

DETTA ÄR ETT FÖRSÄTTSBLAD

Projektnamn	Kontinuerlig leverans av system under test
Beställare	Sigma

Sammanfattning

Här görs en sammanfattning av dokumentet som ska rymmas på denna sida.

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 på resultat om jenkins och maven	ML
12-04-25	4	Lade till Flyway och databas	ML
12-05-03	5	Cobertura samt Cucumber	ML

Innehållsförteckning

Inledning	4
Bakgrund.....	4
Kontinuerlig leverans	4
Exekverbara specifikationer.....	4
Syfte	5
Frågeställningar.....	5
Teori	5
Metod	7
Progress.....	7
Informationsinsamling	8
Genomförande.....	8
Databas och Heroku	8
Maven.....	9
Jenkins.....	10
Heroku.....	10
Användandet av heroku	11
Allmän Struktur.....	11
Projektstruktur.....	12
Applikationen.....	12
Lösningförslagen	12
Historik.....	12
Problem	13
Helhetskonceptet.....	13
Problem vid installation av Jenkins	13
Problem vid installation av databas på Heroku.....	13
Flyway.....	13
Rättigheter på Linux.....	14
Operativsystemsskillnad	14
Resultat.....	14

Svar på frågeställningar	14
Rekommendationer	14
Säkerhet.....	14
Diskussion.....	14
Slutsatser	15
Kontinuerlig leverans och sessioner	15
Heroku.....	15
Kontinuerlig leverans	15
Maven.....	15
Externa verktyg	15
Heroku och autenticering	16
Referenser	16
Bilagor.....	16

Inledning

Bakgrund

Kontinuerlig leverans

Att vid utvecklingen av en produkt regelbundet bygga hela systemet för verifiering av funktionalitet är ofta önskvärt. Utvecklare får snabb feedback på de delar som inte integrerats korrekt med övriga systemet och kan då rätta till detta direkt. Om systemet fungerar korrekt fortsätter ofta utvecklingen och med korta intervaller sätts hela systemet ihop för testning av funktionalitet.

Ett positivt resultat av testningen av hela systemet bör innebära att teamet har en fullt fungerande produkt. Ändå är det ofta med mycket långt intervall som produkten levereras till kund så att denne kan ta del av ny funktionalitet. Tanken med kontinuerlig leverans är att inte bara testa ny funktionalitet regelbundet utan även leverera ny funktionalitet regelbundet. [1]

För att få till denna kedja att regelbundet leverera en applikation till kund underlättar användandet av molntjänst. En molntjänst är som en virtuell, distribuerad server som utan förändring i applikation kan skalas för att hantera mer trafik. Dock krävs en enkelt och entydigt sätt att bygga applikationen eftersom denna ska byggas på flera olika system. För detta används Maven, ett verktyg där byggandet entydigt specificeras (se Teori).

Byggandet och testandet av en komplex ee-applikation kan vara mycket tidskrävande, framför allt om databasen ska initieras och databasanrop ska göras. Därför byggs sällan hela applikationen lokalt under utveckling, utan delar av applikationen testdrivs fram mot en inmemory-databas. Att bygga hela applikationen och testa mot en riktig databas ansvarar en ci-server för. Denna har symboler som indikerar om alla byggen lyckats eller om något fallerat.

Denna typ av leveranssystem blir alltmer populärt och som exempel kan anges att Google Mail leverar i genomsnitt fyra gånger per dag.[5]

Exekverbara specifikationer

Det språk som används i en kravspecifikation är talspråk och definierar ofta en produkt utan konkreta exempel. Det finns inte heller något som säger att det som är angivet verkligen uppfylls av produkten. Med exekverbara specifikationer minskar gapet mellan beställarens och utvecklingsteamets språk och de specifikationer som är angivna består av konkreta exempel som

kan förstås av beställaren och exekveras av utvecklaren.

