

Index

Wordpress i en skalbar AWS-driftsmiljö.....	2
1. Introduktion	2
1.1 Hela materialet finns på ett GitHub-repository	3
1.2 Förkunskaper och installationer.....	3
2. Designa en driftsmiljö för Wordpress	3
2.1 Nätverksstrukturen med VPC, subnät, CIDR, AZ och region	3
2.1.1 CIDR och subnät	3
2.1.2 Säkerhet / SG (Security Groups).....	4
2.1.3 Region och Availability Zones (AZ)	4
2.1.4 En vansinnigt ful vy	4
2.2 Virtuella servrar på AWS (EC2 + ASG + Bastion).....	5
2.2.1 Utrymme till modernisering – Lambdas (Functions)	5
2.3 De olika komponenternas uppgift och syfte	6
2.4 Åtgärder för skalbarhet och säkerhet.....	6
2.5 Vilka molntjänster som utnyttjats	6
3. Avgränsning, provisioning samt konfigurering	7
3.1 Provisionering av driftsmiljön – CF nested stacks	7
3.1.1 S3-bucket för Template-lagring, fattigmans-iaC.....	7
3.2 Konfigurering av applikationen	8
3.3 Verktyg som används	9
3.4 IaC och automation – nested stacks slår till igen	9
3.4.1 Nested Stacks - Djupare förklaring	9
3.5 Utvecklingspotential	10
3.5.1 Docker och containerisering	10
3.5.2 Infrastructure as Code (IaC)-evolution: Terraform	10
3.5.3 Moderniserad CI/CI-pipeline.....	11
3.5.4 Potentiella utmaningar?.....	11
4. Hur kom vi hit egentligen?	11
4.1 Nätverk: På spåret? Målet får ge svaret på resan	11
4.1.1 Ny hink för att ösa upp root-paketet med vår nästlade stack.....	12

4.1.2 Validering av templates – lägg till i verktygslådan!	12
4.1.3 Dags att skapa nätverksstacken och verifiera:	12
4.2 ALB: Andra stoppet på resan mot världsherravälde	13
4.2.2 ALB-SG.....	13
4.2.3 ALB-Target Group	13
4.2.4 Allting hälsosamt än så länge, låt oss inkludera själva load balancern	13
4.2.5 Lyssna noga nu..	14
4.3 App Security Group (ASG) håller oss varma om natten	14
4.3.1 ASG-SG kan vara bra att ha.. men varför är de i varenda fill!12!2	15
4.4 Dags att flexa våra muskler, eller i alla fall bli elastiska.....	15
4.4.1 EFS-SG är en bra start.....	15
4.4.2 Dags att tänja på vårt elastiska systems gränser.....	16
4.5 Dags att börja blogga. Eller i alla fall skapa förutsättningarna	17
4.5.1 InstanceType och Amazon Machine Image (AMI).....	17
4.5.2 Uppskjutningsmallen är ingen vidare översättning för.....	17
4.5.3 Gör om och gör rätt, dags att skapa ett filsystem	18
4.5.4 Dags för en super-seriös validering ultra mega deluxe 3000	18
4.5.5 Små justeringar gör stor skillnad.....	19
4.5.6 Tightar till avgränsningen och förbereder för Maria DB	19
4.5.7 Skarpt läge: nu blir vi bloggiga	21
4.5.8 En databas, en databas, mitt Wordpress-rike för en databas.!.....	25
5. La Grande Finale.....	27
5.1 Uppdatering: Status på applikationen	27
5.2 Bu och bää, styrkor svagheter, vad har vi lärt oss	27
Referenser / noteringar.....	28

Wordpress i en skalbar AWS-driftsmiljö

1. Introduktion

Målet med själva uppgiften är att kunna bygga en driftsmiljö i AWS som kan hosta Wordpress: miljön skall vara skalbar, fungerande och ha en grundläggande robusthet med viss säkerhet.

Avgränsningen för uppgiften nämns specifikt, därför har jag för avsikt att presentera ett "hur bygger vi vidare på detta?"-avsnitt som kan fungera på tre sätt:

- dels visar den på grundläggande kunskap
- dels visar det att skalbarhetsmålet nås och finns med redan initialt
- dels så vill jag visa på en flexibilitet: jag bygger gärna själv mer avancerade lösningar och testar teknik, men:

Ibland är det absolut nödvändigt att följa en uppgiftsbeskrivning, och där kan dokumentationen fungera som ett bra område att fånga upp potentiella förslag till förbättringar. Jag vill gärna simulera det i denna uppgift.

1.1 Hela materialet finns på ett GitHub-repository

Made you look! https://github.com/mymh13/scaleable_aws_turn-in-1

1.2 Förkunskaper och installationer

Om någon vill replikera detta projekt så förutsätts att du har:

- Grundläggande kunskaper i AWS
- Ett AWS-konto med en IAM-användare och nödvändiga rättigheter (S3 exempelvis)
- AWS-CLI installerat och förberett att köras

2. Designa en driftsmiljö för Wordpress

Uppgiftsbeskrivning:

- Basera lösningen på virtuella servrar på AWS
- Beskriv vad de olika komponenterna i din design har för uppgift och syfte
- Beskriv vilka åtgärder du vidtagit för skalbarhet och säkerhet
- Redogör för vilka molntjänster du utnyttjat

2.1 Nätverksstrukturen med VPC, subnät, CIDR, AZ och region

För att en applikation (exempelvis WordPress) skall kunna drivas robust och skalbart krävs att nätverket byggs på en stabil grund. Projektets VPC (Virtual Private Cloud) fungerar som den virtuella motorvägens fundament – en egen avgränsad nätverksyta i AWS där vi kan definiera regler: IP-spann (adressen), routing (transportvägen) och säkerhet (tillgängligheten).

2.1.1 CIDR och subnät

Vi delar upp VPCens IP-spann (10.20.0.0/16) i mindre block (CIDR). Det kan vara olika tjänster (som i vårt nätverk) men även exempelvis olika siter hos ett produktions-/industribolag med ett intranät. På så sätt skapas logiska områden med olika syfte:

- **Publika subnät (10.20.0.0/24, 10.20.1.0/24):** Här placerar vi resurser som behöver kunna nås från Internet, exempelvis vår ALB och Bastion Host. Dessa subnät har

route mot en IGW (Internet Gateway) och resurser får automatiskt publika IP-adresser via `MapPublicIpOnLaunch: true`

- **Privata app-subnät (10.20.10.0/24, 10.20.11.0/24)**: Här kör vi våra EC2-instanser med WordPress. De får ingen publik IP och nås endast via ALB. För att de ändå ska kunna hämta uppdateringar når de ut mot Internet via en NAT Gateway i det publika subnätet
- **Privata db-subnät (10.20.20.0/24, 10.20.21.0/24)**: Här kör vi databasen (MariaDB). Dessa subnät har NAT-route för grundläggande systemuppdateringar men ingen direkt internetexponering. Åtkomst sker endast från applikationslagret via port 3306

2.1.2 Säkerhet / SG (Security Groups)

Security Groups i AWS fungerar som ett filter som bara tillåter godkänd trafik. Exempel:

- ALB-SG: tar emot HTTP (port 80) från Internet (0.0.0.0/0)
- App (WP)-SG: tar emot HTTP enbart från ALB-SG via `SourceSecurityGroupId`
- DB-SG: tar emot DB (port 3306) enbart från App (WP)-subnätens CIDR-block
- EFS-SG: tar emot NFS (port 2049) enbart från App-subnätens CIDR-block
- Bastion-SG: tillåter SSH (port 22) enbart från administratörens IP-adress

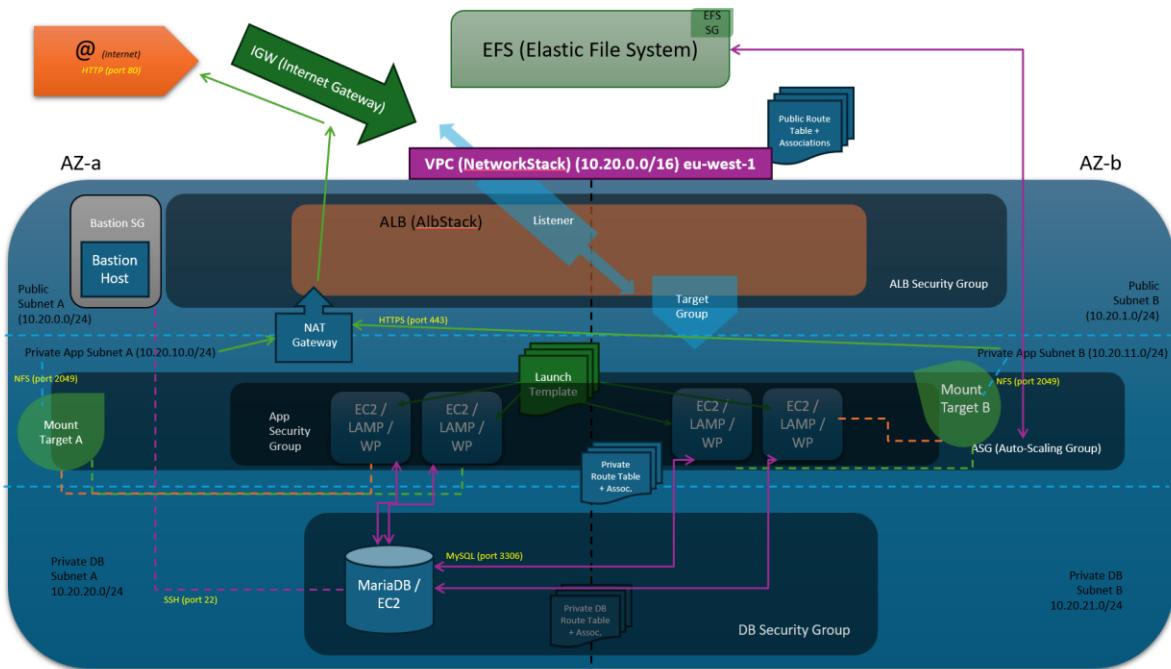
På det här viset byggs nätverket upp i lager, som en lök. Varje lager har sitt syfte och sin åtkomst-begränsning, vilket ger både skydd och skalbarhet. Dessutom använder vi dynamisk SSH-regel där Bastion-SG läggs till som källa för DB-SG via `DbSshFromBastion`-regeln i `root.yaml`.

