UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Jakič

RAZVOJ APLIKACIJE V OGRODJU AJAX

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01717/2010
Datum: 15.12.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **NEJC JAKIČ**

Naslov: **RAZVOJ APLIKACIJE V OGRODJU AJAX**
THE DEVELOPMENT OF APPLICATION USING AJAX FRAMEWORK

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Razvijte aplikacijo za podporo postopku prodaje z uporabo ogrodja AJAX. Pri tem uporabite tudi tehnologiji EJB in Java EE.

Mentor:

doc. dr. Rok Rupnik

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Jakič

RAZVOJ APLIKACIJE V OGRODJU AJAX

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2010

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a _____,

z vpisno številko _____,

sem avtor/-ica diplomskega dela z naslovom:

_____

_____

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  _____

  in somentorstvom (naziv, ime in priimek)

  _____

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

ZAHVALA

Hvala Nataši za spodbudo in motivacijo, celotni družini za priganjanje ter podjetju Comtrade za dovoljenje za uporabo aplikacije PresalesTracker za potrebe dimplomske naloge.

KAZALO

SEZNAM KRATIC

- AJAX - Asynchronous JavaScript and XML
- CGI - Common Gateway Interface
- CSS - Cascading Style Sheets
- DOM - Document Object Model
- EJB - Enterprise JavaBeans
- HTML - HyperText Markup Language
- JNDI - Java Naming and Directory Interface
- JPA - Java Persistence API
- ORM - Object-relational mapping
- POJO - Plain Old Java Object
- SQL - Structured Query Language
- URL - Uniform Resource Locator
- XML - Extensible Markup Language
- ZUML - ZK User Interface Markup Language

POVZETEK

Internet je v petih desetletjih obstoja prehodil pot od plahih začetkov znotraj ameriškega ministrstva za obrambo do sodobnih spletnih aplikacij, ki se lahko v marsičem kosajo s svojimi namiznimi tekmicami. Glavni razlog za razcvet spletnih aplikacij v zadnjih letih je uporaba tehnike AJAX, ki z asinhrono komunikacijo med strežnikom in odjemalcem omogoča visoko odzivnost in interaktivnost. Kako AJAX deluje v praksi in kateri so potrebni gradniki sodobne spletne aplikacije je v diplomski nalogi prikazano na primeru aplikacije PresalesTracker.

Razvoj AJAX aplikacij bi bil brez ustreznih ogrodij naporen in zamuden, zato so prav AJAX ogrodja ena pomembnejših komponent pri razvoju sodobnih spletnih aplikacij. Eno takšnih je ogrodje ZK, ki omogoča enostavno gradnjo uporabniškega vmesnika, z označevalnim jezikom ZUML. Ogrodje ZK samo ugotovi v katerem brskalniku teče, katera verzija JavaScripta je v uporabi, na kakšen način prikazati HTML komponente ipd. ter razvijalcu omogoča, da razvija enotno spletno aplikacijo za vse brskalnike, brez dodatnega dela, ki je potrebno, da JavaScript koda uspešno teče v vseh brskalnikih. Vse procesiranje se dogaja na strežniku, zato so strojne zahteve na strani odjemalca minimalne.

Uporabniški vmesnik je sicer najbolj opazen del spletne aplikacije, vendar je za samo delovanje pomembnejša poslovna logika. Prav tako kot pri razvoju uporabniškega vmesnika so tudi na tem nivoju na voljo različna ogrodja, ki močno poenostavijo razvoj. Pri aplikaciji PresalesTracker je uporabljeno ogrodje Java EE, ki s svojimi osnovnimi komponentami (zrni EJB), omogoča enostavno implementacijo poslovne logike. Z ogrodjem JPA je poskrbljeno tudi za dostop do podatkovne baze in preslikavo objektov podatkovne baze v java objekte.

Ključne besede: AJAX, ZK, Java EE 6, EJB 3.0, Spletne aplikacije

ABSTRACT

In five decades of its existence, internet has journeyed the path from the shy beginnings within USA's ministry of defense, to contemporary web applications, which can in many ways hold its ground against their desktop competitors. The main reason for the blossoming of web applications in recent years can be attributed to the use of AJAX, which enables high responsiveness and interactive experience, thanks to the use of asynchronous communication between the server and the client. The thesis uses application PresalesTracker as an example to show how AJAX behaves and which are the necessary parts of a modern web application.

The development of AJAX applications without the help of frameworks would be tedious and time consuming, thus making AJAX frameworks a component of paramount importance for the development of modern web applications. One such framework is ZK, which enables a simple creation of user interface, through the use of a markup language called ZUML. By determining which browser it is running in, which version of JavaScript is in use, how to display HTML components etc., ZK framework enables the developer to develop a unified web application for all browsers, without the extra work, needed to get the JavaScript code running in all browsers. All processing occurs on the server, reducing the hardware requirements of the client to a minimum.

Whereas the user interface is the most visible part of a web application, the business logic is more important for the proper operation. Just like with user interface development, several frameworks are available to simplify the development of business layer. PresalesTracker application uses Java EE framework, which provides means for a simple implementation of business logic, through the use of EJB beans. Furthermore, JPA framework simplifies access to the database and provides mapping of database objects to Java objects.

Key words: AJAX, ZK, Java EE 6, EJB 3.0, Web applications

1 UVOD

Internet je svojo izredno uspešno pot začel v šestdesetih letih prejšnjega stoletja kot ARPANET, interno omrežje agencije ameriškega ministrstva za obrambo. Bližje javni uporabi je prišel konec osemdesetih in v začetku devetdesetih let, s pojavom HTML-ja (leta 1989 je njegovo specifikacijo napisal Tim Berners-Lee[1]) in prvih brskalnikov.

Internet je pridobival na popularnosti, dostop je imelo vedno več ljudi in pojavljalo se je ogromno različnih spletnih strani. Svojo ponudbo so začela na splet seliti tudi podjetja, čeprav so "v začetku poslovne domače strani redko ponujale kaj več kot kontaktne informacije in nekaj dokumentacije." [1] Vendar pa so uporabniki, navajeni namiznih aplikacij, od spleta začeli pričakovati vedno več in se niso zadovoljili s statičnimi spletnimi stranmi. Te so bile namreč zelo neodzivne, saj je vsaka zahteva s strani uporabnika zahtevala ponovno nalaganje spletne strani s strežnika.

Pojavljalo se je vedno več zahtev po asinhroni komunikaciji med brskalnikom in strežnikom, in ko je leta 2005 Jesse James Garrett sestavil besedo AJAX [2], je na spletu teklo že lepo število aplikacij, ki so v ozadju uporabljale tehnologijo, kot jo opisuje AJAX. Kljub temu bi "le malo število ljudi zamešalo spletno aplikacijo z njenim namiznim bratrancem." [1]

V zadnjih letih pa je sledila neslutena rast spletnih aplikacij in računalništva v oblaku, ki nam dandanes ponuja praktično vse, kar smo nekoč lahko uporabljali le na namizju (elektronsko pošto, zemljevide, pregledovanje dokumentov, naročanje letalskih vozovnic ...). Velika večina teh aplikacij ne bi bila možnih brez AJAXa, zato bom kot temo te diplomske naloge predstavil kaj AJAX pravzaprav je, katere tehnologije stojijo za njim, na kakšen način ga lahko uporabljamo in kaj vse ponuja. Pogled bo dejansko širši, saj bom poleg AJAXa na praktičnem primeru predstavil tudi vse ostale potrebne gradnike (z ogrodjem Java EE) za razvoj sodobne spletne aplikacije, ki uporabniku ponuja visoko odzivnost, hkrati pa ima za uporabnika zelo nizke strojne zahteve, saj v celoti teče na strežniku.

2 ZGODOVINA SPLETNIH APLIKACIJ

V dobrih dveh desetletjih je internet iz majhnega omrežja nekaj znanstvenikov, ki so ga uporabljali za medsebojno deljenje dokumentov, zrastel v ogromno omrežje, ki se še naprej nezadržno širi in pridobiva nove in nove uporabnike. Ta širitev brez dvoma ne bi bila mogoča, če se ne bi s pojavom veliko različnih tehnologij uporabniška izkušnja iz leta v leto izboljševala. Za lažje razumevanje kako so internetne strani in aplikacije prišle do stopnje na kateri so danes, je dobro poznati tudi njihovo zgodovino.

2.1 Statične spletne strani

Prvotna vsebina na spletu so bile zgolj statične spletne strani (večinoma kar znanstveni članki), ki so se s HTML-jem lahko ne zgolj sklicevali na ostale članke ampak tudi dodali povezavo nanje. Dokler je bila uporaba spleta omejena predvsem na objavljanje znanstvenih člankov in ostalih statičnih elementov (npr. urnik predavanj), je to dobro delovalo, vendar se je kmalu pojavila želja po bolj interaktivni izkušnji, po bogatejšem vmesniku, podobnem kot so ga bili uporabniki vajeni iz namiznih aplikacij.

2.2 Dinamične spletne strani

Namizne aplikacije so imele z vidika uporabnosti in interaktivnosti veliko prednost pred spletom, vendar tudi eno veliko pomanjkljivost. Potrebna je bila namestitev na vsakem sistemu posebej. Prav odsotnost namestitve je ena največjih prednosti spletnih aplikacij, saj za dostop do njih potrebujemo zgolj brskalnik in ne potrebujemo nobenih namestitvenih datotek ali sprememb na lokalnem sistemu. Drugače povedano: "ali je bolj verjetno, da bo uporabnik preizkusil aplikacijo, za katero je potrebno le malo več kot klik z miško ali tako, pri kateri mora prenesti in zagnati namestitveno datoteko?" [5] Ker uporabniku za uporabo spletne aplikacije ni treba ničesar nameščati na svoj računalnik, je tudi upravljanje z odvisnostmi (npr. zunanjih knjižnic) in nadgrajevanje aplikacije lažje, saj je potrebna zgolj sprememba na strežniku in uporabniku ni treba ničesar spreminjati.

2.2.1 CGI

Prva rešitev, ki je poskusila v spletne strani vpeljati več dinamičnosti, je bila uporaba CGI skript. Z njihovo pomočjo je bilo mogoče ob zahtevi s strani uporabnika na strežniku npr. dostopati do baze podatkov in prikazati dinamično vsebino, glede na zahtevo uporabnika. Kljub temu se tudi z uporabo CGI skript ne moremo izogniti eni največjih pomanjkljivosti klasičnega spleta - ponovnem nalaganju celotne strani s strežnika ob vsaki uporabnikovi zahtevi.

2.3 Spletne aplikacije

V primerjavi z namiznimi aplikacijami so bile dinamične spletne strani še vedno v podrejenem položaju, saj jim ni uspelo zaobiti osnovnega koncepta interneta - sinhronih zahtev in odgovorov, kar pomeni, da se spletna stran neprestano osvežuje. Ne glede na trivialnost zahteve, tudi če je uporabnik napravil le manjšo spremembo, je bilo treba "celotni dokument poslati nazaj na strežnik in ponovno izrisati celotno stran." [1] Veliko razvijalcev se je odločalo za trik s pomočjo skritih IFRAMEov, da so naložili le del strani ali delali 'skrite' klice na strežnik. "Kljub temu, da so mnogi uporabljali to rešitev, v nobenem primeru ni bila idealna - šlo je bolj za programerski trik." [1] Prave spletne aplikacije so se razvile šele, ko so se pojavile prve rešitve, kako med brskalnikom in strežnikom ustvariti asinhrono povezavo, ki ne potrebuje ponovnega nalaganja celotne strani ob vsaki spremembi.

2.3.1 Applet

Leta 1995 se je pojavil nov programski jezik - Java. Takrat najpopularnejši brskalnik, Netscape Navigator, je podprl Javo in s tem se je odprlo novo področje dinamičnega spleta - Appleti. Appleti so majhni programi, napisani v Javi, ki tečejo znotraj Java Virtual Machine (JVM) v brskalniku in do katerih dostopamo preko brskalnika. Ponujajo možnost bogatega uporabniškega vmesnika, vendar imajo tudi svoje pomanjkljivosti. Zaradi varnosti tečejo v t.i. sandboxu in nimajo dostopa do krajevnih diskov, avtohtonih knjižnic ... Potrebno je tudi, da ima uporabnik nameščeno ustrezno verzijo Jave, slabo napisani appleti pa lahko povzročijo tudi težave na uporabnikovem sistemu.

2.3.2 JavaScript

Ob približno istem času je Netscape razvil skriptni jezik, ki se imenuje JavaScript. "JavaScript je bil zasnovan kot pripomoček za lažji razvoj Appletov s strani načrtovalcev in programerjev, ki niso poznali Jave." [1] Kmalu se je pokazalo, da si celotno stran lahko predstavljamo kot objekt in tako je prišlo do rojstva DOMa. Zaradi zahtevnosti pisanja v JavaScriptu in predvsem zaradi pomanjkljivega upravljanja z napakami ter pomanjkanja razhroščevalnikov (*angl. debugger*) se je veliko razvijalcev v zgodnjih letih distanciralo od JavaScripta.

2.3.3 Flash

Poleti 1996 je podjetje FutureWave razvilo FutureSplash Animator. Podjetje je kmalu kupila Macromedia, ki je produkt preimenovala v Flash. Flash razvijalcem omogoča razvoj dinamičnih aplikacij, ki jih težko ločimo od namiznih aplikacij. Eden večjih problemov Flasha je, da potrebuje programsko opremo na klientu. Čeprav se ta namesti kot vtičnik (angl. plugin) za brskalnik in je v nekaterih brskalnikih že vključen ob namestitvi, so se nekateri uporabniki še vedno izogibali namestitvi, predvsem iz strahu pred virusi. Flash aplikacije tudi precej obremenjujejo podatkovne linije in v času pred širokopasovnimi spletnimi povezavami so se jim uporabniki radi izogibali - "tako se je rodila povezava 'preskoči animacijo'." [1] Za razliko od Appletov (in Jave) je licenca za razvoj Flash aplikacij plačljiva in kot taka manj zanimiva za razvijalce.

