Otto Virtanen

Hibernate ja ICEFaces sovelluskehyksien käyttö MVC-mallin mukaisessa J2EE sovellustuotannossa

Opinnäytetyö
Tietojenkäsittely

Joulukuu 2010

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

KUVAILULEHTI

MIKKELIN AMMATTIKORKEAKOULU Mikkeli University of Applied Sciences	**Opinnäytetyön päivämäärä** 3.12.2010
Tekijä(t) Otto Virtanen	**Koulutusohjelma ja suuntautuminen** Tietojenkäsittely
Nimeke Hibernate ja ICEFaces sovelluskehyksien käyttö MVC-mallin mukaisessa J2EE sovellustuotannossa	

Tiivistelmä

Hibernate sovelluskehyksen avulla voidaan automatisoida olioiden tallennus relaatiotietokantaan, joka on manuaalisesti toteutettuna yksi nykypäivän sovellustuotannon työläimpiä osa-alueita. Opinnäytetyön tavoitteena oli esitellä Hibernate, JSF ja ICEFaces niin hyvin, että lukija voi aloittaa näiden sovelluskehyksien peruskäytön opinnäytetyöstään saaman opastuksen perusteella.

Opinnäytetyössä esitellään yleisimmät arkkitehtuurit ja suunnittelumallit, joita käytetään Hibernate ja ICEFaces sovelluskehyksillä ohjelmoitaessa, opastetaan Hibernaten, JSF:n ja ICEFacesin toimintaa koodiesimerkein ja kerrotaan tärkeimmät ominaisuudet kustakin sovelluskehyksestä.

Opinnäytetyön esimerkkitapauksena tehtiin demo-sovellus, jolla voidaan hallita projekteja ja niihin osallistuvia henkilöitä, sekä tähän kuuluva tutoriaalityyppinen opas, jonka avulla demo-sovellus voidaan toteuttaa vaihe-vaiheelta. Esimerkkitapaus käsittelee monipuolisesti Hibernaten jaJSF:n päälle rakentuvan ICEFaces sovelluskehyksen käyttöä. Kuitenkin demo-sovellus haluttiin pitää helposti lähestyttävänä ja oppaan avulla helposti toteutettavana, joten siitä on tarkoituksella jätetty pois paljon molempien tekniikoiden ominaisuuksia ja työkaluja. Opinnäytetyön pohjalta pois jätettyjen ominaisuuksien opiskelu on kuitenkin helppoa ja Internetistä löytyy paljon materiaalia tähän opiskeluun.

Hibernate ja ICEFaces ovat hyviä tekniikoita nykypäivän web-sovellusten tekoon, sillä ne eivät sisällä lainkaan samoja toiminnallisuuksia, joten molemmille on sovelluksen arkkitehtuurissa selkeät roolit. Hibernate on sovelluksen mallin(model) tekniikka, kun ICEFaces JSF:n avulla toimii sovelluksen ohjaimena(view) ja näkymänä(controller). Hibernate pysyy luultavasti niin pitkään yleisimpänä ORM-ratkaisuna, kuin relaatiotietokantoja vielä käytetään. Hibernate on laadukas sovelluskehys, jonka käyttäjäkunta on valtava, joten sen kehitysyhteisö pysyy varmasti voimissaan vielä pitkään. Hibernaten voi jatkossa korvata kuitenkin oliotietokannat, mutta tuskin lähivuosina. ICEFaces:n tulevaisuus ei kuitenkaan vaikuta kovin valoisalta, koska kilpailua on todella paljon käyttöliittymäkehyksien rintamalla ja käyttäjäkunta on pieni verrattuna muihin vastaaviin tekniikoihin. Lisäksi ICEFaces:n jatkoon vaikuttaa tuleva HTML5 formaatti ja sen tuomat uudet työkalut rikkaiden käyttöliittymien tekoon.

Asiasanat (avainsanat)

Hibernate, ICEFaces, Java Server Faces, MVC-malli, Olio-relaatio mallinnus

Sivumäärä	**Kieli**	**URN**
37 + 52(tutoriaali)	Suomi	URN:NBN:fi:mamk-opinn2010A1741

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi Jukka Selin	**Opinnäytetyön toimeksiantaja** **MHG Systems oy**

DESCRIPTION

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

Date of the bachelor's thesis

3 December 2010

Author(s)
Otto Virtanen

Degree programme and option
Business Information Technology

Name of the bachelor's thesis

Hibernate and ICEFaces frameworks in Model View Controller and J2EE software development

Abstract

Hibernate automates the data conversion between a relational database and software objects, which is one of the most time consuming development tasks in today's web development. The goal of this thesis was to present Hibernate, JSF and ICEFaces so well that after reading the study the readers could start programming basic web-software by using these technologies.

This study presented the most common architectures and design patterns which used when programming with Hibernate and ICEFaces frameworks. The study also described the main features of those frameworks and included several code examples.

The example application was a CRUD(create,read,update,delete) type of web application for project management. By using the program the user can manage projects and people participating in projects. Also a tutorial type of guide was made to ease the understanding of the example application. The example application used Hibernate and ICEFaces in a versatile way but it is still easy to adopt and be made with the help of the tutorial. Both of these frameworks have hundreds of nice features that are not used on the example application but the example application along side the tutorial gives the knowledge learn and use those features easily.

Hibernate and ICEFaces are good technologies to make today's MVC(model, view, controller) web applications. Both technologies had their own features and their own role in the application's architecture. In MVC architecture Hibernate is the program's model when ICEFaces is the application's view and controller. Hibernate will probably be the most widely used ORM technology for a long time. Hibernate is a high quality framework with large number of users. As long as there are relational databases that are used with object oriented software, there is need for Hibernate. Object oriented databases could replace Hibernate but not likely in the next few years. The future of ICEFaces is not that bright. There are quite many frameworks for Java user interfaces with bigger number of users. In addition, the oncoming HTML5 format will probably make ICEFaces useless.

Subject headings, (keywords)

Hibernate, ICEFaces, Java Server Faces, Model View Controller, Object-relational mapping

Pages	Language	URN
37+52(tutorial)	Finnish	URN:NBN:fi:mamk-opinn2010A1741

Remarks, notes on appendices

Tutor
Jukka Selin

Bachelor's thesis assigned by
MHG Systems oy

SISÄLTÖ

1 JOHDANTO

Tämä opinnäytetyö ottaa kantaa siihen, mitä hyötyä avoimen Hibernate-ohjelmistokehyksen käytöstä voi olla Java EE sovelluskehityksessä. Opinnäytetyön toimeksiantajana toimii mikkeliläinen ohjelmistotalo MHG Systems oy, joka on yksi maailman johtavista bioenergia-alan toiminnanohjausjärjestelmien tarjoajista.

Opinnäytetyön tutkimusongelmana on, onko Hibernaten avulla mahdollista nopeuttaa ja parantaa toimeksiantajan ohjelmistotuotteiden tuotekehitystä. Lisäksi toimeksiantaja halusi tietää, miten Hibernate toimii yrityksessä käytössä olevan ICEFaces-sovelluskehyksen rinnalla.

Opinnäytetyön esimerkkitapauksena toteutettiin MHG Systemsille demo-sovellus, jossa on käytetty Hibernate sovelluskehystä MHG:lla jo käytössä olevan ICEFaces-käyttöliittymäsovelluskehyksen kanssa, sekä kuvitettu opas, joka opastaa demo-sovelluksen teon tutooriaalimaisesti vaihe vaiheelta.

Opinnäytetyön ensimmäisessä osassa käsittelen Hibernaten kannalta olennaisia tekniikoita ja arkkitehtuureita. Toisessa osassa esittelen sovelluskehyksien toimintaa, MVC-arkkitehtuurin sekä JavaServerFaces-, ja ICEFaces-sovelluskehyksien käyttöä. Kolmannessa luvussa esittelen kattavasti Hibernaten toimintaa. Opinnäytetyön lopussa esittelen esimerkkitapauksena toteutetun ohjelmiston pääpiirteet.

2 TIETOKANTAOHJELMOINTI JAVA EE YMPÄRISTÖSSÄ

Lähes kaikissa JAVA sovelluksissa tulee tilanne, jossa dataa tai olioiden tila tulee tallentaa pysyvästi johonkin tietovarastoon. Useimmiten tämä tarkoittaa JAVA EE ympäristössä tietokantaa, mutta yhtä hyvin tila voidaan tallentaa myös tekstipohjaisiin tiedostoihin (XML, CSV, EXCEL) tai tietokoneen kovalevylle tavukoodina. JAVA-sovellustuotannossa tietokantaa käytetään nykyisin lähes aina JDBC rajapinnan avulla. Myöhemmin mainittava Hibernate-sovelluskehys, monien muiden tietokantasovelluskehyksien tapaan, on myös rakennettu JDBC rajapinnan päälle. JDBC:tä voidaankin pitää yhtenä tärkeimmistä JAVAN tekniikoista, jonka päälle monet nykypäivän tiedon pysyvän tallentamisen välineet rakentuvat.

2.1 JDBC

Tietokannan tullessa mukaan Java ohjelmiin, tarvitaan tietokanta-ajuri(database driver). Tietokanta-ajuri on ohjelmisto joka kommunikoi tietokannan ja sovelluksen välissä. Ajuri siis tulkkaa tietokantaan kohdistuvat palvelupyynnöt halutun tietokannan syntaksin mukaisiksi pyynnöiksi. Kukin tietokannan hallintajärjestelmä tarvitsee omantyyppisensä ajurin. Tietokanta-ajurin ollessa erillään muusta sovelluksesta, voidaan tietokannan hallintajärjestelmää päivittää ja jopa vaihtaa, niin että itse sovellukseen ei välttämättä tarvitse tehdä muutoksia. (Silander ym. 2010, 301.)

JDBC (Java Database Connectivity) Sunin kehittämä, yllä mainitun kaltainen tietokanta-ajuri Java-ohjelmiin. JDBC ei rajoitu ainoastaan relaatiotietokantojen käyttöön, vaan sen avulla voidaan käyttää myös muita taulukkomuodossa olevia tietolähteitä(esim. Excel ja XML tiedostot). Ohjelmoijalle JDBC tarjoaa luokkia ja rajapintoja, joilla tietokantaa voidaan käyttää tietokannan hallintajärjestelmästä riippumatta. Tärkein näistä lienee Java.sql paketti, joka tarjoaa työkaluja yhteyden luomiseen, sekä tietojen hakuun ja muokkaamiseen. (Silander ym. 2010, 302.)

2.2 Olio-relaatio mallinnus (ORM, Object relational mapping)

Vaikka suurin osa nykyisin suoritetusta ohjelmoinnista on olio-ohjelmointia, oliopohjaiset tietokannat ovat kuitenkin vielä harvinaisia. Yleisin tapa isoja Web-sovelluksia tehdessä on ohjelmoida sovellus olioina, mutta tallentaa olioiden tiedot relaatiotieto-

kantaan. Olioiden tallentaminen relaatiotietokantaan ei onnistu suoraan, vaan olio täytyy muuntaa tietokantaan tallentaessa relaatiomallin mukaiseksi, sekä tietokannasta haettaessa takaisin olioiksi. (Object Relational Mapping Strategies 2010)

”Olio-relaation mallinnuksella tarkoitetaan tekniikkaa, jolla konvertoidaan dataa yhteen sopimattomien olio-ohjelman ja relaatiotietokannan välillä.” (Wikipedia 2010) Orm-mallinnuksessa tuodaan uusi kerros tietokannan ja sovelluksen välille, joka tuo mukanaan tiedon validoinnin, eheyden tarkistuksen, tietokantariippumattomuuden ja transaktioiden hallinnan. Eli ORM-kerros huolehtii myös, että tietokantaan ei voida tallentaa virheellistä, puutteellista tai turvatonta tietoa. Uusi kerros myös mahdollistaa paremmin tietokannanhallintajärjestelmän vaihtamisen, koska käytettävä tietokanta piilotetaan varsinaiselta sovellukselta. (Object Relational Mapping Strategies 2010.) Olio-relaatiomallinnuksen voidaan toteuttaa itse JDBC:llä, mutta usein puhutaan että se on tietokantaohjelmoinnin työläin osuus. Näiden mallien välille onkin kehitetty useita sovelluskehyksiä kuten: Hibernate, Ibatis, Java Persistence Api(JPA), TopLink ja Java Data Objects(JDO). (Object Relational Mapping Strategies 2010.)

Tiedon käsittely olioina

Tiedon käsittelyyn olioina on kaksi koulukuntaa: skandinaavisessa koulukunnassa on tärkeintä mallintaa reaalimaailman käsitteitä olio-ohjelmoinnissa, kun amerikkalainen koulukunta tarkastelee olioita niiden yhteistoiminnan avulla. Todellisuudessa oikeissa sovelluksissa harvoin noudatetaan ainoastaan jompaakumpaa vaihtoehtoa. Sovelluksen perusmalli pystytään mallintamaan helposti reaalimaailman käsitteillä ja suhteilla, mutta laajoille kokonaisuuksille löytyy kuitenkin harvoin suoraan reaalimaailman vastinetta. Joskus myös reaalimaailman käsitteiden mukaan ohjelmoiminen on työläämpää, suorituskyvyltään hitaampaa ja kalliimpaa.(Kuha 2010, 99)

Olioiden mallintaminen

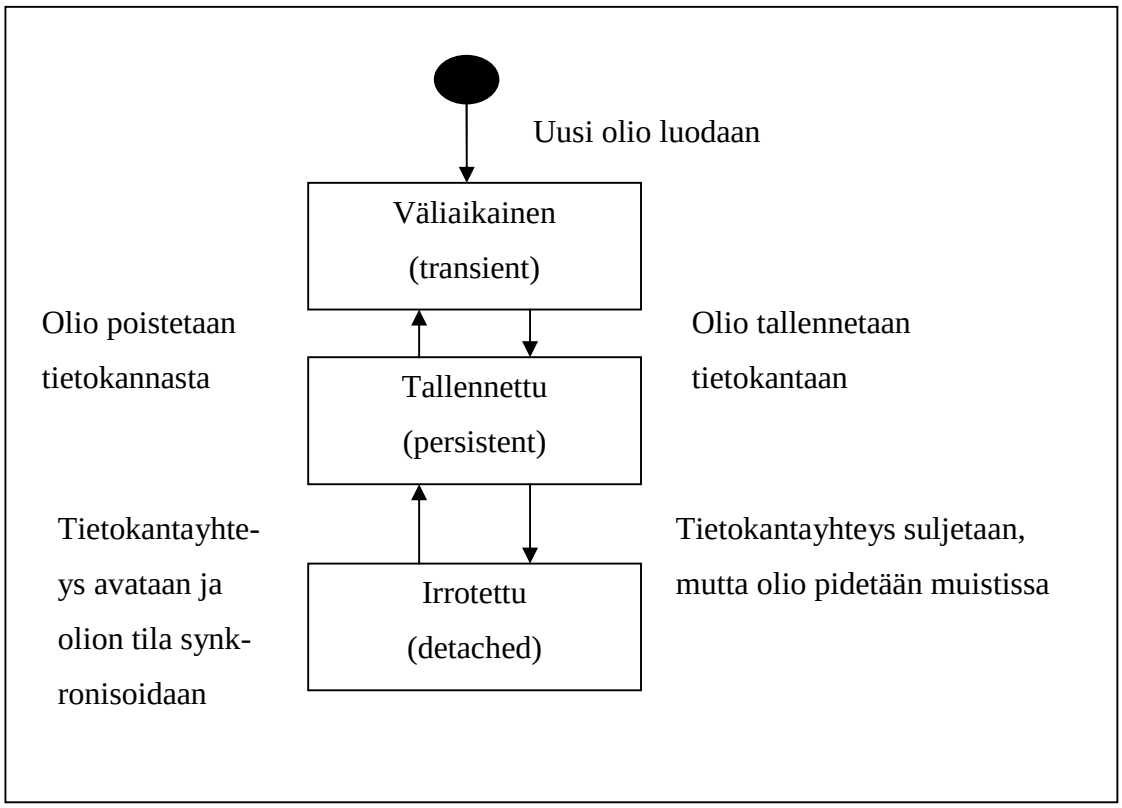
Mallintamisella ts. kuvaamisella (mapping) tarkoitetaan sitä, miten sovelluksen tiedot mallinnetaan relaatio- ja oliomallin välillä. Mallintamisessa kuvataan siis miten sovelluksen oliot, niiden tiedot ja olioiden väliset suhteet tallennetaan pysyvästi relaatiotietokantaan ja haetaan sieltä takaisin. Mallintaminen suoritetaan yleensä joko JAVA

