Oplossingen

2.4.4. Vergelijking van Secret Management tools

Voor het beheer van secrets en gevoelige configuratie werden drie open-source oplossingen geanalyseerd: OpenBao, Infisical, en HashiCorp Vault. Een WRM-analyse werd uitgevoerd om de beste keuze te bepalen voor het veilig beheren van secrets in combinatie met deployments.

Evaluatiecriteria:

1. Licentiekosten & vendor lock-in (x3)
2. Integratie met pipeline & deploy (x3)
3. Beheercomplexiteit (x2)
4. Security features (x3)

De criteria rond licentiekosten, integratie met de pipelines en beheerbaarheid door het eigen team wogen zwaarder door, gezien de wens om onafhankelijk en volledig in eigen beheer te kunnen werken.

Evaluatiefactor	OpenBao	Infisical	HashiCorp Vault
Licentiekosten & vendor lock-in (x3)	9	6	4
Integratie met pipeline & deploy (x3)	6	7	6
Beheercomplexiteit (x2)	6	6	6
Security features (x3)	8	7	8
Inhouse beheerbaarheid (x3)	6	7	6
Totaalscore	99	93	84

Tabel 2-4 Secret Management tools

2.5. Onderzochte architectuuropties

Er werd uitgebreid onderzocht welke technologieën en de opstellingen/combinaties hiervan mogelijk waren om de doelstelling te realiseren. Vier concrete oplossingsrichtingen werden tegenover elkaar gezet:

1. Kestra met Custom Frontend

- Beschrijving: Kestra als workflow engine (job orchestrator), met een losstaande frontend voor gebruikersinteractie.
- Voordelen: Visualisatie van jobs, logging en gebruiksvriendelijke UI.
- Nadelen: Complexe integratie met vaults en authenticatie, moeilijk uit te breiden, en debugging blijft lastig.

2. Kestra stuurt Ansible direct aan

- Beschrijving: Workflows worden gedefinieerd in Kestra en roepen rechtstreeks Ansible-taken aan.
- Voordelen: Minder overhead, snelle setup.
- Nadelen: Niet flexibel, geen rolgebaseerde toegang, moeilijk te integreren met externe tokens of logging.

3. Kestra als frontend voor een Custom API

- Beschrijving: Kestra UI als façade, met backend requests naar een eigen API.
- Voordelen: Geen nood aan aparte frontend.
- Nadelen: Beperkte aanpasbaarheid van Kestra UI, debugging moeilijk, geen vrijheid in UX-keuzes.

4. Gekozen

- Beschrijving: Volledig zelfgebouwde frontend en backend API. Deze API spreekt Ansible Runner aan en handelt de build- en deploytaken asynchroon af.
- Voordelen:
 - Volledige vrijheid en uitbreidbaarheid
 - Beter te debuggen en te loggen (via SEQ, Prometheus)
 - Open source en modulair, dus geen afhankelijkheid van externe vendors
 - Aangepaste authenticatie en RBAC mogelijk
- Nadelen:
 - Langere initiële ontwikkeltijd
 - Moet inhouse ondersteund/ontwikkeld worden

2.6. Verantwoording van de keuze

De gekozen oplossing **custom API in FastAPI**, met Docker Compose, Ansible Runner integratie, Azure DevOps Pipelines en OpenBao sluit perfect aan bij zowel de technische vereisten als de organisatorische mogelijkheden van Wilms NV.

Waarom deze keuze technisch de beste is:

- Modulair en open source: Flexibel aanpasbaar zonder afhankelijk te zijn van externe platformen.
- Security by design: Volledige controle over authenticatie, autorisatie en secrets management via OpenBao.
- Debugging en monitoring: Metrics via Prometheus, logging via SEQ, status-tracking via een eigen API.
- Eenvoudige CI/CD-integratie: API kan rechtstreeks benaderd worden vanuit Azure DevOps of andere systemen.

Waarom deze keuze past bij de organisatie:

- Wilms NV beschikt over een uitgebreid IT-team met:
 - 2 interne infra-engineers
 - 3 fulltime in-house ontwikkelaars
 - extra externe ondersteuning (devs + infra)
- Deze in-house expertise maakt het mogelijk om een custom tool zelf te beheren en verder te ontwikkelen.
- De oplossing sluit aan bij de bestaande evolutie richting containerization en moderne DevOps-methodologie.
- Een groot deel van de tools is al deels gekend binnenin het team waardoor het onderhouden makkelijker is.

3. Realisatie

Tijdens de realisatie is ervoor gekozen om zo modulair mogelijk te werken. De code is daarom opgebouwd uit overzichtelijke blokken en functies, zodat deze gemakkelijk aanpasbaar en goed leesbaar is.

Deze keuze werd gemaakt omdat al vroeg in het proces duidelijk werd dat de eisen van het project voortdurend veranderden.

Dankzij de modulaire opzet van de code en het werken volgens de DevOps-methodologie, met sprints van telkens twee weken, konden aanpassingen snel en flexibel worden doorgevoerd.

In de volgende delen splits ik het project op in subonderdelen en licht ik elk onderdeel afzonderlijk toe. Tot slot toon ik de opgeleverde documentatie en schematische bijlagen die de werking van het project verder verduidelijken.

3.1. Onsite Azure Devops pipelines

Het eerste vraagstuk in het project was het vinden van een manier om Azure DevOps-pipelines op de lokale infrastructuur te laten draaien, of in ieder geval een manier om via Azure DevOps-pipelines de lokale API aan te spreken.

Nadat ik had onderzocht hoe Azure DevOps-pipelines werken, kwam ik tot de conclusie dat deze draaien op agents binnen een agent pool.

Via de Microsoft-documentatie kwam ik al snel uit bij self-hosted Azure DevOps pipeline agents. Hierbij had ik verschillende opties, zoals Linux-agents, Windows-agents en Docker-agents.

Na overleg met mijn opdrachtgever heb ik gekozen voor de Docker-agent, omdat deze het beste past binnen de huidige situatie.

De agent kon op twee manieren authenticeren met Azure: via een Personal Access Token (PAT) of via een Service Principal (SP). Aanvankelijk gebruikte ik de PAT-methode, maar dit gaf later problemen bij het clonen van Git-repositories, omdat het niet mogelijk is om een PAT-token aan te maken voor een SP. Bovendien wordt deze PAT-token ook gebruikt voor het clonen van Git-repositories.

Aangezien de API via de SP moest authenticeren, is besloten de agent te laten authenticeren met de client ID en secret ID van de SP.

Omdat er later in de pipeline gebruik wordt gemaakt van GitVersion voor het automatisch genereren van versietags, moest dit ook geïnstalleerd worden zodat de agent het kan gebruiken.

Dit kan eenvoudig door de Dockerfile van de Linux-agent, die door Microsoft wordt geleverd, aan te passen. We zorgen ervoor dat in de apt-installatiestappen een extra installatie van .NET plaatsvindt, aangezien dit onderliggend wordt gebruikt om GitVersion te installeren.

Fig -----

age / Repos / Files / infra_stage ▾

master ▾ / azp-agent-in-docker / azp-agent-linux.dockerfile

azp-agent-linux.dockerfile

Contents History Compare Blame

```
1 FROM ubuntu:22.04
2 ENV TARGETARCH="linux-x64"
3 # Also can be "linux-arm", "linux-arm64".
4
5 ARG DEBIAN_FRONTEND=noninteractive
6
7 # Needed packages for azure devops agent and gitversion
8 RUN apt update && \
9     apt upgrade -y && \
10    apt install -y curl git jq libicu70 && \
11    apt install -y dotnet-sdk-8.0
12
13 # Install Azure CLI
14 RUN curl -sL https://aka.ms/InstallAzureCLIDeb | bash
15
16 WORKDIR /azp/
17
18 COPY ./start.sh ./
19 RUN chmod +x ./start.sh
20
21 # Create agent user and set up home directory
22 RUN useradd -m -d /home/agent agent
23 RUN chown -R agent:agent /azp /home/agent
24
25 USER agent
26 # Another option is to run the agent as root.
27 # ENV AGENT_ALLOW_RUNASROOT="true"
28
29 ENTRYPOINT [ "./start.sh" ]
30
```

Figuur 3-1 dockerfile azure pipeline agent

3.2. Centraal beheer van config met Openbao

Om een single source of truth te creëren voor zowel de developers als het infra-team, is een systeem of tool nodig die dit kan bijhouden.

Hiervoor is, zoals eerder genoemd, gekozen voor OpenBao.

3.2.1. Installatie

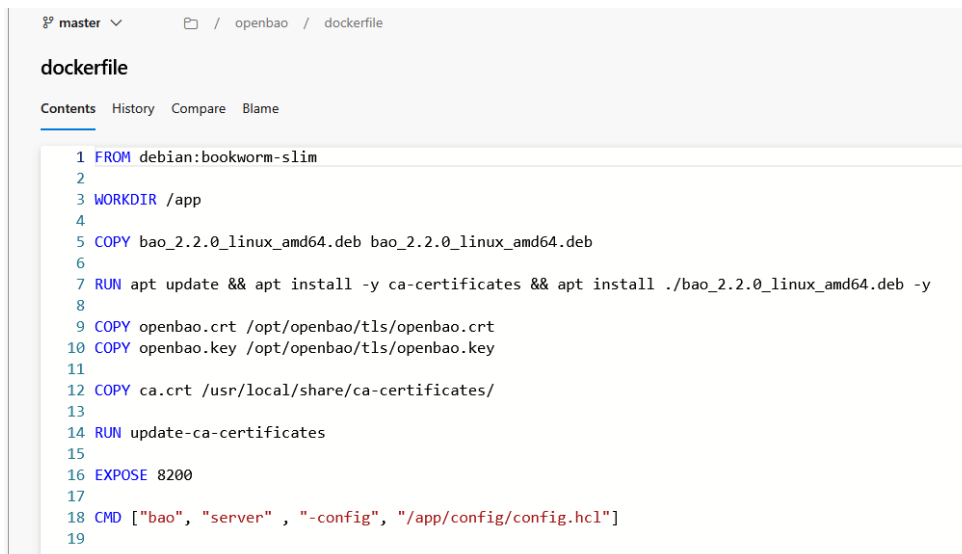
Voor de installatie van OpenBao heb ik gekozen voor Docker, omdat dit past binnen de visie van het bedrijf.

Aanvankelijk heb ik gebruikgemaakt van de officiële Docker-image van OpenBao. Al snel ontdekte ik echter dat deze image was ingesteld als een dev-opstelling. Dit hield in dat de storage backend was geconfigureerd als in-memory, waardoor bij een herstart van de server of een crash van de container alle gegevens, configuraties en secrets verloren zouden gaan.

Na verder zoeken in de documentatie van OpenBao heb ik besloten om zelf een custom container image te maken, inclusief een aangepast configuratiebestand, zodat de storage-methode productie-klaar is.

Ik koos voor de Raft-backend, omdat deze het mogelijk maakt om met een single-node setup te werken, terwijl er later nog steeds de vrijheid is om uit te breiden naar een multi-node setup. Dit maakt het ideaal voor een toekomstgerichte opstelling.

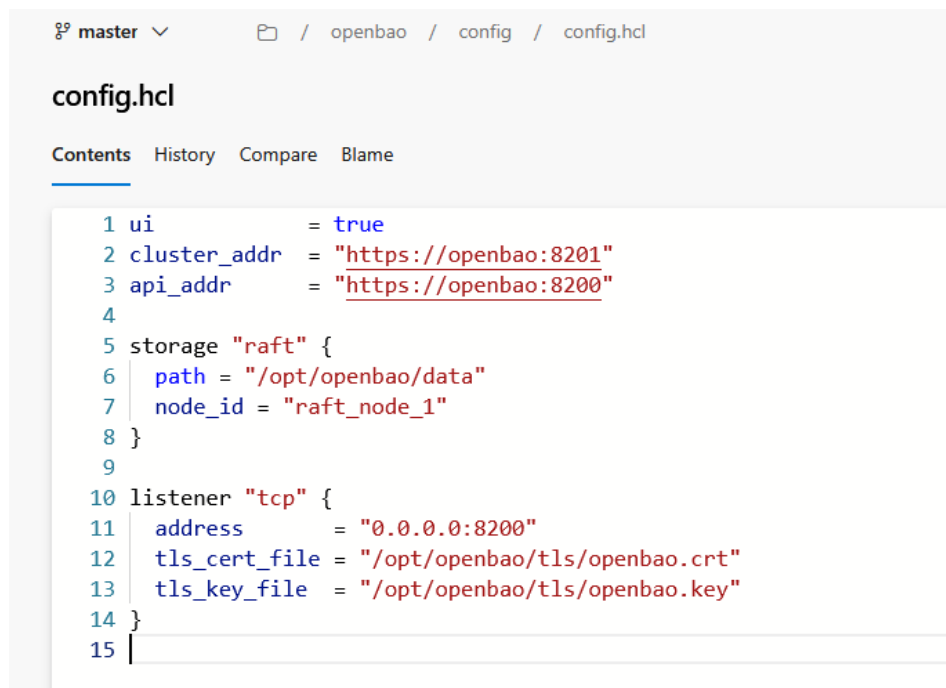
Eerst heb ik een custom Dockerfile gemaakt die de officiële Debian-package van OpenBao installeert, de juiste certificaten toevoegt en OpenBao opstart met het juiste configuratiebestand.



```
1 FROM debian:bookworm-slim
2
3 WORKDIR /app
4
5 COPY bao_2.2.0_linux_amd64.deb bao_2.2.0_linux_amd64.deb
6
7 RUN apt update && apt install -y ca-certificates && apt install ./bao_2.2.0_linux_amd64.deb -y
8
9 COPY openbao.crt /opt/openbao/tls/openbao.crt
10 COPY openbao.key /opt/openbao/tls/openbao.key
11
12 COPY ca.crt /usr/local/share/ca-certificates/
13
14 RUN update-ca-certificates
15
16 EXPOSE 8200
17
18 CMD ["bao", "server", "-config", "/app/config/config.hcl"]
19
```

Figuur 3-2 dockerfile Openbao

Voor de configuratie van OpenBao heb ik gekozen voor een minimale opstelling, waarbij ik de web-UI heb ingesteld, de benodigde adressen heb gedefinieerd en de storage-backend heb geconfigureerd naar Raft.



```
1 ui = true
2 cluster_addr = "https://openbao:8201"
3 api_addr = "https://openbao:8200"
4
5 storage "raft" {
6 | path = "/opt/openbao/data"
7 | node_id = "raft_node_1"
8 }
9
10 listener "tcp" {
11 | address = "0.0.0.0:8200"
12 | tls_cert_file = "/opt/openbao/tls/openbao.crt"
13 | tls_key_file = "/opt/openbao/tls/openbao.key"
14 }
15 |
```

Figuur 3-3 configuratiebestand Openbao

Vervolgens kan deze gebouwd worden en via een eenvoudige Compose-opdracht worden opgestart.

```
e / Repos / Files / infra_stage ▾  
  
master ▾ / openbao / docker-compose.yml  
  
docker-compose.yml  
Contents History Compare Blame  
  
1 name: openbao  
2 services:  
3   openbao:  
4     image: openbao:latest  
5     restart: unless-stopped  
6     container_name: openbao  
7     volumes:  
8       - ./config:/app/config  
9       - ./data:/opt/openbao/data
```

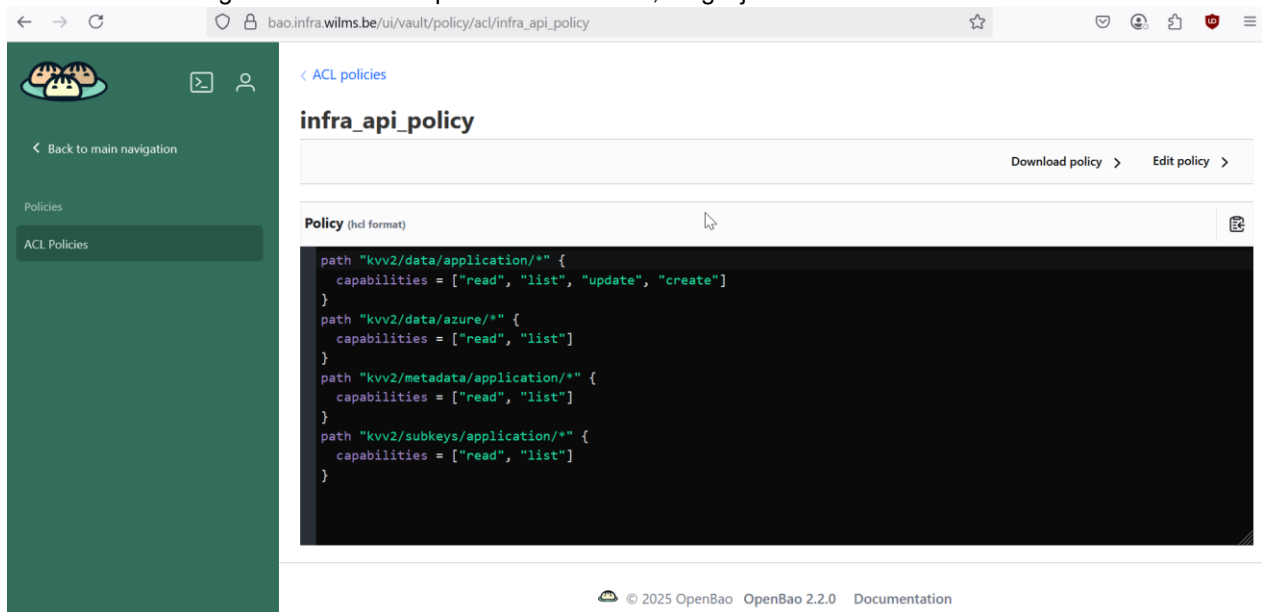
Figuur 3-4 docker-compose bestand Openbao

Vervolgens moesten verschillende componenten binnen OpenBao zelf geconfigureerd worden. Voor het opslaan van configuraties, secrets en omgevingsvariabelen heb ik gekozen voor de Key-Value Pair Secret Engine V2 (KVV2).

