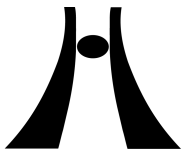
Beteckning:_____



HÖGSKOLAN
I GÄVLE

Akademin för teknik och miljö

Kontinuerlig leverans till molntjänst

Mikael Löf
juni 2012

Examensarbete, 15 högskolepoäng, B
Datavetenskap

Datavetenskapliga programmet
Examinator: Bengt Östberg
Handledare: Thomas Sundberg, Sigma, Anders Jackson, HiG

Kontinuerlig leverans till molntjänst

av

Mikael Löf

Akademin för teknik och miljö
Högskolan i Gävle

801 76 Gävle, Sverige

E-mail:
ndi09mlf@hig.se

Abstrakt

Många företag som utvecklar en webb-applikation försäkrar regelbundet med olika testmetoder att de har en fungerande produkt. Ändå dröjer ofta leverans till kund och slutanvändare. Här beskrivs en lyckad implementation av hur leverans regelbundet kan ske till en molntjänst med ett så kallat blue green pattern som gör att applikationen kan uppdateras med minimal påverkan för slutanvändaren. Lösningen består i att webb-applikationen kan driftsättas i två olika miljöer och att användare alltid, via en html-sida som fungerar som en trafikriktare, skickas till den miljö som har den senaste versionen av applikationen. Genom ett Ant-script driftsätts den nya applikationen på den miljö som för tillfället inte har några besökare. När applikationen svarar på anrop ändrar Ant-scriptet så att slutanvändare i fortsättningen skickas till den nyligen uppdaterade miljön. Slutanvändare loggas regelbundet ut från applikationen så att de också regelbundet besöker trafikriktaren. Av de två testade molntjänsterna, Heroku och Amazon, är Amazon att föredra på grund av större möjligheter för manuell konfiguration.

Nyckelord: Kontinuerlig leverans, Heroku, Amazon, molntjänst, versionshantering, blue green pattern

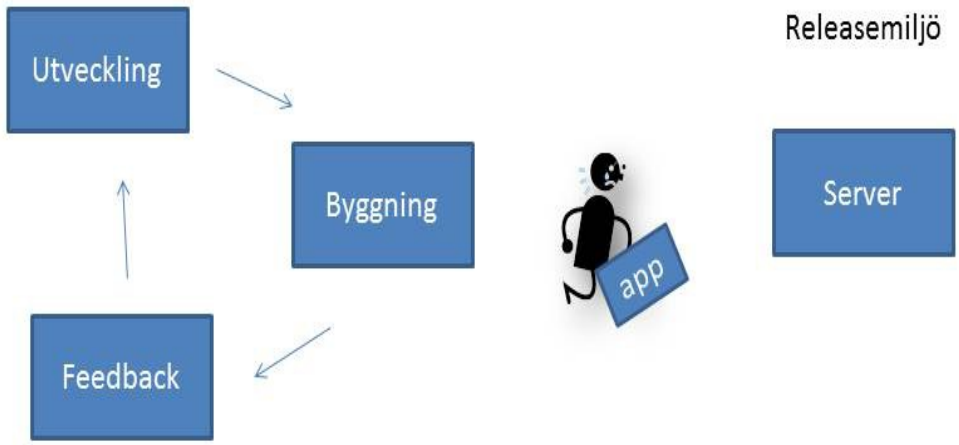
Innehållsförteckning

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.

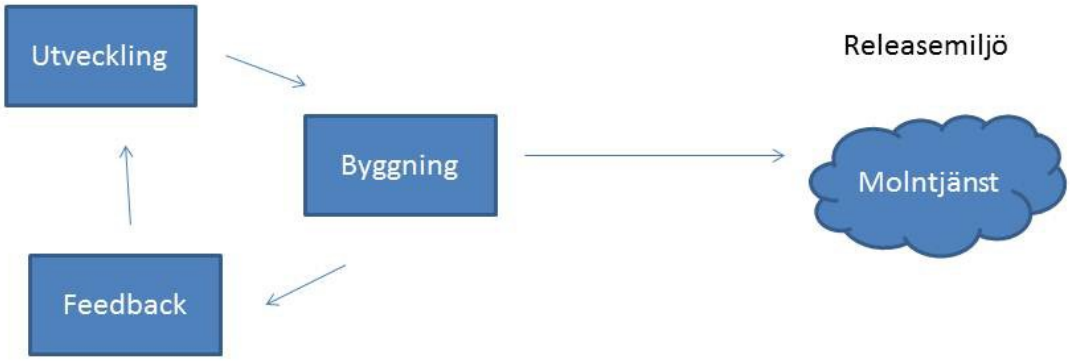


Figur 1: Systemet idag

Ett positivt resultat av testningen av hela systemet bör innebära att teamet har en fullt fungerande produkt. Trots det dröjer det ofta innan produkten levereras till kund så att denne kan ta del av ny funktionalitet, se *figur 1*. Det beror dels på att leverans inom många projekt är 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.

1.1.2 Kontinuerlig leverans

Tanken med kontinuerlig leverans är att inte bara testa ny funktionalitet regelbundet utan även leverera ny funktionalitet regelbundet, se *figur 2*. På detta sätt får kunden tidigt i utvecklingsprocessen ut värde av produkten och kan ge feedback till utvecklingsteamet.



Figur 2: Kontinuerlig leverans till molntjänst

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 vilket omnämns i Mary Poppendiecks presentation [5].

1.1.3 Molntjänst

Kontinuerlig leverans är väl anpassat att användas ihop med en molntjänst. En molntjänst är en tjänst för att distribuerat köra webb-applikationer. Leveransen görs då utan beroende av fysiska servrar.

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 3.3.2*). 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”. Denna uppgift noteras på en post-itlapp som sedan placeras på en Scrum-tavla, *se 3.1.2,* under en av kolumnerna *att göra, aktiv* och *färdig.*

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 *automatiskt driftsättande i release-miljö.* Release-miljön i detta projekt är en *molntjänst*. Jez Humble and David Farley [1] beskriver ingående grundstenarna i kontinuerlig leverans och nämner även fördelarna med att genomföra detta mot en molntjänst.

3.1 Utvecklingsmetod

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.

3.1.1 INVEST

En hjälp när det gäller att besluta hur ny funktionalitet ska tillföras är att hålla sig till INVEST som beskrivs i Bill Wakes's blogg [8].
INVEST är en akronym för *Independent, Negotiable, Valuable, Estimatable, Small, Testable* 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 ”Möjlig att tidsuppskatta” ovan. 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 funktionaliteten. Här kommer även exekverbara specifikationer in vilket gör att kund och utvecklare på förhand kan komma överens om faktiska användningsfall som ska fungera, *se 3.4*.

3.1.2 Scrum

Scrum definieras i Scrum-guiden [11] av Ken Schwaber och Jeff Sutherland och innebär att teamet jobbar i korta iterationer med fokus på väldefinierade uppgifter. Uppgifterna skrivs ner på post-itlappar som sätts upp på en så kallad scrum-tavla. 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.

3.2 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.

3.2.1 Maven

Maven är ett verktyg för byggande av Java-applikationer, utvecklat och dokumenterat av Apache [12]. I en XML-fil vid namn pom.xml (project object model) specificeras vilken typ av fil som bygget ska generera och vilka beroenden och plug-in som behövs. 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 pom-fil och automatiskt importera beroenden.

Maven-arbeten körs i 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.

3.3 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. Det är även möjligt att i efterhand se vem som gjort vad.

Inte bara källkod bör lagras här utan samtliga filer som på något sätt påverkar projektet ska

versionshanteras. Detta gör att tillbakarullning till föregående versioner inte riskerar att ha sönder applikationen på grund av att applikationskoden inte är kompatibel med övrig konfiguration.

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

3.3.1 Git

Ett versionshanteringssystem som använts i detta projekt på grund av den utbredda användningen samt kopplingen till ett community vid namn GitHub, *se 3.3.2*. 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 skapare [13].

3.3.2 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.