5 mukana tulleilla annotaatioilla(metatiedot Java luokissa) tai erillisissä XML-tiedostoissa.(Kuha 2009, 99)

Oliosuhteiden mallintamisella tarkoitetaan sovelluksen olioiden välisten suhteiden mallintamista relaatiomallin keinoin, useimmiten tämä ilmaistaan tietokannassa perusavaimien välisenä linkityksenä. Oliosuhteiden tyyppejä on yhdestä yhteen(1-1), yhdestä moneen(1-n) ja monesta moneen(n-n). Oliosuhteet on helppo ymmärtää esimerkkien avulla. Yhden suhde yhteen olio-suhde voidaan kuvata esimerkiksi henkilön ja puhelinnumeron avulla: yhdellä henkilöllä voi olla yksi numero ja yhdellä numerolla voi olla vain yksi henkilö. Samalla tapaa yhden suhde moneen voitaisiin kuvata henkilön ja puhelin olion avulla: yhdellä henkilöllä voi olla monta puhelinnumeroa, mutta puhelinnumerolla voi olla vain yksi omistaja. Monen suhde moneen voitaisiin periaatteessa myös kuvata puhelinnumeron ja henkilön avulla: henkilöllä voi olla monta puhelinnumeroa ja yhdellä puhelinnumerolla voi olla monta käyttäjää(esim. kotipuhelin). Oliosuhteilla voi olla myös suunta(yksisuuntainen suhde, unidirectional) tai sitä voidaan kulkea molempiin suuntiin(kaksisuuntainen suhde, bidirectional). Suunnalla tarkoitetaan siis sitä, kumman olion kautta navigaatio tapahtuu. Mikäli suunta on kaksisuuntainen, voidaan hakea henkilöön liittyvät puhelinnumerot, kuten myös puhelinnumeroon liittyvät henkilöt. Yksisuuntaisessa suhteessa voidaan hakea tietysti vain yksisuuntaisesti. (Kuha 2009, 99-100)

Olion tila

Sovelluksen oliot ovat palvelimen työmuistissa, toisin kuin tietokannan tiedot. Muistia ei kuitenkaan ole rajattomasti, joten siellä on yleensä tallessa vain osa sovelluksen mallista. Sovellus ottaa yhteyden tietokantaan vasta sitten, kun tietoa ei löydy käsittelymuistista. Olioiden tila voi siis olla eri käsittelymuistissa ja tietokannassa niin kauan kunnes se lopulta tallennetaan tietokantaan. Sovelluskehyksissä olion tila on jaettu kolmeen osaan: Väliaikainen(transient), tallennettu(persistent) ja irrotettu(detached). Alapuolen kuvassa on selvennetty miten useimmat olio-relaatiosovelluskehykset ymmärtävät olion tilan muutokset.



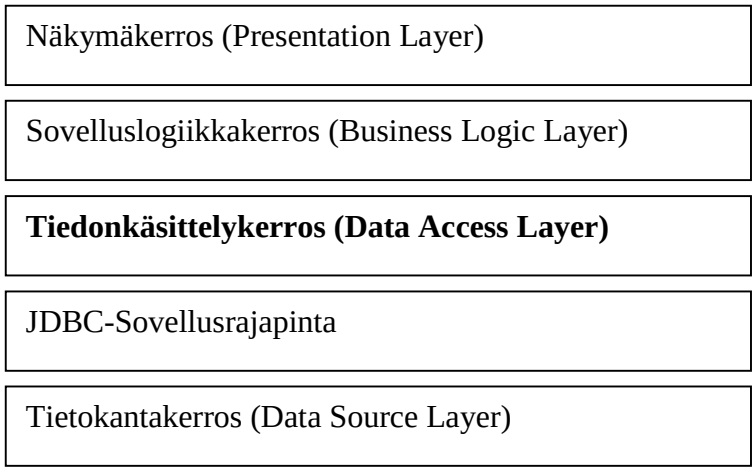
KUVA 1. Olion tila orm-kehyksissä (Kuha 2009, 104)

Olion tilan ollessa *väliaikainen*, kyseinen olio on luotu, mutta sitä ei ole vielä kytketty tietokannan tietoihin. Olion tila on yleensä väliaikainen silloin, kun sille ei ole vielä luotu perusavainta. *Tallennettu* -tilassa oleva olio on yhteydessä tietokannan tietoihin, jolloin olion tila synkronoituu tietokannan rivin kanssa. Olio on silloin *irrotettu*-tilassa, kun tietokantayhteys ei ole aktiivinen. Olion tila on haettu tietokannasta, mutta sen käsittelyyn kuluva aika on liian pitkä, jotta tietokantayhteyttä voitaisiin pitää auki koko muutosten ajan. Näille tilanteille on myös nimi: pitkät transaktiot(long running transaction). Olion tila voidaan muuttaa *tallennettu*-tilaan, kun tietokantayhteys avataan uudelleen.(Kuha 2009, 104)

2.3 DAO-suunnittelumalli

DAO-suunnittelumalli on tärkeimpiä suunnittelumalleja nykypäivän sovellustuotannossa. Aiemmin tietokannan operointi on tehty sovelluslogiikan luokissa, jolloin sovelluslogiikkakerroksessa käsitellään myös tietokantakerroksen toimintaa. Tämä tekee ohjelmasta hankalan muokata ja kehittää, koska tietokannan käsittelylauseet ovat useissa kohtaa koodia. DAO:n ideana on ORM:n tapaan eristää tietokantatoiminnallisuus ohjelmakoodissa omaan kerrokseensa, jolloin muun sovelluksen ei tarvitse tietää

tietokantatoteutuksesta juuri mitään. DAO:n hyödyistä kertoo hyvin myös se, että mikäli taustalla oleva tietokanta vaihdetaan, tulee sovellukseen muutoksia ainoastaan tietokantakerrokseen, kun taas muu sovelluslogiikka pysyy entisellään. Suunnittelumallin mukana sovellukseen tulee uusi tiedonkäsittelykerros(Data Access Layer), joka sisältää metodit tietokantaoperaatioiden toteuttamiseen. Tiedonkäsittelykerroksen sijainti sovelluksen kerrosarkkitehtuurissa on kuvattuna kuvassa 2. (Silander ym. 2010, 438)



KUVA 2. Sovelluksen kerrosarkkitehtuuri DAO -suunnittelumallin mukaisessa ohjelmoinnissa (Silander ym. 2010, 439)

DAO-malli tarjoaa lähinnä suuntaviivat suunnittelulle, se ei ota kantaa miten sovelluslogiikka- ja tiedonkäsittelykerros keskustelevat keskenään. Tiedonkäsittelykerrokselle dataa välitettäessä käytetään myös DTO(Data Transfer Object), eli tiedonsiirto-oliota, joilla pyritään paketoimaan datan käsittelyä ja ylimääräisiä tietokantapäivityksiä. Tiedonkäsittelykerros toteutetaan yleisesti kolmella eri tavalla. Ensimmäisessä vaihtoehdossa tiedonkäsittelykerros koostuu ainoastaan yhdestä luokasta, joka vastaa kaikista tietokantaoperaatioista. Toinen vaihtoehto on luoda jokaista kohdealueen luokkaa kohden oma DAO-luokkansa. Tällöin DAO, luokasta tehdään yksi ilmentymä koko ohjelman suorituksen ajaksi. Kolmas vaihtoehto on tehdä toisen vaihtoehdon tavoin jokaista kohdealueen luokkaa kohden oma DAO-luokkansa, mutta josta tehdään yksi ilmentymä jokaista kohdealueen oliota kohden.(Silander ym. 2010, 439)

3 AVOIMET OHJELMISTOKEHYKSET J2EE SOVELLUSTUOTANNOSSA

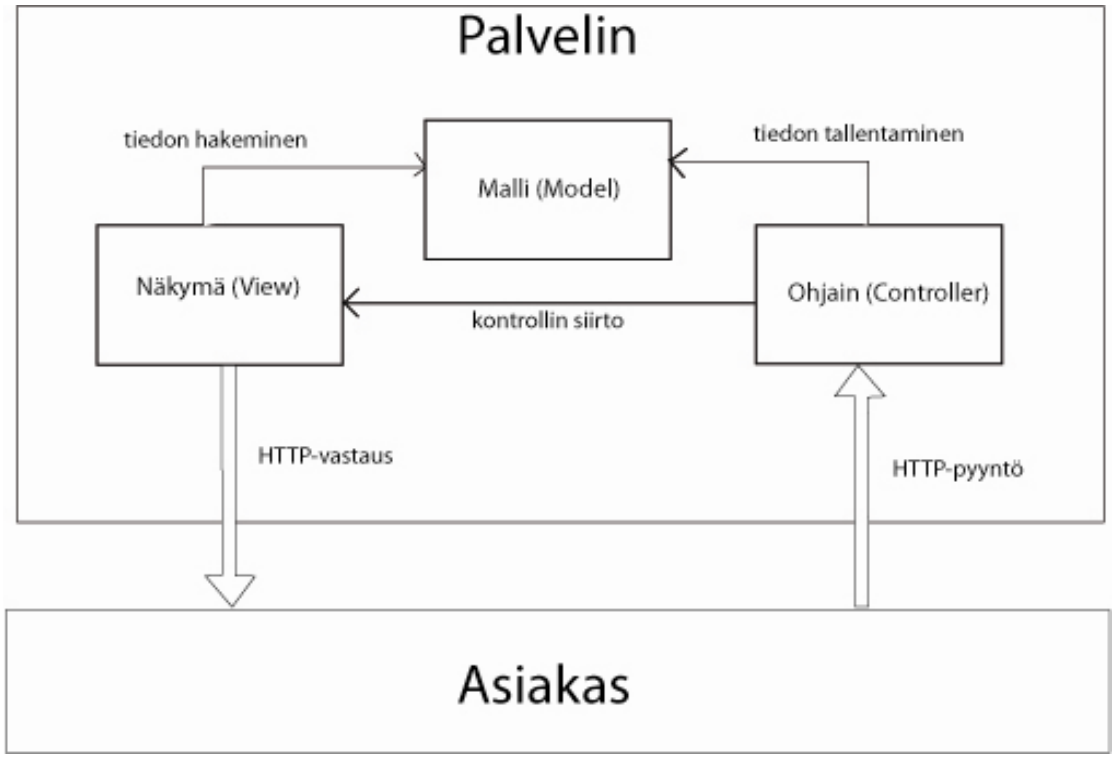
Useimmat verkkosovellukset ovat arkkitehtuurisilta ominaisuuksiltaan samankaltaisia keskenään. Nykypäivän ohjelmoinnissa onkin ymmärretty erottaa nämä yhteiset ominaisuudet erilleen muusta sovelluksesta. Ohjelmistokehys on siis kokoelma luokkia, komponentteja ja rajapintoja yhteisen arkkitehtuurin ja perustoiminnallisuuden toteuttamiseen. (Koskimies & Mikkonen 2005, 187.)

3.1 MVC-arkkitehtuurimalli

Nykypäivän WEB-ohjelmoinnin yksi keskeisin ohjelmistoarkkitehtuuri on MVC(Model, View, Controller), joka perustuu sovelluksen jakamiseen kolmeen osaan: Model-osaan joka huolehtii tiedon tallentamisesta, ylläpidosta ja käsittelystä, View-osaan joka huolehtii asiakkaalle näkyvästä näkymästä ja Controller-osaan joka vastaanottaa käyttäjän pyynnöt ja muuttaa mallia ja näkymää niiden perusteella. (Harju & Juslijn 2009, 245.)

Ohjelmoijille MVC-malli tulee useimmiten ensimmäisen kerran esille, kun aletaan tutustua erilaisiin ohjelmistokehyksiin. MVC-malli on kuitenkin keksitty jo 1970-luvulla, vaikkakin sen periaatteet on esitelty kunnolla vasta 1988 MVC Cookbook-artikkelissa. MVC on arkkitehtuurimalli, vaikka monesti siitä puhutaan suunnittelumallina (design pattern). Suunnittelumalli on kuitenkin tarkempi määritelty ratkaisumalli, kun taas MVC:n avulla ainoastaan esitellään ohjelman rakenne. (Vahtolammi Kari 2010.)

Malli on ainoa kerroksista, joka on yhteydessä tietovarastoon. MVC-mallissa ohjain tai näkymä ottaa yhteyden tietokantaan aina mallin kautta(kts. kuva 3). Malli huolehtii käytännössä sovelluksen liiketoimintalogiikasta, eli tiedon tallentamisesta tietokantaa ja sessioiden käsittelystä. (Vahtolammi Kari 2010.)



KUVA 3. Model-View-Conroller-arkkitehuurimalli(Harju & Juslijn 2009, 246)

Näkymä kerros huolehtii tiedon esittämisestä sovelluksen käyttäjälle, pyytäen tarvittavat tiedot toteutustavasta riippuen joko mallilta tai ohjaimelta. Useimmiten näkymä tekee esityksen web-maailmassa HTML sivuna, mutta yhtä hyvin näkymä voi olla myös tehty muussakin muodossa, kuten XML, CSV, tekstitiedostona tai kuvina. (Eckstein 2007.)

Ohjaimen tärkein tehtävä on vastaanottaa HTTP-pyynnöt asiakkaalta, joiden pohjalta se muokkaa sovelluksen tilaa mallin avulla ja lopuksi ohjain valitsee asiakkaalle oikean näkymän. Toimintojen jakaminen osiin sovelluksessa voi olla välillä vaikeaa, koska esimerkiksi syötteen tarkistaminen voi kuulua toteutustavasta riippuen joko mallille tai ohjaimelle. Tärkeintä MVC-mallin mukaisessa ohjelmassa onkin erottaa malli ja näkymä toisistaan. Näkymä ei sisällä koskaan tietokantakoodia, eikä malli HTML-koodia. Käytännössä JAVA EE-ohjelmoinnissa ohjaimina toimivat Servletit ja erilaiset tapahtumankäsittelijät. (Eckstein 2007.)