2.3.4 AJAX

Vsi že omenjeni pristopi imajo vsaj eno pomanjkljivost, ki je preprečila, da bi postali prevladujoča tehnologija za razvoj spletnih aplikacij. Za razliko od njih "AJAX deluje v večini modernih brskalnikov in za svoje delovanje ne potrebuje nobene lastniške programske ali strojne opreme." [5] AJAX je v resnici bolj tehnika kot svoja tehnologija, pri kateri je zgoraj omenjeni JavaScript eden osnovnih gradnikov. Čeprav AJAX kot tak ni nekaj novega, je šele v zadnjih letih doživel razcvet. "Najnovejša tehnologija povezana z AJAXom - XMLHttpRequest (XHR) - je na voljo že vse od časov Internet Explorerja 5 (na trg je prišel pomladi 1999) kot ActiveX kontrola." [1] Kar je noveg, je podpora v brskalnikih. Od Mozille

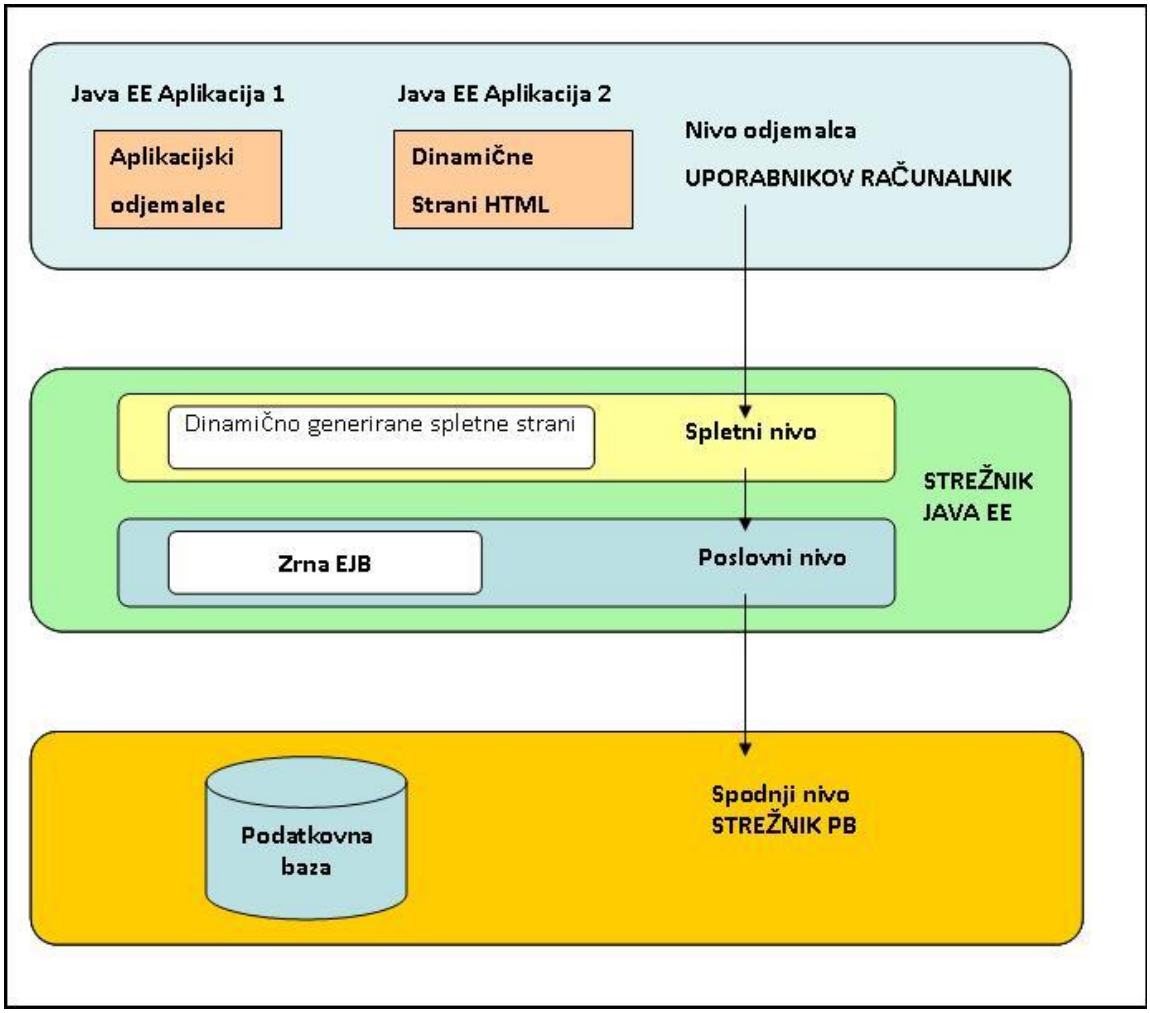
1.0 in Safarija 1.2 naprej, je podpora XHRju tako rekoč univerzalna in je do danes že postala de facto standard.

Čeprav sama tehnologija pravzaprav ni nič novega, pa je pristop, ki ga uvaja AJAX, velik odmik od paradigme zahteva/odgovor. Spletne aplikacije sedaj lahko s strežnikom komunicirajo asinhrono in na ta način opravijo mnoge stvari, ki so bile prej možne zgolj pri namiznih aplikacijah. Kot primer omenimo zgolj napolnjenje padajočega menija glede na neko izbiro na strani. Če uporabnik npr. izbere državo, lahko z asinhronim klicem na strežnik pridemo do seznama mest, ki ga nato prikažemo v padajočem meniju, pri tem pa ni treba ponovno naložiti strani ali imeti seznama mest že vnaprej pripravljenega v skritem polju.

3 PREDSTAVITEV UPORABLJENIH TEHNOLOGIJ

Aplikacija PresalesTracker, ki jo opisujemo v diplomski nalogi, vsebuje vse gradnike, ki so potrebni za sodobno spletno aplikacijo in ima tipično trinivojsko zgradbo.

Slika 1.1: Trinivojska zgradba spletne aplikacije



Vir: [11]

Uporabniški vmesnik je narejen v tehniki AJAX, ob pomoči ogrodja ZK, in omogoča interaktivno izkušnjo, ki še nedolgo nazaj uporabnikom spletnih aplikacij ni bila dosegljiva. Poslovna logika je narejena s platformo Java Enterprise Edition 6 in tehnologijo EJB 3. Za hrambo podatkov je uporabljena podatkovna baza MySql, za preslikavo podatkov iz baze v programske objekte, pa je uporabljena tehnologija Java Persistence API (JPA), ki je del Jave EE.

3.1 Podatkovni nivo

Najnižji nivo v spletni aplikaciji predstavlja podatkovni nivo, ki služi za hranjenje podatkov. Poslovna aplikacija potrebuje podatke, ki so obstojni (*angl. persistent*), za kar se običajno uporablja podatkovna baza. Ker so podatkovne baze dandanes večinoma relacijske, programski jeziki pa objektni, je potrebna tudi preslikava relacijskih objektov v objekte programskega jezika. za kar se uporabljajo posebna orodja za preslikavo. (*angl. Object Relational Mapping - ORM*).

3.1.1 Podatkovna baza MySql

MySql je odprtokodna implementacija relacijske podatkovne baze in je zelo razširjena pri manjših do srednje velikih poslovnih aplikacijah. Vsebuje vse najpomembnejše elemente podatkovne baze, njena največja prednost pred konkurenco pa je cena. Ker gre za odprtokodno rešitev, je seveda na voljo brezplačno.

Zamenjava uporabljene podatkovne baze za neko drugo je tako rekoč trivialna, za kar poskrbi tudi orodje ORM, ki omogoča, da so podrobnosti uporabljene podatkovne baze razvijalcu skrite.

3.1.2 Java Persistence API

Java Persistence API ali krajše JPA je postal del Java EE platforme v verziji Java EE 5. Vsebuje mnogo idej in principov orodja Hibernate, ki je bilo 'de facto' standard za preslikavo relacijskih objektov v Java objekte. JPA, kot večina tehnologij v sklopu Java EE, od verzije 5 naprej, z uporabo anotacij namesto xml deskriptorjev, omogoča mnogo lažji in bolj intuitiven pristop.

Predstavlja vmesno plast med podatkovno bazo in poslovnim nivojem aplikacije in omogoča, da lahko podatkovno bazo kadarkoli, brez večjih težav, zamenjamo, ne da bi aplikacija to opazila.

Ena glavnih prednosti je, da so entitete (in ne več entitetna zrna, kot je bilo pred verzijo 5) navadni Java objekti (*angl. Plain Old Java Object ali POJO*), kar močno poenostavi razvoj in tudi samo razumevanje tehnologije. Logična posledica tega je tudi podpora dedovanju in polimorfizmu.

Za izvajanje operacij nad entitetami skrbi entitetni upravljavec (*angl. Entitiy Manager*), s katerim lahko enostavno izvajamo osnovne operacije nad podatkovnimi objekti, ki so kreiranje, branje, spreminjanje in brisanje (*angl. Create, Read, Update, Delete ali CRUD*).

Z zunanjimi knjižnicami je omogočeno tudi predpomnjenje (*angl. cacheing*) posameznih poizvedb, kar lahko v določenih primerih močno pohitri delovanje aplikacije.

3.2 Poslovna logika

Srednji nivo aplikacije, imenovan tudi poslovni nivo, predstavlja jedro aplikacije, ki določa, kaj aplikacija pravzaprav počne. Medtem, ko je predstavitveni ali podatkovni nivo v aplikaciji dokaj enostavno nadomestiti z drugim, pa je poslovni nivo tako rekoč nenadomestljiv. Pri aplikaciji PresalesTracker je za ta nivo uporabljena platforma Java EE, ena od vodilnih platform za razvoj poslovnih aplikacij.

3.2.1 Java Enterprise Edition

Java Enterprise Edition je platforma posebej namenjena razvoju poslovnih aplikacij. Temelji na programskem jeziku Java in z aplikacijskim strežnikom (pri aplikaciji PresalesTracker je to JBoss) nudi različne sistemske storitve kot so varnost, upravljanje s transakcijami, nadzor sočasnosti, trajnost podatkov ... ter razvijalcu omogoča, da se osredotoči samo na implementacijo funkcionalnosti aplikacije.

Pri aplikaciji PresalesTracker je uporabljena verzija Java EE 6, ki nadgrajuje verzijo Java EE 5, ta pa je prinesla nekaj res revolucionarnih sprememb (podobno kot tudi običajna različice Jave, Java SE 5). Najbolj opazna je uporaba anotacij, ki so zamenjale xml deskriptorje in močno poenostavile razvoj Java EE aplikacij.

Posamezne dele poslovne logike v platformi Java EE predstavimo z java zrni (*angl. Enterprise Java Beans - EJB*). Z verzijo Java EE 5 so bila predstavljena zrna EJB 3.0, ki so, podobno kot ostale komponente v verziji Java EE 5, mnogo enostavnejša za uporabo, od njihovih predhodnikov, zrn EJB 2.1. Pri EJB 3.0 moramo napisati zgolj vmesnik (*angl. interface*) - v njem izpostavimo metode, ki so v posameznem zrnu na voljo - ter samo zrno, ki vmesnik implementira in v katerem povemo, kako posamezna metoda deluje. Ni se nam več treba ukvarjati z lokalnimi in oddaljenimi (*angl. local in remote*) vmesniki, odpadejo tudi vsi xml deskriptorji, saj vse lahko postorimo s pomočjo anotacij.

Ločimo dva tipa zrn EJB, in sicer sejno zrno (*angl. session bean*), ki je lahko s stanjem (*angl. stateful*) ali brez (*angl. stateles*) (v aplikaciji PresalesTracker se uporabljata obe vrsti) ter sporočilno gnano zrno (angl. message-driven bean).

Sejna zrna brez stanja služijo za implementacijo posameznih delov poslovne logike. Aplikacijski strežnik poskrbi, da je v pripravljenosti vedno dovolj zrn, da se vsakemu odjemalcu dodeli lastna kopija, ki mu je na voljo, dokler se ne izvede tisti del logike, ki ga je zahteval.

