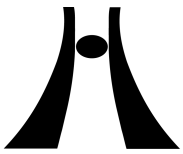
Beteckning:_____



HÖGSKOLAN
I GÄVLE

Akademin för teknik och miljö

Kontinuerlig leverans till molntjänst

Mikael Löf
juni 2012

Examensarbete, 15 högskolepoäng, B
Datavetenskap

Datavetenskapliga programmet
Examinator: namnet här
Handledare: ett namn till här

Kontinuerlig leverans till molntjänst

av

Mikael Löf

Akademin för teknik och miljö
Högskolan i Gävle

801 76 Gävle, Sverige

Email:
ndi09mlf@hig.se

Abstrakt

Denna rapport beskriver en implementation för att uppnå kontinuerlig leverans till molntjänst. Ett lyckat resultat erhålls där applikationen dessutom kan uppdateras med minimal påverkan för slutanvändaren genom ett så kallat blue green pattern. Problem med hur sessionsinformation ska hanteras vid uppdatering av applikationen tas upp och en slutsats dras att av de två testade molntjänsterna, Heroku och Amazon, är Amazon att föredra på grund av större möjligheter för manuell konfiguration.

Revisionshistorik

Datum	Revision	Anmärkning	Signatur
12-04-02	1	Rapport skapad	ML
12-04-12	2	Rapport uppdaterad på ett antal punkter	ML
12-04-19	3	Lade till Jenkins och Maven	ML
12-04-25	4	Lade till Flyway och databas	ML
12-05-03	5	Cobertura samt Cucumber	ML
12-05-10	6	Amazon	ML
12-05-14	7	Förbättrad teori	ML
12-05-16	8	Blue Green Pattern	ML

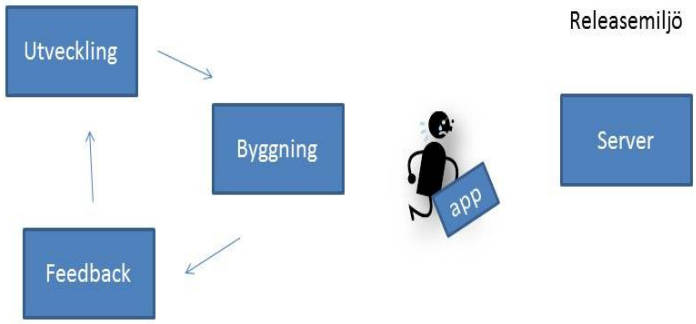
Innehållsförteckning

1 Inledning

1.1 Bakgrund

1.1.1 Systemet idag

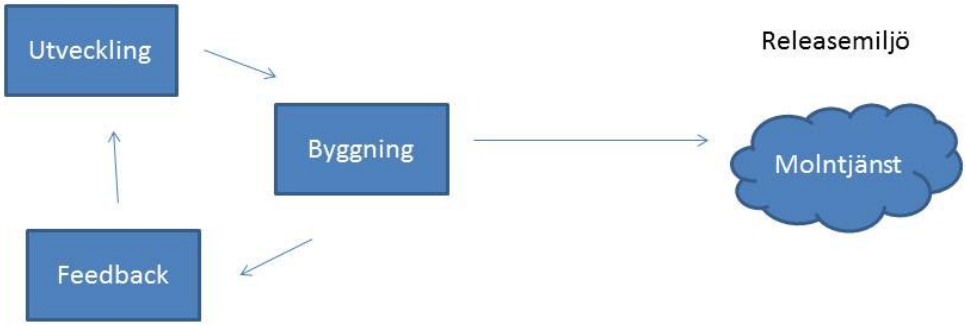
Många mjukvaruprojekt bedrivs idag med så kallad kontinuerlig integration. Det innebär att hela den utvecklade produkten regelbundet byggs för verifiering av funktionalitet. Utvecklare får snabb feedback på de delar som inte integrerats korrekt med övriga systemet och kan då rätta till detta direkt. Dock erhålls ingen feedback från kund eller slutanvändare.
Ett positivt resultat av testningen av hela systemet bör innebära att teamet har en fullt fungerande produkt. Trots det är det ofta med mycket långt intervall som produkten levereras till kund så att denne kan ta del av ny funktionalitet, se *figur 1*. Att detta intervall ofta är långt beror dels på att leverans inom många projektär en högst manuell och tidskrävande process, dels på att utveckling ofta planeras utan slutanvändaren i åtanke varför den nya funktionaliteten eventuellt inte tillför något värde till denne. Exempelvis kan funktionalitet i modellen utvecklas som till en början inte presenteras i gränssnittet.



Figur 1: Systemet idag

1.1.2 Kontinuerlig leverans

Tanken med kontinuerlig leverans är att inte bara testa ny funktionalitet regelbundet utan även leverera ny funktionalitet regelbundet [1] , se *figur 2*. På detta sätt får kunden tidigt i utvecklingsprocessen ut värde av produkten och kan ge feedback till utvecklingsteamet.
Leveransen ska ske i princip helt automatiskt för att säkerställa att processen genomförs likadant vid varje tillfälle och minimera risken för mänskliga misstag. Dessutom ska systemet testas automatiskt för att verifiera den tillagda funktionaliteten.
Denna typ av leveranssystem blir alltmer populärt och som exempel kan nämnas att Google Mail levererar i genomsnitt fyra gånger per dag.[5]



Figur 2: Kontinuerlig leverans till molntjänst

1.2 Syfte

Syftet med det här projektet är att implementera en komplett miljö för kontinuerlig leverans av en webb-applikation till molntjänst.
Önskvärt är även att implementera funktion för att applikationen ska kunna uppdateras utan att detta är märkbart för slutanvändare, det vill säga, utan nertid.
Fokus ska inte ligga på applikationen. Dock behövs en enkel applikation för att möjliggöra verifikation av systemets funktionalitet.

1.3 Frågeställningar

Följande frågeställningar väcktes under samtal med handledare samt uppdragsgivare innan projektet drog igång.

1. Vilka typer av molntjänster finns och vad skiljer dessa åt?
2. Vilken typ samt vilken konkret molntjänst skulle passa bäst för det aktuella projektet?
3. Vilken typ av verktyg krävs för att få en automatiserad process för leverans till molntjänst och hur ser detta ut ur utvecklarens perspektiv?
4. Hur kan uppdatering av applikationen göras utan att detta märks för användare?
5. Hur kan specifikationer anges så att utvecklare och beställare kan komma överens om entydiga krav?

2 Arbetsmetod

Projektet har bedrivits agilt med veckoiterationer löst baserat på scrum-upplägget (se teori). I början av varje vecka har arbetet planerats och i slutet utvärderats. Vid arbete har fokus legat på en enskild uppgift åt gången, exempelvis ”Leverera Hello World till molntjänst”.

3 Teori

De huvuddelar som ingår i kontinuerlig leverans är *en passande arbetsmetod, versionshantering, automatiskt bygge, en byggserver, ett blue green pattern, automatiska tester, automatisk testning av kodkvalité, automatiskt migrering av databasen* samt *automatisk deployment till release-miljö*, i detta projekt en *molntjänst*. [1]

3.1 Sammanfattning av verktyg

Nedan följer en sammanfattning som beskriver hur respektive verktyg som använts kommer in i helhetsbilden. Efterföljande delar av teori tar upp dessa verktyg var för sig.
Applikationsutveckling läggs upp enligt INVEST-modellen. Funktionalitet utvecklas lokalt i java enterprise-miljö och med stöd av ramverket JSF 2. Applikationen testdrivs fram med enhetstester som skrivs med stöd av jUnit. Vid behov av databasuppdateringar skrivs dessa som sql-kod i speciella Flyway-filer för att säkerställa att applikationen alltid körs med korrekt databas i grunden. Beroenden av tredjepartsbibliotek samt eventuella Plug-in som krävs för att köra applikationen specificeras i Maven för att bygget ska kunna genomföras på olika maskiner.
När funktionaliteten är tillagd pushas koden till ett distribuerat git-arkiv. Via git-hub åskådliggörs arkivet. När ny kod pushats upptäcks detta av Jenkins-mjukvaran på byggservern som tar ner den senaste versionen och bygger hela applikationen enligt Maven-specifikation. Därefter utförs högnivåspråk skrivna i Selenium 2. Dessa test är lättlästa för en icke programmeringskunnig då de skrivs med hjälp av Cucumber, vilket ger en syntax liknande löpande text. För att dessa högnivåtest ska kunna utföras på en deployad applikation används Surefire för att förhindra exekvering av testen under den normala testfasen samt Cargo som automatiskt deployar applikationen i en container. Under deployandet till containern utförs nödvändiga databasuppdateringar av Flyway. Sonar ger ett stöd när det gäller kodkvalité.
Om dessa test passerar levereras applikationen, antingen via git till Heroku där applikationen byggs på nytt på samma sätt, eller så skickas den färdiga war-filen via ett ant-script till Amazon.

3.2 Arbetsmodell

Kontinuerlig leverans förutsätter att utvecklingsteamet jobbar i korta iterationer där varje iteration fokuserar på att tillföra värde för kund och slutanvändare. Att exempelvis bygga databaslagret först fungerar inte eftersom detta inte tillför något värde för kunden och knappast kan levereras.

3.2.1 Invest

En hjälp när det gäller att besluta hur ny funktionalitet ska tillföras är att hålla sig till INVEST [8].
Invest är en akronym och innebär att varje tillägg av funktionalitet ska vara:
Oberoende(Independent)
Eftersom funktionaliteten ska kunna levereras till kund får den inte vara beroende av andra, ännu inte skapade funktioner.
Förhandlingsbar(Negotiable)
Den nya funktionaliteten ska specificeras på hög nivå utan tekniska detaljer så att kund och utvecklare under processen kan förhandla om detaljerna för implementationen.
Värdefull(Valuable)
Tillägget ska innebära värde för kunden. Ett tillägg på en låg nivå i systemet som inte representeras i gränssnittet tillför inget värde till kunden då denne inte kan ta del av funktionaliteten.
Möjlig att tidsuppskatta(Estimatable)
Funktionaliteten ska vara så pass liten att det ska gå att uppskatta när den är färdig samt att denna uppskattningen faktiskt ska kunna hållas.
Liten(Small)

Se tidigare punkt. Att under lång tid inte leverera en produkt för att en del av funktionaliteten inte är komplett är dyrt för det utvecklande företaget och det är svårt för kunden att verifiera att utvecklingen går åt rätt håll.