MVC arkkitehtuurin avulla sovelluksesta tulee helpommin muokattava, koska sovelluksen osat ovat erillään toisistaan. Mikäli halutaan muuttaa sovelluksen käyttöliittymää, tehdään muutoksia näkymään ja ohjaimeen. Toisaalta jos halutaan muuttaa taustalla olevaa tietokantaa, muutoksia tulee ainoastaan malliin. MVC mahdollistaa myös

uusien näkymien lisäämisen käytössä olevaan sovellukseen helposti ja helpottaa projekteissa olevien henkilöiden työnjakoa. Ulkoasusta huolehtivan ei tarvitse miettiä mallia tai ohjainta taustalla, vaan hän voi suunnitella ja muokata rauhassa sovelluksen näkymiä. (Eckstein 2007.)

3.2 Java Server Faces 2.0 (JSF)

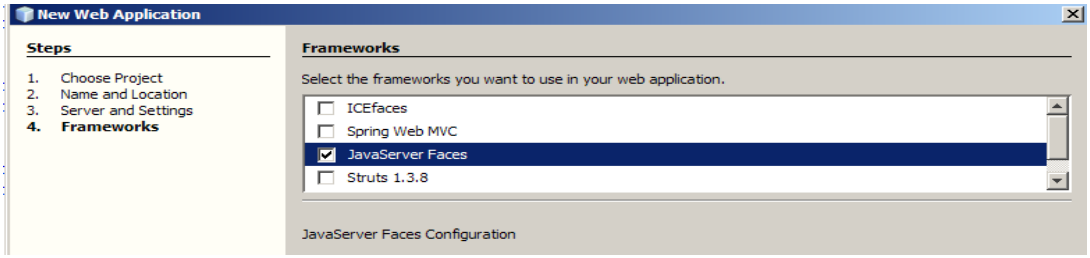
Java Server Faces (JSF) on Sun Microsystems:n kehittämä standardoitu sovelluskehys Java-pohjaisten web-sovelluksien käyttöliittymien luontiin. JSF on palvelinpuolen tekniikka, jonka avulla voidaan tehdä rikkaita Internet-sovelluksia(RIA, Rich Internet Applications). Rikkailla Internet-sovelluksilla tarkoitetaan web-sovelluksia, joiden käytettävyys ja ominaisuudet vastaavat perinteisiä ikkunasovelluksia.
(Mahmoud, 2004.)

JSF:n avulla kirjoitettavan html:n määrä vähenee, sen sisältäessä runsaasti valmiita käyttöliittymäkomponentteja(valintalistat, taulukot, tekstikentät jne.). Komponentteihin voidaan myös yhdistää mm. tiedon oikeellisuuden tarkistuksia. Sovelluskehys huolehtii automaattisesti lomakkeiden tietojen muuntamisen, sekä uudelleenkysymisen mikäli tiedoissa on puutteita. JSF:ää käytettäessä painikkeen tai linkin klikkaus käsitellään suoraan tapahtumana, jolloin vältytään käyttäjän pyyntöjen tutkimisilta. (Vesterholm & Kyppö 2008, 570.)

JSF koostuu Java Server Faces APIsta ja kahdesta tagikokoelmasta. API komponentti huolehtii käyttöliittymäkomponenttien esittämisestä ja niiden tilan hallinnasta. Tarkemmin sanottuna API huolehtii käyttöliittymäkomponenttien- tapahtumankäsittelystä, palvelinpuolen validoinnista ja tietotyyppien muunnoksista. Tagikokoelmat puolestaan mahdollistavat käyttöliittymäkomponenttien luonnin JSF sivuille ja niiden yhdistämisen palvelinpuolen olioihin.

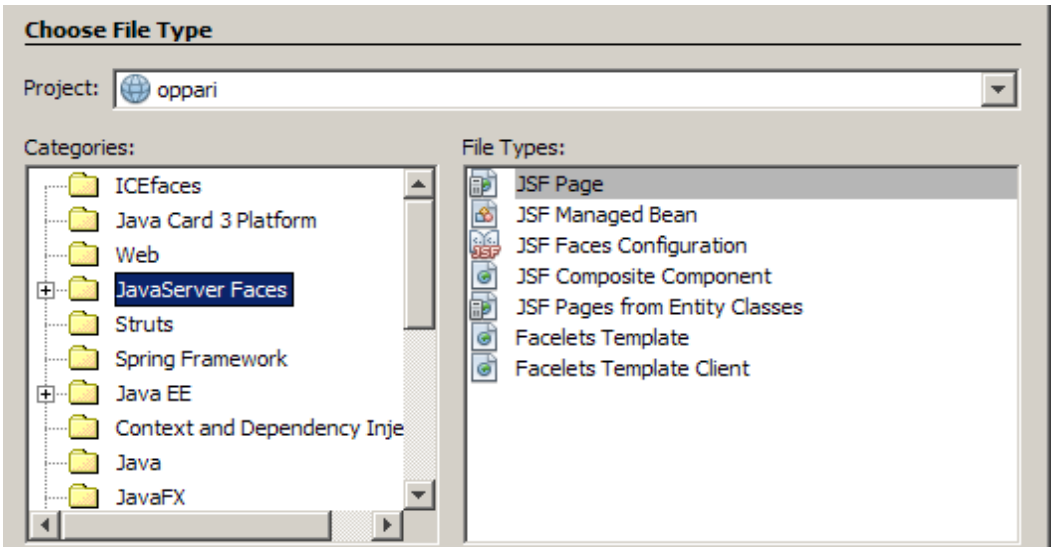
Käyttöönotto

Java Server Faces 2.0 tulee suosituimpien sovelluskehittimien kuten Netbeansin ja Eclipsen uusimpien asennuspakettien mukana, jolloin tiedostotynkien ja koodivinkkien avulla sovelluskehitys on huomattavasti helpompaa. JSF:n käyttöönotto esimerkiksi Netbeans projektissa on seuraavanlainen: Luodaan vain uusi Web-projekti, annetaan projektille nimi ja valitaan sovelluspalvelin, laitetaan ruksi Framework-listasta kohtaan JavaServer Faces(kts. kuva 4), jolloin Netbeans tekee automaattisesti tiedostotyngät JSF:n Facelet-sivulle ja määrittelee FacesServletin web.xml tiedostossa.



KUVA 4. JavaServer Facesin liittäminen Netbeans projektiin

Tapahtumankäsittelyä varten tarvitaan JSF sovelluksessa myös erilliset tapahtumankäsitteljäluokat ja faces-config.xml tiedosto, jossa kerrotaan JSF:lle mistä tapahtumankäsittelijät löytyvät ja miten niihin viitataan Facelet-sivuilla. Sovelluskehittimien avulla saadaan valmiiksi tiedostotyngät, jolloin kehittäjän ei tarvitse muistaa XML-tiedostojen määrittelyjä ulkoa. Kuvassa 5 on esitelty JSF:n tiedostotynkien valikko Netbeans-sovelluskehittimessä.



KUVA 5. JavaServer Facesin tiedostopohjat Netbeans sovelluskehittimessä

Komponentit

JSF-sovelluskehys sisältää kaksi tagikirjastoa, jotka sisältävät valmiita käyttöliittymäkomponentteja ja komponentteihin liitettäviä toimintoja. Komponenttien avulla saadaan yhdistettyä käyttöliittymä palvelinpuolen olioihin. JSP-tiedostot korvattiin versiosta 2.0 lähtien Facelet xHTML sivuilla, joihin on liitetty JSF:n tagikirjastot. Tällä saadaan varmistettua muun muassa se, että näkymä (kts. Luku 3.1) ei sisällä mallille tai ohjaimelle kuuluvia toimintoja. Alapuolen kuvassa 4 on esimerkki JSF sovelluskehyksen Facelet sivusta. (JavaBeat Website, 2009)

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://java.sun.com/jsf/html"
6        xmlns:f="http://java.sun.com/jsf/core">
7      <head>
8          <title>JSF esimerkki</title>
9      </head>
10     <body>
11         <f:view>
12             <h:form>
13                 <h:outputText value="Lisää uusi arvo"/>
14                 <h:inputText value="#{olio.uusiArvo}" />
15                 <h:commandButton actionListener="#{OjainLuokka.lisaaArvo()}"
16                 type="submit" value="Lisää" />
17             </h:form>
18         </f:view>
19     </body>
20 </html>
```

KUVA 6. Yksinkertainen JSF:n facelet –sivu

Aluksi sivulla määritellään, että sivu on xHTML-dokumentti. Varsinaiset JSF:n kirjastot otetaan käyttöön html-tagin sisällä seuraavasti:

xmlns:f=http://java.sun.com/jsf/core”
xmlns:h=http://java.sun.com/jsf/html”

JSF:n H-tagi sisältää sovelluskehyksen omat käyttöliittymäkomponentit. Komponenttilista on todella pitkä ja kattava: taulukoita, pudotusvalikkoja, tekstikenttiä ja painonappeja. JSF:n komponentit ovat älykkäitä monessakin mielessä. Esimerkiksi *h:dataTable* komponentti tulostaa lista-olion perusteella automaattisesti niin paljon rivejä kuin oliossa on arvoja. *h:commandButton* komponentille annetaan suoraan tapahtumankäsittelijäluokka ja lisäysmetodi. Tapahtumankäsittelyluokka on ilmoitettu

JSF:lle erillisessä faces-congi.xml tiedostossa (kts. Kuva 7). F eli JSF:n Core –tagi sisältää Action-tageja, joiden avulla voidaan tehdä erilaisia toimintoja serverillä. F-tagin sisältä löytyy mm. erilaisia komponentteja tiedon validointiin ja konvertointiin, sekä käyttäjän klikkausten kuunteluun. Core –tagikirjastolla voidaan siis varmistaa tiedon oikeellisuus ja konvertoida se halutuksi jo sovelluksen näkymässä. (JavaBeat Website, 2009) Kaikki JSF komponentit tulee sijoittaa *f:view* tagin sisälle ja koska JSF perustuu tietojen lähettämiseen lomakkeen tietona, kaikkien komponenttien tulee olla *h:form* lomakkeen sisällä. (Vesterholm & Kyppö 2008, 571.)

```
13              <h:outputText value="Lisää uusi arvo"/>
14              <h:inputText value="#{olio.uusiArvo}" >
15                  <f:validateLongRange minimum = "10" maximum = "50"/>
16              </h:inputText>
```

KUVA 7. Validaattoreiden käyttö Facelet-sivulla

Kuvassa 5. on määritelty validaattorilla tekstikentälle maksimi-, ja minimimerkkien määrä *validateLongRange* komponentin avulla. Validaattorien ja konvertoijien käyttö ei rajoitu ainoastaan valmiisiin tageihin, omien validaattori-luokkien liittäminen käy myös helposti.(JavaBeat Website, 2009.)

Faces Servlet

JSF:n moottorina toimii Faces Servlet palvelinsovelma, joka käsittelee JSF pyynnöt ja alustaa JSF:n tarvitsemat resurssit. Faces Servlet määritellään web.xml tiedostossa.(Vesterholm & Kyppö 2008, 570.) Web.xml tulee olla jokaisen JSF sovelluksen WEB-INF hakemistossa. Web.xml tiedostossa määritellään se, että Faces Servlet on vastuussa JSF sovelluksen pyyntöjen käsittelystä. Faces Servlet toimii JSF sovelluksen eräänlaisena pääohjaimena (Central Controller). (Lars Voge, 2010.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

KUVA 8. Faces Servletin määrittelyt web.xml tiedostossa

Kuvassa 6 on kuvattu miten web.xml tiedoston avulla voidaan sovelluspalvelimelle ilmoittaa mistä Faces Servlet löytyy, millä polulla tulevia pyyntöjä Faces Servlet käsittelee, miten kauan sessio on voimassa ja mikä tiedosto toimii tervetulosivuna.

Tapahtumankäsittely

Varsinainen tapahtumankäsittely JSF sovelluskehyksessä tehdään erillisissä Backing Bean luokissa. Tapahtumankäsittelyluokat ilmoitetaan JSF:lle erillisessä faces-config.xml tiedostossa, joka on samassa WEB-INF hakemistossa web.xml tiedoston kanssa.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
    <managed-bean>
    <managed-bean-name>BackingBean</managed-bean-name>
        <managed-bean-class>oppari.BackingBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
</faces-config>
```

KUVA 9. faces-config.xml määrittelytiedosto

Yläpuolen koodiesimerkissä ilmoitetaan JSF:lle mistä tapahtumankäsittelyluokka löytyy, miten siihen viitataan Facelet-sivuilla ja mikä on tapahtumankäsittelijän näkyvyys. Näkyvyys voi olla istuntokohtainen(session), sovelluskohtainen(application) tai pyyntökohtainen(request). Sovelluksen viitaessa ensimmäisen kerran BackingBean:iin Facelets sivulla, luo JSF BackingBean olion ja sitoo sen näkyvyysalueen mukaan joko pyyntö-, sovellus-, tai istuntotason olioksi. (Vesterholm & Kyppö 2008, 572.)

```
1   //liitetään painonapin kuuntelijaluokka
2   import javax.faces.event.ActionEvent;
3   public class BackingBean {
4       //konstruktori
5       public BackingBean() {}
6       //ilmentymä DAOLuokasta, jossa on varsinaiset tietokantatoiminnot
7       private DAOLuokka DAOLuokka = new DAOLuokka();
8       //ilmentymä jäsenluokasta
9       private Luokka olio = new Luokka();
10
11      public void lisaaArvo(ActionEvent event){
12              DAOLuokka.lisaaArvo(olio);
13      }
14  }//class
```

KUVA 10. Tapahtumankäsittelyluokka BackingBean

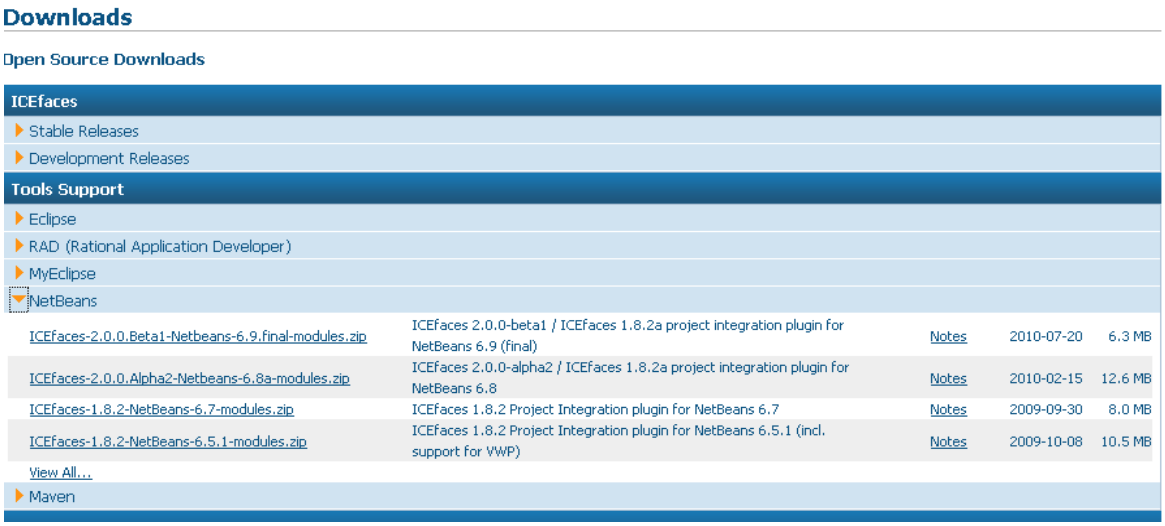
JSF:stä löytyy tapahtumien kuunteluun ja käsittelyyn tarvittavat luokat. Yläpuolen kuvassa 8 on käytetty ActionEvent oliota, jolla voidaan kuunnella painonappien painalluksia Facelet sivuilla. MVC-mallin mukaisesti varsinaiset tietokantahaut ja käsittelyt eivät ole samassa luokassa kuin tapahtumakäsittelijät, vaan niille on omat DAO-luokat.

