Examensarbete, Högskolan på Åland, Utbildningsprogrammet för informationsteknik

UPPGRADERING AV JBOSS EAP & JBOSS AMQ

Anton Jansson & Conny Ljunggren

<48:2017>

Datum för publicering: 22.12.2017

Handledare: Agneta Eriksson-Granskog

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Anton Jansson & Conny Ljunggren
Arbetets namn:	Uppgradering av JBoss EAP and JBoss AMQ
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt:

I detta examensarbete skriver vi om hur vi uppgraderade JBoss EAP och JBoss AMQ för IT-företaget Crosskey.

Företaget ville ha sin redan befintliga version av JBoss och uppgradera den till den nya versionen.

Resultatet av examensarbetet blev att Crosskey fick sin version av JBoss EAP och AMQ uppgraderade och i och med uppgraderingen en version som är snabbare och säkrare.

I uppsatsen beskrivs konfigurering, utbyte av subsystem samt de olika teknikerna som användes.

Nyckelord (sökord):

JBoss, EAP, AMQ, Ansible

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
48:2017	1458-1531	Svenska	21

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
22.12.2017	01.12.2017	22.12.2017

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	Anton Jansson & Conny Ljunggren
Title:	Upgrade of JBoss EAP and JBoss AMQ
Academic Supervisor:	Agneta Eriksson-Granskog
Technical Supervisor:	Crosskey Banking Solutions

Abstract:

In this thesis we write about how we upgraded JBoss EAP and JBoss AMQ for the IT company Crosskey.

The company wanted their version of JBoss upgraded to the latest version.

The result of the thesis was that Crosskey got their version of JBoss EAP and AMQ upgraded and, with the upgrade, a version that is faster and safer.

The essay describes the configuration, subsystem replacements and the different techniques used.

Key words:

JBoss, EAP, AMQ, Ansible

Serial number:	ISSN:	Language:	Number of pages:
48:2017	1458-1531	Swedish	21

Handed in:	Date of presentation:	Approved on:
22.12.2017	01.12.2017	22.12.2017

1 INTRODUKTION

1.1 Syfte

Syftet med detta examensarbetet var att Crosskey ville ta sin redan existerande version av JBoss EAP 6 och uppgradera den till den senaste JBoss EAP 7 för sina applikations-serverar.

I och med denna uppgradering möjliggörs viss funktionalitet som inte har funnits tillgänglig i den tidigare versionen, som t.ex. *Offline CLI*. Det gör det möjligt att konfigurera applikationsservrar i ett offline-läge innan de ansluts. Orsaken till att man vill konfigurera servrarna innan dom går online är för att slippa stänga ner instanserna mellan varje konfiguration och på så vis spara en del tid.

För *message brokers* ville de migrera från JBoss EAP 6 till JBoss AMQ 7. Detta medför en avskalad *message broker* som bör vara stabilare, säkrare och enklare att konfigurera.

1.2 Metod

Vi började med att tillämpa uppgraderingen av JBoss och funktionaliteten som tillkommer för en kund. Vartefter kommer vi att tillämpa tekniken för resterande kunder.

Kunskapen som krävs för att utföra detta arbete kommer i huvudsak från Högskolan på Åland. De tekniker som vi inte har bekantat oss med under utbildningen har vi i huvudsak att lärt oss från nätet och kontinuerlig kontakt med handledare på plats.

1.3 Avgränsningar

Det här arbetet kommer att fokusera på JBoss EAP version 6, 7 och AMQ version 7 och vi går inte in på andra versioner. I början var det planerat att vi skulle skapa en generisk mall av Jboss-installationen som var färdigkonfigurerad med de inställningar som alla instanser behöver. Men vi kom senare fram till att de inställningarna var väldigt få och majoriteten av dem var instansspecifika och att det inte skulle vara värt att lägga tid på det.

1.4 Uppdragsgivare

Crosskey Banking Solutions är ett dotterbolag till Ålandsbanken som har expertis inom banktjänster som eBanking, kort och mobilbetalningar samt kapitalmarknad. Crosskey har kontor i Mariehamn, Stockholm, Helsingfors och Åbo med 230 st heltidsanställda och omsätter ca 32 miljoner euro (Crosskey, 2017).

