ARCADA

Automatiskt spridningsflöde för Dockerbaserade webbprojekt från Gitlab med en Nginx proxyserver

Richard Weber

Richard Weber

Examensarbete

Informationsteknik

2015

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	5103
Författare:	Richard Weber
Arbetets namn:	Automatiskt spridningsflöde för Dockerbaserade webbprojekt från Gitlab med en Nginx proxyserver
Handledare (Arcada):	Magnus Westerlund
Uppdragsgivare:	

Sammandrag:

Syftet med arbetet är att beskriva hur man med Docker kan lösa problem som framkommer i och med de krav som ställs på moderna applikationer. Som bas för arbetet ligger ett system som utvecklats för att skapa ett automatiskt flöde för Dockerbaserade applikationer till en produktionsmiljö. Arbetet börjar med att ta upp teori kring hurdana krav det moderna samhället ställer på applikationer och vilka tekniska krav för arkitekturen detta leder till. Vidare beskrivs de olika systemen som fungerat som komponenter för det praktiska arbetet och sedan beskrivs helhetens funktionalitet. Arbetet avslutas med en genomgång av de saker som skulle kunna vidareutvecklas i systemet, antingen sådana saker som skulle kunna göras direkt eller i framtiden då andra hjälpmedel kommit framåt. Illustrationer i arbetet består av korta kodexempel samt illustrationer som hjälper beskriva systemets arkitektur.

Nyckelord:	Docker, Git, Microservices
Sidantal:	29
Språk:	Svenska
Datum för godkännande:	3.6.2015

DEGREE THESIS	
Arcada	
Degree Programme:	Informations technology
Identification number:	5103
Author:	Richard Weber
Title:	Automatic deployment workflow for Docker based web-projects from Gitlab with a Nginx proxy
Supervisor (Arcada):	Magnus Westerlund
Commissioned by:	

Abstract:

The purpose of the thesis is to describe how one can tackle certain problems that surface when one looks at the requirements set by the expectations of society on modern applications and their architecture. The thesis is based in the practical work of creating a system that will allow for Docker based web projects to be automatically deployed to a production environment. The beginning of the thesis contains some theory on modern applications architecture and how the Docker system can be used as a tool to solve the problems that arise from these requirements. It then goes on to cover the various systems that are used as components in the system that was developed as the practical part of this thesis. Illustrations of code samples and illustrations of the architecture are included to give the reader a better picture of how the system was built and how it works. The thesis is concluded with a chapter that discusses various ways that the system can be improved, either currently or in the future when other tools that could be used have matured.

Keywords:	Docker, Git, Microservices
Number of pages:	29
Language:	Swedish
Date of acceptance:	3.6.2015

INNEHÅLL / CONTENTS

1 Inledning 7

1.1 Bakgrund 7
1.2 Målsättning 8
1.3 Avgränsning 8

2 Microservices 9

2.1 Krav på moderna arkitekturer 10
2.2 Modularitet med services 10
2.3 Utveckling 11

3 Docker 13

3.1 Komponenter 14
3.1.1 Server 15
3.1.2 Klient 15
3.1.3 Register 15
3.1.4 Docker-py 16

4 Git 17

4.1 Git grenar 17
4.2 Gitlab 17
4.2.1 Merge Request 18
4.3 Arbetsflöde 18
4.3.1 Test och produktion 19

5 Utarbetad arkitektur 20

5.1 Komponenter 20
5.2 Controller 21
5.3 Nginx 21
5.4 Flödet 22

6 Vidareutveckling 26

Källor 28

Figurer

Figur 1. Skillnaden mellan Monolitisk och Microservice arkitektur 9
Figur 2. Skillnaden mellan en Docker container och en virtuell maskin 14
Figur 3. Ett Docker kommando 16
Figur 4. Docker-py exempel 16
Figur 5. Schema över komponenters relationer 21
Figur 6. Controllerns exempelkonfiguration 23
Figur 7. Nginxparser exempel 24
Figur 8. Schema över spridningsflödet 25

TERMER OCH FÖRKORTNINGAR

Docker	System för hantering av containrar.
Container	Ett paket som kör en isolerad process.
Git	Ett versionshanteringssystem skapat av Linus Torvalds
Nginx	En webb- och proxyserver.
Virtuell Maskin	En maskin som inte har egen hårdvara utan körs under en annan maskins operativsystem.
Namespace	En samling av resurser som hör ihop.

1 INLEDNING

Detta arbete beskriver två saker, moderna IT arkitekturer som hjälper utvecklare med publicering av mjukvaruprojekt och ett exempelsystem för det. Systemet baserar sig på användning av Git för versionshantering och Docker för paketering av programvara. Systemet skall automatisera så mycket av publiceringsflödet som möjligt. Målet är att en utvecklare inte behöver göra något annat än att överföra sina ändringar till versionshanteringssystemet, efter detta skall allting gå automatiskt och felsäkert. Felsäkerheten i flödet garanteras av Docker, som ser till att projekt inte stöter på problem på grund av att utvecklarnas lokala miljöer avviker från produktionsmiljön.

1.1 Bakgrund

Den traditionella designprincipen för mjukvaruutveckling är att den görs modulärt. Avsikten med användningen av olika externa mjukvarubibliotek är att försnabba utvecklingsprocessen. Användningen av dessa externa bibliotek kan dock skapa problem då ett projekt skall flyttas till produktion. Det är då flera projekt skall köras på samma stack som det kan uppstå problem med olika versioner av bibliotek som inte är kompatibla.

