Adobe Experience Manager

Mike Heymans

2017

Inhoudsopgave

1 Voorwoord

In het kader van mijn opleiding Toegepaste Informatica aan de Hogeschool Gent heb ik onderzoek gedaan naar het Adobe Experience Manager platform, beter gekend als AEM. Het resultaat dat nu voor u ligt is er één van drie maanden hard werk waarin heel wat gevloek heeft plaatsgevonden. In nazicht van dit onderzoek ben ik tevreden over het resultaat maar vooral voldaan met de verworven kennis. Al heeft deze kennis gezorgd voor nieuwe vraagtekens die in de toekomst mijn aandacht zullen opslorpen.

Deze proef was niet mogelijk geweest zonder de inzet van AS Adventure, die zo vriendelijk waren om een licentie beschikbaar te stellen alsook mij te begeleiden in mijn ontdekkingsreis binnen de wereld van AEM. Ik wil ook mijn collegas bedanken voor hun uitmuntende begeleiding en eindeloos geduld.

In het bijzonder wil ik mijn begeleiders Koen Hoof en Steven Rymenans bedanken voor hun ondersteuning tijdens deze reis.

Veel plezier met het lezen van deze scriptie.

2 Reden van onderzoek

Adobe Experience Manager is het gelicentieerde contentmanagementsysteem (CMS) van Adobe. Een CMS-applicatie maakt het mogelijk voor mensen zonder kennis van html of css om webpaginas aan te maken of te wijzigen. Het is ideaal voor websites die vaak de inhoud van hun site wijzigen of snel willen inspelen op actuele gebeurtenissen. Dit komt doordat designers en marketeers zelf de pagina's kunnen editeren in plaats van eerst content uit te denken en de realisatie over te laten aan een front-end developer.

Toen AS Adventure besloot om zijn huidige website, en die van zijn zusterbedrijven, niet langer uit te besteden aan een extern bedrijf maar deze intern te gaan beheren en ontwikkelen, is er besloten om deze met het AEM-platform op te zetten. Dit zou het bedrijf in staat stellen zijn content volledig door de designers te laten beheren, zonder tussenkomst van developers. Denk maar aan de homepage die van achtergrondafbeelding wijzigt of een gepersonaliseerde pagina voor een merk, allemaal mogelijk dankzij AEM.

Dat dit allemaal out-of-the-box mogelijk is, is te mooi om waar te zijn. Voor het content team aan de slag kan moeten er componenten gebouwd, dit is het werk van de ontwikkelaars. Eigenlijk komt het erop neer dat componenten de bakstenen vormen om een website op te bouwen en de designers zijn de metsers. Ze kunnen lustig huisjes bouwen maar wanneer er nood is aan een baksteen met een nieuwe feature moet hiervoor een aanvraag gedaan worden bij de steenbakkers, gespeeld door de ontwikkelaars. Deze stenen bevatten niet enkel HTML en CSS maar ook Java (en diverse frameworks) zijn nodig om dit nieuw type steen te ontwikkelen. De designers krijgen de nieuwe baksteen en kunnen hiermee aan de slag gaan.

Buiten de content die door het marketingteam wordt voorzien is er ook data die vanuit een databank moet komen. Hierbij hebben we het over records die niet manueel worden aangepast maar met duizenden tegelijk of data die niet enkel betrekking heeft op de site maar ook tijdens andere processen een rol spelen. Een voorbeeld hiervan zijn de prijzen, tijdens de solden is het niet reel om elke prijs handmatig aan te passen op de paginas. In het onwaarschijnlijke geval dat het handmatig aanpassen van een specifieke prijs nodig is, zou deze data enkel op de pagina gewijzigd zijn en niet in het ERP. Het is dus logischer dat het ERP deze wijziging doet en door duwt naar de pagina. Ook hier spelen de ontwikkelaars een rol, het is hun taak de systemen die dit mogelijk maken te bedenken en te realiseren.

Toen AS Adventure aan dit project begonnen waren er processen uitgedacht om deze wijzigingen weer te geven en toen de eerste twee sites, Juttu en Yaya, live stonden was iedereen tevreden over het resultaat. Maar deze shops zijn relatief nieuw en beperkt in aanbod, toen de vernieuwde AS Adventure site live ging staken er enkele problemen de kop op. Aangezien deze problemen in productie

voorkwamen was wachten geen optie en moesten deze opgelost worden, niet zozeer met de beste maar wel de snelste oplossing.

Deze proef heeft als nut het herzien van wat er tot nu toe opgebouwd, hoe het AEM-platform aangepast is geweest om aan de behoeften te voldoen en of dit ook effectief de beste manier is om de vereiste functionaliteit te bekomen. Door het in kaart brengen van onze methodologien hopen we een dieper inzicht te verwerven in het platform alsook een leidraad te voorzien voor ontwikkelaars die een eerste keer kennis maken met AEM of zij die reeds werken met AEM en geconfronteerd worden met de problemen waarvoor wij een oplossing zoeken. Zoals het gezegde luidt: het is goed te leren van je fouten maar beter om te leren uit die van een ander.

3 De database

Om onze AEM-applicatie te bouwen hebben we nood aan een database waaruit we onze informatie kunnen halen. De manier waarop we deze informatie halen zal afhangen van model tot model, sommige data zal naar AEM worden gestuurd wanneer deze gewijzigd is, andere zal worden opgehaald wanneer deze vereist is. We zorgen om deze reden dat beide gedachtegangen mogelijk zijn met elk model.

3.1 Apache Cassandra

Apache Cassandra is een open-source NoSql (Not only Sql) database die de dag van vandaag door vele internationale bedrijven in gebruik is genomen. Netflix, Instagram en het CERN hebben in Cassandra een performante, stabiele databank gevonden, die de beschikbaarheid van hun data garandeert.

Deze garantie wordt geboden door de structuur waarmee Cassandra data beheert. Een databank bestaat uit een collectie van noden die een cluster vormen. Het beheer gebeurt decentraal, elke node heeft dezelfde rol, dit maakt dat wanneer een node offline is, de andere ongehinderd blijven werken. Om het volle potentieel van Cassandra te benutten, is het aangeraden om de noden niet enkel over verschillende servers maar ook over verschillende datacenters te verspreiden. Bij deze setup mag er een datacenter offline gaan, de gebruiker zal hier geen weet van hebben. En wanneer het datacenter terug online is, zal de data gedeeld worden met de herstelde noden. Dit is ook positief voor de schaalbaarheid. Wanneer een node wordt toegevoegd, haalt deze zijn configuratie op bij een bestaande node en sluit zich naadloos aan in de cluster.

Bij het aanmaken van een cluster kan men een replicatie factor meegeven. Deze geeft aan op hoeveel verschillende noden een record moet bewaard worden. Bij een factor 3 zal elk record op 3 noden beschikbaar zijn. Des te groter de factor, des te groter de performantie en stabiliteit maar ook de duplicatie aan data vergroot mee. Bij het kiezen van de replicatiefactor moet er een afweging gemaakt worden tussen niveau van beschikbaarheid en het extra geheugen dat de replicatie vereist.

Een groot verschil met een relationele databank is dat het linken van verscheidene tabellen niet mogelijk is. In plaats daarvan werkt Cassandra met collecties

die rechtstreeks in een kolom worden opgeslagen. Dit wordt mogelijk door het aanmaken van UDTs (User Defined Types). Wanneer een lijst van UDTs (of primitieve types) wordt opgeslagen, wordt deze omgevormd naar een json-achtige structuur. Doordat alle data in n rij zitten, hoeven er geen joins gedaan te worden over meerdere tabellen wat de performantie aanzienlijk verhoogt.