1.5 Applikationsserver

En applikationsserver är en server som ofta körs av en specifik programvara och har ett eller flera syften. Några av dess syften kan t.ex. vara för att en applikation är så pass krävande att den behöver köras på en server. Det kan även vara av säkerhetsskäl man väljer att hålla applikationen isolerad eller att man vill isolera applikationen från verksamheten för att hålla verksamheten skyddad, men generellt används det för transaktionsbaserade applikationer (Ipeer, 2017).

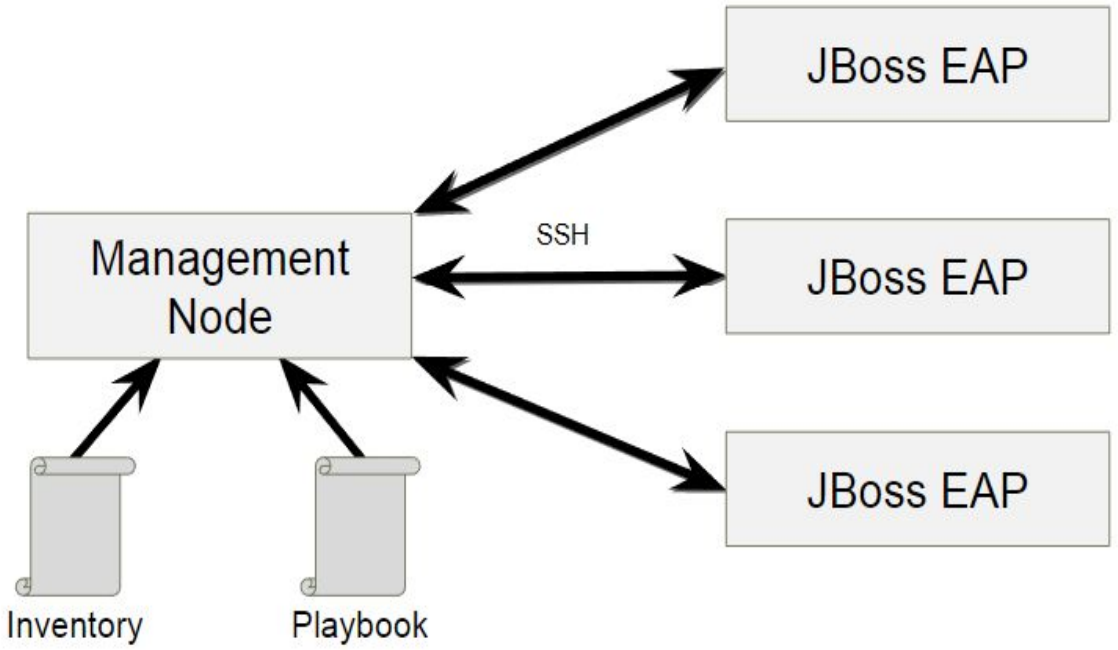
1.6 Message Broker

En *message broker* är ett system för distribution av meddelanden mellan klienter, i vårt fall applikationsservrar. En *message broker* tar emot och placerar meddelanden för upphämtning beroende på konfiguration.

2 TEKNIKER

2.1 Ansible

Ansible är ett system som används bland annat för konfigurationshantering av program och applikationsdriftsättning. Huvudkomponenten i Ansible är så kallade *playbooks*. En *playbook* fungerar som en instruktionsbok och innehåller en eller flera steg där det definieras vad som ska konfigureras för vilken server. På så vis kan man konfigurera och hantera fler servrar på ett smidigt sätt. (Ansible Documentation, 2017b). *Inventory* är en konfigurationsfil som innehåller IP-adresser eller värdnamn till de serverar som ska konfigureras. I Figur 1 ser vi en *management node*. Den här noden är den som konfigurerar och kontrollerar de applikationsservrar som är angivna i *inventory*-filen över ssh (Ansible Documentation, 2017a).



