

# Arkitekturdokument

Easy Recruit

Företag: Disciplines of Lindbäck

Författare: Alexander Lundh, Kim Hammar, Marcel Mattsson

# Innehållsförteckning

<b>4. Introduktion</b>	4
<b>5. Functionality view</b>	5
5.1 Funktionalitet för ansökande	5
5.2 Funktionalitet för rekryterare	10
5.3 Användargränssnitt	10
<b>6. Design view</b>	11
6.1 Överblick	11
6.2 Klient	12
6.3 Presentation	13
<i>Anledning</i>	
<i>Användning</i>	
<i>Sammansatta vyer</i>	
<i>Internationalization and localization</i>	
<i>Validering av användar-data</i>	
<i>Navigering</i>	
<i>Sessions hantering</i>	
<i>Data Transfer Object (DTO)</i>	
6.4 Business	19
<i>Controller</i>	
<i>Designmönster (Model-View-Controller)</i>	
<i>Enterprise JavaBeans</i>	
<i>Tillstånd i modellen</i>	
<i>Synkronisering med databasen</i>	
<i>Transaktionshantering</i>	
<i>Felhantering</i>	
6.5 Integration	22
<i>Data Access Object (DAO) - Design Mönster</i>	
6.6 Code Conventions	24
<i>Filorganisation</i>	
<i>Indentering</i>	
<i>Kommentarer</i>	
<i>Deklarationer</i>	
<b>7. Security view</b>	27
7.1 Kryptering av nätverkstrafik	27
7.2 Lagring av användares lösenord	27
7.3 Autensiering	28
7.4 Auktorisering	28
7.5 Loggning av misslyckade inloggningsförsök	28

7.6 Säkerhetsproblem.....	28
<b>8. Data section.....</b>	<b>30</b>
8.1 Databas tabeller.....	30
<i>Person</i>	
<i>Applications</i>	
<i>Expertise</i>	
<i>Hela databasen och tabellernas relation till varandra</i>	
8.2 O/R Mapping.....	31
<b>9. Non-Functional view.....</b>	<b>33</b>
9.1 Felhantering.....	33
9.2 Loggning.....	34
<i>Container hanterad loggning</i>	
<i>Loggning av application-exceptions</i>	
<i>Vad som loggas</i>	
<i>Logg-destination</i>	
<i>Konfigurerings</i>	
9.3 Transaktioner.....	38
<i>Rollbacks</i>	
9.4 Prestanda och tillgänglighet.....	39
<i>Tillgänglighet</i>	
<i>Prestanda</i>	
9.5 Code Coverage.....	42
<i>API-Dokumentation</i>	
9.6 Testning.....	43
<i>Strategi vid testning</i>	
<i>Enhetstester</i>	
<i>Varför Mocking?</i>	
<i>Integrationstester</i>	
<i>Acceptanstester</i>	
<i>Webbläsartestning</i>	
<i>Code coverage</i>	
<i>Lägga till nya tester</i>	
<i>Lokation</i>	
<i>Namnkonvention</i>	
<i>Enhetstester struktur</i>	
<b>10. Deployment view.....</b>	<b>49</b>
10.1 Fysiska noder.....	50
10.2 Kommunikation mellan noderna.....	50

<b>11. Implementation view</b>	51
11.1 Produkt	51
11.2 Hur man sätter upp och kör applikationen med Netbeans	51
<i>Sätta upp miljö och bygga projektet</i>	
<i>Skapa en lokal databas i Netbeans</i>	
<i>Kör applikationen</i>	
<i>Problem som kan uppstå vid skapande av databas.</i>	
11.3 Hur man bygger och kör applikationen från kommandoraden	53
<i>Bygga projektet</i>	
<i>Kör applikationen</i>	
<b>12. Problem</b>	55
12.1 Automatisering av integrationstester	55
<i>Avvisad lösning</i>	
<i>Förslag på lösning</i>	
<b>13 Källor</b>	56
<b>14. Bilagor</b>	57
14.1 Företagets existerande databasstruktur (gammal)	57
14.2 Webbläsar-testning	57
<i>Internet Explorer 10</i>	
<i>Firefox 26</i>	
<i>Chrome 31</i>	
<i>Olika skärmupplösningar</i>	
<i>800x600</i>	
<i>1136 x 640 (Iphone 5)</i>	

## 4. Introduktion

Denna applikation är skapad för att låta personer ansöka till olika positioner inom ett företag, den är skapad åt företaget i fråga och är utvecklat på KTH Kista.

Applikationen tillåter applikanter att registrera ett konto och därefter skapa ansökningar där de kan söka olika positioner som finns tillgängliga inom företaget och även välja inom vilka datum som ansökande är tillgänglig. Applikanter ska även kunna lägga in erfarenheter och information om sig själva när denna ansökan görs.

Denna information som den ansökande skapar ska kunna läsas av rekryterare. Rekryterare ska kunna söka efter applikanter med hjälp av parametrar så som roll och tillgänglighet. Rekryterare ska även kunna acceptera och neka ansökningar efter att en ansökan har gåtts igenom.

Applikationen skall vara ett verktyg för att underlätta processen för företag/rekryterare att hantera inkommande applikationer.

## 5. Functionality view

Den här sektionen beskriver den huvudsakliga funktionaliteten av systemet, dvs vad användaren kan göra med applikationen.

Rekryteringssystemet stödjer två olika sorters användare, ansökande och rekryterare. De två användarna har skilda användningsområden av systemet som medför att systemets funktionalitet kan delas in i två delar:

- Ansökande som registrerar jobb-applikationer
- Rekryterare som administrerar jobb-applikationer.

### 5.1 Funktionalitet för ansökande

Användare som vill ansöka till rekryteringssystemet kan registrera sig för att få ett personligt konto, se fig 5.1. För att kunna göra en ansökan är det ett krav att användaren är inloggad då det är på det sättet som rekryteraren senare kan identifiera vem som har gjort vilken ansökning.

Efter registrering så kan användaren logga in i systemet där personen har möjlighet att göra ansökningar, se fig 5.2. Med att göra en ansökning menas att användaren specificerar sin kompetens-profil samt datum då användaren har möjlighet att jobba.

# Rekryteringssystem

[Logga in](#)

Var god registrera dig

**Namn**

**Efternamn:**

**Personnummer**

**Email**

**Användarnamn**

**Lösenord**

Company  
name@Copyright  
2016



Fig 5.1. Screenshot från gränssnittet där användaren kan registrera sig som ansökande.

När användaren är inloggad så finns två primära use-cases att välja emellan. Göra ansökningar och lista sina existerande ansökningar se fig 5.2. I Fig 5.3 visas formuläret där användare fyller i sina ansökningar och fig 5.4 visar hur det ser ut när en användare listar sina applikationer.

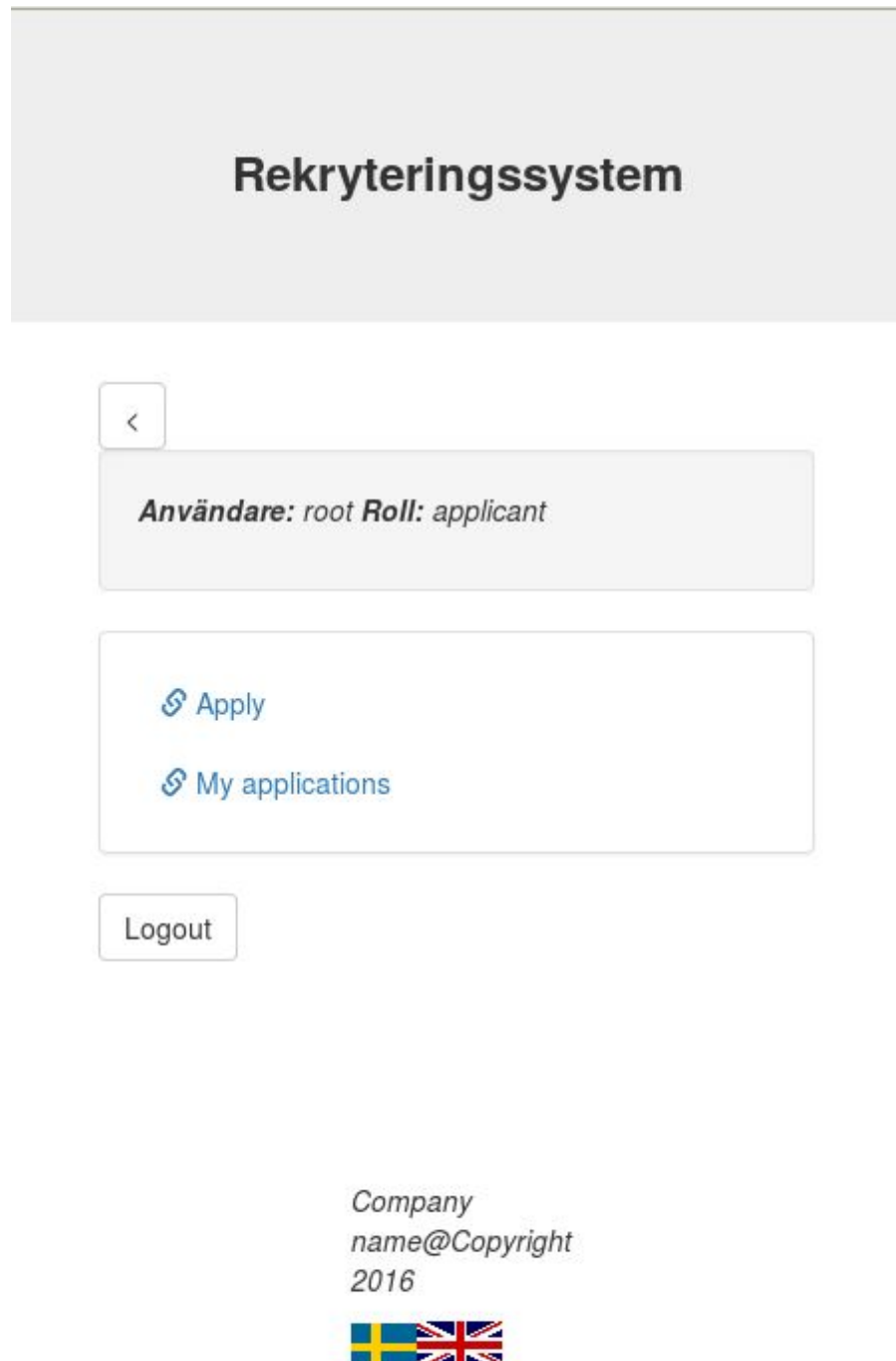


Fig 5.2. Screenshot från "home-page" för en inloggad applicant.



## Rekryteringssystem



Create job-application

Choose your area of expertise

Korvgrillning

Years of experience in the area:

2.0

Available from

06/01/2016

Available to

04/01/2016

Your Application

Expertise: Korvgrillning

Years of experience: 2.0

Availability period: Wed Jan 06 00:00:00 CET 2016 -  
Fri Apr 01 00:00:00 CEST 2016

Hand in application

Cancel

Company  
name@Copyright  
2016



Fig 5.3. Screenshot från formuläret där användare kan registrera applikationer.

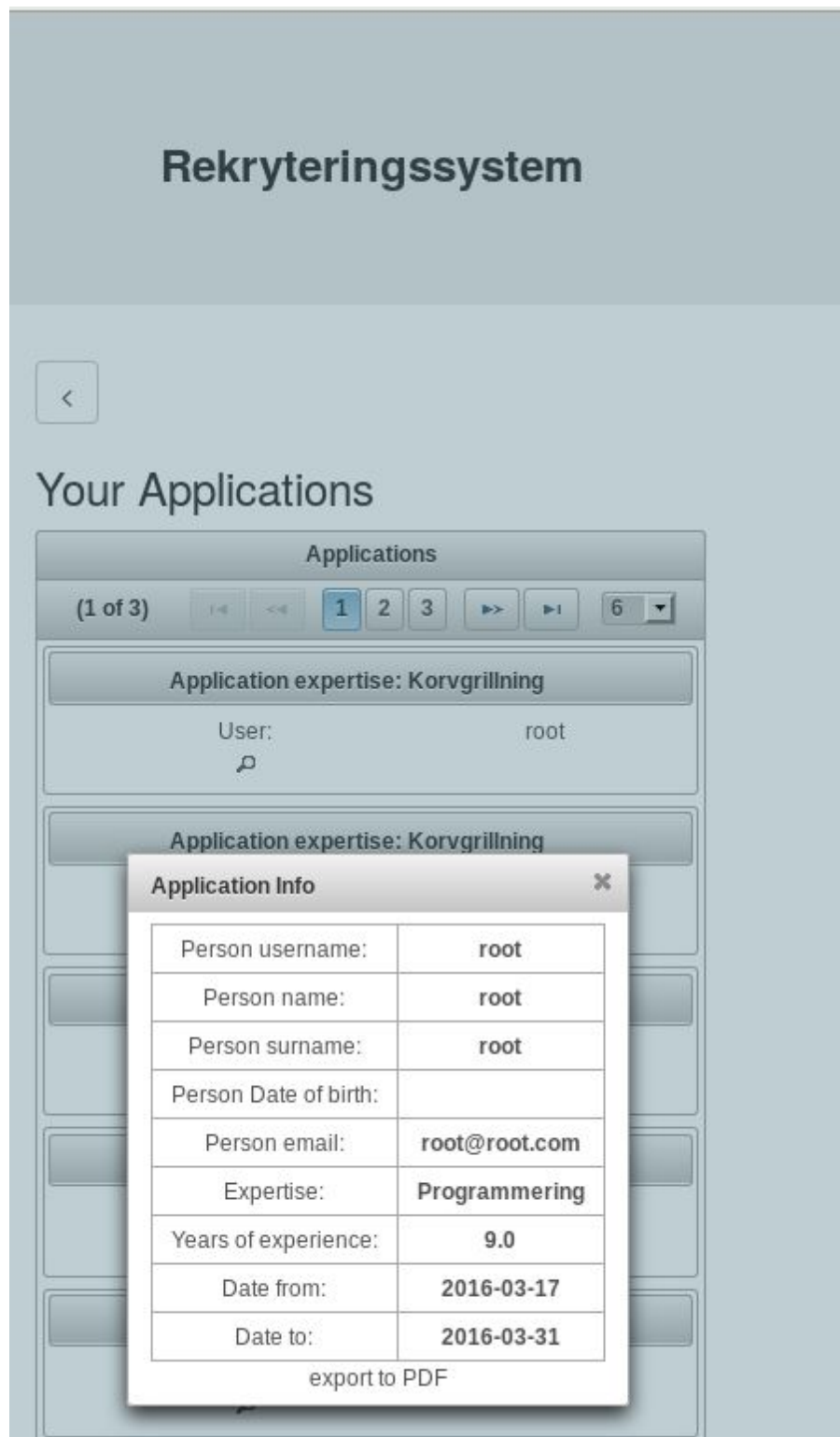


Fig 5.4 Screenshot från sidan där användare kan lista sina applikationer.

## 5.2 Funktionalitet för rekryterare

Rekryterare kan använda systemet för att administrera och söka bland applikationer. För att rekryteraren skall få tillgång till funktionaliteten så måste personen först logga in så att systemet kan autentisera och auktorisera användaren.

Notera att rekryterare har andra rättigheter än ansökande och det krävs att rekryteraren loggar in på ett konto som har den behörigheten.

Gränssnittet som rekryterare användare ser i stort sett ut precis som gränssnittet för ansökande (se fig 5.1-5.4), med skillnaden att viss funktionalitet skiljer sig mellan de två.

## 5.3 Användargränssnitt

Användaren kan interagera med systemet via en webbläsare. Systemet är utvecklat i syfte att webbläsaren skall vara det primära verktyget att interagera med systemet, både för ansökande och rekryterare. Webbläsare som aktivt stöds är:

- Google Chrome fr.o.m. version 31
- Mozilla Firefox fr.o.m. version 26
- Internet Explorer fr.o.m. version 10

I Bilaga A hittar du resultatet från den senaste webbläsartestningen för dessa webbläsare.