Er zijn ook enkele nadelen verbonden aan het kiezen voor Cassandra als database, één van deze nadelen is dat het, out-of-the-box, onmogelijk is om tabellen te querien zoals je met een relationele databank zou doen. Dit komt doordat Cassandra werkt met key-value paren om de data op te slagen en dus enkel aan de hand van een key (of deel van een key) naar een record gezocht kan worden. Een bijgevolg hiervan is dat aggregatie functies enkel op partitie niveau kunnen worden uitgevoerd.

Aangezien Cassandra een non-relationele databank is, is een ander nadeel de afwezigheid van transacties op databank niveau. Enkel op rijniveau kan men gebruik maken van lightweight transacties door een conditie toe te voegen voor een schrijf operatie. Hiervoor zou men een versie kunnen toevoegen aan een tabel zodat wanneer men een schrijf operatie wil doen, er gekeken kan worden of de versie nog overeenstemt. Indien dit niet het geval is wordt de operatie niet uitgevoerd en kan het systeem op een gepaste manier reageren.

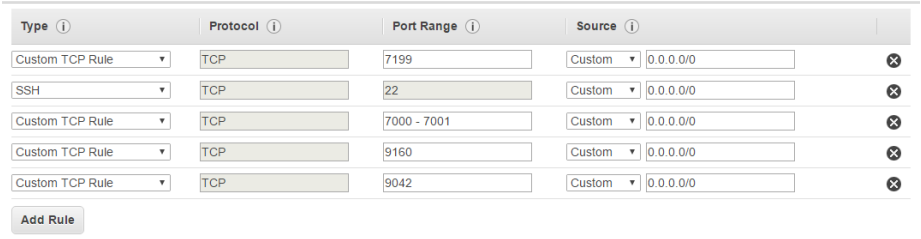
3.2 Een Cassandra cluster opzetten

Nu we onze databank gekozen hebben, is het tijd dat we deze opzetten. De stappen die nodig zijn, beginnend van niets, om een cluster op te zetten worden in dit segment besproken. We zullen eerst de machines aanmaken, vervolgens Cassandra installeren en als laatste stap onze nodes met elkaar in contact brengen. Voor dit project heb ik gekozen om met 3 nodes te werken, dit om toch enige performantie te hebben zonder het prijzig te maken.

3.2.1 De machines

Als eerste maken we de 3 machines aan waarop we Cassandra zullen draaien. Via de AWS-console navigeren we naar EC2, Instances waar we de Launch instance kiezen. Het eerste scherm laat ons kiezen welk besturingssysteem we willen gebruiken, hier kiezen we Amazon Linux AMI. In stap 2 bepalen we hoe krachtig onze machines zullen zijn. Het is perfect mogelijk om met de t2.micro (deze is, voor beperkte tijd, gratis te gebruiken met een Free Tier account) aan de slag te gaan en voor experimentele doeleinden is dit meer dan genoeg. Stap 3 hoeven we enkel het aantal instances dat we willen aanmaken wijzigen naar 3, de rest laten we ongewijzigd. Stap 4 en 5 slagen we over, deze zijn niet van toepassing. In Stap 6 maken we een nieuwe security group aan die de poorten

nodig voor het gebruik van onze databank openzet. Voor onderstaande poorten maken we een Custom TCP Rule en maken we ze compleet publiek (in productie omgevingen is dit ten strengste af te raden.) door de toegestane IP-adressen op 0.0.0.0/0 te zetten. Figuur 1 toont ons hoe we dit bekomen.



Figuur 1: security settings.

Indien deze stap afgehandeld is, krijgen we een overzicht te zien. Als alles naar wens is klikken we op Launch waardoor er een prompt opengaat. Vanuit deze prompt kunnen we een pem-file aanmaken en downloaden, het is zeer belangrijk om deze op een veilige plaats op te slaan, deze is nodig om via ssh een connectie naar onze machines te maken. Bij het aanmaken van nieuwe machines kunnen we dezelfde pem-file hergebruiken zodat we één key hebben voor al onze instances. We slagen het bestand op onder aem-ec2.pem.

3.2.2 Cassandra als service

Nu we onze machines hebben is het tijd om hiervan Cassandra nodes te maken. Via het Instance scherm kunnen we achterhalen wat de publieke IP-adressen zijn van deze machines (laten we ervan uitgaan dat deze 1.0.0.1,1.0.0.2 en 1.0.0.3 zijn.). Volgende stap moet op elke machine identiek herhaalt worden buiten de nodige aanpassingen aan de cassandra.yaml. Laten we eerst op onze machines inloggen via het commando:

```
$ ssh  i  aem-ec2.pem ec2-user@1.0.0.1
```

Nu zijn we ingelogd als ec2-user op onze machine. Om Cassandra 3.x te kunnen draaien hebben we Java 1.8 nodig, via het commando java -version kunnen we dit controleren. Indien de versie lager ligt, zijn we verplicht om Java 1.8 te installeren. Een mogelijke manier om dit te doen is als volgt:

We navigeren naar onze jvm folder

```
$ cd /usr/lib/jvm
```

Vervolgens downloaden we de java 1.8 tar file

```
$ sudo wget --no-cookies --no-check-certificate --
    header "Cookie: gpw_e24=http://www.oracle.com/;
    oraclelicense=accept-securebackup-cookie" http://
    download.oracle.com/otn-pub/java/jdk/8u121-b13/
    e9e7ea248e2c4826b92b3f075a80e441/jdk-8u121-linux-
    x64.tar.gz
```

En pakken we deze uit

```
$ sudo tar xzf jdk-8u121-linux-x64.tar.gz
```

Na het uitpakken kunnen we onze jdk registreren als java optie

```
$ sudo alternatives --install /usr/bin/java java /usr/
    lib/jvm/jdk1.8.0_121/bin/java 2
```

Als we nu het volgende commando runnen zien we twee java mogelijke java engines, diegene dat al genstalleerd was en onze nieuwe. Hier kiezen we om onze nieuwe als default te gebruiken.

```
$ sudo alternatives --config java
```

```
$ Enter to keep the current selection[+], or type
    selection number: 2
```

Als alles correct verlopen is zien we nu 1.8 staan wanneer we opnieuw het commando java -version uitvoeren.

Nu volgt de installatie van Cassandra zelf, hierbij is de eerste stap het toevoegen van de datastax.repo zodat we via het yum commando Cassandra kunnen installeren. Als dit gebeurt is kunnen we via yum zowel Cassandra als Nodetool installeren.

Maak het bestand datastax.repo aan.

```
$ sudo touch /etc/yum.repos.d/datastax.repo
```

Vul deze met de nodige data

```
$ sudo touch /etc/yum.repos.d/datastax.repo
```

En zorg dat deze conform is aan Figuur 2

Nu kunnen we Cassandra en Nodetool installeren.

```
$ sudo yum install dsc30
```

```
$ sudo yum install cassandra30-tools
```

Als we nu Cassandra zouden opstarten op de 3 machines, hebben we 3 clusters met elks 1 node gemaakt. Vanzelfsprekend is dit niet het gezochte resultaat en

```
[datastax]
name = DataStax Repo for Apache Cassandra
baseurl = http://rpm.datastax.com/community
enabled = 1
gpgcheck = 0
```

Figuur 2: datastax.repo

willen we 1 cluster met 3 nodes, om dit te verwezenlijken moeten onze cassandra.yaml aanpassen zoals getoond in Figuur 3 (locatie: /etc/cassandra/conf/-cassandra.yaml).

```
cluster_name: 'Aem-cluster'

seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: "1.0.0.1,1.0.0.2"

listen_address:
# Het publieke IP-adres van de machine
broadcast_address: 1.0.0.1
# Dit stelt poort 9042 open aan alle thirth party applicaties waaronder
# DevCenter en Java-applicaties
rpc_address: 0.0.0.0
# Default gelijk aan het rpc_address waardoor
# we deze manueel op het publieke IP-adres
broadcast_rpc_address: 1.0.0.1
```

Figuur 3: cassandra.yaml

Nu dat onze yamls correct staan kunnen we op onze machines cassandra starten met het commando sudo service cassandra start. Na enkele minuten zien we dan dat de 3 nodes elkaar gevonden hebben via het commando nodetool status.