2.1.3 Region och Availability Zones (AZ)

En AWS-region (exempelvis eu-west-1 som vi använder) är en geografisk plats. Inom varje region finns flera datacenter, så kallade AZ (eu-west-1a, eu-west-1b, etc). Ett subnät i AWS kan bara ligga i en enda AZ. För hög tillgänglighet av vår tjänst (tänk om ett datacenter går ner) så lägger vi därför ut flera subnät över minst två AZ.

2.1.4 En vansinnigt ful vy

Den här är vansinnigt fult, men ger kanske en bild av systemets flöde:



2.2 Virtuella servrar på AWS (EC2 + ASG + Bastion)

Vår arkitektur använder flera typer av EC2-instanser:

- **Auto Scaling Group med Launch Template:** Hanterar WordPress-instanserna elastiskt (1-2 instanser). Launch Template definierar Amazon Linux 2023 med Apache, PHP och nödvändiga moduler. Rolling updates säkerställer zero-downtime deployments.
- **MariaDB på EC2:** En dedikerad databasinstans med automatisk installation och konfiguration via UserData. Inkluderar IAM-roll med SSM-åtkomst för säker hantering.
- **Bastion Host:** Minimal EC2-instans för säker administrativ åtkomst till privata resurser. Fungerar som "hopppunkt" för SSH-tunnel till privata instanser.

2.2.1 Utrymme till modernisering – Lambdas (Functions)

Lambda skulle kunna ersätta EC2 för WordPress genom containerized deployment med Lambda Container Images. Detta skulle ge serverless-fördelar som automatisk skalning, pay-per-request och ingen infrastrukturhantering. Dock innebär WordPress komplexitet utmaningar: persistent storage (EFS krävs fortfarande), cold starts för stora WordPress-installationer och begränsningar i execution time (15 minuter max).

För en moderniserad arkitektur skulle jag separera WordPress i microservices - statisk frontend (S3 + CloudFront), API-funktioner (Lambda), och headless CMS. Docker-container med Lambda Container Images skulle hantera PHP-runtime och WordPress-kod, medan RDS Aurora Serverless skulle ersätta EC2-MariaDB för automatisk skalning av databasen.

2.3 De olika komponenternas uppgift och syfte

- **VPC + Subnät:** Skapar isolerat nätverk med logisk separation av tjänster
- **Internet Gateway:** Ger publika subnät internetåtkomst
- **NAT Gateway:** Tillåter utgående internet från privata subnät
- **Application Load Balancer:** Distribuerar HTTP-trafik över WordPress-instanser med health checks
- **Auto Scaling Group:** Automatisk skalning baserat på belastning och health status
- **EFS:** Delad fillagring för WordPress-filer mellan instanser
- **MariaDB på EC2:** Relationsdatabas för WordPress-innehåll
- **Security Groups:** Nätverkssäkerhet på instansnivå
- **Bastion Host:** Säker administrativ åtkomst

2.4 Åtgärder för skalbarhet och säkerhet

Skalbarhet:

- Auto Scaling Group justerar antalet instanser automatiskt
- Multi-AZ deployment för hög tillgänglighet
- EFS ger delad storage som skalar automatiskt
- Load Balancer distribuerar trafik jämnt

Säkerhet:

- Privata subnät isolerar kritiska komponenter
- Security Groups implementerar "least privilege"-principen
- IMDSv2 aktiverat på alla instanser (HttpTokens: required)
- EFS-kryptering för data at rest
- Bastion Host som enda SSH-ingång
- NAT Gateway för säker utgående trafik

2.5 Vilka molntjänster som utnyttjats

- **AWS EC2:** Virtuella servrar för applikation, databas och bastion
- **VPC:** Virtuellt privat nätverk med subnät
- **Application Load Balancer:** Layer 7 lastbalanserare
- **Auto Scaling Groups:** Automatisk instansskalning
- **Amazon EFS:** Managed filsystem
- **Internet Gateway:** Internetanslutning för publika subnät
- **NAT Gateway:** Utgående internet för privata subnät
- **Security Groups:** Nätverkssäkerhet
- **IAM:** Roller och behörigheter
- **CloudFormation:** Infrastructure as Code
- **S3:** Template-lagring för nested stacks

3. Avgränsning, provisioningering samt konfigurering

Uppgiftsbeskrivning: "Gör en tydlig avgränsning i din design och beskriv sedan steg för steg hur du provisionerar driftsmiljön samt konfigurerar din applikation"

Under punkt 4 finns en löpande dokumentation om hur projektet växte fram, och varför jag valt vilka vägar – vilka problem som uppstått och hur jag löst dem.

3.1 Provisionering av driftsmiljön – CF nested stacks

Provisioneringen sker genom **nested CloudFormation stacks** via root.yaml som fungerar som huvudorkestrator. Root-templetten refererar till child-templates lagrade i S3-bucket enligt mönstret:

[https://s3.\\${AWS::Region}.amazonaws.com/\\${TemplateBucket}/\\${TemplatePrefix}XX-component.yaml](https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}XX-component.yaml)

Deployment-ordningen hanteras via DependsOn-attribut:

1. **NetworkStack** (00-network.yaml) - grundläggande VPC-infrastruktur
2. **AlbStack** (20-alb.yaml) - parallellt med EFS och DB-komponenter
3. **EfsStack** (30-efs.yaml) - filsystem för delad storage
4. **DbStack + DbEc2Stack** (40+45) - databassäkerhet och instans
5. **AsgStack** (50-asg-wordpress.yaml) - WordPress-applikation
6. **BastionStack** (60-bastion-deploy.yaml) - administrativ åtkomst

Outputs från child-stacks används som inputs via !GetAtt, vilket skapar automatiska beroenden mellan stackarna.

3.1.1 S3-bucket för Template-lagring, fattigmans-IaC

Nested stacks kräver att child-templates är tillgängliga via HTTPS-URL, vilket gör S3 till den naturliga lösningen för template-lagring. Vår bucket-struktur ser ut enligt följande:

```
wordpress-iac-<account-id>-<region>/  
|   └── wordpress-iac/  
|       └── templates/  
|           ├── 00-network.yaml  
|           ├── 20-alb.yaml  
|           ├── 30-efs.yaml  
|           ├── 40-db-sg.yaml  
|           ├── 45-db-mariadb-ec2.yaml  
|           ├── 50-asg-wordpress.yaml  
|           ├── 60-bastion-deploy.yaml  
|           └── root.yaml
```

S3-bucket konfiguration:

- **Bucket-namngivning:** Inkluderar account-ID och region för unikhet och säkerhet
- **Versionering:** Aktiverat för rollback-möjligheter av templates

- **Public access:** Blockerat (CloudFormation använder AWS-interna API:er)
- **Encryption:** Server-side-kryptering för template-säkerhet

Deployment-process:

1. **Upload templates:** aws s3 sync ./templates s3://bucket/wordpress-iac/templates/
2. **Deploy stack:** aws cloudformation create-stack --template-url https://s3.region.amazonaws.com/bucket/...root.yaml
3. **URL-resolution:** root.yaml konstruerar child-template-URLer dynamiskt via !Sub-funktioner

Fördelar med denna struktur:

- **Template-versionering:** S3-versionering ger historik och rollback-möjlighet
- **Centraliserad lagring:** Alla templates på en plats för teamåtkomst
- **Automatisk URL-konstruktion:** Dynamiska template-URLs via CloudFormation-variabler
- **Säker distribution:** Templates är tillgängliga för CloudFormation men inte publikt

Template URL-mönster:

```
TemplateURL: !Sub
'https://${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}00
-network.yaml'
```

Detta möjliggör deployment från vilken region som helst, genom att bara ändra bucket-namn och prefix-parametrar, samtidigt som template-URL:erna konstrueras automatiskt för korrekt region.