Deze was via de interface van OpenBao eenvoudig te configureren. Het gekozen pad is hierbij erg belangrijk.

Figuur 3-5 Instellen secret engine Openbao

Vervolgens maken we via de interface een policy aan met de juiste rechten om noodzakelijke acties, zoals het lezen van configuraties en het aanpassen van versies, mogelijk te maken.



Figuur 3-6 Instellen ACL policy Openbao

Vervolgens moet een AppRole geconfigureerd worden. Een AppRole binnen OpenBao is te vergelijken met een App Role of Service Principal (SP) binnen Azure. Een AppRole heeft een ID, een client ID en een secret ID; dit zijn als het ware de credentials van een applicatie. Je kunt dit zien als de gebruikersnaam en het wachtwoord van de API voor OpenBao. We maken deze AppRole aan zodat de API kan authenticeren bij OpenBao.

Het aanmaken kan bijvoorbeeld via de OpenBao API. In figuur ----- is te zien hoe je AppRole inschakelt, hoe je een AppRole met de naam "cicd-approle" aanmaakt en hieraan een policy koppelt (zie figuur [-----]). Ook wordt getoond hoe je de role_id en de bijbehorende secret_id kunt opvragen.

3.2.2. Secret structuur

Om configuratie en secrets bij te houden in Openbao heb ik een voor gedefinieerde structuur opgesteld samen met het interne team, de structuur is zodanig opgesteld dat deze modulair en flexibel is. Deze structuur wordt in de API gebruikt om secrets te zoeken daarom is het heel belangrijk dat deze vanaf de eerste keer een goede structuur had.

We hebben daarom ook gekozen voor de volgende structuur zie fig ----- en fig -----

Bovenaan de boomstructuur is het path van de secret engine, opgevolgd door een algemene azure en "application" key. Onder de "application" key komen dan de verschillende projecten bv. "testapp" onder de applicatie zijn de verschillende omgevingen van de app gedefinieerd bv. Dev, prd, etc. Onder elke omgeving zijn er dan weer bepaalde sub componenten. De "docker" secret heeft alle key-value pairs dat gebruikt worden door docker compose.

Ook is er een "resources" secret, in deze secret kunnen er bestanden en folder paths meegegeven worden vanuit git.

Deze bestanden of de inhoud van de folders worden tijdens deploy mee over gekopieerd naar de deployment host.

Dit is handig voor als er extra configuratie bestanden in git voor een applicatie nodig zijn. Deze kunnen dan op hun beurt meegestuurd worden tijdens deployment.

Bovenaan de boomstructuur onder "azure" is een key "service_principal" deze key heeft alle nodige service principals van azure met hun client en secret id. Voor de opstelling is enkel een SP nodig

Door de azure credentials die nodig gaan zijn voor git in Openbao te plaatsen kunne we deze op een veilige manier opslagen zonder deze in de code of env variabelen van de API te verwerken.

Fig -----

3.3. Builden van containers via Azure Devops

De eerste hoofdfunctionaliteit dat gerealiseerd is is de mogelijkheid om containers te builden vanuit azure devops.

Het build proces zal instaan om de juiste tagging te voorzien voor de commits en containers, ook zal dit proces communiceren met de interne custom API om de builds van de containers te starten en te pushen naar de onsite registry.

Samengevat voor het builden van de containers zijn de volgende functionaliteiten geïmplementeerd.

- De functionaliteit creëren voor het automatisch taggen van commits
- De functionaliteit creëren om builds on site te laten starten vanuit azure devops
- De mogelijkheid voor developers creëren om zelf project en container namen te definiëren
- Feedback van de builds weergeven in azure devops en nodige error handling implementeren
- De functionaliteit creëren om tags automatisch te bumpen in de centrale configuratie store (OpenBao)

In dit deel wordt de pipeline op azure devops uitgelegd en hoe de functionaliteiten hier zijn geïmplementeerd en hoe deze inhaken in de custom API.

Hier wordt later dan ingehaakt wanneer de realisatie van de custom API wordt besproken.

3.3.1. Semantic versioning met GitVersion

Voor dat het bouwen van containers van start kon gaan moest er eerst een manier zijn om automatisch tagging te implementeren op de repositories.

Hiervoor is GitVersion gebruikt.

Met een simple yaml bestand kan GitVersion ingesteld worden om de volgende versioning aan te houden. 1.0.0-suffix.

Dit maakt het mogelijk voor major bumps te doen bv. 1.0.0 → 2.0.0, ook kunnen er minor bumps gebeuren tijdens het development proces bv. 1.1.0 → 1.2.0 en ten slotte een patch bump bv. 1.0.1 → 1.0.2.

De mogelijkheid voor suffixen maakt het mogelijk om pre-release tags te maken voor builds die getest moeten worden. Bv. 1.1.0-frontend-feature-pr

Via de “next-version” key kunnen de developers zelf manueel hun nieuwe major bump bepalen tijdens het begin van een nieuwe sprint.

Aan de hand van simpele regex notaties kan er bepaald worden wat er als een main en hotfix branch wordt aanschouwd.

GitVersion.yml

Contents Highlight changes

```
1 next-version: 1.0.0
2 assembly-versioning-scheme: MajorMinorPatch
3 assembly-file-versioning-scheme: MajorMinorPatch
4 tag-prefix: '[v]?'
5 major-version-bump-message: \+semver:\s?(breaking|major)
6 minor-version-bump-message: \+semver:\s?(feature|minor)
7 patch-version-bump-message: \+semver:\s?(fix|patch)
8 no-bump-message: \+semver:\s?(none|skip)
9 tag-pre-release-weight: 60000
10 commit-date-format: yyyy-MM-dd
11 merge-message-formats: {}
12 update-build-number: true
13
14 branches:
15   main:
16     mode: ContinuousDelivery
17     increment: Minor
18     regex: ^master$|^main$
19     source-branches: []
20     is-source-branch-for: []
21     tracks-release-branches: false
22     is-release-branch: false
23     pre-release-weight: 55000
24
25   hotfix:
26     mode: ContinuousDelivery
27     increment: Patch
28     regex: (?i).*hotfix.*
29     source-branches: []
30     tag: ''
31     is-source-branch-for: []
32     is-release-branch: true
33     pre-release-weight: 30000
34
35   # Catch-all: alles wat niet main/master en niet hotfix/... is
36   other:
37     mode: ContinuousDeployment
38     increment: None
39     # Vangt elke branch-naam in groep "BranchName" tenzij master/main of hotfix erin
40     regex: '^(?!master$|^main$|^.*hotfix.*)(?<BranchName>.+)${}'
41     # Gebruik die groep in de prerelease-tag, met "preview-" ervoor
42     tag: 'preview-{BranchName}'
43     source-branches: []
44     is-source-branch-for: []
45     pre-release-weight: 60000
46
47 ignore:
48   sha: []
49
50 mode: ContinuousDelivery
51 increment: Inherit
52 commit-message-incrementing: Enabled
53 |
```

Figuur 3-7 Gitversion.yml bestand

Dit yaml bestand wordt dan door GitVersion gebruikt in de pipeline om een versie te genereren, en vervolgens wordt deze versie gebruikt om een tag te maken en deze te pushen via git

azure-pipelines.yml

[Contents](#) [Highlight changes](#)

```
46 # Install GitVersion tool to determine semantic version
47 - task: gitversion/setup@3
48   displayName: "Install GitVersion"
49   inputs:
50     versionSpec: "5.x" # Specify the GitVersion version to install (e.g., '5.x' or a specific version like '5.10.x')
51
52 # Output GitVersion variables (SemVer and FullSemVer)
53 - task: gitversion/command@3
54   displayName: "GitVersion@command: Determine Version & Output the FullSemVer variable"
55   name: gitversioning_init # Identifier for this task (used for referencing output variables)
56   inputs:
57     disableShallowCloneCheck: true # Disable shallow clone check because full history is fetched
58     arguments: "/showvariable SemVer /showvariable FullSemVer" # Output the SemVer and FullSemVer variables
59
60 # Parse GitVersion config and output variables to the build server
61 - task: gitversion/execute@3
62   displayName: "Run GitVersion"
63   name: gitversioning # Task identifier to capture the version outputs
64   inputs:
65     updateAssemblyInfo: false # Do not update assembly info files automatically
66     additionalArguments: "/output buildserver /config src/Project/GitVersion.yml" # Specify output format and config file
67
68 # Log the resolved semantic version
69 - script: echo "SemVer is $(gitversioning.SemVer)"
70   displayName: "Log GitVersion SemVer"
71
72 # =====
73 # Step 3: Install NuGet CLI
74 # =====
75 - task: NuGetToolInstaller@1
76
77 # =====
78 # Step 4: Create and Push Git Tag
79 # =====
80 - task: Bash@3
81   displayName: "Create and push Git tag"
82   inputs:
83     targetType: "inline"
84     script: |
85       set -e
86
87       VERSION="$(gitversioning.SemVer)"
88       echo "Creating tag $VERSION"
89
90       # Check if tag already exists
91       if git rev-parse --verify --quiet "refs/tags/$VERSION" >/dev/null; then
92         # Log een echte error in de DevOps UI
93         echo "##vso[task.logissue type=error]Tag '$VERSION' already exists!"
94         # Zorg dat de task faalt
95         exit 1
96       fi
97
98       # Tag and push
99       git config user.email "agent@wilms.be"
100       git config user.name "Build Agent"
101       git tag "$VERSION"
102       git push origin "$VERSION"
103
```

Figuur 3-8 Azure-pipelines.yml bestand

3.3.2. Build trigger

De build trigger is de stap in de pipeline dat de api call zal samenstellen en ook zal gaan uitvoeren.

Voor de build trigger op te stellen in de pipeline moest ik een manier vinden om de pipeline te laten authenticeren

Eerst kan de url van de api worden ingesteld zodat de pipeline weet waar hij de api call naar moet maken. Vervolgens wordt er nog naar dockerfiles gezocht binnenin de repository en worden deze getoont.

azure-pipelines.yml

Contents Highlight changes

```
103
104 # =====
105 # Step 5: Set Global API Base URL
106 # =====
107 - script: |
108     API="http://cicd-infra-api-01:8000"
109     echo "##vso[task.setvariable variable=API_URL]${API}"
110     displayName: "Set global variables for cicd api stages"
111
112 # =====
113 # Step 6: Locate Dockerfiles
114 # =====
115 - script: |
116     echo "Searching for dockerfiles in the repository..."
117
118     # Find files named "dockerfile" (case-insensitive) starting from the current directory
119     dockerfiles=$(find . -type f -iname "dockerfile")
120
121     # If no Dockerfiles are found, output a message; otherwise, list them
122     if [ -z "$dockerfiles" ]; then
123         echo "No dockerfiles found."
124     else
125         echo "Dockerfiles found:"
126         echo "$dockerfiles"
127     fi
128     displayName: "Find Dockerfiles"
129
```

Figuur 3-9 Azure-pipelines.yml bestand

Vervolgens authenticceert de pipeline met de API via curl om een token te verkrijgen.

```
azure-pipelines.yml
Contents Highlight changes

131 # Step 7: Trigger Docker Builds via API
132 # =====
133 - script: |
134     original_url="$(Build.Repository.Uri)"
135     echo "Original URL: ${original_url}"
136
137     # Remove the prefix @ in the HTTPS repository URL for use in the API
138     modified_url=$(echo "${original_url}" | sed -E 's#https://[^@]+@#https://@#')
139     echo "Modified URL: ${modified_url}"
140
141     # Set up a retry loop to retrieve a JWT token (maximum 5 attempts)
142     max_retries=5
143     attempt=0
144     while [ $attempt -lt $max_retries ]; do
145         echo "Attempt ${attempt+1}) to retrieve JWT token..."
146
147         # Make a POST request to the authentication server to obtain a JWT token
148         auth_response=$(curl -s -X POST "${API_URL}/token" \
149             -H "Content-Type: application/x-www-form-urlencoded" \
150             --data-urlencode "username=${API_AGENT_USERNAME}" \
151             --data-urlencode "password=${API_AGENT_PASSWORD}")
152         echo $auth_response
153
154         # Extract the access token using jq
155         BEARER_TOKEN=$(echo "$auth_response" | jq -r '.access_token')
156
157         # If the token is retrieved successfully, exit the retry loop
158         if [ -n "$BEARER_TOKEN" ] && [ "$BEARER_TOKEN" != "null" ]; then
159             echo "JWT token retrieved successfully."
160             break
161         fi
162
163         # If token retrieval fails, wait for 5 seconds and retry
164         echo "Failed to retrieve JWT token. Retrying in 5 seconds..."
165         attempt=$((attempt+1))
166         sleep 5
167     done
168
169     # Exit the script if no valid JWT token was retrieved after all attempts
170     if [ -z "$BEARER_TOKEN" ] || [ "$BEARER_TOKEN" == "null" ]; then
171         echo "Failed to authenticate and retrieve JWT token after $max_retries attempts. Exiting."
172         exit 1
173     fi
```

Figuur 3-10 Azure-pipelines.yml bestand

Daarna wordt er een manifest bestand bekeken. Dit manifest bestand (manifest-docker.json) zal door de developers gemaakt worden en zal bepalen hoe het project en de onderliggende containers benoemd worden en hoe ze gebuild kunnen worden.