Testbar(Testable)

Funktionaliteten ska kunna testas med till största delen automatiska test. Om funktionaliteten är vald på ett korrekt och väldefinierat sätt bör det vara relativt enkelt att verifiera denna funktionalitet. Här kommer även exekverbara specifikationer in vilket gör att kund och utvecklare på förhand kan komma överens om faktiska användningsfall som ska fungera, *se 3.5,*.

3.2.2 Scrum

Scrum innebär att teamet jobbar i korta iterationer med fokus på väldefinierade uppgifter[11]. Ihop med kontinuerlig leverans skulle detta kunna innebära att utveckling av ny funktionalitet, värdefull för kund och slutanvändare, utvecklas under varje iteration och att leverans sker i slutet av iterationen.

3.3 Automatiskt bygge

Då kontinuerlig leverans innebär att produkten ska köras i olika miljöer underlättar det att otvetydigt specificera bygget i ett dokument istället för att manuellt sätta upp denna miljö genom att exempelvis ladda ner tredjepartsbibliotek. Med ett dokument där bygget specificeras säkerställs bland annat att samma version på tredjepartsbiblioteken används i samtliga miljöer.

3.3.1 Maven

Maven är ett verktyg för byggande av java-applikationer [12]. I en XML-fil vid namn pom.xml(project object model) specificeras vilken typ av fil som ska byggas, beroenden och plug-in. Dessa beroenden och plug-in laddas sedan ner(vid behov) från ett Maven-arkiv, antingen Maven central eller ett explicit specificerat och finns därefter tillgängliga lokalt så att framtida byggen kan genomföras snabbare. De flesta moderna utvecklingsmiljöer har stöd för Maven-projekt och kan öppna en pm-fil och automatiskt importera beroenden.

Maven har livscyklar som specificerar i vilken ordning Maven genomför åtgärder. När ett kommando ges till Maven att utföra ett av stegen i livscykeln genomförs alltid alla steg fram till det angivna steget.

De två livscyklar som är vanligast är clean och default. Clean är en simpel livscykel som ser till att städa bort tidigare projekt medan default är ”huvudlivscykeln” och innehåller en mängd steg för bland annat kompilering, integrationstestning och paketering.

3.4 Versionshantering

För att kunna bedriva en säker utveckling av ett projekt där teamet består av ett flertal utvecklare krävs ett verktyg för versionshantering där samtliga versioner av produkten arkiveras och eventuella konflikter i filer uppmärksammas.

Inte bara källkod bör lagras här utan samtliga filer som på något sätt påverkar projektet ska versionshanteras. Detta gör att vid tillbakarullning till föregående versioner riskeras inte att applikationen ska gå sönder på grund av att applikationskoden inte är kompatibel med övrig konfiguration.

Det är även versionshanteringssystemet som pollas av byggservern, vilken uppmärksammar och bygger den senaste versionen av applikationen uppladdad på molntjänsten.

3.4.1 Git

Git är ett versionshanteringssystem som använts i detta projekt på grund av den utbredda användningen samt kopplingen till community GitHub(se nedan). Ett lokalt arkiv kopplas mot ett för projektet gemensamt, distribuerat arkiv. Ändringar läggs till i det lokala arkivet och slås sedan lokalt ihop(merge) med filerna på det distribuerade arkivet. Därefter ”pushas” ändringarna till det distribuerade arkivet.

Git utvecklades från början av Linus Torvalds, Linux utvecklare[13].

3.4.2 GitHub

Ett community där utvecklare kan ladda upp projekt utvecklade med Git versionshanteringssystem. Under detta projekt har den kostnadsfria versionen använts där all kod som lagras på GitHub hanteras som Open Source, dvs blir tillgänglig för allmänheten.

3.5 Testning, kodkvalité och exekverbara specifikationer

Funktionalitet för en produkt definieras av högnivåtest som utvecklare och kund kommer överens om. Med hjälp av exekverbara specifikationer kan utvecklare och kund tala samma språk. Exekverbara specifikationer är konkreta exempel som definieras i ett språk som är läsbart för icke programmeringskunniga och kopplade till automatiska test exempelvis skrivna i jUnit. Dessa test driver sedan utvecklingen och vid införande av ny funktionalitet kan de tidigare testen verifiera att den nya funktionaliteten inte har orsakat fel i tidigare funktionalitet.
På samma sätt definieras på lägre nivå funktionalitet för varje klass genom test som sedan driver utvecklingen framåt.
För att mäta kodkvalité och testteckning finns olika mer eller mindre avancerade verktyg. Huruvida dessa verktyg automatiskt ska kunna stoppa en release tas upp under *7.2*.
Att ha en vältestad produkt är en förutsättning för användandet av kontinuerlig leverans. Det hade inte varit möjligt att vid varje ny release genomföra manuella tester på grund av tidsåtgången samt risken för mänskliga misstag.

3.5.1 jUnit

Ett testverktyg för java[14]. Genom användandet av olika assert-metoder kan funktionalitet testas. Om ett test inte går igenom visas ett tydligt meddelande om det antagande som inte stämmer. Detta möjliggör verifiering av logisk funktionalitet i programmet som testas.

3.5.2 Selenium 2

Ett verktyg för analys av innehållet på en hemsida[15]. Givet namn eller id på ett html-element kan värden på attribut erhållas vilka sedan kan användas för testning. Kombinerat med jUnit ger detta möjlighet till automatiska högnivåtest från det allra yttersta lagret i applikationen, det grafiska gränssnittet.

3.5.3 Cucumber

Cucumber erbjuder ett ramverk för högnivåtest [16]. Uttrycken har formen:

1. Givet att - följt av angivna värden av användaren.
2. När – följt av operationen användaren utfört när testet ska utvärderas.
3. Då – följt av ett booleanskt uttryck som är det uttryck som ska utvärderas.

Specifikationerna skrivs i klartext i en så kallad feature-fil som sedan ”limmas” ihop genom reguljära uttryck med test-kod.

3.5.4 Cobertura

Ett verktyg för kontroll av testtäckning[17]. Verktyget finns som plug-in till Maven och kan då avbryta bygget om testtäckningen inte överstiger angiven nivå. Cobertura mäter testteckning på flera nivåer. Om exempelvis en klass använder sig av en metod i en annan klass anses även denna metods kod blivit testad.

3.5.5 Sonar

Ett verktyg för mätning av kodkvalité ur fler aspekter än testtäckning[18]. Exempelvis påpekas om en öppnad resurs inte stängts ordentligt, om specifika typer använts där möjligheten finns att använda generiska typer, felmeddelanden som ignorerats med mera. Testteckning mäts endast på en nivå i Sonar jämfört med Cobertura som mäter testteckning på fler nivåer.

3.6 Automatisk deployment

För att byggservern ska kunna sköta deployande av applikationen till slutmiljö krävs att denna process är helt automatisk. Högnivåtest i Selenium måste dessutom köras mot en deployad applikation vars container ska stängas efter att testerna körts.

3.6.1 Cargo

Ett verktyg för automatisk deployment [19]. Används som plug-in till Maven. Cargo-pluginet kan ladda ner Tomcat från en url, starta tomcat-containern, deploya önskad applikation under vald maven-fas och sedan stoppa containern under en anna angiven fas. Allting sker helt automatiskt. Utan cargo hade användaren varit

tvungen att manuellt eller genom ett egenskrivet script utföra dessa steg vilket hade varit mycket tidsödande.

3.6.2 Ant

Ant är föregångaren till Maven och kan liksom Maven sköta bygget av en applikation[20]. I detta projekt är det dock funktionalitet för att köra kopiera filer samt köra shell-script på en fjärrdator som utnyttjas. Detta möjliggörs av ett tredjepartsbibliotek utvecklat av jCraft [28].

3.6.3 Surefire

Surefire kan hindra exekverandet av tester under den normala testfasen och istället exekvera dessa under angiven fas[21]. Exempelvis är det ofta önskvärt att köra tester på gränssnittet efter att applikationen byggts och deployats på en container.

3.7 Byggserver

Byggservern ansvarar för att bygga hela applikationen och genomföra önskade tester. Dessutom kan byggservern efter eller innan bygget köra andra typer av script, exempelvis shell-script eller ant-script.

3.7.1 Jenkins

Jenkins är en mjukvara som genom ett webgränssnitt överskådligt presenterar resultatet av samtliga byggda projekt[22]. Ett stort antal plugin finns tillgängliga, exempelvis för genomförandet av kodanalys genom Sonar. Jenkins kan polla ett versionshanteringssystem och starta ett bygge vid pushandet av ny kod.

3.8 Molntjänster

Det finns tre typer av molntjänster [29]. Dessa tre skiljer sig när det gäller öppenhet mot applikationer.
Software as a service(Saas) är den minst öppna modellen och erbjuder i princip en färdig mjukvara för hantering av olika system. Användningsområdet för denna typ av molntjänst är standard-system som exempelvis faktureringssystem och hr-system.
Infrastructure as a service(Iaas) är den mest generella tjänsten och erbjuder i princip bara en distribuerad virtuell maskin. Kunden kan installera eget operativsystem och hantera resursen efter eget tycke. Det enda som skiljer från att ha en dedikerad server är att resursen kan flyttas och lagras på ett flertal olika fysiska servrar utan att detta märks för användaren.
Där emellan finns *Platform as a service(Paas)* som kan lagra och exekvera applikationer utvecklade specifikt för plattformen. Härtill hör bland annat molntjänsten Heroku som i dagsläget är anpassat för bland annat plattformarna ruby, python och java.
För detta projekt är det Platform as a service och Infrastructure as a service som är intressanta. Software as a service lämnas där hän då dessa inte är anpassade för deployande av utvecklad mjukvara.

3.8.1 Heroku

Heroku är en molntjänst av typen Paas integrerad med git för kontinuerlig leverans. Genom en anslutning från ett git-arkiv kan en web-applikations källkod levereras till, och byggas på, en heroku-server. I en så kallad Procfile anges vilket sätt Heroku ska använda för att starta applikationen. [4]
Eftersom Heroku är av typen Paas körs applikationen på en fördefinierad stack som inte är synlig för användaren.
Heroku bygger på dynos, processer som kör exempelvis web-applikationer. Fler dynos kan dedikeras till samma applikation som då kan hantera fler anrop snabbare. Detta kallas för att skala upp applikationen. Om en applikation har två dynos dedikerade försäkras att dessa två alltid finns på två olika fysiska positioner vilket minimerar risken för att driftstörningar uppstår trots att en av servrarna går ner.