3.2 Konfigurering av applikationen

WordPress konfigureras automatiskt via **UserData-script** i Launch Template:

- Installation av Apache, PHP och MySQL-moduler
- Växling från mpm_event till mpm_prefork för mod_php-kompatibilitet
- Aktivering av .htaccess-stöd för WordPress permalinks
- Test-endpoints för hälsokontroller (/php-health.php, /dbtest2.php)
- Automatisk databaskoppling med parametriserade credentials

MariaDB konfigureras via UserData:

- Installation och start av MariaDB-service
- Nätverkskonfiguration för extern åtkomst (bind-address=0.0.0.0)
- Automatisk skapande av databas och applikationsanvändare
- SQL-kommandon har körts säkert via unix_socket (oklart om det skulle vara en permanent lösning, dock, men det tål att nämnas 😊)

3.3 Verktyg som används

- **AWS CLI:** För stack-deployment och S3-uppladdning
- **CloudFormation:** Infrastructure as Code-orchestrering
- **S3:** Template-lagring och versionshantering
- **Bash-script:** Automatiserad deployment-process
- **YAML:** Template-format för läsbarhet
- **Git:** Versionshantering av IaC-kod

3.4 IaC och automation – nested stacks slår till igen

Denna punkt går lite i punkt 1, provisioneringen, men:

Nested Stacks-arkitekturen möjliggör modulär infrastruktur där varje komponent kan utvecklas och deployeras oberoende. Root.yaml fungerar som "styrdokument" med hänvisningar till alla sub-stacks, vilket ger följande fördelar:

Styrkor:

- Modulär utveckling och underhåll
- Återanvändbarhet av komponenter
- Tydlig separation of concerns
- Centraliserad parameterhantering
- Automatiska beroenden via outputs/inputs

Begränsningar:

- Komplexare felsökning vid failures
- S3-beroende för template-lagring
- Längre deployment-tider för hela stacken
- Svårare att göra partiella updates

Deployment-automation sker via S3-bucket där templates packas och laddas upp, följt av ett enda aws cloudformation create-stack-kommando som deployeras hela infrastrukturen.

Samma enkelhet gäller för teardown via delete-stack.

3.4.1 Nested Stacks - Djupare förklaring

Nested stacks löser CloudFormations begränsning på 500 resurser per template genom att dela upp arkitekturen i logiska komponenter. Varje child-stack har sitt eget lifecycle men koordineras av parent-stacken. När root.yaml deployeras skapas en hierarki där outputs från en stack automatiskt blir tillgängliga som inputs till beroende stacks via !GetAtt-funktionen.

Detta möjliggör verkligt Infrastructure as Code där hela produktionsmiljön kan skapas eller förstöras med ett enda kommando, samtidigt som varje komponent kan underhållas och versionshanteras separat. Template-lagringen i S3 ger versionshistorik och möjlighet till rollback av enskilda komponenter.

3.5 Utvecklingspotential

Några punkter där detta projekt hade kunnat ta ytterligare ett steg:

3.5.1 Docker och containerisering

Nuvarande EC2-baserade arkitektur skulle kunna moderniseras med Docker-containers för förbättrad portabilitet, resurseffektivitet, lättare att underhålla. WordPress och MariaDB skulle paketeras som separata containers med:

- **Immutable infrastructure:** Containers med versionstaggade images istället för mutable EC2-instanser
- **Snabbare deployment:** Container-starts på sekunder vs minuter för EC2 UserData-skript
- **Konsistent miljöer:** Identiska runtime-miljöer från utveckling till produktion
- **Resource isolation:** Bättre CPU/minnes-kontroll och säkerhetsavskiljning

Docker Swarm som initial orkestrator skulle ge enkel kluster-hantering med inbyggd load balancing och service discovery, men har begränsningar jämfört med Kubernetes för enterprise-skalning. Det hade fungerat bra som ” nästa steg ” men inte slutmålet.

3.5.2 Infrastructure as Code (IaC)-evolution: Terraform

Terraform skulle ersätta CloudFormation, med en del förtjänster:

Fördelar:

- **Multi-cloud support:** Möjlighet att hantera AWS, Azure, GCP med samma verktyg
- **HCL-syntax:** Mer läsbar än YAML för komplexa konfigurationer
- **State management:** Explicit state-hantering med remote backends (S3 + DynamoDB)
- **Plan-funktion:** Förhandsvisning av infrastrukturändringar innan applicering
- **Moduler:** Återanvändbara komponenter över projekt och team
- **Community:** Större ekosystem av providers och moduler

Nackdelar:

- **Ingen native AWS-integration:** CloudFormation har förstahandsstöd för nya AWS-tjänster (men vi läser in oss ytterligare i det ekosystemet)
- **State-komplexitet:** Risk för state drift och låsning av Terraform state (Terraform hanterar state själv, utanför AWS, medan CF låter AWS göra det åt oss)
- **Drift detection:** CloudFormation har bättre inbyggd drift-hantering

3.5.2.1 Konfigurationshantering via Ansible

Ansible skulle hantera applikationskonfiguration och deployment:

- **Agentless:** SSH-baserat, ingen agent på målsystem
- **Idempotent:** Säker att köra samma playbook flera gånger
- **Tydlig syntax:** Enkel och läsbar konfiguration
- **Rullande deployments:** Koordinerade uppdateringar över instanser

- **Secrets management:** Integration med AWS Secrets Manager/Parameter Store

3.5.2.2 En kombinerad arkitektur skulle kunna se ut så här

```
Terraform → Provision infrastructure (VPC, subnets, load balancers)
```

```
↓
```

```
Docker → Package applications (WordPress, MariaDB containers)
```

```
↓
```

```
Ansible → Deploy containers, configure services, manage secrets
```

3.5.3 Moderniserad CI/CI-pipeline

Med dessa verktyg skulle deployment-processen bli:

1. **Terraform plan/apply:** Infrastruktur-ändringar
2. **Docker build/push:** Container-images till ECR
3. **Ansible playbooks:** Rullande deployment av nya container-versioner
4. **Automatiserad testing:** Infrastruktur och applikations-testning

3.5.4 Potentiella utmaningar?

- **Ökad komplexitet:** Fler verktyg kräver bredare kompetens
- **State management:** Terraform state och Docker registry kräver robust backup-strategi
- **Tool chain coordination:** Integration mellan Terraform, Docker och Ansible behöver orkestrering
- **Inlärningskurva:** Team behöver utbildning i flera nya teknologier

Denna utvecklingsväg skulle ge betydligt mer flexibel och portabel infrastruktur, men kräver investering i kompetens och verktygsintegration.

4. Hur kom vi hit egentligen?

Det här är en löpande dokumentering som visar hur projektet växte fram.

Det första jag gör är att identifiera vad vi skall bygga:

Vi skall hosta wordpress på AWS. Tjänsterna vi skall använda blir ALB, ASG, EFS, och målet är även att sätta upp ett privat VPC vilket bör tas med redan i absolut första stegen: nätverkskonfigurationen bör nog bli absolut först. Vi vill använda LAMP för WP och EC2or för MariaDB och tjänster inom systemet (ASG, provisioning server för WP).

4.1 Nätverk: På spåret? Målet får ge svaret på resan

När ramverket ovan är klart, så ritar jag upp en grovkiss på hur repository layout kan vara:

```
/infra
```

```
  /templates
```

```
    root.yaml # master stack – orchestrates Nested Stacks
```

```
00-network.yaml # VPC, subnets, IGW, NATGW, routes
20-alb.yaml # ALB, Target Group, Listener, SG
30-efs.yaml # EFS, Mount Targets, SG
40-rds-mariadb.yaml # DB Subnet Group, SG, MariaDB instance
50-asg-wordpress.yaml # LT + ASG for LAMP/WordPress, SG, scaling
policy
60-bastion-deploy.yaml # Deploy/Bastion EC2 in public subnet (SSH), SG
/parameters
dev.json # example parameter set for root.yaml
```

För att få en automatisering utan att köra shell-script så tänker jag att de numrerade templatesen laddas upp till en S3-bucket. [Mer om det senare i avsnitt 3.1.1.](#)

4.1.1 Ny hink för att ösa upp root-paketet med vår nästlade stack

Kör en lokal pakacking (aws cloudformation package) för att skapa en ensam rootstack, sedan laddas den upp till en S3-bucket. Jag har sedan tidigare gett min IAM-user S3-rättigheter (Get/Put/List/Delete).

Skapar en ny bucket: tydligt namn (wordpress-iac-<account-id>-eu-west-1), blockerar public access, kör default encryption och enablar versionering (det hjälper om vi rollbackar mallen: jag planerar att ta ner/upp hela projektet med yaml-templatesen så det är najs att ha).

Sedan laddar jag upp mallarna till S3:

```
aws s3 cp infra/templates/00-network.yaml s3://wordpress-iac-<account-id>-eu-
west-1/wordpress-iac/templates/00-network.yaml
aws s3 cp infra/templates/root.yaml s3://wordpress-iac-<account-id>-eu-west-
1/wordpress-iac/templates/root.yaml
```

Jag vill inte hårdkoda mitt account-id i root.yaml, så jag parametriserar det och infogar via Parameters: TemplateBucket och sedan definierar jag det i dev.json TemplateBucket: "account-id".