3.3 ICEFaces

ICEFaces on JSF:n tavoin käyttöliittymien sovelluskehys. ICEFaces onkin itse asiassa eräänlainen AJAX –laajennus(Asynchronous JavaScript And XML) JSF:ään. JSF-käyttöliittymäkehys luo pohjan jokaiselle Icefaces ohjelmalle. ICEFaces-sivu on muodostettu JSF-komponenttipuusta ja kaikki JSF:n standardit työkalut, kuten validointi, konversio ja tapahtumankäsittelijät ovat tuettuina. ICEFaces toteuttaa myös JSF:n elinkaaren. ICEFaces mahdollistaa AJAX-ohjelman teon siten, että kehittäjän ei tarvitse kirjoittaa ainoatakaan riviä Javascript-koodia. (ICEsoft Technologies Inc 2009, 5.)

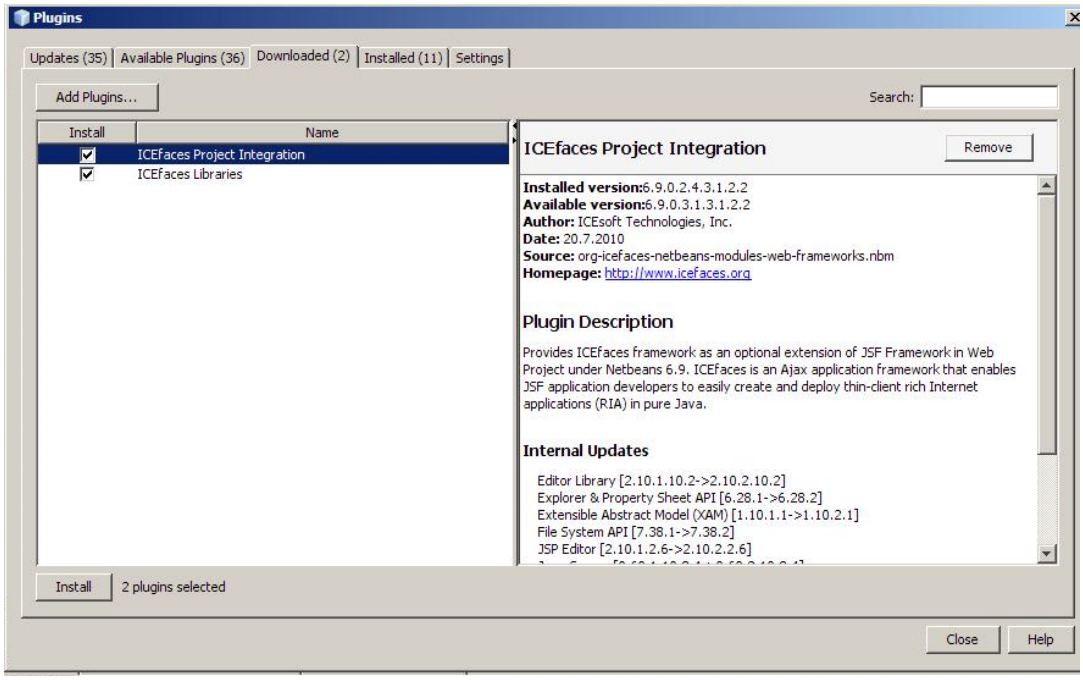
Käyttöönotto

Icefacesin käyttöönotto on tehty myös kehittäjän kannalta helpoksi. Suosituimpiin kehitysvälineiden (Netbeans, Eclipse) eri versioihin on omat integraatio-pluginit, joiden avulla käyttöönotto on helppoa.



KUVA 11. Ladattavat integraatipluginit ICEFacesin sivuilla

Icefaces:n pluginit ja kirjastot löytyvät osoitteesta: http://www.icefaces.org/main/downloads. Kun on ladannut ja purkanut pluginit(integraatio ja kirjastotiedot ovat erikseen), se on helppo asentaa esimerkiksi Netbeansissä Plugins valikon Downloaded välilehdeltä(kts. kuva 12).



KUVA 12. ICEFaces integraatio-pluginien asennus Netbeans sovelluskehittimessä

Arkkitehtuuri

ICEFaces tarjoaa kevyen Ajax sillan serverin ja clientin välille, jonka avulla voidaan vaihtaa sivun sisältöä, lataamatta koko sivua uudestaan. ICEFacesin arkkitehtuuri mahdollistaa myös sen, että sivun muutoksien tallentuminen ei katkaise käyttäjän meneillään olevaa työskentelyä/liikkumista sivulla. JSF sisältää myös oman f-*ajax* Ajax-komponentin, mutta ICEFacesin komponenteilla sivun ajax-toiminnot saadaan automaattisiksi ja läpinäkyviksi. Kun ICEFaces liitetään JSF-sovellukseen, JSF:n ja ICEFacesin komponentit toimivat automaattisesti Ajaxilla, ilman että JSF:n tapaan pitäisi ilmoittaa sovellukselle niitä erikseen *ajax* tagilla. (ICEsoft Technologies Inc 2010.)

Konfigurointi

Versiosta 2 lähtien ICEFaces ei enää vaadi omia servlettejä määriteltäväksi web.xml tiedostossa, ainoastaan JSF:n Faces Servlet tulee olla määritelty oikein.
Kuitenkin web.xml tiedostossa voidaan ICEFacesiin tehdä useita erilaisia konfiguroinnteja, joita osa komponenteista tarvitsee toimiakseen kunnolla. Eri komponettien tarvitsemat konfiguroinnit löytyvät ICEFacesin dokumentaatiosta kätevästi, mutta esimeksi ICEFacesin omat CSS-tyylit saadaan käyttöön kertomalla web.xml:ssä mistä Resource Servlet löytyy(kts. kuva 11). (ICEsoft Technologies Inc 2010.)

```
46      <servlet>
47          <servlet-name>Resource Servlet</servlet-name>
48          <servlet-class>com.icesoft.faces.webapp.CompatResourceServlet</servlet-class>
49          <load-on-startup>1</load-on-startup>
50      </servlet>
51      <servlet-mapping>
52          <servlet-name>Resource Servlet</servlet-name>
53          <url-pattern>/xmlhttp/*</url-pattern>
54      </servlet-mapping>
```

KUVA 13. Resource Servletin määrittely web.xml tiedostossa

Ensiksi sovelluspalvelimelle kerrotaan mistä kyseinen luokka löytyy, miten siihen viitataan ja avataanko se sovelluksen käynnistyessä. *servlet-mapping* tagilla ilmoitetaan missä polussa kyseinen serveltti toimii. Komponenteista esimeksi *ice:gMap* eli ICEFacesin oma sovellus Googlen karttapalvelusta tarvitsee omat konfiguraatiot web.xml tiedostoon. (ICEsoft Technologies Inc 2010)

Komponentit

ICEFaces sisältää tärkeimmät JSF:n komponentit Ajax-versioina, sekä muutamia omia komponentteja. Jokaiselle ICEFaces Facelet-sivuille tulee olla kuvan 11. tapaan JSF:n ja ICEFacesin komponentit liitettynä.

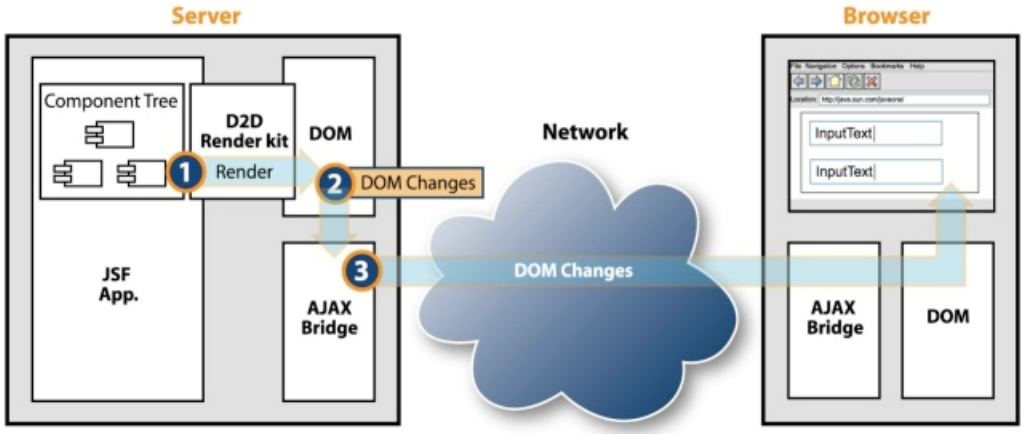
```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3  <html xmlns="http://www.w3.org/1999/xhtml"
4        xmlns:f="http://java.sun.com/jsf/core"
5        xmlns:h="http://java.sun.com/jsf/html"
6        xmlns:ice="http://www.icesoft.com/icefaces/component"
7          <h:body>
```

KUVA 14. ICEFacesin komponenttikirjaston liittäminen Facelet-sivulle

ICEFaces tuo JSF Facelet-sivuille uuden komponenttikirjaston *ice*, josta löytyvät kaikki ICEFacesin ajax-komponentit. ICEFaces-komponentit tulee myös sijoittaa JSF:n tapaan *f:view* tagin sisälle. ICEFaces sisältää oman lomake elementin *ice:form*, jonka sisälle tulee elementit olla sijoitettuna. (ICEsoft Technologies Inc 2009, 5.)

Direct-to-dom renderöinti

Direct-to-Dom renderöinnillä(D2D) tarkoitetaan ICEFacesin tarjoamaa tekniikkaa, jolla voidaan muuntaa JSF:n komponenttipuu suoraan W3C:n standardin mukaiseksi DOM (Domain object model) XML-tietorakenteeksi.



KUVA 15. ICEFaces Direct-to-dom renderöinti. (ICEsoft Technologies Inc 2009, 7)

Kun käyttäjä tekee HTTP-pyynnön sovelluksessa, hakee D2D renderöinti DOM-rakenteeseen muutoksia vain niihin komponentteihin, jotka ovat muuttuneet edellisestä vastauksesta(kts. kuva 15). Tällä tavoin ICEFaces vähentää huomattavasti sovelluskehittäjän työtä, koska enää ei tarvitse ilmoittaa sovellukselle JSF:n tavoin erikseen mitä kenttiä tulee päivittää, vaan ICEFaces hoitaa sen automaattisesti. (ICEsoft Technologies Inc 2009, 7-9.)

4 HIBERNATE

Hibernate on Red Hat:n kehittämä sovelluskehys, jolla saadaan automatisoitua oliomallin tallennus relaatiomalliksi Java-sovellustuotannossa. Hibernate on yleisesti käytetty ORM tekniikka Java sovelluskehityksessä. Hibernate on osa JBoss Java EE ohjelmistopakettia. Hibernate asettuu relaatiotietokannan ja sovelluslogiikan väliin ja se tarjoaa valmiit työkalut olioiden ominaisuuksien tallennukseen ja hakuun tietokannasta, sekä pitää huolen tietoturvasta, tiedon validoinnista ja eheydestä. Hibernate voikin vähentää merkittävästi työaikaa, mikä on aiemmin mennyt datan käsittelyyn SQL:llä ja JDBC:llä ohjelmoitaessa. Hibernaten kehittäjät kertovat tavoitteekseen vähentää 95 prosenttia tavallisia olioiden pysyvyyteen liittyviä ohjelmointitehtäviä. Hibernate sisältää mallinnukset kaikille suosituimmille tietokantojen hallintajärjestelmille(MySQL, MSSQL, PostGreSQL, Oracle jne.). (King ym. 2010, xi.)

Hibernate käyttöönotto

Hibernate on JSF:n tapaan sisäänrakennettu suosituimpiin kehitysvälineisiin(Netbeans, Eclipse). Hibernate ei tarvitse kehitysvälineitä toimiakseen, vaan sitä voidaan käyttää yhtä hyvin JDK paketin ja tietokannan kanssa vaikka tavallisella tekstieditorilla. Kehitysvälineiden käyttö mahdollistaa sen sijaan konfiguraatiotiedostojen tynkien luomisen, syntaksin tarkistuksen, sekä tarvittavien kirjastojen tuomisen projektiin. Luvussa 3 on neuvottu miten JSF tuodaan Netbeans projektiin. Hibernate valitaan samalla tapaa projektiin liittettävien frameworkien listasta(kts. Kuva 4) ja Hibernaten osalta löytyy myös valmiit tiedostopohjat Netbeansin uuden tiedoston lisäyksessä(kts kuva 5.)

Konfiguraatio

Hibernate voidaan konfiguroida tietokannan kanssa usealla eri tavalla, mutta helpoimmat ja suosituimmat tavat on käyttää hibernate.properties tai hibernate.cfg.xml määrittelytiedostoa dokumentin juuressa. Konfiguraatiotiedostoissa ilmoitetaan Hibernatelle ainakin käytettävä tietokanta, tietokannan murre, JDBC-ajuri, tietokannan osoite, tietokannan tunnukset ja missä luokkien mallinnustiedostot sijaitsevat. XML-tiedoston käyttäminen on suotavinta sen esitystavan selkeyden ansiosta.(King ym. 2010, 31-33.)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
4  <hibernate-configuration>
5    <session-factory>
6      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
7      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
8      <property name="hibernate.connection.url">jdbc:mysql://localhost/oppari</property>
9      <property name="hibernate.connection.username">root</property>
10     <property name="hibernate.connection.password">kissakala</property>
11     <!-- Mapping files -->
12     <mapping resource="oppari/Luokka.hbm.xml"/>
13   </session-factory>
14 </hibernate-configuration>
15
```

KUVA 16. Hibernaten XML konfiguraatiotiedosto

Yläpuolen kuvassa ovat ainoastaan ne pakolliset määrittelyt, jotka Hibernate vaatii toimiakseen. Riveillä 6-10 määritellään käytettävän SQL-kielen murre, ilmoitetaan käytettävän tietokannan ajuri, tietokantayhteyden osoite sekä tunnukset ja salasanat. Rivullä 12 olevalla *mapping* elementillä ilmoitetaan Hibernatelle kaikkien luokkakohtaiset mallinnustiedostojen sijainti. Hibernaten asetuksia on satoja, mutta kehittäjän onneksi sovelluskehyksen oletusarvoilla pääsee jo melko pitkälle.
(Silander ym. 2010, 464-465.)

Kohdealueen luokat

Kohdealueen luokissa on sovelluksen instanssimuuttujat, asetus- ja saantimetodit(”getter ja setter”) sekä parametrillinen ja parametriton konstruktori. Asetus ja saantimetodit sekä parametriton konstruktori ovat Hibernaten asettamia vaatimuksia kohdealueen luokalle. Jokaista Hibernaten käsittelemää instassimuuttujaa kohden tulee olla omat getterit ja setterit, pois jättäminen johtaa ajonaikaiseen poikkeukseen. Kohdealueen luokat ovat tavallisia POJO-luokkia(Plain old java object), jotka eivät toteuta mitään rajapintoja tai luokkia. Luokka sisältää siis ainoastaan jäsenmuuttujat ja niiden asetus-, ja saantimetodit(kts kuva 17). (King ym. 2010, 4.)