3.8.2 WebappRunner/JettyRunner

På molntjänster av typen Paas måste uppstartsmetod för applikationen anges. Det skulle då bli mycket omständligt att ange uppstart av en container, deployment på den containern och så vidare. Istället används en runner som genom Maven läggs in i en relativ sökväg och startas med ett enkelt kommando i vilket också filen som ska startas anges. Vid användande av Webapp-runner startas en tomcat-container och med Jetty-Runner startas en Jetty-container.

3.8.3 Postgresql

Heroku använder sig av ett plugin som gör att användaren får tillgång till en databas av typen Postgres. Postgres kan både användas som en delad databas mellan Heroku-applikationerna eller som en dedikerad databas med utökad funktionalitet.

3.8.4 Amazon

Amazon är en molntjänst av typen Iaas. Användaren kan starta virtuella ”instanser” och sedan disponera dessa efter eget tycke[24]. Vid startandet måste operativsystem anges men därefter kan maskinen användas som en vanlig server.
Amazon erbjuder dessutom en mängd plugins, exempelvis en Paas-tjänst för deployande på önskad container.

3.9 Automatisk databasmigrering

För att processen att deploya en applikation utvecklad lokalt på övriga system krävs att även uppdatering av databasen sker automatiskt. Uppdateringar av databasen anges då i en fil som skickas till övriga system tillsammans med källkoden. Vid uppstartandet av applikation i respektive miljö kontrolleras om databasen behöver uppdateras vilket garanterar att applikationen körs på en korrekt konfigurerad databas. Att sköta detta manuellt hade inneburit mycket extrajobb.

3.9.1 Flyway

För varje miljö där applikationen ska deployas krävs en specifik databas. De test som körs mot en databas kan exempelvis inte köras mot samma databs som används i produktion då testen ändrar på innehållet i databasen och kanske måste tömma den på information emellanåt[23].
Kontinuerlig leverans innebär löpande förändringar och så även för databasen. Att utveckla databaserna manuellt med sql-script är mycket tidskrävande men framför ökar riskerna för misstag när samma förändring ska införas på tre olika databaser samtidigt.
Flyway erbjuder funktionalitet för automatisk uppdatering av databaser knutna till applikationen. Dessa uppdateringar anges då i form av ren sql i sql-filer som versionshanteras under projektets källkodskatalog. Flyway erbjuder inte funktionalitet för att rulla tillbaka ändringar som gjorts i databasen med motiveringen att det kan bli mycket komplicerat att rulla tillbaka ändringar som att ta bort en tabell med beroenden och så vidare. Detta bör därför skötas manuellt, exempelvis i en ny migration i Flyway.
Flyway-kommandon kan exekveras genom kommandoraden, som plugin till Maven eller via applikationen. I detta projekt exekveras Flyway genom applikationen. Detta sätt är att föredra då applikationen ansvarar för sin egen databas. Därigenom säkerställs att applikationens version och databasens version alltid är kompatibla.

3.10 Ramverk

Eftersom byggandet av applikationen sker enligt en specifikation i en byggfil spelar det inte stor roll vilken utvecklingsmiljö som används av respektive utvecklare utan detta kan väljas efter eget tycke. De två utvecklingsmiljöer som är mest populära för projekt av denna typ är IntelliJ och Eclipse.
För detta projekt har utvecklingen skett i IntelliJ som, liksom andra utvecklingsmiljöer har stöd för Maven-projekt.

3.10.1 Java Enterprise Edition

Java EE är en specifikation både när det gäller api och runtime-miljö för, framför allt, webb-applikationer. Ee-applikationer körs med hjälp av containrar som i princip fungerar som en jvm[25].

3.10.2 Java Server Faces 2

JSF är ett ramverk för webb-applikationer i java[10]. En xhtml-fil specificerar uppläget på sidan och hämtar värden från ”manage beans” som är skrivna i java-kod.
För varje manage bean specificeras ett scope som avgör livslängde på bönan. De vanligaste scopen är request scoped, bönan lever under en http request, view scoped, bönan lever så länge användaren stannar kvar på samma sida, session scope, bönan lever under hela sessionen samt application scoped, bönan lever hela

applikationens livslängd.

3.11 Blue green pattern

Vid uppdatering av applikationen bör användare fortfarande kunna använda den tidigare versionen fram till dess att den nya versionen är deployad på en server och nåbar. För detta krävs att applikationen ska kunna deployas i två separata miljöer samt funktionalitet för att skifta mellan dessa miljöer [26].

4 Genomförande

4.1 Progress

Första veckan
Till en början var lärokurvan brant. Första veckan ägnades främst åt informationsinhämtning i syfte att skapa större förståelse för projektet.
De områden som kunskap inhmtädes inom var:

- Git för versionshantering.
- Maven för specificering av hur en applikation ska byggas.
- Grundläggande kunskap i jsf för utveckling av webapplikation.

En prototyp för applikationen skapades även första veckan.

Andra veckan
Kunskap om molntjänsten Heroku hämtades och ett exempelprojekt levererades. Efter detta levererades även den tidigare framtagna prototypen till molntjänsten.
Funktionalitet för att reservera en last lades till i applikationen.

Tredje veckan
Fokus låg på att sätta upp en ci-server. Ett flertal problem uppstod i samband med detta(se resultat) men på onsdagen fanns en ci-server med grundläggande funktion för byggning samt leverans ut på molnet. Det har även konstaterats att en rollback på git också generar en rollback på molnet. Dessutom har undersökningar av sessionsöverlevnad trots leverans av ny version av hemsidan undersökts.

Fjärde veckan
Databasfunktionalitet lades till samt funktionalitet för automatisk uppdatering av databasen med programmet Flyway.

Femte veckan
Exekverbara specifikationer lades till i form av Cucumber. Stor tid lades på att felsöka Cucmber-testen som inte kördes vilket visade sig bero på ett s för mycket i ett klassnamn(se Problem).
Även testtäckningsmätning lades till med verktyget Cobertura.

Sjätte veckan
Deployande på Amazons molntjänst genomfördes. Stöd för databas på densamma lades till. Dessutom påbörjades arbete för att införa Hibernate. Detta genomfördes lokalt.

Sjunde veckan
Blue Green Pattern infördes vilket krävde utökad kunskap om ant för att läsa i filer och skriva till filer.

Åttonde veckan
Ant-scriptet förbättrades och gemensam databas infördes för samtliga servrar på molntjänsten. Information om alternativ för sömlösa uppdateringar inhämtades och lades till som framtida förbättring.

4.2 Informationsinsamling

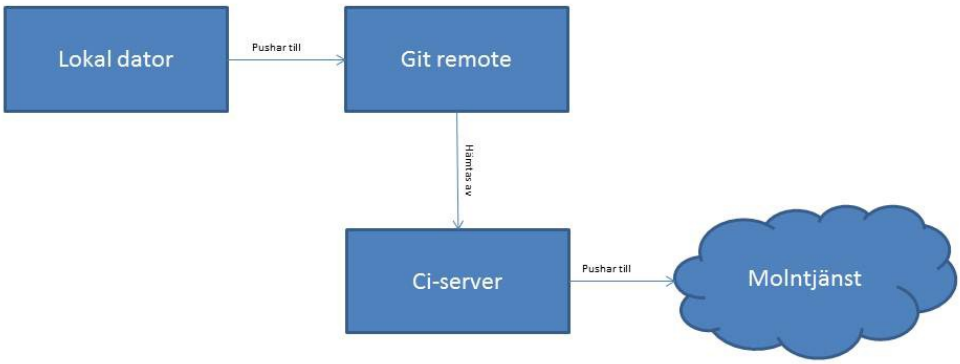
Informationsinhämtningen bedrevs genom sökningar på de verktyg som presenterats av handledare.
Sökmotor: Google.

4.3 Implementation

Nedan följer information om den faktiska implementation som används i dagsläget för att uppnå kontinuerlig leverans.

4.3.1 Allmän Struktur

Den allmänna strukturen för projektet beskrivs av *figur 3.* Observera att två helt oberoende projekt satts upp. Ett projekt avser den applikation som ska levereras till Amazon och ett avser applikationen som ska levereras till Heroku.



Figur 3: Allmän struktur

1 Ny kod pushas till git remote repository.
2 Ci-servern upptäcker ändringen och hämtar ner den senaste koden.
3 Ci-servern bygger den senaste programkoden och testar denan med högnivåtest.
5 Om applikationen kompilerar och tester går igenom pushas koden till molntjänst (heroku) eller levereras till container(Amazon).

4.3.2 Jenkins

Jenkins består av två olika vyer. En för hantering av projektet i vilket applikationen ska levereras till Amazon och en för hantering av projektet där applikationen ska levereras till Heroku.
Herokuvyn består av två Jenkinsprojekt eftersom Heroku kräver att källkodskatalogen ligger i en git root, *se 4.3.4.* Ett av Jenkinsprojekten innehåller byggandet och enhetstesterna för själva applikationen och ett innehåller högnivåtester och ett eftersteg som pushar applikationen till Heroku via git. Båda dessa projekt pollar versionshanteringssystemet och projektet som bygger applikationen triggar dessutom testprojektet. Detta gör att oavsett om applikationen uppdateras eller om testerna uppdateras så byggs det som uppdaterats.
Amazonvyn består av ett projekt som sköter byggande, testning och deployande av applikationen. Detta gör att applikationerna och testerna blir till en enhet som aldrig kan divergera i version. Leveransen i Amazonprojektet sker genom att ett ant-script exekveras i ett eftersteg.

4.3.3 Migration av databasen

För att migrera databasen används FlyWay. Filer innehållande rena sql-kommandon sparas i en katalog namngiven db.migration. FlyWay anges som ett beroende i Maven och från programkoden anges först en

anslutning till databas för FlyWay och därefter anropas metoden migrate(). Alla ändringar som görs i databasen anges i sql-filer och samtliga databaser knutna till applikationen kommer att ta del av uppdateringen.
Eftersom det är önskvärt att endast köra migreringar vid uppstart av applikationen skapas en ManageBean som annoterats med application scope. Värdet på eager sätts sedan till true och koden i bönan kommer att exekveras när applikationen laddas.