4.1.2 Validering av templates – lägg till i verktygslådan!

Efter varje steg jag jobbat med mina templates så validerar jag alltid lokalt, superbra verktyg:

```
aws cloudformation validate-template --template-body
file://infra/templates/root.yaml
```

4.1.3 Dags att skapa nätverksstacken och verifiera:

I det här läget kör jag ett aws cloudformation create-stack-kommando, för att skapa stacken:

```
aws cloudformation create-stack \
--stack-name wp-cf-root \
```

```
--template-url https://s3.eu-west-1.amazonaws.com/wordpress-iac-<account-id>-eu-west-1/wordpress-iac/templates/root.yaml \
--parameters file://infra/parameters/dev.params.json
```

När jag kör --parameters file://infra/parameters/dev.params.json så skickas parametrarna direkt från min lokala fil till CloudFormation-API:t. Filen ligger inte i Git-repot (den är utesluten i .gitignore, jag skickar bara med en example.params.json för att illustrera hur den kan se ut) och laddas inte upp till S3 som våra mallar gör. Det innebär att exempelvis lösenord eller andra känsliga värden aldrig exponeras publikt eller lagras i versionshanteringen, utan endast överförs i en krypterad begäran (HTTPS) till AWS när stacken skapas.

4.1.4.1 Första problemet uppstod! Jag var smart men inte tillräckligt smart..

Jag parametriserar känsliga värden, definierar dem i root.yaml som exempelvis "TemplateBucket", sedan lägger jag det hårdkodade värdet i en .gitignorad dev.params.json.

I vanliga fall brukar det gå bra att bara definiera ett värde som en json-sträng, men CloudFormation kräver arrayer med ParameterKey / ParameterValue och inte ett objekt – så jag fick göra om json-filen till att reflektera detta format. Då funkade det!

4.2 ALB: Andra stoppet på resan mot värlsherravälde

Nu skapar jag ALBn, stegvis, och testar. Så här:

4.2.2 ALB-SG

Lägger till Parameters och Resources för Security Groupen, framförallt så sätter vi SG-Ingress till "tillåt HTTP från Internet" = IpProtocol: tcp, Cidrlp 0.0.0.0/0 med Port 80. SG-Egress är utåt, där tillåter vi allting så vi sätter IpProtocol: -1 med samma Cidrlp som ovan.

4.2.2.1 Verifierar allting med att köra följande:

```
aws cloudformation validate-template --template-body
file://infra/templates/20-alb.yaml
```

4.2.3 ALB-Target Group

ALBn behöver en Target Group att skicka trafiken till. Vi definierar vilket protokoll vi använder och vilken port, vilket här är port 80 och således protokollet HTTP. Vi vill även definiera en HealthCheckPath, den visar vilka HTTP-statuskoder som är "healthy" eller om de är unhealthy. Rangen sätts som 200-399, 200-rangen är "ok" och 300-rangen är "redirects". 400 är vanliga felkoder och 500 serverfel, så de är unhealthy och skall inte med här.

Verifierar igen med [samma verifieringskod](#) som i 4.2.2.1 (Verifierar bara 20-alb.yaml-filen).

4.2.4 Allting hälsosamt än så länge, låt oss inkludera själva load balancern

Det enda vi egentligen tillfogar nu är detta:

- Resources – ALB-type och properties
- Outputs – Sätter värden för ALB-DNS och ALB-ARN.

ARN är Amazon Resource Name, det skapar en unik identifierare för varje resurs i AWS. När vi lägger till den i outputs så kan andra stacks i vår nested setup referera till den. Vi kommer behöva det när vi skapar ASGn för den behöver veta vilken Target Group-ARN som instanserna skall registreras i. Vi kan säga att DNS är den "externa anslutningen" till vår ALB medan ARN är den "interna anslutning" som IaC/automation använder för kopplingar.

Validera! Alltid [validera](#) innan vi tar nästa steg.. 😊

4.2.5 Lyssna noga nu..

I förra steget satte vi en identifierare (ARN) för själva ALB-resursen. Nu behöver vi även definiera en lyssnare (Listener) som tar emot inkommende HTTP-trafik på port 80 och skickar den vidare till vår Target Group (TG).

Det gör vi genom att lägga till en resurs av typen HttpListener i 20-alb.yaml, där vi anger dess egenskaper (properties). Därefter exponerar vi ARN för lyssnare i Outputs, så att andra stackar kan referera till den.

Slutligen behöver vi också lägga till detta i root.yaml under outputs, så att lyssnaren blir tillgänglig för resten av vår infrastruktur. [Validera!](#) Glöm inte! Sedan laddar vi upp 20-alb.yaml till AWS, validerar på AWS och uppdaterar sedan stacken:

```
aws s3 cp infra/templates/20-alb.yaml \
s3://wordpress-iac-<account-id>-eu-west-1/wordpress-iac/templates/20-
alb.yaml

aws cloudformation validate-template \
--template-url https://s3.eu-west-1.amazonaws.com/wordpress-iac-<account-
id>-eu-west-1/wordpress-iac/templates/root.yaml

aws cloudformation update-stack \
--region eu-west-1 \
--stack-name wp-cf-root \
--template-url https://s3.eu-west-1.amazonaws.com/wordpress-iac-<account-
id>-eu-west-1/wordpress-iac/templates/root.yaml \
--parameters file://infra/parameters/dev.params.json
```

4.3 App Security Group (ASG) håller oss varma om natten

Först blir det lite pillande i filer från ALB-avsnittet, men vi behöver få med oss ett ALB SG-ID i outputsen så vi kan dela till andra stashar. Det är inget stort ingrepp: lägg till i Outputs i 20-alb.yaml: AlbSecurityGroupId samt lägg till i Outputen på root.yaml. Vi får inte glömma att uppdater dessa filer sedan på AWS, men låter dem vara tills allt är klart.

4.3.1 ASG-SG kan vara bra att ha.. men varför är de i varenda fil!12!2

Nu så kan vi skapa grunden till ASGn. Vi lägger inte till några resurser än, grunden för dokumentet får helt enkelt bara vara en ASG-SG så vi kan validera det. Det viktigaste här är att se till att VpcId skickas in från NetworkStack Outputs och AlbSecurityGroupId från AlbStack Outputs.

Vi hade kunnat lägga alla SG i samma dokument och slippa ha en massa delande av resurser i outputs, så varför denna arkitektur? Jo:

- Läsbart: Varje stack ”äger” sin egen SG
- Kopplingar: App-SG refererar ALB-SG via param (tydligt beroende)
- Flexibilitet: Om vi byter ALB-stack i framtiden, pekar vi bara om parametern

4.3.1.1 Validering, uppladdning, validering, uppdatering:

```
aws cloudformation validate-template --template-body  
file://infra/templates/20-alb.yaml  
aws cloudformation validate-template --template-body  
file://infra/templates/50-asg-wordpress.yaml  
aws cloudformation validate-template --template-body  
file://infra/templates/root.yaml  
# ladda upp till S3  
aws s3 cp infra/templates/20-alb.yaml s3://wordpress-iac-<account-id>-eu-west-  
1/wordpress-iac/templates/20-alb.yaml  
aws s3 cp infra/templates/50-asg-wordpress.yaml s3://wordpress-iac-<account-  
id>-eu-west-1/wordpress-iac/templates/50-asg-wordpress.yaml  
aws s3 cp infra/templates/root.yaml s3://wordpress-iac-<account-id>-eu-west-  
1/wordpress-iac/templates/root.yaml  
# Validera root från S3  
aws cloudformation validate-template \  
--template-url https://s3.eu-west-1.amazonaws.com/wordpress-iac-<account-  
id>-eu-west-1/wordpress-iac/templates/root.yaml  
# Uppdatera stacken i eu-west-1  
aws cloudformation update-stack \  
--region eu-west-1 \  
--stack-name wp-cf-root \  
--template-url https://s3.eu-west-1.amazonaws.com/wordpress-iac-<account-  
id>-eu-west-1/wordpress-iac/templates/root.yaml \  
--parameters file://infra/parameters/dev.params.json
```

4.4 Dags att flexa våra muskler, eller i alla fall bli elastiska..

Elastic File System (EFS) är nästa steg. Med det så kan vi skala systemet efter belastning.

4.4.1 EFS-SG är en bra start

Det är naturligt att bygga från grunden, oavsett vi bygger hus eller IT-system med infrastruktur. Näst efter nätverket så behöver vi titta på access, roller och regler, det är här

SG kommer in. Varje separat enhet får sina egna regelverk. Vi kan säga att SG är den lokala polisen – i detta fall kommer SG bara tillåta NFS port 2049 från App-(WP)-SG. Validera:

```
aws cloudformation validate-template --template-body  
file://infra/templates/30-efs.yaml
```

EFS saknas i root.yaml, under resurser ligger där bara NetworkStack, AlbStack, AsgStack för stunden. Vi inkluderar även EfsStack, och ser till att Outputs har med EfsSId.