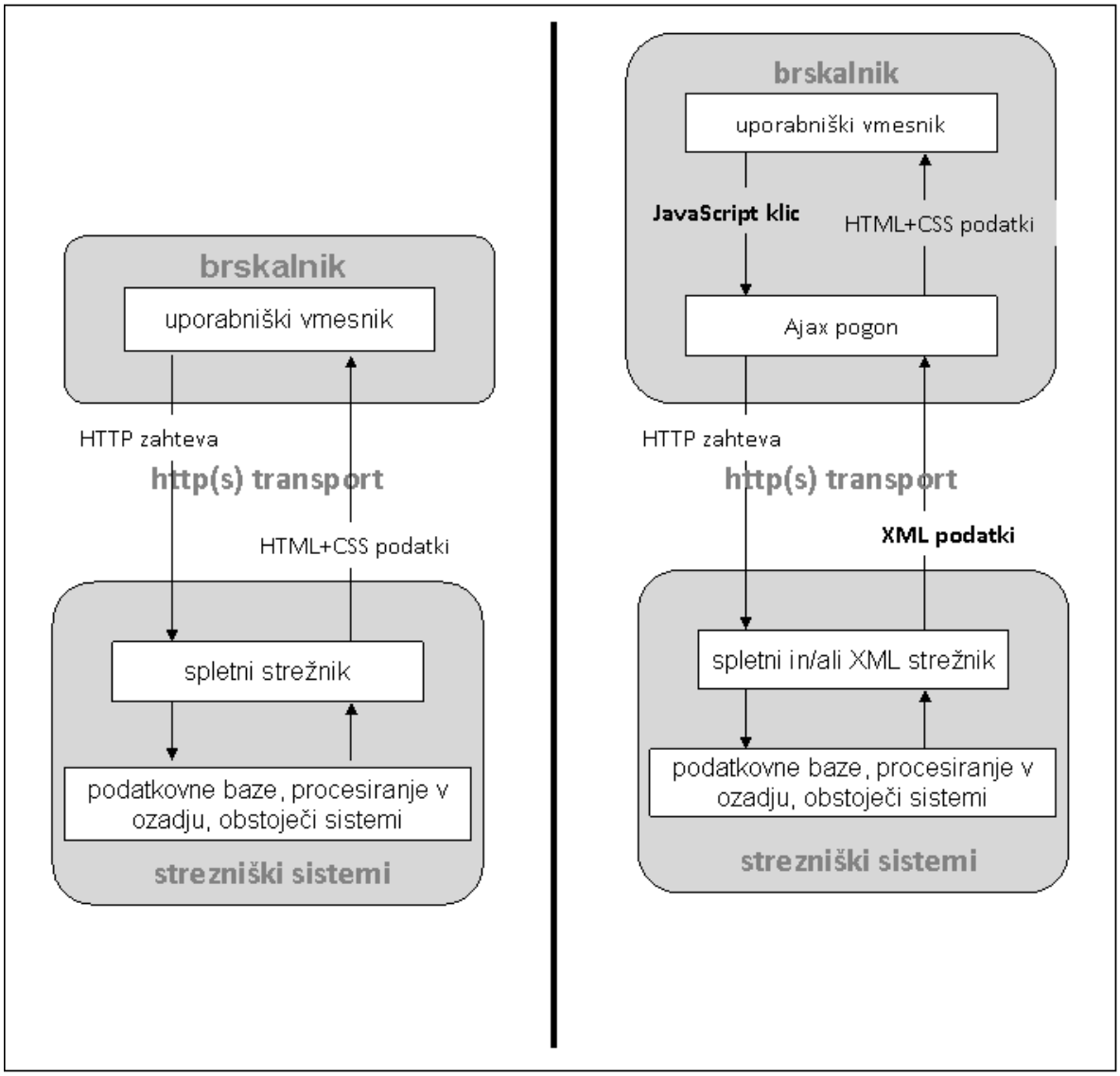
Sejna zrna s stanjem so uporabna za ohranjanje informacij znotraj trenutne seje, saj si zapomnijo stanje in jih lahko uporabimo za npr. hranjenje statusa uporabnika (prijavljen, gost ...), nivoje privilegijev (administrator, uporabnik ...) ...

Sporočilno gnana zrna se, kot pove že ime, odzovejo, ko prispe ustrezno sporočilo in niso krmiljena preko običajne programske logike, ampak ves čas poslušajo, in ko zaznajo pravo sporočilo, izvedejo ustrezno logiko.

3.3 Uporabniški vmesnik

Kot že omenjeno, je uporabniški vmesnik aplikacije PresalesTracker narejen s tehniko AJAX. Ta omogoča dinamično uporabniško izkušnjo, pri kateri je uporabniku prihranjeno čakanje na odgovor s strani strežnika, saj je komunikacija med odjemalcem in strežnikom asinhrona.

Slika 1: Tradicionalni model spletnih aplikacij (levo) v primerjavi z AJAX modelom (desno)



Vir: [6]

Pisanje AJAX kode je lahko zelo zahtevno opravilo, predvsem zaradi obilne uporabe jezika JavaScript na strani klienta. Prav zaradi enostavnosti razvoja in s ciljem bolj stabilnega delovanja, so se razvila številna ogrodja, ki močno olajšajo razvoj AJAX aplikacij. Pri aplikaciji PresalesTracker je bilo za razvoj uporabljeno ogrodje ZK.

3.3.1 ZK

ZK je ogrodje, ki skrije mnogo kompleksnosti AJAX kode, poskrbi za podporo različnim verzijam JavaScripta v različnih brskalnikih, skrbi za konsistentne podatke med klientom in

strežnikom - kljub asinhroni komunikaciji - ter razvijalcem dovoli, da se osredotočijo na samo delovanje spletne aplikacije. Njegova arhitektura je dogodkovno vodena, temelji na različnih programskih komponentah in je strežniško orientirana (*angl. server-centric*), kar pomeni, da se glavnina procesiranja dogaja na strežniku.

Glavni komponenti ZK sta t.i. pogon odjemalca (*angl. Client engine*), ki teče na odjemalcu in pogon za spreminjanje (*angl. Update engine*), ki teče na strežniku. "Delujeta kot par in poskrbita, da se dogodki iz brskalnika prenesejo do aplikacije na strežniku in osvežijo DOM v brskalniku glede na spremembe komponent na strežniku." [12] To pomeni, da celotna aplikacija teče na strežniku in odjemalec, v primeru PresalesTracker je to spletni brskalnik, dejansko služi zgolj za predstavitev podatkov in ne vsebuje nobene poslovne logike.

Glavna prednost takšnega pristopa so zelo nizke strojne zahteve na strani odjemalca, saj so za prikaz aplikacije potrebne res zgolj minimalne zahteve. Vsa poslovna logika je na strani strežnika, kar je dobro tudi iz vidika varnosti, saj poslovnih procesov ne izpostavljamo navzven. Dostopi do deljenih virov so tako lahko neposredni in nam ni treba toliko skrbeti, da bi bili podatki lahko prestreženi. Največja slabost takega pristopa je količina komunikacije med odjemalcem in strežnikom, vendar ogrodje ZK to do neke mere omili, saj je možno posamezne zahteve združevati in tako zmanjšati število klicev proti strežniku. Na voljo je tudi t.i. Client Side Action, ki omogoča izvajanje lastne JavaScript kode na odjemalcu.

4 APLIKACIJA PRESALESTRACKER

PresalesTracker je aplikacija, namenjena internemu sledenju predprodajnih priložnosti (angl. presales opportunities). V podjetju, kjer se aplikacija uporablja, je pridobivanje novih projektov večstopenjski proces. V prvi fazi se identificira t.i. priložnosti, ki se jih uvrsti v lijak (angl. funnel). V tej fazi se naredi grob opis priložnosti, navede za katero podjetje gre, kratek opis, kontaktno osebo, tip priložnosti ... Ta proces opravlja več ljudi (tipično prodajalec v sodelovanju z vodjo razvoja, ki kasneje skrbi za izvedbo).

Pred aplikacijo PresalesTracker se je evidenca novih priložnosti vodila ročno, vsak prodajalec je imel svoj sistem za vnos novih priložnosti, težko je bilo meriti čas, porabljen za posamezno priložnost, kot tudi izvleči različne statistične podatke.

Omogoča vnos novih priložnosti in sledenje skozi celotni življenjski cikel (do novega projekta ali zaprte/neuspele priložnosti). Ker gre za spletno aplikacijo je dostop omogočen tudi prodajalcem na terenu, kar olajša samo komunikacijo. Vsaki priložnosti se lahko določa status (nova, odprta, izgubljena ...), na podlagi statusov se jih lahko tudi pregleduje. Zabeleženo je kateri prodajalec in kateri vodja razvoja sta zadolžena za posamezno priložnost in pomembni mejniki v življenjskem ciklu posamezne priložnosti.

V sami aplikaciji so vgrajene nekatere možnosti poročanja (vgrajena je možnost prikaza grafikonov za različnimi filtri in omejitvami prikaza), za dodatne možnosti pa je poskrbljeno z izvozom v format csv, ki se ga lahko pregleduje in dela obširnejše analize s programi za preglednice (npr. Microsoft Office). Podprt je tudi izvoz v formatu wiki, ki se lahko neposredno uporabi za objavo na interni strani podjetja. Narejena je tudi integracija z aplikacijo, ki skrbi za beleženje dela v podjetju. Ob vsaki novo dodani priložnosti se preko shranjenih procedur (*angl. stored procedure*) v bazi za beleženje dela naredi nov vnos. Na ta način lahko zaposleni beležijo svoje delo na posamezni priložnosti takoj, ko je ta vnesena v sistem.

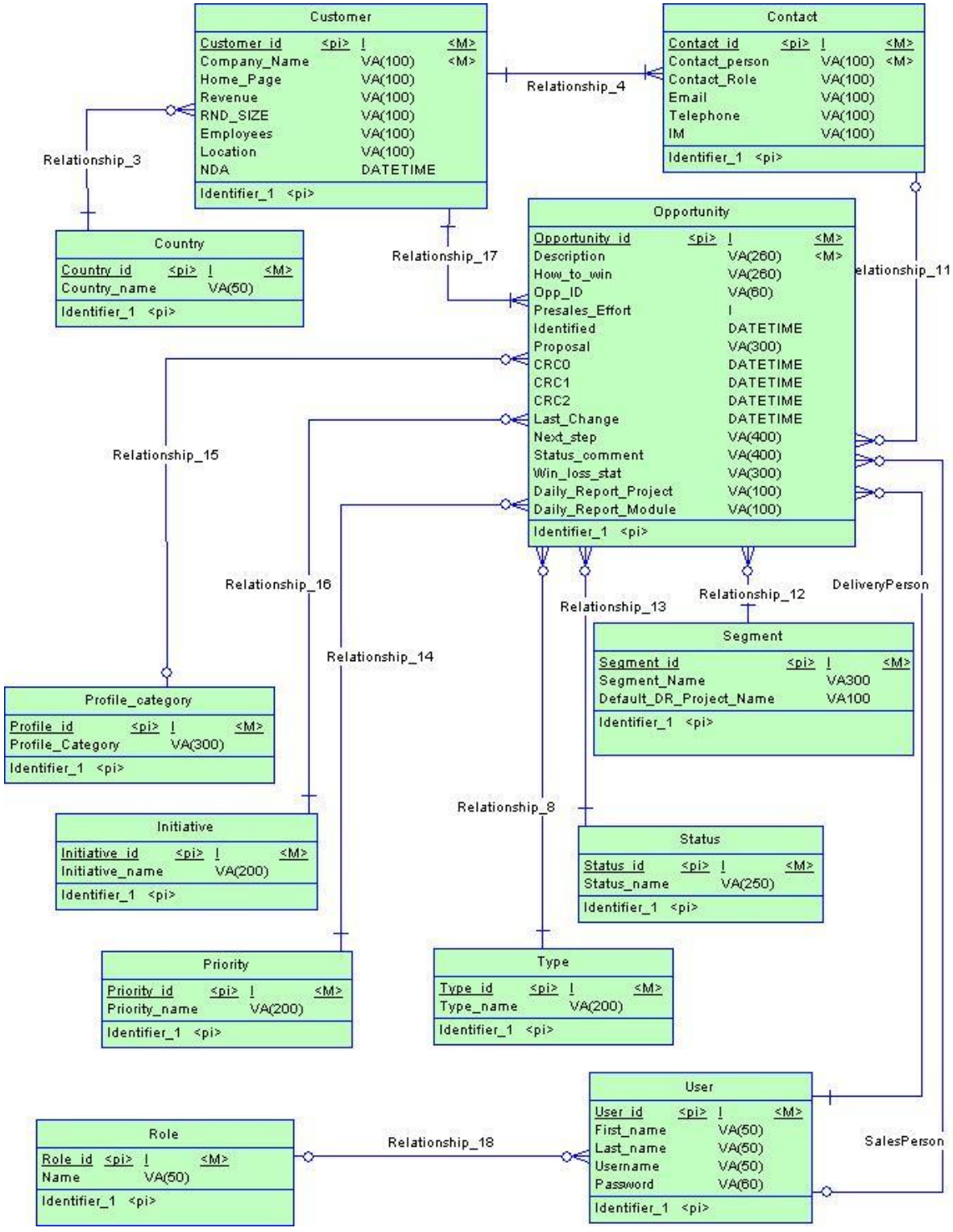
4.1 Podatkovni nivo aplikacije PresalesTracker

Kot je razvidno že iz opisa aplikacije PresalesTracker, je njena glavna entiteta Priložnost (*angl. Opportunity*) okoli katere je zgrajen tudi podatkovni model. V tej entiteti hranimo vse podatke za posamezno priložnost, tako rekoč vse ostale entitete v podatkovnem modelu pa so zgolj njena podpora.

Omeniti velja še entiteti Stranka (*angl. Customer*) in Kontakt (*angl. Contact*) v katerima se hranijo podatki o posameznih podjetjih na katere se nanašajo priložnosti ter tabelo Uporabnik (*angl. User*), ki med drugim služi tudi kot tabela uporabnikov, ki imajo dostop do aplikacije.

Vse ostale entitete v podatkovnem modelu so zgolj šifranti, ki pa so nujno potrebni, da lahko posamezno priložnost predstavimo z vsemi njenimi atributi.

Slika 2: Konceptualni model podatkovnega modela aplikacije PresalesTracker



4.1.1 Podatkovna baza MySql

Kot že omenjeno je MySql vodilna odprtokodna relacijska podatkovna baza. Vgrajene ima vse pomembnejše gradnike podatkovnih baz in je, predvsem zaradi cene (na voljo je breazplačno) najbolj primerna za majhne do srednje velike poslovne aplikacije.

Ker je sam podatkovni model enostaven in ne vsebuje nobenih kompleksnih struktur, nivoja podatkovne baze ne bomo posebej predstavljali. Morda je vredno opozoriti zgolj na to, da je pri kreiranju tabel v podatkovni bazi MySql treba pogon ročno nastaviti na 'InnoDB', saj privzeti pogon 'MyISAM' ne podpira tujih ključev (*angl. foreign keys*).

Slika 3: Primer SQL stavka, ki naredi tabelo SEGMENT

```
/*==============================================================*/
/* Table: SEGMENT                                               */
/*==============================================================*/
create table SEGMENT
(
   SEGMENT_ID                INT            not null AUTO_INCREMENT,
   SEGMENT_NAME              VARCHAR(300),
   DEFAULT_DR_PROJECT_NAME   VARCHAR(100),
   primary key (SEGMENT_ID)
)
type = InnoDB;
```

4.1.2 JPA

Java Persistence API skrbi, da relacijske objekte, ki so shranjeni v podatkovni bazi, lahko uporabljamo kot običajne Java objekte. S tem močno poenostavimo sam razvoj, saj skrijemo večji del kompleksnosti dostopa do podatkovne baze, saj so z vidika poslovne logike vsi podatki na voljo kot Java objekti.