Figur 1. Ansible konfiguration av JBoss EAP applikationsservrar

2.2 Jinja2

Jinja2 är en mallmotor för Python och Ansible. Jinja gör det möjligt att dynamiskt tillgå variabler och användning av uttryck. I Figur 2 kan vi se ett exempel på hur användning av variabler i Jinja kan se ut (Ansible Documentation, 2017c).

```
<html>
    <title>{% block title %}My Website{% endblock %}</title>
    <body>
        {% with %}
            {% set variable = "value" %}
            {% block body %}
            {% endblock %}
        {% endwith %}
    </body>
</html>
```

Figur 2. Jinja2 variabel exempel (SlideShare, 2017)

3 JBOSS EAP

Jboss EAP (Enterprise Application Platform) är en av industrins populäraste applikationsservrar som används för att skapa, distribuera Java-applikationer och tjänster (Tridens 2017). Applikationsservern är *open-source* vilket betyder att källkoden är öppen för att återanvända, modifiera och vidaredistribuera. JBoss EAP är baseras på Java EE mjukvarustandarder som är en samling Java-API:er skapade av Oracle.

EAP integrerar WildFly applikationsserver och är en del av JBoss Enterprise Middleware portfolio vilket innebär att den är prenumerationsbaserad och avgiftsbelagd om man vill använda den till något annat än utveckling. WildFly är dock *open source* och helt gratis att använda (Wikipedia, 2017a).

Det finns två olika driftlägen att använda sig av för EAP-instanser; fristående och domän. Fristående eller *standalone* betyder att man kör en instans av servern. Domän eller *domain* är motsatsen till fristående och betyder att man har flera instanser i en eller flera grupper (Dzone, 2017).

3.1 Uppgradering från JBoss EAP 6 till EAP 7

En stor del av arbetet bestod av att ersätta de gamla subsystemen med dem som kom i den nya versionen av JBoss samt att konfigurera dessa efter Crosskeys behov.

Vi började med att lägga upp en okonfigurerad JBoss-installation av nya versionen på en filserver som vi sedan hämtade på liknande sätt som tidigare. Sedan påbörjade vi processen med att slå ihop konfigurationsfilerna och för att lättare kunna konfigurera instansen i offline-läge. Samtidigt tog vi bort konfigurationer för de avvecklade systemen från EAP 6 och lade till konfiguration för de nya.

3.2 Subsystem & CLI

Ett subsystem fungerar som en tjänsteleverantör som utför en eller flera funktioner. Subsystemet gör inget av sig självt utan lyssnar efter ändringar och exekverar när den blir efterfrågad. (IBM, 2017)

För att konfigurera systemen använder man sig av CLI eller XML. När man konfigurerar systemen med CLI sker ändringarna i realtid och med XML måste ändringarna ske före körning eller så krävs det en omstart för att ändringarna ska börja gälla. CLI står för *Command Line Interface* och är ett textgränssnitt i terminalmiljö och används för att interagera med ett datorprogram. (Wikipedia, 2017b).

Med den nya versionen av JBoss är det möjligt att göra dessa konfigurationer i ett CLI som körs i ett offline-läge före man sätter igång av servrarna. På så vis går det att konfigurera dem utan att utföra en omstart som vissa konfigurationer i den tidigare versionen krävde. Det är tack vare den här nya funktionalitet som vi vinner mest tid. I Figur 3 ser vi ett exempel på subsystemet Undertow där vi kör kommandot read-resource och kollar vad den nuvarande konfigurationens inställningar har för värden. Undertow är en webbserver som ersatte det gamla systemet Web i nya versionen.

```
[standalone@localhost:3190 /] /subsystem=undertow/server=default-server/host=default-host:read-resource
{
    "outcome" => "success",
    "result" => {
        "alias" => ["localhost"],
        "default-response-code" => 404,
        "default-web-module" => "ROOT.war",
        "disable-console-redirect" => false,
        "filter-ref" => {
            "server-header" => undefined,
            "x-powered-by-header" => undefined
        },
        "location" => undefined,
        "setting" => {"access-log" => undefined}
    }
}
[standalone@localhost:3190 /]
```

Figur 3. Konfiguration av subsystemet Undertow

4 JBOSS AMQ

JBoss AMQ är en fristående *message broker* och består endast av system för meddelandehantering, d.v.s. den integrerar endast JMS- delen av Java EE.