Då en programfunktion i ett projekt förekommer ofta eller är komplex så är det optimalt att försöka använda ett existerande programbibliotek för att inte behöva uppfinna hjulet på nytt. Ibland händer det att ett projekt kräver en viss version av ett externt bibliotek på grund av att det till exempel i nyare versioner har ändrats på sättet som man använder biblioteket. Ifall ett annat projekt på samma maskin använder samma bibliotek, men detta projekt är nyare och använder sig av en nyare version av samma bibliotek, kommer det att uppstå problem då man försöker installera båda versionerna under samma operativsystem. Till exempel använder Drupal, som är ett innehållshanteringssystem för att skapa nätsidor med dynamiskt innehåll, en webbserver, programmeringsspråket PHP med en del moduler, och en databas. (Drupal Association, 2014)

Detta är ett exempel på varför Docker behövs samt varför det har utvecklats. Docker är ett system som är designat för att lösa dylika problem genom att ge utvecklare de verktyg som krävs för att de på ett lätt och användarvänligt sätt skall kunna paketera sina projekt. En container blir då ett paket som innehåller hela stacken med alla externa bibliotek, så att inget annat behövs för att kunna köra applikationen.

1.2 Målsättning

Avsikten med arbetet är att redogöra för kraven som ställs på en modern arkitektur samt att ge en exempellösning som hjälper utvecklare att skapa och sprida deras Dockerbaserade projekt till en produktionsmiljö på ett så automatiskt sätt som möjligt. En utvecklare skall inte behöva fundera på var deras projekts container byggs, var den lagras, eller hur extern trafik från användare skall hitta fram.

Detta projekt kommer att fokusera på de komponenter som krävs för att bygga detta system samt hur de kombineras för att skapa ett flöde som är felsäkert och undviker problem med tjänsters tillgänglighet. Därtill tas det även upp teori kring vilka krav som ställs på moderna applikationer och deras arkitektur och hur Docker som ett verktyg kan hjälpa med att lösa dessa problem.

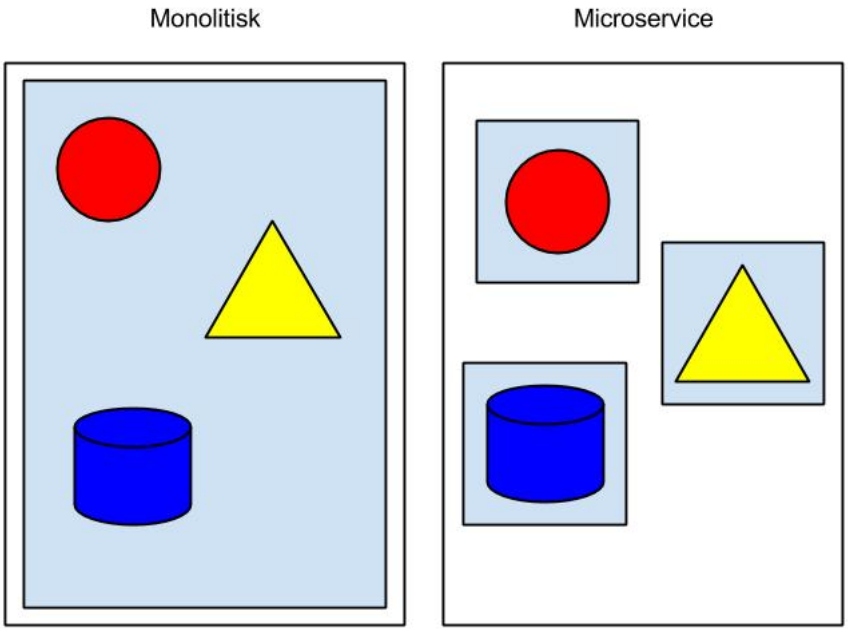
1.3 Avgränsning

Arbetet kommer inte att ge exempel på hur man utvecklar programvara, varken på ett generellt plan eller specifikt för Docker.

2 MICROSERVICES

Följande kapitel tar upp teorin bakom ”Microservice” arkitekturen för programvaruprojekt.

Termen Microservice har kommit till under de senaste åren och beskriver ett sätt att utföra programvaruprojekts arkitektur på ett mera tjänsteorienterat sätt. Traditionellt har stora programvaruprojekt byggts upp så att allting byggs in i ett enda stort projekt och då körs som en helhet (Monolitisk arkitektur). Programvaruprojekt som designats med Microservice arkitektur delar upp programvaran i mindre beståndsdelar som tar hand om specifika uppgifter. Dessa delar (moduler) kommunicerar sedan sinsemellan via standardiserade kommunikationsprotokoll via nätverket.