Nu repeterar vi [steg 4.3.1](#) men med små justeringar: ladda upp 30-efs.yaml + root.yaml. Validera root.yaml mot S3. Uppdatera stack. Ni börjar få kläm på detta nu eller hur?

4.4.2 Dags att tänja på vårt elastiska systems gränser

I förra punkten la jag bara grunden. Under Resources har vi bara EfsSecurityGroup. Nu är det dags att implementera tjänster vi behöver: FileSystem, MountTargetA och MountTargetB. Som vanligt så vill vi sätta Outputs så vi kan kommunicera inom den nästlade stacken.

En sak som sticker ut här, som jag gärna vill dela från Outputs är följande:

```
MountTargetIds:  
  Value: !Join [",", [!Ref MountTargetA, !Ref MountTargetB]]  
  Description: Mount Target resource IDs (en per app-subnät)
```

Värde-raden och !Join är en ”hjälpfunktion”: CloudFormation kan inte själv returnera listor direkt, så vi joinar ihop vår lista till en sträng, separerad med kommatecknen. På så vis kan andra stacks (eller vi, via CLI/console) läsa outputen som en CSV-sträng. Vi kan även splitta den vid behov.

4.4.2.1 Överkurs deluxe, men har jag sagt A får jag säga B

Någon som läser detta kanske undrar hur vi hämtar ut en enskild data ur den kommaseparerade strängen? Hur bryter vi ut datan? Det blir en hel del kod att visa det, så lite kort – vi skapar en parameter CommaExampleList där vi sedan plockar index (0, 1, etc). Det går även att göra det med ett !Split-kommando i Properties.

Ett annat sätt att hantera MountTargetIds hade kunnat vara att specificera dem som separata värden direkt i Outputs. När kan vi vilja använda vilken?

- CommaExampleList är bäst om vi vill hantera det som en lista i child-yamlen
- !Split är smidigt om vi redan har en sträng och inte vill konvertera parametertyper
- Separata outputs är bra om vi **ofta** vill ha enskilda värden

Eftersom EFS i vårt fall skapar ett Mount Target per AZ och EC2-instansen monterar EFS via DNS-namnet, då behöver Wordpress/ASG bara FileSystemId + EFS-SG, inte enskilda MT-ID. Hade vi velat felsöka eller manuellt rensa resurser, då kanske vi hade velat titta på alternativ.

4.5 Dags att börja blogga. Eller i alla fall skapa förutsättningarna

Initialt tänker jag att vi bara tittar på fundamentet och lägger grunden för Launch Template, för då kan vi kolla att ALB -> ASG -> EFS-flödet fungerar som det skall, innan vi rör Wordpress.

4.5.1 InstanceType och Amazon Machine Image (AMI)

Hade en mall med SG för 50-asg-wordpress.yaml redan, la till InstanceType och Amild i parametrarna för att ha några defaultvärden. AMI pekar på den senaste Amazon Linux 2023-Imagen via SSM så vi slipper hårdkoda ID:t. Validerar 50-asg lokalt!

4.5.2 Uppskjutningsmallen är ingen vidare översättning för..

Launch Template. Eller hur, visst låter det illa? Nu börjar det bli spännande (och kanske luktar bränt – vi får se). Vi behöver en Launch Template och tar in FileSystemId som ny parameter (för att montera på instanserna).

root/Resources/AsgStack lägger till FileSystemId-outputen (tydligt mönster nu: inkrementella tillägg, validera, ladda upp, nästa steg rinse-repeat). Sedan validerar vi root.yaml och 50-asg.yaml lokalt, därefter laddar vi upp mallarna till S3, och sist uppdaterar vi stacken. Se [detta avsnitt](#) för mall (bara skifta stacknames).

4.5.2.1 Problem – inkrementell loop – och lösning

Tack vare de små inkrementella stegen och valideringen så fångade jag ett problem här. Det blir en cirkulär loop mellan AsgStack och EfsStacken – AsgStack behöver File-systemId (från EfsStack) men EfsStack behöver AppSecurityGroupId (från AsgStack).

I det här läget kunde jag ha gjort ett mellansteg med Cidrs, men det finns två problem där:

- Dels lägger det till abstraktion / komplexitet som inte behövs
- Dels så skall vi ändå göra klart App-ASG i framtiden

För att slippa skapa någonting som gör systemet krångligare och som sedan dessutom skall tas bort, så gör jag ett litet sidosteg från principen att bara göra små inkrementella förändringar.

4.5.2.2 ..cirklén inte längre sluten

Launch Template får starta en minimal webbserver från lokaldisk. AsgStacken skapas då utan att bero på EfsStack. När ApSG finns så kan EfsStacken ta AppSecurityGroupId som parameter och tillåta endast den på port 2049. Ingen cirkel. När EFS är på plats så kan vi göra en liten update av Launch Template för att montera EFS. Färre och naturliga ingrepp.

root.yaml uppdaterar AsgStack (ta bort FileSystem, lägg till PrivateAppSubnetIds + TargetGroupArn). I 50-asg-wordpress.yaml får vi lägga till AutoScalingGroup samt uppdatera LaunchTemplate. Validerar 50-asg-wordpress och root.yaml. Laddar upp båda till S3. Kör update-stack på aws cloudformation.

4.5.3 Gör om och gör rätt, dags att skapa ett filsystem

Dags att göra det jag gjorde för snabbt senast. Börjar med att uppdatera User Data i LT. I nästa steg lägger jag till en FileSystemId i root.yaml/AsgStack/Parameters. Har även en UpdatePolicy:false på ASGn – vill inte riskera att hela gruppen kan råka ersättas pang bom. Styr ASG-instansernas update-stack genom att följa vissa regler, istället för att bara nukea EC2orna / instanserna.

Validerar 50-asg-wordpress.yaml och root.yaml. Laddar upp till S3. Update-stack.

4.5.4 Dags för en super-seriös validering ultra mega deluxe 3000

Vi har byggt ganska mycket nu, och valideringarna har ju gjorts lokalt med validate-template – och mot S3 med validering mot URLen. Men för att vara säker på att vi är i fas så känner jag att det är dags att testa mer extensivt. Jag går in i AWS / CloudFormation / Stacks på konsollen och kikar på hur stacken ser ut. Det går att se visuellt att en stack ligger som NESTED och vilken stack som är root, senaste uppdateringen, när filen skapats etc.

Allting ser OK ut, det är ju inga kompletta filer för helheten ännu, men det är viktigt att göra små avstamp här och där och se till att allt som byggs är i fas. Jag har många gånger försökt bygga flera moment på en gång, men blir det problem i ett steg så brukar felsökningen ta mycket längre tid än vad jag sparade in på att ”bygga snabbare”, så numera försöker jag verkligen att bryta ner alla moment i ett separat testbart steg om möjligt.

4.5.4.1 Vill visa på hur det kan se ut, och hur vi kan felsöka fel

Jag hade ett ”skitfel” här: mina Group Descriptions var på svenska, och ordet ”tillått” i gruppbeskrivningen gjorde att mina Stacks failade när jag skulle skapa dem. Jag hittade felet genom att kika i Stacks, markera stacken, klicka ”events”-tabben (bilden ifrån senare tillfälle när allt funkar):

The screenshot shows the AWS CloudFormation console with two main panes. The left pane displays a list of stacks, including a nested stack named 'wp-cf-root-AlbStack-1E1CYGCLPBTUH'. The right pane shows the details for this specific stack, with the 'Events' tab selected. A yellow circle highlights the 'Events' tab, and a yellow arrow points from the stack name in the left pane to the same stack name in the right pane. Another yellow arrow points from the 'Events' tab in the right pane to the 'Events (14)' section below it. The 'Events' table lists 14 entries, each with a timestamp, logical ID, status, and detailed status. Most entries show a 'CREATE_COMPLETE' status, except for three which are 'CREATE_IN_PROGRESS' with the reason 'Resource creation Initiated'.

Timestamp	Logical ID	Status	Detailed status	Status reason
2025-08-31 00:56:32 UTC+0200	wp-cf-root-AlbStack-1E1CYGCLPBTUH	CREATE_COMPLETE	-	-
2025-08-31 00:56:31 UTC+0200	HttpListener	CREATE_COMPLETE	-	-
2025-08-31 00:56:30 UTC+0200	HttpListener	CREATE_IN_PROGRESS	-	Resource creation Initiated
2025-08-31 00:56:29 UTC+0200	HttpListener	CREATE_IN_PROGRESS	-	-
2025-08-31 00:56:29 UTC+0200	ALB	CREATE_COMPLETE	-	-
2025-08-31 00:54:01 UTC+0200	TargetGroup	CREATE_COMPLETE	-	-
2025-08-31 00:53:56 UTC+0200	ALB	CREATE_IN_PROGRESS	-	Resource creation Initiated

På det här viset såg jag att det var ”ASCII error on the Group Description” = jag bytte gruppbeskrivningarna till Engelska. Lustigt nog skapar jag alltid beskrivning, kommentarer och filnamn på Engelska, utom just den här gången. Men felsökningsverktygen var kanon!