Toplevel wordt de projectnaam bepaald waaronder elke container/service dan apart gedefinieerd kan worden met hun context en het path naar hun dockerfile. Bv. Project “test-app” heeft twee containers, een “frontend” en “backend” (zie fig)

```
nfra_stage / Repos / Files / infra_stage
> openbao
> openbao-opentofu
> ot
> prd
> Properties
> registry
> registry-testing-certs
> roles
> stg

manifest-docker.json
Contents History Compare Blame

1 {
2   "test-app": {
3     "frontend": {
4       "dockerfile_context": "/test-app/Docker-Compose-Projects/frontend",
5       "dockerfile_path": "Dockerfile"
6     },
7     "backend": {
8       "dockerfile_context": "/test-app/Docker-Compose-Projects/backend",
9       "dockerfile_path": "Dockerfile"
10    }
11  }
12 }
```

Figuur 3-11 Azure-pipelines.yml bestand

In de pipeline wordt dit manifest bestand uitgelezen en wordt door elke service geloopt, de context, bestandslocatie en build argumenten worden hier dan uitgehaald.
En vervolgens gebruikt om de call te maken voor een container image te bouwen.
Jq is een kleine cli tool voor het parsen filteren en transformeren van json en is daarom hier ideaal voor.

```

azure-pipelines.yml
Contents Highlight changes

175 # Parse the manifest JSON file to determine the project name
176 project=$(jq -r 'keys[0]' manifest-docker.json)
177 echo "Project: ${project}"
178 echo "##vso[task.setvariable variable=project_name]$project"
179
180 # Retrieve the list of services defined for the project from the manifest file
181 services=$(jq -r --arg project "$project" '.[${project}] | keys[]' manifest-docker.json)
182
183 # Loop through all services
184 for service in $services; do
185   echo "Processing service: $service"
186
187   # Get the Dockerfile context for the service; default to "/" if not provided
188   dockerfile_context=$(jq -r --arg project "$project" --arg service "$service" '.[${project}][${service}].dockerfile_context // "/"' manifest-docker.json)
189   echo "Dockerfile context for service $service: ${dockerfile_context}"
190
191   # Get the Dockerfile path for the service; default to "Dockerfile" if not provided
192   dockerfile_path=$(jq -r --arg project "$project" --arg service "$service" '.[${project}][${service}].dockerfile_path // "Dockerfile"' manifest-docker.json)
193   echo "Dockerfile for service $service: ${dockerfile_path}"
194
195   # Retrieve the build arguments for the service; default to an empty object if not provided.
196   # This can contain multiple key-value pairs.
197   build_args=$(jq -c --arg project "$project" --arg service "$service" '.[${project}][${service}].build_args // {}' manifest-docker.json)
198   # Uncomment for debugging purposes: log the build arguments
199   #echo "Build arguments for service $service: ${build_args}"
200
201   # Construct the full image name using the registry address, project, and service details
202   image_name="${project}/${service}"
203   echo "##vso[task.setvariable variable=IMAGE_NAME]$image_name"
204   echo "Image name: ${image_name}"
205   # Show tag version for the image full image name + tag gets constructed on api side ansible
206   echo "Image tag: ${gitversioning.SemVer}"
207

```

Figuur 3-12 Azure-pipelines.yml bestand

Vervolgens wordt alles samengevoegd en wordt de build call uitgevoerd

```

208 # Set up a retry loop for sending the POST request to trigger the Docker build (maximum 5 attempts)
209 max_retries_build=5
210 attempt_build=0
211 while [ $attempt_build -lt $max_retries_build ]; do
212   echo "Attempt ${attempt_build+1} to send POST request for service $service..."
213
214   # Send a POST request to the build_async endpoint with build parameters in JSON format,
215   # including the build_args which may contain multiple key-value pairs.
216   response=$(curl -s -X POST "${API_URL}/build_async" \
217     -H "accept: application/json" \
218     -H "Authorization: Bearer ${BEARER_TOKEN}" \
219     -H "Content-Type: application/json" \
220     -d "{
221       \"image_name\": \"${image_name}\",
222       \"image_tag\": \"${gitversioning.SemVer}\",
223       \"dockerfile_context\": \"${dockerfile_context}\",
224       \"dockerfile_path\": \"${dockerfile_path}\",
225       \"repo_link\": \"${modified_url}\",
226       \"branch_tag\": \"${gitversioning.SemVer}\",
227       \"build_args\": ${build_args}
228     }")

```

Figuur 3-13 Azure-pipelines.yml bestand

3.3.3. Automatische version bump

De automatische version bump is de laatste stap in de Azure DevOps-pipeline.

Deze stap is noodzakelijk omdat het deploymechanisme afhankelijk is van een METADATA-tag in een secret binnen OpenBao, onder de Docker-secrets (zie OpenBao-structuurlink). Nadat een nieuwe build is uitgevoerd, bestaat deze versie nog niet in OpenBao voor de applicatie. Daardoor wordt deze versie niet opgenomen in de lijst van beschikbare versies om te deployen. Dit voorkomt dat er versies gedeployed worden zonder bijbehorende configuratie in OpenBao, wat tot mislukte deploys zou leiden.

De laatste stap in de pipeline stuurt een API-call naar het /secrets/bump-endpoint van de API.

Deze call bevat een request body met de keys application en new_tag.

- application is de naam van het project in OpenBao, te vinden onder /kv2/application/{application_name}.
- new_tag is de nieuwe tag, gegenereerd door GitVersion, en wordt daarom direct meegegeven.

zie fig -----

```
318 # Set up a retry loop for the tag bump operation (maximum 5 attempts)
319 bump_max_retries=5
320 bump_attempt=0
321 success=false
322
323 while [ $bump_attempt -lt $bump_max_retries ]; do
324     echo "Attempt $((bump_attempt+1)) to bump tag..."
325
326     response=$(curl -s -w "%{http_code}" -o response.json -X POST "${API_URL}/secrets/bump" \
327         -H "accept: application/json" \
328         -H "Authorization: Bearer ${BEARER_TOKEN}" \
329         -H "Content-Type: application/json" \
330         -d "{
331             \"application\": \"$(project_name)\",
332             \"new_tag\": \"$(gitversioning.SemVer)\"
333         }")
334
335     http_code="${response: -3}" # Extract last 3 characters as HTTP status code
336     body=$(cat response.json)
337
338     echo "HTTP Status Code: $http_code"
339     echo "Response Body: $body"
340
341     if [ "$http_code" -eq 200 ]; then
342         echo "Tag bump succeeded for $(project_name)"
343         success=true
344         break
345     else
346         echo "Failed to bump tag. HTTP $http_code"
347         bump_attempt=$((bump_attempt+1))
348         sleep 5
349     fi
350 done
351
352 if [ "$success" != "true" ]; then
353     echo "ERROR: Failed to bump tag after $bump_max_retries attempts."
354     exit 1
355 fi
356 displayName: 'Bump secret(s) for application'
```

Figuur 3-14 Azure-pipelines.yml bestand

3.4. Custom API

In dit deel leg ik de realisatie van het hoofdcomponent van het project uit: de custom API, en hoe de eerder besproken functionaliteiten van de pipeline hierop aansluiten.

De custom API vormt het hart van het project. Zij fungeert als interface en koppelpunt waar alle componenten en tools samenkomen.

Omdat de API het meest uitgebreide onderdeel is, bestaat deze uit vele verschillende componenten.

De volgende componenten worden besproken:

- Het databasemodel
- Pydantic modellen
- Login
- Token managers
- Ansible playbooks
- Logging
- Metrics
- Endpoints

3.4.1. Het databasemodel

Het databasemodel bepaalt hoe het database schema er zal uitzien, het definieert de tabellen, de kolommen, de datatypes en de relaties tussen de tabellen.

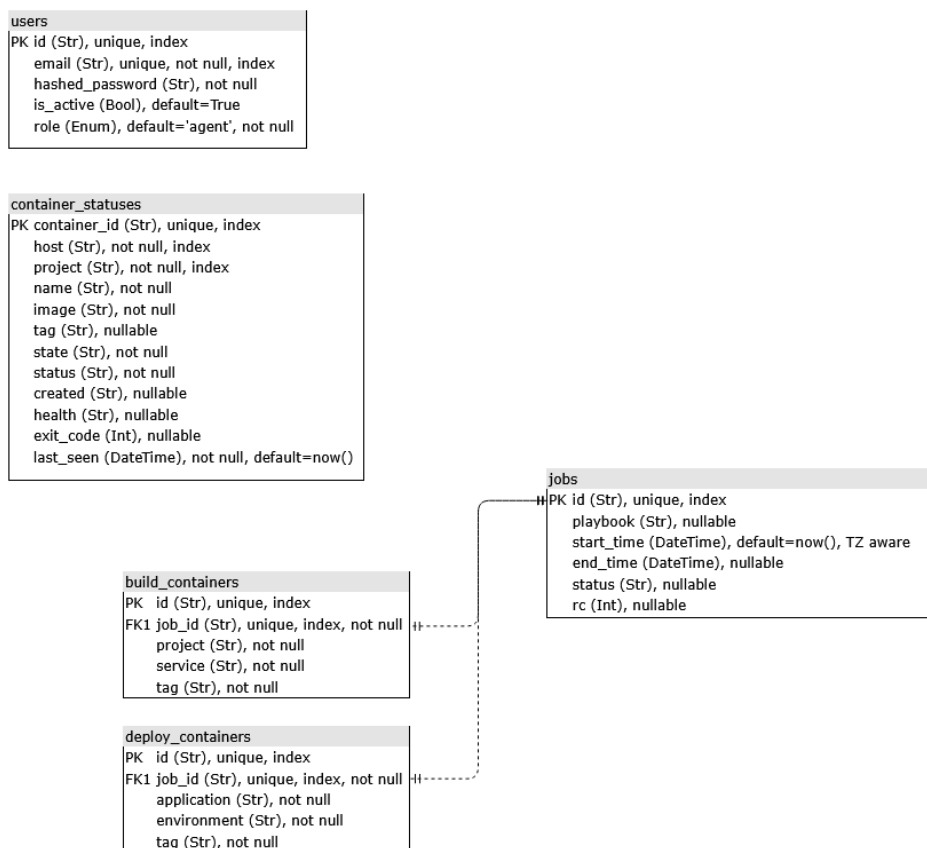
Voor dit project is gekozen voor een simpel databasemodel dat de nodige behoeftes van de API dekt en tegelijkertijd zorgt voor eenvoudige uitbreidbaarheid.

De database beschikt over een user tabel. Deze tabel gaat alle gegevens van de user bijhouden en wordt vooral gebruikt voor authenticatie en autorisatie. Het email veld dient als username, hashed_password bevat het gehashte password, dit wordt gehashed met argon2 backwards compatible met bcrypt.

De container_statuses tabel wordt gebruikt om de gegevens van de huidige draaiende containers in de verschillende omgevingen bij te houden. De info hiervan is de output van het docker compose ls en docker compose ps commando.

De jobs tabel houdt alle info over ansible-runner jobs bij, wanneer deze gestart zijn, hun eindtijd, hun status, zijn ze succesvol of niet?

De build en deploy_containers tabellen worden gebruikt om meer details over een build of deploy job bij te houden. Deze wordt dan ook 1 op 1 gelinkt met een unieke id in de job tabel



Figuur 3-15 ERD diagram database

3.4.2. Pydantic modellen

Binnen mijn FastAPI-project heb ik gebruik gemaakt van Pydantic modellen. Deze modellen worden gebruikt om de structuur van inkomende en uitgaande data te definiëren.

Ze zorgen ervoor dat:

- Data validatie automatisch gebeurt voordat deze in de logica van de applicatie terechtkomt.
- Fouten vroegtijdig worden opgevangen, bijvoorbeeld wanneer vereiste velden ontbreken of een verkeerd datatype hebben.
- De automatische API-documentatie (zoals Swagger UI) van FastAPI correct en duidelijk wordt gegenereerd.

Hieronder licht ik als voorbeeld twee van mijn Pydantic modellen uit:

1. AddSecretsRequest:

Dit model wordt gebruikt voor het ontvangen van secrets voor applicaties die geïmporteerd worden. De structuur is diep genest, zodat secrets per applicatie en per omgeving in bulk kunnen worden geïmporteerd en georganiseerd.

```
44 # Schema for incoming secrets.
45 class AddSecretsRequest(BaseModel):
46     # Schema for uploading application secrets to Openbao.
47     # The structure is nested by app → env → secret group.
48     applications: Dict[str, Dict[str, Dict[str, Any]]]
49
```

Figuur 3-16 Pydantic model AddSecretRequest

Hieronder een voorbeeld:

schemas.py

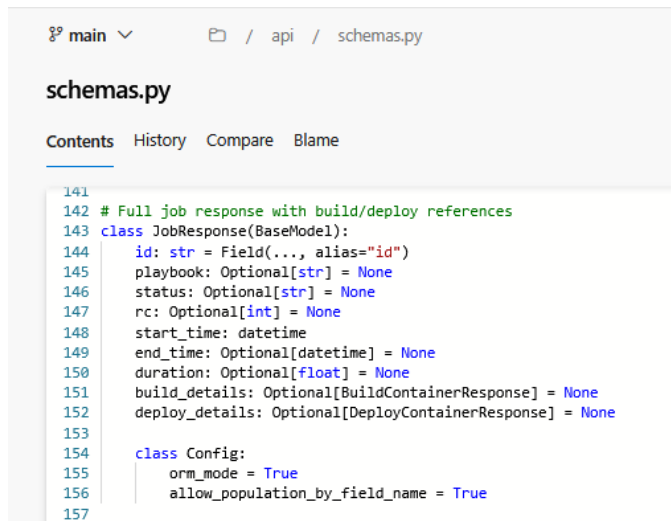
Contents History Compare Blame

```
49
50 # Example structure:
51 class Config:
52     schema_extra = {
53         "example": {
54             "applications": {
55                 "myapp": {
56                     "dev": {
57                         "docker": {
58                             "FRONTEND_IMAGE": "myapp:latest",
59                             "FRONTEND_TAG": "latest",
60                             "METADATA": {
61                                 "tag": "1.0.0",
62                                 "managed_apps": {
63                                     "FRONTEND": "",
64                                     "BACKEND": "",
65                                     "DATABASE": ""
66                                 }
67                             }
68                         },
69                         "database": {
70                             "USER": "foo",
71                             "PASS": "bar"
72                         }
73                     },
74                     "prd": {
75                         "docker": {
76                             "BACKEND_IMAGE": "myapp:1.2.3",
77                             "BACKEND_TAG": "1.2.3",
78                             "METADATA": {
79                                 "tag": "1.2.3",
80                                 "managed_apps": {
81                                     "FRONTEND": "",
82                                     "BACKEND": "",
83                                     "DATABASE": ""
84                                 }
85                             }
86                         },
87                         "database": {
88                             "USER": "foo",
89                             "PASS": "secure!"
90                         }
91                     }
92                 }
93             }
94         }
95     }
96
```

Figuur 3-17 Pydantic AddSecretRequest voorbeeld

2. JobResponse:

Dit model representeert een uitgebreide response van een job, inclusief verwijzingen naar build- of deploy-details. Het bevat onder andere statusinformatie, tijden, returncodes, en optioneel de gekoppelde build- of deploy-data.

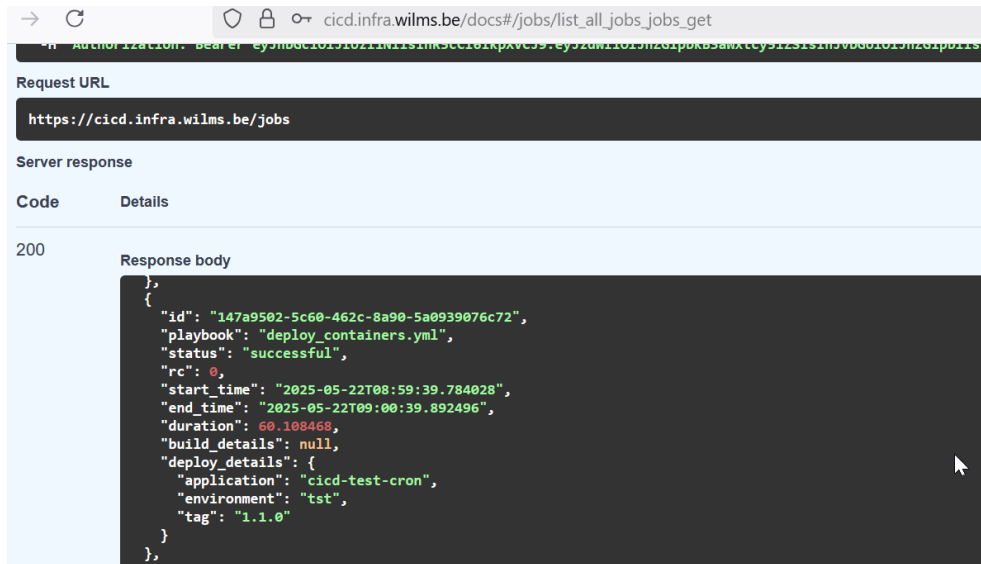


```
141
142 # Full job response with build/deploy references
143 class JobResponse(BaseModel):
144     id: str = Field(..., alias="id")
145     playbook: Optional[str] = None
146     status: Optional[str] = None
147     rc: Optional[int] = None
148     start_time: datetime
149     end_time: Optional[datetime] = None
150     duration: Optional[float] = None
151     build_details: Optional[BuildContainerResponse] = None
152     deploy_details: Optional[DeployContainerResponse] = None
153
154 class Config:
155     orm_mode = True
156     allow_population_by_field_name = True
157
```

Figuur 3-18 Pydantic JobResponse model

Door `orm_mode = True` in Config wordt dit model compatibel met ORM-objecten (zoals uit een database), wat de integratie met databases eenvoudiger maakt.