Figur 1. Skillnaden mellan Monolitisk och Microservice arkitektur

Detta leder till att utvecklingen av stora projekt blir lättare att hantera eftersom man då utvecklar de enskilda modulerna, istället för ett enda stort paket. Det ger också bättre prestanda när de olika modulerna kör i egna processer och då kan underhållas enskilt och skalas ut på hårdvara, dvs. alla delar behöver inte köras på samma maskin. Syftet är

att det skall leda till en mera robust applikation då en modul som får problem inte direkt leder till att hela applikationen får problem. (Fowler, 2014) (Richardson, 2014)

2.1 Krav på moderna arkitekturer

Under senaste åren har beställarens och användarens krav på applikationer ökat exponentiellt. När det för några år sedan var vanligt att applikationers svarstid var några sekunder förväntar sig användare idag att applikationer skall svara på under en sekund. Det skall gå snabbt och smidigt att använda en applikation. Förr ansåg man att system som krävde ett tiotal servrar var stora, idag räcker samma mängd hårdvara inte till för att köra små till medelstora applikationer. Riktigt stora applikationer kräver idag beräkningskraft som körs på tusentals processorer. Underhåll är också något som inte mera förväntas leda till att en applikation är otillgänglig, arkitekturen bör beakta att tjänsten kan underhållas utan att det märks hos slutanvändaren.

Detta sätter nya krav på hur man designar infrastrukturen. Det måste gå att göra underhåll på en modul i stacken utan att användare lider av detta. Stacken måste också kunna balansera användares anrop på ett sådant sätt att det inte blir en flaskhals som gör att användningen blir trög. Felhantering är också kritisk, likaså förståelsen att fel kommer att inträffa vid något skede. Då måste systemet också fortsätta fungera medan man åtgärdar felet. Detta kan lösas med hjälp av balansering och replikering så att flera instanser av en modul jobbar tillsammans och de andra kan ta över om en får ett fel. Isolering krävs också så att den felande länken inte stör de andra instansernas verksamhet.

Ett system måste också kunna reagera på plötsliga förhöjningar i användning och då allokera mera resurser för att hjälpa till med att klara av den ökade mängden anrop. Automatisk skalning av system är också ett ekonomiskt plus då systemet kan skala ner då överloppsresurser inte behövs. (Jonas, Dave, & Martin, 2014)

2.2 Modularitet med services

I projektets bakgrund nämndes användandet av programbibliotek för att göra ett projekt modulärt. När man förverkligar ett projekt med Microservices försöker man ta modula-

riteten ett steg vidare genom att skapa moduler som services istället för bibliotek. Detta gör att funktionaliteten delas upp i flera processer som sedan kan skalas ut på flera virtuella maskiner. När man använder bibliotek för att skapa modularitet sker anrop till bibliotekens funktionalitet i arbetsminnet med hjälp av ett programmeringsspråks definierade metoder för kommunikation. Detta leder till att hela processen skall startas om ifall det tillkommer en ändring i ett av biblioteken.

När man använder services för att skapa modularitet i ett projekt sker kommunikationen mellan moduler med hjälp av ett extern protokoll, t.ex. HTTP. Detta gör att en modul kan köras i en egen process och då kan denna modul startas om, uppdateras eller ersättas utan att påverka hela applikationen. Endast den funktionalitet som modulen har hand om kommer att påverkas.

Det finns även nackdelar med att uppdela en applikation i services, bland annat är ett extern anrop till en annan process mycket långsammare än att kalla den egna processens funktionalitet i arbetsminnet. Att strukturera om en applikation om man vill ändra på modulstrukturen, vilken modul som gör vad och var den kopplas in i applikationens kärna, leder också till mera arbete än om man skulle använda bibliotek i samma process. (Fowler, 2014)

2.3 Utveckling

Att utveckla ett traditionellt programvaruprojekt med en Monolitisk struktur betydde ofta att utvecklingsteamen delades upp enligt arbetsområde och sådan funktionalitet som hade med två eller flera områden att göra, blev ofta komplicerade att förverkliga när kommunikationen brast. När man arbetar med Microservice arkitektur arbetar ett komplett utvecklingsteam per modul, varje modul skall gå att köra utan de andra. När en modul skall kommunicera med en annan ger det ena teamet över dokumentation på hur man kan kommunicera med deras modul.

När man arbetar med ett Microservice projekt med services som moduler måste man vara extra noggrann då man kommunicerar med andra moduler. Ifall en modul blir otill-

gänglig betyder det inte att den egna modulen blir det. Ifall man då skickar ett anrop till den otillgängliga modulen kommer det att uppstå fel.Det år viktigt att man hanterar dessa fel på ett bra sätt så att den egna modulen kan meddela användare om felet på ett snyggt sätt, istället för att krascha.

När man arbetar med monolitiska applikationer blir det ofta så att hela projektet skrivs i samma programmeringsspråk. Detta sker därför att det är det lättaste sättet. Man kan utveckla vissa bibliotek i andra språk, och de skulle ändå fungera, men då måste man ha en utvecklare som kan det språket, vilket tar bort resurser från kärnan av applikationen som ändå utvecklas i ett enda språk. När man jobbar med Microservice arkitekturen är det mycket lättare att använda olika språk för olika saker, enligt metoden rätt verktyg för rätt problem. Detta behöver också utvecklare som kan alla de olika språken man använder, men här är det mera lönsamt än i en Monolitisk applikation. (Fowler, 2014)

3 DOCKER

Detta kapitel tar upp Docker och dess komponenter.