## 6. Design view

Den här sektionen beskriver hur systemet är strukturerat samt vilka design-mönster och principer som har följts under utvecklingen.

### 6.1 Överblick

Systemet arkitektur är baserat på lagermönstret och är indelat i fyra huvudsakliga lager med olika syften:

- Klient. Detta lager finns på klient-sidan och är systemets vy gentemot användaren.
- Presentation. Detta lager är systemets gränssnitt gentemot klient-lagret.
- Business. Innehåller systemets business logik.
- Integration. Ansvarar för systemets persistens. Hanterar lagring i databasen.

Lagerindelningen kan ses på som en "stack" där Klient lagret är överst och Integrations lagret är nederst. Varje lager har endast en koppling till det närmast undre-lagret, detta medför att flödet genom systemet har en enskild riktning och minskar komplexiteten då det är väldefinierat vilka lager som är beroende av varandra. Business logiken blir inte mixad med vyn eller persistenslagret, detta medför att i framtiden så skulle business logiken kunna återanvändas utan modifikation i en applikation med en helt annan vy och persistens. Med lagerindelning blir det alltså enklare att utöka programmet och att modifiera delar av det.

Syftet med lager- designen är att uppnå en design som tillåter att särskilja olika delar av systemet till "isolerade" komponenter som kan kommunicera med varandra genom väldefinierade gränssnitt. Eftersom dessa komponenter är i största mån oberoende av varandra så kan de utan större ansträngning återanvändas in andra system samt bytas ut vid behov.

Ett av huvudmålen med design-mönster är att isolera förändringar i din kod (Eckel, B. 2003). Vi kan förvänta oss att olika delar av systemet kommer behöva modifieras i framtiden då kraven förändras och nya lösningar uppträder. När behovet av att förändra en viss del av systemet uppstår så vill vi undvika att ändringen skall propagera genom koden för hela systemet. Lager- arkitekturen minskar kopplingen mellan olika delar av systemet samt underlättar uppdelning av arbetet mellan flera utvecklare (Lindbäck, L. 2016). Lagerindelning leder även generellt till kod som är bättre lämpad för enhetstester.

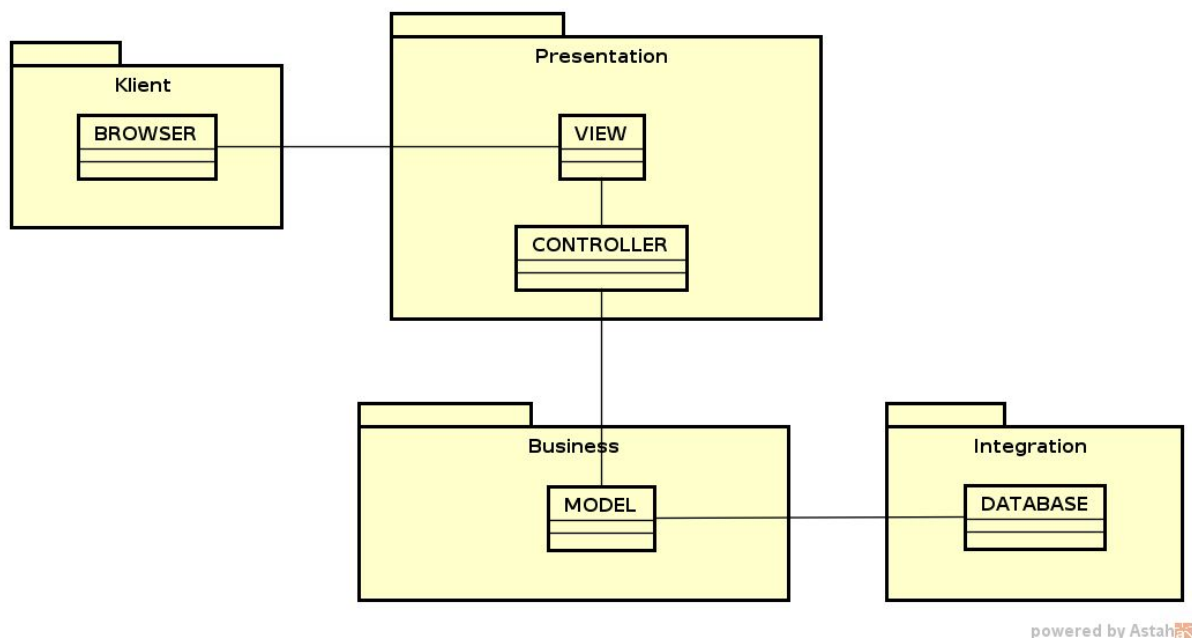


Fig. 6.1 Lager indelningen i systemet samt dess beroenden.

## 6.2 Klient

Klient lagret i applikationen är en webbläsare. Användaren använder applikationen genom att använda en webbläsare och surfa in på hemsidans URL. Användaren kan sedan interagera med sidan genom vy-komponenter som ursprungligen är utvecklade med XHTML, vy komponenterna genererar sedan requests till servern vid behov och presenterar dynamiskt innehåll som skapats i presentations lagret.

Kommunikationen mellan klienten (webbläsare) och servern görs över HTTP/S protokollet, kommunikationen bygger på en generell strategi för klient-server applikationer vilket innebär att klienten skickar requests till servern som svarar. Ett request kan antingen vara ett request för en hel resurs (en sida) eller ett partialt-request, vilket typiskt är ett request för en bit dynamisk data. Ett request för en hel sida behöver inte kodas explicit utan det sker dynamiskt av webbläsaren varje gång en ny URL skrivs in eller om användaren trycker på en länk/komponent som på server-sidan genererar en redirect till en ny sida. Ett partialt request måste däremot specificeras explicit i vy-koden, i detta projekt så används tekniken AJAX för att göra dessa requests. Eftersom ramverket JSF används i detta projekt så görs AJAX-anropen via JSF-taggar som har inbyggt stöd för AJAX, se figur 6.2 för ett exempel på detta.

```

<b:selectOneMenu value="#{applyBean.expertise}" id="expertise" onchange="submit()">
  <f:ajax execute="expertise" render=":content"/>
  <f:selectItems value="#{applyBean.expertiseList}" />
</b:selectOneMenu>

```




Fig 6.2 Kodutdrag som illustrerar hur partiella-requests från webbläsaren till servern görs via AJAX m.h.a. JSF-komponenter. Detta kodutdrag kommer innebära att när selectOneMenu används av användaren så kommer ett partiellt request skickas till servern och sedan uppdateras vyn dynamiskt, utan att sidan behöver laddas om. Specifikt så är det komponenten med id **content** som uppdateras dynamiskt.

Anledningen till att en webbläsare är den primära klienten som är tänkt för systemet är dels för att produktägaren framfört önskemål om det och för att det finns ingen god anledning att utveckla för en specifik plattform eller enhet, då är webbläsaren det självklara valet p.g.a. sin tillgänglighet och anpassning för olika typer av enheter.

## 6.3 Presentation

Presentationslagret finns på servern och innehåller all kod som har med vy-hantering att göra. Presentationslagret ansvarar b.l.a. för navigering, validering av användar data, flödeskontroll i applikationen, sammansatta vyer, internationalisering och lokalisering, sessionshantering, säkerhet (autensiering och auktorisering) och säker kommunikation med webbläsare.

Eftersom detta projekt är en Java EE applikation som använder ramverket JSF så innehåller presentationslagret framförallt Managed Beans som ansvarar för att presentera data till användaren samt skapa dynamiska vyer.

### Användning av UI Komponent Ramverket JSF:

#### Anledning

I detta projekt har JSF valts som framework för hantering av UI komponenter. Detta val grundar sig i stor del främst av tre anledningar.

#### MVC

JSF är ett komponent baserat MVC framework och då projektets struktur grundas i MVC så är användandet av JSF en perfekt matchning. Användandet av återanvändbar kod skapar en miljö som utvecklarna är vana vid.

#### Bekantskap

Utvecklarna har en bakgrund med JSF och Java EE, detta gör att valet även ser till att mindre förstudier krävs för att sätta sig in i vad som behöver göras och erfarenheten från tidigare projekt kan nyttjas.

#### Enkelhet

JSF har en hög inlärningskurva men när strukturen är där gör den väldigt mycket själv

såsom request parametrar, validering och uppdatering av modelvärden. Detta gör att mer tid kan läggas på andra delar vilket leder till en bättre applikation.

## Användning

JSF möjliggör konstruering av dynamiska vyer i en Java EE web applikation. Genom att utveckla presentationslagret med JSF ramverket så blir presentationslagret ett gränssnitt mellan servern och klienten. Presentationslagret hanterar requests och bestämmer vad som skall renderas tillbaka till klienten.

JSF använder sig av “Managed Beans” (Oracle, 2. 2016) för att hantera klient-requests och om nödvändigt, generera anrop ner till applikations kontrollern. Generellt så representerar en managed bean någon typ av dynamisk data som visualiseras m.h.a. XHTML-sidor på klienten. För att binda komponent-värden och objekt i XHTML-sidorna till managed beans eller för att referera metoder från komponent taggar så kan “Expression Language användas”. Exempel på hur detta kan se ut ges i fig 6.3.

```
<b:commandButton styleClass="logOutButton" value="" action="#{loginBean.logout()}">
```

Fig 6.3. I kontexten av denna bild så är “loginBean” en managed bean som innehåller en metod logout(). Genom att specificera detta i action-attributet m.h.a. Expression Language på detta XHTML-element så kommer den metoden att anropas när användaren trycker på knappen.

Managed beans skapas på liknande sätt som en vanlig java klass med undantagen att managed bean måste ha ett specificerat “scope” samt att de med fördelas kan namnges så att de kan refereras från XHTML-sidorna med hjälp av expression language.

## Sammansatta vyer

Som resultat av att vyn representerar en enhetlig hemsida där det enda som skiljer sig mellan olika sidor är själva innehållet och att designen, headern och footern är densamma på alla sidor, så används en teknik av sammansatta vyer för att undvika redundans.

Tekniken bygger på s.k “templates”, genom att fördefiniera en template med statisk header och footer så kan samma template återanvändas för flera sidor och det enda som behöver specificeras är själva innehållet på sidan. För tillfället finns endast behov av en template men det är även möjligt att utöka antalet templates om det skulle önskas.

Fördelen med sammansatta vyer är framförallt att det minskar kodmängden eftersom man undviker upprepning. Så här ser processen av att lägga till en ny vy-sida som använder en template ut:

1. Skapa en ny XHTML-fil och spara den med önskat namn.
2. Specificera vilken template som skall användas för filen genom att använda jsf-taggen “composition”.
3. Definiera vilket huvudinnehåll som skall visas på sidan utöver templatens genom att använda jsf-taggen “define”.

Fig 6.4 visar hur koden för home-page sidan utnyttjar en definierad template.



```
8      <!-- HOME-page -->
9      <ui:composition template="/templates/default/main.xhtml">
10         <ui:define name="content">
11             <b:panel>
12                 <b:listLinks>
13                     <b:navLink href="applicant/index.xhtml" v.
14                     <b:navLink href="recruit/index.xhtml" val
15                     <b:navLink href="login.xhtml" value="#{me:
16                 </b:listLinks>
17             </b:panel>
18         </ui:define>
19     </ui:composition>
```

Fig 6.4. På rad 9 så specificeras vilken template som skall användas. Mellan rad 10 och 18 definieras sidans huvudinnehåll.

Konventionen i detta projekt är att templates definieras i en separat mapp "templates". Mapp-hierarkin för vy-koden ser ut som följande:

- WEB-INF
  - Konfigurationsfiler (ex beans.xml, web.xml)
- applicant
  - Filer som kräver autensiering som "applicant" för att få åtkomst
- recruit
  - Filer som kräver autensiering som "recruit" för att få åtkomst
- resources
  - css
    - stylesheets
  - images
- templates

## Internationalization and localization

Systemet har utvecklats med stöd för att addera flera olika språk att visa användar gränssnittet i. I gränssnittets koden används generiska placeholders för nyckelorden, dessa kommer sedan bytas ut beroende på vilket språk som är valt i webbläsaren.

Språken som stöds specificeras i en bundle där filer för översättning mellan nyckelorden i systemet finns specificerat, se fig 6.5. Det språk som är aktivt väljs av användaren och lagras i en managed bean som ansvarar för att byta språk när användaren vill det. I detta projekt så är denna managed bean namngiven som "LocaleManager". Beanen är kopplad till länkar i vyn som användaren kan trycka på för att byta språk, se fig 6.6.

Genom att flytta ut nyckelorden till separata filer så undgås redundans eftersom man slipper ha flera olika sidor för varje språk (vilket skulle vara fallet om nyckelorden var hårdkodade i



presentations-koden). Att flytta ut nyckelorden gör det även smidigt att lägga till flera språk i framtiden.

Title=Recruitment System	Title=Rekryteringssystem
Username=Username	Username=Användarnamn
Password=Password	Password=Lösenord
Applicant=Applicant	Applicant=Ansökande
Recruiter=Recruiter	Recruiter=Rekryterare
LoginReg=Login/Register	LoginReg=Logga in/Registrera
Name=Name	Name=Namn
Surname=Surname	Surname=Efternamn
ssn=ssn	ssn=Personnummer
Email=Email	Email=Email
Login=Login	Login=Logga in
Register=Register	Register=Registrera
LoginTitle=Please Signin	LoginTitle=Var god logga in
RegisterTitle=Please Register	RegisterTitle=Var god registrera dig
Logout=Logout	Logout=Logga ut
LoggedIn=You are already logged in.	LoggedIn=Du är redan inloggad.
Role=Role	Role=Roll
User=User	User=Användare

Fig. 6.5. Bundles för översättning mellan olika språk att visa webbsidorna med.

```
<h:form>
  <h:commandLink action="#{localeManager.changeLocale}">
    <f:param name="languageCode" value="sv" />
  </h:commandLink>
</h:form>
```

Fig. 6.6. länk i presentationskoden för att anropa localeManager och byta språk till svenska.

Vilken bundle som applikationen skall använda för översättning specificeras i konfigurationsfilen faces-config.xml.

## Validering av användar-data

För validering av användar-data så används ramverket *Bean Validation Framework*. Bean validation ramverket bygger på valideringsregler som man specificerar för fält eller parameterlistor som representerar användardata som ska valideras, se fig 6.7, understruket i rött. Ramverket har en rik uppsättning av fördefinierade regler (Oracle, 1. 2013) som kan användas utan vidare, men vill man ha mer skräddarsydda valideringar för vissa fält så kan man specificera sina egna regler.

```

@Size(min=1, message="You need to enter a name")
private String name;
@Size(min=1, message="You need to enter a surname")
private String surname;
private String ssn;
@ValidEmail
private String email;
@Size(min=2, max=16, message="username needs to be between"
    + " 2 and 16 characters long")
private String username;
@Size(min=6, max=16, message="Password needs to be between"
    + " 6 and 16 characters long")
private String password;

```

Fig 6.7. Valideringsregler för användar-data. Notis: @ValidEmail är en egenutvecklad valideringsregel medan @Size är en fördefinierad regel.

I detta projekt har Bean Validation Framework valts för att det integrerar väl med JSF samt att det separerar validerings-logik från UI-koden. Bean Validation ramverket fungerar även väl för enhetstestning med JUnit som är det ramverk som används för enhetstestning i projektet.

## Navigering

Simpel navigering mellan olika sidor i applikationen görs med hyper-länkar och kräver ingen sofistikerad konfigurering. Dynamisk navigering som sker som resultat av att presentationslagret utför någon action, exempelvis anrop till kontrollern kräver en mer sofistikerad konfigurering. Det genomgående temat i designval för detta projekt är att skapa isolerade komponenter med hög sammanhållning, för att uppnå det i fallet av navigering så vill vi att navigeringen skall vara definierad i en separat modul isolerad från resten av vy-koden, i JSF uppnås detta genom att definiera en uppsättning med navigeringsregler i konfigureringsfilen faces-config.xml, se fig 6.8.