Op de onderstaande figuur kun je zien hoe dit responsemodel er in de praktijk op de interface uitziet.



Request URL: `https://cicd.infra.wilms.be/jobs`

Server response

Code	Details
200	<pre>{ "id": "147a9502-5c60-462c-8a90-5a0939076c72", "playbook": "deploy_containers.yml", "status": "successful", "rc": 0, "start_time": "2025-05-22T08:59:39.784028", "end_time": "2025-05-22T09:00:39.892496", "duration": 60.108468, "build_details": null, "deploy_details": { "application": "cicd-test-cron", "environment": "tst", "tag": "1.1.0" } }</pre>

Figuur 3-19 /jobs endpoint response body

3.4.3. Login

Voor de login hadden we gekozen op een token based authenticatie methode via OAuth 2.0

Om dit te implementeren heb ik de ingebouwde fastapi.security module gebruikt en heb ik een OAuth 2.0 scheme geïmplementeerd. De gebruiker kan dan als volgt via het /token endpoint via zijn email en password inloggen. De gebruiker krijgt dan een token die gebruikt kan worden om calls te maken naar andere endpoints van de api.

De endpoints vereisen de juiste autorisatie om bepaalde functies te kunnen uitvoeren. Zo kan een developer bijvoorbeeld niet een gebruiker aanmaken of verwijderen.

Hier kan je zien wat er bijvoorbeeld gebeurt als een niet aangemelde gebruiker een call probeert te maken naar /jobs, de gebruiker krijgt een code 401 "Not Authenticated".

The screenshot shows a REST client interface with the following sections:

- No parameters**: A section at the top with an "Execute" button and a "Clear" button.
- Responses**: A section containing details of the request and response.
- Curl**: A code block showing the curl command:

```
curl -X 'GET' \
  'https://cicd.infra.wilms.be/jobs' \
  -H 'accept: application/json'
```
- Request URL**: A text field containing `https://cicd.infra.wilms.be/jobs`.
- Server response**: A table with two columns: "Code" and "Details".

Code	Details
401 <small>Undocumented</small>	Error: response status is 401
- Response body**: A code block showing the JSON response:

```
{
  "detail": "Not authenticated"
}
```

Figuur 3-20 Unauthenticated request

In de onderstaande figuur log ik in via mijn email en wachtwoord op de api

The screenshot shows the 'Available authorizations' dialog in the Swagger UI. The dialog title is 'Available authorizations' with a close button (X). The text inside explains that scopes are used to grant different levels of access and that the API requires the following scopes. Below this, the 'OAuth2PasswordBearer (OAuth2, password)' flow is selected. The 'Token URL' is 'token' and the 'Flow' is 'password'. The 'username' field contains 'admin@wilms.be' and the 'password' field is filled with dots. The 'Client credentials location' is set to 'Authorization header' via a dropdown menu. There are empty fields for 'client_id' and 'client_secret'. At the bottom, there are 'Authorize' and 'Close' buttons.

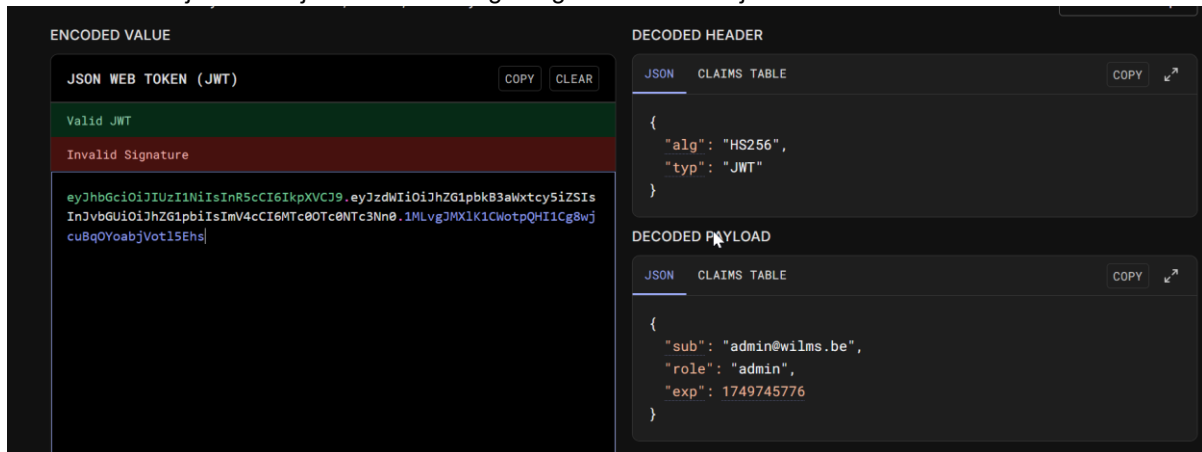
Figuur 3-21 Login page swagger page API

Nadat ik geauthentiseerd ben kan ik bijvoorbeeld wel een call maken naar het /jobs endpoint

The screenshot shows the 'Responses' tab in the Swagger UI. It displays a successful GET request to the '/jobs' endpoint. The 'Curl' section shows the command: `curl -X 'GET' \ 'https://cid.infra.wilms.be/jobs' \ -H 'accept: application/json' \ -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbk83aWxtcy5lZSIsInVubGU0IjoiZG1pbG91IiwiaWF0IjoiMTY5OTY5OH0.8k1czvY7S1yduX7XW6ZwJp1mhiJi7Adre...`. The 'Request URL' is 'https://cid.infra.wilms.be/jobs'. The 'Server response' section shows a '200' status code. The 'Response body' is a JSON array containing one object with details about a failed deployment, including fields like 'id', 'playbook', 'status', 'rc', 'start_time', 'end_time', 'duration', 'build_details', and 'deploy_details'.

Figuur 3-22 Authenticated request

Als we verder kijken in de jwt token die we gekregen hebben kan je ook zien welke rol deze bevat.



Figuur 3-23 Inspectie JWT token

3.4.4. Token managers

Binnen de API zijn er verschillende authenticatiestromen naar externe systemen, zoals Azure DevOps (voor het uitvoeren van Git-clones) en OpenBao (voor het ophalen van configuratie en secrets van applicaties).

Daarom is er een authenticatiemechanisme ingebouwd in de API, zodat er op een gestandaardiseerde en veilige manier verbinding kan worden gemaakt met deze externe systemen.

Voor Azure heb ik gekozen om gebruik te maken van de Microsoft Authentication Library (MSAL). Hiervoor heb ik in de API een algemene functie ontwikkeld die kan worden aangeroepen om een Personal Access Token (PAT) op te halen via de Service Principal van de AppRole van de API in Azure.

De ontwikkelde tokenmanager vraagt een PAT-token aan en houdt deze in memory bij tot de vervaldatum nadert, met een veiligheidsmarge van 15 minuten. Zodra de resterende geldigheidsduur bijvoorbeeld nog slechts 12 minuten bedraagt, vraagt de tokenmanager automatisch een nieuw token aan bij Azure.

Deze aanpak voorkomt dat bij elke aanvraag binnen de applicatie een nieuwe token wordt opgevraagd. Dit is niet alleen veiliger—omdat de client ID en secret niet voortdurend over de lijn worden gestuurd—maar zorgt er ook voor dat Azure-tokens efficiënt worden gebruikt.

Zie onderstaande figuren.

auth.py

Contents History Compare Blame

```
90
91 # -----
92 # AzureTokenManager (MSAL-based)
93 # -----
94 """
95 Retrieves and manages Azure AD token using credentials stored in Vault.
96 """
97 class AzureTokenManager:
98     def __init__(self, vault_manager: VaultTokenManager, secret_path: str):
99         self.vault_manager = vault_manager
100         self.secret_path = secret_path
101         self.token = None
102         self.expiry = None
103         self.lock = threading.Lock()
104
105     def get_token(self) -> str:
106         with self.lock:
107             if not self.token or self._is_expired():
108                 try:
109                     self._refresh_token()
110                 except Exception as e:
111                     logger.error("Error refreshing Azure token: %s", e)
112                     raise
113             return self.token
114
115     def _is_expired(self):
116         return not self.expiry or datetime.now(timezone.utc) >= (self.expiry - timedelta(minutes=5))
117
118     """
119     1. Fetch SP credentials from Vault.
120     2. Use MSAL to acquire an Azure access token.
121     """
122     def _refresh_token(self):
123         logger.info("[Azure] Getting SP creds from Openbao...")
124         try:
125             vault_token = self.vault_manager.get_token()
126             headers = {"X-Vault-Token": vault_token}
127             url = f"{self.vault_manager.vault_addr}/v1/{self.secret_path}"
128             resp = requests.get(url, headers=headers, verify=False)
129
130             if not resp.ok:
131                 logger.error("[Azure] Failed to get SP creds from Openbao, status code: %s", resp.status_code)
132                 raise Exception("Failed to get SP creds from Openbao due to an error response.")
133
134             data = resp.json().get("data", {}).get("data")
135             if not data:
136                 logger.error("[Azure] No SP creds data found in response.")
137                 raise Exception("Failed to get SP creds from Openbao: Missing data")
138
139             client_id = data.get("CLIENT_ID")
140             client_secret = data.get("CLIENT_SECRET")
141             tenant_id = data.get("TENANT_ID")
```

Figuur 3-24 auth.py azuretokenmanager

```
142
143     if not client_id or not client_secret or not tenant_id:
144         logger.error("[Azure] Incomplete SP creds data received.")
145         raise Exception("Failed to get SP creds from Openbao: Incomplete data")
146
147     logger.info("[Azure] Acquiring token via MSAL...")
148     app = ConfidentialClientApplication(
149         client_id=client_id,
150         client_credential=client_secret,
151         authority=f"https://login.microsoftonline.com/{tenant_id}"
152     )
153
154     result = app.acquire_token_for_client(scopes=["499b84ac-1321-427f-aa17-267ca6975798/.default"])
155
156     if "access_token" not in result:
157         logger.error("[Azure] MSAL token acquisition failed. No access token returned.")
158         raise RuntimeError("MSAL token acquisition failed.")
159
160     self.token = result["access_token"]
161     self.expiry = datetime.now(timezone.utc) + timedelta(seconds=result["expires_in"])
162     logger.info("[Azure] Token acquired, expires at: %s", self.expiry)
163 except Exception as e:
164     logger.error("Exception occurred during Azure token refresh: %s", e)
165     raise
166
167 """
```

Figuur 3-25 auth.py azuretokenmanager

Voor Openbao is hetzelfde mechanisme gebouwd. Een token manager dat via de approle credentials tokens opvraagt en bijhoudt.

3.4.5. Ansible playbooks

De grootste taken zoals het bouwen en deployen van containers gebeurt via ansible-runner en ansible playbooks.

Tijdens mijn stage heb ik verschillende playbooks gemaakt voor verschillende functionaliteiten te creëren. Zo is er een playbook voor het bouwen van containers te faciliteren, een playbook voor het deployen van applicaties, een aparte playbook voor het deployen naar de dmz omgeving, een playbook voor het automatisch aanpassen van de proxy na een deploy, etc.

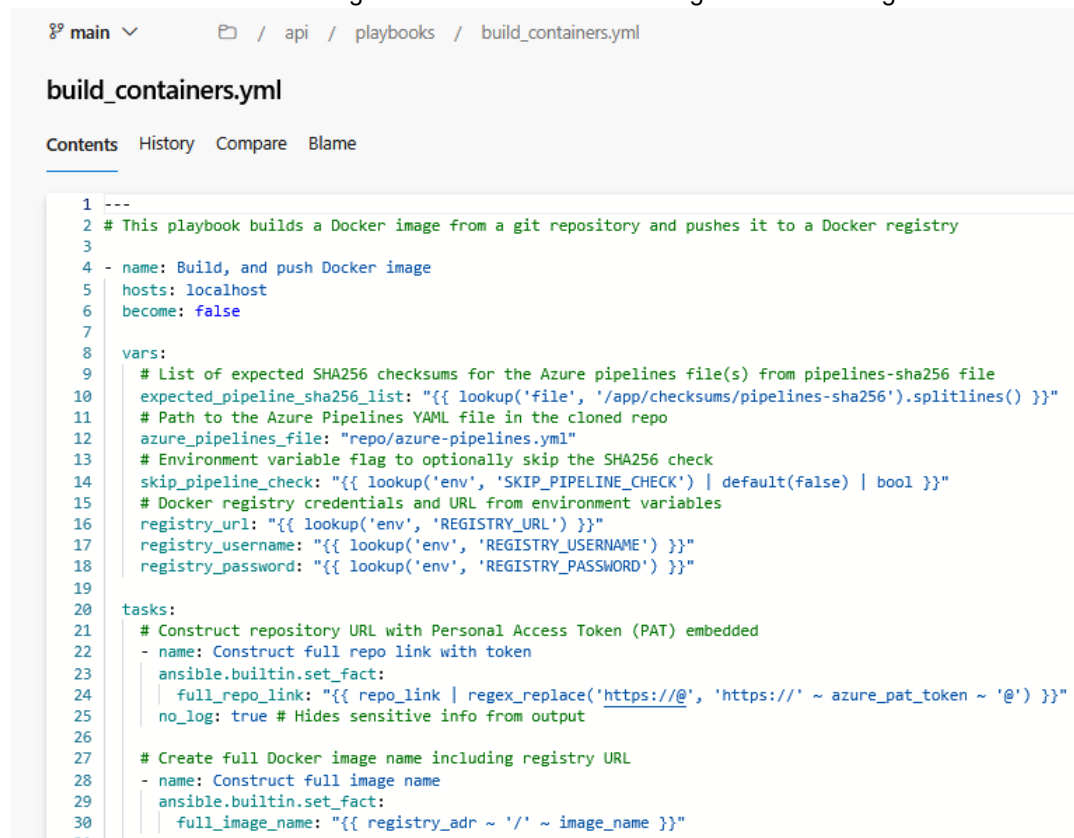
Hieronder ga ik kort de vier belangrijkste playbooks inlichten, wat ze doen en de belangrijkste stappen dat ze nemen.

Eerst het bouwen van containers.

Voor het bouwen van containers heb ik een playbook genaamd **build_containers.yml** gemaakt.

Dit playbook begint met het verzamelen van ingestelde omgevingsvariabelen van de API. Vervolgens stelt het playbook op de API-host (zichzelf) een fact in. Een fact in Ansible kun je vergelijken met een variabele. Deze fact bevat de repo-link die is meegegeven via de API-call binnen de pipeline. Daarnaast wordt er een kortdurende PAT-token van de Service Principal van de API in Azure aan gekoppeld. Deze token is nodig om later geauthenticeerde clones te kunnen uitvoeren.

Ten slotte wordt ook de volledige naam van het containerimage als een fact ingesteld.