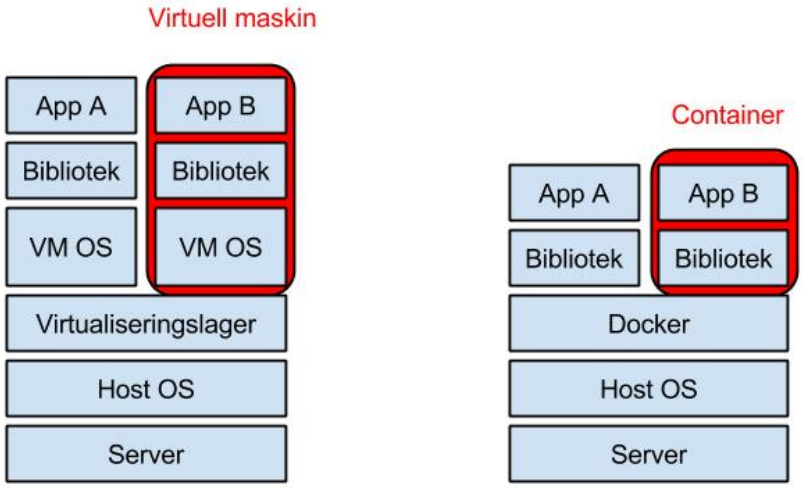
Docker är en plattform som hjälper utvecklare med att bygga och köra distribuerade applikationer. Docker använder sig av moderna funktioner i Linux kerneln som gör det möjligt att isolera processer. Det är detta som utgör grunden för en Docker container. (Docker, Inc, 2014)

Docker är ett mycket bra verktyg då man skall utveckla stora applikationer enligt Microservice arkitektur. Varje modul kan köras i en egen Docker container som en egen isolerad process. Dessa moduler kan sedan kommunicera sinsemellan över nätverksprotokoll. Docker gör det också lättare att ersätta moduler med nya moduler, man stänger ner containern som kör den gamla modulen och startar upp en med den nya. Dessa egenskaper gör att Docker som ett verktyg har de förutsättningar som krävs för att underlätta skapandet av en arkitektur som är flexibel enligt de krav på moderna applikationer som togs upp i kapitel 2.1. Då man snabbt kan skala upp en modul genom att köra igång flera kopior av samma container. I och med att en Docker container isolerar sin process så kräver det att man skapar sin applikation på ett sätt där kommunikationen med andra moduler är flexibel för skalning och isolerbar.

När en stor applikation som utvecklas med denna arkitektur använder Docker som hjälpmedel så blir det också lättare att publicera den. Vilken som helst maskin som kan köra Docker containrar kan köra applikationen. Det kan vara bra då man vill skapa en testmiljö, eller om man skall köra applikationen hos en kund på deras hårdvara.

En Monolitisk applikation kan också köras i en Docker container, men man tjänar inte lika mycket på det eftersom man då i containern skulle köra samma ensamma process som man i vanliga fall skulle köra direkt på en maskins operativsystem. Detta betyder att man inte vinner lika mycket på att använda Docker för stora monolitiska applikationer.

Docker containrar kan likna virtuella maskiner, men under huven är de mycket olika. En virtuell maskin har ett eget operativsystem med processhantering. En container delar operativssystemskärnan (eng. kernel) med Docker hosten och i en container körs endast en process.



Figur 2. Skillnaden mellan en Docker container och en virtuell maskin

3.1 Komponenter

Docker består av flera komponenter som tillsammans utgör helheten. Dessa komponenter har alla en uppgift. De obligatoriska komponenterna för att använda Docker är Dockers server och klient. Därutöver används några andra stödkomponenter.

3.1.1 Server

Dockers serverdel är motorn som tar hand om att köra applikationer i isolerade namespace (Docker, Inc, 2014). Den tar hand om processen och håller reda på den under dess livstid, den har också hand om nätverkskommunikationen för alla containrar som den kör.

Docker servern lyssnar traditionellt på en unix socket, vilket leder till att man endast kan kontrollera den med en klient installerad på samma maskin. Däremot så kan man konfigurera servern att lyssna på en TCP socket, antingen med HTTP, vilket gör att vem som helst med tillgång till nätverket kommer åt att kontrollera Docker servern, eller med HTTPS vilket då kräver att klienter identifierar sig med TLS certifikat. (Docker, Inc, 2014)

3.1.2 Klient

Docker klienten är den programvara som användare använder för att kontrollera servern. Klienten kan köras på samma maskin som servern, men kan också kommunicera med servern över nätverket Till exempel kan en utvecklare köra klienten på sin bärbara dator och därifrån kontrollera en Docker server i en molntjänst. Klienten körs från kommandoraden och har inte ett grafiskt användargränssnitt.

3.1.3 Register

Registret är den del av Docker som sköter om att ett projekts avbild (eng. image) lätt kan flyttas från en utvecklares dator till en test- eller produktionsmiljö. Utvecklare laddar upp en avbild till registret och därefter kan alla som har tillgång till registret hämta ner denna avbild och köra den på vilken som helst maskin som har Docker servern installerad. Registret kan man administrera själv och köra på en server i sitt eget nätverk, eller använda den kommersiella tjänsten Docker Hub. Registret har samma kommunikationsalternativ som servern, man kan ha ett osäkrat register som tillåter nerladdning över HTTP, eller säkra sitt register med TLS och då specificera olika rättigheter för olika användare. (Docker, Inc, 2014) (Docker, Inc, 2014)

3.1.4 Docker-py

Docker-py är ett Python paket som gör det möjligt att skapa Python program som kommunicerar med en Docker server direkt via en TCP socket. Detta är även vad som har använts i den praktiska delen av detta projekt. (\cite{dockerpy})