Vi kommer att gå mer in på JMS i avsnittet 4.1. AMQ har jämfört med EAP snabba uppstartstider och låg resursanvändning. Gällande konfiguration är det också väldigt låg komplexitet på en fristående *message broker* eftersom man inte behöver beakta de system som följer med EAP som inte används.

Tillämpningen av JMS i JBoss AMQ 7 och JBoss EAP 7 heter ActiveMQ Artemis och är en vidareutveckling av HornetQ som är grunden för *message broker*-systemet i JBoss EAP 6. Artemis och HornetQ är p.g.a. detta väldigt lika till konfiguration och beteende vilket har underlättat i migrationen. Följande kapitel använder Red Hat JBoss EAP 7 Configuration Guide som källa.

4.1 Java Message Service

Java Message Service, JMS, är ett API i Java EE som används för att hantera meddelanden mellan olika applikationer. Strukturen för JMS lyder att en leverantör fungerar som ett nav för klienter. I vårt fall är *message broker*-instansen leverantören och applikations-servrarna är klienter. Klienterna skickar och tar emot meddelanden via leverantören, vilket kan ske både synkront eller asynkront.

JMS stöder två modeller för meddelandehantering, köer och prenumerationer. Vi beaktar endast köer här eftersom det är vad vi använder. Kö-modellen använder sig av buffrar, JMS-meddelanden samlade baserat på adresser, för att organisera och vidareförmedla meddelanden. Trots namnet så använder köerna inte någon specifik ordning, meddelanden kan t.ex. plockas från mitten av en kö. JMS garanterar också att meddelanden i köer bara tas emot av en enda mottagare. Klienterna kan skicka meddelanden till köer via en adress. Ifall de bara är intresserade av en viss del av meddelandena i en kö kan de använda en *selector* för att få rätt sortering. Meddelanden kan via en timer sättas i en kö för utgångna meddelanden om ingen klient har plockat upp det inom den angivna tiden.

Vi använder oss av asynkron hantering, d.v.s. klienten som skall ta emot ett visst meddelande behöver inte vara närvarande när meddelandet sänds, utan det kommer då att vänta i kön tills det plockas upp (Wikipedia, 2017c).

4.2 Hög tillgänglighet

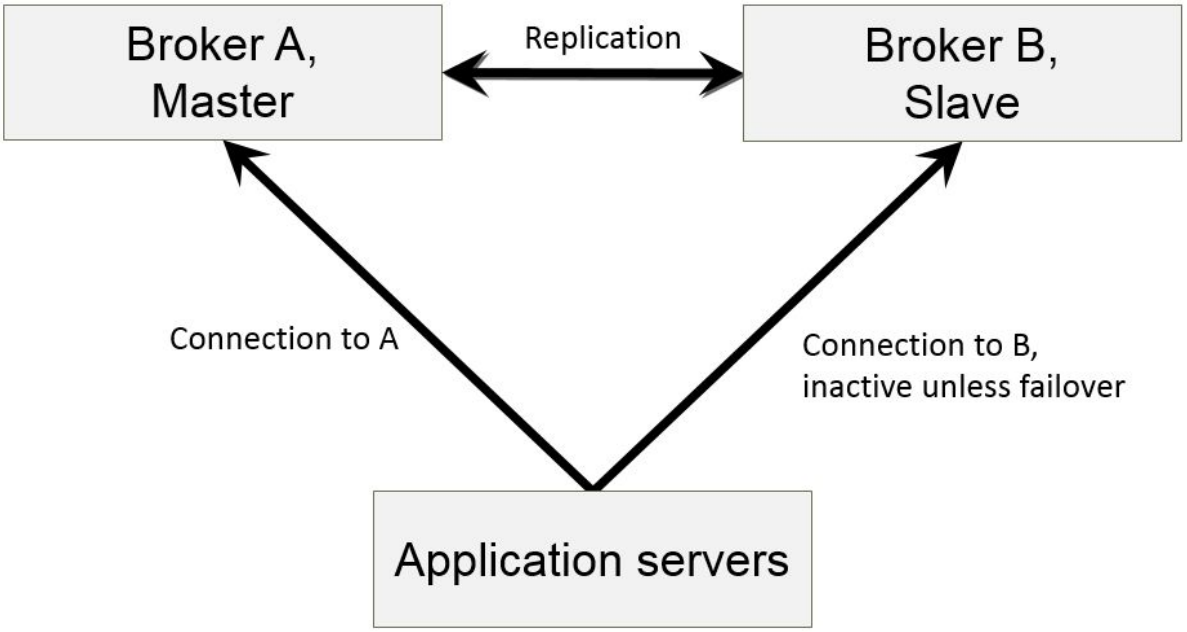
För att kunna hantera fall där en *message broker* inte är aktiv använder vi en modell där vi har två stycken, där Broker A agerar *master* och Broker B agerar *slave*.