Navigeringsreglerna används av containern för att bestämma vilken vy som skall visas som resultat av en viss action.

```

<navigation-rule>
  <from-view-id>recruit/index.xhtml</from-view-id>
  <navigation-case>
    <if>#{loginBean.logout}</if>
    <to-view-id>/index.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>

```

Fig 6.8 Utdrag ur faces-config.xml som visar ett exempel på en navigeringsregel för sidan recruit/index.xhtml som säger att om metoden "logout" returnerar true så skall användarens direkteras till index.xhtml.

Exempel på hur man kan lägga till en ny navigeringsregel i projektet:

1. Skapa en boolean-metod i en managed bean som representerar resultatet av aktionen som avgör vilken typ av navigering som skall följa.
2. Skapa en navigeringsregel i faces-config.xml
  - a. Definiera för vilket sida denna navigeringsregel gäller (from-view-id)
  - b. Definiera metoden i managed bean som avgör utfallet av navigeringen (if)
  - c. Definiera navigeringsutfallet (to-view-id)

## Sessions hantering

HTTP är ett tillståndslöst protokoll, detta medför att sessionshantering är növäändigt för att "komma ihåg" användare, annars skulle användarna behöva identifiera sig med användarnamn och lösenord för varje request till servern. En session startar när användaren besöker sidan för första gången och avslutas antingen av att användaren stänger ner sin webbläsare eller att användaren trycker på en "logga ut" knapp som medför att servern avslutar sessionen explicit.

Eftersom att webbapplikationen lever i en "web-container" som körs på applikationsservern så implementeras sessionshanteringen genom att accessa HTTP-parametrar från den hanterande containern. Sessionshantering går till så att när användaren genomför en lyckad inloggning så uppdateras sessions-attribut om användarens användarnamn, roll och ID, dessa attribut kommer sedan att "kontrolleras" vid varje aktion som användaren genomför som kräver auktorisering.

Sessionshantering medför en del säkerhetsaspekter, mer om detta i sektion 7.

## Data Transfer Object (DTO)

Ett vanligt problem att stöta på vid användandet av lager-mönstret är att, trots att man vill undvika det, så uppstår ett beroende mellan presentationslagret och businesslagret genom att presentationslagret skickar med själva business-logik objektet som beror på parametrar som hämtas från vyn, ner till business lagret (via controllern).

Ett sätt att undvika detta är att istället skicka med alla parametrar i en parameterlista och konstruera objektet i businesslagret istället. Den lösningen blir dock snabbt ohanterlig när parameterlistorna blir för långa, en lösning är då att använda sig av Data Transfer Objects (DTO), och det är även det som används i detta projekt, se fig 6.9.

```
public void register(@NotNull PersonDTO p) throws NoSuchAlgorithmException
{
    em.persist(new Person(p));
}
```

Fig. 6.9. Utdrag från en EJB som hanterar registrering av personer. Utdraget illustrerar hur beroendet mellan presentation och business är eliminerat då business-objektet skapas i business-lagret, och samtidigt så undviks långa parameterlistor genom att använda en DTO.

DTO's är endast menade för att transportera data från presentationslagret till modellen visa kontrollern, alla DTO's bör därför vara oföränderliga (immutable) för att undvika tråd och transaktionsproblem samt göra objekts syfte mer väldefinierat.

## 6.4 Business

Business lagret innehåller systemets applikationslogik. Business lagret ansvarar för om olika tillstånd skall sparas i modellen, när modellen skall synkroniseras med databasen och hur detta skall gå till, transaktionshantering och felhantering.

### Controller

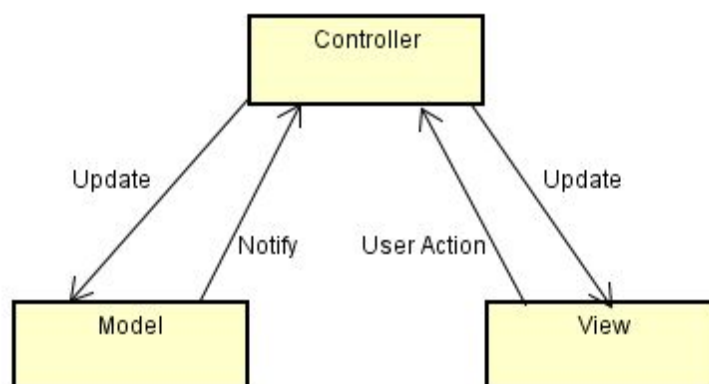
Controllern har som syfte att vara ingångspunkten till systemets business logik. Controllerns uppgift är att delegera (kontrollera) anrop från presentationslagret till business lagret samt att förse presentationslagret med data från business lagret. Controllern inkapsulerar systemets funktionalitet till ett API som används av presentationslagret.

Controllern är nödvändig för att uppnå en design där systemets business logik endast representerar logiken och data, inget annat. Business lagret är inte beroende på varken kontrollern eller presentationslagret, vilket medför en mer flexibel och utvecklingsbar arkitektur.

Idén med en controller är hämtat från Model-View-Controller-mönstret som används i detta projekt.

### Designmönster (Model-View-Controller)

Systemet följer designmönstret Model-View-Controller (MVC) som är ett populärt mönster att följa vid utveckling av user interfaces. Designmönstret delar upp objekten i applikationen i tre olika roller: model, view eller controller. MVC definierar också hur dessa roller skall kommunicera med varandra.



Figur 6.10, Kommunikationen mellan dom olika rollerna i MVC.

**Model** - Direkt hantering av data, logik och regler för applikationen.

**View** - Representerar *user-interface*.

**Controller** - Fungerar som en länk mellan model och view. Tar emot input från view och kommunicerar vidare till model beroende på vilken input den får. För att sedan kommunicera tillbaka ett resultat till view.

I och med att använda detta designmönster, stärker vi underhållbarheten, eftersom objekten får tydliga roller inom systemets funktionalitet. MVC är ett sånt populärt designmönster som medför att många känner igen det eller har jobbat med det sen tidigare.

## Enterprise JavaBeans

Detta system bygger sitt business lager på EJB-ramverket, dvs att applikationslogiken körs i en EJB-container på applikationsservern. Valet av EJB motiveras av att det löser många icke-funktionella problem som gör att applikationskoden får bättre sammanhållning och mindre kod-duplicering.

Utöver att hantera objekts livscykel och beroenden så tar EJB containern även hand om en del icke-funktionella krav så som transaktionshantering (mer om detta i sektion 9 "Non Functional View"), säkerhet m.m så att koden som utför applikationslogiken kan vara fokuserad på dess syfte.

Med EJB-ramverket så består business lagret av flera Enterprise beans där varje bean exponerar ett business-gränssnitt med metoder relaterade till en viss applikationslogik. Denna uppdelning medför att varje EJB får hög sammanhållning och att man enkelt kan utöka modellen vid behov genom att lägga till flera EJB's.

Steg för att lägga till ny business logik i projektet:

1. Skapa en Enterprise Bean i det paket där den hör hemma.
2. Specifiera dess typ samt dess scope.
3. Definiera business gränssnittet
4. Genomför en container hanterad injektion av din Enterprise Bean där den skall anropas (generellt i detta projekt sker dessa anrop från kontrollern).

## Tillstånd i modellen

Modellen består av en uppsättning av EJBs som körs i en EJB-container, om tillstånd skall sparas så måste detta definieras i dess EJBs.

Det finns två typer av fall då tillstånd kan behöva sparas:

1. Modellen behöver spara tillstånd om en specifik användare, exempelvis en virtuell shopping-vagn
  - a. Detta definieras genom att ange scope för EJB'n som `@Stateful`
2. Modellen behöver spara applikationsspecifikt tillstånd som gäller för alla användare.
  - a. Detta kan implementeras med `@Stateless` EJB's.

I denna applikation så behövs inget tillstånd sparas i modellen, varken applikationsspecifikt eller användarspecifikt. Allt tillstånd finns i persistens-lagret. Alltså har alla EJB's i applikationen scope: `@Stateless`.

## Synkronisering med databasen

Synkronisering med databasen sker endast vid de anrop till modellen som berör data som finns i databasen, exempelvis registrering av användare, inloggning osv. I de fallen så

kommer anropet att börja med att hämta data från databasen och avslutas med att uppdatera databasen om nödvändigt.

## Transaktionshantering

Transaktionshantering sker explicit och sköts av EJB-containern. Mer om detta i sektion 9.

## Felhantering

Detta relaterar till stor del till transaktionshanteringen. Beroende på vilken typ av fel som inträffat kan två generella utfall att ske:

1. En rollback av transaktionen kommer att ske och felet rapporteras upp till presentationslagret och visas för användaren. Felet loggas.
2. Ingen rollback kommer att ske utan systemet bedömer felet som ett fel som man kan återhämta sig ifrån och kommer följa en definierad rutin för när fel av denna typen inträffar. Felet loggas.

Inspirerat av EJB's transaktionshantering så namnger vi dessa två typer av fel som "System Exception" och "Application Exception". Vid ett system exception så kommer rollback att ske men inte vid application exception. Generellt så är system exception ett oväntat fel som systemet inte vet hur den skall hantera medans application exception är ett väntat fel, exempelvis att en användare försöker registrera sig med ett redan upptaget användarnamn. Mer om hur dessa exception definieras ges i sektion 9.

Loggningen sker på olika sätt beroende på vilket fel som inträffat. I fallet av ett system-exception så kommer exception att bubbla upp till presentationslagret och loggas av en Interceptor.

Vad gäller application exceptions så kommer dessa exceptions att fångas och hanteras i modellen vilket innebär att även loggningen måste ske i modellen. Mer information om loggningen ges i sektion 9, där definieras även loggnings-nivåerna som systemet använder sig av och som innebär att application exceptions loggas på nivån "WARNING" medans system exceptions loggas på nivån "SEVERE".

## 6.5 Integration

Integrationslagret innehåller databasen som systemet använder samt funktionalitet som hör specifikt till databashantering. I detta projekt så används Java DB (derby) som datakälla och Integrationslagret följer *Data Access Object* mönstret för att separera databashantering från business logiken. Integrationslagret är specifikt skapat för med syftet att separera databas-integrering från applikationslogiken (business lagret).

## Data Access Object (DAO) - Design Mönster

Många Enterprise java applikationer är beroende av någon typ av datakälla. Ofta kan denna datakälla variera beroende på miljö där applikationen är implementerad. Problemet som uppstår då är att det lätt kan bli så att business lagret blir starkt beroende på en viss datakälla, detta går emot den generella design strategin som använts i detta projekt, att utveckla komponenter med minskade beroenden emellan varandra. Vi vill uppnå en design där ändringar av datakälla inte behöver propagera in i business logiken. Lösning: Data Access Object mönstret (Oracle, 3. 2001).

Data Access Object mönstret (DAO) är ett mönster specifikt tillägnat åt det problemet. DAO används för att separera låg-nivå data access API från operationer som hör till business logiken. DataAccessObject mönstret bygger på användning av ett DataAccessObject (DAO), DAO ansvarar för alla anrop till databasen. DAO bör inte ha några beroenden på business lagret och bör inte innehålla någon business logik. Man kan säga att DAO blir ett gränssnitt gentemot databasen som är designat för att användas av business lagret. DAO abstraherar bort specifika detaljer vad gäller data-källa och dylikt, det enda business lagret behöver bry sig om är det publika gränssnittet som DAO exponerar.

Användning av DAO ger mer flexibel design som är bättre anpassad för byte av data-källa. Fig 6.11. Get ett tydligt exempel på hur databas-access är separerad från business logiken med hjälp av DAO.

```
/**
 * Validates a registration.
 *
 * Uses the DataAccessObject to check if the given username is available or already taken.
 *
 * @param username username of the person to lookup
 * @return true if the username exists, otherwise false.
 */
public boolean validateRegistration(@Size(min=3, max=16) String username)
{
    return dao.getPersonByUsername(username) == null;
}
```

Fig 6.11. Kodutdrag från business-lagret. Metoden validerar en registrering genom att kontrollera att det givna användarnamnet är ledigt, för att kontrollera detta så utnyttjas DAO.



## 6.6 Code Conventions

Projektets kodkonventioner kommer att vara baserad på Code Conventions for the Java™ Programming Language (Oracle, 1999), för filorganisation, identifiering, kommentarer och variabler, med vissa modifikationer. Det är ett välskrivet och etablerat dokument som är lätt att följa. Många känner igen eller kan dess kodkonventioner sen innan.

### Filorganisation

Alla sektioner i filen skall vara separerade med blanksteg och en frivillig kommentar. Filer som är längre än 2000 rader bör undvikas.

Varje Java source fil skall ha följande struktur:

- Kommentar om filen.
- Paket och import påståenden.
- Klass och interface deklARATIONER

**Kommentar om filen** skall börja med en c-style kommentar som listar klassnamn, versioninformation, datum och copyright. Nedan ser vi ett exempel från personDTO(Fig 6.12).

```
/*  
 * Classname: PersonDTO  
 * Version: 0.1  
 * Date: 15-2-2016  
 * Copyright Alexander Lundh, Kim Hammar, Marcel Mattsson 2016  
 */
```

Fig 6.12 Konvention för filkommentar

**Paket- och importpåståenden.** Deklarera paket före import (Fig 6.13)

```
package grupp14.IV1201.model;  
  
import grupp14.IV1201.DTO.PersonDTO;  
import grupp14.IV1201.entities.Person;
```

Fig 6.13. Konvention för placering av import och paket deklARATIONER.

**Klass och interface deklARATIONER** skall innehålla en kommentar som ger nödvändig information om klassen eller interfacet ovanför deklARATIONER.

Variablerna skall vara organiserade i följande ordning:

- public
- protected
- private

Metoderna skall vara indelade för att öka läsbarheten och förståelse av klassen och intefacet.

## Indentering

Tab skall vara inställd på 4 mellanslag.

Längden för raderna bör ligga under 100 karaktärer för att öka läsbarheten.

När ett uttryck inte får plats på en rad, skall dessa generella regler för radbrytning följas:

- Radbrytning efter komma.
- Radbrytning efter en operator

Klassernas och metodernas innehåll skall enkapsuleras på följande sätt (måsvingarnas position är markerade med gul färg):

```
private void logException(Method targetMethod, Exception e) throws Exception
{
    Object[] args = {targetMethod.getDeclaringClass().getCanonicalName(),
        targetMethod.getName(), e.getClass().getCanonicalName()};
    LOGGER.log(SEVERE, "{0}.{1} threw {2}", args);
    throw (e);
}
```

Fig 6.14. Positionering av måsvingar för metod och klass definitioner.

## Kommentarer

Projektet skall använda sig av dokumentations kommentarer då de kan bli utdragna till HTML-filer genom att använda javadoc verktyget.

Kommentarerna skall användas för att ge ytterligare information som inte är tillgängligt genom att bara läsa koden. Kommentarer skall bara innehålla relevant information för att förstå programmet.

### Kommentarformat

*Single-line comments* skall användas för kommentarer som får plats på en rad.

```
if (condition) {
    /* Handle the condition. */
    ...
}
```

Fig 6.15. Konvention för single-line kommentarer.

*Block comments* skall användas för klasser, interfaces och metoder. Eller när en kommentar tar upp mer än en rad.

```

/*
 * Here is a block comment.
 */

```

Fig 6.16. Konvention för block-kommentarer.

### Kommentarer för metoder

Skall använda sig av block comments med Javadoc struktur (fig 6.17):

- En kort beskrivning av metoden. Avsluta med punkt.
- En mer utförlig beskrivning av metoden, om nödvändigt. Avsluta med punkt.
- Return-tag (om metoden använder return).
- Parameter-tag (om metoden har parametrar).
- Throw-tag (om metoden kastar exceptions).

```

/**
 * Finds the role of a given person.
 *
 * Uses the entity manager to call the .createNamedQuery method to find a person by username.
 * If the user is found it returns its role. If an exception is caught it will return false.
 *
 * @param username
 * @return
 * @throws NoResultException
 * @throws NonUniqueResultException
 */

```

Fig 6.17. Javadoc kommentar.

För get- och set metoder skall inga kommentarer skrivas.

### Deklarationer

En deklARATION per rad är att föredra, istället för att ha flera deklARATIONER på samma rad.