3.2.3 DevCenter

Nu we onze databank hebben kunnen we onze nodige modellen toevoegen, het gebruikte script kan je terugvinden in de bijlagen. We voorzien enkele standaard tabellen waarmee we een online catalogus kunnen vullen. Voor elke tabel voorzien we een historiek tabel waarin we de gewijzigde rijen van een bepaalde periode kunnen ophalen. Voor het uitvoeren van deze scripts kan men gebruik maken van DevCenter, een open-source tool waarmee een connectie kan gemaakt worden met een cluster. Deze manier van werken is aangenamer dan telkens te

moeten sshen naar een node om daar via het cqlsh commando onze databank te kunnen querien. Eenmaal onze databank operationeel is kunnen we beginnen aan de service die deze zal aanspreken.

3.2.4 Elastic IP

Voor de lezers die van plan zijn om hun machines uit te zetten tijdens de perioden dat je een pauze neemt van deze proef, wil ik melden dat wanneer je machines uitgezet worden, ze hun IP-adres verliezen. Tijdens het rebooten zal er een nieuw adres worden toegekend waardoor deze niet meer matchen met diegene die we in de yamls hebben opgegeven. Hierdoor kunnen de noden elkaar niet meer vinden, dit is iets waar ik persoonlijk ben tegengelopen. Een oplossing is om via AWS een Elastic IP toe te kennen maar dit is niet gratis (!) en betaal je per uur dat je machine afstaat.

4 Data-service

4.1 De Service

Om onze databank aan te spreken gaan we een rest service voorzien waar we enkele simplistische methodes toekennen die het mogelijk maken om: de gewijzigde data periodiek op te halen aan de hand van een datum die fungeert als ondergrens en een endpoint waar we n element kunnen ophalen aan de hand van een id. Dit maakt het mogelijk om periodiek de gewijzigde modellen uit onze databank op te halen en naar AEM te versturen. Of deze manier van werken haalbaar is voor al onze modellen gaan we later uitmaken. Voor producten voorzien we een extra endpoint om deze aan de hand van een categorie op te halen.

We gebruiken hiervoor een Java Spring-boot service omdat deze makkelijk in opzet zijn, gezien het een rest service is kan hier perfect voor een andere programmeertaal gekozen worden. Onze configuratie doen we via een application.yml bestand, deze wordt automatisch door de laatste versie van Spring-boot ondersteund. We voorzien ook de docker-maven-plugin om het onszelf iets makkelijker te maken. We configureren deze zoals Figuur 4. De configuratie die we meegeven is redelijk transparant, imageName duidt aan hoe we onze docker images gaan noemen, een samenstelling van de artifact naam en versie zorgt ervoor dat elke versie van onze service een unieke imageName krijgt. Moest er de nood zijn om een vorige versie terug te zetten, kan dit in een handomdraai. De dockerDirectory duidt waar we onze image willen aanmaken, de properties onder resources vertelt Docker wat er in de image moet worden opgenomen. Als Docker correct genstalleerd is, kunnen we het commando 'mvn clean package docker:build' uitvoeren om onze image te bouwen.

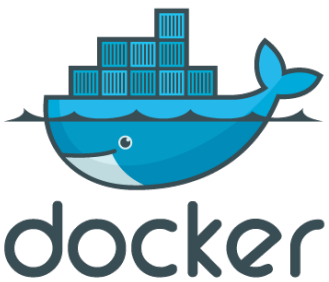
```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.11</version>
    <configuration>
        <imageName>${project.artifactId}:${project.version}</imageName>
        <dockerDirectory>src/main/docker</dockerDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

Figuur 4: Maven Docker Plugin.

Om te connecteren met onze Cassandra cluster maken we gebruik van de Datastax java-driver. Datastax is een bedrijf dat een commercile versie van Apache Cassandra aanbiedt alsook support voorziet. Hun Java-driver is open-source en dus gratis te gebruiken, in mijn ervaring biedt deze een betere ondersteuning voor het gebruik van UDTs tegenover de JPA-implementatie voor Cassandra. Om deze driver te gebruiken moeten we onze klassen die overeenstemmen met een tabel in onze databank annoteren met @Table. De velden in deze klassen krijgen nog de annotatie @Column en @PartitionKey als het gaat om een primary key kolom. Voor de types van onze databank voorzien we ook klassen voor die we annoteren met @Field. De velden hier annoteren we met @Field. Eenmaal we onze tabellen hebben vertaald naar overeenstemmende klassen volstaat het om een sessie met onze cluster aan te maken, bij het aanmaken hiervan volstaat het om het IP-adres van één node te voorzien, de driver zal dan via discovery de locatie van de overige nodes vinden en zo fouttolerantie te waarborgen. Het is toch aan te raden om meerdere IP-adressen mee te geven zodat wanneer er een node niet beschikbaar is tijdens het aangaan van de sessie, deze kan terugvallen op een andere node. Verder voorzien we per model een rest controller met de methoden die we reeds besproken hebben.

4.2 Docker

De volgende stap in ons process is een manier zoeken om onze service in de cloud te draaien. Het is aanvaardbaar om tijdens de ontwikkeling onze service lokaal te draaien maar als we een volwaardig platform willen creeeren moet ook dit onderdeel remote draaien.



Docker is een open-source project dat dit process voor ons zal versimpelen. Vroeger moest er heel wat tijd (en bijgevolg ook budget) gespendeerd worden aan het werkende krijgen van services op verschillende machines. Dit komt omdat niet alle machine hetzelfde geconfigureerd zijn en niet elke machine ondersteund alles even goed. Docker lost dit probleem op door maar één vereiste te hebben, dat de machines de Docker service hebben draaien. Oorspronkelijk ondersteunde enkel Linux systemen deze service native maar ondertussen zijn ook Windows en Mac mee op de kar gesprongen. Om deze lokaal te trainen, voor testing doeleinden, kan men naar de docker website gaat en de gewenste versie downloaden. De tutorials die je daar kan vinden leggen perfect uit hoe je ermee aan de slag kan. Natuurlijk gaan we, na een korte uitlichting, onze Docker remote gaan draaien.

Wie Docker zegt, zegt containers. Om onafhankelijk van de host een applicatie te kunnen draaien, steekt Docker deze, en al het nodige (environment, tools, libraries,settings,...), in een container. Docker bundelt al het nodige samen en maakt er een exporteerbare image van. Deze image kunnen we dan eender waar draaien zonder ons zorgen te moeten maken om infrastructurele verschillen. Dit wil zeggen dat wanneer een image succesvol in een trainingsomgeving draait, deze zonder vrees overgezet kan worden naar een productie omgeving.

Een ander voordeel van het containersysteem is dat men elke container, tijdens het opstarten, specifieke parameters kan meegeven met betrekking tot de resources die deze ter beschikking krijgt. Als men enkele zwaardere services heeft, kan met de container hiervan meer RAM toekennen om aan de behoefte te voldoen. Het is perfect mogelijk om meerdere containers op n machine of verdeelt over meerdere machines te draaien wat de schaalbaarheid en beschikbaarheid waarborgt. Zelfs een release hoeft geen downtime meer te betekenen, de containers kunnen n voor n vervangen worden met een nieuwere versie.

Buiten de configuratie met betrekking tot de resources kan men ook omgevingsvariabelen meegeven (hoe dit mogelijk is zien we dadelijk). Dit heeft als voordeel dat we dezelfde container kunnen gebruiken voor onze trainings-en productieomgeving door ander variabelen, zoals de databaselocatie en credentials, urls van andere services, enz., mee te geven tijdens het starten van onze containers.

4.3 AWS en Docker

Nu wordt het tijd dat we onze services in de cloud gaan draaien. Hiervoor zijn enkele stappen nodig beginnend bij de installatie van de nodige software op onze ontwikkelings machine. De eerste installatie is die van de AWS Command Line Interface (of kortweg AWS CLI), gelukkig voor ons heeft Amazon hier een uitstekende handleiding[1] voor. Het is belangrijk niet te vergeten om na de installatie deze ook te configureren[2].

Het tweede wat we nodig zullen hebben is Docker op onze ontwikkelings machine, niet omdat we onze containers hier gaan draaien maar omdat we deze hier gaan bouwen. Voor diegene zonder ervaring met Docker raad ik deze manier aan om een beter begrip te krijgen van hoe dit in zijn werk gaat. De ervaren lezer mag natuurlijk zijn images op zijn gekozen manier bouwen. Om Docker te installeren kan de Linux-gebruiker zijn shell gebruiker, Mac en Windows hebben minder geluk en zullen een installer[3] moeten gebruiken.

Eenmaal onze ontwikkelingsmachine klaar is keren we terug naar AWS om een ECS-cluster (EC2 Container Service) op te zetten, hierin gaan we onze contai-

[1] http://docs.aws.amazon.com/cli/latest/userguide/installing.html
[2] http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html
[3] https://docs.docker.com/engine/installation/

ners laten draaien. Een cluster kan men zien als een logische groepering van machines, wanneer we dezelfde image meermaals deployen zullen de benodigde containers automatisch verdeelt worden over de machines in onze cluster.

Voor we aan onze cluster beginnen maken we snel 2 IAM rollen aan, n rol voor onze machines en n rol voor de service die we op onze cluster zullen starten. De rol voor de machines geven we een logische naam: ecs-instance-role en geven we de permissie AmazonEC2ContainerServiceforEC2Role Geeft de machine schrijfrechten op CloudWatch en ECS, leesrechten op ECR. De rol voor de service noemen we ecs-service-role en geven we de de permissie AmazonEC2ContainerServiceRole (schrijf-en leesrechten op EC2 en ELB).

Een ECS cluster opzetten is geen werk, even navigeren naar het scherm, kiezen om een nieuwe te maken, we geven deze een naam(bv. tst-ecs-cluster) en voor nu selecteren we de optie 'Create an empty cluster'. Wanneer we deze cluster aanmaken dan zien we deze verschijnen maar zonder machines in, deze toevoegen is een volgende stap.

Vervolgens maken we een Launch configuration aan, dit zal beschrijven hoe onze machines opgestart moeten worden. De configuratie heeft als voordeel dat we dit maar eenmaal moeten doen en dat alle machines binnen onze cluster met de zelfde configuratie opstarten. DIt is grotendeels gelijk aan het maken van een machine, let wel op dat je als AMI[4] de Amazon ECS-optimized[5] kiest. Dit zal onze machines automatisch van Docker voorzien en deze starten wanneer de machine boot. Ken de ecs-instance-rol toe aan onze machines bij de stap Configure details. In deze stap voegen we ook een script toe onder User data (Advanced Details). Hier schrijven we het script als gezien in Figuur 5. Dit zorgt ervoor dat de machines die met deze configuratie starten aan onze cluster worden toegevoegd.