3.4 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 användningsfall som definieras i ett språk som är läsbart för icke programmeringskunniga och kopplade till automatiska test exempelvis skrivna i jUnit. Dessa test specificerar vad som ska utvecklas 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 driver utvecklingen framåt.

För att mäta kodkvalité och testtäckning finns olika mer eller mindre avancerade verktyg. Huruvida dessa verktyg automatiskt ska kunna stoppa en release tas upp under *7.6.*

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.

3.4.1 jUnit

Ett testverktyg för Java [14]. Genom att sätta upp olika villkor som ska vara uppfyllda 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.

3.4.2 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.

3.4.3 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 det booleska uttryck som ska utvärderas.

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

3.4.4 Cobertura

Ett verktyg för kontroll av hur stor del av koden som exekveras av något test [17]. Verktyget finns som plug-in till Maven och kan då avbryta bygget om testtäckningen inte överstiger angiven nivå. Cobertura mäter testtäckning på flera nivåer. Om exempelvis en klass använder sig av en metod i samma, eller en annan klass,

anses även denna metods kod blivit testad.

3.4.5 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. Testtäckning mäts endast på en nivå i Sonar jämfört med Cobertura som mäter testtäckning på fler nivåer.

3.5 Automatisk driftsättning

För att byggservern ska kunna sköta driftsättande av applikationen till slutmiljö krävs att denna process är helt automatisk. Högnivåtest i Selenium körs också mot en automatiskt driftsatt applikation vars container dessutom ska stängas automatiskt efter att testerna körts.

3.5.1 Cargo

Ett verktyg för automatisk driftsättning [19]. Används som plug-in till Maven. Detta plug-in kan ladda ner Tomcat från en URL, starta Tomcat-containern, driftsätta önskad applikation under vald Maven-fas och sedan stoppa containern under en annan 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.

3.5.2 Ant

Ant är föregångaren till Maven och kan liksom Maven sköta bygget av en applikation [20]. Ant kan även kopiera filer samt köra shell-script på en fjärrdator. Detta möjliggörs av ett tredjepartsbibliotek utvecklat av jCraft [28].

3.5.3 Surefire

Surefire kan hindra exekverande 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 driftsatts på en container.

3.6 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.

3.6.1 Jenkins

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

3.7 Molntjänster

Jeff Caruso diskuterar i en artikel [29] på Network World tre typer av molntjänster. 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 standardsystem som exempelvis faktureringssystem och andra administrationsverktyg.

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 in-house är att resursen kan flyttas och lagras på ett flertal olika fysiska servrar utan att detta märks för användaren. Hit hör molntjänsten Amazon.

Där emellan finns *Platform as a service(Paas)* som kan lagra och exekvera applikationer utvecklade specifikt för plattformen. Hit hör bland andra 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 driftsättning av egenutvecklad mjukvara.

3.7.1 Heroku

Heroku är en molntjänst av typen Paas integrerad med Git. Genom en anslutning från ett Git-arkiv kan en webb-applikations källkod levereras till, och byggas på, Heroku. I en så kallad Procfile anges vilket sätt Heroku ska använda för att starta applikationen. Detta är väldokumenterat av Heroku. På Herokus hemsida finns en dokumentation som annars är tidvis knapphändig [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 webb-applikationer. Fler dynos kan tillägnas samma applikation som då kan hantera fler anrop snabbare. Detta kallas för att skala upp applikationen. Om en applikation har två dynos tillägnade försäkras att dessa två alltid finns på olika fysiska servrar vilket minimerar risken för att driftstörningar uppstår trots att en av servrarna går ner.

3.7.1.1 Webapp-runner/Jetty-runner

På molntjänster av typen Paas måste uppstartsmetod för applikationen anges. Det skulle då bli mycket omständligt att ange uppstart av en container, driftsättning 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.

3.7.1.2 Postgres

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

3.7.2 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (ec2) är en molntjänst av typen Iaas. Användaren kan starta virtuella ”instanser” och sedan disponera dessa efter eget tycke [24]. Vid startandet av en instans måste operativsystem anges men därefter kan maskinen användas som en vanlig server.

Amazon erbjuder dessutom en mängd plug-in, exempelvis en Paas-tjänst för driftsättning på önskad container.

I fortsättningen kommer Amazon ec2 omnämnas som endast Amazon eller endast ec2.

3.8 Automatisk databasmigrering

För att underlätta processen att driftsätta en applikation utvecklad lokalt på övriga system bör detta ske automatiskt [23]. För varje miljö där applikationen ska driftsättas krävs en specifik databas. De test som körs mot en databas kan exempelvis inte köras mot samma databas som används i produktion då testen ändrar på innehållet i databasen och kanske måste tömma den på information emellanåt. Uppdateringar av databasen anges då i en fil som skickas till övriga system tillsammans med källkoden. Vid uppstart av applikationen i respektive miljö kontrolleras om databasen behöver uppdateras vilket garanterar att applikationen körs på en databas av rätt version. Att sköta detta manuellt hade inneburit mycket extraarbete.

3.8.1 Flyway

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. Tillbakarullning av steg där både applikationskod och databasen modifierats kan innebära problem, *se 4.4.13*.

Flyway-kommandon kan exekveras genom kommandoraden, som plug-in till Maven eller via applikationen. I detta projekt exekveras Flyway genom applikationen. Detta sätt är att föredra då applikationen ansvarar för sin egen databas och ser till att den är uppdaterad till korrekt version.

3.9 Ramverk

Vid utvecklandet av en större applikation används ofta ett stort antal färdiga funktioner förpackade i ramverk.

3.9.1 Java Enterprise Edition

Java Enterprise Edition (Java EE) är en specifikation både när det gäller API och runtime-miljö för, framför allt, webb-applikationer. Enterprise-applikationer körs med hjälp av containrar som fungerar som behållare för applikationerna [25]. Containrarna kopplar URL:er till Java-applikationer för visning av webb-sidor. Tomcat är ett exempel på en snabbstartad container som uppfyller delar av Javas enterprise-API och som passar för mindre projekt.

3.9.2 Java Server Faces 2

Java Server Faces (JSF) är ett ramverk för webb-applikationer i Java EE [10]. En xhtml-fil specificerar upplägget på sidan och hämtar värden från ”manage beans” som är skrivna i Java-kod.

För varje manage bean specificeras ett scope som avgör livslängden 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.

3.10 Utvecklingsmiljö

Eftersom byggandet av applikationen sker enligt en specifikation i en byggfil kan utvecklare välja utvecklingsmiljö 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.

3.11 Blue green pattern

Vid uppdatering av applikationen bör användare fortfarande kunna använda den tidigare versionen fram till dess att den nya versionen är driftsatt på en server och nåbar. För detta krävs att applikationen ska kunna driftsättas i två separata miljöer samt funktionalitet för att skifta mellan dessa. Martin Fowler diskuterar i ett blogginlägg [26] mönstret och hur det skulle kunna implementeras för servrar som finns in-house.

3.12 Sammanfattning av verktyg

Nedan följer en sammanfattning av hur respektive verktyg som använts kommer in i helhetsbilden.

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 GitHub 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 högnivåtest skrivna i Selenium 2 direkt på gränssnittet för att verifiera applikationens funktionalitet. 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 driftsatt applikation används Surefire för att förhindra exekvering av testen under den normala testfasen samt Cargo som automatiskt driftsätter applikationen i en container. Under driftsättning på containern utförs nödvändiga databasuppdateringar av Flyway. Sonar ger ett stöd när det gäller kodkvalité.

Om samtliga test passerar levereras applikationen, antingen via Git till Heroku där applikationen byggs på nytt på samma sätt, eller så skickas den färdigbyggda applikationen via ett Ant-script till Amazon.

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 inhämtades inom var Git för versionshantering, Maven för specificering av hur en applikation ska byggas och grundläggande kunskap i JSF för utveckling av webb-applikation.
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 5*) men på onsdagen fanns en CI-server med grundläggande funktion för byggande samt leverans till Heroku. Det konstaterades även att en tillbakarullning på Git också generar en tillbakarullning på Heroku. Dessutom undersöktes sessionsöverlevnad trots leverans av ny version av hemsidan.