Deklarera variabler i början av varje block (Ett block är kod som är omgiven av "{" och "}") .  
Vänta alltså inte på att deklarerar variabler precis när de skall användas.

```

void myMethod(){
    int a = true;
    int b = 3;

    if(a){
        return b;
    }
}

```

Fig 6.18. Konvention för deklARATIONER inuti metodre.

Det finns ett undantag av denna regel. Den gäller när man skall indexera *for-loopar*.

```

for(int i = 0; i < 10; i++)

```

Fig 6.19 for-loop deklARATION.

## 7. Security view

Applikationen hanterar personlig information och lösenord, vilket medför att säkerhet är en viktig aspekt vid utveckling av systemet.

Säkerhetsaspekter som tas hänsyn till i applikationens nuvarande skapnad är:

### 7.1 Kryptering av nätverkstrafik

Sidor som tar emot och överför känslig data från klient till server och vice-versa har som krav att HTTPS måste användas då HTTPS stöder kryptering av användar-datan. Detta är specificerat i konfigurationsfilen web.xml, se fig 7.1

```
62      <!-- All requests with confidential information should be HTTPS -->
63      <security-constraint>
64          <web-resource-collection>
65              <web-resource-name>confidential</web-resource-name>
66              <url-pattern>/login.xhtml</url-pattern>
67              <url-pattern>/register.xhtml</url-pattern>
68              <http-method>GET</http-method>
69              <http-method>POST</http-method>
70          </web-resource-collection>
71          <user-data-constraint>
72              <transport-guarantee>CONFIDENTIAL</transport-guarantee>
73          </user-data-constraint>
74      </security-constraint>
```

Fig 7.1. Detta utdrag från web.xml definierar en "security constraints" som säger att alla requests av GET/POST till login-page och register-page måste ske över HTTPS eftersom dessa sidor hanterar överföring av känslig data.

Rad 66-67 Definierar sidorna som denna security-constraint gäller, rad 68-69 definierar HTTP-metoderna.

### 7.2 Lagring av användares lösenord

Inga lösenord sparas i klartext. Detta för att öka säkerheten om det skulle ske att någon illvillig person skulle komma över databasen.

Lösenorden sparas istället som ett hashvärde som beräknas av SHA-512 algoritmen (Secure-Hash-Algoritmen), 512 innebär att hashet är av storlek 64 bits, vilket anses tillräckligt säkert.

## 7.3 Autensiering

Autensiering sker via en metod som först producerar det tillhörande hash-värdet från det lösenord användaren angivit och sedan jämför det hash-värdet med värdet som finns i databasen.

## 7.4 Auktorisering

Efter autensiering så används HTTP-sessioner för att "komma ihåg" användaren, dessa sessioner innehåller sessionsID vilket är användarnamnet samt användarens roll (ansökande eller rekryterare).

## 7.5 Loggning av misslyckade inloggningsförsök

Misslyckade inloggningsförsök loggas till applikationsserverns primära logg (information om var loggen hittas ges i sektion 11). Hur denna information om misslyckade inloggningsförsök skall användas är ännu inte bestämt eller implementerat. En möjlig idé för framtiden är att implementera en timeout baserat på IP-adressen av användaren som gjort för många felaktiga inloggningsförsök, alternativt att maila ut informationen till personen vars konto det berör.

## 7.6 Säkerhetsproblem

HTTP, som används för kommunikationen mellan klienten och servern är ett tillståndslöst protokoll, vilket innebär att HTTP-sessions är nödvändiga för att "komma ihåg" användaren, och inte tvinga användaren att logga in på nytt för varje request.

Användandet av HTTP sessions introducerar en del säkerhetsproblem som inte är hanterade i projektets nuvarande skepnad. Ett hot mot sättet att auktorisera användare via HTTP-sessions är att om Sessions-ID't, som identifierar sessionen unikt, kan förutspås, kapas eller gissas av en illvillig användare så innebär det att den illvilliga användaren kan få tillgång till privat information om den användaren och lura systemet eftersom systemet identifierar sessions-ID'et med en specifik användare.

Som nämnts så använder systemet HTTPS för krypterad kommunikation mellan användare och server, detta minimerar risken av att en illvillig användare får tag på sessions-ID via en "avlyssnings-attack". Men det finns fortfarande saker som kan göras för att utöka säkerheten. Eftersom att sessions-ID'n kommer att lagras både på servern och i användarens browser så bör sessions-ID't ha en timeout och kräva ny autensiering efter en viss tid för att öka säkerheten i situationer då en illvillig användare kan få tag på användarens klient-maskin.

På grund av tidsbrist har säkerheten inte kunnat optimerats full ut, möjliga förbättringar att se över vid vidareutveckling av systemet är:

1. Förlänga längden av sessions-ID't för att försvåra för brute-force attacker
2. Använda något slags hash-värde som sessions-ID istället för sessions-ID'n starkt kopplade till information som kan avslöja något om användaren.
3. Inför en väldefinierad timeout på sessioner

## 8. Data section

### 8.1 Databas tabeller

Företagets existerande databas kommer att modifierats. En överblick på deras databas finns i bilaga 14.1.

#### Person

I detta projekt kommer *Role* tabellen representera en sträng som en ny kolumn i *Person* tabellen. Det resulterar i att *role\_id* kommer att strykas i den redan existerande *Role* tabellen. Anledningen till detta beslut, är att det anses överflödigt att ha två olika tabeller då en person bara kan välja mellan två olika roller (applicant eller recruiter).

Person
- id : BIGINT PRIMARY KEY - name : VARCHAR(255) - surname : VARCHAR(255) - ssn : VARCHAR(255) - email : VARCHAR(255) - username : VARCHAR(255), unique - password : VARCHAR(255) - role : VARCHAR(255)

Fig 8.1 Databas tabell för personer.

#### Applications

En ny tabell application skall representera en ansökan.

Application
- id : BIGINT PRIMARY KEY - yearsOfExperience : float - personID : BIGINT PRIMARY KEY - expertiseID : BIGINT PRIMARY KEY - dateFrom : java.sql.Date - dateTo : java.sql.Date

Fig 8.2 Databas tabell för applikationer.

#### Expertise

Samma tabell som i den gamla databasen, med ett nytt namn.



Fig 8.3 Databas tabell för expertis.

Hela databasen och tabellernas relation till varandra

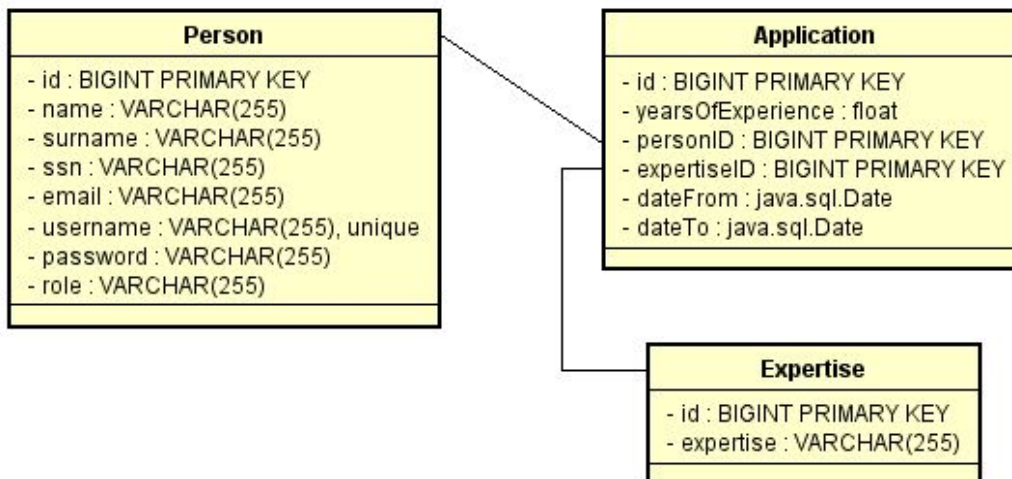


Fig 8.4 Modell över relationsdatabasen.

Tabellen *Application* har två främmande nycklar:

- personID som pekar till tabellen *Person*
- expertiseID som pekar till tabellen *Expertise*.

Anledningen till detta är för att undvika redundans.

## 8.2 O/R Mapping

Projektet använder JPA och därav även entities som hanterar relationerna i databasen.

Tabeller i databasen är skapade av de entities som finns i programmet, dessa entities är i grunden javaklasser där namnet blir tabellnamnet och de variabler som finns i klassen blir kolumner i tabellen.

För att skapa primära nycklar använder man JPA notationen `@Id`(se figur 8.5) följt av den variabel man vill använda som primär nyckel. Om man vill använda sammansatta primära nycklar används notationen flera gånger. Här nedan(figur 8.5) ser vi entiteten *Person* som använder sig av en *BigInteger* som primär nyckel.



```
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private BigInteger id;
```

Figur 8.5

För att skapa främmande nycklar används någon av taggarna: @ManyToOne, @OneToMany, @OneToOne, @ManyToMany beroende på vilken multiplicitet relationen har, exempel på hur främmande-nyckeln som representerar relationen mellan en Applikation och en Person i systemet, implementeras ges i figur 8.6.

```
@ManyToOne
private Person person;
```

Fig 8.6 Främmande nyckel mellan Entiteterna Applikation och Person.

Att använda entites och JPA har valts för projektet då utvecklarna har bekantskap med strukturen samt att det är lättare för javautvecklare att hantera javaklasser och dess struktur snarare än att behöva använda SQL queries. Detta då hanteringen av databasen utförs direkt med javakod, även hanteringen av relationer kan då göras genom javakod då man har tillgång till definierade annotationer så som @ManyToOne som kan placeras direkt på fält i entiteter för att skapa främmandencklar.

## 9. Non-Functional view

Den här delen beskriver de icke-funktionella kraven som har tagits till hänsyn under utveckling av detta system.

### 9.1 Felhantering

För situationer när oväntade fel, som det inte finns någon logisk förklaring till, exempelvis om krypteringen i `java.security*` skulle falla, eller om metoder för att få tillgång till containers kontext kastar ett exception, så används en generisk felmeddelande sida. Att ha en generisk felmeddelande sida som inte är speciellt informativ kan ses som dålig design men i fallet av detta system så är det ett bättre val än att låta användaren fullfölja en defekt aktion. Förhoppningen är att den generiska felmeddelande sidan skall behövas så sällan som möjligt.

Att definiera vilken sida som skall visas vid oväntade felmeddelanden görs i en konfigurationsfilen `web.xml`, se fig 9.1:

```
<!-- Error-page shown when unexpected server-exceptions are thrown -->
<!-- HTTP-code 500 means "Internal Error" -->
<error-page>
  <error-code>500</error-code>
  <location>/servererror.xhtml</location>
</error-page>
```

fig. 9.1 Utdrag från `web.xml`.

Om ett oväntat exception sker så är användaren skickad till en generisk error-sida, se fig 9.2

# Rekryteringssystem

**Internal Server-error** we're sorry about this. Please try again in a minute.

[Back to home page](#)

Company  
name@Copyright  
2016



Fig. 9.2. Screenshot på sidan som visas vid oväntade server-fel.

Vid framtida utveckling av systemet är det möjligt att man kan minska antalet "oväntade fel" genom att göra en genomförlig testning av systemet i en produktionsmiljö och på så sätt identifiera fler "väntade fel".

I fallet när väntade "fel" uppstår, exempelvis misslyckade inloggningsförsök, misslyckade registreringsförsök och liknande så finns fördefinierade felmeddelanden och sidor som används för att kommunicera vad som gått fel till användaren.

## 9.2 Loggning

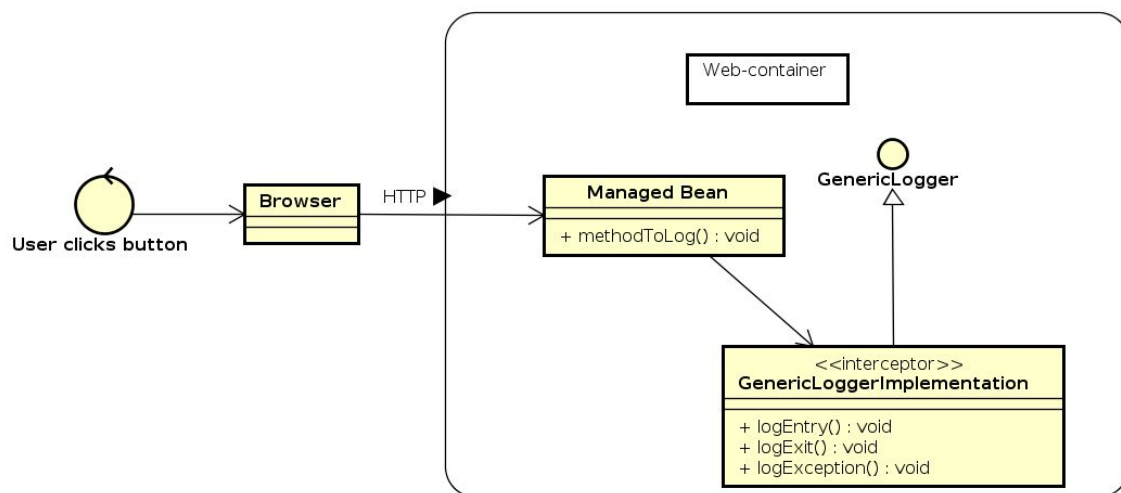
Alla systemets primära event och aktioner loggas. Det finns även stöd för att utöka loggningen vid behov.

Loggningen i systemet är uppdelad i två olika loggningsmetoder:

1. Container hanterad loggning med Java Interceptors.
2. Egenutvecklat loggningssystem för loggning av `applicationExceptions` som uppstår i modellen.

## Container hanterad loggning

Denna loggning används för alla metoder som anropas av CDI-containern, men används inte för anrop emellan olika metoder inuti container, i de fallen så används en specifik logManager som beskrivs i nästa sektion. Loggningen är integrerad med containern där koden körs. Inteceptor-bindings används för att specificera vilka metoder som skall loggas och vilken typ av loggning som skall göras. Inteceptor-klasser används för att specificera vilka metoder/aktioner som skall utföras vid loggningen. Metoderna som är specificerade i inteceptor-klassen anropas av containern varje gång en metod annoterad med en associerad inteceptor-binding anropas, se fig 9.3 för en överblicksbild av hur detta går till.



powered by Astah

Fig 9.3 Container-hanterad loggning med Java Interceptors.

Genom att utnyttja interceptors så separeras loggningen från systemets applikationslogik, se fig 9.4, vilket gör koden mer sammanhängande och lättare att underhålla. Att allt logg-relaterat kan specificeras på ett och samma ställe innebär även att det är en trivial uppgift att lägga till / ta bort loggning från systemet.

```
28 @GenericLogger
29 @Named(value = "loginBean")
30 @RequestScoped
    public class LoginBean
```

fig. 9.4. Implementation av loggning i systemet. Bilden illustrerar separationen av loggning och applikationslogiken. Att notera en klass deklaration med Loggning medför att alla metoder i klassen som anropas av CDI-containern kommer att loggas.

## Loggning av application-exceptions

Dessa exceptions fångas och loggas av en logManager, se fig 9.5.

```
}catch (NoResultException | NonUniqueResultException e) {  
    logManager.log("GET EXPERTISE REQUEST FOR NON-EXISTING NAME", Level.WARNING);  
    return null;  
}
```

Fig 9.5 Loggning av exceptions och events som sker internt i koden och inte kan göras av container hanterad loggning.

## Vad som loggas

Kraven på vad som skall loggas enligt krav-specifikationen från produktägaren är att alla större event skall loggas, loggningen är impementerad på olika nivåer, för att endast logga "större event" skall loggningsnivån vara satt till INFO. Följande loggas på INFO-nivån (nivåer med högre grad än INFO loggas naturligtvis också):

- Registrering av en ny användare
- Inloggning av en användare
- Utloggning av en användare
- Ny applikation
- Administrering av befintliga applikationer

Riktlinjerna för loggning i systemets nuvarande skepnad är att logga mer än bara "större event" men att göra detta på olika nivåer, vilket medför att man enkelt kan filtrera hur pass omfattande loggning som skall användas, exempelvis kan det vara en bra idé ur prestandaperspektiv att inte logga på "FINEST" när systemet är i produktion, däremot kan det vara väldigt händigt under utveckling. I stora drag kan man beskriva systemets loggning så här:

Logg-nivå	Typer av event
FINEST	Information om metodanrop och parametrar som skickas med. Användbart för debugging
INFO	Större event. Misslyckade inloggningsförsök, registreringsförsök och liknande. Användbart för att upptäcka scammer-försök och liknande.
WARNING	Hanterade exceptions.
SEVERE	Oväntade server-fel. Öväntade Exceptions som inte hanterats.