4.1.2.1 Nastavitve JPA

Potrebne nastavitve za uporabo JPA vsebuje deskriptor XML, ki se nahaja v datoteki META-INF/persistence.xml. V deskriptorju določimo ime persistenčne enote (angl. persistence unit), JNDI ime podatkovne zbirke in ostale lastnosti, ki jih potrebujemo.

Slika 4: Deskriptor persistence.xml za aplikacijo PresalesTracker

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="presalesEJB">
    <jta-data-source>java:/PresalesTrackerDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.show_sql" value="false" />
      <!--
        Second level cache Caching of all entities Requires ehCache jar file
      -->
      <property name="hibernate.cache.use_second_level_cache" value="true" />
      <property name="hibernate.cache.provider_class"
        value="net.sf.ehcache.hibernate.EhCacheProvider" />
      <property name="hibernate.transaction.manager_lookup_class"
        value="org.hibernate.transaction.JBossTransactionManagerLookup" />
      <!--
        Named query caching Requires query hints on named queries
      -->
      <property name="hibernate.cache.use_query_cache" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

V primeru aplikacije PresalesTracker so lastnosti za dostop do podatkovne baze določene s podatkovnim virom (*angl. data source*) na strežniku, kjer določimo URL naslov podatkovne baze ter uporabniško ime in geslo za dostop, v deskriptorju PU pa nato podamo zgolj JNDI ime podatkovnega vira. Določimo tudi kateri dialekt naj govori (v našem primeru je to MySql) in po želji tudi dodatne lastnosti.

Aplikacija PresalesTracker uporablja drugonivojsko predpomnenje (*angl. second level cache*) in za pogosto uporabljane poizvedbe tudi predpomnenje poizvedb (*angl. query cache*), zato tu določimo tudi ustrezne parametre.

To so tudi vse nastavitve, ki so potrebne za uporabo JPA, ogrodje bo namreč samo poiskalo vse razrede označene z anotacijo @Entity in jih prepustilo v upravljanje entitetnemu upravljavcu.

4.1.2.2 Entitete

Vsaka relacija oz. tabela v podatkovni bazi je predstavljena s svojim Java razredom, imenovanim entiteta. Kot že omenjeno, so entitete običajni java razredi (POJO), dodatno označeni z ustreznimi anotacijami.

Za izdelavo entitet, ki ustrezajo podatkovnemu modelu aplikacije lahko uporabimo razna orodja, lahko pa entitete enostavno napišemo tudi sami. Vsak entitetni razred mora vsebovati anotaciji @Entity in @Id - prva pove, da gre za entiteto, druga pa določi kateri atribut predstavlja primarni ključ relacije. Implementacijo entitete si bomo podrobneje ogledali na primeru entitete Uporabnik (angl. User).

Slika 5: Konceptualni model relacije Uporabnik (angl. User)

Iz konceptualnega modela je razvidno, da entiteta Uporabnik vsebuje 5 atributov, polega tega pa še povezavo mnogo-proti-ena, ki se v fizičnem modelu preslika v tuji ključ. Kot smo že omenili, so entitete navadni java objekti (POJO), zato lahko pričakujemo, da je entiteta Uporabnik java razred z vsemi atributi, ki so prisotni v konceptualnem modelu.

Slika 6: Entiteta Uporabnik - začetni model

```
package si.hsl.presalestracker.beans.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "user", schema = "presalesDB")
public class User implements Serializable {

	private static final long serialVersionUID = 5358196829721976986L;

	@Id
	@Column(name = "USER_ID")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int userId;

	@Column(name = "LAST_NAME")
	private String lastName;

	@Column(name = "USERNAME")
	private String username;

	@Column(name = "FIRST_NAME")
	private String firstName;

	public User(String lastName, String firstName, String username) {
		this.lastName = lastName;
		this.firstName = firstName;
		this.username = username;
	}
}
```

Kot je razvidno iz slike 5 je temu res tako, edina razlika od java razredov, kot jih poznamo, so anotacije - te so v pomoč, da JPA pravilno poveže objekte med seboj. Kot že rečeno, sta za vsako entiteto obvezni anotaciji @Entity in @Id, vse ostale pa so opcijske. Z anotacijo @Entity označimo, da gre za entiteto, anotacija @Id pa pove kateri atribut predstavlja primarni ključ v relaciji. V entiteti Uporabnik se dodatno pri primarnem ključu uporablja še anotacija @GeneratedValue, ki omogoča, da pri dodajanju novih zapisov JPA sam poskrbi za ustrezno vrednost primarnega ključa.

Poleg tega sta uporabljeni še anotaciji @Table in @Column, ki omogočata, da se imena atributov v java razredu in ustrezna imena stolpcev v podatkovni bazi razlikujejo.

Če relacija Uporabnik ne bi bila vezana na ostale relacije v podatkovnem modelu, bi bilo to vse kar je potrebno storiti, v našem primeru pa moramo poskrbeti še za predstavitev povezave na relacijo Vloga (*angl. Role*).

Slika 7: Entiteta uporabnik - implementacija tujega ključa (*angl. foreign key*)

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "ROLE_ID")
private Role defaultRole;
```

Pri povezavi na tuji ključ moramo podati številnost povezave (v primeru entitete Uporabnik z anotacijo @ManyToOne, saj ima en Uporabnik dodeljeno eno Vlogo) in povedati, kateri stolpec v relaciji Uporabnik predstavlja vrednost primarnega ključa relacije Vloga - v našem primeru je to ROLE_ID. Pri anotaciji za številnost lahko podamo tudi način, kdaj naj se iz podatkovne baze prebere odvisna entiteta - v tem primeru smo izbrali FetchType.EAGER, kar pomeni, da se vedno, ko iz baze preberemo uporabnika, poleg tega prebere tudi ustrezen zapis iz relacije Vloga. Če bi tudi pri vlogi potrebovali seznam uporabnikov z določeno vlogo, bi to lahko dosegli z obratno anotacijo, torej @OneToMany, kjer bi v spremenljivko dobili seznam Uporabnikov.

Zelo uporabna lastnost JPA je ta, da je atribut defaultRole kar tipa Role (to je entiteta relacije Vloga) in ne zgolj vrednost primarnega ključa. To pomeni da lahko do vloge uporabnika pridemo kar preko metode getDefaultRole(). Entiteta Uporabnik je sedaj popolna in jo lahko uporabljamo v aplikaciji.

4.1.2.3 Predpomnjenje

Za pohitritev delovanja in zmanjšanje števila dostopov do podatkovne baze, aplikacija PresalesTracker uporablja tudi predpomnjenje. Kako ga omogočimo smo videli že pri pregledu deskriptorja persistence.xml, na sami entiteti pa z anotacijami določimo še dodatne lastnosti.

Slika 8: Predpomnenje na primeru entitete Uporabnik (*angl. User*)

```
@Entity
@Table(name = "user", schema = "presalesDB")
@NamedQueries( {
  @NamedQuery(name = "getUserByUsername",
          query = "SELECT u FROM User u WHERE u.username = :name",
          hints = {
          @QueryHint(name = "org.hibernate.cacheable",
                     value = "true") }),
  @NamedQuery(name = "getUsersByDefaultRole",
          query = "SELECT u FROM User u WHERE u.defaultRole.name = :roleName",
          hints = {
          @QueryHint(name = "org.hibernate.cacheable",
                     value = "true") }),
  @NamedQuery(name = "getAllUsers",
          query = "SELECT u FROM User u",
          hints = {
          @QueryHint(name = "org.hibernate.cacheable",
                     value = "true") }) })
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class User implements Serializable {
```

Predpomnenje omogočimo z anotacijo @Cache, kjer določimo tudi, kakšen tip predpomnenja si želimo. V primeru entitete Uporabnik je to CacheConcurrencyStrategy.NONSTRICT_READ_WRITE, ki je primeren za entitete, ki se večinoma berejo in kjer je malo verjetno, da bi dve transakciji sočasno poskusili spremeniti entiteto.

Poleg tega entiteta Uporabnik uporablja tudi anotacijo @NamedQueries, s katero lahko določimo poizvedbe, za katere predvidevamo, da se bodo pogosto klicale. Pri poizvedbah nad entiteto Uporabnik je zelo verjetno, da se bodo poizvedbe z istimi argumenti klicale večkrat, brez vmesnih sprememb entitete (novi uporabniki se dodajajo redko), zato se uporablja tudi predpomnenje poizvedb, ki ga omogočimo z namigi (*angl. hints*) pri posamezni poizvedbi.

Imenske poizvedbe (*angl. named queries*) omogočajo tudi zaščito pred napadi vrste vrivanje sql (*angl. sql injection*), saj se vrednost parametrov, ki se uporabijo za posamezno poizvedbo avtomatsko preveri pred izvedbo poizvedbe in se doda ustrezne ubežne znake (*angl. escape characters*).

4.1.2.4 Entitetni upravljavec

Preko entitetnega upravljavca izvajamo vse dostope do podatkovne baze in ga uporabljamo za kreiranje in brisanje entitet ter izvajanje poizvedb nad njimi. V okolju Java EE je za dostop do entitetnega upravljavca zadolžen kar vsebnik (*angl. container*), zato je ravnanje z njim izredno enostavno. Do njegove instance najlažje pridemo z vrivanjem odvisnosti (*angl. dependency injection*).

Slika 9: Pridobivanje instance entitetnega upravljavca

```
@PersistenceContext
private EntityManager em;
```

Ko enkrat pridobimo instanco entitetnega upravljavca, lahko na njem kličemo metode, s katerimi upravljamo entitete. Njegovo uporabo si bomo ogledali na primeru preverjanja uporabniškega imena pri prijavi uporabnika v aplikacijo PresalesTracker.

Slika 10: Uporaba imenske poizvedbe z entitetnim upravljavcem

```
@PersistenceContext
private EntityManager em;