The image shows a screenshot of a code editor displaying the 'build_containers.yml' Ansible playbook. The editor has a light gray background with a dark sidebar on the left showing a file tree with 'main' and 'api/playbooks/build_containers.yml'. The top of the editor shows the file path 'api / playbooks / build_containers.yml'. Below the path, the title 'build_containers.yml' is displayed, followed by tabs for 'Contents', 'History', 'Compare', and 'Blame'. The main content area shows the YAML code of the playbook, which is numbered from 1 to 30. The code includes a comment at line 2 stating the purpose of the playbook, followed by play metadata (name, hosts, become) and a 'vars' section with several environment variable lookups. The 'tasks' section contains three tasks: constructing a repository URL with a PAT token, creating a full Docker image name including the registry URL, and another task for setting a fact. The code is syntax-highlighted with colors like green for comments, blue for keywords, and black for values.

```
1 ---
2 # This playbook builds a Docker image from a git repository and pushes it to a Docker registry
3
4 - name: Build, and push Docker image
5   hosts: localhost
6   become: false
7
8   vars:
9     # List of expected SHA256 checksums for the Azure pipelines file(s) from pipelines-sha256 file
10    expected_pipeline_sha256_list: "{{ lookup('file', '/app/checksums/pipelines-sha256').splitlines() }}"
11    # Path to the Azure Pipelines YAML file in the cloned repo
12    azure_pipelines_file: "repo/azure-pipelines.yml"
13    # Environment variable flag to optionally skip the SHA256 check
14    skip_pipeline_check: "{{ lookup('env', 'SKIP_PIPELINE_CHECK') | default(false) | bool }}"
15    # Docker registry credentials and URL from environment variables
16    registry_url: "{{ lookup('env', 'REGISTRY_URL') }}"
17    registry_username: "{{ lookup('env', 'REGISTRY_USERNAME') }}"
18    registry_password: "{{ lookup('env', 'REGISTRY_PASSWORD') }}"
19
20   tasks:
21     # Construct repository URL with Personal Access Token (PAT) embedded
22     - name: Construct full repo link with token
23       ansible.builtin.set_fact:
24         full_repo_link: "{{ repo_link | regex_replace('https://@', 'https://~ azure_pat_token ~ '@') }}"
25         no_log: true # Hides sensitive info from output
26
27     # Create full Docker image name including registry URL
28     - name: Construct full image name
29       ansible.builtin.set_fact:
30         full_image_name: "{{ registry_addr ~ '/' ~ image_name }}"
31
```

Figuur 3-26 Ansible playbook build_containers.yml

Vervolgens wordt de repository gekloond en gecontroleerd of het pipeline-bestand overeenkomt met de bekende pipeline-hashes. Dit is een beveiligingsmechanisme om te voorkomen dat een ontwikkelaar of een kwaadwillende gebruiker zomaar de pipeline in Azure aanpast en ongeautoriseerde code of API-calls uitvoert.

Als de API ziet dat de hash niet overeenkomt, wordt het proces afgebroken en wordt dit gelogd.

```

46 # Notify if pipeline hash check is skipped
47 - name: Debug - Skipping pipeline check?
48   ansible.builtin.debug:
49     msg: "Skipping pipeline hash check because SKIP_PIPELINE_CHECK is true."
50   when: skip_pipeline_check
51
52 # Run sha256sum command on the azure-pipelines.yml file to verify integrity
53 - name: Check SHA256 hash of azure-pipelines.yml
54   ansible.builtin.command: "sha256sum {{ azure_pipelines_file }}"
55   register: sha256_output
56   changed_when: false
57   failed_when: sha256_output.rc != 0
58   when: not skip_pipeline_check
59
60 # Extract only the SHA256 hash value from command output
61 - name: Extract actual SHA256 from command output
62   ansible.builtin.set_fact:
63     actual_sha256: "{{ sha256_output.stdout.split()[0] }}"
64   when: not skip_pipeline_check
65
66 # Fail if the actual SHA256 does not match any of the expected checksums
67 - name: Fail if azure-pipelines.yml hash does not match any expected hash
68   ansible.builtin.fail:
69     msg: "SHA256 mismatch! Expected one of {{ expected_pipeline_sha256_list }}, Got: {{ actual_sha256 }}"
70   when: not skip_pipeline_check and (actual_sha256 not in expected_pipeline_sha256_list)
71

```

Figuur 3-27 Ansible playbook build_containers.yml

Vervolgens wordt er ingelogd op de lokale container registry en wordt de Dockerfile gebouwd met de juiste bestanden, context en build-argumenten die via de API-call door de pipeline zijn meegegeven. Daarna wordt de image naar de registry gepusht.

```

71
72 # Log in to the Docker registry using provided credentials
73 - name: Log in to registry
74   community.docker.docker_login:
75     username: "{{ registry_username }}"
76     password: "{{ registry_password }}"
77     registry: "{{ registry_url }}"
78     reauthorize: true
79
80 # Notify if login was successful
81 - name: Debug - Docker login
82   ansible.builtin.debug:
83     msg: "Logged in to registry successfully."
84
85 # Build the Docker image using specified Dockerfile, context and build args
86 - name: Build Docker image
87   community.docker.docker_image_build:
88     rebuild: always
89     pull: true
90     nocache: true
91     path: "repo{{ dockerfile_context }}"
92     dockerfile: "{{ dockerfile_path | default('Dockerfile') }}"
93     name: "{{ full_image_name }}"
94     tag: "{{ image_tag }}"
95     args: "{{ build_args }}"
96
97 # Notify if the Docker image was built successfully
98 - name: Debug - Docker build
99   ansible.builtin.debug:
100     msg: "Image built successfully."
101
102 # Push the built Docker image to the specified registry
103 - name: Push Docker image to Registry
104   community.docker.docker_image_push:
105     name: "{{ full_image_name }}"
106     tag: "{{ image_tag }}"
107
108 # Final confirmation message with image name and tag
109 - name: Debug - Final confirm message
110   ansible.builtin.debug:
111     msg: "Image: {{ image_name }} with tag: {{ image_tag }} has been built and pushed to {{ registry_url }} successfully."
112

```

Figuur 3-28 Ansible playbook build_containers.yml

Het tweede playbook is **deploy_containers.yml**. Dit playbook is verantwoordelijk voor het deployen van applicatiestacks naar elke omgeving buiten de DMZ.

Net zoals bij het build-playbook worden eerst enkele omgevingsvariabelen verzameld. Vervolgens wordt de infrastructuurrepository, die de single source of truth is voor de Compose-bestanden, gekloond.

Daarna wordt alle metadata van de secrets opgehaald uit OpenBao voor de betreffende applicatie en doelomgeving.

```
31 |
32 | # List available secret keys for a specific app/environment
33 | - name: List keys for the specified application/environment from Openbao
34 |   ansible.builtin.uri:
35 |     url: "{{ base_metadata_url }}/{{ deploy_application }}/{{ deploy_environment }}"
36 |     method: LIST
37 |     headers:
38 |       X-Vault-Token: "{{ api_token }}"
39 |     validate_certs: false
40 |     register: list_response
41 |     no_log: true # Prevents secret data from being printed
42 |
43 | - name: Debug - Output the list of keys
44 |   ansible.builtin.debug:
45 |     var: list_response
46 |
```

Figuur 3-29 Ansible playbook deploy_containers.yml

Vervolgens worden alle versies van het Docker-secret opgehaald voor de betreffende applicatie en omgeving.

```
46 |
47 | # Retrieve version metadata for the 'docker' secret
48 | - name: Retrieve secret versions for docker secret
49 |   ansible.builtin.uri:
50 |     url: "{{ base_metadata_url }}/{{ deploy_application }}/{{ deploy_environment }}/docker"
51 |     method: GET
52 |     headers:
53 |       X-Vault-Token: "{{ api_token }}"
54 |     validate_certs: false
55 |     register: docker_secret_metadata_response
56 |     no_log: true
57 |
```

Figuur 3-30 Ansible playbook deploy_containers.yml

In OpenBao worden alle versies van secrets bijgehouden in de metadata, ook als de betreffende secret is verwijderd.

Om te voorkomen dat een verwijderde versie geselecteerd kan worden, is er een controle ingebouwd die verwijderde secrets negeert.

```
63 # Filter out destroyed versions of the docker secret
64 - name: Set fact for non-destroyed Docker secret versions
65   set_fact:
66     docker_active_versions: >-
67     {{
68       docker_secret_metadata_response.json.data.versions
69       | dict2items
70       | selectattr('value.destroyed', 'equalto', false)
71       | map(attribute='key')
72       | map('int')
73       | sort
74       | list
75     }}
76
77 # Fetch all active docker secret versions
78 - name: Retrieve data for each active Docker secret version
79   ansible.builtin.uri:
80     url: "{{ base_data_url }}/{{ deploy_application }}/{{ deploy_environment }}/docker?version={{ item }}"
81     method: GET
82     headers:
83       X-Vault-Token: "{{ api_token }}"
84     validate_certs: false
85     loop: "{{ docker_active_versions }}"
86     register: docker_secret_data_responses
87     no_log: true
88
89 # comment out for debugging
90 #- name: Debug - Output the retrieved Docker secret data
91 #   ansible.builtin.debug:
92 #     var: docker_secret_data_responses
93
94 # Retrieve other keys' data (excluding 'docker')
95 - name: Retrieve data for each discovered key
96   ansible.builtin.uri:
97     url: "{{ base_data_url }}/{{ deploy_application }}/{{ deploy_environment }}/{{ item }}"
98     method: GET
99     headers:
100       X-Vault-Token: "{{ api_token }}"
101     validate_certs: false
102     loop: "{{ list_response.json.data['keys'] | difference(['docker']) }}"
103     register: data_response
104     no_log: true
105
```

Figuur 3-31 Ansible playbook `deploy_containers.yml`

Daarna wordt in de metadata van de secrets gekeken naar de tag-keys om alle tags te selecteren die overeenkomen met de deployment-tag. Vervolgens wordt een laatste controle uitgevoerd om de meest recente versie te selecteren op basis van de aanmaakdatum. Dit is het uiteindelijke Docker-secret dat wordt gebruikt om de deployment uit te voeren.

```

110
111 # Filter versions that match a specific deploy tag
112 - name: Set fact for versions matching deploy_tag
113   set_fact:
114     matching_tag_versions: >-
115     {{
116       docker_secret_data_responses.results
117       | selectattr('json.data.data.METADATA.tag', 'defined')
118       | selectattr('json.data.data.METADATA.tag', 'equalto', deploy_tag)
119       | map(attribute='item')
120       | list
121     }}
122
123 - name: Debug - Output versions with matching deploy_tag
124   debug:
125     msg: "Version(s) with METADATA.tag == {{ deploy_tag }}: {{ matching_tag_versions }}"
126
127 # Find the newest matching version by creation time
128 - name: Find the newest version from matching_tag_versions
129   set_fact:
130     latest_matching_version: >-
131     {{
132       docker_secret_metadata_response.json.data.versions
133       | dict2items
134       | selectattr('key', 'in', matching_tag_versions | map('string') | list)
135       | sort(attribute='value.created_time', reverse=true)
136       | first
137       | default({})
138       | dict2items
139       | selectattr('key', 'equalto', 'key')
140       | map(attribute='value')
141       | first | int
142     }}
143
144 - name: Show newest matching version
145   debug:
146     msg: "Newest version matching tag {{ deploy_tag }} is: {{ latest_matching_version }}"
147
148 # Fetch the selected docker secret version
149 - name: Retrieve docker secret for latest matching version
150   ansible.builtin.uri:
151     url: "{{ base_data_url }}/{{ deploy_application }}/{{ deploy_environment }}/docker?version={{ latest_matching_version }}"
152     method: GET
153     headers:
154       X-Vault-Token: "{{ api_token }}"
155     validate_certs: false
156     register: latest_docker_secret_response
157     no_log: true
158

```

Figuur 3-32 Ansible playbook *deploy_containers.yml*

Vervolgens worden de overige secrets uitgelezen en omgezet naar .env-bestanden.

```

159
160 # Initialize empty non-docker secret store
161 - name: Init empty non_docker_data
162   set_fact:
163     non_docker_data: {}
164
165 # Combine non-docker secret data
166 - name: Add secret to non_docker_data
167   set_fact:
168     non_docker_data: "{{ non_docker_data | combine({ item.item: item.json.data.data }) }}"
169   loop: "{{ data_response.results }}"
170   when: item.item not in ['docker']
171   no_log: true
172
173 # Merge docker and non-docker secrets into a single structure
174 - name: Combine docker + non-docker secrets
175   set_fact:
176     openbao_data: >-
177     {{
178       {'docker': latest_docker_secret_response.json.data.data}
179       | combine(non_docker_data)
180     }}
181   no_log: true
182
183 # Generate .env files for each key group
184 - name: Render environment files (.env, .env.key) from retrieved data
185   ansible.builtin.template:
186     src: env.j2
187     dest: "./{{ 'env' if item.key == 'docker' else '.env.' ~ item.key }}"
188     vars:
189       current_group_name: "{{ item.key }}"
190       current_group_data: "{{ item.value }}"
191     loop: "{{ openbao_data | dict2items }}"
192     when: item.key not in ['resources', 'resource', 'settings', 'setting']
193     no_log: true
194

```

Figuur 3-33 Ansible playbook *deploy_containers.yml*

Vervolgens worden de benodigde directories aangemaakt op de remote deployment-host en worden de bijbehorende env-bestanden gekopieerd. Daarna wordt de stack opgezet, waarna de Traefik attach-taak wordt uitgevoerd.

```

312
313 # Play 5: Copy generated .env files to remote server
314 - name: Copy .env files to remote server
315   hosts: "{{ deploy_environment }}"
316   become: false
317   gather_facts: no
318
319   tasks:
320     # Debug: Display local .env* files before copying
321     - name: Debug - List local .env* files before copying
322       ansible.builtin.find:
323         paths: .
324         patterns: ".env*"
325         delegate_to: localhost
326
327     # Transfer each local .env file to the appropriate remote path
328     - name: Copy all generated .env files to the remote deployment folder
329       ansible.builtin.copy:
330         src: "{{ item }}"
331         dest: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}/{{ item | basename }}"
332         mode: "0644"
333         backup: true
334         with_fileglob:
335           - "./*.env*"
336

```

Figuur 3-34 Ansible playbook `deploy_containers.yml`

```

'''
378 # Play 6: Deploy the Docker Compose stack (up)
379 - name: Deploy containers using docker compose
380   hosts: "{{ deploy_environment }}"
381   become: false
382   gather_facts: no
383
384   vars:
385     infra_repo: "{{ lookup('env', 'INFRA_REPO') }}"
386
387   tasks:
388     # Start services using Docker Compose (without rebuilding images)
389     - name: Bring up containers using Docker Compose
390       community.docker.docker_compose_v2:
391         project_src: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}"
392         state: present
393         build: never
394         recreate: always # Ensures a fresh state each deployment
395
396 # Play 7: Include Traefik attach task
397 - name: Include Traefik task for network attach
398   hosts: "{{ deploy_environment }}"
399   gather_facts: no
400   become: false
401   vars:
402     traefik_name: "{{ lookup('env', 'TRAEFIK_NAME') | default('traefik', true) }}"
403     traefik_repo_folder: "{{ lookup('env', 'INFRA_REPO_TRAEFIK_FOLDER') | default('proxman', true) }}"
404     proxman_name: "{{ lookup('env', 'INFRA_REPO_TRAEFIK_FOLDER') | default('proxman', true) }}"
405
406   tasks:
407     - ansible.builtin.include_tasks: task_traefik_attach_and_change.yml
408

```

Figuur 3-35 Ansible playbook `deploy_containers.yml`

In het deploy-playbook wordt aan het einde de taak “**task_traefik_attach_and_change.yml**” aangeroepen. Dit aparte task-bestand zorgt er, indien nodig na een deploy, voor dat het Docker-netwerk of meerdere Docker-netwerken met de tag `enabled.traefik` worden toegevoegd aan de container van de proxy. Hierdoor kunnen de gedeployde applicaties die publiek beschikbaar moeten zijn, automatisch van TLS/HTTPS worden voorzien.