4.3.4 Heroku

Heroku är av typen Paas och relativt lätt att komma igång med och kombinerat med verktyget webapprunner eller jetty-runner kan web-applikationer sättas upp lokalt eller via heroku med några enkla kommandon.
Ett problem med Heroku är att ett projekt som ska levereras till en Heroku-appcontainer måste ligga i en git-root. Ett projekt som består av fler Maven-moduler kan således inte levereras till Heroku direkt från projektets ordinarie git-arkiv. En lösning till detta är att hålla applikationsmodulen i ett separat projekt som placeras i en git-root. Andra moduler får sedan specificera ett beroende till den byggda artefakten. Även om det möjligtvis skulle gå att ladda upp ett multimodulprojekt till heroku och låta Heroku bygga både applikation och tester är detta inte önskvärt eftersom total kontroll över byggmiljön hos Heroku inte kan erhållas.
Det är osäkert om web-app runner och jetty-runner passar vid driftssättande av en web-applikation. Dock är det till dessa applikationer instruktionerna för att leverera en java ee applikation pekar.[3]
Vid användandet av molntjänster som Heroku som bygger applikationen utifrån en specifikation i Maven är det viktigt att i byggspecifikationen(pomen) specificera versionen för beroenden och plugins. Detta för att applikationen ska byggas på samma sätt lokalt som på molntjänsten. Det är också att föredra att de specificerade beroendena och plugins finns i ett allmänt arkiv. I detta projekt har endast bibliotek från maven central använts.

4.3.4.1 Databas och Heroku

Eftersom applikationen som utvecklats ska ha tillgång till en databas både lokalt för testning, på ci-servern för testning och på molntjänsten kan uppgifter om databasen inte specificeras hårdkodad i koden. Istället måste drivrutinen anges som ett Maven-beroende och uppgifter om url, användarnamn och lösenord måste hämtas ut från en miljövariabel. Eftersom Heroku kallar denna miljövariabel SHARED_DATABASE_URL måste den heta så även i system som utvecklaren förfogar över. Denna miljövariabel skrivs på formen: databas://anvnamn:lösenord@url/databasnamn. Efter att miljövariabler har satts och beroendet i Maven specificerats ser koden för att upprätta en databasförbindelse, lokalt eller på Heroku relativt enkel ut:

```
private static Connection getConnection() throws URISyntaxException, SQLException {

        URI dbUri = new URI(System.getenv("SHARED_DATABASE_URL"));
        String username = dbUri.getUserInfo().split(":")[0];
        String password = dbUri.getUserInfo().split(":")[1];
        String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + dbUri.getPath();

        return DriverManager.getConnection(dbUrl, username, password);
    }
```

Figur 4: Databaskoppling

4.3.4.2 Användandet av heroku

Förutsatt att ett projekt versionshanteras med Git och byggs via Maven utförs några enkla steg för att leverera projektet till Heroku.
Först installeras Heroku Toolbelt. Path-variabeln i Windows sätts så att Heroku-kommandon blir tillgängliga via commando-prompten. Därefter används ett kommando för inloggning på Heroku:

```
Heroku login
```

Heroku kräver att sättet som applikationen ska startas explicit specifiseras. Detta anges i en fil som måste heta Procfile och ligga i git-rooten. Vid användning av web-app runnern blir denna fil enkel med den enda raden:

```
web:    java $JAVA_OPTS -jar target/dependency/webapp-runner.jar --port $PORT target/*.war
```

web: anger att applikationen är en web-applikation med ett interface som ska publiceras. Resten av texten anger bara att jar-filen, webapp-runner.jar, ska startas med argumentet port och target/*.war. Port är en miljövariabel som redan satts hos heroku och eftersom endast en war-applikation finns i den relativa sökvägen target/ anges att programmet ska starta samtliga war-filer i katalogen.
Sedan skapas en remote-koppling från ägarens git-arkiv till Heroku.

```
Heroku create –stack cedar
```

Det som återstår är att pusha applikationen till heroku.

```
Git push Heroku master
```

Nu returneras en webb-adress där applikationen presenteras. Denna web-adress kan bytas ut mot en annan under domänen herokuapp.com. Även herokuapp.com kan givetvis bytas ut mot egen domän, antingen genom ett consol-kommando eller genom Herokus hemsida.
Heroku använder begreppet stack för den miljö som applikationen levereras till vid en push från git. Den senaste stacken kallas cedar och befinner sig fortfararande i beta-stadiet. Dock är det först i cedar-stacken som stöd för java-applikationer finns varför denna stack används i detta projekt. Tidigare stackar, aspen och bamboo gav endast stöd för ruby.

4.3.4.3 Projektstruktur

Eftersom Heroku kräver att den applikation som ska byggas har sin src-katalog i rooten för den git-katalog som pushas till Heroku är projektet inte ett multi-modulprojekt. Istället är applikationen och högnivåtestningen två separata projekt.
Eftersom testen i testprojektet är skrivna i Selenium och körs direkt mot en deployad webapplikation kan dessa inte köras under fasen test som de normalt skulle göra. En container startas under Maven-fasen pre-integrationtest och stoppas under post-integrationtest varför testen bör köras under fasen integration-test när container är startad. För detta används verktyget Surefire i vilket anges att testen ska skippas under den normala testfasen men köras under integrations-testsfasen.

4.3.5 Amazon

Amazon är en Infrastructure as a service (IAAS) och erbjuder betydligt fler möjligheter än Heroku men är svårare att konfigurera. Vid användandet av Amazon finns en hel del plugins som kan underlätta deployande av applikationer. Alternativet till detta är att installera en maskin via amazon och sedan konfigurera den precis som en fysisk server men via ssh.

4.3.5.1 Användandet av Amazon

Med hjälp av ett plugin som heter Elastic Beenstalk kan en war-fil enkelt deployas och sedan besökas på en adress *mittnamn.*elasticbeanstalk.com. Dock kräver detta användandet av amazons console, vilket försvårar kontinuerlig leverans och eftersom detta plugin fungerar precis som en Paas erhålls ingen ytterligare frihet jämfört mot Heroku. Av dessa anledningar användes i detta projekt inget plugin av denna typ utan en instans startades via Amazon vilken sedan konfigurerades via ssh på i stort sett samma sätt som en lokal server hade konfigurerats.
I dagsläget ser användandet av Amazons molntjänst ut som följer:
Två linux-instanser är startade via Amazon och konfigurerade med hjälp av ssh via puTTy. Jenkins byggserver ansvarar för deployandet till den Tomcat-container som startats på amazon-instansen. Detta görs via en ant-fil som kör scp direkt mot maskinen.
För att möjliggöra användandet av eget domännamn har ett ”elastic ip” kvitterats ut. Detta fungerar som ett virtuellt ip-nummer som är gratis hos Amazon och som via dns kan bindas till ett domännamn.
Skalning på Amazon kan antingen genomföras genom att begära att den instans som kör webapplikationen får utökat minne och kapacitet eller genom att använda Amazons tjänst för lastbalansering. En ny instans skapas då och läggs till i listan för vilka instanser som lastbalanseraren ska balansera mellan.

4.3.5.2 Databas och Amazon

Att få igång databasen på Amazon gick till på precis samma sätt som på en lokal maskin. Den enda

skillnaden är att ingen grafisk installerare kan användas. Eftersom Flyway används behövs ingen ytterligare arbete för att få en fungerande databas.
Det som dock är önskvärt är att ha en gemensam databas för samtliga instanser. Förslagsvis avsätts en instans för detta ändamål. I dagsläget används en av de befintliga instanserna även till att erbjuda databas för övriga instanser. Anslutningsuppgifter specificeras sedan i miljövariabler så att tester kan köras mot en lokal databas och inte mot den delade produktionsdatabasen. Det som kan ställa till problem vid installation av en gemensam databas är säkerhetsregler, i övrigt fungerar det precis som att installera en lokal databas. Först måste porten öppnas till databasen på den maskin som erbjuder databasen. Detta är enkelt gjort genom att lägga till en säkerhetsregel i Amazon-consolen. Därefter måste säkerhetsinställningarna modifieras för databasen. Detta görs i filen pg_hba.conf som hittas där Postgres installerats. Här anges att anslutningar från ett specifikt ip-nummer eller ett intervall av ip-nummer tillåts. Eventuellt måste även modifiering av filen postgres.conf göras. En rad i denna fil anger varifrån tcp-anslutningar tillåts. I vissa installationer av Postgres är default-värdet localhost. Detta ändras till * för att tillåta tcp-anslutningar från andra datorer. Även om det inte tydligt anges i dokumentation måste databasservern nu startas om.

4.3.5.3 Projektstruktur

På Amazon kan en mer önskvärd projektstruktur användas än på Heroku då Amazon-maskinen fungerar som en inhouse-server. En top-pom specificerar de undermoduler som finns, i detta fall main och test. När ny kod pushas till git byggs hela projektet i den ordning som är specificerad i top-pomen. I detta projekt byggs först modulen main för att få fram en ny version av produktionskoden. Därefter byggs modulen test som verifierar funktionalitet. Om testerna passerar körs ett ant-script som levererar produkten till molntjänsten. Allt detta ligger i ett Jenkins-projekt. Varför denna metod är att föredra är att produktionskod och testkod alltid körs och pushas ihop. Om dessa är uppdelade i två projekt och både testkod och produktionskod uppdaterats under samma utvecklingsfas kan problem uppstå eftersom dessa måste levereras till byggservern separat.

4.3.6 Maven

Den fil som definierar bygget för Maven, pom.xml, ser likadan ut på Heroku och Amazon. Dock krävs ett plugin, webapp-runner endast på Heroku men eftersom detta är ett plugin som på Heroku definieras i en separat fil orsakar detta inga problem på Amazon utan ignoreras helt.

Viktiga delar i maven-pomen:

Automatisk deployment på tillfällig Tomcat-container

Detta genomförs eftersom applikationen måste vara deployad på en container för att seleniumtesterna ska kunna köras.