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

Femte veckan
Exekverbara specifikationer lades till med hjälp av Cucumber. Stor tid lades på att felsöka Cucumber-testen som inte kördes vilket visade sig bero på ett s för mycket i ett klassnamn(*se 4.4.6*).
Även mätning av testtäckning lades till med verktyget Cobertura.

Sjätte veckan
Driftsättning 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.
Dessutom har relevant facklitteratur använts.

4.3 Implementation

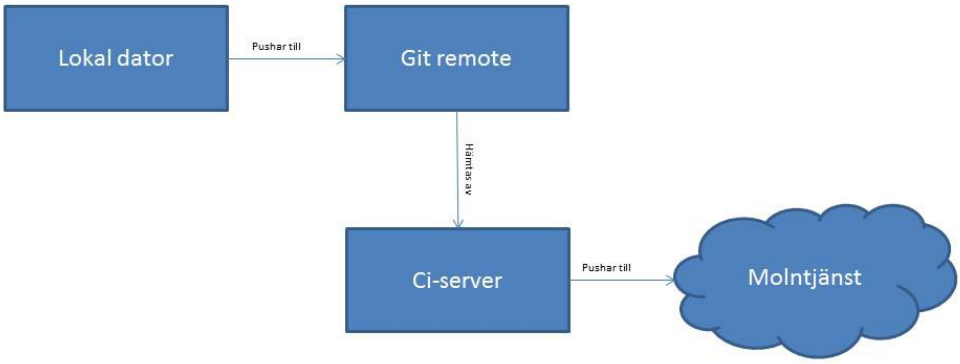
Nedan följer information om den faktiska implementation som används i dagsläget för att uppnå kontinuerlig leverans.

4.3.1 Allmän Struktur

Den allmänna strukturen för projektet beskrivs av *figur 3*. Ny kod pushas till Git remote arkiv när utvecklaren vill publicera den version av applikationen som utvecklats lokalt. CI-servern upptäcker

ändringen och hämtar ner den senaste koden. CI-servern bygger den senaste programkoden och testar denna med högnivåtest. Om applikationen kompilerar och om alla tester går igenom pushas koden till molntjänst (Heroku) eller levereras till container (Amazon).

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

4.3.2 Jenkins

Jenkins består i dagsläget 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.

Heroku-vyn består av två Jenkins-projekt eftersom Heroku kräver att källkodskatalogen ligger i en Git-root, *se 8.1.4.* Ett av Jenkins-projekten 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.

Amazon-vyn består av ett projekt som sköter byggande, testning och driftsättning av applikationen. Detta gör att applikationerna och testerna blir till en enhet som aldrig kan divergera i version. Leveransen i Amazon-projektet sker genom att ett Ant-script som exekveras i ett eftersteg.

Eftersom högnivåtesten är skrivna i Selenium och körs direkt mot en driftsatt webb-applikation 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 fasen för integrationstest.

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 specificerats genomförs nu på samtliga databaser knutna till applikationen.

Eftersom det är önskvärt att endast köra migrering vid uppstart av applikationen skapas en manage bean som annoteras 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. Kombinerat med verktyget Webapp-runner eller Jetty-runner kan webb-applikationer startas på Heroku med några enkla kommandon. Dessa runners kan även användas för att köra webb-applikationer lokalt.

Det är osäkert om Webapp-runner och Jetty-runner passar vid driftsättande av en webb-applikation. Dock är det till dessa applikationer instruktionerna för att leverera en Java EE-applikation pekar på Herokus hemsida [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 plug-in. Detta för att applikationen ska byggas på samma sätt lokalt som på molntjänsten. Det är också att föredra att specificerade beroenden och plug-in finns i ett allmänt arkiv. I detta projekt har endast 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 hårdkodas. 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 kommando-prompten. Därefter används ett kommando för inloggning på Heroku:

```
Heroku login
```

Heroku kräver att sättet som applikationen ska startas explicit specificeras. Detta anges i en fil som måste heta Procfile och ligga i Git-rooten. Vid användning av Webbapp-runner 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 webb-applikation med ett interface som ska publiceras. Resten av texten anger 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
```

Heroku bygger nu applikationen och presenterar den på en webb-adress som kan avläsas i consolen. Denna webb-adress kan bytas ut mot en annan under domänen herokuapp.com. Även herokuapp.com kan givetvis bytas ut mot egen domän, antingen genom ett konsol-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 fortfarande 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

Ett problem med Heroku är att ett projekt som ska levereras till en app-container 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 applikationsmodulen placeras 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.

4.3.5 Amazon

Amazon är en Infrastructure as a service (Iaas) och erbjuder betydligt fler möjligheter än Heroku. Eftersom ingen färdig konfiguration finns för att driftsätta applikationer måste detta göras manuellt genom kommunikation med instanser via protokollet Secure Shell (SSH). Dock finns plug-in som liknar Paas, *se 4.3.5.1* nedan.

4.3.5.1 Användandet av Amazon

Med hjälp av ett plug-in som heter Elastic Beenstalk kan en applikation enkelt driftsättas och sedan besökas på en adress *mittnamn.*elasticbeanstalk.com. Eftersom detta plug-in fungerar precis som en Paas erhålls ingen ytterligare frihet jämfört mot Heroku. Därför användes i detta projekt inget plug-in 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 driftsättning på den Tomcat-container som startats på Amazon-instansen. Detta görs via en Ant-fil som kopierar filer med secure copy (SCP) direkt till maskinen.

För att möjliggöra användandet av eget domännamn har ett ”Elastic ip” kvitterats ut. Detta fungerar som ett virtuellt ip-nummer och är kostnadsfritt hos Amazon. I övrigt fungerar det exakt som ett vanligt ip-nummer och kan alltså via DNS bindas till önskad domän.

Skalning på Amazon kan antingen genomföras genom att begära att den instans som kör webb-applikationen får utökat minne och kapacitet eller genom att använda Amazons tjänst för lastbalansering. Lastbalanseringen går till så att en ny instans skapas vars uppgift är att skicka vidare http-requests till den underliggande instans som för tillfället har minst last. Vid uppskalning skapas en ny instans, genom att kopiera en befintlig, som läggs till i listan för instanser som lastbalanseraren har möjlighet att skicka http-requests till. Amazon kan även konfigureras för att automatiskt skala en applikation beroende på en mängd olika faktorer så som svarstid, besökarantal och liknande.

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 inget ytterligare arbete för att få en fungerande databas.

Det som dock är önskvärt är att ha en gemensam databas för samtliga instanser. Antingen startas en ny instans som enbart hostar en databas eller så används ”Amazon RDS (Relational Database Service)”. I dagsläget används en av de befintliga instanserna även till att erbjuda databas för övriga instanser men om automatisk skalning av databasen önskas bör Amazon RDS användas. Anslutningsuppgifter specificeras sedan i miljövariabler istället för hårdkodade i källkoden så att olika databaser kan nyttjas lokalt, på CI-servern och på molntjänsten. Tester kan då köras mot en lokal databas och applikationen i produktion kan köra mot produktionsdatabasen. Installationen av denna gemensamma databas fungerar i princip som installation av en lokal databas, det som skiljer är att modifiering av säkerhetsinställningar krävs för att tillåta anslutningar från andra maskiner.

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 Transmission Control Protocol (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 mer som en inhouse-server. En topp-pom specificerar de undermoduler som finns, i detta fall main och test. När ny kod pushas till Git byggs hela projektet av Jenkins i den ordning som är specificerad i topp-pomen. I detta projekt byggs först modulen main för att få fram en ny version av applikationen, förpackad i en war-fil. Därefter byggs modulen test som verifierar funktionalitet genom högnivåtest. Om testerna passerar körs ett Ant-script som levererar produkten till molntjänsten. Allt detta ligger i ett Jenkins-projekt. Denna uppsättning är att föredra då 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. Ett plug-in, Webapp-runner, krävs endast på Heroku men eftersom det är ett plug-in som på Heroku startas med hjälp av en separat fil uppstår inga problem på Amazon utan Webapp-runner ignoreras helt.

Viktiga delar i Maven-pomen:

Automatisk driftsättning på tillfällig Tomcat-container

Detta genomförs eftersom applikationen måste vara driftsatt 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 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 driftsättas. 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 6 Ange installationsfolder

Därefter anges vad som ska driftsättas. Då det är nödvändigt att den artefakt som ska driftsättas 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 8 Exekveringsåtgärder

Webapp-runner

För att Webapp-runner ska finnas tillgänglig när applikationen ska startas på Heroku anges detta som ett plug-in. Maven skapar då en katalog där Webapp-runner placeras vilken sedan anges vid uppstart. Om koden i *figur 9* lagts till i pomen kan Webapp-runnern startas med följande kommando:

```
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 har utvecklats. Väl inloggad kan användare

registrera laster och sedan boka tillgängliga laster som sedan presenteras i en lista för den användaren. Applikationen består i dagsläget av en xhtml-sida för varje användarsida. Varje av dessa sidor har en motsvarande böna. Dessutom finns en böna med sessions-scope 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 4.4.9*.