Dessa exempel beskriver skillnaden mellan att använda Dockers klient på kommandoraden och Docker-py. Det som händer i båda fallen är att man kontaktar en Docker server som finns på IP-adressen 1.2.3.4 och lyssnar på port 2375. Efter att kontakten har upprättats så skickas kommandot som berättar för Docker att den skall bygga en avbild som taggas med namnet ”example”. Man berättar också var källkoden för avbilden finns, i detta fall i foldern example i filsystemets root.

```
docker -H tcp://1.2.3.4:2375 build -q -t 'example' /example/
```

Figur 3. Ett Docker kommando

```
builder = Client(base_url="tcp://1.2.3.4:2375")
builder.build(
            path     =   '/example',
            quiet    =   True,
            pull     =   True,
            rm       =   True,
            tag      =   'example'
            )
```

Figur 4. Docker-py exempel

4 GIT

I kapitlet nedan förklaras kort vad Git är och hur det används i systemet.

Git är ett på öppen källkod baserat versionhanteringssystem med fokus på distribuerat arbete (då flera personer jobbar på samma kodbas). Det som är speciellt med Git är hur det hanterar grenar i projekt i jämförelse med andra versionhanteringssystem. När en utvecklare arbetar på ett projekt som använder Git för versionshantering sparar han ändringar genom att använda Gits commit kommando. Detta skapar en ny revision av projektet. Git hanterar revisioner genom att lagra pekare som pekar på en avbild av projektets nuvarande läge, en sådan avbild kallas för en commit. (Git, 2014) (Git Branching - Branches in a Nutshell, 2014)

4.1 Git grenar

Att använda grenar i versionshantering betyder att man tar ett steg åt sidan i utvecklingsprocessen och kan då isolera arbetet mot en viss funktion utan att blanda in dessa ändringar i det generella flödet av projektet. I Git fungerar grenar så att när man skapar en gren skapar Git en till pekare med metadata. När man sedan skapar commits till en viss gren kommer denna pekare att uppdateras så att den på det viset håller reda på vilka ändringar som hör till vilka grenar. (Git Branching - Branches in a Nutshell, 2014)

Detta betyder att en utvecklare som skall jobba på en ny funktion för ett projekt kan skapa en gren där han kan utveckla denna gren isolerad från vad som händer med resten av projektet. När funktionen är färdig så sammanslår (eng. merge) utvecklaren ändringarna i hens gren med den allmänna utvecklingsgrenen.

4.2 Gitlab

Gitlab är en programvara som baserar sig på Git. Den har en hel del funktioner utöver det som Git erbjuder, som bland annat ett webbgränssnitt där man kan se projektets historia, diskutera problem (sk. issues), och merge requests. (GitLab, 2014)

Gitlab kan köras på egen infrastruktur så som vi gör för detta projekt, men Gitlab erbjuder också en kommersiell molntjänst, gitlab.com, var de har hand om körandet av systemet.

4.2.1 Merge Request

Merge requests (även kända som 'Pull requests' från andra verktyg) är en funktion i Gitlab (och andra visuella Git lösningar) som gör att när man skall sammanslå en gren med en annan så kan man skapa en diskussion om ändringarna. Detta ger en möjlighet för att hålla en bättre kontroll på kodkvaliteten i ett projekt. (Making a Pull Request, u.d.)

4.3 Arbetsflöde

Arbetsflödet definierar hur man strukturerar sitt arbete med Git. Vilka regler som sätts upp för utvecklare så att alla jobbar på samma sätt med Git. Detta gör att samarbete mellan flera utvecklare blir smidigare.

Det flöde som detta projekt har som grund från versionhanteringens sida är ett flöde där master grenen (eng. branch) motsvarar den kod som körs i produktion. Därtill har man en develop gren var det alldagliga utvecklingsarbetet görs. Ur denna develop gren skapar utvecklare egna grenar för specifika funktioner som de jobbar på.

Master grenen är skyddad, vilket betyder att utvecklare inte kan göra commits i den direkt. Detta för att försäkra sig om att all kod som kommer in till produktion har blivit granskad av en annan utvecklare.

För att sammanslå (eng. merge) kod från develop grenen till master gör man en så kallad release gren. I denna gren kommer inga nya funktioner mera med utan endast buggfixar. För att sedan sammanslå denna gren med master så skapar man en merge request var man ännu diskuterar koden och finslipar arbetet före man godkänner merge requesten.

När en merge request till master blir godkänd så sammanslås koden i master grenen och därefter tar systemet över och ser till att projektets nyaste version kommer till produktion. (Driessen, 2010)

4.3.1 Test och produktion

Eftersom systemet automatiskt tar ner projektet från master grenen och placerar det i produktionsmiljön så vill man vara säker på att det kommer att gå bra. Detta gör man genom att ha en testmiljö. Denna miljö har ett liknande system som tar ner develop grenen och kör den på egna servrar. Detta ger också utvecklare, och andra intressenter, möjligheten att se hur nästa version av projektet kommer att se ut.