En master fungerar som en huvud-broker och har hand om all meddelandehantering så länge den är aktiv. En *slave* är en reserv och vid behov, dvs när huvud-broker instansen inte är aktiv, så tar den över meddelandehanteringen.

I Figur 4 kan vi se Broker A som *master*, applikations-servrarna kommer då alla att vara kopplade till den och använda dess resurser för meddelandehanteringen. Broker B, som är *slave* i det här läget, kommer då endast att kontrollera att Broker A är aktiv. Ifall Broker A av någon anledning skulle bli inaktiv kan Broker B ta över rollen som *master*, detta kallas *failover*. Broker B kommer då att ta över alla resurser Broker A använder för meddelandehanteringen, d.v.s. *acceptor* och köer. Alla applikationsservrar kommer också att byta över till B-kopplingen.

Vi använder oss också av *failback* vilket betyder att om Broker A blir aktiv efter att ha varit inaktiv så kommer den leta igenom klustret efter broker-instansen som har tagit över dess roll och ta tillbaka sin roll som *master*.

Dessa broker-instanser ligger på olika värdar för att vidare säkerställa att de inte blir inaktiva på samma gång. När man har dem på olika värdar så måste kopiering ske av köerna för meddelanden från *master*- till *slave*-instansen. Ifall *failback* sker kopieras det andra vägen. Detta visas i Figur 4 med *replication*-relationen.



Figur 4. Message broker master-/slave-konfiguration.

4.3 Message Brokers i Kluster

Eftersom vi använder två olika värdar där vi vill ha en *master-slave*-uppsättning så behöver vi använda oss av ett kluster för att koppla samman dessa två *message brokers*. Ett kluster här är en samling *message brokers* som känner till varandra och kan koppla sig mot varandra.

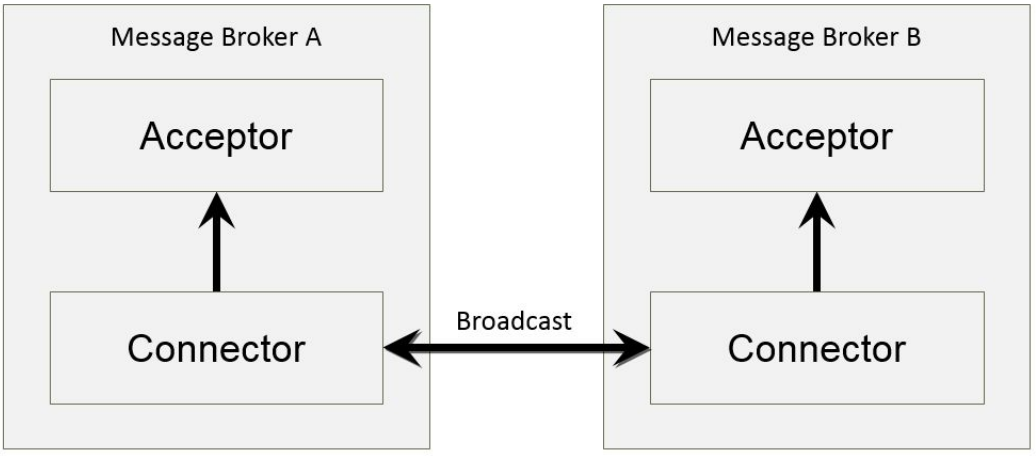
För att koppla sig till en *message broker* använder både *message brokers* och applikationsservrar *connectors*, som innehåller informationen som behövs för att hitta målet. I Figur 5 kan man se att vår *connector* har en ip, port, protokoll och SSL information.