Först anges vilken typ av container som ska användas samt om den är befintlig eller om maven ska sköta även installation och nedladdning. I detta fall anges endast en url till en zip-fil för maven som då automatiskt laddar ner och installerar containern under projektets bygg-root.

```
<configuration>
            <container>
                <containerId>tomcat7x</containerId>
                <zipUrlInstaller>
                   <url>http://www.apache.org/dist/tomcat/tomcat-7/v7.0.27/bin/apache-tomcat.zip</url>
                </zipUrlInstaller>
            </container>
```

Figur 5 Installerare för Tomcat

Sedan anges hemkatalogen för servern så att Maven vet var senare angivna artefakter ska deployas.
Ingen absolut sökväg anges utan projektets bygg-root specificeras följt av id:t på containern som angivits ovan.

```
                    <configuration>
                        <home>${project.build.directory}/tomcat7x</home>
```

Figur 6Ange installationsfolder

Därefter anges vad som ska deployas. Då det är nödvändigt att den artefakt som ska deployas finns tillgänglig är denna dessutom angiven som ett beroende tidigare i pom-filen.

```
                        <deployables>
                            <deployable>
                                <groupId>se.ndi09mlf.exjob</groupId>
                                <artifactId>LoadPlannerMain</artifactId>
                                <type>war</type>
                            </deployable>
                        </deployables>
                    </configuration>
                </configuration>
```

Figur 7 Ange artefakt

Till sist anges de exekveringsåtgärder som behövs. Containern ska startas innan integrationstesten och avslutas efteråt.

```
                <executions>
                    <execution>
                        <id>start</id>
                        <phase>pre-integration-test</phase>
                        <goals>
                            <goal>start</goal>
                        </goals>
                    </execution>
                        <execution>
                            <id>stop</id>
                            <phase>post-integration-test</phase>
                            <goals>
                                <goal>stop</goal>
                            </goals>
                        </execution>
                </executions>
            </plugin>
```

Figur 8Exekveringsåtgärder

Webapp-runner

För att en webapprunner ska finnas tillgänglig när applikationen ska startas på molntjänst anges detta som ett plugin. Maven skapar då en katalog där webapp-runnern placeras och som sedan anges vid uppstart. Efter att ha gjort en maven-packetering av en pom med nedanstående kod finns en webapp-runner som startas med kommandot java –jar target/dependency/webapp-runner.jar *relativ-sökväg-till-war*.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals><goal>copy</goal></goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>com.github.jsimone</groupId>
                        <artifactId>webapp-runner</artifactId>
                        <version>7.0.22.3</version>
                        <destFileName>webapp-runner.jar</destFileName>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Figur 9 Webapp runner

4.3.7 Applikationen

Applikationen är ett lastbokningssystem. Funktion för inloggning finns. Väl inloggad kan användare registrera laster och sedan boka tillgängliga laster som sedan presenteras i en lista för den användaren. Appliaktionen består i dagsläget av en xhtml-sida för varje användarsida. Varje av dessa sidor har en motsvarande böna. Dessutom finns en böna med sessionsscope som håller reda på inloggad användare samt en applikationsböna som genomför eventuella uppdateringar av databasen när applikationen startas för första gången.
På grund av problem med datatable är även den sida där laster kan bokas session scoped, se problem 4.5.10.

4.3.8 Cucumber

Cucumber används i projektet ihop med de Selenium-tester som kör mot en deployad hemsida. Det finns helt klart ett syfte med denna typ av exekverbara specifikationer efter som de ökar läsbarheten så pass mycket att ej programmeringskunniga bör kunna ta del och förstå dessa specifikationer.
Klartexten i feature-filen har ingen inblandning av programmerings-syntax. En feature-fil kan se ut så här:

```
Feature: ReserveLoad

Scenario: Reserve a load
    Given a load with content coal
    When you reserve it
    Then the load will be reserved for you
```

Figur 10 Cucumber Feature

Varje steg kopplas ihop med en bit kod som översätter klartexten till program-kod.
Dock har applikationen i detta projekt ingen riktig beställare vilket gör att en del av syftet med testerna försvinner lite. Dock förenklar det även för programmerarna att ha klartext över vad det är som ska testas, vilken funktionalitet.

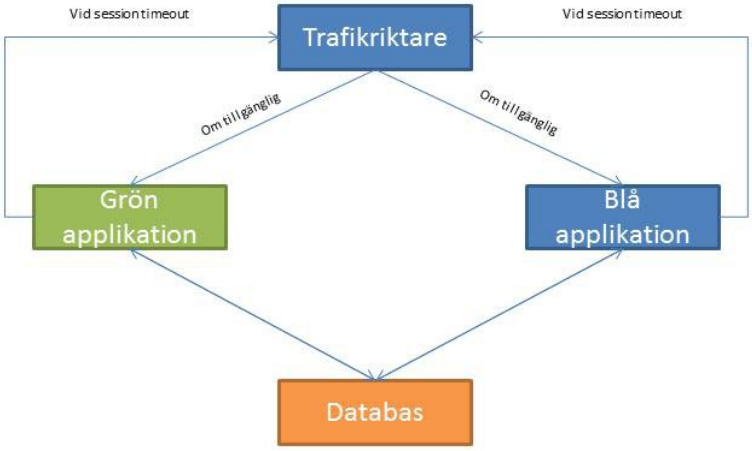
4.3.9 Testning och testteckning

I dagsläget används två typer av tester. Den första typen är enhetstestning. Dessa testar funktionaliteten av enskilda klasser och ska gå fort att köra, då de körs ofta, varför exempelvis en riktig databas inte bör användas.
Den andra typen av tester är de Selenium-tester som testar funktionaliteten på gränssnittsnivå.
Tyngdpunkten vid testandet ligger på enhetstesterna som verifierar funktionen. Fowler [6] talar i sin artikel om testpyramiden där han tar upp att end to end-testning, exempelvis gränssnittstest bör hållas till ett minimum och att om ett sådant test fallerar beror detta oftast på för dålig enhetstestning. End to end-test är också bräckliga mot förändringar i gränssnittet och långsamma då de använder sig av en driver för att

simulera användarinteraktion.
Ett försök att införa Cobertura för att mäta testteckning har genomförts men detta gav upphov till en del problem,*se 4.4.8*. Dessa problem samt att den siffra som erhålls vid mätning av testteckning egentligen inte säger så mycket i sig orsakade att Cobertura uteslöts ur projektet. Cobertura mäter hur många rader i produktionskoden som exekveras av ett test. Dock säkerställs inte ens att något test utförs i testmetoden och testtäckningssiffran ger en dålig bild av kvalitén på testerna.
Ett bättre sätt att mäta kodkvalité är Sonar som dels mäter testtäckning men även andra aspekter av kodkvalité. I dagsläget körs Sonar som ett plugin till Jenkins och efter varje bygge av applikationen erhålls en rapport. På detta sätt kan inte heller dålig kodkvalité stoppa en release. Kodkvalité är viktigt men den kan förbättras under tiden en produkt är släppt till kund och funktionalitet i produkten bör vara så så pass vältestad att dålig kodkvalité inte ska kunna betyda ej fungerande produkt så länge testen går igenom.

4.3.10 Blue Green Pattern

Den lösning på problemet med blue green pattern som används idag är att slutanvändare som vill logga in på applikationen alltid först måste besöka en enkel html-sida, trfikriktare, på en apache-server, *se Figur 11*. Denna sida modifieras så att den alltid skickar besökaren vidare till den instans som för tillfället är aktiv. Sessioner kan överleva maximalt 30 minuter, därefter skickas besökaren till ovan nämnda html-sida för att denna ska kunna avgöra vilken instans användaren ska logga in på. Detta gör att vid omdirigering av trafiken till en ny instans(eller instansgrupp bakom lastbalanserare) kan den gamla instansen ha aktiva sessioner max 30 minuter efter detta. Därefter kan denna ”gamla” instans uppdateras utan att användare blir drabbade.
För att uppnå detta krävs en del ant-script. Dessa ant-script måste, vid uppdatering av applikationen, kolla vilken instans som inte har några besökare, stänga av containern på denna server, uppdatera denna applikation, starta containern, vänta på att sidan åter ska svara efter uppdatering, ändra i html-filen på apache-servern som dirigerar användare till rätt sida så att denna nu pekar på den nyligen uppdaterade applikationen. I dagsläget genomförs dessa steg i ant genom anrop till fristående ant-targets vilket innebär att samma kod, med minimal modifiering, skulle kunna användas om fler instanser skulle önskas. Redirect ska då också ske till respektive lastbalansera för de olika färgerna istälelt för direkt till de körande instanserna.
Detta mönster kräver även funktionalitet för att logga ut användaren efter 30 minuter men default-implementationen för sessions i jsf är att sessionerna får förnyad längd så fort användaren av hemsidan är aktiv. Problemet löses genom att explicit sätta en sessionsvariabel vid inloggande med namnet time. Time får värdet av serverklockan när en användare påbörjar sin session och vid varje sidnavigering anropas ett filter som, med hjälp av denna variabel, räknar ut om användaren varit inloggad för länge och skickar i så fall användaren till startsidan för automatisk avigering till rätt server. Filtret mappas till samtliga sidor som ligger efter inloggningsrutan och körs således varje gång en inloggad användare navigerar på sidan. Mappningen görs i web.xml. Detta medför ett problem och det är att användaren kan ange urlen till inloggningssidan på ”fel” server och har då möjligheten att logga in på denna, se 4.4.12
Denna lösning har två tillkortakommanden. Dels skapar detta en mindre användarvänlig applikation för användaren då dessa endast kan vara inloggade 30 minuter i sträck. Dels betyder detta att applikationen max kan uppdateras en gång var 30e minut (dock kan denna tid ändras efter tycke).
Viktigt är att den fil som avgör vilken instans som är aktiv inte versionshanteras. I detta fall skulle inte deployment ske varannan gång på varannan instans då uppgifter om vilken instans som är aktiv skulle skrivas över vid uppladdning av nya källfiler.



Figur 11: Green Blue Pattern

4.4 Problem

4.4.1 Helhetskonceptet

Till en början var det svårt med förståelsen för hela konceptet. Frågan är då om problemet ska brytas ner i mindre delar eller om fokus ska läggas på helheten. I mitt fall fick jag helheten förklarad av handledare innan själva implementationen delades upp i mindre delar.