Att klassificera loggning i olika nivåer medför en mer konfigurations-vänlig approach, där man enkelt kan öka/minska vad som loggas vid behov.

Vid beslut av vad som skall loggas så är det alltid en balansgång då loggning kan vara väldigt användbart och skall inte underskattas men samtidigt så vill man inte logga för mycket då det kan medföra latency. Missbruk av loggning kan också vara kontraproduktivt då om man loggar för mycket så blir det ett stort jobb att söka igenom loggfilerna för att hitta den viktiga informationen.

En djupare analys av möjliga attacker av illvilliga användare skulle kunna ge information för att utöka loggningen samt komplettera de logg-nivåerna som används idag.

## Logg-destination

Systemet loggar till standard-loggen av applikationsservern där den körs. Loggen sparas även av applikationsservern. Vid användning av glassfish som applikationsserver kan man även ladda ner serverloggen som en .zip fil genom att göra följande:

1. Öppna upp glassfish admin konsol (<http://localhost:4848/>)
2. Navigera till filen namngiven som "Domain"
3. Tryck på tabben "Domain Logs"
4. Tryck på knappen "Collect Logs"

## Konfigurerings

Loggningen klassificeras efter "nivåer" som talar om hur pass allvarligt ett logg-meddelande är. Generellt så ligger loggning av systemets primär-aktioner på nivån "FINEST" medands loggning av exceptions och andra oväntade fel ligger på "SEVERE".

Vilka nivåer som skall synas i standard-loggen kan konfigureras via admin-gränssnittet för applikationsservern. För att ändra nivån på loggningen av olika typer så måste man ändra i koden, se fig 9.6 det är dock utvecklat så att man bara behöver ändra på ett ställe i koden för att uppdatera nivån på loggningen för alla ställen där den typen av loggning görs.

```
LOGGER.log(INFO, "FAILED LOGIN ATTEMPT");
```



Fig 9.6 Exempel på hur logg-nivån definieras för olika typer av loggning.

## 9.3 Transaktioner

Transaktionshantering i systemet är byggt på container hanterade transaktioner. Genom att flytta transaktionshanteringen till containern uppnås en mer robust transaktionshantering då det inte finns någon risk att vi glömmer att starta/stoppa transaktioner utan istället sköts detta av ramverket EJB. EJB kommer av default starta och stoppa transaktioner när anrop görs till modell-lagret i systemet, se fig 9.7.

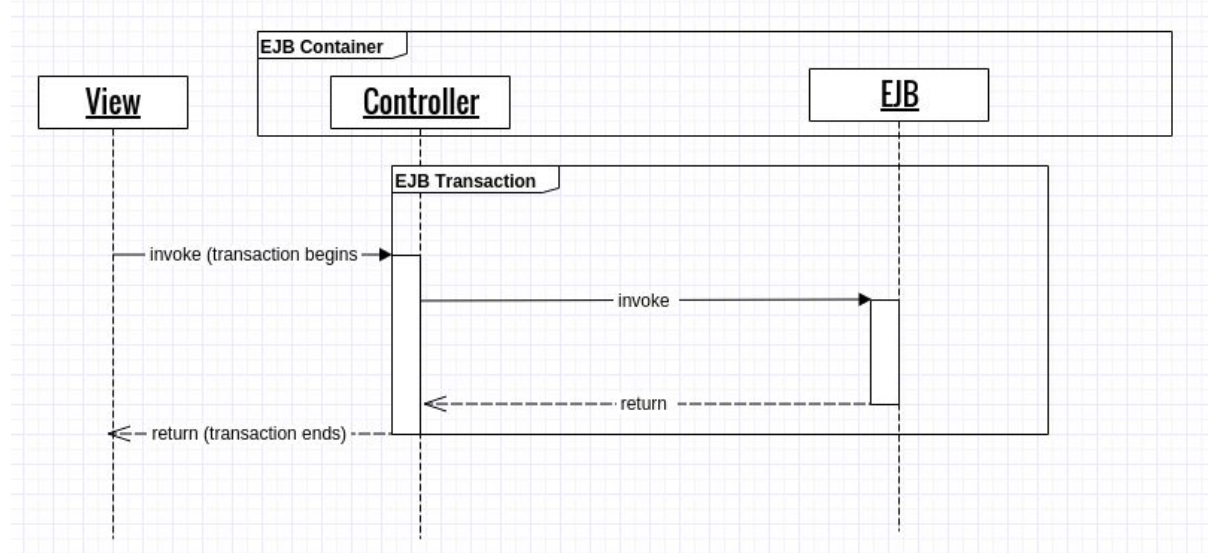


Fig. 9.7. Sekvens diagram över systemets transaktionshantering.

Metoder i EJB har även möjlighet till rollbacks.

Generellt sätt är användning av container-hanterade transaktioner att föredra över bean-hanterade transaktioner p.g.a. att det gör applikations-koden enklare och minskar kopplingen mellan olika delar av systemet. T.e.x. möjliggör EJB att transaktioner kan hanteras deklarativt, och behöver inte startas explicit (Oracle, 1. 2016).

### Rollbacks

Det finns två sätt att verkställa en rollback av en transaktion. Det sätt som utnyttjas i detta system är att Container-hanterade transaktioner automatiskt genomför en rollback på en transaktion om ett SystemException uppstår och inte fångas (Oracle, 2. 2016), se fig 9.8.

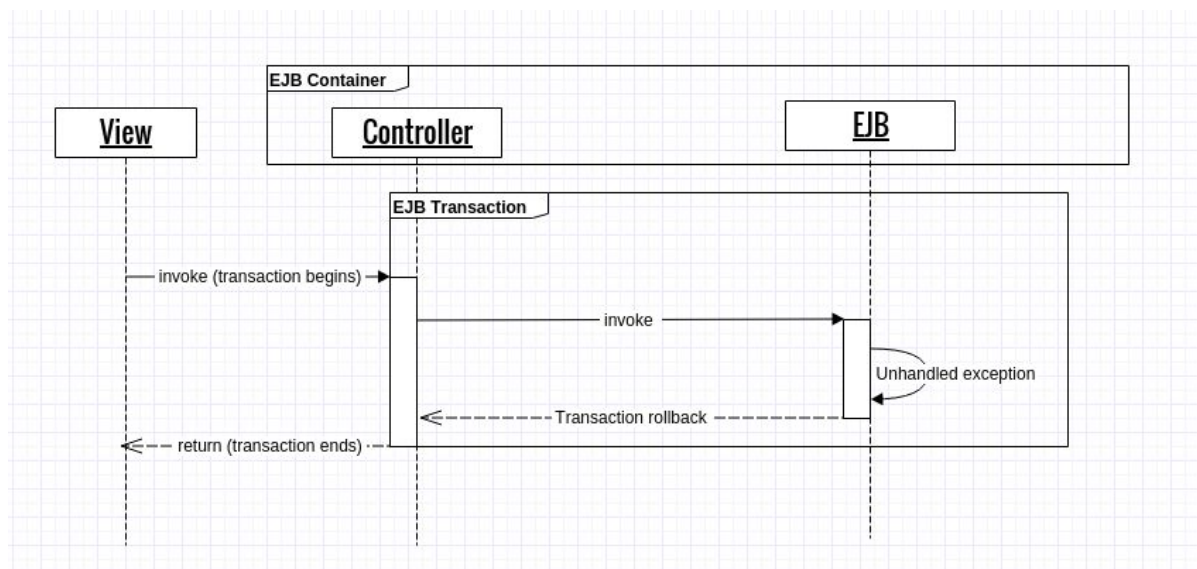


Fig. 9.8. Sekvensdiagram över hur rollback av transaktioner sköts i systemet.

SystemExceptions inkluderar s.k “oväntade exceptions” som uppstår i okontrollerade situationer. Sen EJB version 3.0 så skiljer EJB på ApplicationException och SystemException, ApplicationException medför inte automatisk rollback, det gör däremot SystemException. ApplicationException lämpar sig bäst för “väntade” exceptions, exempelvis valideringsproblem av input-data. Av standard så är ApplicationException exceptions som inte ärver av RuntimeException eller RemoteException. Vill man att även ett ApplicationException skall resultera i en EJB-rollback så kan detta specificeras med annotationen “@ApplicationException(rollback=true)”.

Detta sätt att genomföra rollbacks är lämpligt för det här systemet då i dagsläget finns det inget behov att göra rollbacks på transaktioner om inte sköts av en container och det finns heller inget behov av att inte göra en rollback ifall ett system-exception uppstår under en transaktion.

## 9.4 Prestanda och tillgänglighet

### Tillgänglighet

Systemet är tillgängligt över http-protokollet och webbläsare som stöds aktivt är:

- Google Chrome fr.o.m. version 31
- Mozilla Firefox fr.o.m. version 26
- Internet Explorer fr.o.m. version 10

Resultat från den senaste webbläsartestningen finns i Bilaga A.

### Prestanda

Systemet skall kunna hantera ca 15 000 applikationer under en två veckors period. Detta medför krav på så väl databas-hanteringssystemet samt applikationsserver.



Applikationsservern skall vara skalbar i den mening att prestandan förbättras när man lägger till processorkraft. Det är svårt att estimerar konkreta prestanda krav som kommer krävas utan att ha kört applikationen i produktion men för att ha riktlinjer vid prestanda-tester så kan vi uppskatta att servern bör klara av följande:

- Databas-lagret skall utan problem kunna lagra 15 000 användares data.
- Applikationsservern skall kunna hantera 100st användare åt gången med en svarstid under 0.5s
- Server-errors får uppstå högst 2 gånger på 100 requests.

Ett load-test av applikationsservern har gjorts för att få en känsla av vilka typer av hårdvaruresurser som kommer krävas för att uppnå prestanda kraven.

### Load-test av applikationsservern:

Testet gjordes med JMeter och var konfigurerat så att 100st trådar skickar request simultant, alternerat mellan två olika sidor, home-page och login-page. JMeter är ett open-source verktyg designat för att utföra load-tester och mäta prestanda för framförallt webb-applikationer (Apache Software Foundation. 2016)

Testet genomfördes på en server med 8 cores och 16GB RAM.

Resultat:

Sida	No. request	Medel-svarstid	Error %	No. request/s
Home-page	5204	67 ms	0.27%	31.3
Login-page	5187	634 ms	3.20%	31.2
TOTAL	10391	350 ms	1.73%	62.5

Som syns från resultatet så ligger error % nära den eftersökta nivån och man kan anta att servern kan uppnå en ytterligare minskning av error % vid utökning av processorkraft. Svarstiden ligger i snitt med marginal under den eftersökta nivån (0.35s vs 0.5s).

En intressant observering från resultatet, som inte var väntat, är att login-sidan har betydligt sämre svarstid än home-sidan, se graf i fig 9.9.

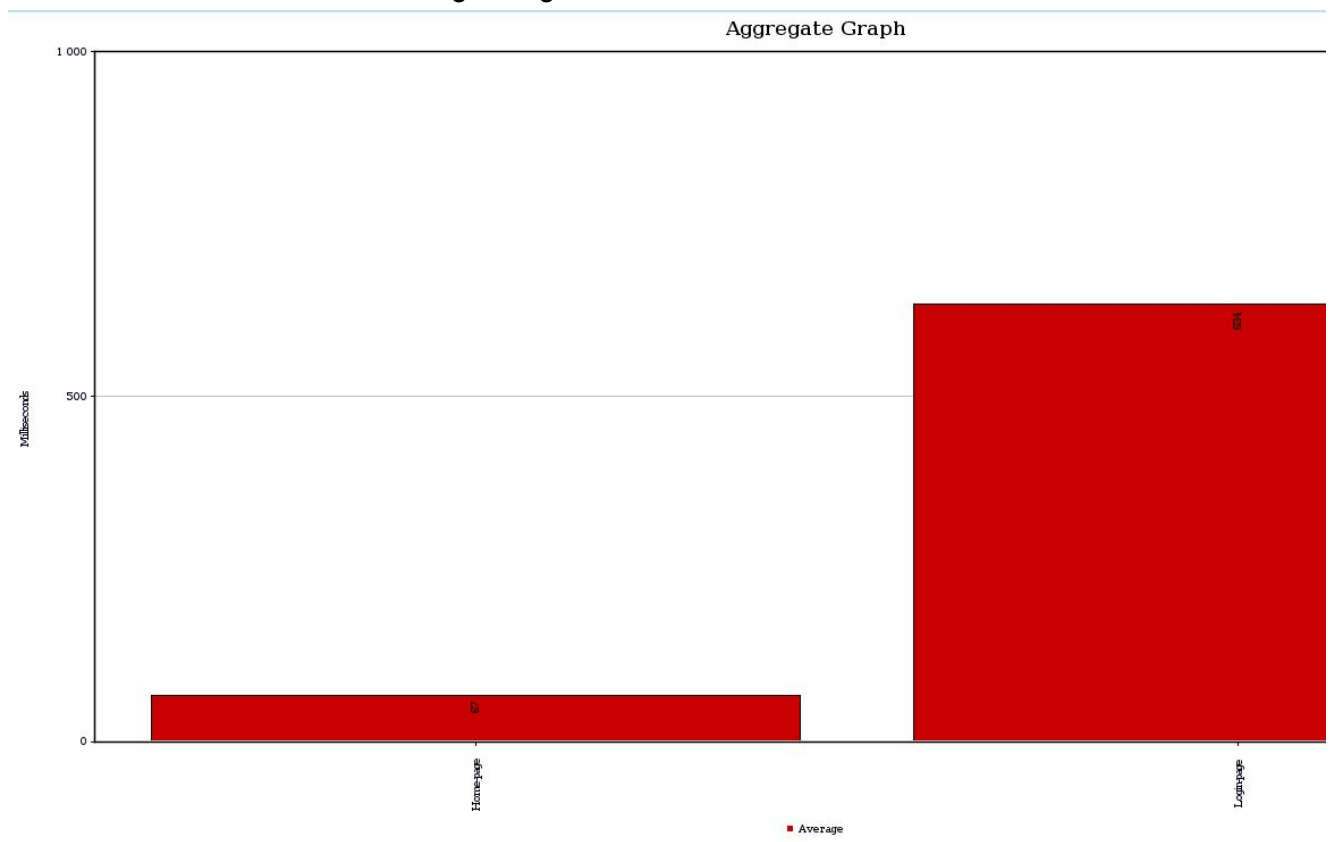


Fig. 9.9. Graf över svarstiden för load-testet.

De fel som räknas in i felprocent i load-testet är alla av typen: java.net.SocketException: Socket closed, vilket beror på en timeout från servern.

Orsaken till den stora skillnaden i svarstid mellan homepage och loginpage är inte fastställt och är något som måste analyseras vidare.

Några orsaker som troligtvis bidrar, men osannolikt är de enda anledningarna:

- Antal bytes som returneras vid ett request till homepage är: 5606. Antal bytes som returneras vid request till loginpage är: 6548
- Vid rendering av login-sidan så kontrolleras om användaren är inloggad eller ej, dvs. HTTP-sessions parametrar kontrolleras, och beroende på resultatet så renderas olika innehåll till användaren. Denna kontroll görs inte vid rendering av homepage.

Detta load-test har gjorts utan optimering av http-cachen, det är något som kan tittas på i framtiden vid behov av ökad prestanda.

## 9.5 Code Coverage

Målet inom utvecklingsgruppen är att 90% av all kod som innehåller någon form av logik skall vara täckt av tester. Dvs. getters/setters och delegeringsmetoder behöver inte vara testade explicit. För tillfället så är målet nått, se fig 9.10.