En *acceptor* är vad *message brokers* använder för att ta emot anslutningar, d.v.s. vad en *connector* ansluter sig mot. Enda skillnaden här på informationen i vår *connector* och *acceptor* är SSL-informationen.

```
<acceptors>
    <acceptor name="artemis">
        tcp://0.0.0.0:{{brokerHostPort}}?
        protocols=CORE;
        sslEnabled=true;
        keyStorePath={{instance_home_dir}}/server.jks;
        keyStorePassword={{keystore_password}}
    </acceptor>
</acceptors>
<connectors>
    <connector name="opposite-broker-connector">
        tcp://{{jboss_instance.jmsBrokerOppositeBrokerHost}}:{{brokerHostPort}}?
        sslEnabled=true;
        trustStorePath={{instance_home_dir}}/cacerts;
        trustStorePassword={{keystore_password}}
    </connector>
</connectors>
```

Figur 5. Exempel på en acceptor och en connector för JBoss AMQ 7

För att bygga upp klustret använder vi *static discovery*, vilket betyder att för varje broker-instans anger man en lista med värdnamn och port till *connectors* för alla andra brokers. Varje broker har också en *connector* till sin egen *acceptor* som den kommer att dela med sig av till klustret. Det är via den som sedan andra brokers kan ansluta sig. Som vi kan se i figur 6.



Figur 6. Ett kluster med två message brokers

4.4 Uppgradering från JBoss EAP 6 till JBoss AMQ 7

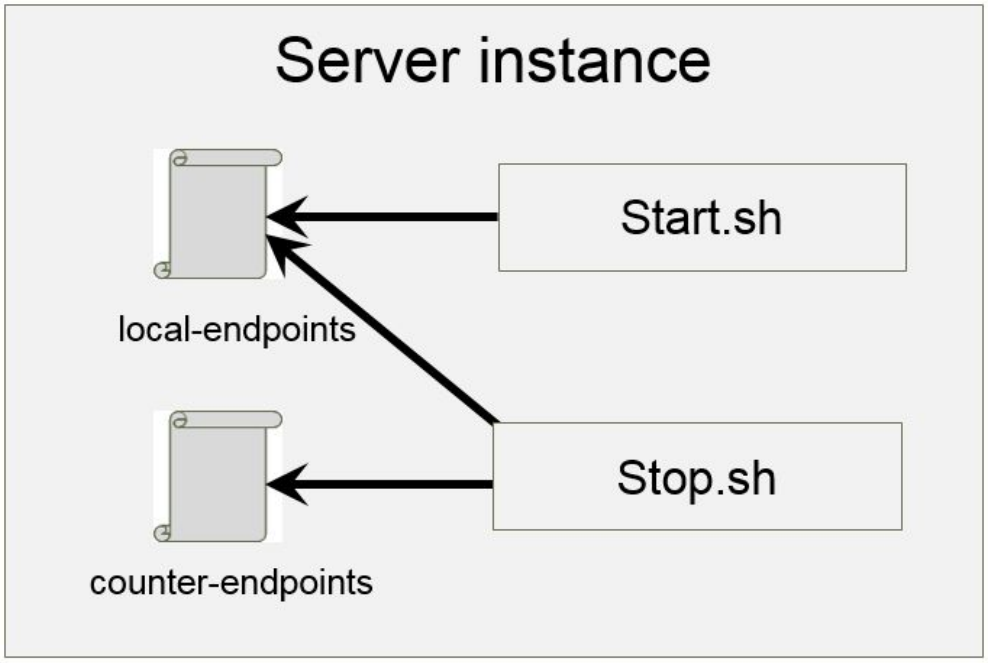
Migrationen bestod främst av att byta ut CLI-konfigurationen i EAP till XML-konfiguration i AMQ. Vissa detaljer är ändrade som t.ex. protokollet som vi kan se i Figur 5 *acceptor*-konfiguration, vilket det inte fanns några valmöjligheter för i EAP 6. Den största delen av uppgraderingen var dock bara att kartlägga och föra över den äldre konfigurationen.