```
#!/bin/bash
echo ECS_CLUSTER=tst-ecs-cluster > /etc/ecs/ecs.config
```

Figuur 5: ECS bootup script.

Nu we een Launch configuration hebben kunnen we deze gebruiken om effectief machines aan te maken. Dit doen we door een Auto Scaling Group aan te maken via de EC2 module. Tijdens het opzetten van deze groep kiezen we de Launch configuration die we juist hebben aangemaakt. We geven de groep een naam en hoeveel machines we willen bij de opstart ervan alsook een subnet, dit subnet kan gebruikt worden om strengere security filters in te stellen door enkel machines binnen dit subnet met elkaar te laten communiceren. Men zou dan via routing kunnen werken om een bepaald deel van de endpoints publiek

[4] Amazon Machine Image
[5] te vinden onder AWS Marketplace

toegankelijk te maken. In dit project gaan we hier niet verder op in. Onder Scaling Policies kunnen we beschrijven onder welke voorwaarden er meer machines moeten worden bijgezet, bijvoorbeeld als het CPU verbruik van onze cluster gedurende 5 minuten boven de 80% is. Omgedraaid kunnen we ook instellen wanneer er machines verwijdert mogen worden buiten de piekuren. Als alles goed verlopen is zien we na creatie een aantal machines, gelijk aan het gekozen aantal, opstarten onder ons EC2 Instances scherm. Deze machines zouden, dankzij het ecs-script, ook te zien zijn als bij onze ECS-cluster gaan kijken onder Instances.

Nu rest ons enkel nog het toevoegen van een loadbalancer en we zijn klaar om onze Docker containers te bouwen en te deployen naar onze nieuwe cluster. Een loadbalancer fungeert als gateway naar onze machines, inkomende requests op onze loadbalancer worden verdeelt en doorgestuurd naar de machines die de balancer als gezond beschouwt. Om de gezondheid van een machine te controleren doet deze periodiek een request naar de machines, zolang de machines met een status 200 antwoorden beschouwt de balancer ze als gezond. Hoe frequent en hoe vaak een antwoord positief of negatief moet zijn om de status van de machine te veranderen kan ingesteld worden. Ook de snelheid waarmee de machine moet antwoorden kan geconfigureerd worden, standaard is dit 5 seconden maar als je met gratis micro machines werkt verhoog je dit best. De netwerk capaciteiten van deze machines zijn beperkt waardoor de limiet van 5 seconden overschreven kan worden. Indien dit gebeurt, haalt de balancer deze machines uit rotatie.

Het volgende deel van het plan is bouwen van een image, via de maven plugin, en deze uploaden naar onze EC2 Container Repository. Vervolgens kunnen we een nieuwe service definiren voor onze ECS cluster, hiervoor kiezen we als task definition onze docker image. Op het einde van deze stap kunnen we, via de loadbalancer, onze services callen.

5 Obsidian Scheduler

Obsidian is een Java-based scheduler waar een organisatie wederkerende jobs kan instellen. Obsidian is gratis in gebruik als je enkel alleenstaande instances hebt lopen, d.w.z. geen master-slave verhoudingen tussen instances.

Om Obsidian te installeren hebben we een machine nodig waar reeds een Tomcat en databank op draait. Voor de database hebben we de keuze uit enkele populaire namen zoals MySQL en MS SQL. Wanneer aan deze twee vereisten zijn voldaan kunnen we de obsidian-x.x.x.jar downloaden en runnen met volgend commando.