@SuppressWarnings("unchecked")
@Override
public User checkUsername(String username) {
    log.info("checkUsername invoked for " + username);
    List<User> results = em.createNamedQuery("getUserByUsername")
            .setParameter("name", username).getResultList();
    if (results.size() == 1) {
        User user = results.get(0);
        log.info("Returning user " + user);
        return user;
    } else if (results.size() >= 1) {
        log.error("There are two users with username " + username);
        throw new IllegalStateException(
                "There are two users with username:" + username);
    } else {
        log.warn("User doesn't exist");
        return null;
    }
}
```

Na instanci entitetnega upravljavca pokličemo metodo createNamedQuery(), ki poišče imensko poizvedbo z imenom 'getUserByUsername' (kako definiramo imenske poizvedbe smo videli v poglavju 4.1.2.3). Nato nastavimo potrebne parametre imenske poizvedbe in zaženemo samo poizvedbo.

4.2 Poslovna logika aplikacije PresalesTracker

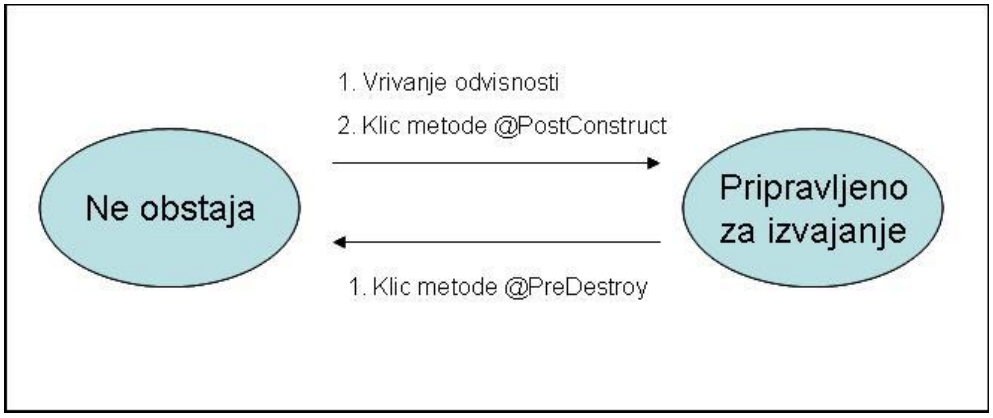
Poslovna logika aplikacije PresalesTracker je realizirana z ogrodjem Java EE. Njeni glavni gradniki so java zrna, ki predstavljajo posamezne enote poslovne logike. Java zrno je, kot že večkrat omenjeno, navaden java objekt ali POJO, dodatno označen z anotacijami. Delimo jih na sejna zrna in sporočilno gnana zrna. Sejna zrna nadalje delimo na sejna zrna brez stanja (*angl. stateless session beans*) in sejna zrna s stanjem (*angl. stateful session beans*).

Glavni namen sejnih zrn v Java EE aplikacijah je, da vsebujejo vso poslovno logiko in se pri uporabniškem vmesniku lahko osredotočimo zgolj na predstavitev podatkov in ne na njihovo procesiranje. Sejna zrna se lahko uporabljajo iz več različnih uporabniških vmesnikov in tako ohranjajo enotno poslovno logiko aplikacije. Za njih skrbi vsebnik (angl. container), ki je zadolžen, da je vsakemu odjemalcu na voljo njegova instanca sejnega zrna, s čimer je poskrbljeno tudi za večnitne dostope. Dostop do posameznih zrn je omogočen z JNDI.

4.2.1 Sejna zrna brez stanja

Sejna zrna brez stanja so najbolj pogosto uporabljena oblika java zrn. Služijo za implementacijo posameznih poslovnih pravil aplikacije, ki jih nato uporablja predstavitvena plast. V posamezno metodo sejnega zrna brez stanja vključimo neko zaključeno celoto poslovne logike, ki je nato na voljo za uporabo. Za to, da je odjemalcem vedno na voljo instanca sejnega zrna, skrbi vsebnik. Ta ob zagonu navadno naredi več instanc posameznega sejnega zrna, ki so nato na voljo odjemalcem, ko jih ti potrebujejo.

Slika 11: Življenjski cikel sejnega zrna brez stanja



Vir: [10]

Ko vsebnik kreira posamezne instance sejnega zrna brez stanja, najprej poskrbi za vsa morebitna vrivanja odvisnosti (*angl. dependency injection*), ki jih zrno vsebuje, nato pa izvede metodo označeno z anotacijo @PostConstruct, če taka metoda obstaja. Metodo označeno s @PostConstruct navadno uporabimo za odpiranje resursov, ki jih zrno potrebuje. Ko se ta metoda izvede je instanca sejnega zrna na voljo za uporabo. Vsebnik običajno kreira več instanc posameznega sejnega zrna, tako da so ta posameznemu odjemalcu takoj na voljo. Instanca sejnega zrna odjemalcu pripada samo toliko časa, da se klicana metoda izvede do konca, nato pa se instanca zrna vrne v stanje 'Pripravljeno za izvajanje'.

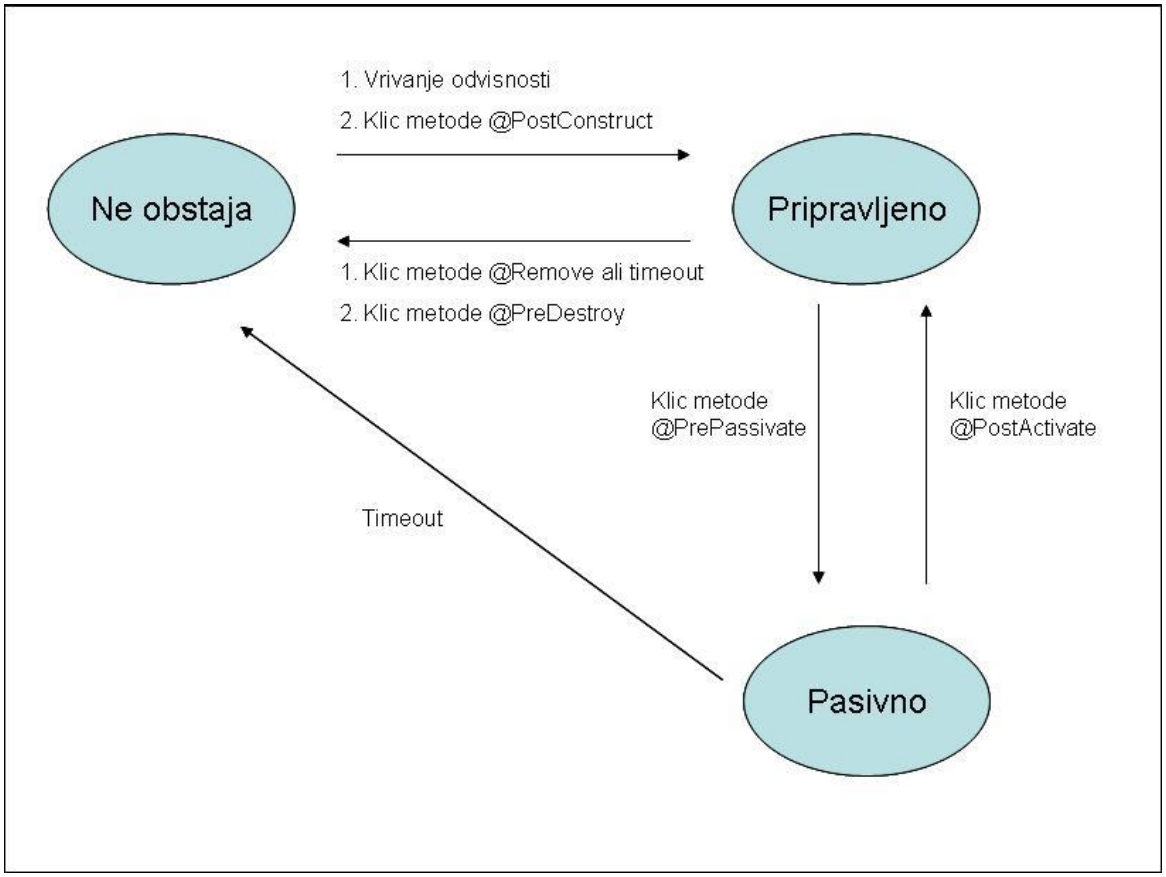
Vsebnik predvideva koliko instanc posameznega sejnega zrna bo v določenem trenutku potreboval in dinamično dodaja nove instance ali odstranjuje obstoječe. Preden vsebnik odstrani posamezno instanco sejnega zrna vedno izvede metodo označeno s @PreDestroy, če taka metoda obstaja. Metodo označeno s @PreDestroy navadno uporabimo za zapiranje odprtih resursov.

4.2.2 Sejna zrna s stanjem

Sejna zrna s stanjem so v primerjavi s sejnimi zrni brez stanja bolj kompleksna, saj si zapomnijo svoje stanje med klici različnih metod in tako z odjemalcem ohranjajo pogovorno stanje (angl. conversational state). Iz vidika odjemalca vsakemu odjemalcu ves čas pripada ista instanca sejnega zrna (strežnik lahko interno odjemalcu sicer nameni različne instance sejnega zrna, vendar poskrbi, da je stanje notranjih spremenljivk konsistentno, zato odjemalec

zrno vidi kot isto instanco), ki si zapomni svoje notranje stanje in služi hranjenju podatkov odjemalca. Gre za neke vrste nadgradnjo sejnega objekta, ki lahko hrani bolj kompleksne podatke. Uporabimo jih kadar potrebujemo razširjeno funkcionalnost sejnega objekta in kadar moramo znotraj seje posameznega odjemalca hraniti določene podatke.

Slika 12: Življenjski cikel sejnega zrna s stanjem



Vir: [10]

Življenjski cikel sejnega zrna s stanjem pozna v primerjavi z življenjskim ciklom sejnega zrna brez stanja eno dodatno stanje, in sicer 'Pasivno'. Kot rečeno, je odjemalcu ves čas trajanja njegove seje na voljo ista instanca sejnega zrna, vendar je odjemalec morda ne potrebuje ves čas. Kadar vsebnik nima dovolj prostora za hranjenje vseh instanc sejnih zrn s stanjem jih lahko pošlje v pasivno stanje (navadno z algoritmom Least Recently Used) in zapiše na disk. Preden vsebnik zrno pošlje v pasivno stanje, mora poklicati metodo označeno z anotacijo @PrePassivate, če ta obstaja. Če odjemalec želi poklicati metodo na zrnu, ki je v pasivnem stanju, mora vsebnik zrno najprej vrniti v stanje 'Pripravljeno'. To naredi tako, da pokliče

metodo označeno z anotacijo @PostActivate, če ta obstaja in šele nato izvede klic metode, ki ga je zahteval odjemalec.

Ker odjemalcu ves čas pripada ista instanca sejnega zrna s stanjem, mora vsebnik hraniti toliko instanc zrn, kolikor odjemalcev uporablja aplikacijo (hranjenje instanc v zalogi, kot v primeru sejnih zrn brez stanja, ni možno). Zato je pomembno, da odjemalec ob prenehanju uporabe aplikacije eksplicitno uniči svojo instanco sejnega zrna s stanjem. To stori s klicem metode označene z anotacijo @Remove, kar vsebniku sporoči, da naj ob izvedbi metode do konca uniči instanco zrna. Vsebnik instanco lahko uniči tudi v primeru, da pride do timeouta.

4.2.3 Sporočilno gnana zrna

Sporočilno gnana zrna so namenjena predvsem za asinhrone naloge, saj sodelujejo z Java sporočilnim sistemom (angl. JMS - Java Message Service). Omogočajo, da odjemalcu ni treba čakati na zaključek operacije, saj se ta izvede asinhrono, odjemalec pa lahko medtem nemoteno dela naprej. Sporočilno gnana zrna se v aplikaciji PresalesTracker ne uporabljajo.

4.3 Uporabniški vmesnik aplikacije PresalesTracker

Uporabniški vmesnik aplikacije PresalesTracker je v celoti narejen z ogrodjem ZK. Teče v vseh modernih spletnih brskalnikih in ima zelo nizke strojne zahteve na strani odjemalca, saj se vsa poslovna logika izvaja na strežniku. Za prikaz se uporabljajo komponente, ki so del ogrodja ZK.

Uporabniški vmesnik lahko ustvarimo na več načinov - lahko se odločimo za programski pristop in komponente ustvarimo v java kodi, lahko uporabimo jezik ZUML, lahko pa oba pristopa seveda tudi kombiniramo. V aplikaciji PresalesTracker je uporabljen pristop z jezikom ZUML, ki temelji na XML-ju. Vsak XML element v jeziku ZUML ogrodju ZK pove katero komponento naj naredi.

4.3.1 Uporaba povezave komponent

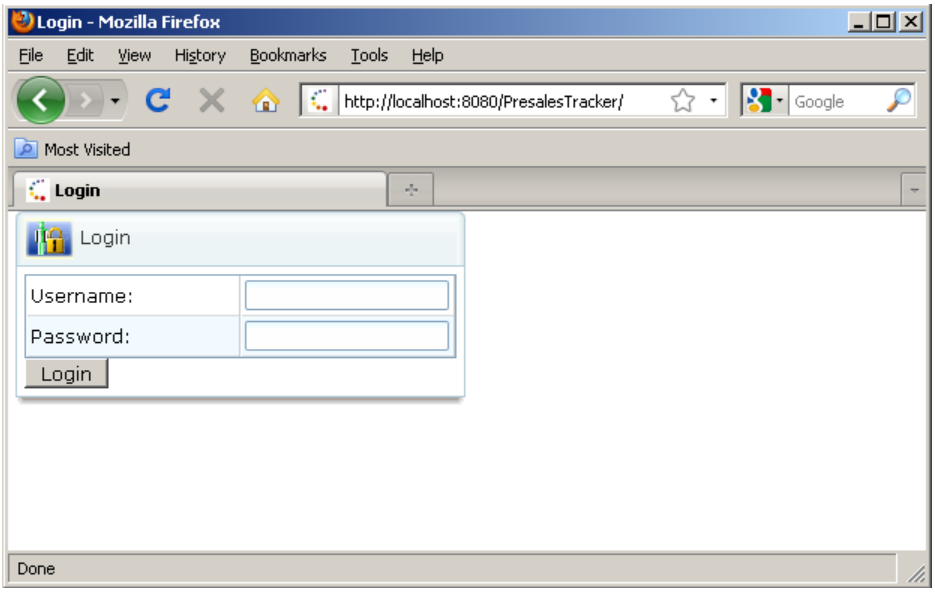
Za dostop do posameznih komponent uporabniškega vmesnika iz pripadajoče java kode, se uporablja povezava komponent. Kot primer si bomo ogledali formo za prijavo uporabnika v aplikacijo PresalesTracker.

Slika 13: Login.zul

```
<?xml version="1.0" encoding="UTF-8"?>
<?page title="Login"?>

<window apply="si.hsl.presalestracker.zk.LoginWindow">
  <style src="/style/style.css" />
  <groupbox mold="3d" width="300px">
    <caption image="/img/login1.gif" label="Login" />
    <grid>
      <rows>
        <row>
          <label value="Username: " />
          <textbox id="username" constraint="no empty: Enter username!" />
        </row>
        <row>
          <label value="Password: " />
          <textbox id="password" constraint="no empty: Enter password!"
            type="password" />
        </row>
      </rows>
    </grid>
    <button label="Login" forward="onOK" />
  </groupbox>
</window>
```

V datoteki Login.zul definiramo katere komponente naj vsebuje forma za prijavo uporabnika. Gre za zelo enostavno formo, ki vsebuje element tipa *groupbox*, dve vnosni polji tipa *textbox* in gumb s katerim potrdimo izbiro. Vsi ostali elementi služijo za razporeditev prikaza. Na obeh vnosnih poljih je definirana tudi omejitev (*angl. constraint*), ki služi, da je v vnosnih poljih, ob pritisku gumba 'Login' vedno vnesena vrednost. Iz datoteke Login.zul ZK nato pripravi datoteko HTML, ki se prikaže odjemalcu in vsebuje tudi vso potrebno JavaScript kodo za povezavo s strežnikom.

Slika 14: Prikaz forme za prijavo uporabnika



V datoteki Login.zul je definiran tudi java razred, ki se uporablja za obdelavo zahtev te forme. Dogodek, ki ga sproži pritisk na gumb 'Login' preusmerimo k metodi onOK v razredu LoginWindow.

Slika 15: LoginWindow.java

```
package si.hsl.presalestracker.zk;

import org.zkoss.zk.ui.Executions;
import org.zkoss.zk.ui.Session;
import org.zkoss.zk.ui.util.GenericForwardComposer;
import org.zkoss.zul.Messagebox;
import org.zkoss.zul.Textbox;
import org.zkoss.zul.Window;

import si.hsl.presalestracker.beans.entity.User;
import si.hsl.presalestracker.beans.session.Login;
import si.hsl.presalestracker.beans.session.UserStatus;
import si.hsl.presalestracker.util.Constants;
import si.hsl.presalestracker.util.ServiceLocator;

import com.cenqua.shaj.Win32Authenticator;