Målet är uppsatt med motivationen att genom att ha vältestad applikationslogik kod så kan utvecklingsteamet känna en större säkerhet vid varje incheckning av ny kod, buggar som ny kod förmed sig upptäcks förhoppningsvis redan vid första bygget av applikationen.

Anledningen till att målet ligger på 90% och inte 100% (även om 100% är eftersträvarsvärt) är för att vissa delar av en metod kan inte alltid testas på ett lämpligt sätt, då är det bättre att acceptera en nivå på 90% än att desperat försöka uppfylla 100%.

IV1201 > grupp14.IV1201.model

### grupp14.IV1201.model







Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Ctxy	Missed Lines	Missed Methods	Missed Classes
ApplicationEJB		81 %		n/a	1 8	3 22	1 8	0 1
HttpSessionBean		93 %		n/a	1 5	1 6	1 5	0 1
LoginEJB		100 %		n/a	0 4	0 16	0 4	0 1
SHA512		100 %		100 %	0 3	0 9	0 2	0 1
RegisterEJB		100 %		n/a	0 4	0 8	0 4	0 1
Total	18 of 246	93 %	0 of 2	100 %	2 24	4 61	2 23	0 5

Fig 9.10. Utdrag från code-coverage rapporten över hur mycket av applikationslogiken som är testat.

## API-Dokumentation

Javadoc-kommentarer skall finnas för att publika definitioner.

## 9.6 Testning

Den här sektionen beskriver hur tester är implementerade i projektet och hur man lägger till nya tester.

### Strategi vid testning

I detta projekt är strategin för testning att 90% av applikationslogiken skall vara täckta av enhetstester samt att varje förväntat "use-case" och även undantagsfall skall vara testas i acceptanstesterna.

Detta innebär alltså att getters/setters och liknande inte täcks av enhetstesterna och att acceptanstesterna fokuserar på systemets användningsområden snarare än att verifiera att mindre enheter som input-validering eller dylikt fungerar som det ska.

Motivationen för denna strategi är att uppnå en balans mellan testning och utveckling, utvecklingsteamet ser ett stort värde i att ha vältestad kod (mer om detta i sektionen om icke funktionella aspekter) men inser samtidigt vikten av att göra avgränsningar för att inte spendera onödigt med tid på tester. Exempelvis har valet av att fokusera på användningsområden snarare än att testa mindre applikationslogik vid acceptanstester, för att det skulle innebära en redundans att testa samma saker som enhetstesterna redan täcker.

### Enhetstester

Enhetstester är utvecklade med syftet att testa en mindre delar av programmet. En "enhet" i programmet är minsta möjliga del av kodbasen som kan testas. Detta involverar även att designa sitt program så att varje del av programmet kan testas individuellt. I sin artikel "Unit Testing Your Application with JUnit" (Stegeman, J. 2010), beskriver Stegeman god praxis att följa vid design av enhetstester för webbapplikationer. På ämnet "What should I Test?" så nämner Stegeman tre egenskaper som enhetstester bör ha:

- De ska testa en liten "bit" (eller "enhet") av koden. Om testerna testar "för mycket" så minskar dess värde eftersom det då blir tvetydigt vilket del av koden som får testet att misslyckas.
- De ska inte bero på externa resurser så som databaser. Anledningen för detta är att testerna skall kunna köras i en variation av miljöer och så att flera concurrent exekveringar av enhetstesterna inte blandar sig i varandra.
- De skall exekveras snabbt. Anledningen till detta är att uppmuntra att testerna ska exekveras så ofta som möjligt, i princip vid varje kompilering. Om testerna tar för lång tid att exekvera så leder det ofta till att testerna körs mer sällan.

I detta projekt har ramverken JUnit och mockito används för enhetstestning. JUnit är ett generellt ramverk för enhetstester som gör det smidigt att skapa tester för java-klasser. Valet av JUnit är grundat på att det är ett ramverk som är brett använt och som integrerar väl med vårt bygg-verktyg: Maven.

Mockito är ett mocking-ramverk för Java (Mockito, 2016). Anledningen till valet av just Mockito som mocking-ramverk här är p.g.a preferenser samt att det har stöd för integrering med JUnit.

### Varför Mocking?

Som sagt så är syftet med enhetstester att testa mindre delar av programmet, detta kan på vissa ställen vara svårt då man ofrånkomligen har externa beroenden mellan olika lager av applikationen på vissa ställen. Det är här mocking kommer till nytta. Att "mocka" ett objekt innebär att skapa ett typ av "simulerat" objekt som representerar ett verkligt objekt på ett kontrollerat sätt.

Genom att använda mockning så kan enheter i kodbasen testas individuellt trots att de har externa beroenden på exempelvis en databas eller liknande. Med väl definierade mock-objekt så kan man utöver att testa logiken i en metod med externa beroenden, även testa att metoden faktiskt anropar/använder sitt beroende på det tänkta sättet. Se fig 9.11. och fig 9.12. För ett exempel på hur mockning används för enhetstester i detta projekt.

```
41 public boolean validateRegistration(@Size(min=3, max=16) String username)
42 {
43     return dao.getPersonByUsername(username) == null;
44 }
```

fig 9.11. En metod i modellen av programmet som skall enhetstestas tillsammans med sin EJB.

```
57 @Test
58 public void testValidateRegistration() throws Exception
59 {
60     DataAccessObject dao = mock(DataAccessObject.class);
61     Person mockPerson = mock(Person.class);
62     when((dao.getPersonByUsername("test"))).thenReturn((mockPerson));
63     instance.setDao(dao);
64     assertFalse(instance.validateRegistration("test"));
65     when((dao.getPersonByUsername("test"))).thenReturn((null));
66     assertTrue(instance.validateRegistration("test"));
67 }
```

fig 9.12. Test av metoden validateRegistration() i en EJB. Rad 60-61 skapar mockobjekt som skall användas under testet. rad 62 definierar det simulerade beteendet av att anropa metoder på mockobjekten. Rad64 är assertions som avgör om testet lyckats eller misslyckats.

Enhetstesterna i detta projekt är en del av processen av att bygga projektet och kommer att köras vid varje bygge.

### Integrationstester

Integrationstester genomförs för att demonstrera att olika delar av systemet fungerar tillsammans. Integrationstester är heltäckande och mer omfattande att sätta upp. Jämfört med enhetstester så är integrationstester långsamma och grovkorniga (Bien, A. 2011).

Att enhetstester är mer finkorniga är en av orsakerna till att de alltid bör köras före integrationstesterna i projektets "test-cykel".

I stora drag kan man säga att tester som involverar:

- Databas
- Nätverk
- Server
- Micro services

Är integrationstester och inte enhetstester.

Integrationstester slöar ner processen av att bygga ditt projekt, det är därför att föredra att göra integrationstester utanför byggprocessen.

För tillfället är automatisering av integrationstester ännu inte implementerat. För att integrationstesta applikationen så är det bästa sättet att bygga och köra applikationen, alternativt köra acceptanstesterna vilket på ett sätt genomför integrationstester implicit. I framtiden kan det vara värt att kolla på Arquillian som är ett ramverk för att integrationstesta Java EE applikationer.

## Acceptanstester

Acceptanstestning är en typ av testning som görs allra sist i test-cykeln som börjar med enhetstester och sedan integration/system tester och avslutas med acceptanstester. Acceptanstestning innebär att testa systemet för att avgöra om det har uppnått kriterierna från kunden/beställaren. Om alla acceptanstester uppfyller kraven så skall systemet vara redo att levereras till slutanvändarna.

I kontexten av detta projekt, där en webbapplikation är produkten så är acceptanstestning synonymt med användartestning och prestandatestning. I detta projektet så har acceptanstestningen automatiserats med verktyget *Selenium* (Selenium, 2016). Selenium är ett verktyg för att automatisera processen av att testa en webbapplikation via en webbläsare. Med selenium så simuleras en verklig användare genom att programmera vanliga användningsområden av applikationen. Se fig 9.13. För konkreta exempel på hur automatiserade acceptanstester är implementerade i detta projekt.

```
80      /* Test login with username and password. */
81      private void testApplicantLogin(String username, String password)
82      {
83          driver.navigate().to("http://localhost:8080/IV1201/");
84          /* Click "Ansökande" page */
85          driver.findElement(By.id("j_idt19")).click();
86          /* Make sure user is directed to login-page. */
87          Assert.assertEquals("https://localhost:8181/IV1201/login.xhtml",
88                              driver.getCurrentUrl());
```

fig 9.13. Utdrag från ett test-case för acceptans-testning. På rad 83 simuleras att användaren navigerar till index-sidan. På rad 85 simuleras att användaren klickar på "ansökande"-sidan. På Rad 87 så testas att användaren blev skickad till login-sidan.

Motivationen bakom automatisering av acceptans-testning är att spara tid. Med Selenium så kan acceptans-testningen göras på några sekunder, jämfört med om acceptanstestningen görs manuellt vilket skulle ta flera minuter. Detta innebär att acceptanstestningen kan genomföras ofta och exakt.

Valet av Selenium för browser-automatiseringen beror framförallt på att inget jämförbart alternativ har hittats samt att Selenium integrerar väl med vårt byggverktyg Maven och med test-ramverket JUnit.

## Webbläsartestning

Syftet med webbläsartestning är för att se att webbapplikationen uppfyller krav på rendering på olika versioner av webbläsare samt olika upplösningar.

Webbläsartestningen i detta projekt har gjorts med tjänsten *CrossBrowserTesting* (CrossBrowserTesting, 2016). Se bilaga A för resultat från testningen.

## Code coverage

I detta projekt har verktyget JaCoCo (EcEmma, 2016) använts för att mäta ifall projektet uppfyller målen vad gäller testning. Valet av JaCoCo är motiverat att det kan göra precis det vi eftersöker hos ett verktyg av denna typ, samt att det integrerar väl med vårt byggverktyg Maven.

Code Coverage är integrerat i byggprocessen och exekveras vid varje bygge av projektet. Vid varje bygge så skapar JaCoCo en rapport över hur pass kompletta applikationens tester är, se fig 9.14.

IV1201

IV1201







Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 <a href="#">grupp14.IV1201.view</a>	<div><div></div></div>	0 %	<div><div></div></div>	0 %	68	68	149	149	54	54	6	6
 <a href="#">grupp14.IV1201.utli</a>	<div><div></div></div>	5 %	<div><div></div></div>	0 %	16	18	85	93	15	17	3	4
 <a href="#">grupp14.IV1201.entities</a>	<div><div></div></div>	17 %	<div><div></div></div>	0 %	61	63	76	92	43	45	4	6
 <a href="#">grupp14.IV1201.DTO</a>	<div><div></div></div>	0 %		n/a	14	14	28	28	14	14	2	2
 <a href="#">grupp14.IV1201.controller</a>	<div><div></div></div>	0 %		n/a	14	14	17	17	14	14	1	1
 <a href="#">grupp14.IV1201.model</a>	<div><div></div></div>	93 %	<div><div></div></div>	100 %	2	24	4	61	2	23	0	5
Total	1 381 of 1 685	18 %	66 of 68	3 %	175	201	359	440	142	167	16	24

Fig 9.14. Exempel på en code-coverage rapport som genereras vid bygge av projektet.

## Lägga till nya tester

### Lokation

Information om var i projekt-strukturen testerna skall placeras finns i Deployment-sektionen.

## Namnkonvention

I detta projekt så är enhetstester en del av bygg-processen. Maven kommer att köra alla tester som finns i test-mappen och som följer följande namnkonvention:

```
"**/Test*.java"  
"**/*Test.java"  
"**/*TestCase.java"
```

Ifall några tester med andra namn skall inkluderas eller att vissa tester skall exkluderas så specificeras det i pom.xml filen.

Exempelvis i detta projekt så exkluderas acceptance-testerna från bygg-processen då dessa tester endast är användbara när applikationen är deployad till applikationsserver se fig 9.15.

```
<plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-surefire-plugin</artifactId>  
  <version>2.17</version>  
  <configuration>  
    <excludes>  
      <exclude>**/ViewTest.java</exclude>  
    </excludes>  
  </configuration>  
</plugin>
```

fig. 9.15. Utdrag ur pom.xml

Acceptance-testerna måste köras explicit av utvecklaren, dessa kommer hoppas över av maven i bygg-processen.

## Enhetstester struktur

Som nämnts så används JUnit ramverket för att strukturera enhetstesterna. I detta projekt så används en praxis av att varje klass som testas har en enskild test-suite, detta för att få en tydlig avgränsning och undvika osammanhållande tester.

En test-suite är en fil som innehåller både själva testerna och generell kod för att skapa instanser som skall användas under testerna.

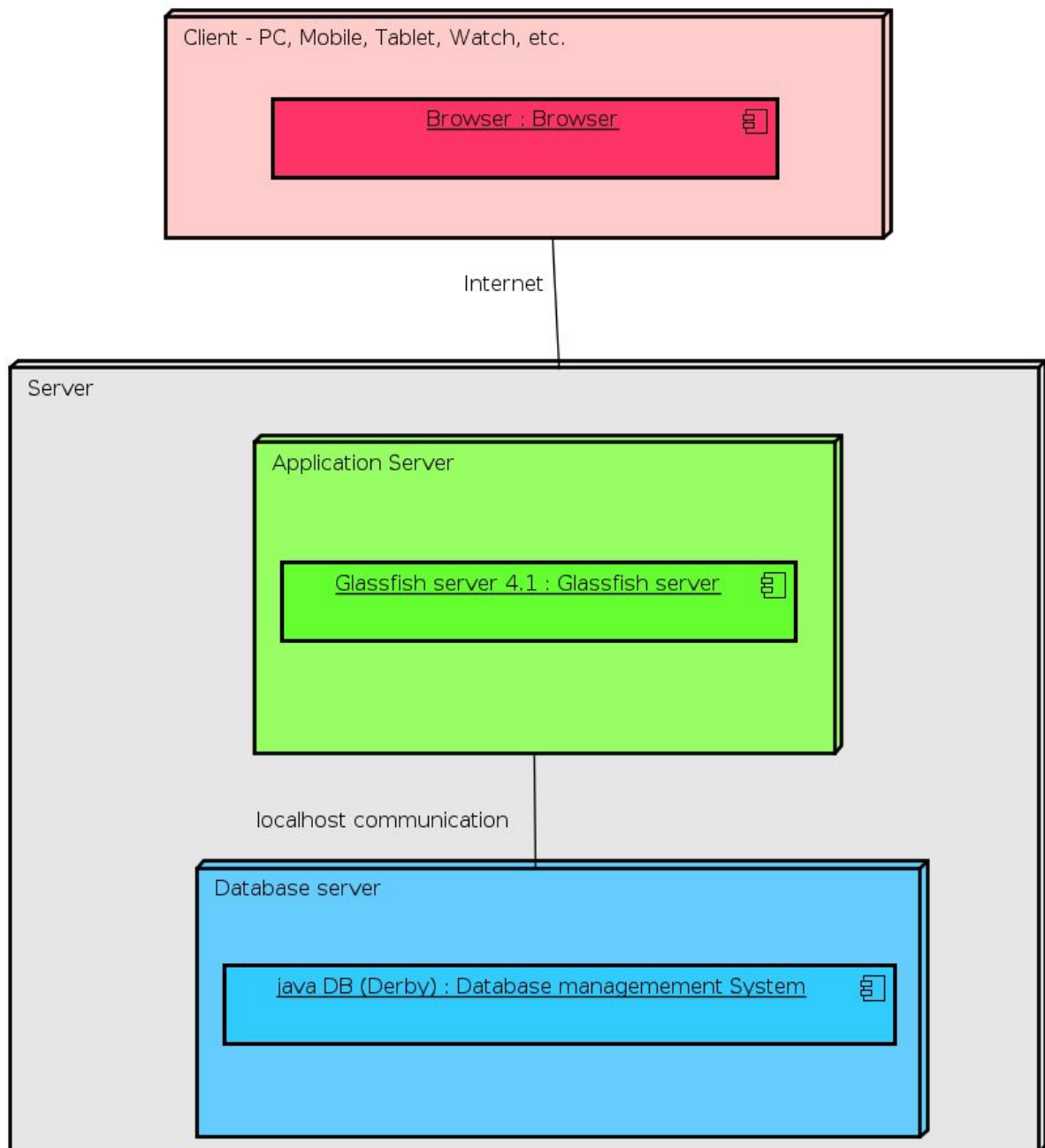
### Lägga till ett enhetstest:

1. Skapa en JUnit test-suite i test-directoriet (finns specificerat i deployment-sektionen) som följer namnkonventionen.
  - a. Ifall Netbeans används som IDE så kan detta göras genom att högerklicka på filen som skall testas och trycka "Tools" -> "Create/Update tests"
2. Annotera alla metoder i suiten som är tester med @Test.