4.5.4.2 Testar även health check på ALBn:

The screenshot shows the AWS CloudFormation console with two tabs: 'Resources' and 'Outputs'. The 'Resources' tab lists several resources with their status and creation time:

- wp-cf-root-AlbStack-1E1CYGCLPBTUH: NESTED, CREATE_COMPLETE, 2025-08-31 00:53:43 UTC+0200
- wp-cf-root: NESTED, CREATE_COMPLETE, 2025-08-28 12:21:00 UTC+0200
- wp-cf-root: NESTED, CREATE_COMPLETE, 2025-08-28 12:20:58 UTC+0200

The 'Outputs' tab shows the following details for the ALB:

Output Key	Value	Description
AlbDNS	alb/5779d6bed4f0c092	Public DNS för ALB
AlbSecurityGroupId	sg-01a63931b93bc4c7b	-
AppSecurityGroupId	sg-0398eb1b80a6b0a08	SG-ID för webbinstanser (används av LT/ASG, namnet hintar om det)
EfsSecurityGroupId	sg-02f93cae11b352594	-
ListenerArn	arn:aws:elasticloadbalancing:eu-west-1:355235952342:listener/app/wp-cf-alb/5779d6bed4f0c092/b	-

A yellow arrow points from the 'wp-cf-root' resource in the Resources list to the 'AlbDNS' output value in the Outputs table. Another yellow arrow points from the 'AlbDNS' value to the URL in the browser bar.

Browser bar: ← → ⏪ ⓘ Not Secure wp-cf-alb-1147505718.eu-west-1.elb.amazonaws.com

ASG OK (utan EFS)

Server: ip-10-20-11-17.eu-west-1.compute.internal

Nuff said. Ibland räcker det bra med att ha en liten health check och kolla URLEN.

4.5.5 Små justeringar gör stor skillnad

Det är dags att övergå från att köra lokalt och gå över till att köra över Internet. ASG får montera EFS (ASG behöver FileSystemId). Samtidigt får vi inte fastna i cirkelberoendet som dök upp mellan AsgStack och EfsStack.

Nyckeln är att låta EFS-SG tillfälligt tillåter NFS (port 2049) från app-subnätens CIDR (istället för från App-SG). Då kan AppStack referera FileSystemId från EfsStack utan loop. Ett fåtal justeringar av Resources, Parameters etc, små förändringar men en annan route.

Validera lokalt 30-efs, 50-asg-wordpress och root.yaml. Ladda upp. Uppdatera root-stacken.

4.5.6 Tightar till avgränsningen och förbereder för Maria DB

Det gör vi genom att skapa DB-SGn som en förberedelse för nästa steg, egen EC2.

4.5.6.1 Börjar med att ändra i den templetten jag redan hade

40-rds-mariadb.yaml var det utkastet jag skapade initialt för att ha som nested stack för mariadb + sg. Tanken var att möjligtvis använda RDS (AWS managed databastjänst). Nu bygger jag en EC2-baserad MariaDB, baserad på utbildningsmaterialet, så att ha "rds" i namnet blir missvisande. Döper om filen till 40-db-sg.yaml.

Skapar resources för Db-SG, tillåter port 3306 från App-SG. Output Db-SG-Id. Validerar. Laddar upp. Kör dock inte update-stack! Vi vill lägga till instansen i nästa avsnitt först.

4.5.6.2 Bygger en helt ny yaml, 45-db-mariadb-ec2.yaml för att hantera MariaDB-instansen

Vi behöver skapa underlaget för MariaDB på en EC2-instans: IAM-roll, plus en instans i ett privat DB-subnet. Det blir en del parametrar att sätta, 45-db-etc-filen blir en av de längsta, men det är viktigt att definiera rätt parametrar nu. Det kräver lite mer av testning med:

Validering lokalt som vanligt av yaml-filerna. Ladda upp till S3. Uppdatera root nu. Sedan verifierar vi att stacken är complete på AWS Stacks, och i det här läget vill jag gärna ansluta till EC2an för att kolla system-status. Databasen är oerhört viktig i den här setupen, när den väl är uppe skall helst bara App-SG (Wordpress) kunna prata med DBn och sedan skall helst DBn vara uppe och aldrig röras.

4.5.6.3 Här fick jag ett fel på update-stacks

Stack Name	Status	Last Updated
wp-cf-root	UPDATE_COMPLETE	2025-08-28 12:20:58 UTC+0200
DbEc2Stack	CREATE_COMPLETE	2025-08-31 23:04:23 UTC+0200
AsgStack	CREATE_COMPLETE	2025-08-31 23:04:01 UTC+0200
EfsStack	CREATE_COMPLETE	2025-08-31 00:56:46 UTC+0200
AlbStack	CREATE_COMPLETE	2025-08-31 00:53:43 UTC+0200
NetworkStack	CREATE_COMPLETE	2025-08-28 12:21:00 UTC+0200

CF > Stacks > root-stacken och Event-tabben visade "CREATE_FAILED", med texten "Requires capabilities: [CAPABILITY_IAM]". Vad jag missat var helt enkelt att slänga in – capabilities som flagga när jag körde update-stack. Det är en bekräftelse-rad som talar om för CloudFormation att min uppdatering skapar eller ändrar IAM-resurser, även i nested stacks.

Jag la ju till IAM i DbEc2Stack, så jag behövde uttryckligen godkänna skapa/ändra. Uppdaterade mina create/update-templates i README så den inkluderar – capabilities. Nu börjar setupen bli mer avancerad så det gäller att hålla tungan rätt i mun.

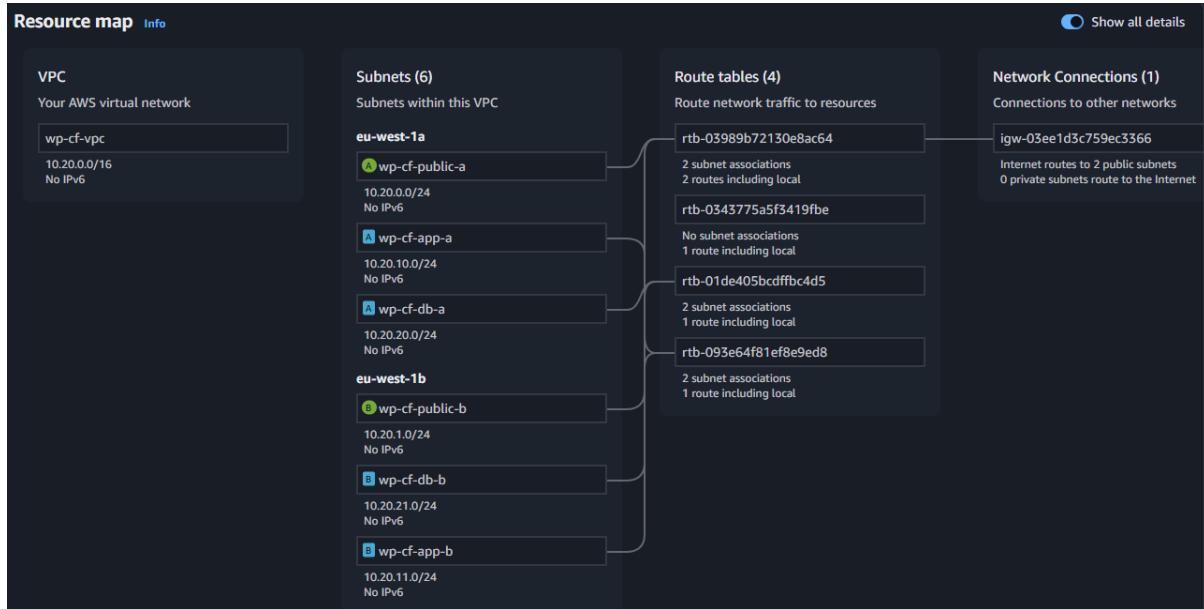
4.5.6.4 Kort verifierings-check innan ASG-Wordpress

Nu vill jag bara verifiera SG-reglerna:

- ALB-SG -> App-SG: TCP port 80 (plus ev.443) till ASG-instanser
- App-SG -> DB-SG: TCP port 3306
- App-SG -> EFS-SG: TCP port 2049 (NFS)

4.5.7 Skarpt läge: nu blir vi bloggiga

Än så länge har jag jobbat med lokal routing, min Resource Map visar 0 privata subnät routas till internet. Mina subnät (10.20.10.0/24 och 10.20.11.0/24) har bara local-route.