Syfte

Syftet med det här projektet är att ta ett helhetsgrepp om utvecklingen av en webapplikation, från idé till slutanvändare. Vid projektets slut ska en modell finnas för kontinuerlig leverans av en automatiskt testad produkt till molntjänst.

Frågeställningar

Vilka typer av molntjänster finns?

Vilken typ av molntjänst passar för det här projektet?

Vilket stöd finns för införandet av databas?

Hur ska uppdateringar av databasen ske?

Hur ser relationen mellan versionshanteringssystem och molntjänst ut?

Vad har en ci-server för uppgift och var kommer den in i strukturen?

Teori

Git

Ett versionshanteringssystem som använts i detta projekt på grund av den utbredda användningen samt kopplingen till communityt GitHub(se nedan). Ett lokalt repository finns på användarens dator vilket är kopplat mot ett distribuerat repository. Ändringar läggs till i det lokala repositoryt som sedan lokalt slås ihop(merge) med filerna på det distribuerade repositoryt. Därefter ”pushas” ändringarna till det distribuerade repositoryt.

GitHub

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

Cobertura

Ett verktyg för kontroll av testtäckning. Verktöget finns som plugin till Maven och kan då avbryta bygget om testtäckningen inte överstiger angiven nivå.

Selenium 2

Ett verktyg för analys av innehållet på en hemsida. Givet namn eller id kan värden på attribut erhållas vilka sedan kan användas för testning. Kombinerat med junit ger detta möjlighet till automatiska acceptans-tester utifrån interfacet.

Cargo

Ett verktyg för automatisk deployment. Används som plugin till maven.

Cucumber

Cucumber erbjuder ett ramverk för högnivåtest genom reguljära uttryck. Uttrycken har formen:

1. Givet att - följt av angivna värden av användaren.”

2. När – följt av operationen användaren utfört när testet ska utvärderas.
3. Då – följt av ett booleanskt uttryck som är det uttryck som ska utvärderas.

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

Java EE

JSF 2

Ett verktyg för generering av html-sidor via java. En xhtml-fil specificerar uppläget på sidan och hämtar värden från en ”manage bean”.

JUnit

Ett testverktyg för java. Genom användandet av olika assert-metoder kan funktionalitet testas. Om ett test inte går igenom visas ett tydligt meddelande om det antagande som inte stämmer.

Molntjänster

Det finns 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 standard-system som exempelvis faktureringsystem och hr-system.

Infrastructure as a service(IaaS) är den mest generella tjänsten och erbjuder i princip bara en distribuerad virtuell maskin. Kunden kan installera eget operativsystem och hantera resursen efter eget tycke. Det enda som skiljer från att ha en dedikerad server är att resursen kan flyttas och lagras på ett flertal olika fysiska servrar utan att detta märks för användaren.

Där emellan finns *Platform as a service* som kan lagra och exekvera applikationer utvecklade specifikt för plattformen. Härtill hör bland annat molntjänsten Heroku som i dagsläget är anpassat för bland annat plattformarna ruby, python och java. För detta projekt är det denna typ av molntjänst som är intressant eftersom minimalt arbete krävs för önskad funktion. Fördelen över en vanlig ”inHouse”-server är att med molntjänsten kan produkten testas i sin tänkta miljö regelbundet. Ingen installation eller konfiguration behövs.

Heroku

Heroku är en molntjänst integrerad med git för kontinuerlig leverans. Genom en anslutning från ett git-arkiv kan en web-applikations källkod levereras till, och byggas på, en heroku-server. I en så kallad Procfile anges vilket sätt Heroku ska använda för att starta applikationen. [4]

Heroku bygger på dynos, processer som kör exempelvis applikationer. Fler dynos kan dedikeras till samma applikation som då kan hantera fler anrop snabbare. Detta kallas för att skala upp applikationen. Om en applikation har två dynos dedikerade försäkras att dessa två alltid finns på två olika fysiska positioner minimerar risken för att driftstörningar uppstår trots att en av servrarna går ner.