```
$ java –jar Obsidian–Install–n–n–n.jar –console
```

Het '-console' zorgt ervoor dat we onze installatie zonder een UI kunnen configureren, hier vullen we onderhanden in in welke folder Obsidian terecht komt en de database gegevens. Eenmaal compleet hebben we een war in onze opgegeven folder gekregen, deze deployen we in onze Tomcat. Als we de default gegevens hebben laten staan kunnen we nu op poort 8080 van onze machine aan de UI van onze obsidian. We kunnen inloggen met de gebruiker admin en paswoord changeme.

Onze installatie is voorzien van enkele standaard jobs maar het is mogelijk om via de Java api zelf jobs te coderen. De api voorziet de interface InterruptableJob die een klasse dwingt om de execute methode te implementeren, deze heeft een JobContext als parameter. Obsidian scant de lib folder van Tomcat en pikt de klassen die deze interface implementeren op als jobs. Om vanuit de UI parameters te kunnen meegeven moeten we onze klasse ook voorzien met de Configuration annotatie waarin we array van @Parameter aan meegeven. Elke parameter geven we een naam, een type, of deze vereist is en een eventuele default waarde. Tijdens het runnen van onze job kan je de parameterwaarde uit de JobContext halen.

Voor dit project maken we twee jobs aan, één die alle categorien ophaalt en doorstuurt naar AEM en eentje die enkel de gewijzigde categorien ophaalt. Beide jobs hebben twee stappen waarvan de eerste het ophalen van de categorien is. De eerste job doet dit zonder parameter en krijgt alle categorien binnen, de tweede doet dit met een datum en krijgt enkele de categorien die na deze datum gewijzigd zijn binnen. De volgende stap is identiek voor de jobs, itereren over de categorien en deze één voor één doorsturen naar AEM, dit zal het pagina generatie proces starten.

6 Architectuur van AEM

In dit hoofdstuk gaan we de bouwstenen van AEM bekijken, de frameworks die worden gebruikt om de geboden functionaliteit mogelijk te maken en de interfaces die we ter beschikking krijgen om onze applicatie te beheren. De reden dat we dit doen is omdat deze ingebakken zijn in AEM, wie met AEM aan de slag gaat, zal ongetwijfeld met deze technologien te maken krijgen. Het is aan te raden eerst deze sectie te lezen om een beter begrip te krijgen wat we doen en waarom we dit doen. We beginnen bij het open-source gedeelte en bekijken vervolgens waarvoor we betalen.

6.1 OSGi

OSGi (Open Services Gateway initiative) is een framework dat ons in staat stelt om Java applicaties uit verschillende modules op te bouwen. Deze modules worden bundels genoemd die onafhankelijk van elkaar genstalleerd, verwijdert en vervangen kunnen worden in een OSGi container. Een bundel bestaat uit de jars en resources nodig voor de interne werking van deze bundel. Tijdens het bundelen kan er gespecificeerd worden welke packages zichtbaar zijn voor de andere bundels, als we niets configureren zal geen enkele package publiek beschikbaar zijn. Dit is in contrast met de normale werking van jars waar elke jar op het classpath aan alle (publieke) klassen kan. Wanneer we een bundel installeren (of verwijderen) hoeven we de container niet stop te zetten. Dit geeft dat er geen down time is tijdens het updaten van onze applicatie.

Omdat de bundels onafhankelijk van elkaar genstalleerd worden, kan een bundel niet rechtstreeks rekenen op klassen voorzien door een andere bundel. Indien bundels toch afhankelijk zijn van elkaar worden er interfaces voorzien die geregistreerd worden in een service laag. Stel dat bundel A een externe StoreService gebruikt om een winkel te kunnen ophalen, zolang deze interface geregistreerd is in de service laag kan onze bundel starten zonder probleem. Wanneer bundel B start met een implementatie van StoreService (bv. StoreServiceImpl), kunnen we deze registreren in de service laag. Bundel A luistert naar veranderingen in de service laag en zodra StoreServiceImpl beschikbaar is, zal deze in gebruik genomen worden. In het geval dat bundel B verwijdert word, zal deze ongeregistreerd worden en zal bundel A hiervan op de hoogte zijn.

6.2 Apache Felix

Apache Felix is de open source OSGi-container van Apache en wordt gebruikt door AEM voor het installeren van de bundels die onze componenten bevatten. In deze container zullen onze bundels draaien en met elkaar communiceren om

samen de applicatie van zijn functionaliteit te voorzien.

6.3 JCR en Apache Jackrabbit

De Java Content Repository API (JCR) is een api die gebruikt kan worden om data op te slaan in een boomstructuur. Deze boom bestaat uit twee zaken: nodes en properties, nodes vormen de toppen van onze boom waardoor we kunnen navigeren, de top van onze bomen heet de rootnode. De properties zijn de blaadjes van onze toppen die effectieve data bevatten zoals een stuk tekst of getallen. We kunnen door onze nodes navigeren aan de hand van de methodes die de api beschikbaar stelt, we kunnen nagaan of onze node kinderen heeft en hoe deze heten alsook onze ouder bekijken. De data structuur van onze nodes kunnen we definiren aan de hand van een type. Dit type kan gebruikt worden om bepaalde velden af te dwingen alsook restricties omtrent de toegestane velden op te leggen. Dit type kan ook indiceren dat alles is toegestaan.

In functie van een CMS applicatie kunnen we onze boom beschouwen als de data op onze pagina's. Onze rootnode is onze homepage en diens kinderen vormen onze navigatie, wanneer we naar de pagina merken navigeren bewegen we ons vanaf onze rootnode naar diens kind 'merken'. De kinderen van de node 'merken' stellen dan elk een effectief merk voor. Omdat de data ongestructureerd is hoeft niet elk merk dezelfde properties te bevatten, tijdens het generen van de pagina kunnen we zaken tonen, of juist niet, aan de hand van de aanwezigheid van bepaalde properties. Dit is een voorbeeld om aan te tonen hoe je met JCR een website kunt opbouwen.

Het is belangrijk om te begrijpen dat het niet de bedoeling is om traditionele data op te slagen in JCR, enkel content. Apache Jackrabbit is Apaches implementatie van JCR en is gentegreerd in Apache Sling.

6.4 Apache Sling

Sling is een framework van Apache dat de gebruiker in staat stelt om REST calls te doen via http. Het ondersteund verscheidene bestand types waaronder json, xml en html. Sling is het framework dat AEM gebruikt om aan de hand van een url van een request de juiste node te vinden en diens data terug te geven in de correcte vorm. Via een extensie mee te geven kunnen we aan Sling duidelijk maken hoe we onze data willen, bv. door .json achteraan een url toe te voegen, weet Sling dat we de data als json terug willen.

Sling is reeds geintegreerd met zowel Apaches Jackrabbit alsook Felix waardoor het de eigenschappen van beiden heeft. Dit wil niet zeggen dat JCR de enige manier is om met Sling data op te slaan, indien gewenst kan er ook met andere

databanken zoals SQL of MongoDB gewerkt worden.

Wanneer een Sling bundel genstalleerd word, registreert het een ResourceProviderFactory bij de Service laag van onze OSGi container.

6.5 Author

6.5.1 CRX

Nu we de open source onderdelen van AEM hebben gezien is het tijd om de gelicentieerde delen bespreken. Het eerste dat we bespreken is Adobe CRX wat, simpelweg gezegd, Adobes implementatie is van Apache Jackrabbit en Sling. Buiten alle features die deze frameworks bieden heeft Adobe bijkomende functionaliteit toegevoegd, anders zou het maar raar zijn om hiervoor te betalen. We bekijken kort diegene die we gebruiken tijdens dit project.

Zoals we reeds gezien hebben is de datastructuur van JCR een boom waarin we kunnen navigeren. Een feature die CRX voorziet is een UI waarmee we kunnen navigeren door onze nodes, de CRXDE. Met deze interface kunnen we onze nodes bekijken zoals we een filesystem zouden gebruiken. We kunnen nodes expanderen en hun kinderen bekijken om ook deze te expanderen en zo dieper in onze boom af te dalen, of we kunnen de properties van de node zelf bekijken. De kracht van deze UI is dat we een gestructureerd overzicht van de nodes zien die onze website vormgeven.

De package manager is een ander belangrijk onderdeel van de CRX en geeft ons de optie om onze OSGi bundels via een interface te managen. Met deze interface kunnen we bundels aan onze applicatie toevoegen of bestaande bundels verwijderen. Buiten beheren welke bundels genstalleerd zijn kunnen we deze ook starten en stoppen vanuit de package manager. Moest er een bundel niet starten, kunnen we deze opzoeken in de package manager en krijgen we de reden hiervoor te zien. Dit versimpelt de manier waarop we onze applicatie beheren en is dus van grote toegevoegde waarden voor de ontwikkelaars.

6.5.2 Editor

Alles wat we tot hiertoe gezien hebben heeft als functie het voorzien van componenten die bepaalde functionaliteit bezitten. Eenmaal we de componenten ontwikkeld hebben en via onze bundels genstalleerd zijn, is het tijd voor de designers aan de slag te gaan. De belangrijkste feature van AEM is de mogelijkheid om content te voorzien, gebruik makend van onze componenten. We kunnen deze slepen op templates, configureren en voorzien van inhoud. Een designer kan zo een volledige pagina, inclusief de navigatie hiernaar toe, opbouwen

zonder een enkele html tag of javascript regel te schrijven. Wanneer een pagina af is, kan men deze ook publiceren via AEM zodat het resultaat door de wereld gezien kan worden.

6.5.3 DAM

De DAM (Digital Asset Manager) is een opslag plaats waar we onze verschillende media kwijt kunnen zoals foto's of video's. Eenmaal opgeslagen in de DAM kunnen we deze gebruiken op onze website door middel van referentie, later zien we hoe dit praktisch in zijn werk gaat.

6.6 Dispatcher

De dispatcher is AEM's eigen caching laag en kan tevens dienen als loadbalancer. Doordat de publieke requests via de dispatcher gaan voorziet deze een extra beveiligingslaag. Dit komt omdat we AEM kunnen afschermen voor requests van buiten ons netwerk. Om een dispatcher op te zetten installeren we de module op een web server en configureren deze. Deze configuratie bevat, onder anderen, waar onze AEM draait, welke soort bestanden er gecached moeten worden en hoe de cache van deze bestanden vervalt.

Wanneer een dispatcher een request binnen krijgt wordt deze eerst geanalyseerd of deze in aanmerking komt voor onze cache. Buiten onze configuratie zijn er ook enkele standaard regels waarna gekeken wordt om dit te bepalen. Enkel de HTTP methode GET komt in aanmerking voor de cache, indien het request een andere methode bevat kan deze niet gecached worden. Ook wanneer er een request parameter wordt meegegeven kan de request niet gecached worden aangezien zo'n request een variabel antwoord heeft.

Er zijn twee manieren waarop een cache kan vervallen, de eerste zijnde met de auto-invalidate feature. Wanneer bestanden onder deze groep vallen, zal de dispatcher telkens de versie van zijn cache vergelijken met die van AEM, indien er een mismatch is zal de dispatcher opnieuw de request naar AEM doorsturen. Dit betekent dat wijzigingen gemaakt aan onze pagina's onmiddellijk zullen doorkomen op de live site.

Wanneer bovenstaande methode niet gewenst is, misschien willen we meer controle over het tijdstip dat pagina's geregenereerd worden, kunnen we de auto-invalidate functie uit laten. Als we nu onze cache willen invalideren zullen we hiervoor een call moeten doen naar de dispatcher. Wanneer deze binnenkomt vervalt de cache onmiddellijk en eerst volgend request krijgt de nieuwe content te zien.

7 Een eerste project

In de vorige hoofdstukken hebben we services voorzien die data op verscheidene manieren kunnen ophalen, het enige wat nog ontbreekt is een applicatie om deze te gebruiken. In dit deel gaan we onze eerste AEM applicatie bouwen en proberen we de verschillende manieren uit om deze applicatie van data te voorzien.

7.1 Een author lokaal draaien

Tijdens het ontwikkelen van een AEM-applicatie is het aangeraden om een lokale author te draaien. We kunnen deze omgeving gebruiken om onze geschreven componenten een eerste keer in actie te zien zonder deze op een remote server te moeten deployen. Voor het opstarten van een author heb je twee zaken nodig: de quickstart jar en een license.properties file, beiden te verkrijgen via de Adobe website. Indien we een geldige licentie hebben kunnen we author starten door simpelweg de jar te runnen.

Standaard start de author op poort 4502 van onze machine, om te verifiren dat het opstarten succesvol is verlopen surfen we naar http://localhost:4502/projects in onze browser. Dit opent de UI van de author waarbij we, onderhanden, kunnen bekijken welke sites er reeds zijn. Als je de jar zonder parameters hebt gestart zouden er al enkele voorbeelden gedefinieerd moeten zijn. Natuurlijk zijn we hier niet in geintreseed en gaan we zelf een website opzetten. De website, en bijhorende componenten, kan je rechtstreeks via deze UI aanmaken maar is niet aan te raden. De mogelijkheden zijn beperkt en er is geen spraken van versie beheer, voor ons project gaan we een andere aanpak gebruiken.

7.2 Een Maven project aanmaken

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
 -DarchetypeGroupId=com.adobe.granite.archetypes
 -DarchetypeArtifactId=aem-project-archetype
 -DarchetypeVersion=10
 -DarchetypeCatalog=https://repo.adobe.com/nexus/content/groups/public/
```