4.3.8 Cucumber

Cucumber används i projektet ihop med de Seleniumtester som kör mot en driftsatt hemsida. Det finns helt klart ett syfte med denna typ av exekverbara specifikationer eftersom de ökar läsbarheten så pass mycket att ej programmeringskunniga bör kunna ta del av och förstå dessa specifikationer.

Klartexten i feature-filen har ingen inblandning av programmerings-syntax. En feature-fil kan se ut som i *figur 10.*

```
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 genom reguljära uttryck vilket gör klartexten exekverbar.

4.3.9 Testning och testtäckning

I dagsläget används två typer av tester. Den första typen är enhetstestning. Dessa testar funktionaliteten av enskilda klasser och ska, då de körs ofta, gå fort att köra. Att köra dessa tester med en riktig databas i botten skulle därför inte fungera. Istället används en klass, skriven i Java-kod, som ersätter databasen och som använder sig av en Hash-tabell för att spara värden.

Den andra typen av tester är de Seleniumtester som testar funktionaliteten på gränssnittsnivå.

Tyngdpunkten vid testandet ligger på enhetstesterna som verifierar funktionen. Fowler talar i en artikel [6] om något han kallar 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 det oftast på för dålig enhetstestning. End to end-test är också bräckliga när det gäller förändringar i gränssnittet och ofta långsamma. Dock fyller de ett syfte eftersom de kan utföra test utan att ha tillgång till källkoden och eftersom de körs direkt på gränssnittet och testar det slutanvändaren upplever.

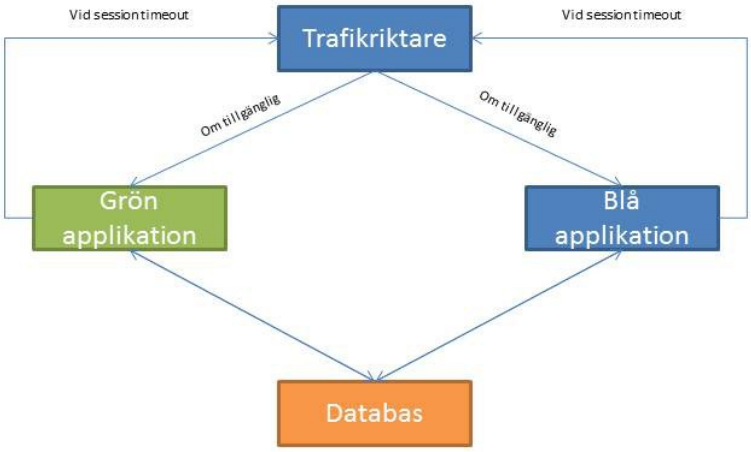
Ett försök att införa Cobertura för att mäta testtäckning har genomförts men detta gav upphov till en del problem, *se 4.4.7*. Cobertura mäter hur många rader i produktionskoden som exekveras från testmetoder. Dock säkerställs inte ens att något test utförs i testmetoden och testtäckningssiffran ger en dålig bild av kvalitén på testerna. Dessa problem sammanlagt resulterade i att Cobertura uteslöts ur projektet.

Ett bättre sätt att mäta kodkvalité är Sonar som också mäter testtäckning men även andra aspekter av kodkvalité. I dagsläget körs Sonar som ett plug-in 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 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 sömlös uppdatering som används idag är blue green pattern, *se 3.11,* och är implementerat så att slutanvändare som vill logga in på applikationen alltid först måste besöka en enkel html-sida, trafikriktare, 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. Dessa 30 minuter kan justeras och avgörs av hur ofta applikationen ska kunna levereras. Om leverans ska kunna ske en gång om dygnet kan

denna tid vara exempelvis 20 timmar (en viss marginal bör användas) men här förutsätts att applikationen levereras var 30:e minut.



Figur 11: Blue Green Pattern

För att uppnå detta krävs en del Ant-script. Dessa Ant-script måste, vid uppdatering av applikationen, kontrollera vilken instans som inte är aktiv, stänga av containern på den servern, uppdatera applikationen, starta containern, vänta på att sidan åter ska svara efter uppdatering och slutligen 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 en lastbalanserare som ligger framför instanserna istället för direkt till de körande instanserna.

Detta mönster kräver även funktionalitet för att logga ut användaren efter 30 minuter men default-implementationen för sessioner i JSF är att sessionerna får förnyad längd så fort användaren av hemsidan är aktiv. Problemet löses genom att, via ett filter som fångar upp alla http-requests, explicit sätta en sessionsvariabel vid inloggning med namnet time. Time får värdet av serverklockan när en användare påbörjar sin session och vid varje sidnavigering anropas detta filter som, med hjälp av variabeln time, räknar ut om användaren varit inloggad för länge och skickar i så fall användaren till startsidan för automatisk navigering 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 en fil som heter web.xml. Detta medför ett problem som innebär att användaren kan ange URL till inloggningssidan på den icke aktiva servern och har då möjligheten att logga in på denna, se *4.4.11.* Under diskussion tas två nackdelar med lösningen upp, *se 7.7.*

Viktigt är att den fil som avgör vilken instans som är aktiv inte versionshanteras. I detta fall skulle inte driftsättning ske varannan gång på varannan instans då uppgifter om vilken instans som för tillfället är aktiv skulle skrivas över vid uppladdning av nya källfiler.

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 detta fall inhämtades först information om helhetsbilden för att sedan bryta ner den i delar som praktiskt kunde implementeras.

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 vilket inte blir fallet om katalogen skapas av någon annan användare. Lösningen till 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 presenterades felmeddelandet ”Public Key”. 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 webbläsardrivrutinen. Problemet löstes genom att installera ett plug-in till Jenkins som heter XVNC och som bistår med funktionalitet för att öppna en webb-läsare.

Ytterligare ett problem bestod i att Selenium inte kunde köra sina tester vid driftsättande på Jenkins. Anledning var att sökvägen som var angiven till hemsidan som skulle testas var `localhost:8080`. Detta fungerade lokalt när webb-sidan driftsattes med Webapp-runner eftersom denna driftsätter på just `localhost:8080`. Vid driftsättande från Jenkins kunde inte Webapp-runner användas då den inte enkelt går att stänga av med ett kommando. Istället driftsätts hemsidan på en tillfällig Tomcat-server och sökvägen till applikationen blir 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. Till en början hade inte databasdrivrutinen angetts som ett beroende i Maven vilket ställde till problem. När detta åtgärdats och en miljövariabel som specificerade anslutningsuppgifter till databasen hittats, kunde till slut en anslutning upprättas.

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 plug-in till Maven. Problemet var dock att migrationen inte fungerade om Maven startades med kommandot clean install utan endast install måste anges.

Dessutom var det 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 Postgres 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. En total ominstallation av Postgres löste problemet.

4.4.5 Skillnader mellan Windows och Linux

Vid ett tillfälle märktes tydligt att olika operativsystem kan orsaka problem. 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 aldrig upp till GitHub, förmodligen beroende på att Windows-versionen av Git, som från början är utvecklat till och testat på Linux, inte såg detta som någon ändring då Windows i grunden inte gör skillnad på stora och små bokstäver (case sensitive).

4.4.6 Cucumber

Cucumber är i stort enkelt att komma igång med. Dock uppstår ett problem i kombination med Surefire, *se 3.6.3*. 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 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 vilka växlar Heroku kör Maven med. Ett tredje alternativ skulle kunna vara att använda ett plug-in 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