WebappRunner/JettyRunner

På molntjänster av typen Paas måste uppstartsметод för applikationen anges. Det skulle då bli mycket omständligt att ange uppstart av en container, deployment på den containern och så vidare.

Istället används en runner som genom maven läggs in i en relativ sökväg och startas med ett enkelt kommando i vilket också filen som ska startas anges. Vid användande av Webapp-runner startas en tomcat-container och med Jetty-Runner startas en Jetty-container.

Maven

Maven är ett verktyg för byggande av java-applikationer. I en xml-fil (pom.xml) specificeras vilken typ av fil som ska byggas, beroenden och plugins. Dessa beroenden och plugins laddas sedan ner(vid behov) från ett maven repository, antingen maven central eller ett explicit specificerat. Vid nedladdningen lagras även de nedladdade modulerna i lokalt repository vilket betyder att modulerna i fortsättningen inte behöver laddas ner. De flesta moderna ide:n har stöd för maven-projekt och kan öppna en pom-fil och automatiskt importera beroenden.

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

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

Postgresql

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

Flyway

Databasanvändning bygger på persistens och att databasen befinner sig i närheten av applikation för snabb exekvering. Därför krävs ett flertal olika databaser; lokalt, på ci-servern samt på molntjänsten. Kontinuerlig leverans innebär löpande förändringar och så även för databasen. Att utveckla databaserna manuellt med sql-script är mycket tidskrävande men framför ökar riskerna för misstag när samma förändring ska införas på tre olika databaser samtidigt.

Flyway erbjuder funktionalitet för automatisk uppdatering av databaser knutna till applikationen.

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 osv. Detta bör därför skötas manuellt, exempelvis i en ny migration i flyway.

Flyway-kommandon kan exekveras genom kommandoraden, som plugin till maven eller via applikationen. I detta projekt exekveras Flyway genom applikationen, dels för att det är mycket lättare att få tag på de miljövariabler som behövs på Heroku och dels för att applikationen då ansvarar för att hålla sin databas uppdaterad.

Surefire

Kan användas som ett plugin till Maven för att låta test exekvera under ett annat steg i livscykeln än vad de normalt gör. Detta är användbart när testen måste köras efter att en applikation deployats i en container.

Metod

Projektet har bedrivits agilt med veckoiterationer. 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".

Applikationen har utvecklats i små steg där varje steg innebär utökad funktionalitet för användaren.

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.
- Grundläggande kunskap i jsf för utveckling av webapplikation.

En prototyp för applikationen skapades även första veckan.

Andra veckan

Kunskap om molntjänsten Heroku hämtades och ett exempelprojekt levererades. Efter detta levererades även den tidigare framtagna prototypen till molntjänsten.

Funktionalitet för att reservera en last lades till i applikationen.

Tredje veckan

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

Fjärde veckan

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

Femte veckan

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

Även testtäckningsmätning lades till med verktyget Cobertura.

Informationsinsamling

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

Genomförande

Databas och Heroku

Eftersom applikationen som utvecklats ska ha tillgång till en databas både lokalt för testning och på molntjänsten kan uppgifter om databasen inte specificeras hårdkodad i koden. Istället måste drivrutinen anges som ett mavenberoende 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 maven 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);
}

```

Vanligtvis vid användning av jdbc-drivrutiner används metoden `Class.forName()` för att statiskt ladda in en drivrutin i `DriverManager`. Detta är dock inte nödvändigt eftersom detta sköts genom maven.

Migration

För att migrera databasen används `FlyWay`. Filer innehållande rena sql-kommandon sparas i en katalog namngiven `db.migration`. `FlyWay` anges som ett beroende i Maven och från programkoden anges först en anslutning till databas för `FlyWay` och därefter anropas metoden `migrate()`. Alla ändringar som görs i databasen anges i sql-fil och samtliga databaser knutna till applikationen kommer att ta del av uppdateringen.

Eftersom det är önskvärt att endast köra migreringar vid uppstart av applikationen skapas en `ManageBean` som annoterats med `Scope application`. Värdet `eager` sätts sedan till `true` och koden i bönan kommer att exekveras när applikationen laddas.

Maven

Viktiga delar i maven-pomen:

Automatisk deployment på tillfällig Tomcat-container

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

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