Eerst wordt de volledige Docker-hostinformatie opgevraagd op de deployment-host. Vervolgens filter ik de netwerken en sla ik de bijbehorende IDs op. Daarna vraag ik voor alle Docker-netwerk-IDs de labels op en koppel deze aan hun ID.

Vervolgens maak ik een lijst waarin ik alle Docker-netwerken met hun ID opsla die de tag `enabled.traefik` bevatten.

```

1 # Retrieve Docker host information including available networks
2 - name: Retrieve Docker host info (including networks)
3   community.docker.docker_host_info:
4     networks: true
5   register: docker_host_info
6
7 # Extract only the network IDs from the retrieved network info
8 - name: Set fact - Extract only network IDs
9   ansible.builtin.set_fact:
10     docker_network_ids: >-
11       | {{ docker_host_info.networks | map(attribute='Id') | list }}
12
13 # Debug output of all discovered network IDs
14 - name: Debug - Output network IDs only
15   ansible.builtin.debug:
16     var: docker_network_ids
17
18 # Get detailed metadata for each Docker network using their IDs
19 - name: Get detailed info for each Docker network by ID
20   community.docker.docker_network_info:
21     name: "{{ item }}"
22   loop: "{{ docker_network_ids }}"
23   register: detailed_network_info
24
25 # Build a dictionary mapping network IDs to their label metadata
26 - name: Build dictionary mapping network IDs to their labels
27   ansible.builtin.set_fact:
28     network_labels: "{{ detailed_network_info.results | map(attribute='network') | items2dict(key_name='Id', value_name='Labels') }}"
29
30 # Debug output of the mapping between network IDs and their labels
31 - name: Debug - Output network IDs and their labels
32   ansible.builtin.debug:
33     var: network_labels
34
35 # Initialize an empty list to hold IDs of networks with Traefik enabled
36 - name: Initialize list for traefik-enabled networks
37   ansible.builtin.set_fact:
38     traefik_enabled_network_ids: []
39
40 # Add networks labeled as Traefik-enabled to the list
41 - name: Add traefik-enabled networks to list (by ID)
42   ansible.builtin.set_fact:
43     traefik_enabled_network_ids: "{{ traefik_enabled_network_ids + [item.key] }}"
44   when: item.value is defined and item.value['enabled.traefik'] is defined and item.value['enabled.traefik'] == 'true'
45   loop: "{{ network_labels | dict2items }}"
46

```

Figuur 3-36 Ansible playbook task_traefik_attach_and_change.yml

Vervolgens verbind ik de Traefik-proxycontainer met alle netwerken die het label `traefik.enabled` hebben. Hiervoor gebruik ik `append`, zodat een netwerk dat al verbonden is maar geen label heeft, niet wordt losgekoppeld van de container.

Daarna verzamel ik de netwerknaam van alle `traefik.enabled`-netwerken en sla ik deze op. Vervolgens lees ik de bestaande Compose-file van Traefik in, houd ik het basisnetwerk van de Traefik-proxy bij en voeg ik de lijst met alle `traefik.enabled`-netwerknamen hieraan toe. Deze bijgewerkte Compose-file schrijf ik vervolgens weg, zodat deze up-to-date is met de nieuwe deployment.



```
51
52 # Ensure the Traefik container is connected to each Traefik-enabled network
53 - name: Ensure 'proxman_traefik' is connected to all traefik-enabled networks
54   community.docker.docker_network:
55     name: "{{ item }}"
56     connected:
57       - proxman-{{ deploy_environment }}-traefik-01
58     appends: true
59   loop: "{{ traefik_enabled_network_ids }}"
60   loop_control:
61     label: "{{ item }}"
62
63 # Initialize a list to hold names of Traefik-enabled networks
64 - name: Initialize traefik_enabled_network_names as an empty list
65   ansible.builtin.set_fact:
66     traefik_enabled_network_names: []
67
68 # Collect the network names for all Traefik-enabled networks
69 - name: Build list of traefik-enabled network names
70   ansible.builtin.set_fact:
71     traefik_enabled_network_names: "{{ traefik_enabled_network_names + [ item.network.Name ] }}"
72   loop: "{{ detailed_network_info.results }}"
73   when: item.network.Id in traefik_enabled_network_ids
74
75 # Debug output of the network names Traefik will be attached to
76 - name: Debug - Show traefik enabled network names
77   ansible.builtin.debug:
78     var: traefik_enabled_network_names
79
80 # Read the existing Traefik Docker Compose file from disk (base definition)
81 - name: Read existing compose.yml
82   ansible.builtin.slurp:
83     src: "./repo/{{ deploy_environment }}/{{ traefik_repo_folder }}/compose.yml"
84   register: raw_compose
85   delegate_to: localhost
86   run_once: true
87
88 # Parse the YAML structure from the read content
89 - name: Parse compose.yml into a data structure
90   ansible.builtin.set_fact:
91     compose_obj: "{{ raw_compose.content | b64decode | from_yaml }}"
92   delegate_to: localhost
93   run_once: true
94
```

Figuur 3-37 Ansible playbook `task_traefik_attach_and_change.yml`

Als laatste wordt de Compose-file gecommit naar Git, zodat de single source of truth up-to-date blijft. De API voegt hierbij automatisch een gegenereerd commitbericht toe met daarin de toegevoegde netwerken en de datum waarop dit heeft plaatsgevonden, zodat precies kan worden bijgehouden wat er is veranderd.

```

243
244 # Stage the updated file for commit
245 - name: Stage the updated docker-compose file for git commit
246   ansible.builtin.command:
247     argv:
248       - git
249       - add
250       - compose.yml
251     args:
252       chdir: "./repo/{{ deploy_environment }}/{{ traefik_repo_folder }}"
253     delegate_to: localhost
254     run_once: true
255     when:
256       - git_diff.rc != 0
257
258 # Commit changes with an automated message
259 - name: Commit changes to the traefik docker-compose file
260   ansible.builtin.command:
261     argv:
262       - git
263       - commit
264       - -m
265       - "Auto-update Traefik networks from job: {{ job_id }} to traefik compose, added: {{ traefik_enabled_network_names }} to networks on: {{ now(utc=true, fmt='%Y-%m-%dT%H:%M:%SZ') }}"
266     args:
267       chdir: "./repo/{{ deploy_environment }}/{{ traefik_repo_folder }}"
268     delegate_to: localhost
269     run_once: true
270     when:
271       - git_diff.rc != 0
272   ignore_errors: true
273
274 # Push committed changes to the remote Git repository
275 - name: Push committed changes to the remote repository
276   ansible.builtin.command:
277     argv:
278       - git
279       - push
280       - origin
281       - HEAD
282     args:
283       chdir: "./repo/{{ deploy_environment }}/{{ traefik_repo_folder }}"
284     delegate_to: localhost
285     run_once: true
286     when:
  
```

Figuur 3-38 Ansible playbook task_traefik_attach_and_change.yml

Voor deployment naar de DMZ verloopt de procedure grotendeels hetzelfde, maar vanwege veiligheidsmaatregelen in de DMZ-omgeving is het niet toegestaan om verbinding te maken met het interne netwerk. Dit zorgt voor problemen bij de deploymentstap waarbij de Docker Compose-stack wordt opgestart. Omdat communicatie van de DMZ naar het interne netwerk geblokkeerd is, kan de DMZ-host de containerimages van de lokale interne registry niet ophalen.

We hebben verschillende oplossingen overwogen. De eenvoudigste was het draaien van een kopie van de interne registry in de DMZ. Dit heeft echter enkele nadelen:

Ten eerste is er dubbel zoveel opslagruimte nodig, omdat het een exacte kopie betreft.

Een andere mogelijkheid was om een manier te vinden om de images vanuit het interne netwerk naar de DMZ-host te brengen. Dus in plaats van DMZ → intern, zouden we de images intern → DMZ moeten overzetten.

Hiervoor heb ik gekozen voor het gebruik van docker image save en docker image load. Hiermee kunnen de images op de API-host, die beide binnen het interne netwerk draaien, gepulled worden, opgeslagen als een tar-archief, overgezet naar de DMZ en daar ingeladen worden.

Buiten de onderstaande stappen is het playbook voor deployment naar de DMZ gelijk aan het reguliere deployment playbook.

De image- en tagparen worden uit de secret in OpenBao gehaald en samengevoegd, zodat ze later gebruikt kunnen worden om de juiste namen te genereren.

deploy_containers_dmz.yml

[Contents](#) [History](#) [Compare](#) [Blame](#)

```
197 # Extract _IMAGE and _TAG from docker secrets for use in tagging
198 - name: Extract _IMAGE and _TAG items from openbao_data.docker
199   set_fact:
200     image_vars: >-
201     {{
202       openbao_data.docker
203       | dict2items
204       | selectattr('key', 'search', '_IMAGE$')
205       | list
206     }}
207     tag_vars: >-
208     {{
209       openbao_data.docker
210       | dict2items
211       | selectattr('key', 'search', '_TAG$')
212       | list
213     }}
214   delegate_to: localhost
215   no_log: true
216
217 - name: Initialize image_tag_pairs
218   set_fact:
219     image_tag_pairs: []
220   delegate_to: localhost
221
222 # Create a match list of image:tag combinations from docker secrets
223 - name: Construct image:tag list from openbao_data.docker
224   set_fact:
225     image_tag_pairs: "{{ image_tag_pairs + [image ~ ':' ~ tag] }}"
226   loop: "{{ image_vars }}"
227   vars:
228     # Extract the prefix used for both image and tag, e.g., FOO_IMAGE -> FOO
229     prefix: "{{ item.key | regex_replace('_IMAGE$', '') }}"
230     image: "{{ item.value }}"
231     # Find corresponding tag using the same prefix
232     tag: >-
233     {{
234       tag_vars
235       | selectattr('key', 'equalto', prefix ~ '_TAG')
236       | map(attribute='value')
237       | first | default('latest')
238     }}
239   delegate_to: localhost
240   no_log: true
```

Figuur 3-39 Ansible playbook `deploy_containers_dmz.yml`

Tijdens het deploy-gedeelte worden de images lokaal op de API-host gepulled via het Compose-bestand. Hiervoor heb ik de `compose_pull`-module gebruikt, zodat de images alleen gepulled worden en niet opgestart.

Vervolgens worden de images opgeslagen als tar-archieven, waarbij de bestandsnamen gebaseerd zijn op de eerder samengestelde image- en tagparen.

De tar-archieven worden daarna gekopieerd naar de DMZ-host, ingeladen in de Docker daemon van die host, en vervolgens verwijderd.

Tot slot wordt de `docker_compose`-module gebruikt om de stack op te starten. In de module is de optie "pull" op never gezet, zodat de DMZ-host nooit zelf probeert images te pullen wanneer deze lokaal niet beschikbaar zijn.

```
deploy_containers_dmz.yml

Contents History Compare Blame

427 tasks:
428 - name: Pull the images from the compose file locally
429   community.docker.docker_compose_v2_pull:
430     project_src: "."
431     register: pulled_images
432     delegate_to: localhost
433     no_log: true
434
435 - name: Save the pulled images to tar files
436   community.docker.docker_image_export:
437     names: "{{ item }}"
438     path: "/opt/docker/{{ item | regex_replace('/:/', '_') }}.tar"
439     loop: "{{ hostvars['localhost']['image_tag_pairs'] }}"
440     delegate_to: localhost
441
442 - name: Copy the tar files to the remote server
443   ansible.builtin.copy:
444     src: "/opt/docker/{{ item | regex_replace('/:/', '_') }}.tar"
445     dest: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}/{{ item | regex_replace('/:/', '_') }}.tar"
446     mode: "0644"
447     loop: "{{ hostvars['localhost']['image_tag_pairs'] }}"
448
449 - name: Load Docker images from tar files on remote host
450   community.docker.docker_image_load:
451     path: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}/{{ item | regex_replace('/:/', '_') }}.tar"
452     loop: "{{ hostvars['localhost']['image_tag_pairs'] }}"
453     register: load_results
454
455 - name: Print loaded image names
456   debug:
457     msg: "Loaded images from {{ item.item }}: {{ item.image_names | default([]) | join(', ') }}"
458     loop: "{{ load_results.results }}"
459
460 - name: Remove tar files from remote after loading
461   file:
462     path: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}/{{ item | regex_replace('/:/', '_') }}.tar"
463     state: absent
464     loop: "{{ hostvars['localhost']['image_tag_pairs'] }}"
465
466 - name: Bring up containers using Docker Compose
467   community.docker.docker_compose_v2:
468     project_src: "/opt/docker/{{ deploy_environment }}/{{ deploy_application }}"
469     state: present
470     build: never
471     recreate: always
472     pull: never
```

Figuur 3-40 Ansible playbook `deploy_containers_dmz.yml`

3.4.6. Logging

Voor logging heb ik centrale logging via Seq ingesteld. Hiervoor heb ik de Python-module seqlog gebruikt. Vervolgens heb ik een Seq-logger geconfigureerd die de logs van Python opvangt. Deze wordt bij het opstarten van de applicatie geïnitieerd, zodat logging vanaf het begin beschikbaar is.

main.py

[Contents](#) [History](#) [Compare](#) [Blame](#)

```
1 # main.py
2
3 import os
4 import logging
5 import seqlog
6 from fastapi import FastAPI, HTTPException, Depends, Request, Response
7 from sqlalchemy.orm import Session
8 import uvicorn
9 import crud
10 import models
11 import schemas
12 import auth
13 import docs
14 import asyncio
15 import httpx
16 import truststore
17 import ssl
18 import metrics
19 import yaml
20 from pydantic import ValidationError
21
22 from schemas import BuildVars, DeployVars, BumpTagRequest, ManualPlaybookVars, AddSecretsRequest
23 from models import Job, UserRole, ContainerStatus
24 from database import engine, SessionLocal, get_db
25 from functions import start_build_container, start_deploy_container, start_bump_secret_tags, periodic_container_sync, start_manual_playbook, start_add_secrets
26 from auth import check_allowed_roles
27 from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
28 from fastapi.openapi.utils import get_openapi
29 from fastapi.routing import APIRouter
30 from managers import azure_token_manager, vault_manager
31
32 from prometheus_fastapi_instrumentator import Instrumentator
33 from prometheus_client import generate_latest, CONTENT_TYPE_LATEST
34
35 SEQ_SERVER_URL = os.getenv('SEQ_URL')
36 SEQ_API_TOKEN = os.getenv('SEQ_TOKEN')
37 APPLICATION_NAME = "Infrastructure-Automation-API"
38
39
40 # Set up centralized Seq logging/logger
41 seqlog.log_to_seq(
42     server_url=SEQ_SERVER_URL,
43     api_key=SEQ_API_TOKEN,
44     level=logging.INFO,
45     batch_size=10,
46     auto_flush_timeout=10
47 )
48 seqlog.set_global_log_properties(Application=APPLICATION_NAME)
49
50 logger = logging.getLogger(__name__)
51
```

Figuur 3-41 API main.py

3.4.7. Metrics

Voor metrics hadden we de keuze gemaakt om de API te integreren met de bestaande Prometheus- en Grafana-stack die intern bij Wilms draait.

Hiervoor heb ik de Python-modules `prometheus_fastapi_instrumentator` en `prometheus_client` gebruikt.