5 SERVER OCH APPLIKATIONSHANTERING

I samband med uppgraderingen hade vår uppdragsgivare ett önskemål om att vi skulle lyfta ut funktionaliteten för hanteringen för starta och stoppa applikationer och server-instanser. Funktionaliteten för detta fanns i Ansible och genom att lyfta ut det till bash-skript och sedan låta Ansible kalla på dessa skript får man en lösare koppling och möjliggör manuell körning av skripten.

Det fanns redan ett API för HTTP-förfrågningar som kontrollera applikationer med kommandon såsom starta och stoppa samt kontrollera status för applikationerna som vi kunde använda oss av.

Vi började med att skriva de applikationers URL som var aktuella för den specifika instansen till en fil. Dem läste vi sedan in rad för rad i bash-skriptet och utförde antingen en status check, uppstart eller avstängning. Tidigare när man behövde stänga ner applikationerna av olika anledningar var man tvungen att gå in manuellt och stänga ner applikationer eller tvinga en avstängning av instansen.

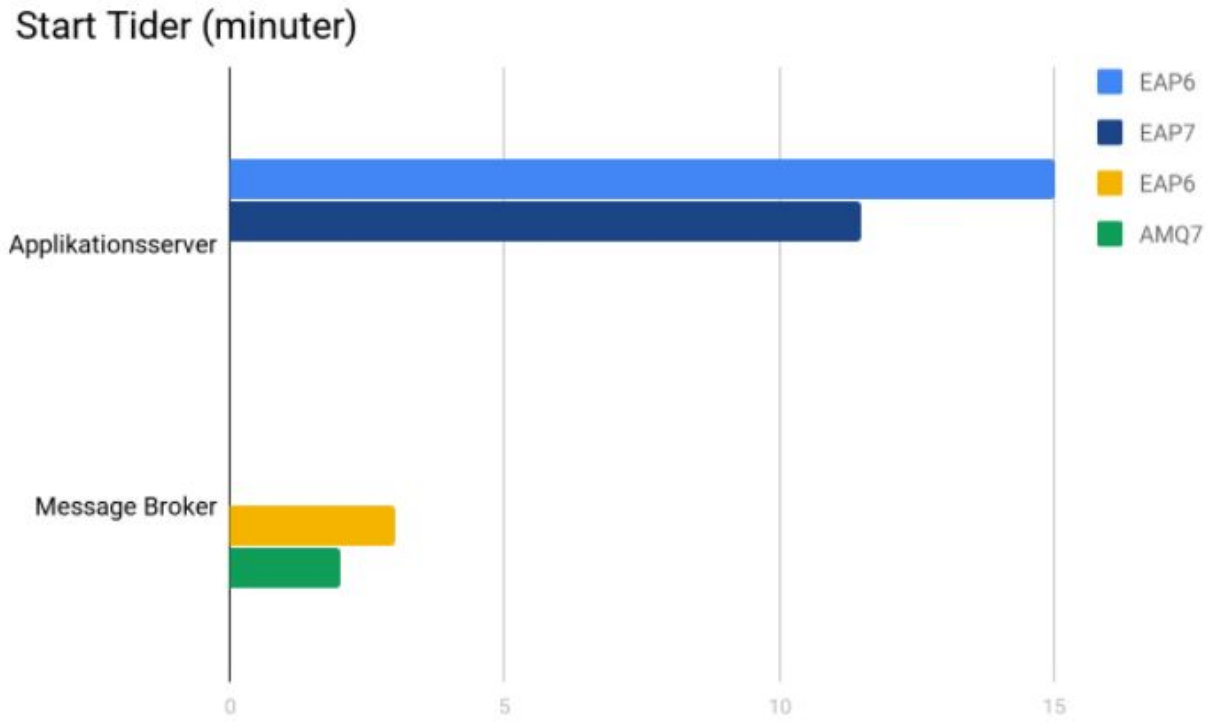


Figur 7. Relationer mellan skripten och applikationerna

6 RESULTAT

Vi mätte tiden för uppstart av fem instanser av EAP 6 och jämförde det mot EAP 7. Vi valde fem instanser eftersom det är vad vi använder i vår server-uppsättning. Vi startade heller inga applikationer i dessa instanser, de är alltså tomma och tidsmätningarna reflekterar endast kopiering och konfiguration av EAP. Starttiden för EAP 6 var ca 15 minuter och för EAP 7 ca 11,5 minuter. Den största skillnaden i tid här kommer från Offline CLI-konfigurationen i EAP 7. Start av applikationer bör inte ge någon betydande tidsskillnad.

För *message brokers* jämförde vi en EAP 6-instans mot en AMQ 7-instans. Här tog EAP 6 ca 3 minuter och AMQ 7 ca 2 minuter. Som vi kan se i figur 8.



Figur 8. Tidsmätning för gamla versionen till nya

7 SLUTSATSER

Under projektets gång jobbade vi med att försöka få JBoss EAP 6 applikationsservrar uppgraderade till versionen EAP 7 till vår arbetsgivare Crosskey. Vi jobbade också med att byta ut JBoss EAP 6 *message broker* till en *standalone broker* AMQ 7. Vi har jobbat med tekniker som var delvis främmande för oss så som t.ex. Ansible, bash-skript och applikationsservrar.