Uppstarten av kontinuerlig leverans mot Amazons molntjänst orsakade ett antal problem som dessutom var svåra att felsöka eftersom osäkerhet fanns om orsaken till problemet låg hos själva molntjänsten eller inte. Det första problemet uppstod när SCP skulle användas mot Amazon-maskinen. 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 Amazon-maskinen. Detta berodde på att katalogen försökte skapas i användar-rooten 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 problem testades också användandet av WinSCP men det visade sig att överföringshastigheten var så låg att det blev oanvändbart.

Ett annat problem uppstod vid installationen av den Tomcat-container som var tänkt att användas för webb-applikationen. De filer som behövdes fördes över och korrekt kommando angavs för uppstart av containern. För att verifiera att uppstarten gått som den ska angavs sedan en adress till maskinen vilket, när en körande Tomcat finns, bör presentera Tomcats index-sida. Sidan visades inte och det var nu svårt att avgöra vad felet berodde på. Miljövariabler på maskinen var satta korrekt och containern borde varit igång. Till slut, efter sökande på forum, fanns ett inlägg där Steffen Opel uppmärksammar att problemet berodde på säkerhetsinställningar på Amazon. 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 driftsatt 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 och efter att detta problem uppmärksammats noterades att Tomcat alltid måste startas om vid varje uppdatering av applikationen.

4.4.9 JSF 2, datatable och request scope

I 3.9.2 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, det korta scopet, request scope i ett fleranvändarsystem. Om en datatable populeras enligt information i databasen och denna information uppdateras från annat håll än den aktuella hemsidan kommer innehållet i datatablen och det som visas på sidan inte stämma överens. I denna applikation ska en enskild rad i datatablen kunna väljas (för bokning av last) men fel last kommer därför bokas. 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 men problemet är att alla klasser som berörs då måste implementera Java-interfacet serializable vilket ibland kan vara svårt om tredjepartsbibliotek används.

4.4.10 Sessioner och skalning

Eftersom sessionsinformation sparas på den server som användaren för tillfället är uppkopplad mot kan detta innebära problem vid skalning på molntjänst. Vid skalning på molntjänst används flera instanser och varje http-request riktas om mot den server som för tillfället har minst last. På Amazon löses detta 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. På Heroku finns inte den möjligheten utan där måste sessionsinformation sparas externt, exempelvis i en databas. Detta kan innebära att prestandan sänks då databasanrop måste göras varje sessionsinformation ska kollas upp.

4.4.11 Explicit angivande av URL till inloggningssida

Problem kan 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 görs. Om användare däremot alltid loggar in via startsidan inträffar inte problemet.

En lösning skulle kunna vara att det från startsidan skickas med en variabel med information till inloggningssidan om att användaren kommer dit på ett korrekt sätt. Om denna variabel 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 relaterat till 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 lägga 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.

4.4.13 Tillbakarullning av databas

Med det verktyg, Flyway, som i dagsläget används finns ingen funktionalitet för tillbakarullning. Om ett utvecklingssteg, där både applikationskod och databasen modifierats, ska rullas tillbaka kan detta innebära en del manuellt arbete. Tanken är att ett sådant steg ska kunna ångras endast genom att rulla tillbaka i versionshanteringssystemet men databasen kommer inte automatiskt rullas tillbaka. Antingen måste då databasen manuellt rullas tillbaka eller så måste versionsinformationen om vilken databasversion som är den aktiva ändras i samtliga databaser. Sedan måste den fil som specificerat den senaste uppdatering ändras till att specificera tillbakarullningen. Båda dessa metoder kräver mycket manuellt arbete men även med ett databasmigrationsverktyg som stöder tillbakarullning, hade detta steg behövt utföras separat från tillbakarullningen av applikationskod.

5 Resultat

Här besvaras frågeställningarna från *1.3*. Tillfredsställande svar har hittats åt samtliga frågeställningar.

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

5.1 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.*

5.2 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 är bäst lämpad till ett projekt av den här typen, *se 8.1.*

5.3 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 B.*

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

Genom ett så kallat blue green pattern kan en ny version driftsättas 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 implementation av blue green pattern, *se 4.3.10*

5.5 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 angiven standard för JSF-applikationer. En förbättring som förmodligen skulle behöva genomföras vid lansering av applikationen är ett bättre säkerhetssystem, exempelvis genom containerns inbyggda funktionalitet för säkerhet.

6.2 Utökade tester

I dagsläget gör applikationens småskalighet 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 att 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 projektets struktur.

6.4 Införande av Hibernate

Hibernate är ett bibliotek för Java som försöker lösa problemet att strukturen hos ett objektorienterat projekt skiljer sig så pass kraftigt mot en strukturen i 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 in-memory databas, HyperSonic. Dock har tidsbrist 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 4.4.11*. Under nämnda stycke finns förslag på åtgärder för att undkomma problemet. Dessa åtgärder bör implementeras.

6.6 Skalning

I detta projekt har inte skalning i någon större utsträckning genomförts på grund av de kostnader detta medför. Skalning innebär att mängden last som kan hanteras av en webb-applikation snabbt kan ökas eller minskas beroende på trycket på hemsidan och är en av de största vinsterna med att använda molntjänster och bör därför utvärderas.

6.7 Lasttestning

Eftersom skalning inte testats i någon större utsträckning har inte heller lasttestning genomförts. Det är inte särskilt intressant att utföra lasttestning när de kostnadsfria alternativen på Heroku och Amazon innebär att applikationen körs på maskiner med minimal kapacitet och endast i utvärderingssyfte. En intressant undersökning skulle vara att införa lasttester och se hur dessa påverkas av att skala upp miljön.

6.8 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 lokal molntjänst. 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 administratörer och de som loggar in på sidan tas hand om av lokala servrar. 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 openStack [31].

7 Diskussion

7.1 Allmän diskussion

Detta projekt har varit utmanande då det har handlat om de delar som jag, på förhand, haft minst kunskap om. 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 delar av Java enterprise-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 felsökning vid införande av tredjepartsbibliotek. 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 tänkt 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 extraarbete, 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 Knapphändig tillgänglig 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 efter. 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 Förbättring av blue green pattern