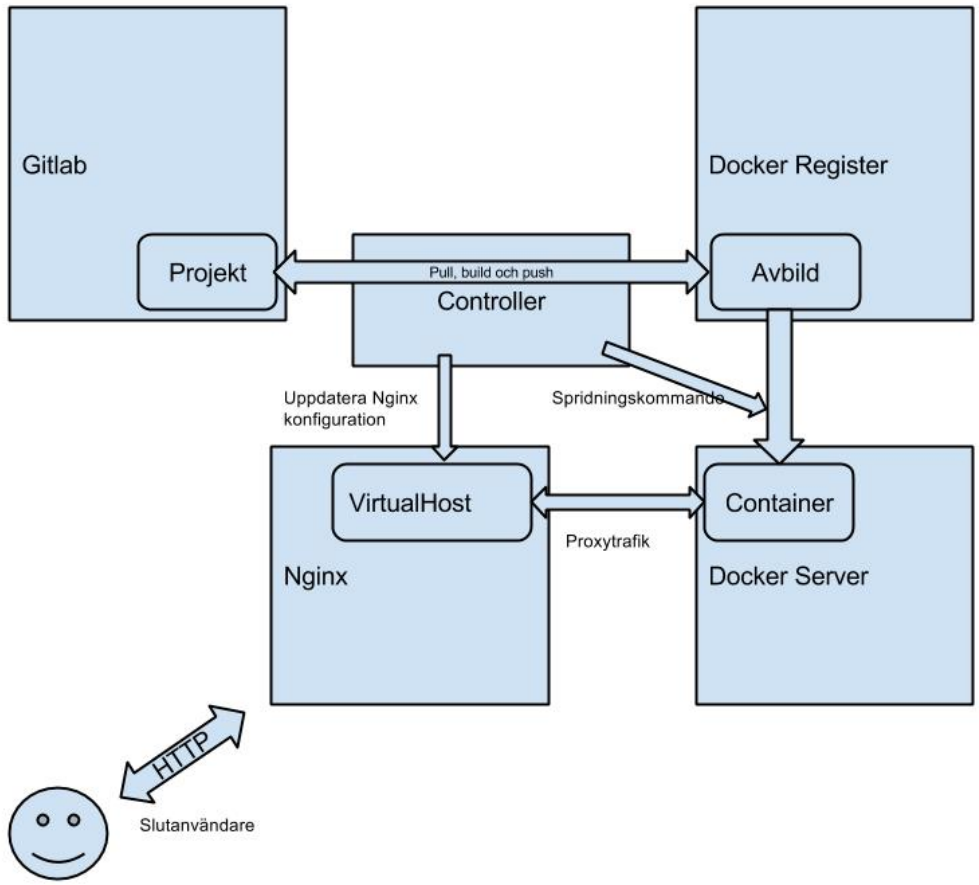
5 UTARBETAD ARKITEKTUR

En kombination av Docker och Git är vad vi eftersträvar i detta projekt. Genom att utveckla ett system där utvecklare endast behöver använda Git för att kunna flytta sina ändringar till produktion, erbjuder vi maximal stabilitet genom att garantera att varje gång nya ändringar skall överföras så sker de på samma sätt, inga är inblandade och kan därför inte göra inkonsekventa misstag.

5.1 Komponenter

De komponenter som används för att åstadkomma detta är Gitlab och Python. Vi använder Gitlab som centralt system där vi lagrar projektens källkod och historia. Utvecklare skall kunna använda Docker för deras lokala utvecklingsmiljö om de så önskar, men de skall inte vara ett krav. Ett projekt har dock ett krav på att innehålla en fil med namnet ”Dockerfile” som beskriver hur Docker containern skall skapas. (\cite{dockerfile})

Python används som språk för den kontrollerande mjukvaran som tar hand om att projektet tas ner från Gitlab, byggs till en Docker container och sedan placeras i produktion. En Nginx webbserver används som proxy, detta betyder att slutanvändares webbläsare skickar anrop till denna proxyserver som sedan vidarebefordrar trafiken till rätt Docker container.



Figur 5. Schema över komponenters relationer

5.2 Controller

Controllern är den mjukvara som kontrollerar hela systemet. Controllern är skriven i programmeringsspråket Python och använder sig av Docker-py biblioteket för att kommunicera med Docker servrar och Docker register (Docker-py, 2014). Controllern använder sig även av ett bibliotek som hjälper med serialiseringen av Nginx konfigurationsfiler. (fatiherikli/nginxparser, u.d.)

5.3 Nginx

Nginx är en modern HTTP server och proxy. Den kan fungera som en traditionell webbserver som kan ge ut innehåll åt användare eller vidarebefordra anrop till en eller

flera andra servrar. Detta gör det möjligt att hålla applikationsservrar på ett privat nätverk var endast Nginx proxyservern är tillgänglig utifrån.

I detta projekt används Nginx på det ovannämnda sättet. Docker servrarna, Gitlab och controllern är på ett privat nätverk där de kan kommunicera sinsemellan medan Nginx är tillgänglig från Internet och sedan vidarebefordrar anrop till rätt Docker container. Controllern konfigurerar Nginx genom att skapa konfigurationsfiler som definierar vilket domännamn och vilken sökväg som skall vidarebefordras till vilken container

5.4 Flödet

När en utvecklare använder kommandot 'git push' för att skicka sina ändringar till Gitlab så kommer Gitlab att ta emot ändringarna, varefter Gitlab skickar ett HTTP POST anrop till controllern. Detta HTTP anrop innehåller data som bland annat berättar för mottagaren vilket projekt det handlar om.