```

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

```

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

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

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

```

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

```

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>
```

Webapp-runner

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

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

Jenkins

Jenkins bygger själva applikationen (hemsidan) automatiskt vid uppdatering i versionskontrollsystemet. Byggandet av applikationen triggar ett annat projekt, innehållande seleniumtesterna. Detta projekt deployar applikationen från föregående projekt i en tillfällig tomcat-container, kör selenium-tester och om dessa går igenom pushar projektet till heroku för leverans till molnet. Koden för att pusha till heroku körs i ett så kallat post step som jenkins automatiskt kör efter bygget om det gått bra (valbart är att köra det även om bygget inte gick bra). För användaren är den enda åtgärd som behövs är att göra en push till git och sedan uppdateras hemsidan automatiskt efter att allt blivit byggd. Eftersom ci-servern aldrig gör någon commit till git pushas alltid det senaste från git-repot till heroku. Det betyder att rollbacks fungerar automatiskt med denna konfiguration.

Heroku

Heroku är relativt lätt att komma igång med och kombinerat med verktyget webapprunner eller jetty-runner kan web-applikationer sättas upp lokalt eller via heroku med några enkla kommandon. Ett problem med Heroku är att ett projekt som ska levereras till en Heroku-appcontainer måste ligga i en git-root. Ett projekt som består av fler Maven-moduler kan således inte levereras till Heroku direkt från projektets ordinarie git-arkiv. En lösning till detta är att hålla applikationsmodulen i ett separat projekt som placeras i en git-root. Andra moduler får sedan specificera ett beroende till den byggda artefakten. Även om det möjligtvis skulle gå att ladda upp ett multimodulprojekt till heroku och låta Heroku bygga både applikation och tester är detta inte önskvärt eftersom total kontroll över byggmiljön hos Heroku inte kan erhållas.

Det är osäkert om web-app runner och jetty-runner passar vid driftsättande av en web-applikation. Dock är det till dessa applikationer instruktionerna för att leverera en java ee applikation pekar.[3]

Vid användandet av molntjänster som Heroku som bygger applikationen utifrån en pom är det viktigt att i byggspecifikationen(pomen) specificera versionen för beroenden och plugins. Detta för att applikationen ska byggas på samma sätt lokalt som på molntjänsten. Det är också att föredra att de specificerade beroendena och plugins finns i ett allmänt arkiv. I detta projekt har endast bibliotek från maven central använts.

Användandet av heroku

Förutsatt att ett projekt versionshanteras med Git och byggs via maven utförs några enkla steg för att leverera projektet till Heroku.

Först installeras Heroku Toolbelt. Path-variabeln i Windows sätts så att Heroku-kommandon blir tillgängliga via commando-prompten. Därefter används ett kommando för inloggning på Heroku:

```
Heroku login
```

För att projektet ska kunna köras på Heroku-servern specificeras hur applikationen ska köras. Detta anges i en fil som måste heta Procfile och ligga i git-rooten. Vid användning av web-app runnern blir denna fil enkel med den enda raden:

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

Sedan skapas en remote-koppling från ägarens git-arkiv till Heroku.

```
Heroku create -stack cedar
```

Det som återstår är att pusha applikationen till heroku.

```
Git push Heroku master
```

Nu levereras en web-adress där applikationen presenteras. Denna web-adress kan bytas ut mot en annan under domänen herokuapp.com.

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.