@SuppressWarnings("serial")
public class LoginWindow extends GenericForwardComposer {
	private Textbox username;
	private Textbox password;
	private Window self;

	public void onOK() throws InterruptedException {
		Session session = self.getPage().getDesktop().getSession();
		Login loginBean = ServiceLocator.lookup(Login.class);
		User loggedInUser = loginBean.checkUsername(username.getValue());

		boolean userExistsInDoman = Win32Authenticator.checkWin32Password(
			Constants.HERMES_DOMAIN, username.getValue(), password
				.getValue(), null);

		if ((loggedInUser != null) && userExistsInDoman) {
			UserStatus  userStatus =
				ServiceLocator.lookup(UserStatus.class);
			userStatus
				.login(loggedInUser.getDefaultRole(),
					username.getValue());
			session.setAttribute(Constants.USER_STATUS, userStatus);
			Executions.sendRedirect("/pages/index.zul");
		} else {
			userStatus.logout();
			Messagebox.show("Username and password not correct!");
		}
		session.setAttribute(Constants.USER_STATUS, userStatus);
	}
}
```

Razred LoginWindow razširja razred GenericForwardComposer, ki poskrbi, da se spremenljivke razreda avtomatsko povežejo s komponentami strani. Tako se spremenljivki *username* in *password* v razredu LoginWindow povežeta z vnosnima poljema iz datoteke login.zul in imamo v java razredu vrednost vnesenega uporabniškega imena in gesla na voljo enostavno prek klica funkcije *getValue()*.

4.3.2 Uporaba povezave podatkov

V prejšnjem primeru smo si pogledali kako iz java razreda dostopamo do komponent uporabniškega vmesnika, v naslednjem primeru pa bomo prikazali, kako lahko podatke vidne v uporabniškem vmesniku in v pripadajočem java razredu med seboj povežemo (*angl. data binding*). V ta namen uporabimo razširitev razreda AnnotateDataBinderInit.

Slika 16: AddCustomerInitiator.java

```
package si.hsl.presalestracker.zk.customer;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;

import org.zkoss.zk.ui.Page;
import org.zkoss.zkplus.databind.AnnotateDataBinderInit;

import si.hsl.presalestracker.beans.entity.Country;
import si.hsl.presalestracker.beans.session.Common;
import si.hsl.presalestracker.comparators.CountryComparator;
import si.hsl.presalestracker.util.ServiceLocator;

public class AddCustomerInitiator extends AnnotateDataBinderInit {
    @Override
    @SuppressWarnings("unchecked")
    public void doInit(Page page, Map arg1) {
        Common commonBean = ServiceLocator.lookup(Common.class);
        List<Country> allCountries = new ArrayList<Country>();
        allCountries.add(new Country());
        allCountries.addAll(commonBean.getAllCountries());
        Collections.sort(allCountries, new CountryComparator());
        page.setAttribute("countries", allCountries);

        super.doInit(page, arg1);
    }
}
```

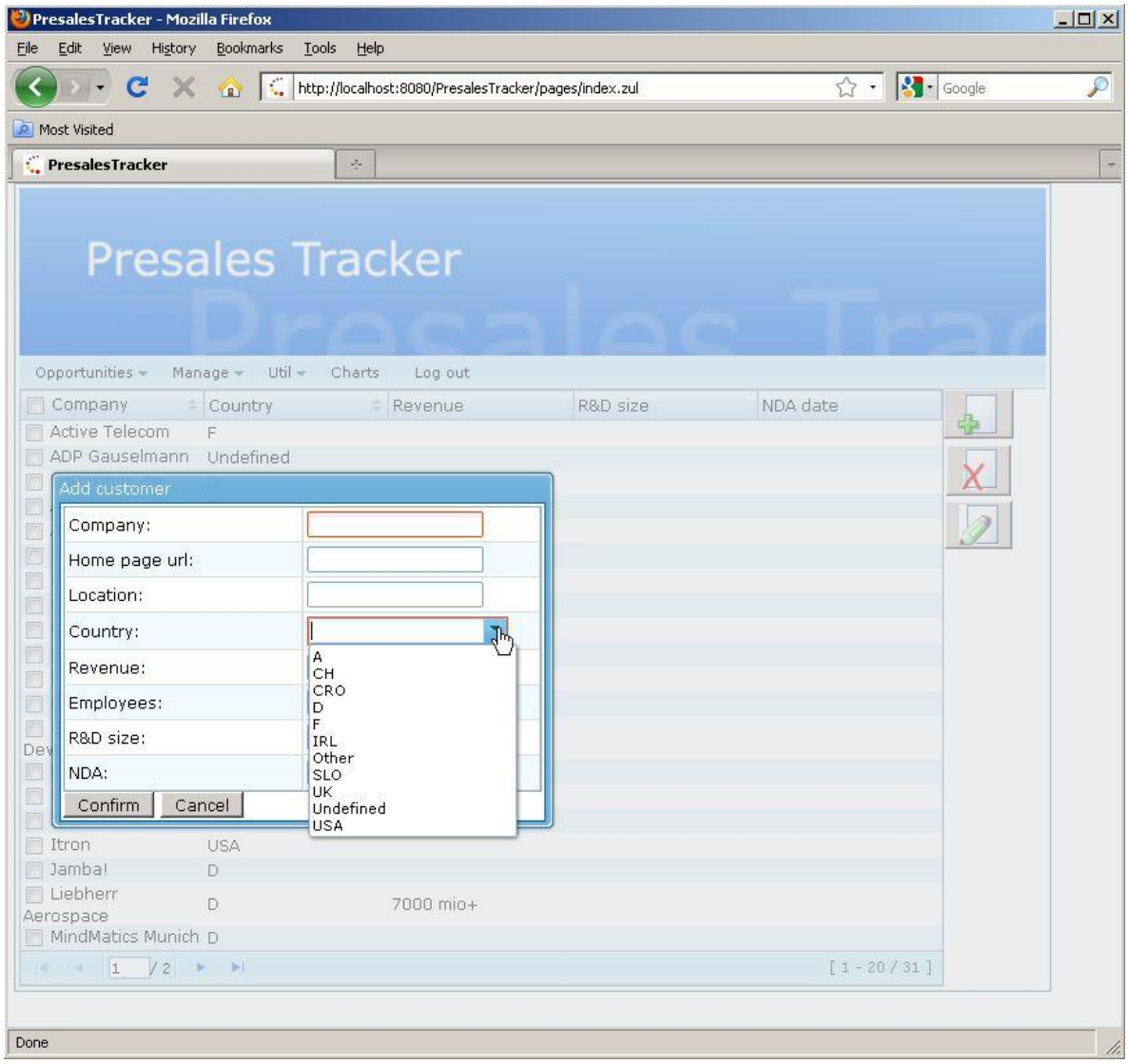
Metoda doInit se pokliče preden ZK prične z ustvarjanjem komponent in v njej lahko napolnimo podatkovne strukture, ki jih potrebujemo za predstavitev na določeni strani. Strani nato določimo kateri razred naj uporabi ob inicializaciji.

Slika 17: Uporaba povezave podatkov

```
<combobox id="country"
          selectedItem="@{addCustomerWindow.customer.country}"
          sclass="mandatory" model="@{countries}"
          onSelect='country.setSclass("")'>
  <comboitem self="@{each='country'}"
             label="@{country.countryName}"
             value="@{country.countryId}">
  </comboitem>
</combobox>
```

Spremenljivko 'countries' lahko sedaj uporabimo kot vir podatkov za combobox, kjer s posebno spremenljivko, imenovano 'each', napolnimo combobox s podatki, ki smo jih pred tem pripravili. Ker so posamezni elementi vzeti iz seznama objektov, ki so tipa 'Country' imamo tudi neposreden dostop do vseh njenih atributov in lahko brez dodatnega kodiranja določimo, da so v comboboxu vidna imena posameznih držav, vrednost posameznega elementa pa je kar primarni ključ iz podatkovne baze. Na ta način lahko dostopamo tudi do vseh ostalih atributov definiranih v posameznem java razredu in na enostaven način napolnimo podatkovne strukture.

Slika 18: Prikaz forme za dodajanje nove stranke



Kot vidimo iz primera, se izbrana država shrani tudi v spremenljivko *customer* v razredu, ki pripada oknu AddCustomerWindow in sicer kot atribut *country* znotraj objekta *customer*. Podobno so povezani tudi vsi ostali atributi, zato je dodajanje nove stranke zelo enostaven korak.

Slika 19: AddCustomerWindow.java

```
package si.hsl.presalestracker.zk.customer;

import org.zkoss.zul.Window;

import si.hsl.presalestracker.beans.entity.Customer;
import si.hsl.presalestracker.beans.session.ManageCustomers;
import si.hsl.presalestracker.util.ServiceLocator;
import si.hsl.presalestracker.util.GuiConstants.ModalWindowActionType;

@SuppressWarnings("serial")
public class AddCustomerWindow extends Window {
    private Customer customer = new Customer();

    private ModalWindowActionType actionType;

    public void onOK() {
           ManageCustomers manageCustomerBean = ServiceLocator
                 .lookup(ManageCustomers.class);
        manageCustomerBean.addCustomer(customer);

        actionType = ModalWindowActionType.CONFIRM;
        this.detach();
    }

    public void onCancel() {
        actionType = ModalWindowActionType.CANCEL;
        this.detach();
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public ModalWindowActionType getActionType() {
        return actionType;
    }
}
```

Kot vidimo, je dovolj zgolj, da objekt *customer*, ki je že ustreznega tipa podamo kot argument sejnemu zrnu, ta pa ga zapiše v podatkovno bazo.

4.3.3 Prikaz grafov s knjižnico JFreeChart

Ogrodje ZK vključuje tudi knjižnico JFreeChart (glej [8]) in vnaprej pripravljene komponente, ki omogočajo enostaven prikaz grafov. Aplikacija PresalesTracker to uporablja za prikaz osnovnih statističnih podatkov o obstoječih priložnostih. Kot primer si bomo pogledali kako se prikazujejo grafi glede na izbran status posamezne priložnosti.

Slika 20: ZUML koda za prikaz grafa

```
<chart id="chart" type="pie" threeD="true" fgAlpha="130"
  title="Opportunities by status" />
```

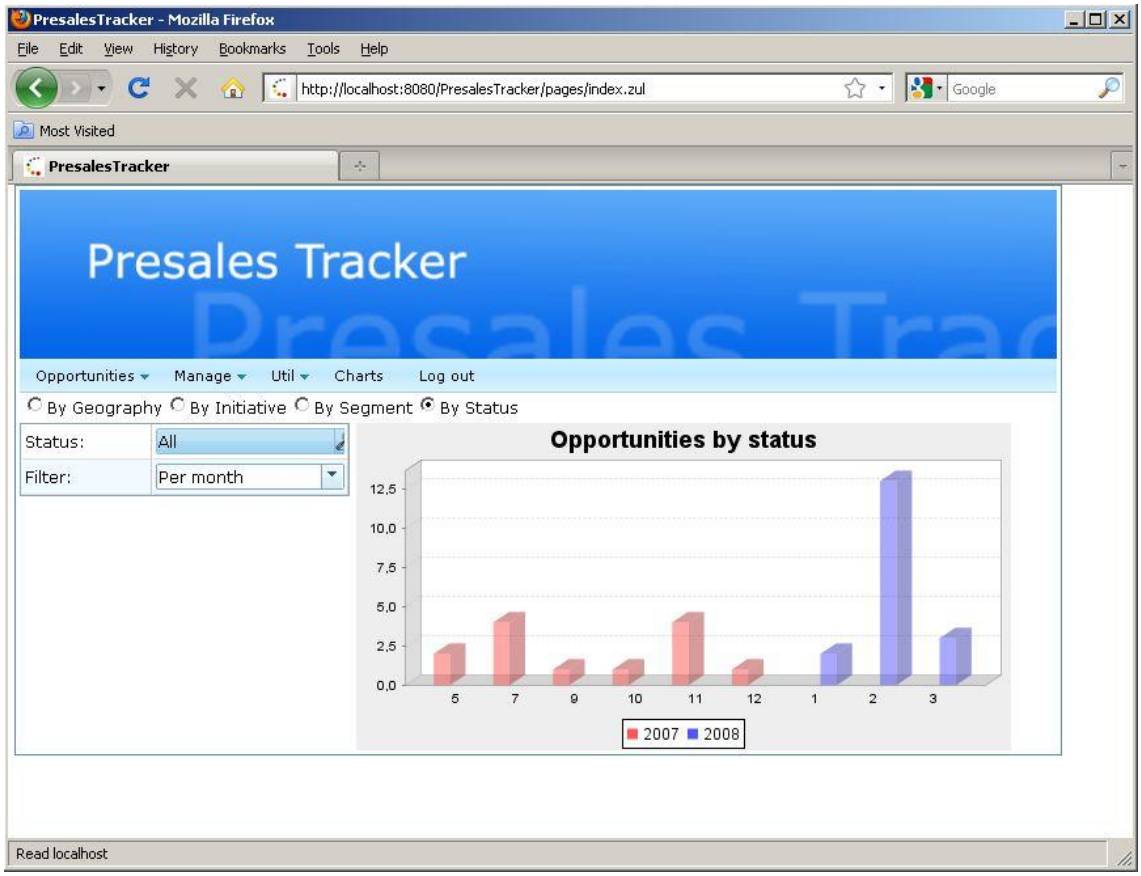
Kot vidimo iz slike 19, sta za prikaz grafa z ogrodjem ZK potrebni zgolj dve vrstici kode. To je dovolj, da ogrodje ZK, s knjižnico JFreeChart, pripravi vse potrebno za izris grafa v brskalniku, poskrbeti moramo le še za ustrezne podatke, kar storimo v pripadajočem java razredu.

Slika 21: Java koda potrebna za izris grafa

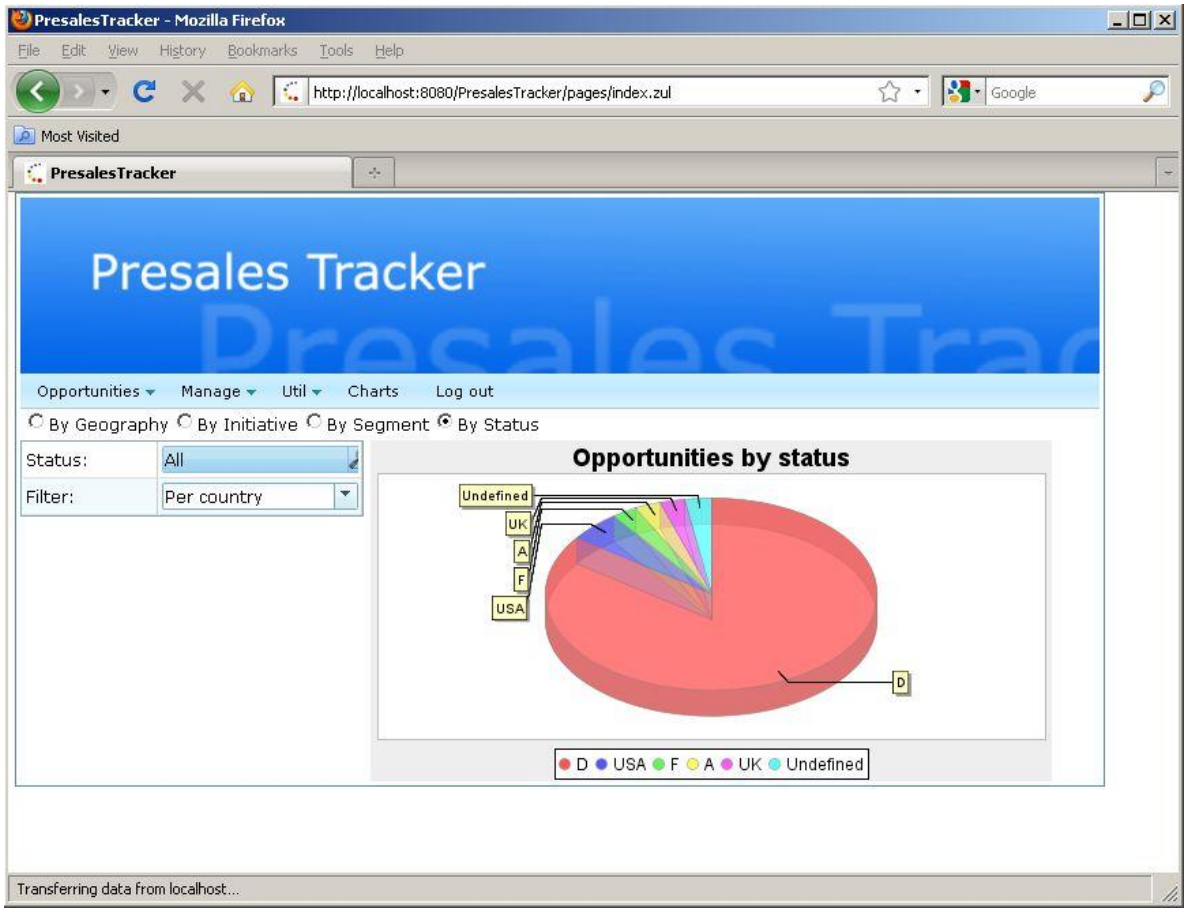
```
public void drawChart() {
	if (selectionFilter.equals(ChartBasicInitiator.PER_MONTH)) {
		// Draw bar chart
		drawBarChart(getBarChartData());
	} else {
		// Draw pie chart
		drawPieChart(getPieChartData());
	}
}