## 10. Deployment view

Fig 10.1 illustrerar de fysiska noderna som är involverade i deployment-processen av applikationen samt relationerna mellan dessa.



powered by Astah

Fig 10.1. Deployment diagram

## 10.1 Fysiska noder

- **Klient:**
  - Webbläsaren som användaren använder för att interagera med applikationen finns lokaliserad på klienten.
- **Server:**
  - Applikationsservern där webbapplikationen körs finns lokaliserad på servern.
  - Databasservern där databas-hanteringssystemet körs finns lokaliserad på servern

## 10.2 Kommunikation mellan noderna

Klienten och Servern kommunicerar med hjälp av Internet-uppkoppling över HTTP/S-protokollet. API:et som kommunikationen följer är definierat på server-sidan och ges till klienten i form av XHTML-sidor samt mappningar mellan URL och resurser.

Applikationsservern och databasen befinner sig båda på servern och kommunicerar därför över localhost, men skulle lika gärna kunna finnas på olika servrar rent fysiskt vilket skulle kräva kommunikation över internet. Kommunikationen mellan databas och applikationsserver sker över Java Database Connectivity (JDBC), vilket är ett API som definierar hur java-program accessar en databas.

# 11. Implementation view

Den här sektionen beskriver konkret produkten som levereras och hur den skall användas.

## 11.1 Produkt

Produkten levereras i en mapp-hierarki som följer standarden för java-projekt som byggs med Maven. I root-mappen hittar du:

- README.md : överblick av vad produkten gör och hur den kan användas.
- pom.xml : konfigurationsfil som maven använder för att bygga produkten
- src :
  - main: mapp för källkoden
  - test: mapp för enhetstester
- target: mapp som innehåller det senaste bygget av applikationen
  - site/jacoco/index.html: code coverage rapport från senaste bygget.
  - target/site/apidocs/index.html: javadoc

## 11.2 Hur man sätter upp och kör applikationen med Netbeans

Netbeans skall användas som IDE. Anledningen till detta är för att Netbeans stödjer väldigt mycket funktionalitet som är lätta att använda sig av. Det är också en populär IDE som många Java EE utvecklare använder idag.

### Sätta upp miljö och bygga projektet

- Installera Netbeans som har support för Java EE (<https://netbeans.org/downloads/>). Denna version tillåter en även att installera servrarna *glassfish* och *apache tomcat*. Detta projekt använder sig av *glassfish*.
- Ladda ner projektet som en zip-fil (<https://github.com/Limmen/IV1201>).
- Öppna Netbeans och importera den nedladdade zip-filen. (File - Import Project - From Zip ... ).
- Bygg projektet genom att högerklicka på projektet i Netbeans och välj *Build with dependencies*.

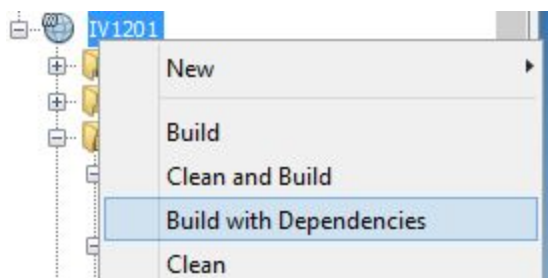


Fig 11.1 Bygga projektet från inuti Netbeans

## Skapa en lokal databas i Netbeans

Det måste skapas en lokal databas för att kunna testa applikationen. Projektet förväntar sig en databas med namnet *IV1201* som man kan nå genom att ange *root* i både *username* och *password*.

Navigera till *Services* och högerklicka på *JavaDB*. Välj sedan *Create Database...*

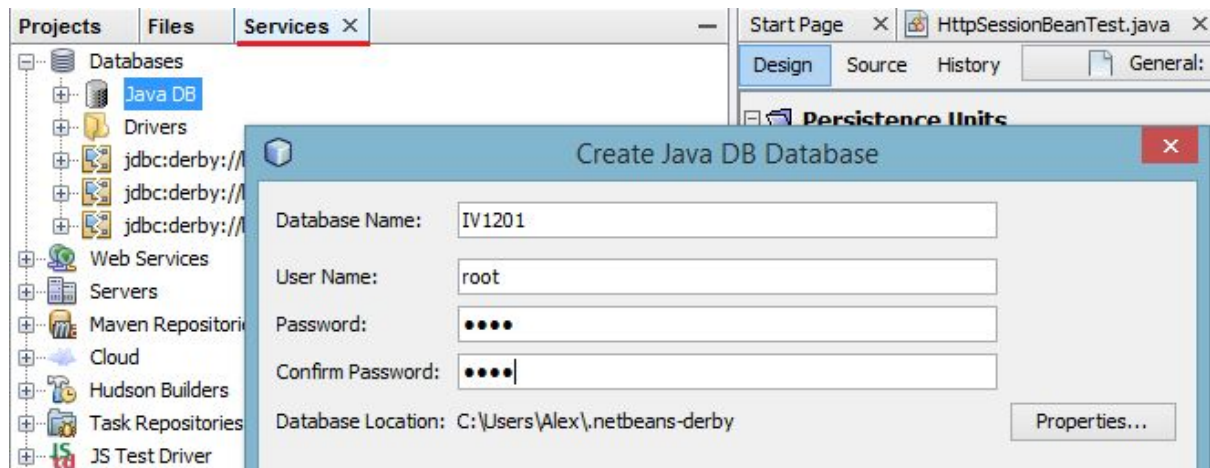


Fig 11.2 Skapa databas med Netbeans.

Om ett annat namn önskas, leta upp *persistance.xml* filen och ändra i *Data Source*:-fältet från *IV1201* till det önskade namnet.

## Kör applikationen

När projektet är byggt och en databas är applikationen redo att köras. Högerklicka på projektet och välj *Run*. Webbläsaren kommer att öppnas automatiskt och applikationen körs.

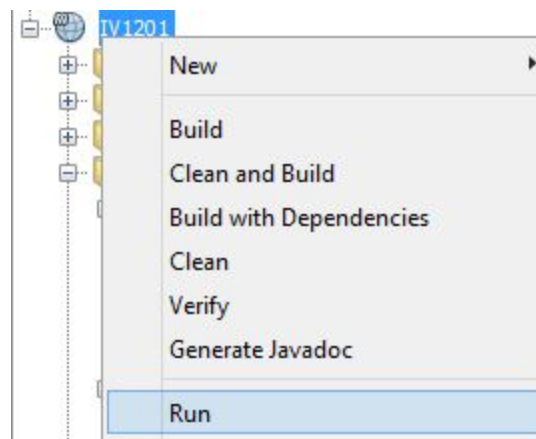


Fig 11.3 Köra applikationen från Netbeans (startar applikationssevern och deployar applikationen).

## Problem som kan uppstå vid skapande av databas.

Vissa upplever problem med att få applikationsservern (glassfish) att connecta med databasen. Ifall dessa problem uppstår så kontrollera följande:

1. Kontrollera att persistence.xml finns (src/main/resources/META-INF/persistence.xml)
2. Kontrollera i persistence.xml att jta-data-source = jdbc/iv1201 (versaler och gemener är viktigt)
3. Öppna upp glassfish konsolen (<http://localhost:4848/>)
4. Navigera: Resources -> JDBC -> JDBC Resources
5. Kontrollera att det finns en resource som heter jdbc/iv1201, Om inte, skapa en resource med exakt det namnet.
6. Kontrollera i din resource vilket "Pool name" som är specificerat.
7. Navigera: Resources -> JDBC -> JDBC Connection Pools
8. Välj det pool-namn som fanns specificerat i din JDBC resource
9. Tryck på tabben "Additional properties"
10. Kontrollera att följande uppgifter finns specificerat:
  - a. URL: jdbc:derby://localhost:1527/IV1201
  - b. User: root
  - c. DataBaseName: IV1201
  - d. Password: ROOT (Versaler)
  - e. Servername: localhost
  - f. Portnumber: 1527
11. Spara och starta om applikationsserver
12. Om du lyckas kontrollera alla steg ovan så ska det nu fungera.

## 11.3 Hur man bygger och kör applikationen från kommandoraden.

Maven är bygg-verktyget som används i detta projekt.

### Bygga projektet

1. Hämta projektet som nämns ovan (sektion 11.2).
2. Skapa databas som nämns ovan (sektion 11.2).
3. Navigera till mapp i ditt filsystem där projektet finns.
4. Ta bort gamla .class filer och dylikt från senaste bygget genom ange kommandot :  
mvn clean
5. Bygg projektet med kommandot: mvn package

## Kör applikationen

1. Efter att du genomfört ett lyckat bygge av projektet så skapas en mapp "target" med de producerade filerna. I target mappen så hittar du "IV1201-1.0-SNAPSHOT.war" .war (Web Application Archive) - filen innehåller en packeterad version av webb-applikationen.
2. Deploya .war filen till applikationsservern. För att deploya till Glassfish server, följ anvisningarna som ges av Oracle:  
<https://docs.oracle.com/cd/E19798-01/821-1757/6nmni99ao/index.html>

## 12. Problem

### 12.1 Automatisering av integrationstester

Automatisering av integrationstester är i projektets nuvarande skepnad inte implementerat men utvecklingsteamet kan se att det skulle vara lönsamt att spendera tid på att implementera detta vid vidareutveckling av systemet.

#### Avvisad lösning

Ett försök av att implementera integrationstester med JUnit har gjorts men resultaten som uppnåddes bedömdes inte tillräckligt effektiva. Anledningen är att Java EE applikationer kommer med ett antal speciella egenskaper vilket gör att generella integrationstester så som JUnit kan erbjuda, i vår uppfattning inte räcker till.

Ytterligare lösningar för att implementera integrationstester har setts över men p.g.a. Brist på tid och tidigare erfarenheter av dessa lösningar så har de inte implementerats.

#### Förslag på lösning

Arquillian (Jboss, 2016) är ett integrations-test ramverk för Java EE applikationer. Vår uppfattning är att Arquillian uppfyller de krav som ställs för att implementera integrationstester för en Java EE applikationer, Arquillian kan dessutom integreras med JUnit.

# 13 Källor

## Design view

1. Lindbäck, L. (2016), Java EE Layers. Lecture notes available at:  
<https://www.kth.se/social/course/IV1201/page/lecture-notes-9/>
2. Eckel, B. (2003), Thinking in Patterns with java
3. Oracle, 1. (2013), <http://docs.oracle.com/javaee/6/tutorial/doc/qircz.html>
4. Oracle, 2. (2013) <http://docs.oracle.com/javaee/6/tutorial/doc/bnaqm.html>
5. Oracle, 3. (2001)  
<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

## Non-functional view

1. Oracle, 1. (2016), Transactions in EJB applications. Tillgängligt på:  
[http://docs.oracle.com/cd/E21764\\_01/web.1111/e13731/trxeb.htm#WLJTA229](http://docs.oracle.com/cd/E21764_01/web.1111/e13731/trxeb.htm#WLJTA229)
2. Oracle, 2. (2016), Container-Managed Transactions. Tillgängligt på:  
<http://docs.oracle.com/javaee/1.4/tutorial/doc/Transaction3.html>
3. Apache Software Foundation (2015). Apache JMeter. Tillgängligt på:  
<http://jmeter.apache.org/>
4. Mockito. (2016) Tillgängligt på: <http://mockito.org/>
5. Stegeman, J. (2010), Unit Testing Your Application with JUnit. Tillgängligt på:  
<http://www.oracle.com/technetwork/articles/adf/part5-083468.html>
6. Selenium. (2016) Tillgängligt på: <http://www.seleniumhq.org/>
7. Bien, A (2011), Integration Testing for Java EE. Tillgängligt på:  
<http://www.oracle.com/technetwork/articles/java/integrationtesting-487452.html>
8. CrossBrowserTesting. (2016). Tillgängligt på: <https://crossbrowsertesting.com>
9. Eclemma. (2016) JaCoCo Java Code Coverage Library, tillgängligt på:  
<http://eclemma.org/jacoco/>

## Problem

1. JBoss. (2016) Tillgängligt på:  
[http://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html\\_single/](http://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/)

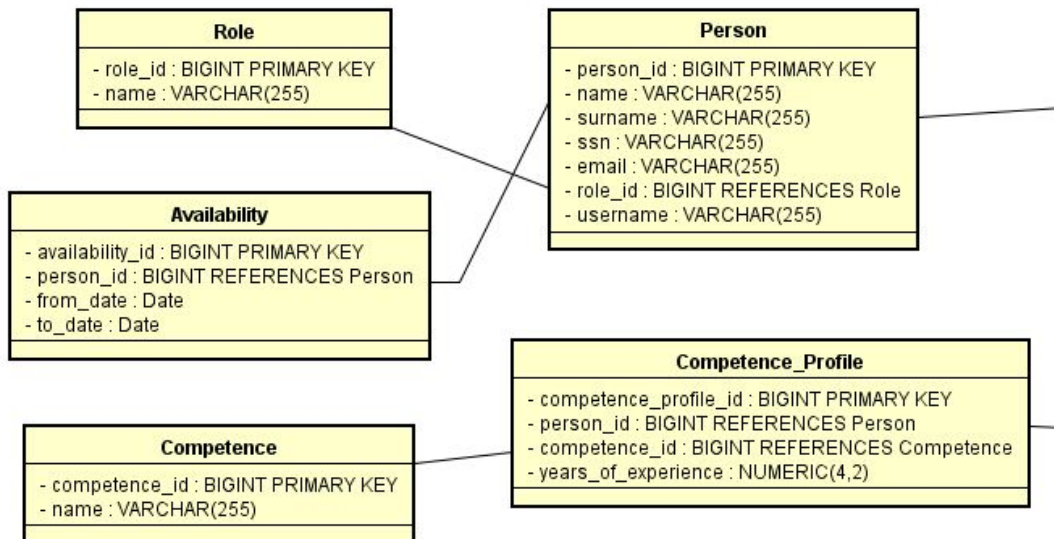
## Code Conventions

1. Oracle. (1999), Code Conventions for the Java <sup>TM</sup> Programming Language.  
Tillgängligt på:  
<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>