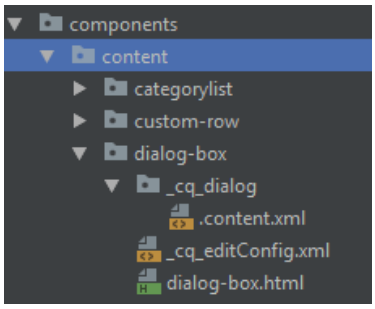
Het gebruiken van een Maven project heeft verscheidene voordelen: we kunnen rest endpoints voorzien die men kan gebruiken om data naartoe te sturen (pagina's generen), business logica ontwikkelen in de vorm van Java-klassen en we kunnen het project beheren via git waardoor we een historiek van aanpassingen hebben. Het handmatig aanmaken van zo'n project kan lastig zijn, gelukkig voor ons heeft Adobe een template project voorzien die we kunnen gebruiken. Wanneer we volgend commando uitvoeren (vereist Maven versie 3.3.1 of hoger) wordt ons gevraagd om enkele zaken te specificeren zoals de naam van onze website, de naam van onze modules en OSGi-bundels. Wanneer deze zijn ingevuld wordt het project aangemaakt en kunnen we aan de slag gaan.

7.3 Structuur van het Project

Wanneer we het project openen zien we dat de template voorzien is van meerdere modules die elk dezelfde prefix maar andere suffix hebben, we overlopen kort de belangrijksten. De core-module bevat onze Java-klassen: repositories,dto's, rest controllers, onze modellen, enz. De ui.apps-module bevat het frontend gedeelte: de html, CSS, JavaScript, templates, enz. De ui.content-module bevat onze effectieve website, wanneer we deze deployen worden alle pagina's vervangen door wat er zich in deze module bevindt. Als onze Maven profielen ongewijzigd zijn gebeurt dit samen met het deployen van de ui.apps-modulen (dus wanneer we onze componenten bundel uploaden). Vanzelfsprekend willen we niet bij elke deploy alle pagina's van onze website verwijderen dus splitsen we deze uit naar een ander profiel. Voor het coderen van componenten houden we ons voornamelijk bezig met de core en ui.apps-module.