Till en början funderade jag på hur applikationen skulle kunna uppdateras medan en slutanvändare var inloggad. I princip skulle applikationen efter en sidnavigering plötsligt ändrat utseende eller funktionalitet. Dock tror jag inte att detta är önskvärt. Om jag hade varit inloggad på en hemsida och denna ändrar utseende efter en sidnavigering hade jag uppfattat detta som förvirrande.

Det som istället är önskvärt är att användare får vara inloggade på den version av applikationen de loggat in på så länge de önskar och vid ny inloggning komma till den senaste versionen av applikationen. Eventuellt ska sessioner ha en maxtid på exempelvis ett dygn eller liknande men denna tid ska inte vara beroende av hur ofta applikationen uppdateras. I dagsläget loggas användaren ut lika ofta som nya uppdateringar ska kunna utföras. Detta fungerar bra om uppdateringar sker exempelvis en gång om dygnet. Slutanvändare får då alltid vara inloggade ett dygn innan de måste loggas ut och besöka startsidan. Dock har en mer extrem approach tagits i det här projektet och jag har utgått från att applikationen ska kunna uppdateras med 30 minuters intervall. Användare får då endast vara inloggade 30 minuter i streck vilket kan vara störande ur användarperspektiv beroende på syftet för applikationen.

För att kunna tillåta slutanvändare att vara inloggade hur länge de vill men samtidigt ofta uppdatera applikationen skulle nya instanser behöva startas varje gång leverans sker. När sedan samtliga sessioner avslutats på en äldre instans ska denna automatiskt stängas. Detta ställer stora krav på Amazons funktionalitet för att starta och stoppa instanser beroende på aktivitet. Jag har inte fått fram information om allt detta kan utföras med Amazon API Tools men troligtvis finns funktionaliteten. Den stora utmaningen skulle bli att rikta trafik mot den nyligen skapade instansen trots att ip-adressen eller publik DNS på förhand inte är känd.

7.5 Blue green pattern och skalning

Som tidigare nämnts har skalning testats i mycket begränsad utsträckning på grund av kostnader det medför. Här förs dock ett kortare resonemang om vilka aspekter av skalning som bör undersökas ihop med blue green pattern.

Om applikationen ska kunna skalas kommer övergången i blue green pattern ske mellan två lastbalanserare med ett antal instanser bakom. Det är dock inte önskvärt att ha dessa instanser igång hela tiden på grund av kostnaden det medför. Dessutom önskas kanske automatisk skalning beroende på last för ytterligare kostnadseffektiv användning. Amazon erbjuder denna funktionalitet och kan exempelvis skala upp en applikationen när svarstiden på hemsidan överskridit en viss tid. Det som skulle vara mest önskvärt är att kunna starta upp en instans under lastbalanseraren vid skifte och sedan låta denna automatiskt skalas upp beroende på besökarantal. Minimalt arbete hade då behövt utföras i det Ant-script som sköter driftsättande och inte fler maskiner än vad som behövs startas. På samma sätt hade den automatiska skalningen gjort att alla instanser till slut stoppas under den lastbalanserare som inte längre ska vara aktiv. Dock hade en instans behövt vara kvar med känd adress för att möjliggöra deployandet av nästa version av applikationen. Frågan är i detta fall om den automatiska skalningen skulle gå tillräckligt fort om all trafik skulle riktas över under 30 minuter (vilket skulle vara fallet om uppdateringar ska kunna ske var 30:e minut).

Applikationen skulle också kunna skalas manuellt. I detta fall skapas lika många instanser under varje lastbalanserare och instanserna under den lastbalanserare som inte används skulle då stoppas när de inte är aktiva. Detta görs automatiskt via Amazon API Tools. Samtliga dessa instanser skulle anges i Ant-scriptet som sköter deployandet.

7.6 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 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 kodkvalitén innan leverans. Att inte publicera ändringar som passerat acceptans-tester på grund av, exempelvis, för låg testtäckning eller kodkvalité, är förmodligen ett dåligt system. För användaren av hemsidan märks inte den dåliga kodkvalitén, bara att denna fått ny funktionalitet. Dock bör självklart den typen av problem rättas till i efterhand.

7.7 Nackdelar med implementation av blue green pattern

Den lösning som finns idag för att kunna uppdatera hemsidan regelbundet har två tillkortakommanden.

Det första tillkortakommandet är att användare måste loggas ut regelbundet, i dagsläget var 30:e minut. Det finns andra webb-applikationer som fungerar på detta sätt, framför allt hos banker där stort fokus ligger på säkerheten. Dessutom kombineras denna utloggningsregel med en regel som säger att en användare max får vara inaktiv i exempelvis 5 minuter. Därmed loggas de flesta användare ut inom de 30 minuter de max får vara inloggade.

Det andra tillkortakommandet är att uppdateringar av applikationen max kan genomföras lika ofta som maxtiden för sessioner. Om denna tid är 30 minuter kan uppdateringar max ske var 30:e minut.

8 Slutsatser

8.1 Heroku eller Amazon

8.1.1 Paas kontra Iaas

Heroku är en lättanvänd molntjänst av typen Paas och riktad mot en mindre användargrupp. Eftersom Heroku levererar en färdig plattform måste projektet vara utvecklat på en plattform som stöds av Heroku 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 runtime är angivna enligt ett mönster på molntjänsten måste dessa anges på samma sätt i den lokala miljön för att möjliggöra utläsning i runtime.

Amazon, som är av typen Iaas, erbjuder en mycket större frihet då man har direkt tillgång till den instans som kör webb-applikationen.

8.1.2 Skalning

Amazon har ett något mer komplicerat API för skalning. Heroku erbjuder istället färdig funktionalitet för skalning som uppnås genom ett enkelt kommando.

På Heroku skapar skalning problem vid användande av sessioner då sessionsinformation eventuellt inte är känd när ett anrop görs. 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].

På Amazon kan så kallade sticky sessions användas för säkerställa att sessioner fungerar som de ska trots uppskalad applikation. Detta minskar inte prestandan på det sätt som externa sessioner gör. Dock gör det att last inte kan fördelas över instanser bakom en lastbalanserare lika väl då varje instans måste behålla en session tills denna avslutas. Med externa sessioner kan varje http-request skickas till den instans som för tillfället har minst att göra.

8.1.3 Databas

Heroku stödjer endast databaser av typen Postgres i botten. Dock finns plug-in som gör att annan SQL-dialekt kan användas men i botten ligger en Postgres-databas. På Amazon finns möjlighet att installera den databas som önskas.

8.1.4 Projektstruktur

Ett stort problem med Heroku är att källkod och 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 mer rörig bild av projektet och sämre översikt jämfört med Amazon där projektet kan struktureras efter eget önskemål.

8.1.5 Sammanställning

Detta sammanställt gör att Amazon i detta projekt erbjudit en bättre upplevelse ur användarperspektiv. Applikationens begränsade omfång gör dock att slutsatser rörande uppträdande vid driftsättning är svåra att dra.

8.2 Kontinuerlig leverans utan nertid

Önskemålet är att låta nyanlända slutanvändare av webb-applikationen få besöka den senaste versionen av applikationen medan redan inloggade slutanvändare får vara inloggade och ha tillgång till den tidigare versionen så länge de önskar. Detta uppnås inte helt och hållet idag utan slutanvändare måste regelbundet loggas ut. Hur detta skulle kunna lösas diskuteras under *7.4*.