private void drawPieChart(Map<String, Integer> data) {
	if (data != null) {
		PieModel model = new SimplePieModel();
		for (String s : data.keySet()) {
			model.setValue(s, data.get(s));
		}
		chart.setType("pie");
		chart.setModel(model);
	} else {
		PieModel model = new SimplePieModel();
		chart.setType("pie");
		chart.setModel(model);
	}
}

private void drawBarChart(Map<Integer, Map<Integer, Integer>> data) {
	if (data != null) {
		CategoryModel model = new SimpleCategoryModel();
		List<Integer> yearKeys = new
				ArrayList<Integer>(data.keySet());
		Collections.sort(yearKeys);
		for (Integer year : yearKeys) {
			Map<Integer, Integer> yearData = data.get(year);
			List<Integer> monthKeys = new
				ArrayList<Integer>(yearData.keySet());
			Collections.sort(monthKeys);
			for (Integer month : monthKeys) {
				model.setValue(Integer.toString(year), Integer
					.toString(month), yearData.get(month));
			}
		}
		chart.setType("bar");
		chart.setModel(model);
	} else {
		CategoryModel model = new SimpleCategoryModel();
		chart.setType("bar");
		chart.setModel(model);
	}
}
```

V java razredu je treba pripraviti ustrezne podatke, jih shraniti v model in komponenti za izris določiti kateri tip grafa naj izriše ter ji podati podatkovni model. Tako smo zgolj z nekaj vrsticami kode prišli do grafične predstavitve podatkov, ki se lahko prikaže na več različnih načinov.

Slika 22: Prikaz stolpčnega grafa



Slika 23: Prikaz tortnega grafa



4.3.4 Razvrščanje podatkov v tabeli

Aplikacija PresalesTracker za prikaz obstoječih priložnosti uporablja tabelo z več stolpci, ki jih je, za zagotavljanje lažjega pregleda nad priložnostmi, možno vsakega posebej razvrstiti naraščajoče ali padajoče. Tudi tu je v veliko pomoč ogrodje ZK, zagotoviti je treba zgolj implementacijo algoritma za preverjanje razmerja večje / manjše, vso ostalo delo pa za nas opravi ZK.

Slika 24: ZUML koda potrebna za razvrščanje stolpcev

```
<listhead sizable="false">
  <listheader label="Company" sortAscending="${cmpCompanyA}"
    sortDescending="${cmpCompanyD}" width="115px" />
  <listheader label="Country" sortAscending="${cmpCountryA}"
    sortDescending="${cmpCountryD}" width="70px" />
  <listheader label="Opp Description" sortAscending="${cmpDescA}"
    sortDescending="${cmpDescD}" width="185px" />
  <listheader label="Status" sortAscending="${cmpStatusA}"
    sortDescending="${cmpStatusD}" width="68px" />
  <listheader label="Segment" sortAscending="${cmpSegmentA}"
    sortDescending="${cmpSegmentD}" width="72px" />
  <listheader label="Sales" sortAscending="${cmpSalesA}"
    sortDescending="${cmpSalesD}" width="80px" />
  <listheader label="Delivery" sortAscending="${cmpDeliveryA}"
    sortDescending="${cmpDeliveryD}" width="80px" />
</listhead>
```

V ZUML kodi je treba elementu *listheader* zgolj povedati kateri primerjalnik (*angl. comparator*) naj uporabi za naraščajoče in katerega za padajoče razvrščanje, ZK pa poskrbi za vse ostalo. V java kodi nam tako preostane zgolj inicializacija primerjalnikov in pa sama implementacija primerjalnikov, ki jo lahko za vse stolpce naredimo zgolj z enim java razredom, saj tabela vsebuje objekte tipa *Opportunity*.

Slika 25: Primerjalnik za razvrščanje priložnosti

```
package si.hsl.presalestracker.comparators;

import java.util.Comparator;

import si.hsl.presalestracker.beans.entity.Opportunity;
import si.hsl.presalestracker.beans.entity.User;

public class OpportunityComparator implements Comparator<Opportunity> {

    public enum Column {
        COMPANY, COUNTRY, DESCRIPTION, STATUS, SEGMENT, SALES, DELIVERY;
    }

    private boolean ascending;
    private Column column;

    public OpportunityComparator(boolean ascending, Column column) {
        this.ascending = ascending;
        this.column = column;
    }

    @Override
    public int compare(Opportunity opp1, Opportunity opp2) {
```

```
    if ((opp1 == null) && (opp2 == null)) {
        return 0;
    } else if (opp1 == null) {
        return 1;
    } else if (opp2 == null) {
        return -1;
    }

    int val = 0;
    switch (column) {
    case COMPANY:
        val =
           opp1.getCustomer().getCompanyName().toUpperCase().compareTo(
                opp2.getCustomer().getCompanyName().toUpperCase());
        break;
    case COUNTRY:
        val = opp1.getCustomer().getCountry().getCountryName()
                  .toUpperCase().compareTo(opp2.getCustomer().getCountry().
                  getCountryName().toUpperCase());
        break;
    case DESCRIPTION:
        val = opp1.getDescription().toUpperCase().compareTo(
                opp2.getDescription().toUpperCase());
        break;
    case STATUS:
        val = opp1.getStatus().getStatusName().toUpperCase().compareTo(
                opp2.getStatus().getStatusName().toUpperCase());
        break;
    case SEGMENT:
        val = opp1.getSegment().getSegmentName().toUpperCase().
           compareTo(opp2.getSegment().getSegmentName().toUpperCase());
        break;
    case SALES:
        val = compareUsers(opp1.getSalesPerson(),
                                opp2.getSalesPerson());
        break;
    case DELIVERY:
        val = compareUsers(opp1.getDeliveryPerson(),
                                opp2.getDeliveryPerson());
    default:
        break;
    }

    return ascending ? val : -val;
}

private int compareUsers(User user, User user2) {
    if ((user == null) && (user2 == null)) {
        return 0;
    } else if (user == null) {
```

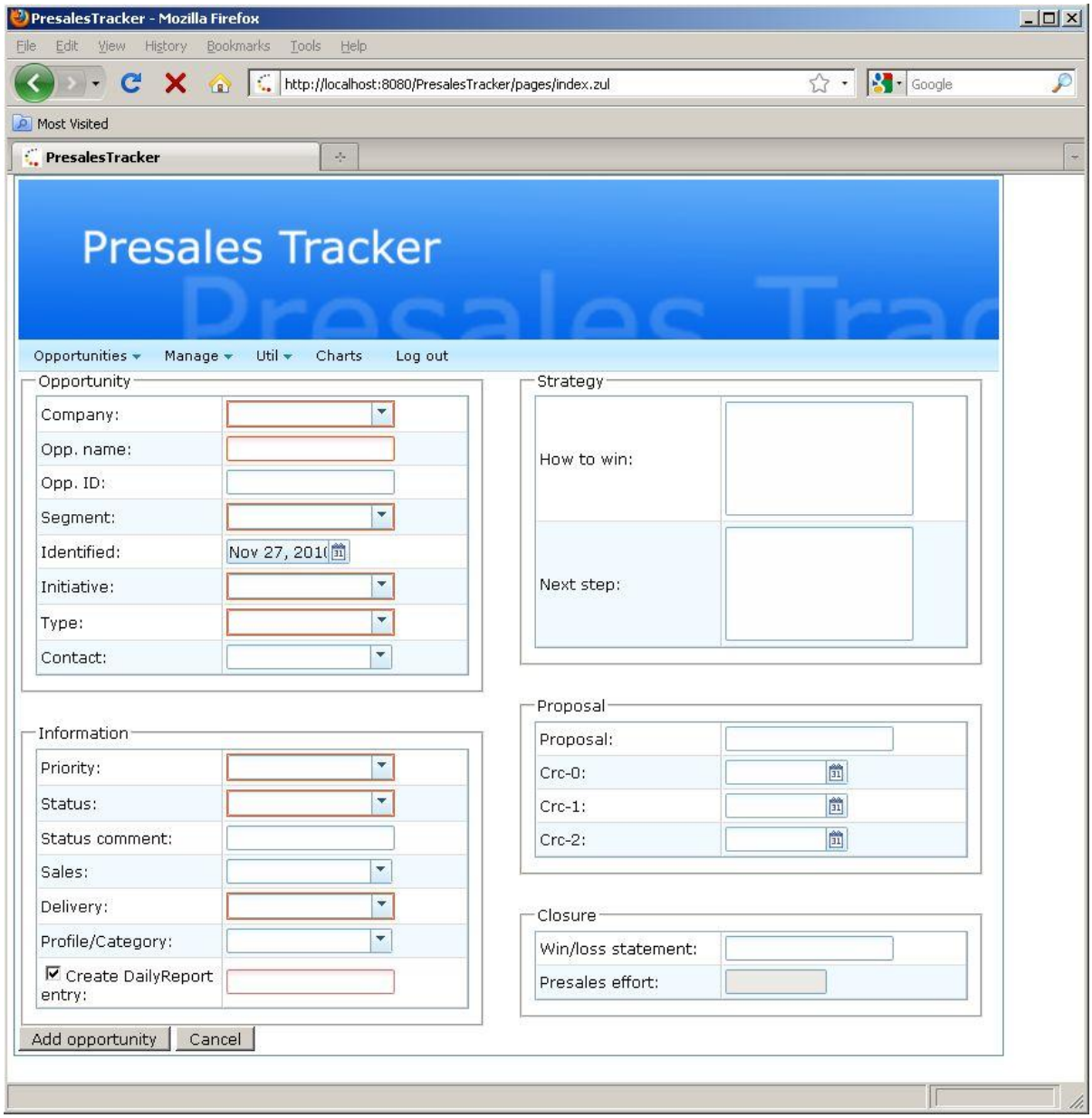
```
            return -1;
        } else if (user2 == null) {
            return 1;
        }
        String name = user.getFirstName() + " " + user.getLastName();
        String name2 = user2.getFirstName() + " " + user2.getLastName();
        return name.compareTo(name2);
    }
}
```

4.4 Prikaz funkcionalnosti aplikacije PresalesTracker

Kot primer, kako vsi trije nivoji aplikacije PresalesTracker delujejo kot celota, si bomo ogledali kako poteka dodajanje nove priložnosti. Sestavljeno je iz nekaj korakov, ki so porazdeljeni med posameznimi nivoji aplikacije.

Uporabniku se prikaže vnosna forma, kamor vnese zahtevane podatke o novi priložnosti.

Slika 26: Vnosna forma za dodajanje nove priložnosti



Obvezna polja so s pomočjo css označena, ob dodajanju nove priložnosti pa se izvede tudi validacija, ki preveri, da imajo vsa obvezna polja dejansko vnesene zahtevane vrednosti. Pri komponentah tipa *Combobox* to izvedemo v java razredu, saj je vedno izbran prvi element v seznamu, zato validacija, ki jo vsebuje ogrodje ZK, ne zadošča.

Slika 27: Koda za preverjanje vnesenih vrednosti v komponentah tipa *Combobox*