7.4 Een component maken

Voor wie nog geen ervaring heeft met het aanmaken van componenten via een Maven project kan het moeilijk zijn om het doel van elk bestand te achterhalen, bijkomend is de documentatie schaars en vooral gericht op het werken met de author. In dit stuk gaan we een simpele component aanmaken en verklaren we wat waarvoor dient.



Een component bestaat uit verschillende files gedefinieerd onder een folder die de naam van de component draagt. Ter illustratie maken we een dialog-box component, als eerste maken we de folder aan. In deze folder definiren we een dialog-box.html wat de structuur zal beschrijven. Als basis gebruiken we een Bootstrap panel waarbij we een titel en een tekst kunnen invullen. Vervolgens voegen we een _cq_editConfig.xml wat ons toelaat te definiren hoe deze component gebruikt kan worden, enkele voorbeelden zijn: of deze via de author mag versleept worden, of de inhoud gewijzigd kan worden en op welke type pagina's deze component geplaatst kan worden. Als laatste hebben we een .content.xml, hierin kunnen we specificeren wat er aan de component kan meegegeven worden wanneer deze op een pagina wordt gesleept. In ons voorbeeld voorzien we drie zaken: een titel, een tekst en een grote.

We kunnen ook een Java-klasse voorzien waarop we de velden van de .content.xml kunnen injecteren. De Java klasse werkt als placeholder voor de velden