Voor de metrics heb ik verschillende definities opgesteld, waaronder een gauge die het aantal jobs per status weergeeft.

```
8
9 # === METRIC DEFINITIONS ===
10
11 JOBS_BY_STATUS = Gauge(
12     "infra_api_jobs_by_status_total",
13     "Total number of jobs per status",
14     labelnames=["status"]
15 )
16
```

Figuur 3-42 Metrics job status gauge

Een histogram dat het aantal jobs in tijdsbuckets bijhoudt, om zo een overzicht te krijgen van hoelang een gemiddelde job duurt. Dit is gefilterd op het gebruikte playbook, bijvoorbeeld build of deploy.

```
17 JOBS_DURATION = Histogram(
18     "infra_api_jobs_duration_seconds",
19     "Histogram of job durations in seconds",
20     buckets=(1, 5, 10, 30, 60, 120, 300, 600, 1800, 3600, float("inf")),
21     labelnames=["playbook"]
22 )
23
```

Figuur 3-43 Metrics job duration bucket

Ook had ik een counter toegevoegd om het aantal gestarte jobs weer te geven.

```
23
24 JOBS_STARTED = PromCounter(
25     "infra_api_jobs_started_total",
26     "Total number of jobs started by playbook",
27     labelnames=["playbook"]
28 )
29
```

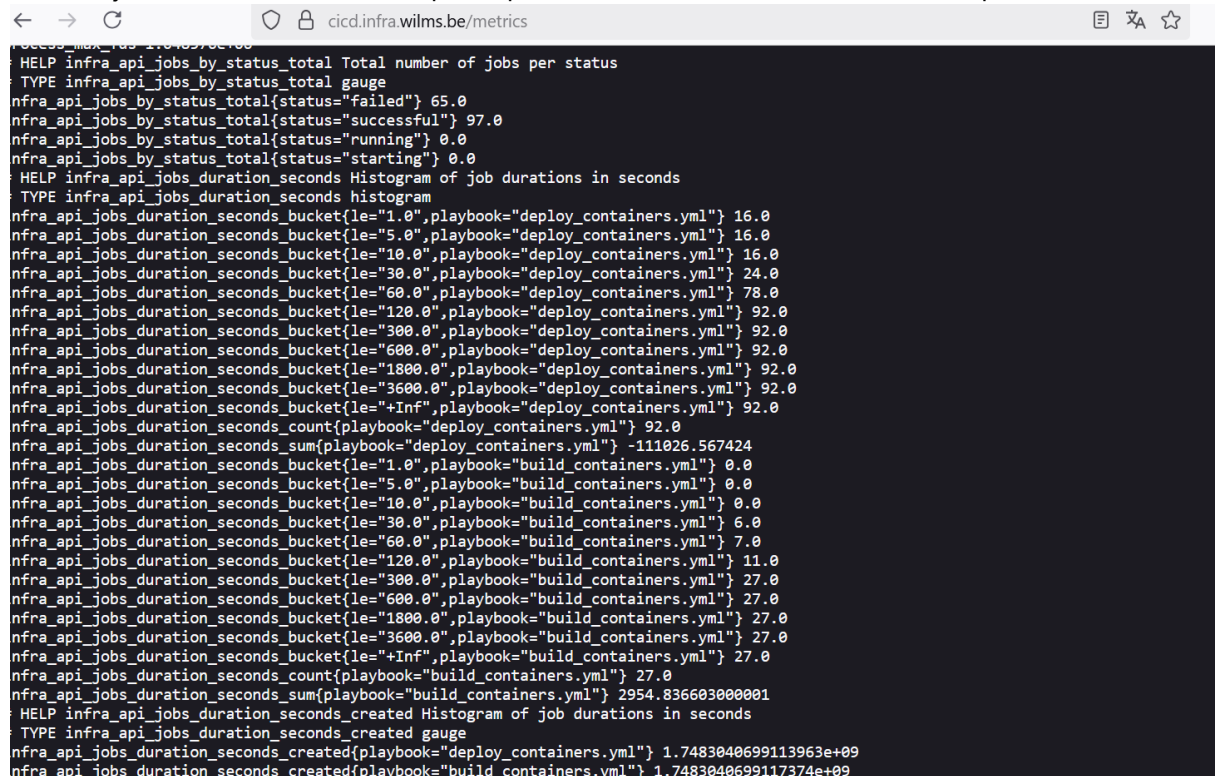
Figuur 3-44 Metrics jobs started counter

Daarnaast is er een gauge voor het aantal actieve jobs.

```
29
30 JOBS_IN_PROGRESS = Gauge(
31     "infra_api_jobs_in_progress_total",
32     "Number of jobs currently in progress"
33 )
34
```

Figuur 3-45 Metrics jobs in progress gauge

Uiteindelijk levert dit een /metrics-endpoint op waar Prometheus data van de API kan ophalen.



```
HELP nfra_api_jobs_by_status_total Total number of jobs per status
TYPE nfra_api_jobs_by_status_total gauge
nfra_api_jobs_by_status_total{status="failed"} 65.0
nfra_api_jobs_by_status_total{status="successful"} 97.0
nfra_api_jobs_by_status_total{status="running"} 0.0
nfra_api_jobs_by_status_total{status="starting"} 0.0
HELP nfra_api_jobs_duration_seconds Histogram of job durations in seconds
TYPE nfra_api_jobs_duration_seconds histogram
nfra_api_jobs_duration_seconds_bucket{le="1.0",playbook="deploy_containers.yml"} 16.0
nfra_api_jobs_duration_seconds_bucket{le="5.0",playbook="deploy_containers.yml"} 16.0
nfra_api_jobs_duration_seconds_bucket{le="10.0",playbook="deploy_containers.yml"} 16.0
nfra_api_jobs_duration_seconds_bucket{le="30.0",playbook="deploy_containers.yml"} 24.0
nfra_api_jobs_duration_seconds_bucket{le="60.0",playbook="deploy_containers.yml"} 78.0
nfra_api_jobs_duration_seconds_bucket{le="120.0",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_bucket{le="300.0",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_bucket{le="600.0",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_bucket{le="1800.0",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_bucket{le="3600.0",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_bucket{le="+Inf",playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_count{playbook="deploy_containers.yml"} 92.0
nfra_api_jobs_duration_seconds_sum{playbook="deploy_containers.yml"} -111026.567424
nfra_api_jobs_duration_seconds_bucket{le="1.0",playbook="build_containers.yml"} 0.0
nfra_api_jobs_duration_seconds_bucket{le="5.0",playbook="build_containers.yml"} 0.0
nfra_api_jobs_duration_seconds_bucket{le="10.0",playbook="build_containers.yml"} 0.0
nfra_api_jobs_duration_seconds_bucket{le="30.0",playbook="build_containers.yml"} 6.0
nfra_api_jobs_duration_seconds_bucket{le="60.0",playbook="build_containers.yml"} 7.0
nfra_api_jobs_duration_seconds_bucket{le="120.0",playbook="build_containers.yml"} 11.0
nfra_api_jobs_duration_seconds_bucket{le="300.0",playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_bucket{le="600.0",playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_bucket{le="1800.0",playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_bucket{le="3600.0",playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_bucket{le="+Inf",playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_count{playbook="build_containers.yml"} 27.0
nfra_api_jobs_duration_seconds_sum{playbook="build_containers.yml"} 2954.836603000001
HELP nfra_api_jobs_duration_seconds_created Histogram of job durations in seconds
TYPE nfra_api_jobs_duration_seconds_created gauge
nfra_api_jobs_duration_seconds_created{playbook="deploy_containers.yml"} 1.7483040699113963e+09
nfra_api_jobs_duration_seconds_created{playbook="build_containers.yml"} 1.7483040699117374e+09
```

Figuur 3-46 Metrics endpoint

3.4.8. Endpoints

De custom API heeft uiteindelijk 21 endpoints, verdeeld over negen categorieën:

De categorieën zijn als volgt:

- Users
- Runners
- Jobs
- Containers
- Apps
- Secrets
- Azure
- Metrics
- Utility

Users: bevat alles wat te maken heeft met het aanmaken, verwijderen en opvragen van gebruikers. Ook het token-endpoint voor authenticatie valt hieronder.

users Operations/Endpoints that involve users. The login/token logic is also defined here.		^
POST	/token Login For Access Token	⌵
GET	/users Read Users	⌵
POST	/users Create User	⌵
GET	/users/{user_id} Read User	⌵
DELETE	/users/email/{email} Delete User By Email	⌵

Figuur 3-47 Swagger user categorie

Runners: bevat alles wat Ansible Runner op de achtergrond aanstuurt. Hieronder vallen bijvoorbeeld het /build-async endpoint om containers te bouwen, het /deploy-async endpoint om applicaties te deployen, en het endpoint om handmatig eigen playbooks te starten, zoals bijvoorbeeld een database-backup playbook.

runners Operations that involve ansible runners and playbooks.		^
POST	/build_async Build Container Async	⌵
POST	/deploy_async Deploy Container Async	⌵
POST	/run_manual_playbook Run Manual Playbook Async	⌵

Figuur 3-48 Swagger runners categorie

Jobs: bevat alles wat te maken heeft met het opvragen van Ansible Runner jobs.

jobs Operations that involve ansible jobs.		^
GET	/jobs List All Jobs	⌵
GET	/job/{job_id} Get Job Status By Id	⌵

Figuur 3-49 Swagger jobs categorie

Containers: Containers: bevat alle endpoints voor het opvragen van draaiende containers in de verschillende omgevingen.

containers Operations that involve retrieving container info.		^
GET	/containers Read Containers	🔒 ▼
GET	/containers/host/{hostname} Read Containers By Hostname	🔒 ▼
GET	/containers/project/{project} Read Containers By Project	🔒 ▼

Figuur 3-50 Swagger containers categorie

Apps: bevat alle endpoints voor het opvragen van de beschikbare apps die gedeployed kunnen worden, inclusief de mogelijke omgevingen en versies/tags waarin ze gedeployed kunnen worden.

apps Operations that involve retrieving info for and of deployable apps.		^
GET	/deployable_apps List Deployable Apps	🔒 ▼
GET	/deployable_apps/{app}/environments List Deployable App Environments	🔒 ▼
GET	/deployable_apps/{app}/tags List Deployable App Tags	🔒 ▼

Figuur 3-51 Swagger apps categorie

Secrets: bevat alle endpoints die aanpassingen kunnen maken binnen OpenBao. Bijvoorbeeld een version bump van een applicatie na een nieuwe build, of het (bulk) importeren van secrets voor een bepaalde applicatie.

secrets Operations that involve adding and/or bumping secrets in openbao.		^
POST	/secrets/bump Bump Secret Tags Endpoint	🔒 ▼
POST	/secrets/add Add Secrets	🔒 ▼

Figuur 3-52 Swagger secrets categorie

Azure: bevat het endpoint om de huidige PAT-token handmatig op te vragen.

azure Operations that involve Azure.		^
GET	/azure_token Get Azure Pat	🔒 ▼

Figuur 3-53 Swagger azure categorie

Metrics: is het endpoint dat Prometheus gebruikt om metrics te scrapen, zodat deze centraal in de monitoring stack geïmporteerd kunnen worden.

metrics Operations that involve prometheus metrics.		^
GET	/metrics Serve Metrics	▼

Figuur 3-54 Swagger metrics categorie

Utility: bevat endpoints zoals cleanup, bijvoorbeeld om de initiële admin user te verwijderen na de eerste opstart.



Figuur 3-55 Swagger utility categorie

3.5. Documentatie

Om het eerder beschreven project goed te documenteren en in kaart te brengen heb ik gebruik gemaakt van verschillende manieren van documenteren in het project.

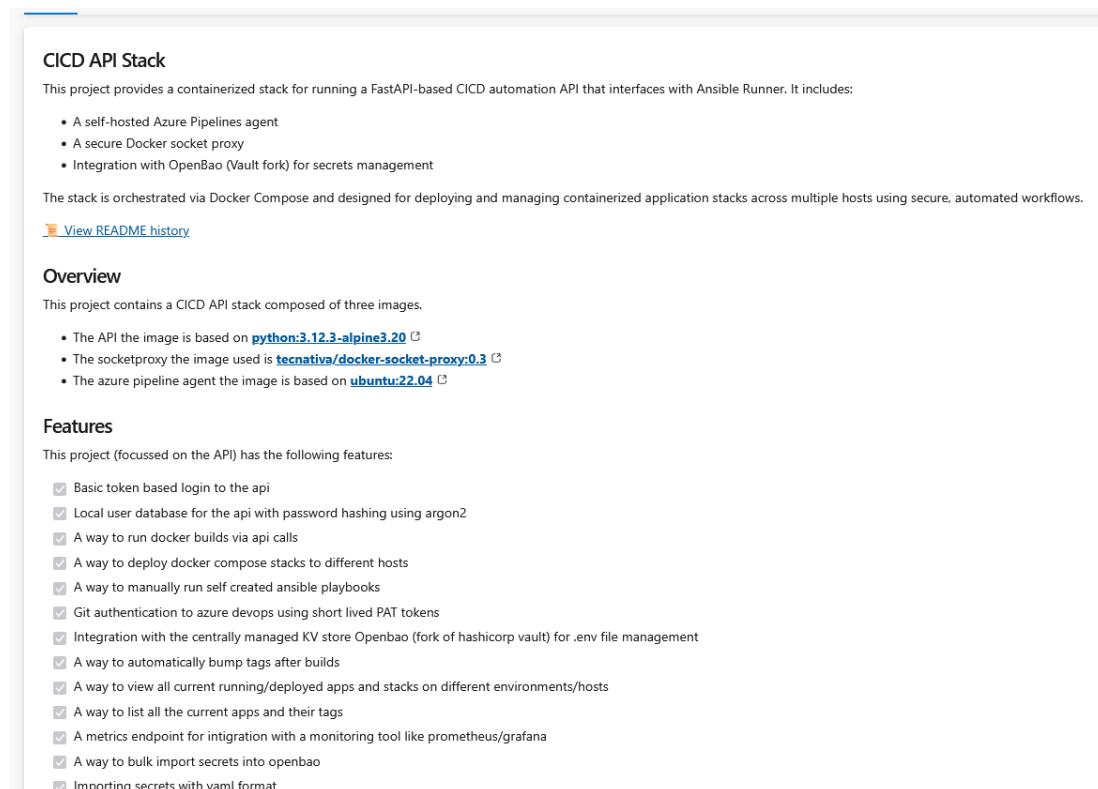
In de volledige code staan er code comments dat meer inzicht geven over wat bepaalde delen van de code doen.

Doorheen de volledige fastapi python code en de ansible playbook is dit aangehouden.

Voor de api zelf heb ik gebruikt gemaakt van de eerder getoonde swagger docs. Swagger docs geeft een snelle en makkelijke en overzichtelijke manier om documentatie te schrijven en te genereren voor de api

Ten slotte heb ik voor het project in de definitieve repo van Wilms een uitgbereide readme geschreven, deze readme bevat code blocks for een quick setup, uitleg van de nodige structuren bijvoorbeeld de secret structuur in Openbao. Communicatie diagrammen dat elk proces meer in kaart brengt.

In de onderstaande diagrammen kan u verschillende snippets terugvinden van de opgeleverde readme en de bijgevoegde communicatiediagrammen.