När controllern tar emot detta anrop så lagrar den först projektets namn i en variabel, varefter den kollar upp ifall det finns konfiguration tillgänglig för projektet i fråga. Ifall det inte finns en tillgänglig konfiguration, som berättar mera detaljer för controllern om vad som skall göras med projektet, förkastas anropet och ingenting mera görs.

```
config = {
    'buildServer': 'tcp://127.0.0.1:4243',
    'hostServer': 'tcp://127.0.0.1:4243',
    'registry': '0.0.0.0:5000',
    'gitPath': '/root/git',
    'nginxServer': 'user@127.0.0.1',
    'projects': {
        'git-project-name': {
            'domain': 'foo.bar.com',
            'path': '/',
            'environment': {
                'foo': 'bar',
            },
            'port': 80,
        }
    }

}
```

Figur 6. Controllerns exempelkonfiguration

Ifall controllerns konfiguration innehåller detaljer om projektet i fråga, utför controllern en 'git pull' operation för att hämta projektets källkod från Gitlab. När koden är hämtad, använder controllern sig av Docker-py biblioteket för att kontakta en Docker server som är specificerad som 'buildServer' i controllerns konfiguration. Controllern använder sig sedan av denna server för att bygga en Docker container av projektet som innehåller alla de bibliotek och tjänster som projektet kräver för att kunna köras. Efter att containern är skapad, laddar controllern upp den till det Docker register som är specificerat i controllerns konfiguration. Detta kan antingen vara den kommersiella tjänsten Docker Hub, eller ett Docker register som man kör själv någon annanstans.

När containern har blivit uppladdad till registret kontaktar controllern en annan Docker server. Det är denna server som kommer att köra Docker containern. Controllern berättar för servern att den skall hämta ner den nyaste versionen av containern från registret och köra den.

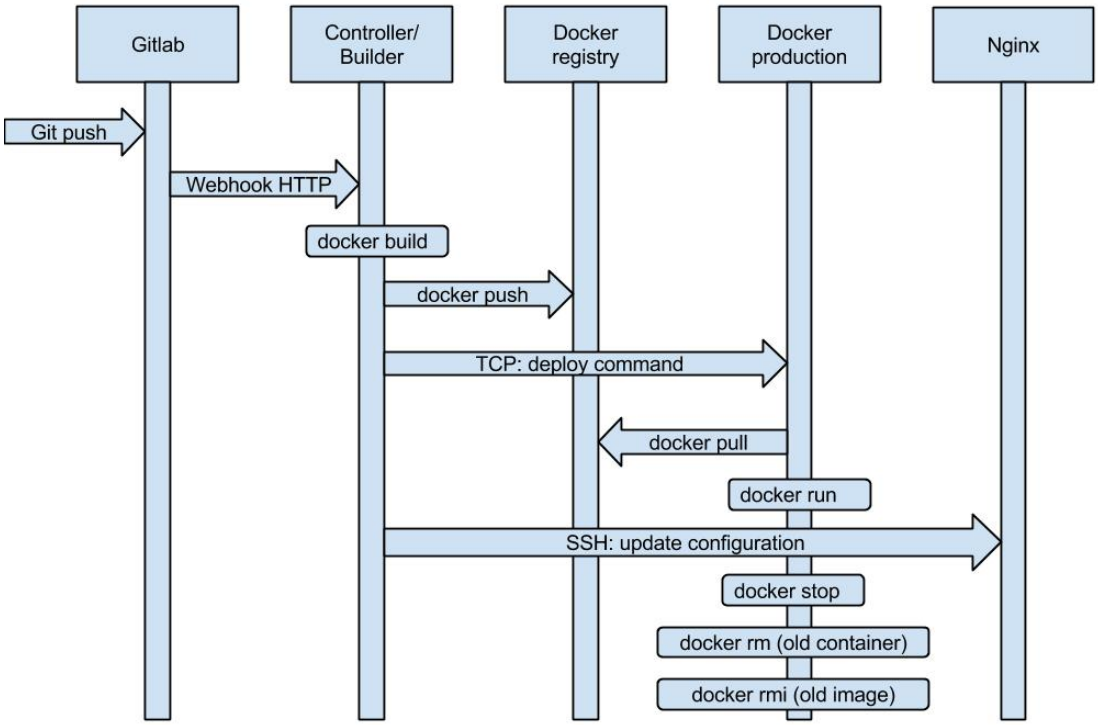
När containern är igång på servern tar controllern reda på vilken port Docker servern har vidarebefordrat till containern. Controllern skapar en Nginx konfigurationsfil som har samma namn som projektet och innehåller detaljer om var projektet hittas. Denna

Nginx konfigurationsfil ges sedan åt den server som agerar som Nginx proxy, som tar emot och styr alla containrars publika trafik, så att den vet var projektet finns.

```
nginx_config = [
    ['server'], [
        ['listen', '80'],
        ['server_name', 'foo.com'],
        [['location', config['projects'][repo]['path']], [
            ['proxy_set_header', 'X-Real_IP $remote_addr'],
            [
                'proxy_set_header',
                'X-Forwarded-For $proxy_add_x_forwarded_for'
            ],
            ['proxy_read_timeout', '90'],
            ['proxy_http_version', '1.1'],
            ['proxy_pass', 'http://1.2.3.4:49152'],
        ]]
    ]
]
```

Figur 7. Nginxparser exempel

När Nginx proxyservern har blivit konfigurerad stängs eventuella gamla containrar för projektet ner. Detta leder till att en slutanvändare inte kommer att uppleva något uppehåll när hen använder tjänsten, eftersom Nginx kommer att styra trafik till den nya containern före den gamla stängs ner. Efter att allt detta är gjort återgår controllern till att vänta på nya anrop från Gitlab.