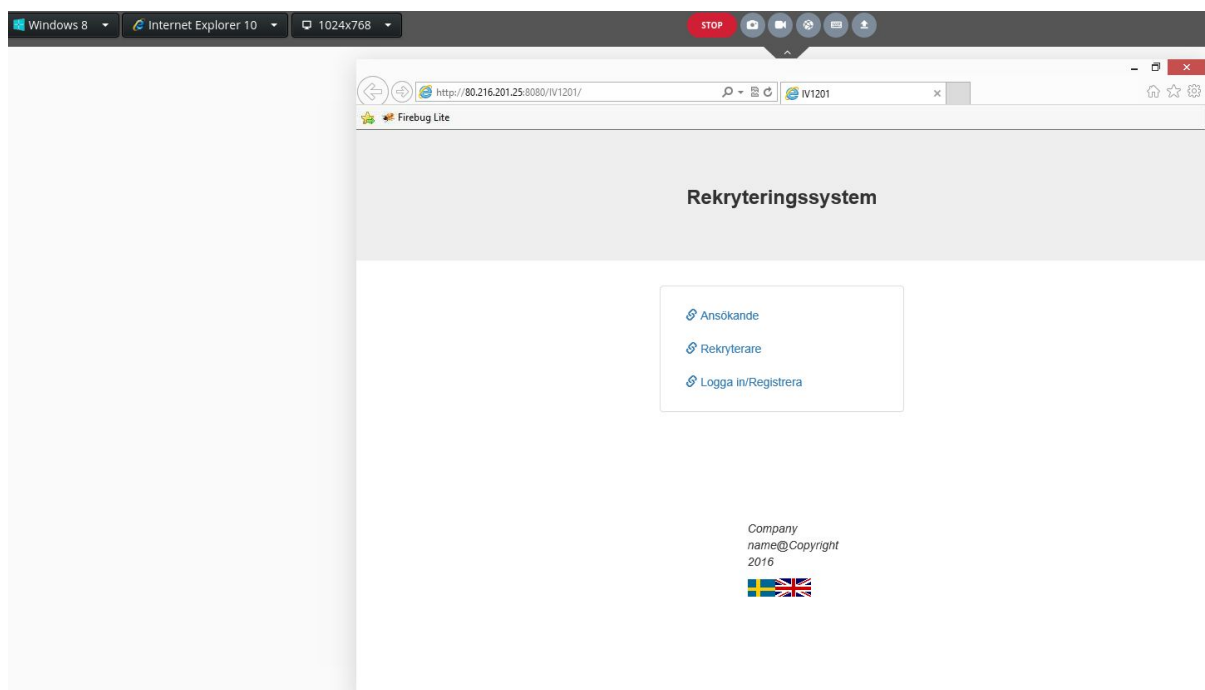
## 14. Bilagor

### 14.1 Företagets existerande databasstruktur (gammal)

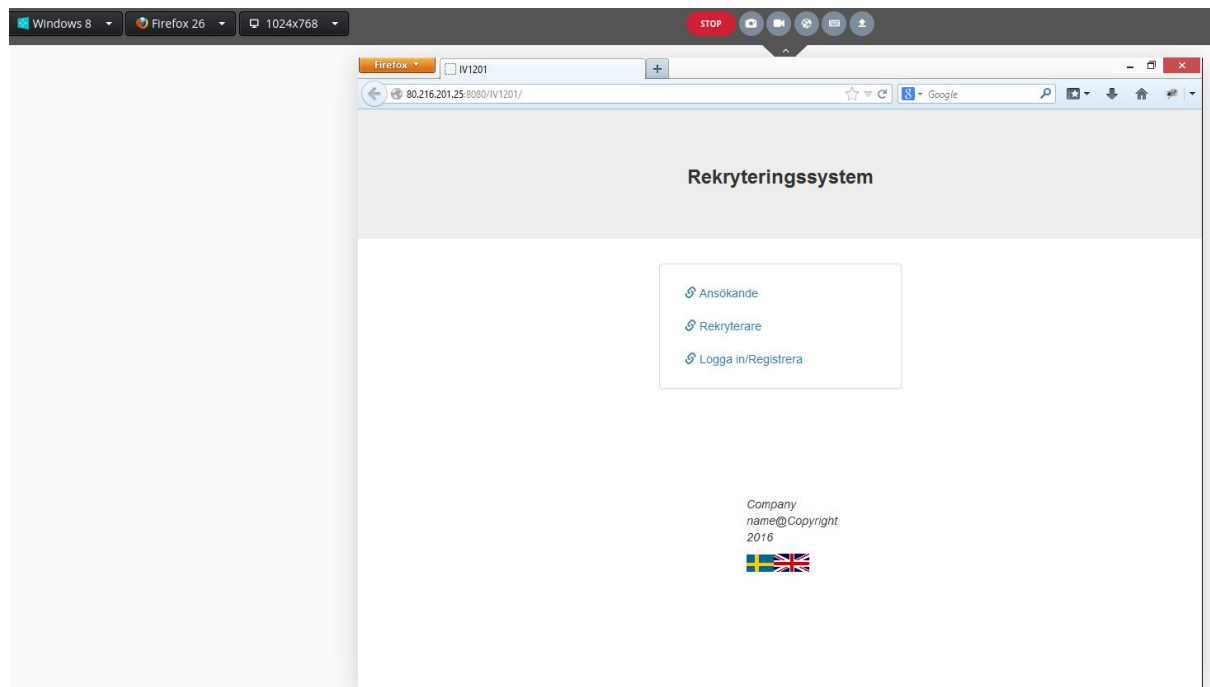


### 14.2 Webbläsar-testning

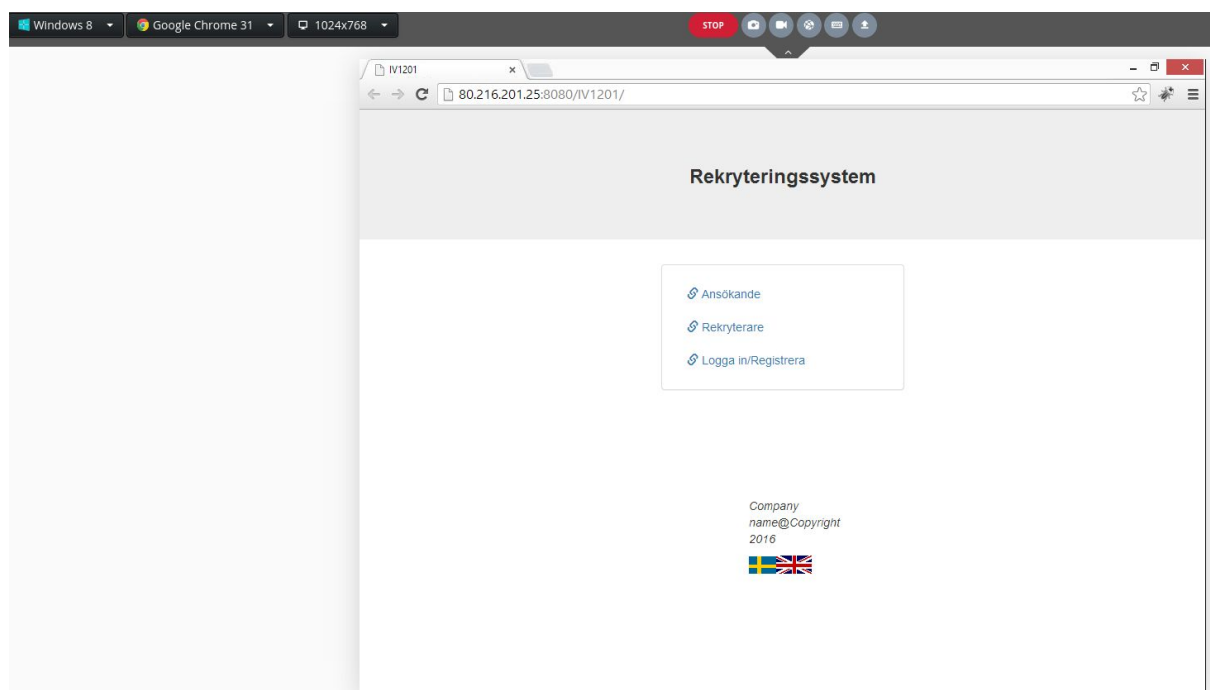
#### Internet Explorer 10



## Firefox 26

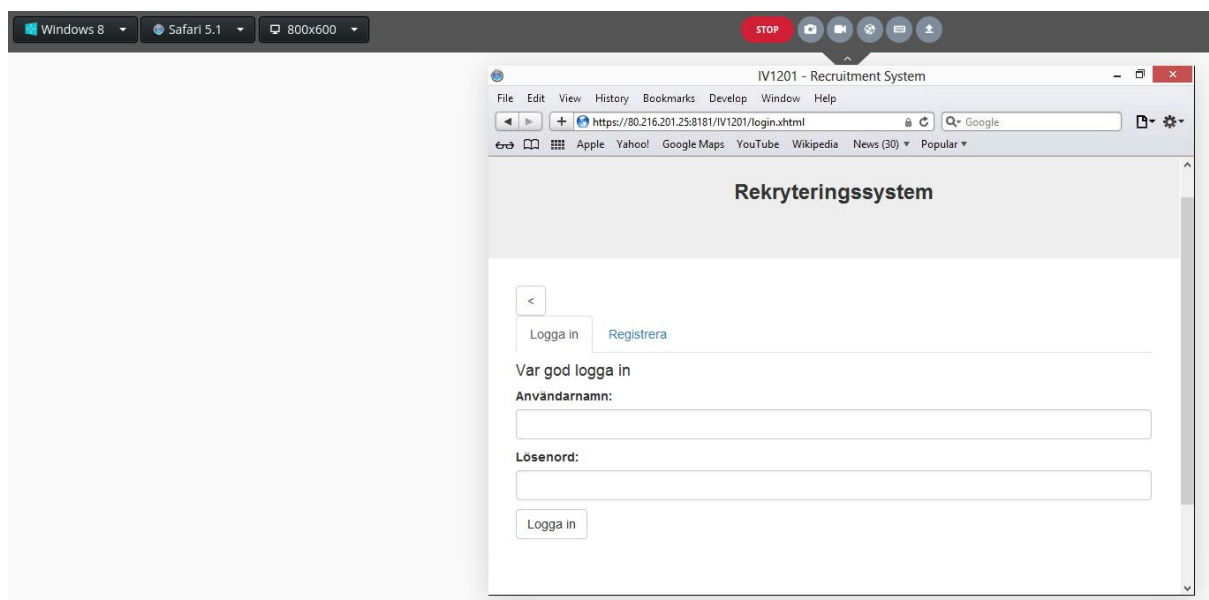
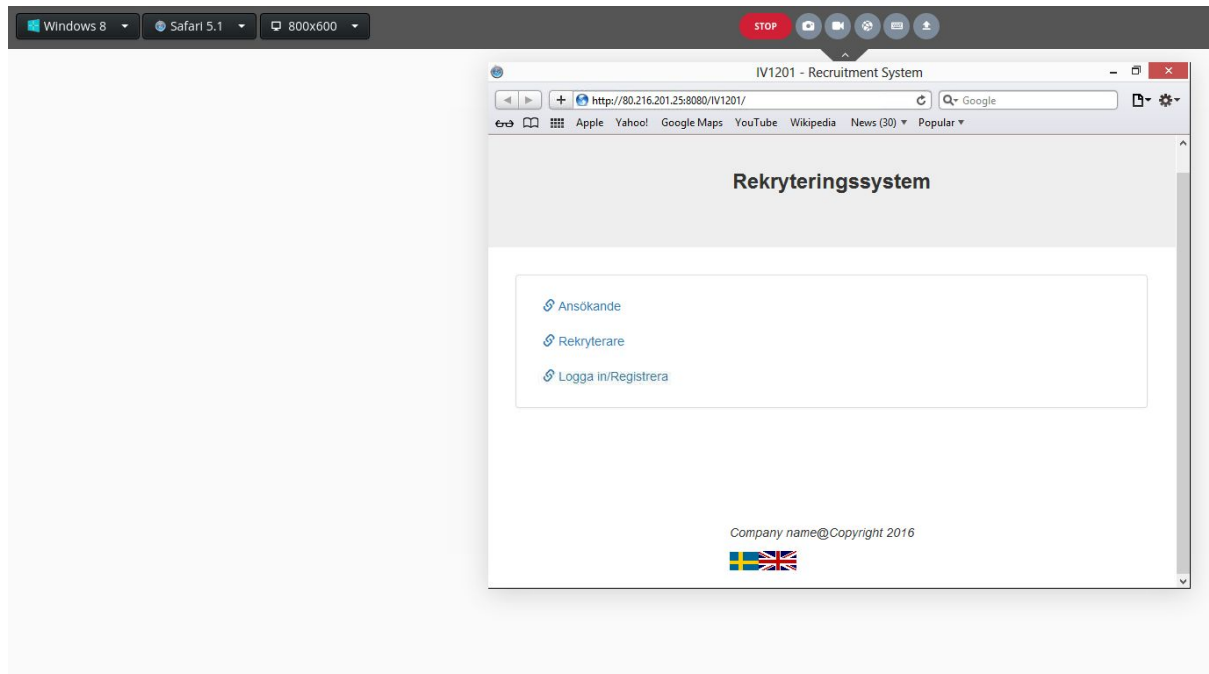


## Chrome 31

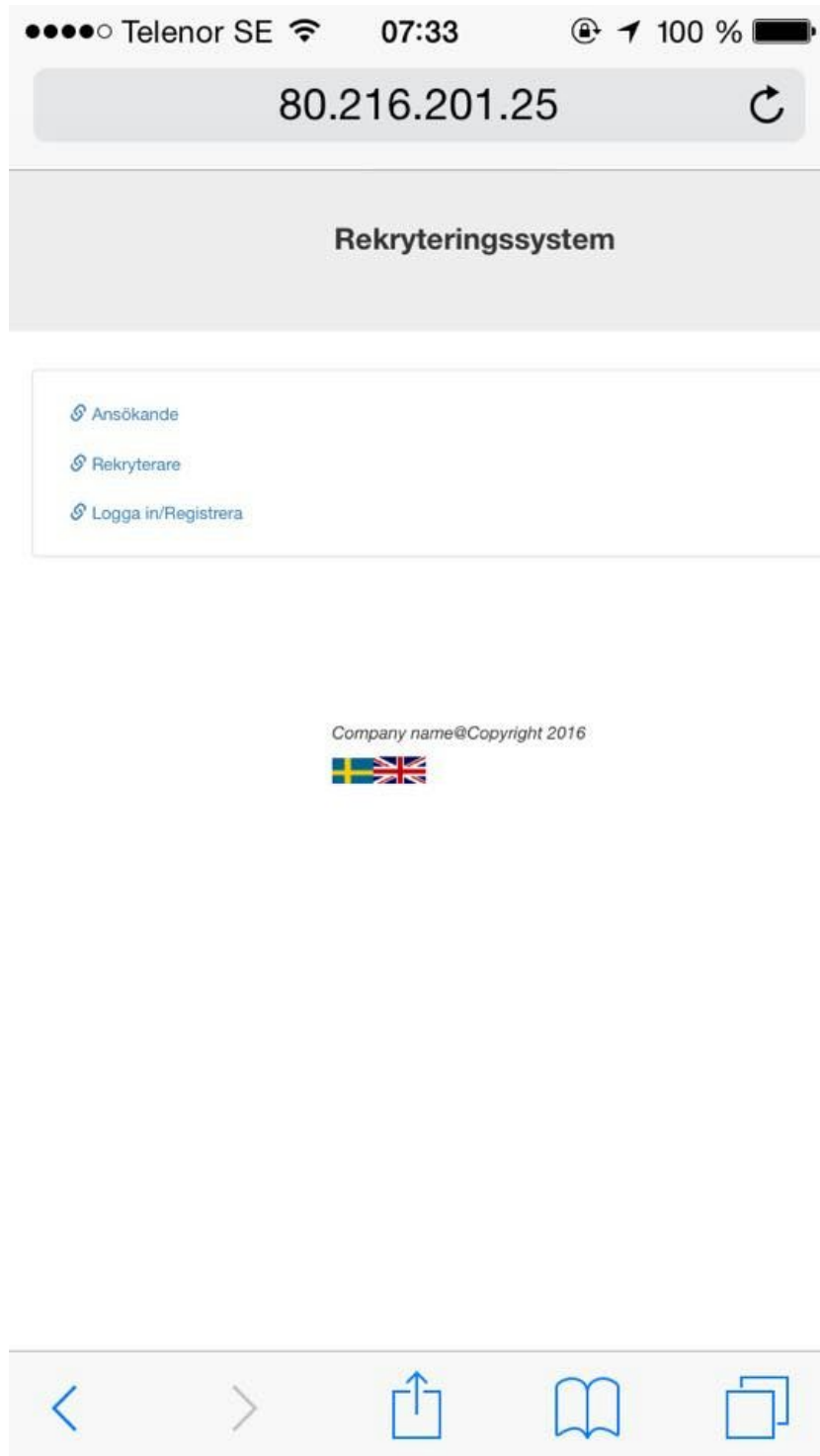


## Olika skärmupplösningar

800x600



1136 x 640 (Iphone 5)



80.216.201.25

## Rekryteringssystem

<

Logga in Registrera

Var god logga in

Användarnamn:

Lösenord:

Logga in

Company name©Copyright 2016



# Var god logga in

Användarnamn:

test

Lösenord:

Logga in