```
private void checkRequiredValue(Combobox combobox) {
    if (combobox.getSelectedItem() == null
            || combobox.getSelectedItem().getLabel() == null
            || combobox.getSelectedItem().getLabel().equals("")) {
        throw new WrongValueException(combobox, "Please select an item!");
    }
}
```

Po uspešno opravljeni validaciji je treba najprej v interni aplikaciji za beleženje dela (aplikacija DailyReport) ustvariti ustrezno strukturo, kamor se nato lahko beleži delo opravljeno na sami priložnosti in na koncu še dodati novo priložnost v podatkovno bazo. Za oboje se uporabijo ustrezna sejna zrna, ki implementacijo poslovne logike skrijejo od uporabniškega vmesnika.

Slika 28: Java koda predstavitvene plasti za dodajanje nove priložnosti

```
public void onClick$addOpportunityButton() {
    log.info("Add opportunity invoked");
    checkRequiredValue(company);
    checkRequiredValue(segment);
    checkRequiredValue(initiative);
    checkRequiredValue(type);
    checkRequiredValue(priority);
    checkRequiredValue(status);
    checkRequiredValue(delivery);

    if (dailyReportCheckbox.isChecked()) {
        String drProjectName = dailyReportEntry.getText();
        String drModuleName = opportunity.getCustomer().getCompanyName();
        String drTaskName = opportunity.getDescription();
        Session session = this.getPage().getDesktop().getSession();
        UserStatus userStatus = (UserStatus) session
                .getAttribute(Constants.USER_STATUS);
        String username = userStatus.getUsername();
        DailyReport dailyReportBean = ServiceLocator
                .lookup(DailyReport.class);
        try {
            dailyReportBean.createDrEntryForOpportunity(username,
                    drProjectName, drModuleName, drTaskName);
            opportunity.setDailyReportProject(drProjectName);
            opportunity.setDailyReportModule(drModuleName);
        } catch (DRException e) {
            log.error("Error creating opportunity", e);
            try {
                Messagebox.show("Opportunity can not be added. "
                        + "Problems with DR database. "
                        + "See log for details.", "Add opportunity",
                        Messagebox.OK, Messagebox.ERROR);
            } catch (InterruptedException ex) { // No action necessary
            }
            return;
        }
    }

    ManageOpportunity manageOpportunityBean = ServiceLocator
            .lookup(ManageOpportunity.class);
    manageOpportunityBean.addOpportunity(opportunity);
    try {
        Messagebox.show("Opportunity successfully added!",
                "Add opportunity", Messagebox.OK, Messagebox.EXCLAMATION);
    } catch (InterruptedException e) { // No action necessary
    }
    Executions.sendRedirect("/pages/index.zul");
}
```

Kot vidimo, se najprej izvede ustrezna validacija, nato pa se s sejnim zrnom DailyReport izvede dodajanje nove strukture v aplikacijo DailyReport in nato še s sejnim zrnom ManageOpportunity dodajanje nove priložnosti v podatkovno bazo. Na nivoju uporabniškega vmesnika je to vse, saj so vse podrobnosti komunikacije z aplikacijo DailyReport in s podatkovno bazo vsebovane znotraj ustreznih sejnih zrn.

Slika 29: Koda sejnega zrna DailyReport za dodajanje strukture nove priložnosti

```
@Stateless
@Local
public class DailyReportBean implements DailyReport {
	private static final Logger log =
				Logger.getLogger(DailyReportBean.class);
	@Resource(mappedName = "java:/DailyReportDS")
	private DataSource dataSource;
	private Connection connection;

	@SuppressWarnings("unused")
	@PostConstruct
	private final void initConnection() {
		log.info("Initializing connection.");
		try {
			connection = dataSource.getConnection();
		} catch (SQLException e) {
			log.error("Connection could not be initialized.", e);
		}
	}

	@SuppressWarnings("unused")
	@PreDestroy
	private final void closeConnection() {
		log.info("Closing connection.");
		try {
			connection.close();
		} catch (SQLException e) {
		}
	}

public void createDrEntryForOpportunity(String username,
	  String drProjectName, String drModuleName, String drTaskName)
	  throws DRException {
		DRProject drProject = getProjectInfo(drProjectName);

		int projectId = drProject.getProjID();
		int createModule = createModule(projectId,
					drModuleName, username);
		List<DRModule> modules = getProjectModulesInfo(projectId);
		int moduleId = -1;
		for (DRModule module : modules) {
			if (module.getModuleName().equals(drModuleName)) {
				moduleId = module.getModuleID();
				break;
			}
		}

		String categoryList = DR_CATEGORY_LIST_STRUCTURE;
		int createTask = createTask(drTaskName, moduleId, categoryList,
			username);

		log.info("Inserting DR entry status. Module: '" + createModule
			+ "' Task: '" + createTask + "'");
	}
}
```

Sejno zrno DailyReport z anotacijo @Resource vrine podatkovni vir (*angl. data source*), ki se prebere iz ustrezne xml datoteke, dostopne na strežniku, in nato v metodah označenih z anotacijama @PostConstruct in @PreDestroy poskrbi za odpiranje in zapiranje povezave s podatkovno bazo. Na ta način dosežemo, da vsaka instanca sejnega zrna odpre svojo

povezavo, in da to stori samo enkrat ter jo, ko se instanca sejnega zrna uniči, tudi zapre. Preko klica metode *createDrEntryForOpportunity* nato s klici ustreznih metod in shranjenih procedur (*angl. stored procedures*) ustvari ustrezno strukturo za novo priložnost v aplikaciji DailyReport. Če pri ustvarjanju strukture pride do napake, je plast uporabniškega vmesnika obveščena z izjemo (*angl. exception*) in lahko uporabniku prikaže ustrezno sporočilo.

Slika 30: Koda sejnega zrna ManageOpportunity za dodajanje nove priložnosti

```
package si.hsl.presalestracker.beans.session;

import javax.ejb.Local;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import si.hsl.presalestracker.beans.entity.Opportunity;

@Stateless
@Local
public class ManageOpportunityBean implements ManageOpportunity {
	@PersistenceContext
	private EntityManager em;

	@Override
	public void addOpportunity(Opportunity opportunity) {
		em.persist(opportunity);
	}
}
```

Celotna koda za dodajanje nove priložnosti je izredno kratka, saj že na nivoju uporabniškega vmesnika kot podatkovni vir za prikaz uporabljamo kar entiteto *Opportunity*, zato je treba zgolj vriniti entitetnega upravljavca in mu kot argument podati entiteto. S tem korakom je dodajanje nove priložnosti zaključeno, na podatkovnem nivoju ni potrebno nobenega posebnega dela, saj smo za vse poskrbeli že z nastavitvami ogrodja JPA.

Iz primera je razvidno kako ogrodji ZK in Java EE poenostavita razvoj, saj skrijeta veliko število kompleksnosti ter razvijalcu omogočata, da se osredotoči predvsem na implementacijo poslovne logike aplikacije in se ne obremenjuje z ostalimi stvarmi.

Gradnja uporabniškega vmesnika z jezikom ZUML je izredno enostavna, hkrati pa vsi gradniki delujejo po načelu AJAX in omogočajo uporabniku interaktivno izkušnjo in odziven uporabniški vmesnik. S sejnimi zrni in ogrodjem JPA je tudi implementacija poslovne logike enostavna, saj za dostop in shranjevanje podatkov v podatkovno bazo običajno zadostuje že

ena vrstica programske kode. S tem ko poslovno logiko skrijemo v sejna zrna, omogočimo tudi enostavno dodajanje različnih odjemalcev.

5 ZAKLJUČEK

V diplomski nalogi sem na primeru aplikacije PresalesTracker prikazal, katere gradnike vsebuje sodobna spletna aplikacija, kako jih povežemo v celoto in kaj nudi uporabniku.

Predstavil sem, da s tehniko AJAX omogoča odzivnost in uporabniško izkušnjo, ki še pred kratkim uporabnikom spletnih aplikacij ni bila na voljo in se že skoraj lahko postavi ob bok namiznim aplikacijam, pred katerimi pa ima eno, zelo pomembno, prednost. Ker gre za spletno aplikacijo, ki v celoti teče v spletnem brskalniku (brez dodatnih zahtev), na strani odjemalca ne potrebuje ničesar drugega, kot delujoč spletni brskalnik. S tem odpadejo vse tegobe, ki jih s seboj prinese nameščanje namiznih aplikacij, vključno s težavami ob posodobitvah. Spremembe ob posodobitvah ali novih verzijah so potrebne le na strežniku, kjer pa mnogo enostavneje nadzorujemo celotno okolje in tudi na ta način zagotovimo, da so posodobitve enostavne in ne ogrožajo stabilnosti odjemalca.

Minimalne so tudi strojne zahteve, saj aplikacija v celoti teče na strežniku in odjemalec služi le in zgolj za prikaz podatkov, zato aplikacija PresalesTracker uspešno teče na praktično vseh sistemih.

Uporabniški vmesnik je zelo odziven in interaktiven ter uporabniku z asinhronimi klici na strežnik omogoča nemoteno delo, saj mu nikoli ni treba čakati na ponovno nalaganje spletne strani. Še več, s knjižnico JFreeChart, ki je sestavni del ogrodja ZK, so uporabniku nemudoma na voljo tudi razni grafikoni, ki močno izboljšajo uporabniško izkušnjo in omogočajo boljši pregled nad obstoječimi priložnostmi.

Vse do sedaj naštete prednosti so iz vidika uporabnika zagotovo dobrodošle, vendar imajo mnogo manjšo vrednost, če s seboj prinašajo težaven in dolgotrajen razvoj. Da temu ni tako, je poskrbljeno z uporabo ogrodij ZK in Java EE, ki omogočata izredno enostaven razvoj kompleksnih spletnih aplikacij, saj sami poskrbita za mnogo stvari in razvijalcu omogočata, da se osredotoči na samo funkcionalnost.

Ogrodje ZK samo ugotovi v katerem brskalniku teče, katera verzija JavaScripta je v uporabi, na kakšen način prikazati HTML komponente ipd. ter razvijalcu omogoča, da razvija enotno

spletno aplikacijo za vse brskalnike, brez dodatnega dela, ki je potrebno, da JavaScript koda uspešno teče v vseh brskalnikih. Razvijalcu skrije vso kompleksnost JavaScripta in poskrbi, da se osredotoči zgolj na predstavitev podatkov, hkrati pa podpira izredno enostavno integracijo s plastjo poslovne logike, z uporabo povezave podatkov (*angl. data binding*). Uporabniškemu vmesniku lahko tako podamo kar entitete podatkovne baze in nam ni potrebno skrbeti za dodatno pretvorbo podatkov pred prikazom in ponovno, ko spremenjene podatke zapišemo nazaj v podatkovno bazo.

Dodatno razvoj poenostavi ogrodje Java EE (skupaj z aplikacijskim strežnikom), ki omogoča, da razvijalec s sejnimi zrni implementira posamezne elemente poslovne logike. Ni se mu treba ukvarjati s tem kako bodo uporabniki do zrn dostopali, ni potrebno skrbeti za dovolj veliko število zrn v pripravljenosti ipd., saj je vse to že del ogrodja Java EE in pripadajočega aplikacijskega strežnika.

Z ogrodjem JPA, ki je standardni del Jave EE, je izredno poenostavljen tudi dostop do podatkov, na nivoju poslovne logike skorajda ni treba poznati spodaj ležečega podatkovnega modela, saj razvijalec ves čas upravlja le z java objekti. Hkrati je poskrbljeno tudi za stalno konsistentnost teh java objektov s podatki v podatkovni bazi, vključno s predpomnjenjem preteklih poizvedb, ki močno pohitri delovanje aplikacije, saj potrebno komunikacijo s podatkovno bazo zmanjša na minimum.

Aplikacija PresalesTracker je sicer namenjena zgolj za interno uporabo v podjetju, kjer je bila razvita, vendar to ne zmanjšuje uporabnosti predstavljenih tehnologij. Osredotočimo se le na trenutno najbolj razširjenega ponudnika spletnih aplikacij, Google in lahko vidimo, da njihove spletne storitve (ki jih je vsak dan več) temeljijo na enakih principih, kot so bili opisani v tej diplomski nalogi in tudi na široki množici uporabnikov spletnim aplikacijam napovedujejo svetlo prihodnost.

KAZALO SLIK

VIRI IN LITERATURA

- [1] R. Asleson, N. T. Schutta, Foundations of Ajax, New York: Springer-Verlag, 2006
- [2] J. Gehtland, B. Albraith, D. Almaer, Pragmatic Ajax, Raleigh: The Pragmatic Bookshelf, 2006
- [3] A. T. Holdener, Ajax: The Definitive Guide, Sebastopol: O'Reilly Media, Inc., 2008
- [4] S. D. Olson, Ajax on Java, Sebastopol: O'Reilly Media, Inc., 2007
- [5] N. T. Schutta, R. Asleson, Pro Ajax and Java Frameworks, New York: Springer-Verlag, 2006
- [6] Ajax: A New Approach to Web Applications. Dostopno na: http://adaptivepath.com/ideas/essays/archives/000385.php
- [7] Ajax (Programming). Dostopno na: http://en.wikipedia.org/wiki/Ajax_%28programming%29
- [8] JfreeChart. Dostopno na: http://www.jfree.org/jfreechart/
- [9] MySQL Standard Edition. Dostopno na: http://www.mysql.com/products/standard/
- [10] Session Beans. Dostopno na: http://www.roseindia.net/ejb/SessionBean.shtml
- [11] The Java EE 5 Tutorial. Dostopno na: http://download.oracle.com/javaee/5/tutorial/doc/bnaay.html

- [12] ZK Architecture Overview. Dostopno na: http://www.zkoss.org/doc/architecture.dsp