```java
public class Luokka {
    //Hibernaten vaatima oletuskonstruktori
    public Luokka(){}
    //Jäsenmuuttujat
    private Long id;
    private String arvo;

    public String getArvo() {
        return arvo;
    }
    public void setArvo(String arvo) {
        this.arvo = arvo;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
}//class
```

KUVA 17. POJO-luokka

Luokkakohtaiset mallinnustiedostot

Mallinnus Java luokkien ja tietokantataulujen välillä hoidetaan XML mallinnustiedostossa tai käyttämällä Javan Annotaatioita. XML tiedostoa käytettäessä jokaisen tietokannan taulu kuvataan erillisissä XML-tiedostoissa. Mallinnustiedostot ovat metadataa, jolla kerrotaan kohdealueen luokat, niiden väliset assosiaatiot ja luokkahierarkia, sekä miten ne liitetään tietokantaan. Alapuolen kuvassa on määritelty mallinnustiedoston rakenne yksinkertaisella esimerkkiluokalla. (King ym. 2010, 4.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class catalog="cruddemo" name="oppari.Luokka" table="oppari">
    <id name="id" type="long">
      <column name="id"/>
      <generator class="identity"/>
    </id>
    <property name="arvo" type="text">
      <column name="arvo" not-null="true"/>
    </property>
  </class>
</hibernate-mapping>
```

KUVA 18. Luokkakohtainen Hibernate-mallinnustiedosto

SessionFactory ja Session rajapinnat

SessionFactory rajapinta pitää nimensä mukaisesti huolta Hibernate sessioiden luomisesta. SessionFactory:stä luodaan yksi ilmentymä tietokantaa kohden sovelluksen lataamisen yhteydessä. SessionFactoryn luontia varten tehdään usein oma apuluokka, joka hoitaa SessionFactoryn luonnin. Netbeans-sovelluskehittimestä löytyy valmis tiedostopohja tämän luokan luontiin.(kts. Kuva 12.)

```
1  import org.hibernate.cfg.AnnotationConfiguration;
2  import org.hibernate.SessionFactory;
3
4  public class HibernateUtil {
5      private static final SessionFactory sessionFactory;
6
7      static {
8          try {
9              // tekee SessionFactory olion hibernate.cfg.xml konfiguraatiotiedoston pohjalta
10             sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
11         } catch (Throwable ex) {
12             // Kirjoittaa poikkeuksen logiin
13             System.err.println("Initial SessionFactory creation failed." + ex);
14             throw new ExceptionInInitializerError(ex);
15         }
16     }
17     //saantimetodi oliolle
18     public static SessionFactory getSessionFactory() {
19         return sessionFactory;
20     }
21 }
22
```

KUVA 19. HibernateUtil-apuluokka sessionFactory-olion hallintaan

Hibernate Session on rajapinta, jolla Hibernatessa käsitellään olioiden tallennus ja haku tietokannasta. Kun SessionFactory luodaan ainoastaan kerran tietokantaa kohden, Session rajapinnan ilmentymiä luodaan ja tuhotaan jatkuvasti sovelluksessa. Kaikki Hibernaten varsinaiset tietokantatoiminnot sijoitetaan näiden sessioiden sisälle. (Silander ym. 2010, 470-471.)

```
1   //Hibernaten vaatimat luokat
2   import org.hibernate.Session;
3   import org.hibernate.Transaction;
4
5   public class DAOLuokka {
6
7       public void lisaaArvo(Luokka olio){
8       //Otetaan SessionFactory ilmentymä HibernateUtil luokasta
9       Session sess = HibernateUtil.getSessionFactory().openSession();
10      Transaction tx = null;
11            try{
12                //aloitetaan transaktio
13                tx = sess.beginTransaction();
14                //tallennetaan olio tietokantaan
15                sess.save(olio);
16                //hyväksytään transaktio
17                tx.commit();
18            }
19            catch (Exception e){
20                //perutaan transaktio
21                tx.rollback();
22            }
23            finally {
24                //suljetaan sessio
25                sess.close();
26            }//try
27        }
28  }//class
```

KUVA 20. Session rajapinnan käyttäminen

Yläpuolen kuvassa on käytetty session rajapintaa olion lisäyksessä tietovarastoon. Aluksi sessio avataan SessionFactory-pääluokasta, HibernateUtil-apuluokan avulla. Seuvaavaksi alustetaan ja poikkeusrakenteen sisällä aloitetaan transaktio. Hibernaten hienous olioiden tallennuksessa tietovarastoon tulee hyvin esille rivillä 15, jossa olio varsinaisesti tallennetaan Hibernaten Session-rajapinnan *save()*-metodin avulla tietokantaan. Hibernate osaa tallentaa suoraan olion tietokantaan mallinnuksien avulla, eikä erillisiä instanssimuuttujia tarvitse tallentaa erikseen. Mikäli transaktioi onnistuu, se hyväksytää *commit()*-metodilla, mutta mikäli tallennuksessa on tapahtunut virhe, se perutaan *rollback()*-metodilla. Kävi kummin tahansa, lopuksi istunto suljetaan *close()*-metodilla.

4.1 Hibernate ja kyselykielet

Hibernate sovelluskehyksen avulla voidaan hakea ja päivittää tietokannan rivejä useilla eri tavoilla. Hibernate tukee myös tavallisia SQL-lauseita, kuten alla kuvattua kaikkien henkilot_taulu taulun tietojen hakua.

SELECT * FROM taulun_nimi;

SQL kyselyn tuloksena saadaan lista kohdealueen olioita. Hibernaten avulla voidaan muuntaa tietokantahaun tuloksen suoraan kohdealueen olioiksi, mikä nopeuttaa tulok-

sien käsittelyä huomattavasti perinteiseen JDBC:llä ohjelmointiin verrattuna. SQL suoritetaan ohjelmassa session olion *createSQLQuery()*-metodilla, jonka parametriksi annetaan SQL-kysely. Metodin paluuarvo on *SQLQuery*-tyyppiä,
(Silander ym. 2010, 470-471.)

Hibernaten kanssa on mahdollista käyttää tavallisen SQL:n lisäksi Hibernaten omaa kyselykieltä HQL:ää (Hibernate Query Language) tietojen hakuun tietokannasta. HQL on syntaksiltaan hyvin paljon SQL:n kaltainen, mutta HQL:ssä haut ja kyselyt eivät kohdistu SQL:n tapaan tietokannan riveihin, vaan sovelluksen olioihin. Hibernate muuttaa käyttäjän tekemistä HQL kyselyistä käytössä olevan tietokannan mukaisia hakuja. Koska HQL on oliopohjainen, ymmärtää se myös muita olio-ohjelmoinnin käsitteitä (periytyvyys, muunneltavuus ja assosiaatio). (King ym. 2010, 201.)

Yksinkertaisin kaikkien olioiden haku halutusta luokasta on seuraavanlainen:

from Luokka

HQL:n ainoa pakollinen osa on *from*-osa, joka määrittää minkä luokan ilmentymä haetaan. Tämä palauttaa kaikki *HenkiloLuokka*-luokan oliot, sekä kaikki sen alaluokkien oliot. (Silander ym. 2010, 471-472.)

Dynaamisiin kyselyihin Hibernate tarjoaa erillisen Criteria Api:n. Criteria API on rajapinta, jonka avulla voidaan käsitellä pysyviä luokkia kätevästi rajapinnan tarjoamien metodien avulla. Criteria Api on täysin oliopohjainen kyselyiden toteuttamisen malli. Criterian käyttäminen vaatii melko paljon tutustumista, jotta sen generoimien SQL-lauseiden optimoiminen onnistuu. Criterian parhaat käyttömahdollisuudet ovat monimutkaisien hakusivujen kanssa, jolloin eri rajaukset on helppo toteuttaa lisäämällä Criterian metodeja hakuperusteiden mukaan. (Ferguson Smart John, 2008.)

4.2 Assosiaatioiden mallintaminen

Olennainen hyöty Hibernate sovelluskehyksen käytöstä on helppo tapa kuvata luokkien välisiä assosiaatioita luokkakohtaisissa mallinnustiedostoissa Luokkien välisten assosiaatioiden(ts. suhteiden) mallintaminen on usein hankalimpia vaiheita sovellus-

tuotannossa. Assosiaatioita voi käytännössä olla kolmea eri typpiä: yhden suhde yhteen (1-1), yhden suhde moneen tai monen suhde moneen.

Luokkien väliset assosiaatiot kuvataan luokkakohtaisissa mallinnustiedostoissa (kts. kuva 20).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class catalog="oppari" name="oppari.Luokka" table="oppari">
    <id name="id" type="long">
      <column name="id"/>
      <generator class="identity"/>
    </id>
    <property name="arvo" type="text">
      <column name="arvo" not-null="true"/>
    </property>
    <!-- monen suhde yhteen assosaatio -->
    <many-to-one name="viitattu_luokka" column="viite_avain" not-null="true">
  </class>
</hibernate-mapping>
```

KUVA 21. Assosiaatioiden mallinnus luokkakohtaisessa mallinnustiedostossa

Edellisessä koodiesimerkissä on esitetty monen suhde yhteen assosiaatio. *Many-to-one*-tagilla määritellään liitettävä luokka. *Name*-atribuutilla annetaan liitettävän luokan nimi, *column*-arvolla puolestaan liitettävän luokan tietokantataulun pääavaimen kentän nimi. Käytännössä Hibernate luo nyt automaattisesti viiteavaimen myös päätauluun, jotta liitos onnistuu tietokannassa. Kun viitatun luokan olioita halutaan liittää pääluokan tuloksiin, tulee kohdealueen luokassa olla saanti- ja asetus metodit viitattavalle oliolle kuvan 21. kaltaisesti.

```
1   public class Luokka {
2       //Hibernaten vaatima oletuskonstruktori
3       public Luokka(){}
4       //Jäsenmuuttujat
5       private Long id;
6       private String arvo;
7       //Viitatun luokan esittely
8       private ViitattuLuokka viitattuLuokka;
9       public String getArvo() {
10          return arvo;
11      }
12      public void setArvo(String arvo) {
13          this.arvo = arvo;
14      }
15      public Long getId() {
16          return id;
17      }
18      public void setId(Long id) {
19          this.id = id;
20      }
21      //saanti ja asetusmetodit viitatulle luokalle
22      public ViitattuLuokka getViitattuLuokka() {
23          return viitattuLuokka;
24      }
25      public void setViitattuLuokka(ViitattuLuokka viitattuLuokka) {
26          this.viitattuLuokka = viitattuLuokka;
27      }
```

KUVA 22. Assosiaatioiden määrittely kohdealueen luokassa

4.3 Välimuisti

Välimuistin (Cache) avulla saadaan huomattavia etuja sovelluksien tietokantakäsittelyyn, esimerkiksi pelkän JDBC rajapintaan nähden. Välimuisti on paikallinen kopio tietokannan tilasta, joka sijaitsee palvelimen kiintolevyllä tai keskusmuistissa. Välimuistin käyttö vähentää olennaisesti tietokantahakuja, joilla on iso merkitys verkkosovellusten suorituskykyyn. Hibernate käyttää kolmea erilaista välimuistia olioille: ensimmäisen tason välimuistia(First Level Cache), toisen tason välimuistia(Second Level Cache), sekä kyselytason välimuisti (Query Level Cache). (Ferguson Smart John, 200.5)

Ensimmäisen tason välimuisti liitetään aina Session olioon, joten se on transaktiotason välimuisti(Transaction Scope Cache, TSC). TSC on sidottu yhteen työn yksikköön(unit of work), jolloin se on voimassa yhden työnyksikön suorittamisen ajan. Ensimmäisen tason välimuistissa oliot ladataan ja haetaan välimuistista ainoastaan yhden session sisällä. Hibernate käyttää oletuksena ensimmäisen tason välimuistia. .(Ferguson Smart John, 2005.)

Toisen tason välimuisti liitetään aina SessionFactory olioon, joten se taas on prosessitason välimuisti(Process Scope Cache, PSC). PSC on välimuistityyppi, jota voidaan käyttää monesta työn yksiköstä. Näin ollen välimuistiin ladataan olioita koko sovelluksen käyttöön, ei ainoastaan tietyn käyttäjän session käyttöön kuten ensimmäisen tason välimuistissa. Toisen tason välimuistilla saadaan siis vähennettyä tietokantakyselyitä enemmän kuin ensimmäisen tason välimuistissa, koska yhden käyttäjän aikaansaama paikallinen kopio on ladattavissa kaikkien muiden käyttäjien kesken. Toisen tason välimuisti toimii parhaiten tilanteessa, jolloin tietoa pääasiassa luetaan ja päivitetään harvoin. Jos tietoa päivitetään usein, toisen tason välimuisti tulee olla poissa käytöstä yhtäaikaisen päivityksen tuomien ongelmien takia. (Ferguson Smart John, 2005.)

Kyselytason välimuistin avulla voidaan laittaa välimuistiin olioiden sijasta varsinaisia tietokantakyselyiden tuloksia.

Välimuistitekniikka on monimutkaista ja markkinoilla on useita kaupallisia ja avoimia välimuistitoteutusta(Cache implementation). Hibernate tukee neljää erilaista avointa välimuistitoteutusta: EHCache (Easy Hibernate Cache), OSCache (Open Symphony Cache), SwarmCache ja JBoss TreeCache. (Ferguson Smart John, 2005.)

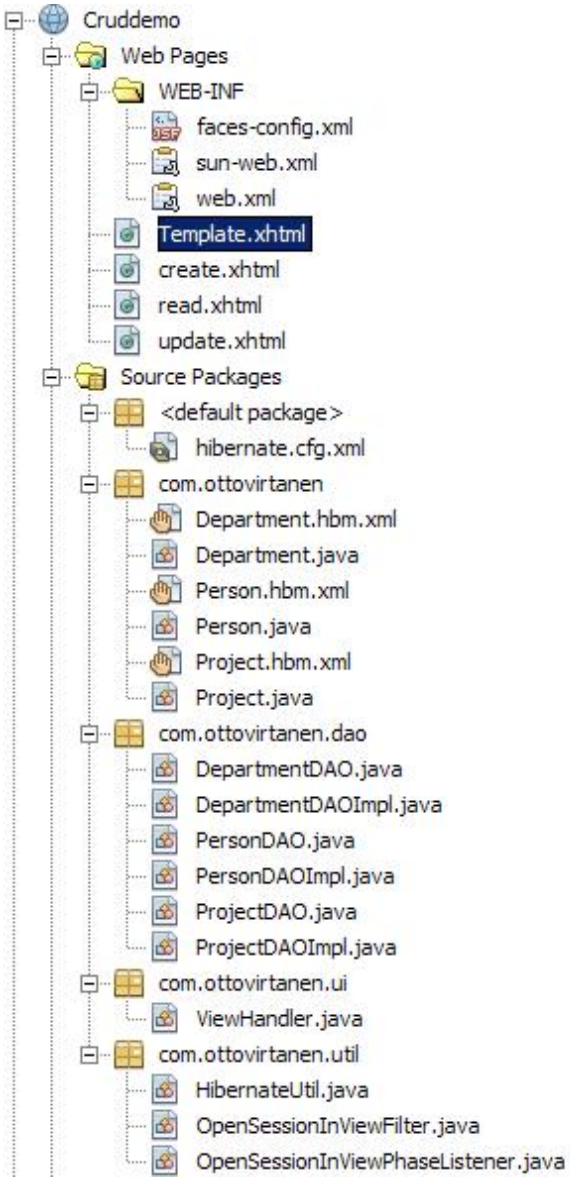
5 ESIMERKKITAPAUS

Esimerkkitapauksen toimeksiantajana toimi mikkeliläinen Bio-energialaitosten tuotannonohjausjärjestelmiin erikoistunut ohjelmistoyritys. Esimerkkitapauksen aihe oli toteuttaa demo-sovellus, jossa on käytetty Hibernate-sovelluskehystä ICEFaces-sovelluskehyksen kanssa. Demo-sovelluksen rinnalla syntyi myös tutoriaalityyppinen opas demo-sovelluksen tekoon vaihe vaiheelta(kts. liite 1).

Demo-sovelluksesta päätimme tehdä yksinkertaisen projektinhallintatyökalun, jonka avulla voi ylläpitää projekteja, niihin kuuluvia henkilöitä ja osastoita. Käyttöliittymän ohjelmointiin valitsimme JSF:n päälle rakentuvan ICEFaces Ajax-sovelluskehyksen 2.0 version, koska ICEFacesin aiempi versio on yrityksen tuotteissa käytössä ja uuden version ominaisuuksista ja toiminnasta haluttiin käyttökokemusta. Varsinainen esimerkkitapauksen pääaihe oli selvittää, mitä hyötyjä Hibernate ORM-sovelluskehyksen käytöstä voisi olla yrityksen tuotteissa. Tietokannanhallintajärjestelmänä sovellukseen valittiin MySQL ja sovelluspalvelimeksi Glassfish 3, yrityksen toiveiden pohjalta. Seuraavassa on esitelty sovelluksen toimintaa ja ominaisuuksia pääpiirteittäin. Tarkempi kuvaus sovelluksesta ja ohjeet sen tekoon löytyvät liitteenä olevasta oppaasta.

5.1 Demo-sovelluksen ominaisuudet

Esimerkkitapauksen demo-sovellus koostuu CRUD(create,read,update,delete) – operaatioilla varustetuista Facelet-sivuista, JSF:n tapahtumankäsittelijäluokasta, DAO-rajapinnoista ja luokista, POJO-luokista, HibernatenUtil-apuluokasta ja JSF:n tapahtumankuuntelijoista, sekä Hibernaten, JSF:n konfiguraatio XML-tiedostoista(kts. kuva 21).



KUVA 23. Demo-sovelluksen hakemistorakenne

Demo-sovelluksessa on kolme kohdealueen luokkaa: Project, Department ja Person. Project-, ja Person-luokka ovat assosiaatiosuhteeltaan moneen-suhde-moneen, eli henkilöllä voi olla monta projektia ja yhteen projektiin voi osallistua monta henkilöä. Lisäksi Person ja Department sisältävät yhden-suhde-moneen assosiaation, eli jokainen henkilö voi olla vain yhdessä osastossa, mutta yhdellä osastolla voi olla useita henkilöitä.

Projektiin liittyvien henkilöiden liittäminen projektien haun yhteyteen on sovelluksessa toteutettu Set rakenteella seuraavasti:

```
6   public class Project implements Serializable{
7
8       private int projectId;
9       private String projectName;
10      private String projectCode;
11      private String projectDescription;
12      private Set persons = new HashSet(0);
13
```

KUVA 22. Assosiaatioiden määrittelyt POJO-luokassa

Project.java POJO-luokkaan on lisätty Set olio persons, sekä saanti-, ja asetusmetodit sille.

Project.hbm.xml luokkakohtaisessa mallinnustiedostossa on ilmoitettu assosiaation tyyppi, luokka ja olion tyyppi samoin kun se POJO-luokassa on ilmoitettu.

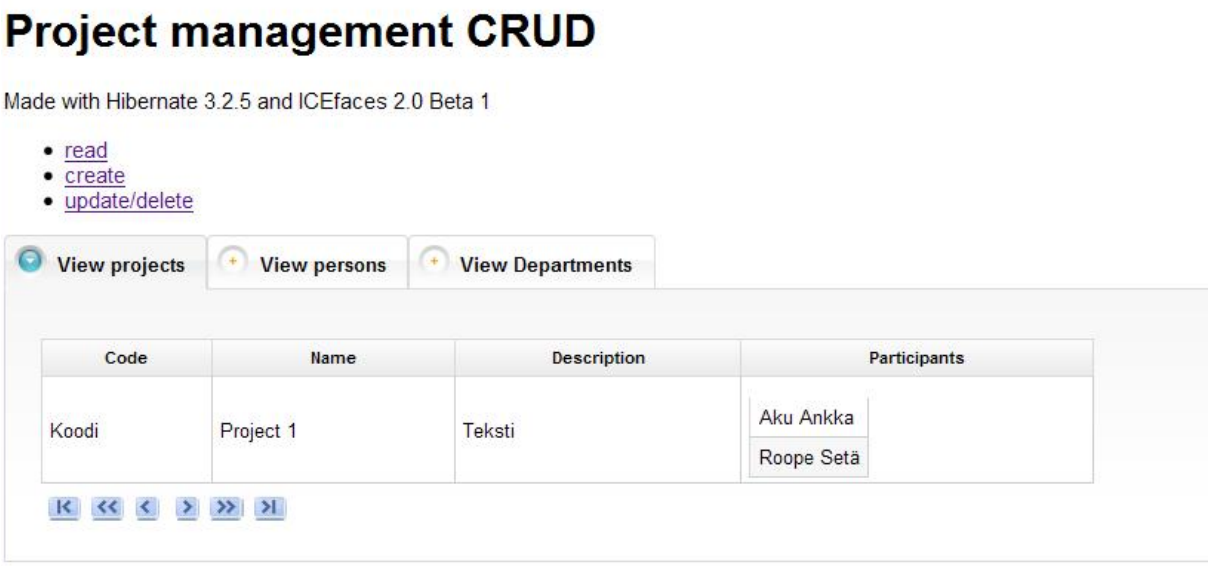
```
23      <set name="persons" table="project_person">
24        <key>
25          <column name="project_id" not-null="true"/>
26        </key>
27        <many-to-many entity-name="com.ottovirtanen.Person" order-by="person_id">
28          <column name="person_id" not-null="true"/>
29        </many-to-many>
30      </set>
```

KUVA 24. Assosiaatioiden määrittelyt mallinnustiedostossa

Pakettien juurihakemistosta löytyy Hibernaten konfiguraatiotiedosto, jossa on määritelty tietokantayhteydet ja ilmoitettu mistä luokkakohtaiset mallinnustiedostot löytyvät. Hakemistosta *com.ottovirtanen* löytyy kohdealueen POJO-luokat ja luokkakohtaiset mallinnustiedostot, joissa tehdään varsinainen mallinnus tietokannan rivien ja kohdealueen luokkien välillä. *com.ottovirtanen.dao* hakemistosta löytyy DAO-rajapinnat ja rajapinnan toteuttavat luokat, joista löytyvät varsinaiset tietokantaoperaatiot.

5.2 Demosovelluksen tiedosto MVC-mallissa

Sovelluksen näkymän(view) muodostavat Facelet-sivut on toteutettu mahdollisimman yksinkertaisesti, jotta demo-sovellus on helpompi toteuttaa tutoriaali-dokumentin pohjalta. Kuvassa 25 on sovelluksen ulkoasu Read–sivulla, joka tulostaa taulukkoon Hibernatella tietokannasta haetut projektit ja niihin osallistuvat henkilöt. Sivun tyylitiedostot on ICEFacesin tarjoamasta oletusteemasta.



KUVA 25. Demo-sovelluksen projektien listaussivu

Projektit tuodaan sivulle ICEFacesin *dataTable* –komponentin avulla. Komponenttiin liitetään lista-olio, jonka perusteella *dataTable* tulostaa taulukon rivejä tarvittavan määrän. Kuvassa 26. on esitelty miten *dataTable*-komponenttia käytetään Facelet-sivulla.

```
<ice:panelTabSet>
    <ice:panelTab label="View projects">
        <ice:panelGrid columns="1">
            <!-- Tässä tulostetaan listProject() metodin palauttama List -olio -->
            <ice:dataTable value="#{ViewHandler.listProjects()}"
            title="taulu" width="100%" rows="10" id="projectTable"
            var="item">
            <!-- Tässä annetaan taulukon sarakkeet, niiden arvot ja otsikot sarakkeille -->
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Code"/>
                    </f:facet>
                    <ice:outputText value="#{item.projectCode}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Name"/>
                    </f:facet>
                    <ice:outputText value="#{item.projectName}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Description"/>
                    </f:facet>
                    <ice:outputText  value="#{item.projectDescription}" />
                </ice:column>

                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Participants"/>
                    </f:facet>
                    <ice:dataTable value="#{item.persons}" var="persons" >
                        <ice:column>
                            <ice:outputText value="#{persons.firstName} #{persons.lastName}"/>
                        </ice:column>
                    </ice:dataTable>
        </ice:panelGrid>
    </ice:panelTab>
```

KUVA 26. Näkymän tulostus Facelet-sivulla

Sovelluksen ohjaimen(controller) toteuttavat tapahtumankäsittelijäluokka (ViewHandler) ja JSF:n tarjoama FacesServlet. Viewhandler:ssä on varsinainen tapahtumien käsittely ja FacesServlet hoitaa käytännössä automaattisesti käyttäjän pyyntöjen vastaanottamisen ja välittämisen tapahtumankäsittelijäluokalle. *listProjects()*-metodi tapahtumankäsittelijäluokassa kutsuu samannimistä metodia DAO-luokassa, jossa on toteutettu olioiden varsinaiset haut tietovarastosta Hibernaten tarjoamien työkalujen avulla.

```
217   // ------------PROJECT CRUD OPERATIONS ------------------>
218       public List listProjects(){
219           List projects;
220           projects = projectDAOImpl.listProjects();
221           currentProject.clear();
222           getProjectMenuItems();
223           return projects;
224       }
225       public void createProject(ActionEvent event){
226            projectDAOImpl.addProject(newProject);
227            getProjectMenuItems();
228            newProject.clear();
229       }
230       public void updateProject(ActionEvent event){
231            projectDAOImpl.updateProject(currentProject);
232            getProjectMenuItems();
233       }
234       public void deleteProject(ActionEvent event){
235            projectDAOImpl.deleteProject(currentProject);
236           //Clearing the currentProject values from input fields
237           currentProject.clear();
238           getProjectMenuItems();
239           getPersonMenuItems();
240       }
```

KUVA 27. Crud-operaatiot tapahtumankäsittelijäluokassa

Varsinaiset kohdealueenluokat(POJO-luokat) ja DAO-luokat huolehtivat sovelluksen mallista(model). Demo-esimerkin tietohaut toteutettiin pääasiassa Hibernaten HQL-kyselykielellä.