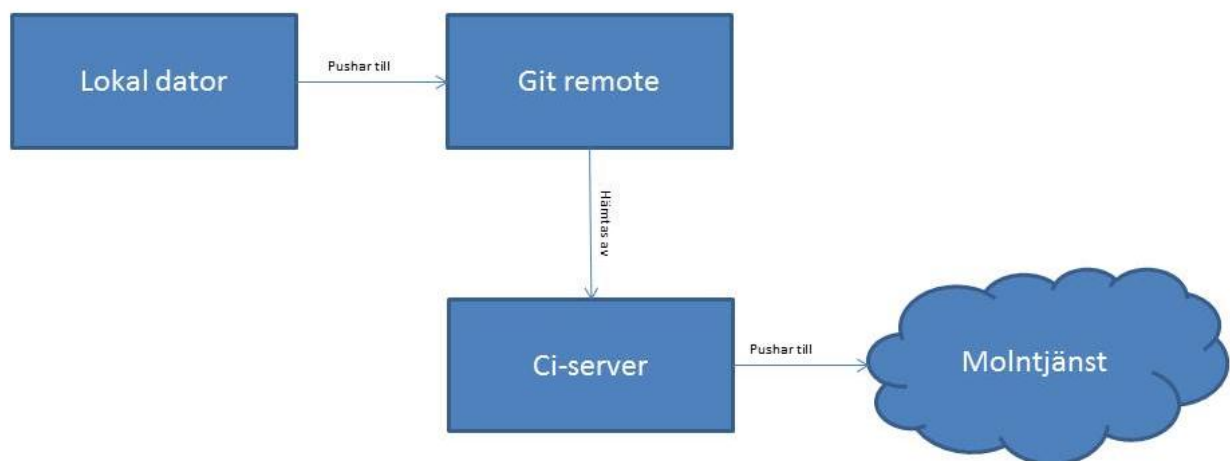
Proc-filen som ligger i applikations-projektets rot består i detta projekt av en enkel rad för att starta applikationen i en webapprunner:

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

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

Allmän Struktur

Den allmänna strukturen för projektet beskrivs av bilden nedan:



- 1 Ny kod pushas till git remote repository.
- 2 Ci-servern upptäcker ändringen och hämtar ner den senaste koden.
- 3 Ci-servern bygger den senaste programkoden.
- 4 Om programmet byggts korrekt triggas ett annat projekt på ci-servern som genomför seleniumtester.
- 5 Om även dessa tester går igenom pushas koden till molntjänst för nytt bygge där och exekvering av webapplikationen.

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

Projektstruktur

Eftersom Heroku kräver att den applikation som ska byggas har sin src-katalog i rooten för den git-katalog som pushas till Heroku är projektet inte ett multi-modulprojekt. Istället är applikationen och

integrationstestningen två separata projekt.

Jenkins bygger applikationen när versionskontrollsystemet uppdateras och om bygget går bra triggas testprojektet att köra acceptanstester. Därefter kan testprojektet, vid lyckade tester, deploya applikationen på molntjänst om detta önskas vara automatiskt. Dock kan testerna uppdateras utan att applikationen uppdateras och önskemålet är då att köra testerna på den senaste versionen av applikationen. Lösning är att låta testprojektet också polla versionskontrollsystemet och köra om dessa uppdateras.

Eftersom testen i testprojektet är skrivna i Selenium och körs direkt mot en deployad webapplikation måste denna inte köras under fasen test som de normalt skulle göra. Container startas under maven-fasen pre-integrationtest och stoppas under post-integrationtest varför testen bör köras under fasen integration-test när container är startad. För detta används verktyget Surefire i vilket anges att testen ska skippas under den normala testfasen men köras under integrations-testsfasen.

Applikationen

Applikationen är ett lastbokningssystem.

Cucumber

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

```
Feature: ReserveLoad
```

```
Scenario: Reserve a load
```

```
    Given a load with content coal
```

```
    When you reserve it
```

```
    Then the load will be reserved for you
```

Varje steg kopplas ihop med en bit kod som översätter klartexten till program-kod.