4.4.2 Problem vid installation av Jenkins

Installationen av Jenkins på linux-maskin orsakade stora problem.
Ett problem som uppstod vid användandet av Jenkins på linux ubuntu var att jenkins inte hade tillgång till de ssh-nycklar som angivits. Problemet bestod i att Jenkins måste äga katalogen .ssh där nycklarna ligger. För att komma förbi problemet som var mycket tidsödande då ingen bra guide hittades trots tydligt felmeddelande, var att kopiera in .ssh med tillhörande nycklar under jenkins hemkatalog (i detta fall /var/lib/jenkins på grund av installation med apt-get) och sedan ange ägare jenkins (chown). Efter att detta gjorts kvarstod problemet utan ytterligare felmeddelande i jenkins console. Problemet var nu att .ssh-katalogen hade för generösa rättigheter vilket gör att ubuntu bortser från nyckeln då denna är osäker. Genom att sätta ner rättigheterna till 700 löstes problemet och ett lyckat bygge i jenkins genomfördes.

Ett annat problem var att köra Selenium från jenkins-servern. Felkoden sade att ingen display var angiven vid försök att komma åt webläsardrivrutinen. Problemet löstes genom att installera ett plugin till jenkins som heter xvnc och som bistår med funktionalitet för att öppna en webläsare.

Ytterligare ett annat problem bestod i att Selenium inte kunde köra sina tester vid deployandet på jenkins. Anledning var att sökvägen som var angiven till hemsidan som skulle testas var localhost:8080. Detta fungerade lokalt när websidan deployades med webapprunner eftersom denna deployar på just localhost:8080. Vid deployandet från jenkins kunde inte webapprunnern användas då den inte enkelt går att stänga av med ett kommando. Istället deployas hemsidan på en tillfällig tomcat-server och sökvägen till applikationen blev istället localhost:8080/*applikationsnamn.*

4.4.3 Problem vid installation av databas på Heroku

Vid installation av databas på Heroku fanns mycket knapphändig dokumentation att tillgå. Ingen dokumentation förklarade att databasdrivrutinen skulle specificeras som ett beroende i Maven.

4.4.4 Flyway

För att kunna använda flyway både lokalt, på ci-server samt på molntjänst krävs att inställningar om

användarnamn och lösenord specificeras i miljövariabler. Ett problem som uppstod vid angivandet av dessa var att de måste anges med stora bokstäver.
Till en början installerades Flyway som ett maven-plugin. Detta skapade en mängd olika problem som att migrationen inte fungerade om maven startades med kommandot clean install utan endast install måste anges.
Dessutom var det stora problem med att få igång FlyWay på molntjänsten samt att läsa ut miljövariabler till maven på molntjänsten. Efter att detta tillvägagångssätt ändrats till att istället från programkoden invokera önskade ändringar fungerade FlyWay ganska snart på både lokal dator samt på molntjänst. Dock uppstod problem med autentiseringen på ci-servern.
Problemet denna gång var att en tidigare version av postgresql redan var installerat på systemet varför port 5432 som är standard var upptagen. Den nya versionen lade sig då på port 5433 vilket inte var fallet på övriga system där databasen skulle migreras.

4.4.5 Skillnader hos operativsystem

Vid ett tillfälle märktes tydligt att olika operativsystem kan orsaka problem. Dock är det möjligt att detta är en fördel vid användandet av molntjänst eftersom det ofta är oklart vilket operativsystem som används på montjänsten, alternativt att ett operativsystem används som det är svårt att få tillgång till i den lokala utvecklingsmiljön.
Det som hände vid detta tillfälle var att en refaktorering genomfördes på en klass som hette WrongPassWordException som bestod i en namnändring till WrongPasswordException. Dock gick det fel på vägen mellan lokal dator och ci-server. Ändringen på källfilen pushades dock aldrig upp till github, förmodligen beroende på att windows-versionen av git inte såg detta som någon ändring då windows i grunden inte är case sensitive.

4.4.6 Cucumber

Cucumber är i stort enkelt att komma igång med. Dock uppstår ett problem i kombination med surefire(se Teori). För att Surefire ska hitta den runner som startar Cucumber-testen krävs att denna klass har ordet Test i slutet eller början. I det här fallet var RunCukeTests angivet som namn vilket gjorde att Cucumber-testen aldrig kördes. Efter att s:et tagits bort från namnet avhjälptes problemet.

4.4.7 Cobertura på molntjänst

Heroku bygger applikationer utan hänsyn till test genom att ange växeln `-DskipTests=true`. Detta medför att cobertura stoppar bygget eftersom testtäckningen blir 0%.
En allmän och fin lösning på detta hade kunnat vara att läsa av om maven körts med växeln `-DskipTests=true` och i så fall inte låta cobertura-testerna stoppa bygget. Dock finns det ingen dokumentation om hur detta skulle genomföras. Profiler är inte heller en möjlighet eftersom det inte går att styra över med vilka växlar Heroku kör Maven. Ett tredje alternativ skulle kunna vara att använda ett plugin i Jenkins som sköter kontrollen men problemet då är att det inte går att testa lokalt först om testtäckningen är tillräcklig.

4.4.8 Amazon

Upstarten av kontinuerlig levernas mot Amazons molntjänst orsakade ett antal problem som dessutom var svåra att debugga eftersom osäkerhet fans om orsaken till problemet låg hos själva molntjänsten eller inte.
Det första problemet uppstod när scp skulle användas mot Amazonmaskinen. Det kommando som användes var på formen:
scp *katalognamn* username*@amazonmaskin.com*:/
Felmeddelandet som erhölls sade att rättigheter inte fanns till det katalognamn som angetts. Efter ett tags felsökande fanns att detta felmeddelande inte avsåg rättigheterna på den lokala katalogen utan att rättigheter inte fanns att skapa samma katalog på amazonmaskinen. Detta berodde på att katalogen försökte skapas i användarrooten eftersom ingen ytterligare sökväg angivits. Genom att ange följande kommando istället rättades felet till:
scp *katalognamn* username*@amazonmaskin.com*:relativ/sökväg/
Under felsökandet av nyss nämnda problemet testades också användandet av WinSCP men det visade sig att överföringshastigheten var så pass låg att det blev oanvändbart.

Ett annat problem uppstod vid installationen av den Tomcat-container som var tänkt att användas som container för webapplikationen. De filer som behövdes fördes över och korrekt kommando angavs för uppstart av container. För att verifiera att uppstarten gått som det ska angavs sedan en address till maskinen vilket, när en körande tomcat finns, bör presentera Tomcats index-sida. Dock visades inte denna sida och det var nu svårt att avgöra vad felet berodde på. Miljövariabler på maskinen var satta korrekt och containern borde varit igång. Tillslut, efter sökande på internet fanns att problemet berodde på säkerhetsinställningar på Amazon.[7] Detta rättades enkelt till genom att via Amazon-consolen ange en säkerhetsregel som tillät inkommande tcp-anslutningar.
Ett tredje problem uppstod efter att applikationen redan blivit deployad och uppdaterad ett antal gånger på amazon-maskinen. Efter att ett antal, relativt stora, förändringar genomförts i applikationen fungerade plötsligt inte anslutningen till databasen. Problemet löstes genom omstart av tomcat vilket skapar frågor om orsaken till problemet samt om tomcat bör startas om vid varje deploy, vilket skulle orsaka betydligt större ”downtime”.

4.4.9 JSF 2, datatable och request scope

I teoridelen beskrivs de olika scope en böna kan ha. God sed bör vara att alltid ange scope som är så små som möjligt, ge bönan en så kort livstid som möjligt, för att undvika att ”förorena” sessionen och spara onödig information. Dock uppstår problem vid användandet av datatable och request scope i ett multianvändarsystem. Om en datatable populeras enligt information i databasen och denna information uppdateras från annat håll än den aktuella hemsidan kommer innehålet i datatablen och det som visas på sidan inte stämma överens. Om då, som i denna appliaktion, en enskild rad i datatablen ska kunna anges(i detta fall för bokning och med hjälp av en knapp) kommer effekten bli att fel rad anges eftersom knapptryckningen kommer ange en rad i form av ett heltal och vid klickandet kommer datatablen populeras på nytt med den nya informationen. Fel last kommer därför att bokas åt användaren. Detta är ett känt problem som nämns i boken Core Java Server Faces [9].
Lösningen i detta projekt har blivit att ange bönan som innehåller datatablen som session scoped. Ett alternativ hade varit att ange scopet som view scoped, se teori, men problemet är att alla klasser som berörs då måste implementera serializable vilket ibland kan vara svpårt om tredjepartsbibliotek används.

4.4.10 Sessioner och kontinuerlig leverans

Eftersom sessionsinformation sparas på den server som användaren för tillfället är uppkopplad mot kan detta innebära problem vid kontinuerlig leverans mot molntjänst. Vid skalning av molntjänst används flera instanser och varje http-request riktas om mot den server som för tillfället har minst last. Detta löses relativt enkelt genom att låta servrarna använda så kallade sticky sessions. Det innebär att användare som startat en session mot en server fortsätter kommunicera med just den servern under hela sessionen.
Det stora problemet uppstår när blue green pattern ska införas(se teori). Vid införande av detta mönster önskas nämligen just att föra över samtliga användare till en anna server för att möjliggöra uppdatering av den för tillfället oanvända servern. En lösning skulle kunna vara att externialisera sessionerna, det vill säga, att spara all sessionsinformation på en extern server. Problemet med denna lösning uppstår framför allt i kombination med jsf. Vanliga php-sessioner kan lagra nyckel- och värdepar. Denna information kan lätt sparas i en databas även om applikationen blir aningen långsammare. Jsf har en mer komplex användning av sessioner då hela bönor markeras med ett visst scope. De största problemen uppstår vid användandet av datatables, se 4.5.10. Användandet av externa sessioner skulle innebära att all denna information skulle behöva lagras i en databas.
I dagsläget avslutas alla sessioner istället efter 30 minuter och användaren loggas då ut. Detta ger en möjlighet att omdirigera användaren till den nya servern och alla sessioner på den gamla servern avslutas(se implementation).