8.3 Maven

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

8.4 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 mer komplex.

8.5 Heroku och autentisering

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 på Heroku. För en webb-applikation som körs ”in-house” används ofta containerns så kallade realm för att enkelt och säkert autentisera användare. Dock kräver 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 på Heroku finns inga installerade konfigurationsfiler för servern varför användandet av realmer blir krångligt, om ens möjligt. Istället används i projektet en böna med sessions-scope som håller reda på inloggad användare. På varje sida görs en kontroll av inloggad användare. Är detta värde null skickas besökaren till inloggningssidan.

8.6 Selenium

Selenium har fördelen 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.

8.7 Kontinuerlig leverans som leveransmodell

Att regelbundet leverera en produkt tillför värde både för kunden som regelbundet kan se och utvärdera ny funktionalitet för produkten, och 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.

9 Tack

Jag vill avsluta med att tacka mina handledare på Sigma, Thomas Sundberg och Johan Karlsson. De har varit ett stort stöd under mitt examensarbete. Tack!

10 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) *Getting started with opesStack, openStack*, http://wiki.openstack.org/GettingStarted, (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)

Bilaga A: Ordlista

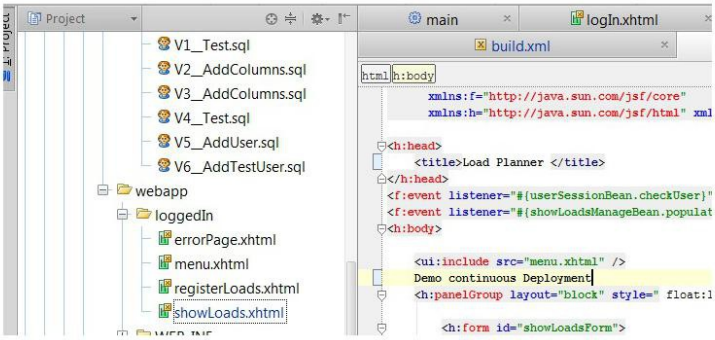
- Nertid – Den tid som webb-applikationen inte är tillgänglig för användare.
- Sömlös – Utan nertid.
- Exekvera – Köra igång.
- Container – En container erbjuder en runtime-miljö för en Java enterprise-applikation.
- Datatable – En komponent i JSF som presenterar information i en tabell.
- In-house server – Server kan nås fysiskt.
- WinSCP – Verktyg för kopiering av filer mellan maskiner via SCP-protokollet.
- Migrering – Uppdatering av databasen utan att tidigare information går förlorad. Kan även innebära att system flyttas från en plats till en annan, men i detta fall är det den tidigare förklaringen som gäller. Anledningen till detta är att Flyway kallar sin mjukvara för ett verktyg för databasmigrering.
- Runner – Verktyg som fungerar som en container utan att någon container behöver installeras.
- Polla – Regelbundet kontrollera om förändring skett.
- Git-root – Den högsta katalognivån för ett projekt som versionshanteras med Git.
- Instans – Amazons begrepp för en körande maskin av någon typ.
- Remote – På annan fysisk maskin.
- Push (pusha) – Att skicka kod från lokalt arkiv till remote arkiv.
- Agilt – Ett samlingsnamn för utvecklingsmetoder som bygger på arbete i korta iterationer och i nära samarbete med kunden.
- Artefakt – Benämning för de filer som skapas av Maven när ett projekt byggs.
- War-fil – Web Application Archive. En artefakt som innehåller en webb-applikation

Bilaga B: Användarperspektiv

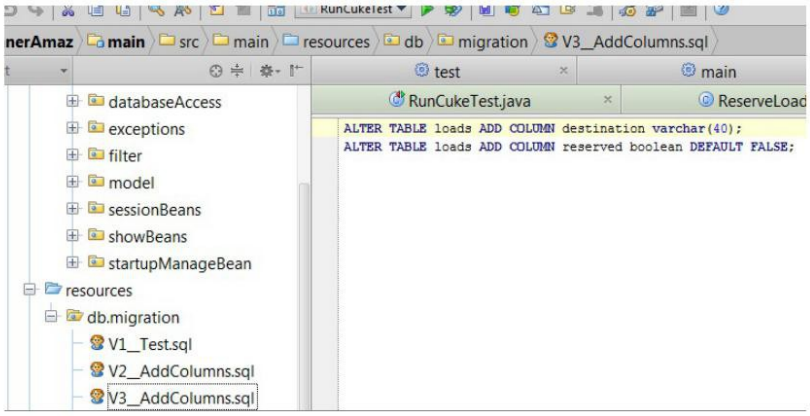
Bilagan är en kort lathund för användare av systemet. Originalet är en presentation i Open Office Impress.

Utveckling på den lokala maskinen

Utvecklingen genomförs på den lokala maskinen med ramverket JSF 2. I detta fall läggs texten ”Demo Continuous Deployment” till för att demonstrera processen som sker när en uppdatering genomförs.



Uppdateringar av databasen genomförs genom att ange önskad sql i en sql-fil i katalogen db/migration. Flyway kommer då att fånga upp dessa uppdateringar som genomförs både på databaser som är avsedda för testmiljön och på databasen som används i drift. Här införs en ny kolumn i databasen loads. Notera namngivningen av Flyway-filen (V3_AddColumns.sql). Flyway känner automatiskt av om en fil skapas i denna katalog med ett högre versionsnummer än det högst befintliga. Meta-information om aktuellt versionsnummer för specifika databaser sparas i den berörda databasen.

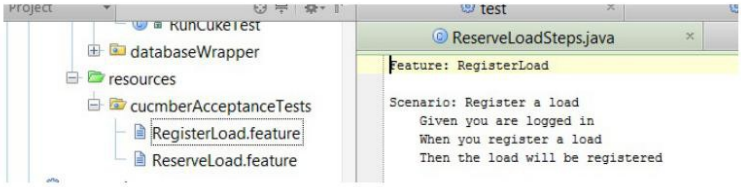


Utveckling på den lokala maskinen

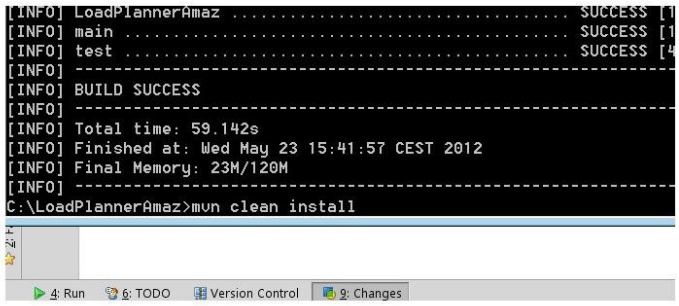
Beroenden av tredjepartsbibliotek och eventuella plugins specificeras i en fil vid namn pom.xml. Maven använder denna fil för att bygga applikationen. Nedan visas top-pomen i Amazonprojektet. Här har undermoduler specificerats (main och test) och dessutom jUnit som beroende. Eftersom jUnit inkluderas i top-pomen kommer det att finnas tillgängligt även i undermoduler.

Utveckling på den lokala maskinen

Högnivåtest anges i Cucmber-syntax. Så kallade feature-filer specificerar ett avgränsat användningsscenario. Dessa test ”limmas” sedan ihop med programkod som kör tester på värden erhållna med hjälp av Selenium.



Nu kan projektet byggas lokalt för att säkerställa att testerna går igenom. Om projektet byggs utifrån top-pomen kommer först applikationen byggas och därefter kommer den tillfälligt deployas och högnivåtesterna körs. På bilden byggs projektet utifrån top-pomen som ligger i projektets root. Här byggs projektet med kommandot mvn clean install vilket säger åt Maven att först ta bort tidigare byggda versioner och sedan köra install vilket i princip innebär att kompilera och exekvera tester.

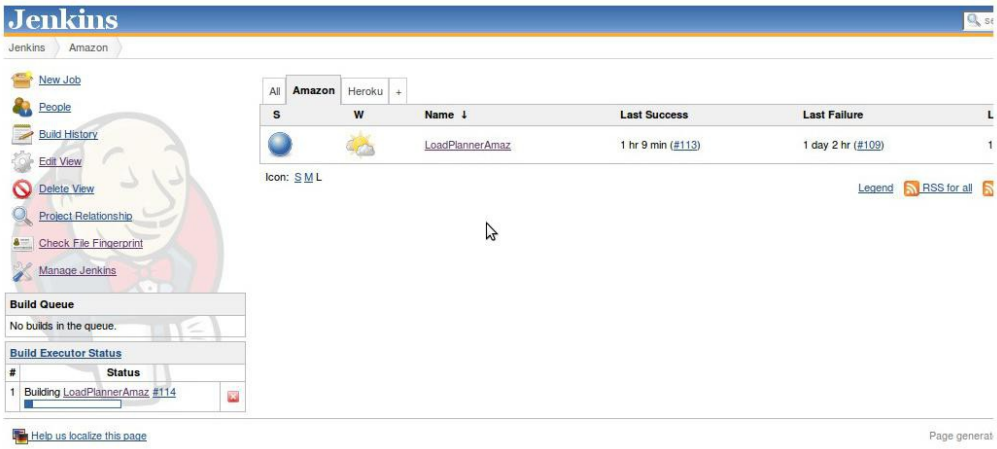


Utveckling på den lokala maskinen

När önskad funktionalitet lagts till och applikationen är redo för publicering görs först en git commit och sedan en git push för att skicka upp källkod till versionshanteringssystemet. I detta fall genomförs en comit från ide:t eftersom detta är integrerat med git.

Jenkins

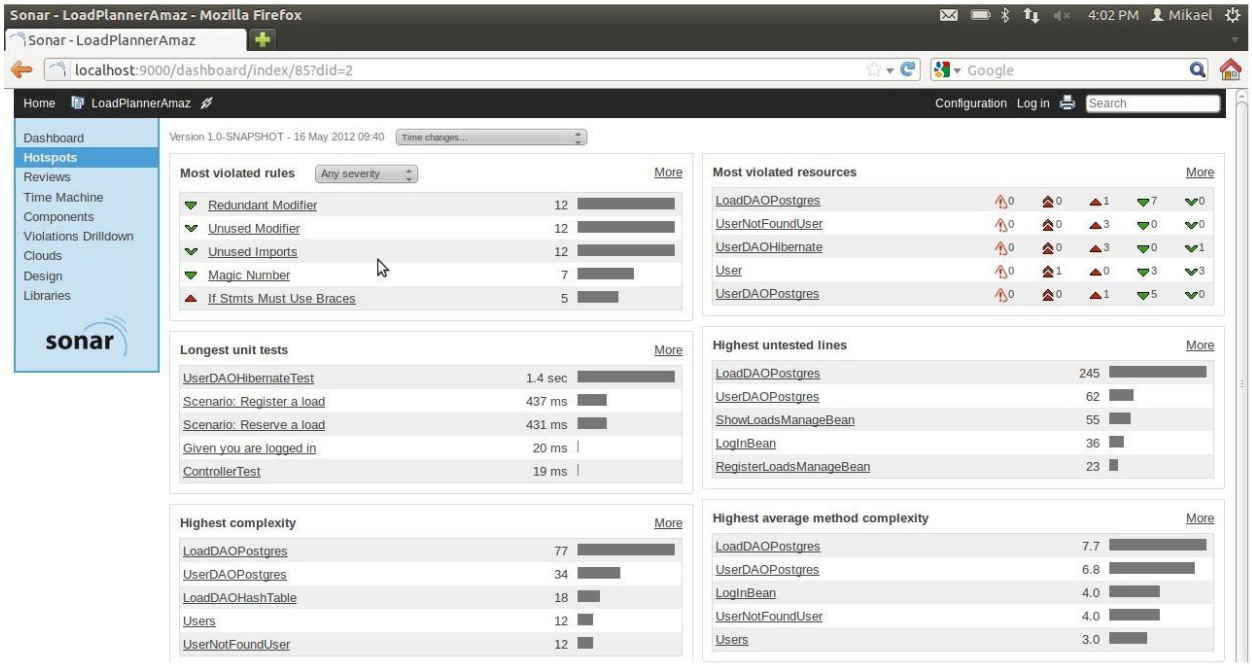
När Jenkins upptäcker att en ny version av projektet laddats upp till versionshanteringssystemet kommer denna hämtas ner och byggas på precis samma sätt som på den lokala datorn. I gränssnittet presenteras bland annat projekt som byggs för tillfället, status för det senaste bygget samt en bild som representerar det genomsnittliga resultatet från de senaste byggena i form av en väderlek.



Om bygget går bra och testerna går igenom kommer Jenkins köra det ant-script som deployar applikationen på molntjänsten.

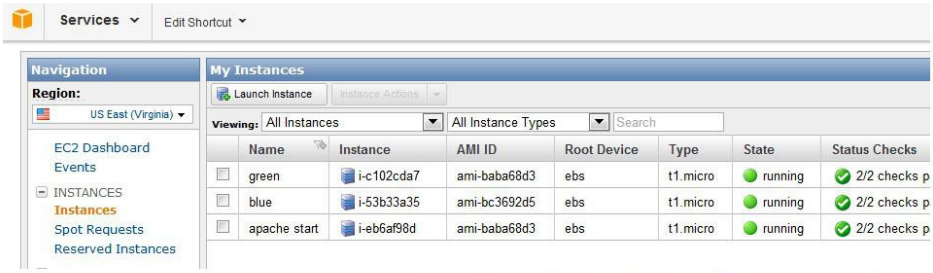
Jenkins

Sonar finns som plug-in till Jenkins och för projektet kan väljas att en Sonar-analys ska göras. Denna presenteras i ett gränssnitt på port 9000 på byggservern.



Molntjänst

I consolen på Amazons hemsida kan körande instanser administreras. Nya instanser kan startas som ser exakt likadana ut som befintliga instanser och lastbalanserare kan kopplas in vilket möjliggör skalning. Här syns den blåa instansen, den gröna instansen samt den apache-server som riktar trafiken mot rätt maskin.



Om en instans läggs till måste ip-nummer eller public dns införas i ant-scriptet i filen build.xml för att denna maskin ska bli uppdaterad med den nya versionen. Observera att om skalning ska genomföras måste även en lastbalanserare sättas in (vilket görs enkelt via Amazons console) och trafikriktaren måste rikta trafik mot lastbalanseraren istället för rakt mot instanserna.

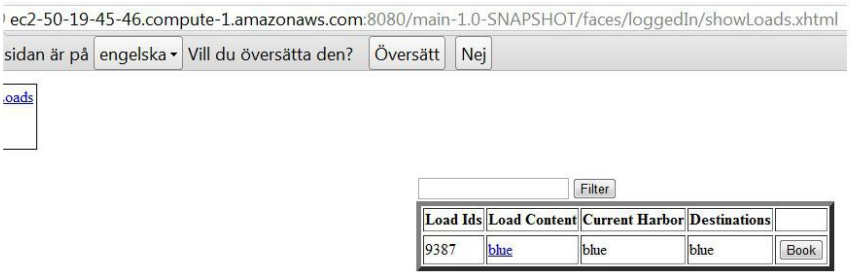
```
<target name="greenDeployment" depends="greenBlueTest" if="isGreen">
    <antcall target="deployNewVersion">
        <param name="forHost" value="ec2-23-20-215-166.compute-1.amazonaws.com"/>
    </antcall>
    <antcall target="redirectUsersAndChangeNextDeployment">
        <param name="forHost" value="ec2-23-20-215-166.compute-1.amazonaws.com"/>
        <param name="oldColor" value="green"/>
        <param name="newColor" value="blue"/>
    </antcall>
</target>