Dock har applikationen i detta projekt ingen riktig beställare vilket gör att en del av syftet med testerna försvinner lite. Dock förenklar det även för programmerarna att ha klartext över vad det är som ska testas, vilken funktionalitet.

Testning och testteckning

I dagsläget används två typer av tester. Den första typen är enhetstestning. Dessa testar funktionaliteten av enskilda klasser och ska gå fort att köra, då de körs ofta, varför exempelvis en riktig databas inte bör användas.

Den andra typen av tester är de Selenium-tester som testar funktionaliteten på UI-nivå. **Som komplement skulle även integrations-test införas för att testa funktionalitet på högsta nivå i modellen med den riktiga databasen inkopplad.**

Tyngdpunkten vid testandet ligger på enhetstesterna som verifierar funktionen. Fowler [6] talar i sin artikel om testpyramiden där han tar upp att end to end-testning, exempelvis ui-test bör hållas till ett minimum och att om ett sådant test fallerar beror detta oftast på för dålig enhetstestning. End to end-test är också bräckliga mot förändringar i gränssnittet och långsamma då de använder sig av en driver för att simulera användarinteraktion.

Ett försök att införa Cobertura för att mäta testteckning men detta gav upphov till en del problem(se Problem). Dessa problem samt att den siffra som erhålls vid mätning av testteckning egentligen inte säger så mycket i sig orsakade att testteckning uteslöts ur projektet. Cobertura mäter hur många

rader i produktionskoden som exekveras av ett test. Dock säkerställs inte ens att något test utförs i testmetoden och testtäckningssiffran ger en dålig bild av kvalitén på testerna.

Lösningsförslagen

Om det fanns alternativa lösningar, beskriv dessa och hur man valde.

Infrastructure as a service

Som ett alternativ till den molntjänst som används i dagsläget skulle en iaas använts istället (se teori). Dock hade detta krävt mer konfiguration samt registrering.

Historik

Vecka 1 – grundläggande kunskap om maven och jsf inhämtades.

Vecka 2 – molntjänsten Heroku används för leverans av prototyp.

Vecka 3 – Ci-server sattes upp.

Vecka 4 – databasstöd till applikationen.

Vecka 5 – exekverbara specifikationer samt Cobertura för testtäckning.

Problem

Helhetskonceptet

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

Problem vid installation av Jenkins

Installationen av Jenkins på linux-maskin orsakade stora problem.

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

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

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

Problem vid installation av databas på Heroku

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

Flyway

För att kunna använda flyway både lokalt, på ci-server samt på molntjänst krävs att inställningar om användarnamn och lösenord specificeras i miljövariabler. Ett problem som uppstod vid angivandet av dessa var att de måste anges med stora bokstäver.

Till en början installerades Flyway som ett maven-plugin. Detta skapade en mängd olika problem som att migrationen inte fungerade om maven startades med kommandot `clean install` utan endast `install` måste anges.

Dessutom var det stora problem med att få igång FlyWay på molntjänsten samt att läsa ut miljövariabler till maven på molntjänsten. Efter att detta tillvägagångssätt ändrats till att istället från programkoden invokera önskade ändringar fungerade FlyWay ganska snart på både lokal dator samt på molntjänst. Dock uppstod problem med autentiseringen på ci-servern.

Problemet denna gång var att en tidigare version av postgresql redan var installerat på systemet varför port 5432 som är standard var upptagen. Den nya versionen lade sig då på port 5433 vilket inte var fallet på övriga system där databasen skulle migreras.

Rättigheter på Linux

Ett återkommande problem har varit att jenkins-användaren förlorar rättigheter till den katalog (target) där byggda artefakter lagras. Eftersom maven-kommandot som körs på ci-servern är `clean install` misslyckas operationen under den första livscykeln `clean`. För att ta bort katalogen target krävs skrivrättigheter men av någon anledning har dessa rättigheter automatiskt ändrats.