Trots att vi låg under en snäv tidsram lyckades vi att få både applikationsservrar och *message broker* uppgraderade till den senaste versionen för en kund. Det fanns inte tid att verkställa ändringarna för resterande kunder inom examensarbetets tidsram men det är något vi kommer att jobba vidare med efter projektet.

8 KÄLLFÖRTECKNING

Crosskey (2017) December 2017 https://www.crosskey.fi

IBM (2017) What is a subsystem? December 2017

https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieaf200/ieaf200_What_is_a_Subsystem_.htm

Wikipedia (2017b) Command line interface November 2017

https://en.wikipedia.org/wiki/Command-line_interface

SearchIToperations (2017) Ansible playbook December 2017

http://searchitoperations.techtarget.com/definition/Ansible-playbook

Wikipedia (2017) Ansible December 2017

https://en.wikipedia.org/wiki/Ansible_(software)

SlideShare (2017) Intro to Jinja2 Templates December 2017

https://www.slideshare.net/alanhamlett1/san-francisco-flask-meetuptemplates

Ipeer (2017) Applikationsserver December 2017

https://www.ipeer.se/applikationsserver.php

Ansible Documentation (2017c) Templating (Jinja2)

http://docs.ansible.com/ansible/latest/playbooks_templating.html

Red Hat JBoss EAP 7 Configuration Guide December 2017

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html/configuration_guide/

Red Hat JBoss AMQ 7 Configuration Guide December 2017

https://access.redhat.com/documentation/en-us/red_hat_jboss_amq/7.0/html/using_amq_broker/

Wikipedia (2017c) Java Message Service November 2017

https://en.wikipedia.org/wiki/Java_Message_Service

Tridens (2017) Combine a robust, yet flexible, architecture for your enterprise. December 2017

http://www.tridens.si/expertise/jboss-middleware/jboss-enterprise-application-platform/

Wikipedia (2017a) JBoss Enterprise application platform December 2017

https://en.wikipedia.org/wiki/JBoss_Enterprise_Application_Platform

Dzone (2017) Getting started with JBoss Enterprise application platform 7 December 2017

https://dzone.com/refcardz/getting-started-jboss

Ansible Documentation (2017a) Inventory

http://docs.ansible.com/ansible/latest/intro_inventory.html

Ansible Documentation (2017b) Playbooks

http://docs.ansible.com/ansible/latest/playbooks.html