<target name="blueDeployment" depends="greenDeployment" if="isBlue">
    <antcall target="deployNewVersion">
        <param name="forHost" value="ec2-50-19-45-46.compute-1.amazonaws.com"/>
    </antcall>
    <antcall target="redirectUsersAndChangeNextDeployment">
        <param name="forHost" value="ec2-50-19-45-46.compute-1.amazonaws.com"/>
        <param name="oldColor" value="blue"/>
        <param name="newColor" value="green"/>
    </antcall>
</target>
```

Slutanvändarens upplevelse

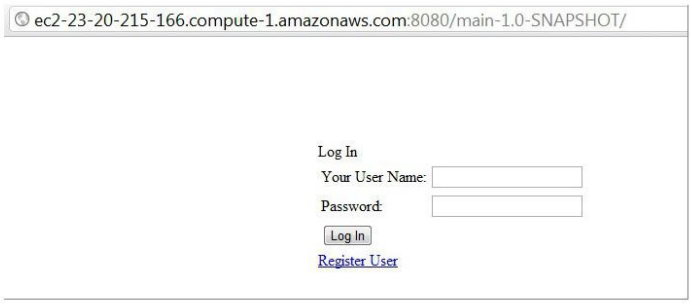
Användaren anger url:en till apache-servern och skickas vidare till den aktiva instansen, i detta fall instansen med public dns ec2-50-19-45-46.

Användaren loggar in och tillåts vara inloggad i 30 minuter.

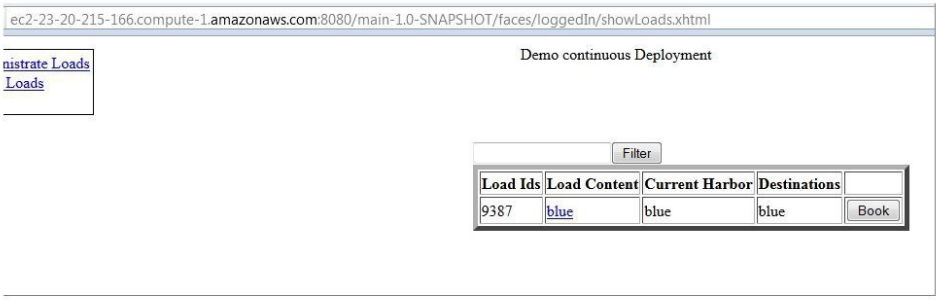


Slutanvändarens upplevelse

Efter dessa 30 minuter loggas användaren ut och skickas till apache-servern som dirigerar användaren till rätt maskin. Eftersom vi nyligen lagt till texten ”Demo Continuous Delivery” och publicerat denna ändring bör användaren nu skickas till den nya maskinen. Notera förändringen av url:en.



Nu syns uppdateringen på hemsidan. Användaren har påverkats genom att max kunna vara inloggad i 30 minuter i sträck(oavsett om en uppdatering genomförts eller inte).



Bilaga C: 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
12-06-01	9	Rapporten färdigställs	ML