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0"
          xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
          jcr:primaryType="nt:unstructured"
          jcr:title="Dialog BOx Properties"
          sling:resourceType="cq/gui/components/authoring/dialog">
    <content
            jcr:primaryType="nt:unstructured"
            sling:resourceType="granite/ui/components/foundation/container">
        <layout
                jcr:primaryType="nt:unstructured"
                sling:resourceType="granite/ui/components/foundation/layouts/fixedcolumns"/>
        <items jcr:primaryType="nt:unstructured">
            <column
                    jcr:primaryType="nt:unstructured"
                    sling:resourceType="granite/ui/components/foundation/container">
                <items jcr:primaryType="nt:unstructured">
                    <title jcr:primaryType="nt:unstructured"
                           sling:resourceType="granite/ui/components/foundation/form/textfield"
                           fieldLabel="Title"
                           name="./title"/>
                    <text jcr:primaryType="nt:unstructured"
                           sling:resourceType="granite/ui/components/foundation/form/textfield"
                           fieldLabel="Text"
                           name="./text"/>
                    <size jcr:primaryType="nt:unstructured"
                           sling:resourceType="granite/ui/components/foundation/form/textfield"
                           fieldLabel="Size"
                           name="./size"/>
                </items>
            </column>
        </items>
    </content>
</jcr:root>
```

Figuur 6: .content.xml

van onze paginanode, als we een property ’¡custom-tag¿:title’ hebben voorzien we hiervoor een String en annoteren die met @Inject en @Named(’¡custom-tag¿:title’). Dit maakt het mogelijk om fallbacks in te bouwen alsook extra logica te voorzien, bv. een service gebruiken om data op te halen die we in onze html willen gebruiken.

Nu is het mogelijk om via Sightly onze klasse te koppelen aan onze html door het attribuut ’data-sly-use.dialogBox=”¡package naam¿.¡klasse naam¿” in een html tag toe te voegen. Het stuk na ’data-sly-use.’ definiert hoe we deze gaan aanspreken, in ons voorbeeld hebben we deze dialogBox genoemd. Nu kunnen we binnen de html tag onze klasse gebruiken via volgende syntax: $dialogBox.title, gegeven er een getTitle() methode op onze Java-klasse staat.

7.5 Pagina's genereren

7.5.1 Wat is het?

Als we pagina's gaan genereren, gaan we wijzigingen van het ERP doorduwen naar AEM. AEM gaat aan de hand van deze informatie noden aanmaken in onze JCR repository en per taalvariant een pagina voorzien. Eenmaal de pagina's aangemaakt zijn, kunnen ze via de corresponderende url worden opgehaald.

7.5.2 Hoe werkt het?

Om deze optie te voorzien moeten er enkele zaken gecodeerd worden in Java, met name de functionaliteit om via de SlingRepository de data weg te schrijven naar een node en de PageManager van AEM gebruiken om de pagina's aan te maken. Alsook moeten we een HTML template voorzien die we kunnen gebruiken om onze categorie weer te geven. Tijdens het genereren van de pagina kunnen we verwijzen naar deze template om een standaard look te geven aan de pagina. Eenmaal deze pagina er is kan het contentteam aan de slag om deze een persoonlijke touch te geven.

Om een node aan te maken moeten we eerst een Session verkrijgen waarin we kunnen werken, dit kunnen we doen door de login methode op onze SlingRepository aan te roepen met onze credentials. JcrUtils is een voorziene klasse waarmee we noden kunnen aanmaken, hetgeen we nodig hebben is een pad waaronder we deze willen opslaan, het type dat we aan de node willen toekennen en onze reeds verkregen sessie. Als we de node hebben aangemaakt rest ons nog het over mappen van de velden naar onze noden, indien ons model velden bevat die een klasse op zich zijn, moeten we met geneste noden werken. Dit nesten heeft geen beperkingen en kan zo diep als nodig gaan. Het is aan te raden om deze velden te voorzien met een prefix om een duidelijk onderscheid te maken dus de properties die we zelf creeren en diegene die door JCR aangemaakt zijn. Stel dat we een veld 'title' hebben die we wensen op te slaan, doen we dit onder '¡custom-tag¿:title'. Wanneer de velden zijn overgezet mogen we niet vergeten om de save methode van onze sessie aan te roepen zodat onze data gepersisteerd wordt.

De volgende stap is een pagina voor ons model aanmaken, dit per taal waarin onze website beschikbaar is. Een pagina aanmaken gebeurt op een gelijkaardige manier als een node. We gebruiken de PageManager die de AEM api ons beschikbaar stelt. Omdat we zaken gaan persisteren moeten we ook hier werken met een sessie. Via de pageManager kunnen we onze root pagina van de website opvragen. De directe kinderen van deze pagina stellen de talen voor waarin onze site beschikbaar is. We itereren over deze lijst waarbij we per kind (lees taal) een pagina genereren. We maken deze pagina niet rechtstreeks onder de

taalpagina aan maar voorzien een tussen pagina, bv. categorien. Wanneer we een categorie met id 14 hebben gaan we de pagina hiervoor opslaan onder ¡onze site¿/nl/categorieen/14. Het pad dat we dan meegeven aan de create methode van de pageManager is het pad van de ouder waaronder we een pagina willen maken. Naast deze parameter geven we ook de pagina naam mee (in ons voorbeeld 14), de template (de locatie van de html die we gebruiken om onze pagina te tonen) en een modeltitel (bv. 'category'). Nu de pagina is aangemaakt kunnen we hiervan de node opvragen en kunnen we, conform aan vorige paragraaf, deze voorzien van properties. Ook hier prefixen we best onze namen.

In vorige paragraaf gebruiken we een template om onze categorie te tonen, deze gaan we zelf voorzien. Het is best om hier gebruik te maken van het componenten systeem waarvoor AEM gekend is. In plaats van een specifieke html te coderen gaan we deze opslitsen in herbruikbare componenten. Hoe we componenten bouwen hebben we reeds gezien in een vorig hoofdstuk.

Eenmaal we de nodige componenten hebben aangemaakt kunnen deze gebruikt worden om een template te voorzien. We kunnen de componenten rechtstreeks aan onze template toevoegen of een tussen component voorzien die kleinere componenten combineert. Dit kan handig zijn wanneer de zelfde combinatie van componenten meermaals gebruikt wordt (denk maar aan een navigatie), ook de css en js bestanden kunnen we in deze wrapper inladen zodat we ons daarover geen zorgen moeten maken wanneer we templates samenstellen. Eenmaal de template af is kan deze gebruikt worden om automatisch pagina's te genereren of om via de author handmatig een pagina toe te voegen.

7.5.3 De voordelen

Het voordeel van deze manier van werken is dat wanneer een gebruiker de pagina opvraagt, deze reeds voorzien is van data waardoor er geen repositories worden aangesproken. Uiteraard is dit wel het geval als we componenten toevoegen die dynamische content voorzien maar de properties die we aan onze node hebben toegevoegd zijn reeds aanwezig

Een bijkomend voordeel is dat, door het feit dat er een pagina aangemaakt is, we deze via de author kunnen bewerken. We maken weliswaar een pagina via een standaard template maar hier hoeft het niet bij te blijven. We kunnen deze via de author gaan editeren en van specifieke content zoals een achtergrond afbeelding, een paragraaf met extra informatie of een uitgelicht product voorzien. Pagina's genereren is de enigste manier van werken die dit mogelijk maakt.

7.5.4 De nadelen

Het grote nadeel van deze methode is dat dit een vrij intensieve operatie is en het genereren zelden beperkt is tot n enkele pagina, wanneer een site drie talen heeft zal AEM telkens drie pagina's aanmaken. Dit wordt nog eens gexpandeerd indien men via een blueprint werkt. Stel dat we een .com site hebben met drie talen, een .be site met drie talen en een .nl site met twee talen. Dit betekent dat per categorie AEM zeven pagina's moeten voorzien. Voor een model waar relatief weinig wijzigingen aan gebeurt is dit doenbaar, zoals ons categorie voorbeeld. Maar als we dit willen doortrekken naar producten, waar er mogelijks duizenden per dag wijzigen, is dit niet houdbaar. Onze author zou continue belast worden met het genereren van pagina's en potentieel bezwijken onder de load. Een tweede author plaatsen is zelden een optie omdat dit een tweede licentie vereist.

7.6 Server Side Includes

7.6.1 Wat is het?

SSi is een scriptingstaal die de mogelijkheid biedt om data in een bestaande html-pagina te injecteren. Deze injectie kan een effectief bestand betreffen maar kan evengoed een functie zijn, bv. de huidige datum ophalen. Vlak voor het versturen van de html zal de webserver alle SSI commando's opsporen en omzetten naar gewone html. De bestandtypes die men via SSI ophaalt kan verschillende vormen aannemen waaronder Json (bv. het resultaat van een rest call) en html (bv. een footer die genjecteerd wordt). In het geval van Json zou men het resultaat kunnen gebruiken voor een javascript functie, bij html kan deze rechtstreeks worden afgebeeld. Best practice is om de SSI commando's in commentaar te zetten zodat deze verdwijnen wanneer de webserver SSI niet ondersteund.

7.6.2 Waarom SSI?

Binnen AEM kunnen we templates bouwen aan de hand van bestaande componenten, een goed voorbeeld is een template die een navigatie bevat, we willen immers op de meeste pagina's onze navigatie tonen. Wanneer deze template gebruikt wordt genereren we een pagina met daarop onze navigatie en een gespecifieerde inhoud. Als een gebruiker deze pagina ophaalt passeert dit verzoek de dispatcher, als de pagina nog in zijn cache zit kan deze onmiddelijk worden weergegeven. Indien dit niet het geval is moet deze opgehaalt worden bij een publisher, vervolgens wordt deze op de dispatcher gecached en doorgestuurd naar de gebruiker. Dit betekent dat de hele pagina, inclusief navigatie, gecached wordt op de dispatcher.

Zolang de navigatie ongewijzigd blijft brengt dit geen problemen te weeg. Stel nu dat er een nieuwe categorie aangemaakt wordt en deze zichtbaar moet worden in de navigatie. We hebben in het vorige hoofdstuk gezien hoe we categorie pagina's kunnen maken en hoe de navigatie deze oppikt. Het enige obstakel om deze live te krijgen is de dispatcher die de navigatie op elke pagina apart heeft gecached. We kunnen wachten tot deze verloopt maar dit tijdstip is voor elke pagina anders. Langzaam aan zullen de gecachte pagina's verlopen waardoor deze opnieuw moeten worden opgehaald (met de nieuwe navigatie). Tijdens deze overgangsperiode zal de navigatie op onze website inconsistent zijn, sommige pagina's hebben de oude (gecachte) versie en andere hebben reeds de nieuwe.

Sommige lezers zullen nu denken: ẅe kunnen toch de hele cache tegelijk invalideren? Dan moet de dispatcher wel de nieuwe versie ophalen.¨ Deze gedachtegang is correct maar heeft énén groot nadeel: de publisher moet dan mogelijk enkele honderden pagina's tegelijk genereren wat kan leiden tot een crash. Buiten piekuren en met een beperkt aantal pagina's kan deze methode haalbaar zijn, voor een internationale enterprise applicatie is dit niet realistisch en moet er een andere oplossing gezocht worden.

Deze oplossing kan men bekomen met het gebruik van SSI, in plaats van onze template rechtstreeks van de navigatie component te voorzien gebruiken we SSI om deze toe te voegen. Als men nu een pagina opvraagt zal de dispatcher deze laten genereren door de publisher maar deze zal de SSI-tag nog niet resolven. Deze tag wordt mee gecached, met de rest van de pagina, en zal pas op het laatste moment ervoor zorgen dat de navigatie wordt genjecteerd. Wanneer deze injecte moet gebeuren zal de dispatcher de navigatie opvragen bij de publisher en deze cachen. Alle pagina's gaan nu de gecachte versie van onze navigatie injecteren en als deze wijzigt volstaat het om enkel de cache van de navigatie te invalideren.

Het is belangrijk om te bezeffen dat SSI enkel op de dispatcher zal worden uitgevoerd wat geen probleem vormt in productie, elk request passeert hier. Voor het contentteam vormt dit wel een probleem, zij werken op de author en aangezien deze de injectie niet gaat uitvoeren zullen zij de navigatie niet te zien krijgen tijdens het editeren van pagina's. Een mogelijkheid is om overal waar men SSI wil gebruiken deze tag in een IF-statement te incapsuleren, indien de pagina niet vanop de author wordt opgevraagt gebruikt men de tag, anders injecteert men de data rechtstreeks.

7.6.3 De nadelen

Eén nadeel hebben we reeds gezien, om het editeren van pagina's aangenaam te houden moeten we overal waar we SSI willen gebruiken extra statements voorzien om hetzelfde resultaat op de author te bekomen. Bij uitbundig gebruik kan dit leiden tot onoverzichtelijke code die moeilijk te onderhouden is.

Een ander nadeel is dat de commando's worden uitgevoerd vlak voor de webserver de html verstuurd. Dit heeft als gevolg dat de gebruiker moet wachten tot alle commando's zijn uitgevoerd voor hij de pagina te zien krijgt. Als we een andere html injecteren zal deze wachttijd nihil zijn. Wanneer men meerdere rest calls wil doen kan, afhankelijk van de duur van deze calls, de wachttijd oplopen. In dit geval moet men de overweging maken om over te stappen op javascript waar de calls worden uitgevoerd nadat de pagina is geladen.