Operativsystemsskillnad

Vid ett tillfälle märktes tydligt att olika operativsystem kan orsaka problem. Dock är det möjligt att detta är en fördel vid användandet av molntjänst eftersom det ofta är oklart vilket operativsystem som används på molntjänsten, alternativt att ett operativsystem används som det är svårt att få tillgång till i den lokala utvecklingsmiljön.

Det som hände vid detta tillfälle var att en refaktorering genomfördes på en klass som hette `WrongPassWordException` som bestod i en namnändring till `WrongPasswordException`. Dock gick det fel på vägen mellan lokal dator och ci-server. Både på github och ci-servern ändrades namnet på klassen till det nya angivna. På lokal dator ändrades dessutom namnet på källfilen. Ändringen på källfilen pushades dock aldrig upp till github, förmodligen beroende på att windows-versionen av git inte såg detta som någon ändring då windows i grunden inte är case sensitive.

Cucumber

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

Cobertura på molntjänst

Heroku bygger applikationer utan hänsyn till test genom att ange växel `-DskipTests=true`. Detta medför att cobertura stoppar bygget eftersom testtäckningen blir 0%.

En allmän och fin lösning på detta hade kunnat vara att läsa av om maven körts med växel `-`

`DskipTests=true` och i så fall inte låta cobertura-testerna stoppa bygget. Dock finns det ingen dokumentation om hur detta skulle genomföras. Profiler är inte heller en möjlighet eftersom det inte går

att styra över med vilka växlar Heroku kör Maven. Ett tredje alternativ skulle kunna vara att använda ett plugin i Jenkins som sköter kontrollen men problemet då är att det inte går att testa lokalt först om testtäckningen är tillräcklig.

Resultat

Vilka typer av molntjänster finns?

Det finns tre olika typer av molntjänster (se teori)

Vilken typ av molntjänst passar för det här projektet?

Valet står mellan Platform as a service och infrastructure as a service. På grund av enkelheten att komma igång samt kostnadsfriheten vid begränsad användning valdes Platform as a service och mer specifikt molntjänsten Heroku.

Vilket stöd finns för införandet av databas?

Heroku stödjer Postgres och kostnadsfritt ingår in liten, ej konfigurerbar databas. Heroku fungerar även ihop med Hibernate. Vissa problem kan uppstå vid införandet av databas eftersom användaruppgifter för databasen på Heroku ändras dynamiskt. Därför måste dessa läsas ut från miljövariabler och eftersom koden bör vara likadan lokalt som på Heroku måste dessa miljövariabler införas även lokalt.

Hur ska uppdateringar av databasen ske?

För uppdatering och migrering av databasen kan verktyget Flyway användas. Med fördel används Flyway via programkod vilket gör att applikationen ansvarar för att dess egen databas är korrekt.

Hur ser relationen mellan versionshanteringssystem och molntjänst ut?

Från ci-servern skapas en remote uppkoppling mot ett repository på Heroku. När ett projekt pushas till Heroku mastern byggs projektet och startas automatiskt på Heroku.

Vad har en ci-server för uppgift och var kommer den in i strukturen?

Ci-servern bygger hela projektet vid ändring på git-repositoryt. Sedan kör ci-servern nödvändiga tester och pushar projektet till Heroku om testerna gått igenom.

Rekommendationer

Säkerhet

Säkerheten i applikationen följer inte angivnen standard för jsf-applikationer beroende på sättet den körs på. En förbättring som förmodligen skulle behöva genomföras vid lansering av applikationen är ett bättre säkerhetssystem.

Utökade tester

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

typer av tester för verifiera helhetskonceptet.

Diskussion

Detta projekt har varit utmanande då det har handlat om de delar som jag haft minst kunskap om på förhand. Istället för algoritmer och design-mönster, som är det jag tidigare fokuserat på, har det handlat om att få verktyg att fungera och få ihop en modell för leverans. Det har gjort att inlärningskurvan har varit brant.