Figur 8. Schema över spridningsflödet

6 VIDAREUTVECKLING

I detta kapitel granskas arbetet kritiskt och förslag på vidareutveckling för systemet gås igenom.

Det finns några funktioner som systemet skulle ha nytta av för att göra det mera flexibelt och för att kunna skala upp olika projekts prestanda enligt behov.

Att göra det möjligt att definiera flera än en ”hostServer” i systemets konfiguration för att kunna skala projekt horisontellt över flera servrar skulle vara fördelaktigt. Detta skulle se till att inte alla projekt körs på en server, utan sprids ut jämt över flera. En lösning på detta skulle vara att definiera en lista på servrar som skulle användas för publicering, och sedan välja den som har minst att göra. Ett annat alternativ skulle vara att använda Dockers Swarm verktyg som skulle ta hand om detta automatiskt utan några ändringar i detta system. Däremot så är Swarm ännu under utveckling och är inte i skrivande stund färdigt för produktionsmiljöer. (Docker Swarm, 2014)

Möjligheten att sätta med projektets konfiguration i en fil i projektets egna kodbas skulle göra systemet mera flexibelt. Då skulle man inte behöva konfigurera om systemet för varje nytt projekt utan systemet skulle istället läsa projekts konfiguration från en fil som kommer med projektet. Detta skulle kunna vara en simpel textfil som beskriver konfigurationen. Till exempel så skulle YAML, som är en serialiseringsstandard som hjälper till att skapa filer som är lätta att läsa både för människor och maskiner, kunna användas för denna fil. (YAML, u.d.)

YAML skulle även kunna användas som systemets egna konfiguration, istället för det Python objekt som används för tillfället. Det skulle tillåta konfigurationsändringar utan att kräva en omstart av systemet.

En säkerhetsförbättring som skulle vara bra att göra skulle vara att lägga till stöd för användande av TLS säkrad TCP kommunikation mellan controllern och Docker servrar. Då skulle man kunna använda detta med servrar som ligger utanför ens egna interna nätverk, till exempel med servrar i ett publikt moln. Ifall man skulle använda Docker

Machine verktyget för att skapa ett Docker Swarm kluster så skulle Machine ta hand om detta automatiskt. Docker Machine är ännu under utveckling och inte redo för produktion. (Docker Machine, 2014)

Docker har flera verktyg, bl.a. Machine och Swarm, som skulle kunna användas i samband med detta systems controller för att göra skalbarhet och säkerhet lättare tillgängligt. Meningen är att följa med hur dessa verktyg mognar och sedan implementera funktionalitet i controllern som använder dessa verktyg där var de passar in.

KÄLLOR

Docker Machine. (2014). Hämtat från https://github.com/docker/machine den 19 April 2015

Docker registry. (2014). Hämtat från https://github.com/docker/docker-registry den 8 Mars 2015

Docker Swarm. (2014). Hämtat från https://docs.docker.com/swarm/ den 19 April 2015

Docker, Inc. (2014). *Docker basics*. Hämtat från https://docs.docker.com/articles/basics/ den 8 Mars 2015

Docker, Inc. (2014). *Dockerfile*. Hämtat från https://docs.docker.com/reference/builder/ den 16 Mars 2015

Docker, Inc. (2014). *Understanding Docker*. Hämtat från https://docs.docker.com/introduction/understanding-docker/ den 8 Mars 2015

Docker, Inc. (2014). *What is Docker?* Hämtat från https://www.docker.com/whatisdocker/ den 8 Mars 2015

Docker, Inc. (2014). *Working with Docker hub*. Hämtat från http://docs.docker.com/userguide/dockerrepos/ den 8 Mars 2015

Docker-py. (2014). Hämtat från https://github.com/docker/docker-py den 8 Mars 2015

Driessen, V. (2010). *A successful Git branching model*. Hämtat från http://nvie.com/posts/a-successful-git-branching-model/ den 24 Mars 2015

Drupal Association. (2014). *Drupal System Requirements*. Hämtat från https://www.drupal.org/requirements den 8 Mars 2015

fatiherikli/nginxparser. (u.d.). Hämtat från https://github.com/fatiherikli/nginxparser den 8 Mars 2015

Fowler, M. (2014). *Microservices*. Hämtat från http://martinfowler.com/articles/microservices.html den 21 April 2015

Git. (2014). Hämtat från http://git-scm.com/about/ den 24 Mars 2015

Git Branching - Branches in a Nutshell. (2014). Hämtat från http://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell den 24 Mars 2015

Gitlab. (u.d.). Hämtat från https://about.gitlab.com/features/ den 24 Mars 2015

Jonas, B., Dave, F., & Martin, T. (2014). *The Reactive Manifesto*. Hämtat från http://www.reactivemanifesto.org/ den 6 Maj 2015

Making a Pull Request. (u.d.). Hämtat från https://www.atlassian.com/git/tutorials/making-a-pull-request/ den 24 Mars 2015

Richardson, C. (2014). *Pattern: Microservices Architecture*. Hämtat från http://microservices.io/patterns/microservices.html den 21 April 2015

YAML. (u.d.). Hämtat från http://yaml.org/ den 19 April 2015