4.4.11 Explicit angivande av url till inloggningssida

Eftersom sessionsinformation endast finns tillgänglig när en användare är inloggad på en sida kan ett problem uppstå om användaren explicit anger en url till inloggningssidan i adressfältet till den server som för tillfället inte är den aktiva. Användaren har då möjlighet att logga in på denna server och problemet som uppstår är att användaren då kan loggas ut innan de 30 minuter som användare ska kunna vara inloggad på sidan. Detta sker om en ny uppdatering görs av applikationen.
Om användare däremot alltid loggar in via startsidan händer inte detta problem.
En lösning till rpoblemet skulle kunna vara att det från startsidan skicaks med information till

inloggningssidan om att användaren kommer dit på ett korrekt sätt. Om denna variabel dock inte är satt skulle användaren kunna skickas tillbaka till startsidan och på så sätt komma till korrekt server. Problemet kan dock fortfarande inträffa om användaren tar upp en inloggningssida och sedan väntar med att logga in. Detta skulle kunna lösas genom att ha en session som startas när användaren kommer till inloggningssidan och om exempelvis ingen inloggning skett inom ett visst antal minuter skickas användaren till startsidan. Denna tid får då räknas in i den minimumtid som krävs mellan varje ny uppdatering av applikationen.

4.4.12 Cachning av statiska html-sidor

Ett problem som är liknande det i 4.4.11 upptäcktes när Google Chrome testades som webb-läsare. Originalinställningen i Chrome är att cacha statiska html-sidor vilket skapade problem när användaren loggades ut efter att sessionen avbrutits. Användaren hamnade då på Apache-servern men skickades tillbaka till den sida som denne kom ifrån trots att länken i html-dokumentet på Apache-servern ändrats. Detta berodde på att Chrome hade cachat html-sidan och inte laddat in den senaste versionen.
Problemet löstes genom att läga till följande html-taggar innan den tag som skickade användaren vidare, *se figur 12.*

```
<META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-CACHE">
<META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-STORE">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META HTTP-EQUIV="Expires" CONTENT="-1">
```

Figur 12: Html-kod för att undvika cachning

Dessa taggar gör att ingen webb-läsare ska kunna cacha denna sida utan tvingas ladda om den varje gång.

5 Resultat

1.Vilka typer av molntjänster finns och vad skiljer dessa åt?

De tre huvudgrupperna av molntjänster är Software as a Service, Platform as a Service och Infrastructure as a Service. Det som skiljer dessa åt är framför allt möjligheten till manuell konfiguration. *Se 3.8.*

2.Vilken typ samt vilken konkret molntjänst skulle passa bäst för det aktuella projektet?

De två typer av molntjänster som är intressanta för detta projekt är Platform as a Service och Infrastructure as a Service. De två konkreta molntjänster som testats är Heroku och amazon där Amazon hittills framstått som den molntjänst som bäst är lämpad till ett projekt av den här typen, *se 8.2.*

3.Vilken typ av verktyg krävs för att få en automatiserad process för leverans till molntjänst och hur ser detta ut ur utvecklarens perspektiv?

I teoridelen tas en mängd verktyg upp som möjliggör kontinuerlig leverans. Bland dessa finns verktyg för automatiskt bygge, automatisk databasmigration, automatiska tester med mera.
För användning ur användarperspektiv, se *bilaga 1.*

4.Hur kan uppdatering av applikationen göras utan att detta märks för användare?

Genom ett så kallat blue green pattern kan kan en ny version deployas på en server som för tillfället inte har några aktiva sessioner. När servern sedan kör den nya versionen av applikationen och svarar på anrop slussas användare över till denna. För implemenation av blue green pattern, *se 4.3.10*

5.Hur kan specifikationer anges så att utvecklare och beställare kan komma överens om entydiga krav?

Cucumber är ett av många verktyg som gör att krav skrivna i klartext kan exekveras i form av högnivåtest, *se 4.3.8.*

6 Rekommendationer

6.1 Säkerhet

Säkerheten i applikationen följer inte angivven standard för jsf-applikationer. En förbättring som förmodligen skulle behöva genomföras vid lansering av applikaitonen är ett bättre säkerhetssystem.

6.2 Utökade tester

I dagsläget gör applikationens litenhet att alla de tester som bör finnas i ett stort projekt inte finns. Enhetstesterna har inte heller full täckning och testar inte samtliga fall. Eftersom produkten inte kommer att användas i någon större skala låg inte fokus på detta utan på att ha exempel på olika typer av tester för verifiera helhetskonceptet.

6.3 Införande av Ajax för en dynamisk upplevelse

Jsf 2 har i sitt grundutförande stöd för ajax för dynamisk uppdatering av hemsidan. Applikationen i detta projekt skulle kunna lyftas genom införandet av ajax. Exempelvis för att rensa fält, dynamiskt uppdatera delar av sidor samt för en renare organisation av projektes struktur.

6.4 Införande av Hibernate

Hibernate är ett bibliotek för java som försöker lösa problemet att ett objektorienterat projekt skiljer sig så pass kraftigt mot en relationsdatabas. Objekt kan persisteras och Hibernate sköter då det underliggande arbetet.
I detta projekt har Hibernate införts lokalt och enhetstest har verifierat funktionaliteten mot en inmemory databas, Hypersonic. Dock har tisdsbrist hindrat att Hibernate införts som databasmappare även mot en riktig databas. Detta kan vara ett framtida förbättrande.

6.5 Åtgärda problem med explicit angivande av inloggningssida

I dagsläget finns en risk att användare loggar in på fel server, se 4.4.12. Under nämnda styckee finns förslag på åtgärder för att undkomma problemet vilka bör implememenderas.

6.6 Skalning

I detta projekt har inte skalning i någon större utsträckning genomförts på grund av kostnader detta medför. Detta är dock en av de största vinsterna med att använda molntjänster och bör därför implementeras eller i alla fall testas.

6.7 Kombinera lokal server med molntjänst

En framtida förbättring skulle kunna vara att kombinera en privat molntjänst som tar en grundbelastning med en publik molntjänst. Fördelen är att ingen löpande kostnad finns för den privata molntjänsten.
En annan fördel är att säkerheten, eller säkerhetskänslan, är större med lokalmolntjänst. Detta skulle kunna utnyttjas för sidor som fungerar på ett liknande sätt som exempelvis Youtube där den största delen användare inte loggar in men där möjligheten finns. En publik molntjänst skulle då kunna ta hand om det ojämna antal besökare som endast vill visa videoklipp och en privat molntjänst administrtörer och de som loggar in på sidan. Många av de mjukvaror som idag används för privata molntjänster använder sig av Amazons API för att möjliggöra denna kombination. Exempelvis kan nämnas Eucalyptus [30] och Debian openStack[31].

6.8 Modifiera blue green pattern

I dagsläget loggas användare ut var 30:e minut för att sedan slussas över till en annan server för att möjliggöra updpateringar av den första servern. Ett mer önskvärt tillvägagångssätt skulle vara att applikationen uppdateras utan att detta är märkbart för användaren. Detta skulle innebära att sessionsinformation måste sparas på ett ställe som samtliga servrar kommer åt.

Ett tillvägagångssätt för att uppnå detta skulle vara att ha samtliga jsf-bönor annoterade med request scope och alltid manuellt spara sessionsinformation i databasen, alternativt implementera HttpSession i en egen klass som sparar all sessionsinformation i en databas. Det skulle innebära en del merjobb för programmeraren och problem när användandet av datatables önskas, *se 4.4.9*. Att använda Tomcats support för Cluster-lösningar är inte ett alternativ då detta går via multicast och ec2 inte stödjer detta.
Externa sessioner ihop med en lastbalanserare borde kunna lösa problemet med sömlösa uppdateringar om de servrar som ska uppdateras innan uppdateringen tas bort från lastbalanseraren så att de inte kan få några anrop, uppdateras och sedan läggas till igen. Ec2 api tools erbjduer funktionalitet för att genomföra dessa steg från kommandoraden.
Problemet med datatables skulle eventuellt kunna lösas genom att ange dessa som ViewScoped och sätta state_saving_method till client för att spara informationen som behövs på klientens dator.
Det problem som då kvarstår är om externa sessioner minskar prestandan alltför mycket.

7 Diskussion

7.1 Allmän diskussion

Detta projekt har varit utmanande då det har handlat om de delar som jag haft minst kunskap om på förhand. Istället för algoritmer och design-mönster, som är det jag tidigare fokuserat på, har det handlat om att få verktyg att fungera och få ihop en modell för leverans. Det har gjort att inlärningskurvan har varit brant.
Att arbeta med java ee stacken skiljer sig mycket mot att exempelvis utveckla en swing-applikation i en lokal miljö. Fokus ligger mycket på de verktyg som används vid byggandet och testandet av applikationen. I detta projekt har en trivial applikation utvecklats mest för att testa tillvägagångssättet att utveckla en applikation varför inga tyngre algoritmer förekommer. Det svåraste var att initialt greppa syftet med olika verktyg och tillvägagångssätt.
Att arbeta i korta iterationer med fokus på en sak i taget passar mig mycket bra. Det är mycket enklare att jobba om man jobbar mot ett tydligt mål.
Väldigt mycket tid lades på att få databasmigrationen att fungera och 90 % av tiden var felsökning. Det är mycket frustrerande att under en halv vecka i princip inte alls komma framåt i projektet. Dock är detta faktorer man får räkna med när projektet handlar om att ta fram en miljö snarare än att utveckla produktionskod. Vid utvecklingen av produktionskod kan ofta problem isoleras och utvecklingen kan fortfarande fortgå. Vid användandet av verktyg krävs det att man gör på tänk sätt och när fel uppstår måste dessa lösas innan man kan gå vidare.

7.2 Linuxkunskaper

Tyvärr hade jag dålig kunskap om linux innan projektets start. Att bestämma att ci-servern skulle ligga på en linux-maskin orsakade en del mer jobb, inte på grund av att det skulle vara mindre passande att ha en ci-server på en linux-maskin, utan för att mina kunskaper var för dåliga om rättigheter och övrig konfiguration på linux.

7.3 Lite information

Det märks tydligt att kontinuerlig leverans, framför allt i kombination med green blue pattern är ganska nytt och det finns lite vedertagna metoder att jobba på. Vid sökning på google efter continuous delivery och session är den översta träffen ett foruminlägg som jag själv har skrivit. Svarat på detta foruminlägg har James Ward som är en av grundarna till Heroku då kunskapen i ämnet inte är så utbrett hos övriga utvecklare.

7.4 Utökning från green blue pattern till sann sömlös uppdatering

Om jag hade haft lite mer tid hade förmodligen kontinuerlig leverans utan nertid och utan att användare måste loggas ut implementeras. Anledningen till att jag inte lade mer fokus på detta sättet från början var att jag var säkrare på att det sätt som används idag fungerar och jag ville ha en fungerande implementation.