Att arbeta med java ee stacken skiljer sig mycket mot att exempelvis utveckla en swing-applikation i en lokal miljö. Fokus ligger mycket på de verktyg som används vid byggandet och testandet av applikationen. I detta projekt har en trivial applikation utvecklats mest för att testa tillvägagångssättet att utveckla en applikation varför inga tyngre algoritmer förekommer. Det svåraste var att initialt greppa syftet med olika verktyg och tillvägagångssätt.

Att arbeta i korta iterationer med fokus på en sak i taget passar mig mycket bra. Det är mycket enklare att jobba om man jobbar mot ett tydligt mål.

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

Väldigt mycket tid lades på att få databasmigrationen att fungera och 90 % av tiden var felsökning. Det är mycket frustrerande att under en halv vecka i princip inte alls komma framåt i projektet. Dock är detta faktorer man får räkna med när projektet handlar om att ta fram en miljö snarare än att utveckla produktionskod. Vid utvecklingen av produktionskod kan ofta problem isoleras och utvecklingen kan fortfarande fortgå. Vid användandet av verktyg krävs det att man gör på tänk sätt och när fel uppstår måste dessa lösas innan man kan gå vidare.

Slutsatser

Kontinuerlig leverans och sessioner

Kontinuerlig leverans går ut på att leverera en produkt ofta, i extremfallen många gånger per dag. Ofta är också denna applikation en hemsida som använder sig av sessioner för att hålla reda på värden som behöver överleva sidladdningar. Sessionsinformation sparas i normalfallet på servern men med kontinuerlig leverans orsakar detta problem eftersom "servern" startas om vid varje uppdatering. Sessionerna går då förlorade och ett felmeddelande visas om en redan inloggad användare försöker komma åt en sida. Ett alternativ till att spara sessionsinformation på servern är att låta klientdatorn hålla reda på informationen. Med jsf anges detta enkelt i en fil som heter web.xml. Detta gör att felmeddelandet försvinner men den information som visas för användaren är den från startsidan så sessionen förefaller ändå inte överlevt.

Heroku

Heroku är en lättanvänd molntjänst om än anpassat efter ett smalt användningsområde. Eftersom Heroku levererar en färdig plattform måste projektet vara utvecklat på en plattform som stöds samt med fördel hanteras på versionshanteringssystemet Git.

Vid användandet av molntjänst får ofta den lokala utvecklingsmiljön anpassas efter miljön på molntjänsten. Om exempelvis miljövariabler som läses ut i run-time är angivna enligt ett mönster på molntjänsten måste dessa anges på samma sätt i den lokala miljön för att möjliggöra utläsning i run-time.

Kontinuerlig leverans

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

Maven

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

Externa verktyg

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

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. För en web-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 finns inga installerade konfigurationsfiler för servern varför användandet av realmer blir krångligt om möjligt. Istället används i nuläget i projektet en böna med sessions-skop som håller reda på inloggad användare. På varje sida görs en kontroll av inloggad användare. Är detta värde null skickas besökaren till inloggningssidan. Vid inloggning kontrolleras angivna värden mot databasen som i detta fall lagrar användare.

Selenium

Selenium har fördelen att testen som skrivs med hjälp av Selenium testar funktionalitet på allra högsta nivå, användarnivå. 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.

Referenser

- [1] Jez Humble and David Farley: Continues Delivery
- [2] James Ward Webapp Runner – Apache Tomcat as a Dependency Tillgänglig på:
<http://www.jamesward.com/2012/02/15/webapp-runner-apache-tomcat-as-a-dependency> (websida)
- [3] <https://devcenter.heroku.com/articles/spring-mvc-hibernate>
- [4] <http://www.heroku.com/>
- [5] Mary Poppendieck: Lean Leaders Workshop (presentation)
- [6] Martin Fowler TestPyramid <http://martinfowler.com/bliki/TestPyramid.html>

Bilagor

Dokument, diagram, etc. som inte "platsar" i den löpande texten i rapporten. T.ex. kravspecifikationen, avtal, o dylikt.