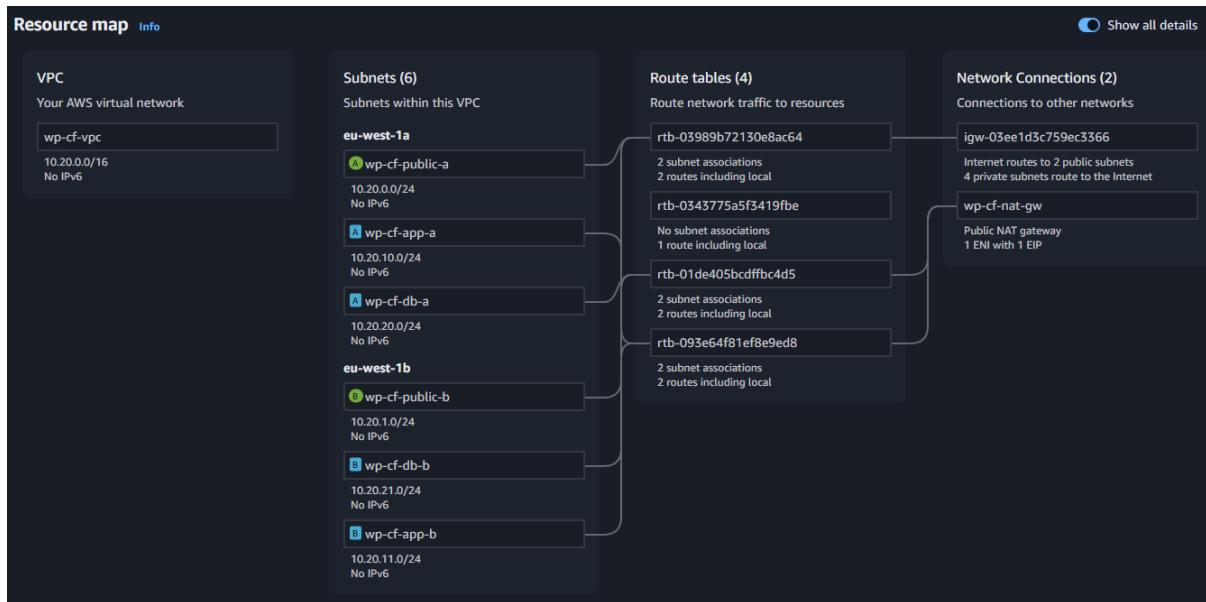
4.5.7.1 Slänger in NAT + default routes för privata subnät i 00-network.yaml först

Vad gör en NAT (Network Address Translation) Gateway? Den låter resurser i mina privata subnät (t ex EC2 i PrivateAppSubnet) nå ut till Internet för utgående trafik – om vi vill köra uppdateringar server-side som ex ”yum update” eller tillåta hämtning av WordPress-paket och liknande så behövs detta.

Det fina är att dessa resurser får ingen publik IP så de kan inte nås direkt från Internet. Trafiken går via NAT gatewayen så servrar kan nå ut, men inte nås utifrån.

Obs! Vi måste notera kostnadsaspekten på att använda en NAT Gateway: den kostar lite över 30 kr i månaden + datatrafik. Jag valde att ta med den i uppgiften eftersom jag bara har systemet uppe i några dagar.

Validerar, laddar upp, kör update-stack. You know the drill. Jämför nedanstående med ovan:



4.5.7.2 Alternativa lösningar till NAT som är mer kostnadseffektiva

Vill vi kunna köra uppdateringar i privata instanser utan att köra med NAT-tjänsten så hade vi kunnat titta på lösningar som de här:

- VPC-endpoints: jobba med Gateway endpoints via S3-bucket
- NAT-instans: Ersätt NAT GW med en billig EC2 (ex t3.nano) i public subnet med IP-forward och iptables Masquerade. Ok för labb men inte riktigt i cloud native-anda
- Packa in AMIs med alla paket förinstallerade: httpd, php, wp, plugins, då behöver de inte köra uppdateringar. De blir immutable, men merjobb i förarbetet

4.5.7.3 Testar att ALB -> ASG är healthy med en minimal Apache-sida + förklrarar User Data

I detta steg rör jag bara User Data i 50-asg-wordpress.yaml. Jag kör detta:

```

UserData:
  Fn::Base64: !Sub |
    #!/bin/bash -xe
    exec > /var/log/userdata.log 2>&1

    # Support both AL2 (yum) and AL2023 (dnf)
    PKG="yum"
    command -v dnf >/dev/null 2>&1 && PKG="dnf"

    $PKG -y update
    # httpd + php is enough for now
    $PKG -y install httpd php || $PKG -y install httpd

    systemctl enable httpd

    mkdir -p /var/www/html
    cat >/var/www/html/index.php <<'PHP'
    <?php
      header('Content-Type: text/plain');

```

```
        echo "OK from " . gethostname() . "\n";
    ?>
PHP

systemctl restart httpd
```

Förklarar några koncept i den:

- Fn::Base64 gör att userdata skickas Base64-kodad, på instansen avkodas det av cloud-init
- -xe: -x betyder att bash skriver ut kommandot innan det körs – bra för loggning, och -e betyder ”exit on error”, dvs cloud-init stoppar istället för att gå vidare med halvinstallerade gejer om det blir ett problem
- exec > byter den nuvarande processens stdout/stderr (out och error) till filen jag specificerar, allt loggas dit. 2>&1 redirectar ”file descriptor 2” (stderr) till samma destination som stdout
- PKG är packagemanager, jag justerar den till att dynamiskt köra ”yum” på Amazon Linux 2 och ”dnf” på Amazon Linux 2023. ”command -v dnf” returnerar 0 om dnf finns, då byts det

4.5.7.4-a Känner ingen OrdPress, EFS och User Data, men jag pillar på er

Lägger till en Parameter: EfsFileSystemId i 50-asg-wordpress.yaml. Nu skall vi mounta EFS och symlink av wp-content i User Data. User Data får köra en räcka installs nu: httpd, php, amazon-efs-utils, nfs-utils.

Här behöver vi också lägga till EfsFileSystemId under Parameters i AsgStacken i root. Laddar upp root.yaml och 50-asg-wordpress.yaml till S3. Kör update-stack. Bekräftar i konsollen på AWS att Target Group fortfarande är Healthy.

Det blir fel! Jag skapade EfsFileSystemId men hade redan FileSystemId. Här blir det tydligt hur viktigt det är med namngivning och tydlighet! FileSystemId är alltså EFS-ID! Så jag har nu skapat en ny variabel som är en dubblett av den gamla, men som pekar på olika namn. Det optimala hade varit att allting nu hette Efs-etc, men eftersom File-etc ligger överallt så tog jag bara bort den nya raden i root, och bytte i User Data. Nu funkar allt!

4.5.7.4-b Vad är symlink och varför är himlen blå?

Symlink är en pekare: /var/www/html/wp-content är en länk som pekar på /mnt/efs/wp-content. Vi lurar WP att wp-content ligger i webroot men filerna ligger på EFS och delas av alla instanser. Det görs genom ln -sfn /mnt/path /var/path.

4.5.7.5 Kritiska element avlöser varandra i detta skede

Vi behöver ha kopplingen till databasen fungerande från App/WP-instansen. Vi behöver ett antal Parametrar i 50-asg-wordpress.yaml: DbEndpoint, DbName, DbUser, DbPassword. Sedan får User Data uppdateras med att lägga till en WordPress-installation och wp-config. Lägger till ParameterKey/Value för DbName, DbUser och DbPassword i vår params.json. Måste även lägga till dem som parametrar i root så den kan hitta dem.

Här leker jag med döden och lägger till en ”salter”: WordPress använder hemliga nycklar och så kallade ”salts” för att göra cookies/sessioner svårare att gissa, och skydda lösenordshashar. Jag kan inte låta bli att göra det mer komplicerat, suck. Typiskt mig. Så vi hämtar ett gäng slumpade AUTH_KEY och knyter dem till wp-config.php. För att detta inte skall råka paja resten av systemet så har vi lagt till || true i slutet, då fortsätter scriptet även om vi skulle misslyckas att krydda vår blog.

Laddar upp filerna till S3, kör update-stack.

4.5.7.6 Massiva problem med allting ungefär

Det blir för långt att skriva hela detta, men jag skrev en [LinkedIn-inlägg](#) om det. Citerar delar av det inlägget:

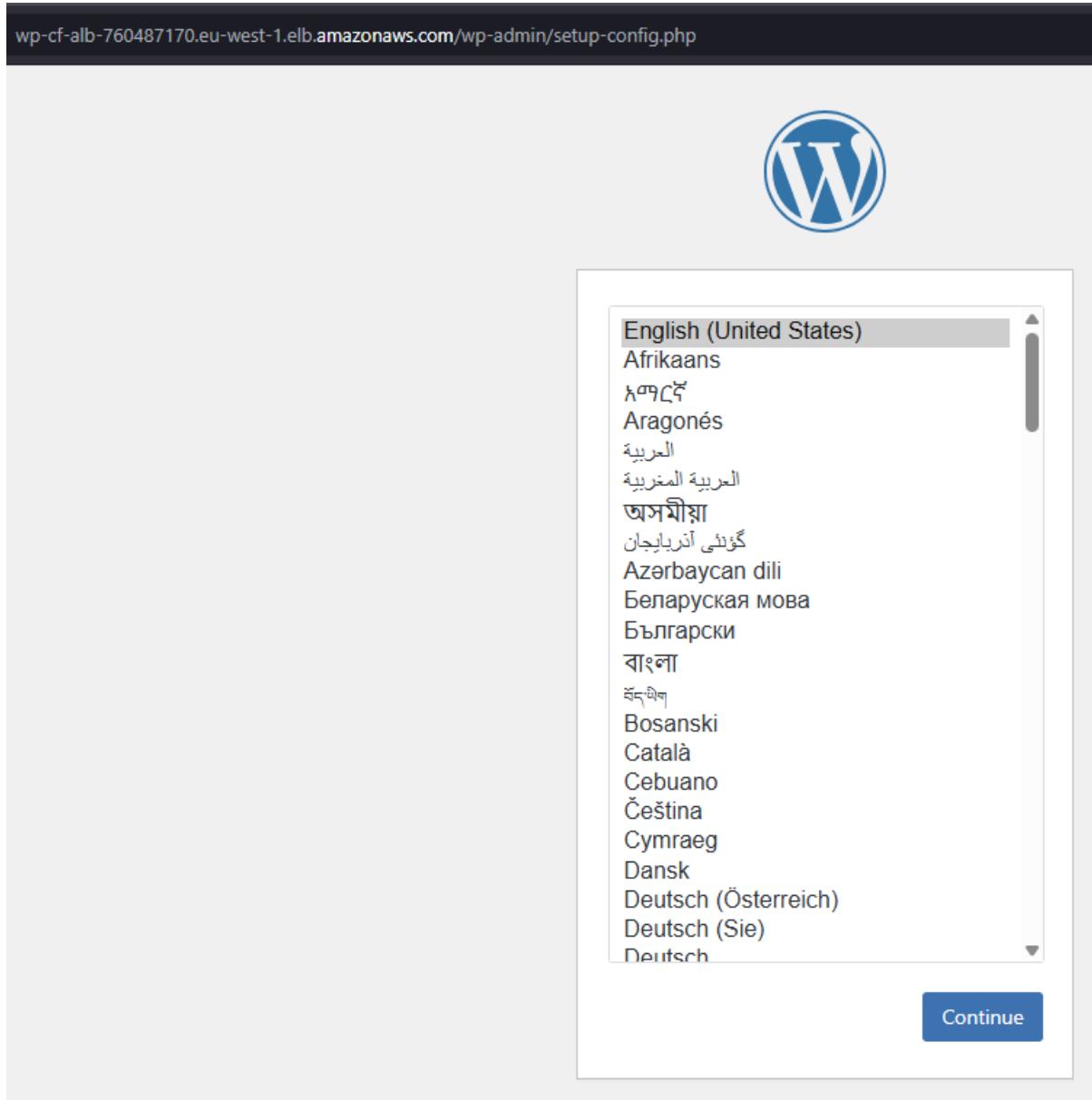
” Jag är trött och vill forcerat. Får en 502 Bad Gateway. I trötthetsrusen så slänger jag in hela kodbasen till en LLM och ber den leta efter skäl till varför jag skulle få en 502. Den vill ändra UpdatePolicy på ASGn till InstanceRefresh. Ok, jag testar. ASGn går ner, fastnar i en UPDATE_ROLLBACK_COMPLETE.

Jag kollar loggar i CLI och Event, upptäcker direkt att InstanceRefresh är en felaktig parameter så byter tillbaka till UpdatePolicy. Skickar upp till S3. update-stack på root-yamlen. Inget händer. Ser i loggarna att ASGn är fast i en loop och inte vill skrivas över. Jag tar ner alla stackarna och rebuildar genom create-stack och root. Allting grönt! Men 502 Bad Gateway igen.

I det här läget går jag tillbaka till att använda min egen hjärna istället för att dumpa kod och loggar på LLMen. Små steg. Tar bort all WP + PHP + Apache. Kör en kort health-html i User Data och deployar - jag ser health-sidan. Så nät/DNS/SG/TG/ALB/NAT/IGW/listaalltihelanätverket fungerar. Det är applikationslagret som är problemet.

Uppdaterar User Data med enbart Apache/PHP, ingen WP än. Allting fungerar. Uppdaterar User Data med WP i nästa steg.”

4.5.7.7 ä-n-t-l-i-g-e-n



Så nu finns WP-installern där på sin EC2a som den skall. Det som återstår är att koppla DBn till den. Vad lär vi oss av detta? Små inkrementella steg är alltid, alltid, rätt. Även när LLMer blir inblandade. Felsökningen blir så pass mycket lättare och du får en överblick.

4.5.8 En databas, en databas, mitt Wordpress-rike för en databas..!

Låt oss knyta ihop säcken. Med en databas är setupen komplett.

4.5.8.1 Databasens privata IP är bra att ha

Jag hade redan DbName, DbUser och DbPassword som Parameters i dev.params.json, root.yaml och i 50-asg-wordpress.yaml. Jag skapade dem även i 45-db-mariadb-ec2.yaml (inser även att jag skriver "db" två gånger i det filnamnet, klantigt). För även in dem i DbEc2Stack i root.

Nu kan vi gå vidare med att fylla i formuläret för db connection details för WP. Jag har redan satt värden för Database name, Username och Password i .json-filen. Men vi behöver komma på vad DB-instansen har för privat IP (inte publik DNS). Databasen skall ju bara nås av WP och ingenting annat än WP! Den hittar jag via AWS Console – CloudFormation – DbEc2Stack – Outputs. Se nedan:

Key	Value	Description
DbAz	eu-west-1a	AZ of the DB instance
DblInstanceId	i-0c4dfff6610afe40e	EC2 instance ID for MariaDB
DbPrivateIp	10.20.20.105	Private IP of the DB instance

Det borde se ut ungefärlig så här:

Below you should enter your database connection details. If you are not sure about these, contact your host.

Database Name	wordpress
Username	blog_king
Password
Database Host	10.20.20.105
Table Prefix	wp_

Submit

4.5.8.2 Fel fel fel fel fel, dags att söka rätt stig igen

En fördel med de här små inkrementella stegen är att felsökningen blir så naturlig. Nu fick jag ett problem: när jag klickar Submit får jag en 502 Bad Gateway. För att undvika att göra samma misstag som igår när jag felsökte flera saker samtidigt så väljer jag ett minimalt steg: jag lägger till en test i User Data som kollar att vi når den privata IPn, och sedan printer ut det i en sida som får denna länk: <http://<ALB-DNS>/dbtest.php> – och den sidan returnerade "TCP OK" med IP- + portnumret. Bra! Så routingen är ok!

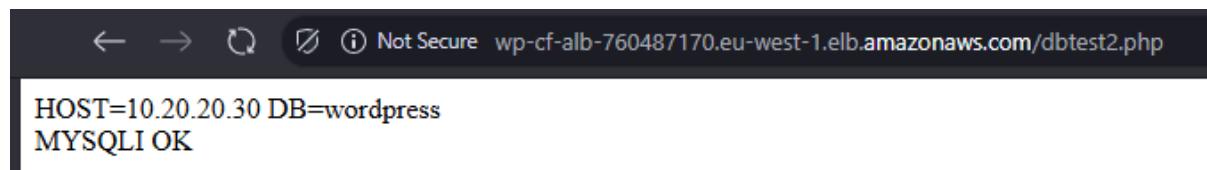
5. La Grande Finale

Nåja. Jag satt med databasen och finishing touch när jag blev tvungen att släppa det och gå in för att även få till dokumentation i tid. Det är tjugo minuter kvar innan detta skall lämnas in. Applikationen är inte 100% klar men väl 99%.. Se nedan:

5.1 Uppdatering: Status på applikationen

Databasen är uppe, den ligger på sin egen instans, jag har kört health checks på alla nätverksåtkomster – ~~det enda som saknas är att SSHa in via Bastion till DB och fixa access för WP. Det är inte många minuters jobb.~~

Nu är det löst.



5.2 Bu och bä, styrkor svagheter, vad har vi lärt oss

Always do

- Små inkrementella förändringar har varit guld
- Dokumentera on the fly, det är lätt att glömma vad du gjort och det är ett väldigt stort stöd efteråt (även om du gör många saker samtidigt, kognitiv belastning etc)

Never do

- Skapa inte onödig komplexitet (liksom jag gjort), det ökar dramatiskt felprocenten, underhållsbehov och läsbarhet av kod / lösningar

Var det rätt lösningar / hade du gjort något annorlunda?

Jag tycker min lösning i stort sett var bra, det som kunde varit bättre var strukturen på vissa arbetsmoment. Men de tekniska valen tycker jag är solida, och har fungerat bra både för den här uppgiften, men även för att bygga vidare på.

Referenser / noteringar

1. Studieguidens tutorial: <https://cloud-developer.educ8.se/clo/3.-scalable-cloud-applications/1.-tutorials/1.-provision-a-vm-on-aws-using-cloudformation/index.html>
2. Tutorial: Host a WP-blog on AL2023:
<https://docs.aws.amazon.com/linux/al2023/ug/hosting-wordpress-aml-2023.html>
3. Install WP on Amazon Linux 2023: <https://repost.aws/articles/ARarysTJYQTh-zCNMftlolwA/install-wordpress-on-amazon-linux-2023-al2023>
4. LLM: Partners (and enemies) in crime ChatGPT and Claude