7.5 Automatisk eller semi-automatisk leverans

Huruvida processen med att publicera den nya versionen på molntjänsten efter att ny kod pushats upp till versionshanteringssystemet ska vara automatisk eller inte beror på syftet med applikationen. I de flesta fall är det troligtvis önskvärt att ha denna process manuell eftersom att det inte är säkert att en applikation som passerar alla tester bör levereras.
Det är inte heller helt klart om det motsatta är önskvärt, att hindra publicering om någon typ av testdata inte går igenom, exempelvis när en större organisation gör tester på kodkvaliten innan leverans. Att inte publicera ändringar som passerat acceptans-tester på grund av, exempelvis, för låg testteckning eller kodkvalité, är förmodligen ett dåligt system. För användaren av hemsidan märks inte den dåliga kodkvaliten, bara att denna fått ny funktionalitet. Dock bör självklart den typen av problem rättas till i efterhand.

8 Slutsatser

8.1 Heroku eller Amazon

Heroku är en lättanvänd molntjänst av typen Paas om än riktad mot en mindre användargrupp. Eftersom Heroku levererar en färdig plattform måste projektet vara utvecklat på en plattform som stöds samt med fördel hanteras på versionshanteringssystemet Git.
Vid användandet av molntjänst av typen Paas får ofta den lokala utvecklingsmiljön anpassas efter miljön på molntjänsten. Om exempelvis miljövariabler som läses ut i run-time är angivna enligt ett mönster på molntjänsten måste dessa anges på samma sätt i den lokala miljön för att möjliggöra utläsning i run-time.
Amazon erbjuder en mycket större frihet då man har direkt tillgång till den instans som kör webapplikationen. Dock kräver användandet av Amazon mer konfiguration och är svårare att skala då nya maskiner måste installeras vid skalning. Heroku erbjuder färdig funktionalitet för skalning och skalning uppnås genom ett enkelt kommando.
Heroku stödjer endast databaser av typen Postgres i botten. Dock finns plugin som gör att anna sql-dialekt kan användas men i botten ligger en Postgres-databas. På Amazon finns möjlighet att installera den databas som önskas.
Ett stort problem med Heroku är att den pom som specificerar bygget av själva applikationen måste ligga i git-rooten. Detta gör att multimodulprojekt inte kan användas. Istället används två separata projekt, ett för applikationen och ett för högnivåtester. Detta skapar en betydligt mer rörig bild av och sämre översikt.
Med den konfiguration som används i dagsläget går det mycket långsamt
Skalning är en viktig funktion vid användandet av molntjänst. Att sömlöst skapa möjlighet att hantera mer last. För Heroku skalas applikationer lätt genom att skriva ett enkelt kommando i consolen. Dock skapar detta problem vid användande av sessioner då sessionsinformation eventuellt inte är känd när ett anrop görs då dena information inte är delad över hela applikationer. James Ward, en av utvecklarna till Heroku, föreslår att ett plug in till containern bör användas för att hantera sessionsinformation på en extern databas[27].
I Amazon erhålls skalning genom att manuellt starta en ny instans som en kopia av den gamla och sedan placera dessa bakom samma lastbalanserare. Användandet av sticky sessions gör att sessionsinformation alltid är känd av den server som hanterar anropet.
Detta sammanställt gör att Amazon i detta projekt erbjudit en något bättre upplevelse ur användarperspektiv. Applikationens litenhet gör dock att slutsatser rörande uppträdande vid driftssättning är svåra att dra.

8.2 Kontinuerlig leverans

Att regelbundet leverera en produkt tillför värde både för kunden som regelbundet kan se och utvärdera ny funktionalitet för produkten, samt för utvecklare som snabbt kan få feedback på utförda ändringar. Missförstånd mellan beställare och utvecklare upptäcks och kan snabbt rättas till. Att ha processen automatiserad minskar riskerna för att den mänskliga faktorn ska försena leveransen.

8.3 Kontinuerlig leverans och sessioner

Det finns inget självklart tillvägagångssätt för att kunna leverera en uppdatering till en applikation utan nertid men ändå få sessionerna att överleva. Antingen måste sessionsinformationen på något sätt externialiseras eller så måste användare loggas ut regelbundet.

8.4 Maven

Användning av fler olika system under ett projekt kräver användning av ett extern byggverktyg där bygget otvetydigt specificeras.

8.5 Externa verktyg

Att jobba med externa verktyg kräver ofta mer tid än beräknat. Ofta kan en dag gå åt att lösa ett problem som uppstått på grund av felaktig konfiguration. Vid användandet av molntjänst kommer dessutom ytterligare en faktor in och felsökning blir än mer komplex.

8.6 Heroku och autenticering

Att exekvera applikation i en runner-container minskar möjligheterna till konfiguration av containern. Detta är en känd nackdel med att leverera en produkt till en molntjänst och i detta projekt påverkade detta utformandet av autentiseringsfunktionaliteten. För en web-applikaiton som körs ”in-house” används ofta containerns så kallade realm för att enkelt och säkert autentisera användare. Dock krväver detta konfiguration av servern för att tala om för denne var den ska leta efter registrerade användare. I och med att applikationen körs via en runner finns inga installerade konfigurationsfiler för servern varför användandet av realmer blir krångligt om möjligt. Istället används i nuläget i projektet en böna med sessions-skop som håller reda på inloggad användare. På varje sida görs en kontroll av inloggad användare. Är detta värde null skickas besökaren till inloggningssidan. Vid inloggning kontrolleras angivna värden mot databasen som i detta fall lagrar användare.

8.7 Selenium

Selenium har föredelen att testen som skrivs med hjälp av Selenium testar funktionalitet på allra högsta nivå, användarnivån. Det finns inget steg utanför som kan ställa till det utan det Selenium testar är det som användaren upplever. Dock gör detta att Selenium är mycket nära kopplat till namn och utseende på gränssnittet. Tidigt in i designen av en hemsida kan stora förändringar ske. Exempelvis flyttas sidor, utseende på menyer ändras och så vidare. Vid varje ändring av den typen måste då även Selenium-testerna skrivas om för att matcha utseendet.

9 Referenser

1) Jez Humble and David Farley, Continues Delivery, 2011
2) *Webapp Runner – Apache Tomcat as a Dependency*, James Ward, http://www.jamesward.com/2012/02/15/webapp-runner-apache-tomcat-as-a-dependency (besökt 2012-04-10)
3) Getting started with Spring MVC Hibernate on Heroku/Cedar, Heroku Devcenter, https://devcenter.heroku.com/articles/spring-mvc-hibernate (besökt 2012-04-10)
4) Cloud application platform, Heroku, http://www.heroku.com/(besökt 2012-04-10)
5) Mary Poppendieck, Lean Leaders Workshop, presentation, 2012
6) *TestPyramid,* Martin Fowler, http://martinfowler.com/bliki/TestPyramid.html, (besökt 2012-04-14)
7) *How to install tomcat on aws ec2 instance with ubuntu,* Steffen Opel, foruminlägg, http://stackoverflow.com/questions/9163185/how-to-install-tomcat-on-aws-ec2-instance-with-ubuntu (besökt 2012-05-07)
8) *INVEST in good stories and SMART tasks,* Bill Wake, http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/ (besökt 2012-04-02)
9) David Geary, Cay Horstmann, Core Java Server Faces, Third edition, 2010
10) Ed Burns, Chris Schalk, Java Server Faces The Complete Reference, 2006
11) *Scrumguide,* Ken Schwaber och Jeff Sutherland, http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20SE.pdf (besökt 2012-04-02)
12) *Welcome to apache Maven,* Apache, http://maven.apache.org/ (besökt 2012-04-04)
13) *About Git,* Git, http://git-scm.com/about (besökt 2012-04-04)
14) *Getting started*, jUnit, http://junit.sourceforge.net/#Getting *(*besökt 2012-04-04*)*
15) *What is Selenium*, Selenium, http://seleniumhq.org/ (besökt 2012-04-11)
16) *Cucumber,* Cucumber, https://github.com/cucumber/cucumber/wiki/ (besökt 2012-05-01)
17) *What is Cobertura,* Cobertura, http://cobertura.sourceforge.net/ (besökt 2012-05-01)
18) *Sonar,* Sonar, http://www.sonarsource.org/ (besökt 2012-05-01)
19) *Cargo,* Cargo, http://cargo.codehaus.org/Home (besökt 2012-04-09)
20) *Welcome Apache Ant,* Apache, http://ant.apache.org/ (besökt 2012-05-15)
21) *Maven Surefire Plugin*, Apache Maven Project, http://maven.apache.org/plugins/maven-surefire-plugin/ *(*besökt 2012-04-19)
22) *Meet Jenkins,* Jenkins https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (besökt 2012-04-16)
23) *Why do you need a database migration tool?*, Flyway, http://code.google.com/p/flyway/wiki/WhyDatabaseMigrations (besökt 2012-04-23)
24) *Amazon Elastic Compute Cloud,* Amazon, http://aws.amazon.com/ec2/ (besökt 2012-05-07)
25) *Your First Cup,* Oracle, http://docs.oracle.com/javaee/6/firstcup/doc/gcrky.html#gcroc (besökt 2012-04-04)
26) *BlueGreenDeployment,* Martin Fowler, http://martinfowler.com/bliki/BlueGreenDeployment.html (besökt 2012-05-14)
27) *Using mongoDb for a Java Web App's HttpSession,* James Ward http://www.jamesward.com/2011/11/30/using-mongodb-for-a-java-web-apps-httpsession (besökt 2012-04-19)
28) *Jcraft,* Jcraft, http://www.jcraft.com/c-info.html (besökt 2012-05-07)
29) *IaaS vs. PaaS vs. SaaS*, Jeff Caruso, http://www.networkworld.com/news/2011/102511-tech-argument-iaas-paas-saas-252357.html (Besökt 2012-04-04)
30) *What is Eucalyptus,* Eucalyptus, http://www.eucalyptus.com/learn/what-is-eucalyptus , (besökt 2012-05-24)
31) *OpenStack,* Debian, http://wiki.debian.org/OpenStack, (besökt 2012-05-24)
32) *Clustering/Session Replication How To,* Apache, http://tomcat.apache.org/tomcat-6.0-doc/cluster-howto.html (besökt 2012-05-25)

10 Bilagor

1. Användning ur utvecklarens perspektiv