Figuur 3-56 API readme

Contents

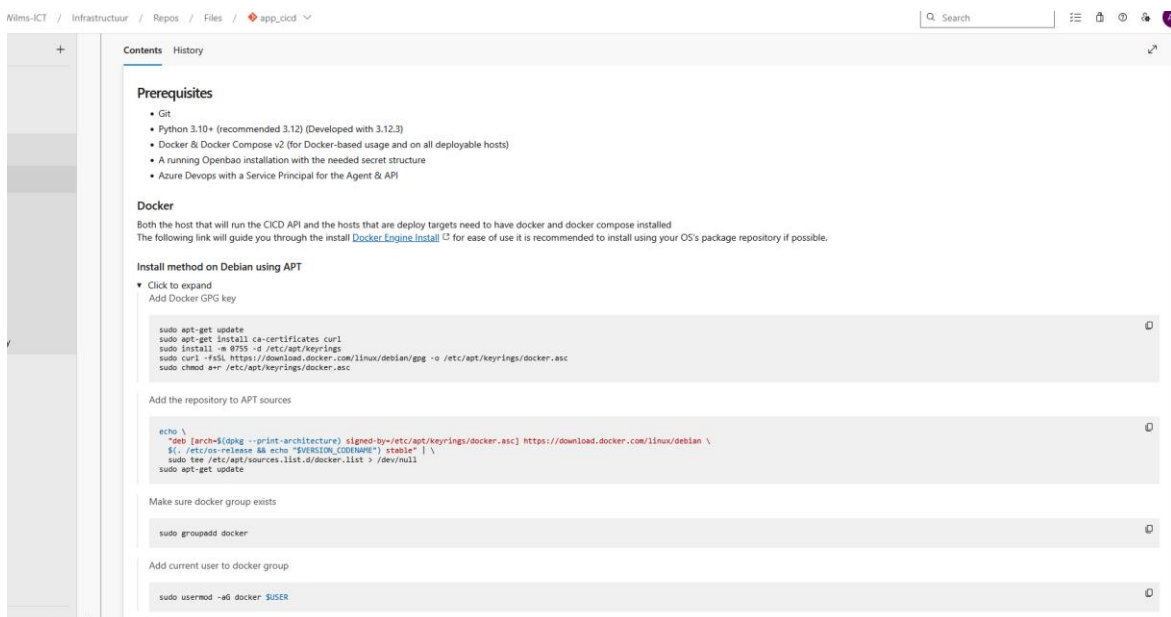
History

☒ Deploying to DMZ environments where access to an internal or external registry is forbidden

Table Of Contents

Contents

- [CICD API Stack](#)
- [Overview](#)
- [Features](#)
- [Table Of Contents](#)
- [Prerequisites](#)
 - [Docker](#)
 - [Install method on Debian using APT](#)
 - [Openbao](#)
 - [Structure](#)
 - [Azure SP Structure](#)
 - [Application Structure](#)
 - [General Structure](#)
 - [ex. infra_stage in two environments](#)
 - [Policies](#)
 - [Approle](#)
 - [KVV2](#)
- [Requirements](#)
 - [Used Packages](#)
 - [Used Ansible modules](#)
- [Environment Variables](#)
 - [Docker](#)
 - [CICD-API](#)
 - [Azure Pipeline Agent](#)
 - [Docker Socket Proxy](#)
- [Usage](#)
 - [Basic usage](#)
 - [docker compose \(recommended \)](#)
 - [Bare Metal API only](#)
 - [Communication flows during usage](#)
 - [Build flow](#)
 - [Deploy flow](#)

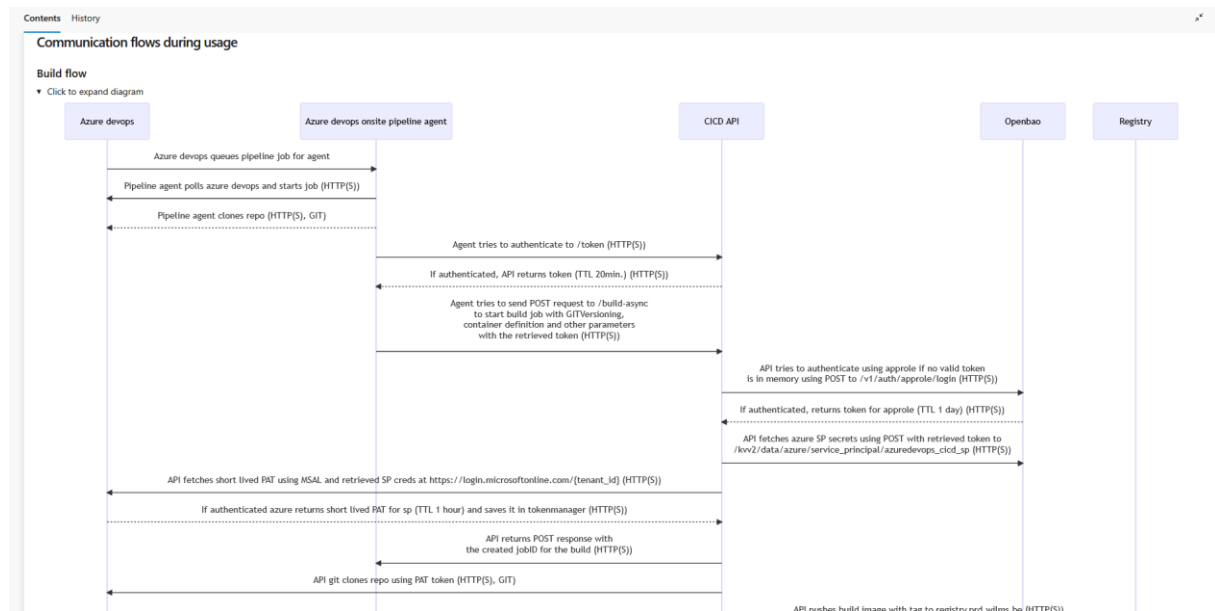


<div> <div>Contents</div> <div>History</div> </div> <div> <div>Click to expand</div> <div> <div>Requirements</div> <div> <div>Python packages (see requirements.txt)</div> <div> <div>Ansible collections:</div> <div>community.docker</div> </div> </div> </div> <div> <div>Used Packages</div> <div> <div>To run the API the following python libraries/packages as defined in requirements.txt, when using docker the installation of these packages is already included.</div> <table> <tr> <th>Package</th><th>Version</th><th>Use/Description</th></tr> <tr> <td>ansible</td><td>11.5.0</td><td>Core library to install ansible capabilities on the CICD host/container</td></tr> <tr> <td>aiofiles</td><td>23.2.1</td><td>Library to handle async handle local file operations. Needed for reading files like pipeline checksum for instance</td></tr> <tr> <td>ansible-runner</td><td>2.4.1</td><td>Library to make interfacing with ansible easier and more efficient inside of python code</td></tr> <tr> <td>fastapi</td><td>0.115.12</td><td>Framework to build API's with python</td></tr> <tr> <td>pydantic</td><td>1.10.22</td><td>Library needed for data validation, mostly used in the validation of parameters and bodies in API calls</td></tr> <tr> <td>SQLAlchemy</td><td>1.4.54</td><td>An ORM library used for sql operations inside of the API</td></tr> <tr> <td>uvicorn</td><td>0.34.2</td><td>A ASGI web server for python that can handle async frameworks</td></tr> <tr> <td>requests</td><td>2.32.3</td><td>Library used for fast and quick HTTP operations</td></tr> <tr> <td>seqlog</td><td>0.4.1</td><td>Library used to log to Seq</td></tr> <tr> <td>pip-system-certs</td><td>4.0</td><td>A library used to make pip use the default system trust store instead of the bundled CA's</td></tr> <tr> <td>python-multipart</td><td>0.0.20</td><td>Python streaming multipart parser</td></tr> <tr> <td>python-jose</td><td>3.4.0</td><td>A Javascript/JSON based encryption</td></tr> </table> </div> </div> </div>			Package	Version	Use/Description	ansible	11.5.0	Core library to install ansible capabilities on the CICD host/container	aiofiles	23.2.1	Library to handle async handle local file operations. Needed for reading files like pipeline checksum for instance	ansible-runner	2.4.1	Library to make interfacing with ansible easier and more efficient inside of python code	fastapi	0.115.12	Framework to build API's with python	pydantic	1.10.22	Library needed for data validation, mostly used in the validation of parameters and bodies in API calls	SQLAlchemy	1.4.54	An ORM library used for sql operations inside of the API	uvicorn	0.34.2	A ASGI web server for python that can handle async frameworks	requests	2.32.3	Library used for fast and quick HTTP operations	seqlog	0.4.1	Library used to log to Seq	pip-system-certs	4.0	A library used to make pip use the default system trust store instead of the bundled CA's	python-multipart	0.0.20	Python streaming multipart parser	python-jose	3.4.0	A Javascript/JSON based encryption
Package	Version	Use/Description																																							
ansible	11.5.0	Core library to install ansible capabilities on the CICD host/container																																							
aiofiles	23.2.1	Library to handle async handle local file operations. Needed for reading files like pipeline checksum for instance																																							
ansible-runner	2.4.1	Library to make interfacing with ansible easier and more efficient inside of python code																																							
fastapi	0.115.12	Framework to build API's with python																																							
pydantic	1.10.22	Library needed for data validation, mostly used in the validation of parameters and bodies in API calls																																							
SQLAlchemy	1.4.54	An ORM library used for sql operations inside of the API																																							
uvicorn	0.34.2	A ASGI web server for python that can handle async frameworks																																							
requests	2.32.3	Library used for fast and quick HTTP operations																																							
seqlog	0.4.1	Library used to log to Seq																																							
pip-system-certs	4.0	A library used to make pip use the default system trust store instead of the bundled CA's																																							
python-multipart	0.0.20	Python streaming multipart parser																																							
python-jose	3.4.0	A Javascript/JSON based encryption																																							

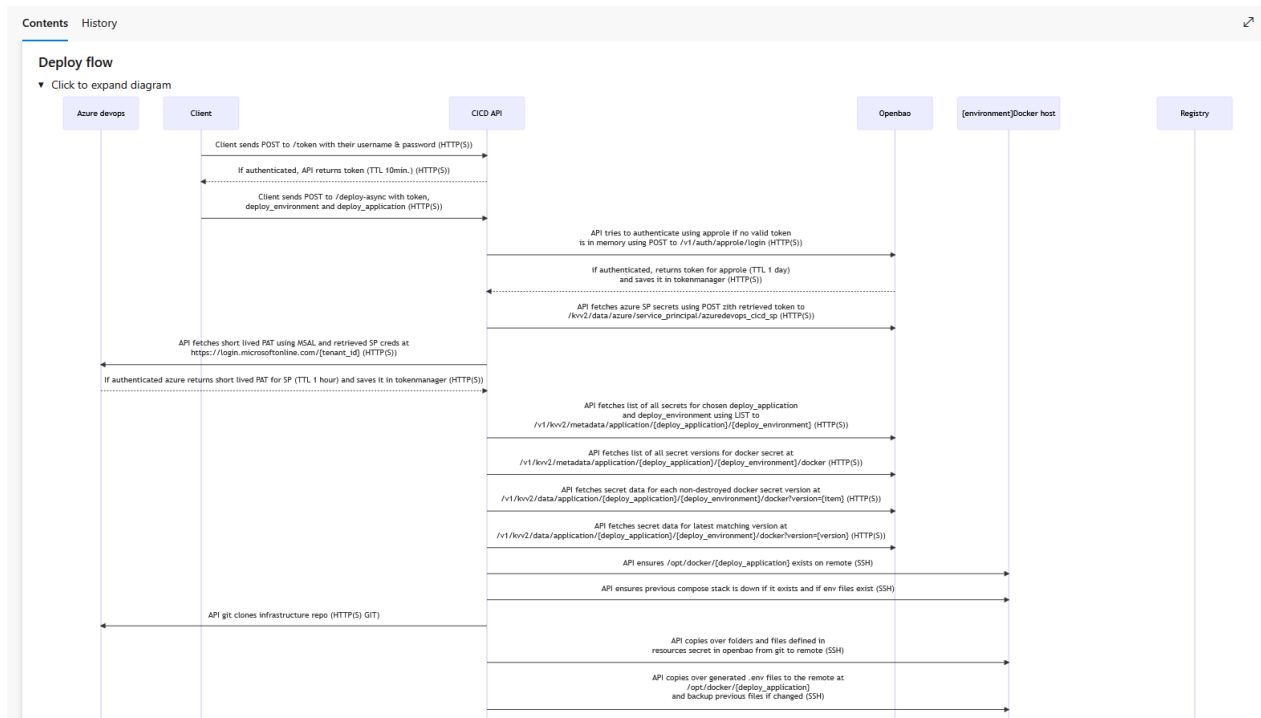
Figuur 3-59 API readme python packages

<div>Contents</div> <div>History</div>			
<div>Environment Variables</div>			
<div>Docker</div>			
This compose uses env variables to fill in the compose.yml file to make it as dynamic as possible. The variables are as defined in .env.example			
Variable Name	Description	Required	Default
'API_CONTAINER_NAME'	Name of the API container	✓	
'API_IMAGE'	Image of the API container	✓	
'API_TAG'	Image tag of the API container	✓	
'API_NETWORK_1'	Network name of API_NETWORK_1	✓	
'API_NETWORK_2'	Network name of API_NETWORK_2	✓	
'API_VOLUME_1'	Volume/mount definition/mapping of API_VOLUME_1	✓	
'API_VOLUME_2'	Volume/mount definition/mapping of API_VOLUME_2	✓	
'API_VOLUME_3'	Volume/mount definition/mapping of API_VOLUME_3	✓	
'API_VOLUME_4'	Volume/mount definition/mapping of API_VOLUME_4	✓	
'API_VOLUME_5'	Volume/mount definition/mapping of API_VOLUME_5	✓	
'API_VOLUME_6'	Volume/mount definition/mapping of API_VOLUME_6	✓	
'API_VOLUME_7'	Volume/mount definition/mapping of API_VOLUME_7	✓	
'API_ENV_FILE'	Env file to use for the API container	✓	
'AGENT_CONTAINER_NAME'	Name of the agent container	✓	
'AGENT_IMAGE'	Image of the agent container	✓	
'AGENT_TAG'	Image tag of the agent container	✓	
'AGENT_NETWORK_1'	Network name of AGENT_NETWORK_1	✓	

Figuur 3-60 API readme env variabelen



Figuur 3-61 API readme build flow



Figuur 3-62 API readme deploy flow

4. Besluit

Na mijn stage bij Wilms NV kijk ik positief terug op de gehele stageperiode. Ondanks onzekerheden en het vaak buiten mijn comfortzone treden, ben ik uiteindelijk zeer tevreden met mijn stageplek. Het team was ideaal en het was zeer leerzaam om te zien hoe IT eruitziet in een productiebedrijf.

Mijn project en de vrijheid die Wilms mij hierbij heeft gegeven, hebben mij enorm geholpen om out-of-the-box te denken. Achteraf gezien was het modulair opzetten van het project een ideale keuze, omdat het project voortdurend in beweging was. Dit maakte het ook altijd interessant, want een probleem dat er de ene dag niet was, deed uiteindelijk toch zijn intrede.

Terugkijkend ben ik verrast door de hoeveelheid werk die ik heb kunnen leveren. De eerste maand leek het alsof er niet veel zou worden afgerond tegen het einde van de stage, maar daarna is het werk al snel in een hogere versnelling gekomen.

Ook heb ik mij kunnen verdiepen in programmeren, wat normaal niet echt de focus is binnen Cloud & Cybersecurity. De mogelijkheid om dit te combineren met kennis die ik uiteindelijk nodig had, zoals Ansible, Docker, etc., gaf mij een goed gevoel en extra motivatie tijdens de stage.

Ik heb veel geleerd van het developmentteam bij Wilms. Zo heb ik meer event-driven leren programmeren om zo veel mogelijk resources te besparen. Ook kwam ik in aanraking met tools waarvan ik nog nooit had gehoord, waardoor mijn blik op dat vlak een stuk breder werd.

Een van de belangrijkste punten die ik meeneem nu mijn stage is afgerond, is het feit dat ik weet dat ik en mijn project daadwerkelijk iets hebben betekend voor Wilms. Ik heb mogen zien hoe het in productie werd geplaatst, ik heb live bugs gefixt en ik heb deel mogen uitmaken van het team. Tot slot heb ik aan het einde ook meegeholpen met de frontend die voor mijn project nu in opbouw is.

Over het algemeen kijk ik heel positief terug op mijn stage bij Wilms. Er was een goede sfeer, het team was geweldig en ik voelde mij er goed.

Ik heb veel geleerd voor de toekomst en hoop dat ik weer in zo'n omgeving terecht kan komen.

LITERATUURLIJST

Ansible Runner — *Ansible Runner Documentation*. (z.d.). Geraadpleegd van <https://ansible.readthedocs.io/projects/runner/en/latest/>

FastAPI. (z.d.). Geraadpleegd van <https://fastapi.tiangolo.com/>

Introduction to Puppet. (z.d.). Geraadpleegd van https://www.puppet.com/docs/puppet/6/puppet_overview.html

Poberezhnyk, A. (2024, 29 juli). *10 Reasons Why CI/CD is Important for DevOps*. Geraadpleegd van <https://cloudfresh.com/en/blog/10-reasons-why-ci-cd-is-important-for-devops/>