```
23        public List listProjects() {
24           List projects = null;
25           //Getting the sessionfactory from HibernateUtil.java
26           Session session = HibernateUtil.getSessionFactory().getCurrentSession();
27           try {
28               session.beginTransaction();
29               //creating a list object from db by HQL query
30               projects = session.createQuery("from Project order by projectId").list();
31               session.getTransaction().commit();
32           } catch (HibernateException e) {
33                session.getTransaction().rollback();
34            }
35           return projects;
36       }//listProjects
```

KUVA 28. Projektien haku tietovarastosta Hibernaten tarjoamien työkalujen avulla

Yläpuolen kuvassa haetaan aluksi SessionFactoryUtil-luokasta avoinna oleva Hibernate sessio. Seuraavaksi riveillä 28-31 aloitetaan transaktio, tehdään tietohaku

HQL:llä ja laitetaan tulos *list()*-metodilla listarakenteeseen, sekä hyväksytään transaktio mikäli haku on onnistunut. Lopuksi palautetaan *projects*-niminen listaolio tapahtumankäsittelijäluokalle.

MVC-mallia ja laiskaa latausta käytettäessä ei voida varsinaista sessiota avata ja sulkea mallissa, jotta sovellus saadaan toimimaan oikein. Sessio ehtii sulkeutua mallissa, kun näkymä yrittää vielä saada laiskalla latauksella liitetyt oliot mukaan hakutulokseen. Tähän yleinen ratkaisumalli on avata ja sulkea sessio näkymässä(Open session in View). Session avaaminen näkymässä toteutetaan erillisen HTTP-filtterin avulla, joka avaa session pyynnön alussa ja lopettaa sen aivan vastauksen lopussa.
JSF:n ja ICEFacesin kanssa tarvittiin vielä erillinen sovelluksen JSF:n tilan kuuntelija(phase listener), koska käyttäjän pyynnöt eivät tule aina pelkkinä HTTP-pyyntöinä.

6 PÄÄTÄNTÖ

Tutkimusongelmani oli määrittää, voisiko Hibernate-sovelluskehyksestä olla merkittäviä etuja MHG Systems oy:n tuotekehityksessä, sekä miten Hibernate-sovelluskehys toimii yhdessä yrityksen käyttämän ICEFaces-ovelluskehyksen kanssa. Toimeksiantaja oli kiinnostunut saamaan käyttökokemusta Hibernaten toiminnasta ICEFaces-käyttöliittymäkomponenttien kanssa ja mahdollisista ongelmista mitä näiden kahden sovelluskehyksen yhteiskäytössä voisi ilmaantua. Työn edetessä tuli toimeksiantajalta myös ehdotus oppaan kirjoittamiseksi demo-sovellukselle, joka toimisi yrityksen sovelluskehittäjille tutoriaalina.

Esimerkkitapauksena toteutettu demo-sovellus onnistui mielestäni hyvin, vaikka kiireen takia sovellukseen jäi muutamia puutteita. Kuitenkin pääosiltaan sovellus ja tutoriaali toimivat hyvin tarkoituksessaan, eli kertovat kattavasti Hibernaten-, ja ICEFaces-sovelluskehyksien käytöstä MVC-mallin mukaisessa J2EE-sovellustuotannossa. Kehitettävää ja parannettavaa sovelluksesta löytyy muun muassa tapahtumankäsittelyssä. Esimerkiksi varsinaisia JSF:n tarjoamia validaattoreita tai kovertoijia ei ole käytetty lainkaan. Tämä oli tietoinen valinta, koska demo-sovellus haluttiin pitää mahdollisimman helposti lähestyttävänä ja toteutettavana oppaan pohjalta. Liian monet ominaisuudet eivät helpottaisi muutenkin haastavien työkalujen opettelua. Demo-sovellus ja tutoriaali toimivat sellaisenaan hyvänä oppimateriaalinen edellä mainittujen tekniikoiden opiskeluun.

Tutkimusongelman vastaukseksi opinnäytetyötä tehdessäni selvisi, että Hibernaten avulla voidaan säästää huomattavasti aikaa verrattuna perinteiseen ORM-mallin tekoon käsin JDBC:llä. Hibernaten avulla sovelluksen mallista saadaan myös helpommin hallittava ja selkeämpi. Kehyksen käyttö myös vähentää huomattavasti olioiden tallennukseen tarvittavan koodin määrää sovelluksessa. Hibernate-sovelluskehyksen kattava hallitseminen vaatii melko paljon opiskelua, mutta toisaalta kaikki opiskelu tuo varmasti käytetyn ajan takaisin, koska sovelluksen mallin ohjelmoiminen nopeutuu huomattavan paljon verrattuna perinteisiin menetelmiin. Aivan pieniin sovelluksiin en Hibernaten käyttöä suosittele, mutta jos luokkia ja niiden suhteita on vähänkään enemmän, tuo Hibernate selkeyttä sovelluksen tietokantarakenteeseen huomattavan paljon. Hibernate tekee myös sovelluksesta helposti laajennettavan, koska uudet muutokset luokkarakenteeseen on helppo toteuttaa.

Osana tutkimusongelmaa oli myös selvittää, miten hyvin Hibernate-sovelluskehys toimii ICEFaces-sovelluskehyksen kanssa. Käyttökokemuksieni perusteella pääosiltaan Hibernate toimii JSF:n ja sen Ajax-laajennuksen ICEFaces:n kanssa moitteettomasti, koska ICEFaces tai JSF ei ota kantaa varsinaiseen olioiden tallennuksen toteutukseen, eikä Hibernate ota kantaa sovelluksen käyttöliittymään ollenkaan. Hibernate ja JSF(laajennuksineen) onkin mielestäni hyvä tapa toteuttaa nykypäivän Java EE-sovellus, koska sovelluskehykset eivät sisällä päällekkäisiä toiminnallisuuksia ollenkaan, jolloin sovelluskehyksillä on selkeät roolit sovelluksessa. Mikäli sovellus olisi tehty esimerkiksi Hibernate-, ja Spring-sovelluskehystä käyttämällä, roolien jako ei olisi niin selkeää, koska Spring sisältää käytännössä paljon samoja toiminnallisuuksia kuin Hibernate. MVC-malli toi kuitenkin mukanaan muutamia haasteita, toisaalta nämä haasteet olisi kohdattu luultavasti myös millä tahansa muulla kehyksellä Hibernaten kanssa ohjelmoitaessa.

Haasteita esimerkkitapauksen teossa riittikin loppuun asti. Aikaisempi vähäinen JAVA-ohjelmoinnin kokemus oli ehkä suurin syy hankaluuksien ilmaantumiseen. Opinnäytetyön teosta suuri aika menikin JAVA:n ja olio-ohjelmoinnin opiskeluun. Sovelluskehyksien oikeaoppinen käyttäminen vaati myös useiden eri arkkitehtuurimallien opiskelua(MVC, DAO), joiden ymmärtämisessä meni aikansa, varsinkin kun olio-ohjelmointia en ollut juurikaan harrastanut ennen opinnäytetyötä. Hibernaten ja ICEFacesin yhteistoiminnassa tuli myös ongelmia (sessioiden avaaminen näkymässä(kts. luku 6.2).

Tulevaisuuden kannalta näen Hibernaten jatkavan de-facto-asemassa JAVA:n ORM-toteutusmallina, sen selkeän arkkitehtuurin, helppokäyttöisyyden ja laajan käyttäjäkunnan ansiosta. ICEFaces-sovelluskehyksen uskon kuitenkin JAVA:n käyttöliittymäsovelluskehyksien kentällä häviävän pikkuhiljaa pois, JSF:n ja Springin laajentaessaan komponentteja paremmin Ajaxia tukeviksi ja monipuolisemmiksi. Lisäksi markkinoilla on myös muita ilmaisia Ajax-komponenttikirjastoja ja sovelluskehyksiä todella liikaa, jotta ICEFaces:n kaltainen kehys saisi tarpeeksi ison yhteisön jatkossa ympärilleen. Mielenkiinnolla odotan myös miten HTML5 muuttaa JAVA EE-sovellusten käyttöliittymien toteutusta.

Halusin opinnäytetyöksi sellaisen aiheen, jota tehdessä tulisi opeteltua uusia tekniikoita kattavasti ja juuri parempaa aihetta en olisi siltä kannalta voinut saada. Kuitenkin

useiden tekniikoiden opettelu vei paljon aikaa ja toisaalta opiskelun määrä paisui opinnäytetyön mittakaavassa melko suureksi. Työnhakua ja jatkoa varten opinnäytetyön aikana opituista tekniikoista on varmasti hyötyä, joten panostaminen varmasti kannatti.

LÄHTEET

Eckstein Robert, 2010. Java SE Application Design With MVC. Oracle Technology Network. WWW-dokumentti. Luettu 5.11.2010.

Ferguson Smart, John 2008. Hibernate Querying 102 : Criteria API . WWW-dokumentti. Luettu 2010.2010. http://www.javalobby.org/articles/hibernatequery102/

Ferguson Smart John, 2005. Speed Up Your Hibernate Applications with Second-Level Caching. WWW-dokumentti. Luettu 7.11.2010. http://www.devx.com/dbzone/Article/29685

Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole 2010. Hibernate Reference Documentation. JBoss Inc. WWW-dokumentti.

ICEsoft Technologies Inc 2009. ICEfaces Developer's Guide 1.8. ICEsoft Technologies Inc.

JDBC, 2010. Java Database Connectivity home page. Sun Microsystems Inc .WWW-dokumentti. Luettu: 9.7.2010. *http://java.sun.com/products/jdbc/index.jsp*.

Jendrock, Eric, Ball, Jennifer, Carson, Debbie, Evans, Ian, Fordin, Scott, Haase, Kim. The Java EE Tutorial. WWW-dokumentti. Luettu 20.9.2010. Päivitetty 6/2010. http://download.oracle.com/javaee/5/tutorial/doc/

JavaBeat Website, 2009. Introduction to JSF Core Tags Library WWW-dokumentti. Luettu 5.11.2010. http://www.javabeat.net/articles/15-introduction-to-jsf-core-tags-library-1.html

Kuha, Janne 2008. Tehokas Java EE-sovellustuotanto. Helsinki: Docendo.

Vorge Lars, 2010. JSF (JavaServer Faces) Tutorial. WWW-dokumentti. Luettu 6.11.2010. Päivitetty: 15.3.2010. http://www.vogella.de/articles/JavaServerFaces/article.html

Object Relational Mapping Strategies. WWW-dokumentti. Luettu 20.9.2010 http://www.objectmatter.com/vbsf/docs/maptool/ormapping.html.

Silander, Simo, Ollikainen, Vesa, Peltomäki, Juha 2010. Java. Helsinki: Docendo.

Vahtolammi, Kari 2010. MVC- Malli, peruskauraa frameworkkien käyttäjille. WWW-dokukentti. Luettu 5.10.2010. http://vahtolam.wordpress.com/2010/09/23/mvc-malli-peruskauraa-frameworkkien-kayttajille/

Vesterholm, Mika, Kyppö, Jorma 2008. Java-ohjelmointi. Helsinki: Talentum.

Wikipedia, 2010. Wikipedia, The Free Encyclopedia. Model-View-Controller. Luettu 3.11.2010. WWW-dokumentti, http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Wikipedia, 2010. Wikipedia, The Free Encyclopedia. Object Relational Mapping. Luettu: 11.7.2010. WWW-dokumentti. http://en.wikipedia.org/wiki/Object-relational_mapping

LIITE 2(1).

Monisivuinen liite

LIITE 2(2).

Monisivuinen liite

Otto Virtanen

HIBERNATE 3.2.5 JA ICEFACES 2.0 BETAN KÄYTTÖ NETBEANS YMPÄRISTÖSSÄ

Elokuu 2010

MIKKELIN AMMATTIKORKEAKOULU

SISÄLTÖ

1 JOHDANTO

Seuraavassa dokumentissa on esitelty Hibernate 3.2.5 ja Icefaces 2.0 beta sovelluskehyksien toimintaa Netbeans 6.9 ympäristössä. Demosovelluksena toimii projektien hallintaan perustuva web-sovellus, jolla voidaan hallita projekteja ja niihin kuuluvia henkilöitä. Tietokannanhallintajärjestelmänä sovelluksessa toimii MySQL.

Demo on esitetty kolmessa eri osassa, ensiksi suoritetaan ainoastaan yhdellä Project luokalla toimiva sovellus. Seuraavassa vaiheessa lisätään Person luokka, jonka avulla selostetaan monen-suhde-moneen tapaus Hibernate ympäristössä ja lopuksi lisätään vielä Department luokka, jolla kuvataan yhden-suhde-moneen. Tällä on pyritty pitämään Demosovellus helpommin ymmärrettävänä ja lähestyttävänä.

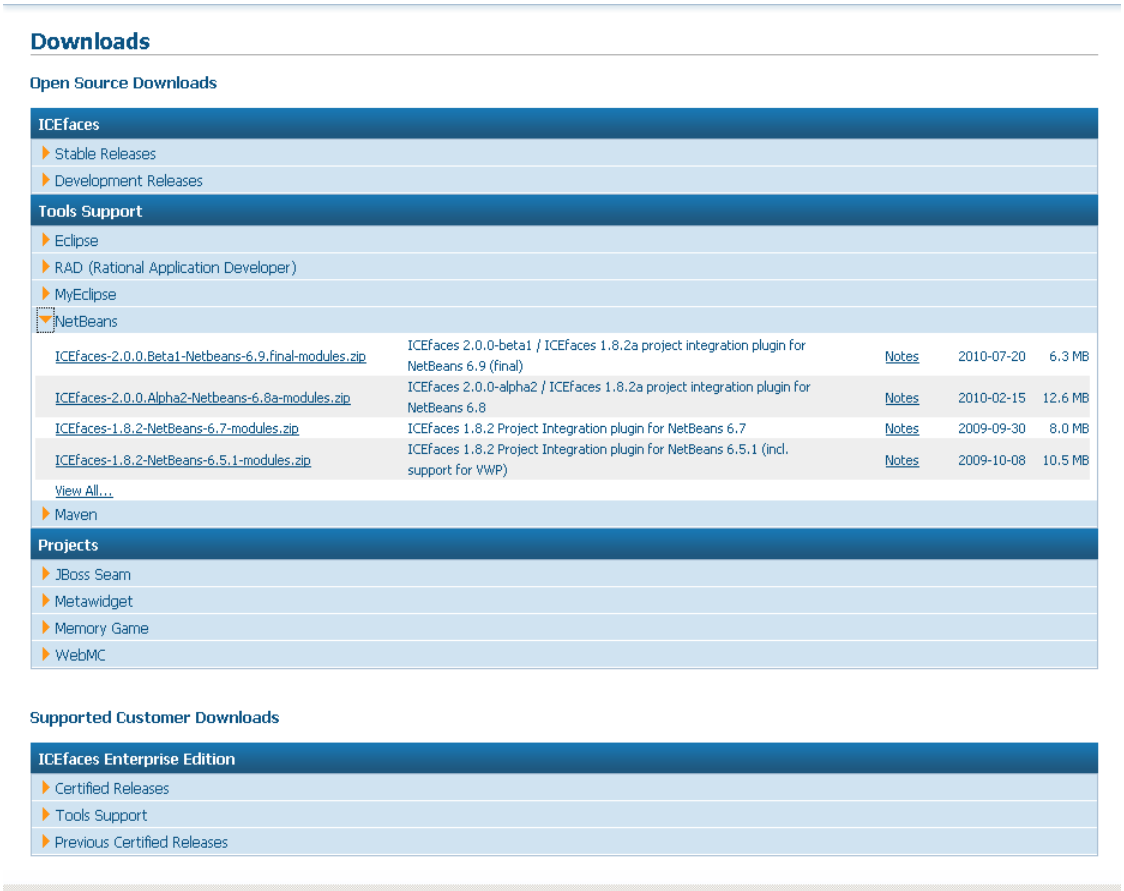
2 KÄYTTÖÖNOTTO

2.1 Netbeansin käyttöönotto

Netbeansin versiosta 6.8 asti Hibernate on ollut integroituna välineessä, joten Hibernate on suoraan kehittäjän käytettävissä. Mikäli käytössä on vanhempi versio kehitysvälineestä, tulee käyttäjän asentaa erillinen Hibernate –plugin Netbeansiin, mikäli haluaa kaikki kehitysvälineen toiminnot käyttöönsä. Toinen vaihtoehto on ladata molemmista sovelluskehyksistä uusimmat kirjastot ja liittää ne projektiin, tällöin ei voida käyttää kuitenkaan esim. valmiita tiedostopohjia.

2.2 Icefacesin käyttöönotto

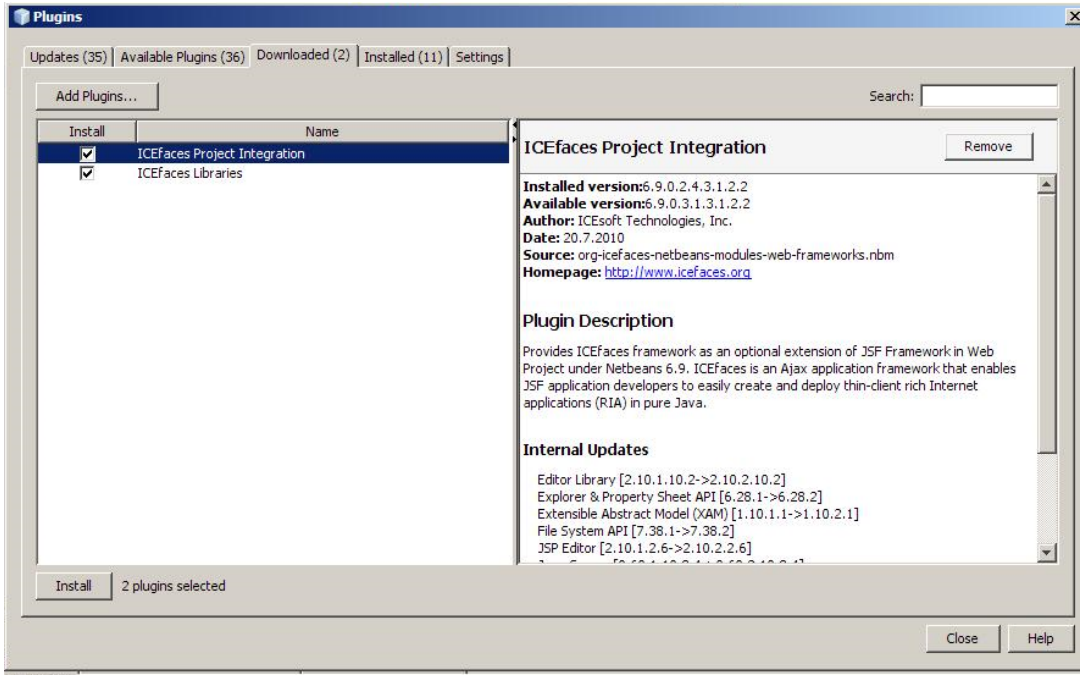
Icefacesin käyttöönotto on tehty myös kehittäjän kannalta helpoksi. Netbeansin ja Eclipsen eri versioihin on omat integraatio-pluginit, joiden avulla käyttöönotto on helppoa.



Icefaces:n pluginit löytyvät osoitteesta: http://www.icefaces.org/main/downloads. Ladattuasi ja purettuasi pluginit(integraatio ja kirjastotiedot ovat erikseen), se on helppo asentaa Netbeansissä Tool-Plugins valikon Downloaded välilehdeltä.

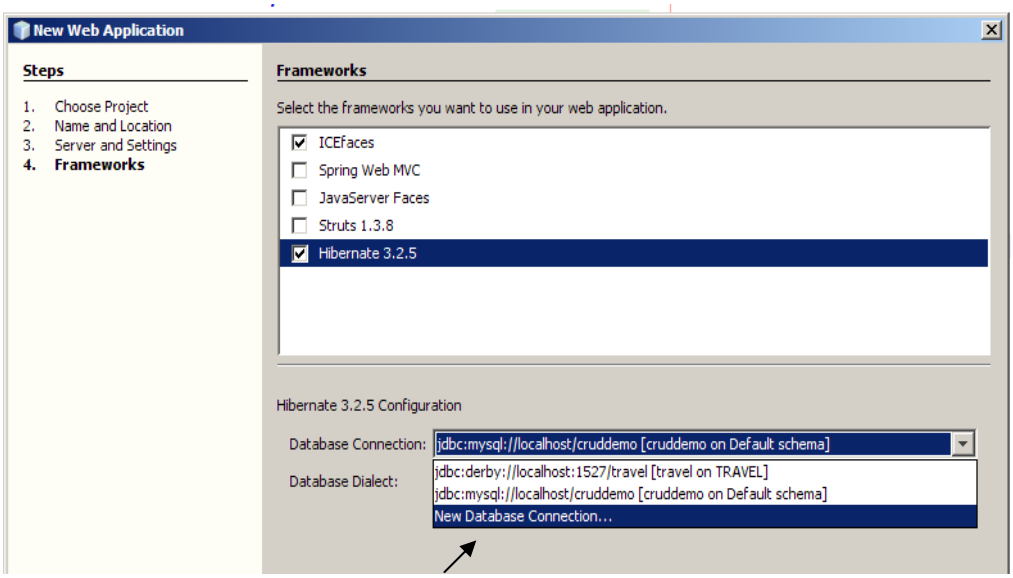
LIITE 2(3).

Monisivuinen liite

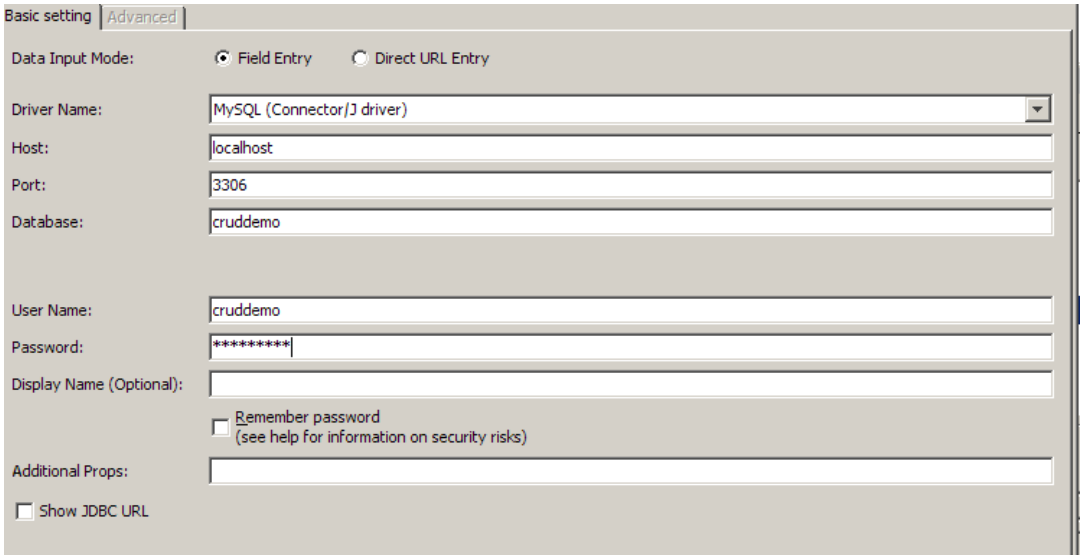


3 PROJEKTIN LUOMINEN

Aloitetaan demon tekeminen ajamalla sql.zip paketissa oleva SQL-scripti tietokantaan, tämä luo tarvittavat taulut valmiiksi. Seuraavaksi luodaan Java web projekti, johon sisällytetään ICEfaces ja Hibernate frameworkit.

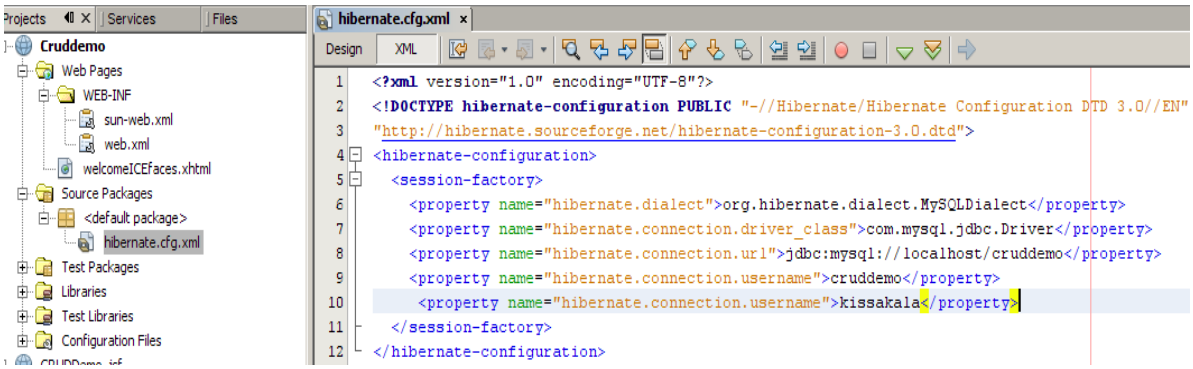


Tässä vaiheessa voidaan valita jo Hibernaten käyttämä tietokanta, jolloin framework tekee jo osan tarvittavista tietokanta-asetuksista hibernate.cfg.xml tiedostoon.



4 HIBERNATEN KONFIGUROINTI

Nyt kun Hibernate on liitetty projektiin, tulee luonnin yhteydessä Source Packages juureen hibernate.cfg.xml niminen Hibernaten konfiguraatiotiedosto. Tiedosto on suositeltavaa pitää juuressa, koska tällöin Hibernate löytää sen automaattisesti.



Kuten kuvasta huomaa, on XML-konfiguraatiotiedostoon ilmestynyt projektia luodessa tehdyt tietokanta-asetukset. Netbeansin työkaluissa on myös Design –näkymä, josta voi ”velhon” avulla muuttaa Hibernaten asetuksia.

Tässä on siis vasta tietokantayhteyteen tarvittavat määrittelyt ja Hibernaten konfigurointiin onkin satoja erilaisia asetuksia. Hibernateen on kuitenkin määritelty hyvät oletusarvot useimpiin asetuksiin. Muutamat asetukset on kuitenkin hyvä käydä läpi.

<property name="hibernate.show_sql">true/</property>

hibernate.show_sql:n ollessa käytössä, Hibernate kirjoittaa konsoliin kaikki muunnetut sql-lauseet. Tästä on hyötyä sovellusta kehittäessä, koska tällöin nähdään Hibernaten muodostamat varsinaiset SQL hakulauseet.

<property name="hibernate.hbm2ddl.auto">update</property>

Hbm2.dll:n avulla Hibernate voi luoda tietokannan schema automaattisesti luokkien mallinnustietodostojen pohjalta. Vaihtoehdot ovat:

- Create – tekee tietokannan, tuhoten aikaisemman datan
- Create-drop - pudottaa scheman session lopuksi ja luo uuden käännettäessä ohjelmaa.
- Update – päivittää schemaa luokkien mallinnustiedostojen mukaan.
- Validate – validoi scheman tekemättä muutoksia dataan.

<property name="hibernate.connection.pool_size"></property>

Connection poolilla asetetaan yhteyssäiliön koko. Oletusarvoisesti 20.

<property name="hibernate.cache.provider_class"> </property>

Hibernate.cache.provider_class asetuksella määritellään mitä välimuistitekniikkaa halutaan käyttää. Oletuksena Hibernate käyttää EHCachea välimuistin toteuttamiseen. Välimuistin käyttämisellä voidaan nopeuttaa sovelluksen suorituskykyä, tämän tehdessä paikallisen kopion tietokannan tilasta kiintolevylle tai keskusmuistiin. Välimuistin avulla vältetään turhat käynnit tietokantaan ja tiedot haetaan nopeammasta välimuistista.

`org.hibernate.cache.NoCacheProvider arvolla saadaan toisen tason välimuistin pois käytöstä.`

<property name="current_session_context_class">thread**</property>**

Current_session_context_class:n avulla kerrotaan Hibernatelle missä yhteydessä sessionFactory olio sidotaan. Arvon ollessa thread(oletus), sessionfactory:n pyynnöt sidotaan säikeisiin.

<mapping resource="com/ottovirtanen/Project.hbm.xml"/>

Konfiguraatiotiedostossa myös ilmoitetaan varsinaisten luokkien mallinnustiedostojen sijainti Hibernatelle.

Lopuksi Konfiguraatiotiedoston tulisi olla seuraavanlainen:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/cruddemo</property>
    <property name="hibernate.connection.username">KÄYTTÄJÄTUNNUS</property>
    <property name="hibernate.connection.username">SALASANA</property>
     <!-- Write all SQL statements to console -->
     <property name="hibernate.show_sql">true</property>
    <!--
       Automatically validates or exports schema to the database when Sessionfactory is created
       With create-drop , schema will be dropped when the SesionFactory is closed explicitly.
     -->
     <property name="hibernate.hbm2ddl.auto">update</property>
     <!-- Mapping files -->
    <mapping resource="com/ottovirtanen/Project.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

5 POJO LUOKAN LISÄÄMINEN

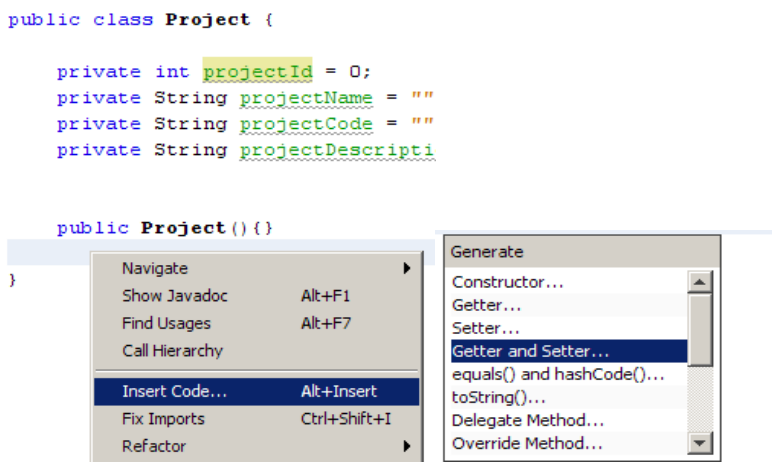
> HUOM!
> Netbeansin välineistä löytyy valmiita apuvälineitä POJO-luokkien ja mallinnustiedostojen tekoon tietokannan riveistä. Tästä on ohjeet luvussa 6.

Seuraavaksi esittelemme luokan joka edustaa projektia. Teemme yksinkertaisen Java-Bean luokan, jolla on parametritön konstruktori(vaatimus pysyville Hibernate luokille, jotta Reflection Api toimisi oikein).

Luodaan uusi Project.java luokka seuraavilla ominaisuuksilla:

```
package com.ottovirtanen;
public class Project {
   private int projectId = 0;
   private String projectName = "";
   private String projectCode = "";
   private String projectDescription = "";
   //Constructor
   public Project(){}
}
```

Netbeansissä on kätevä apuväline Getter ja Setter metodien luomista varten. Painamalla oikeaa hiirennäppäintä koodissa(tai alt+insert) saadaan Insert Code.. valikko, josta valitaan Getter and Setter.. valinta.



Tällä tavoin vältytään Getter ja Setter metodien kirjoittamiselta käsin, joka on melko työlästä ainakin silloin kun luokassa on paljon muuttujia.

Project.java tulisi olla nyt seuraavankaltainen:

```
public class Project {
   private int projectId = 0;
   private String projectName = "";
   private String projectCode = "";
   private String projectDescription = "";
   public Project(){}

   public String getProjectCode() {
      return projectCode;
   }
   public void setProjectCode(String projectCode) {
      this.projectCode = projectCode;
   }
   public String getProjectDescription() {
      return projectDescription;
   }
   public void setProjectDescription(String projectDescription) {
      this.projectDescription = projectDescription;
   }
   public int getProjectId() {
      return projectId;
   }
   public void setProjectId(int projectId) {
      this.projectId = projectId;
   }
   public String getProjectName() {
      return projectName;
   }
   public void setProjectName(String projectName) {
      this.projectName = projectName;
   }}
```

6 HIBERNATEN XML- MALLINNUSTIEDOSTOJEN LUONTI

Seuraavaksi luodaan mallinnustiedostot, joissa ilmoitetaan Hibernatelle tietokannan taulut ja rivit erillisessä XML-tiedostossa.
Luodaan uusi Hibernaten XML-tiedosto seuraavilla arvoilla samaan polkuun POJO-luokan kanssa.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class catalog="cruddemo" name="com.ottovirtanen.Project" table="project">
    <id name="projectId" type="integer">
      <column name="project_id"/>
      <generator class="identity"/>
    </id>
    <property name="projectName" type="string">
      <column name="project_name" not-null="true"/>
    </property>
    <property name="projectCode" type="string">
      <column name="project_code" not-null="true"/>
    </property>
    <property name="projectDescription" type="string">
      <column name="project_description" not-null="true"/>
    </property>
</hibernate-mapping>
```

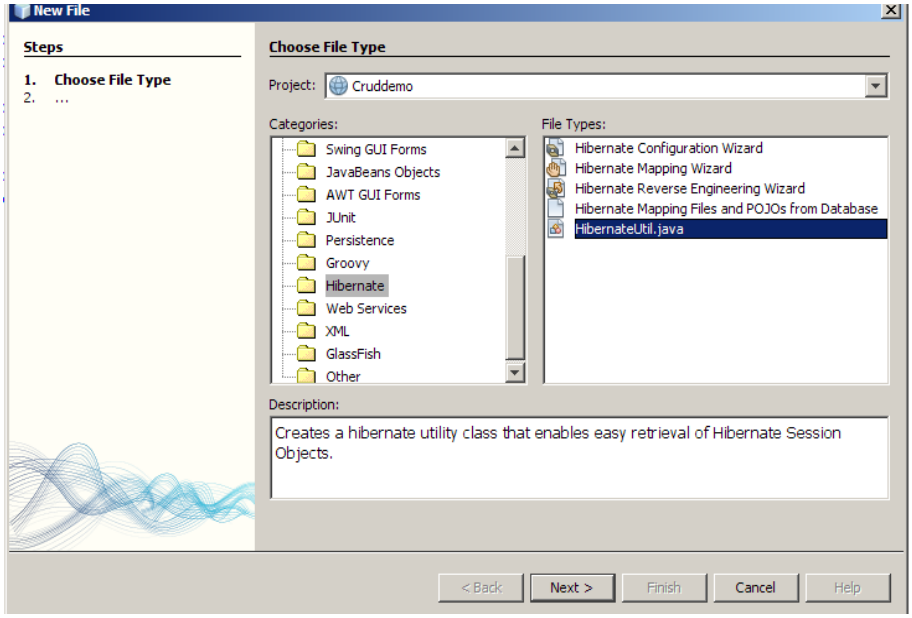
Jokaisen XML-mallinustiedoston alkuun listätään Hibernaten oma Doctype määrittely, jonka avulla Hibernate tunnistaa tiedoston omakseen. Kaikki mallinnus tulee tapahtua <hibernate-mapping> tagien sisällä. **Class-tagin** sisällä määritellään tietokannan scheman nimi, POJO-luokan sijainti ja tietokannan taulun nimi. **Id-tagin** sisällä määritellään taulun pääavain, luokan id, tyyppi ja tapa miten pääavain muodostetaan. Generator luokan ollessa identity, Hibernate jättää id:n luonnin tietokannan vastuulle(auto increment). **Property** –tagilla määritellään muut taulun kentät vastaamaan luokan ominaisuuksia.

7 DAO –LUOKAN TOTEUTUS

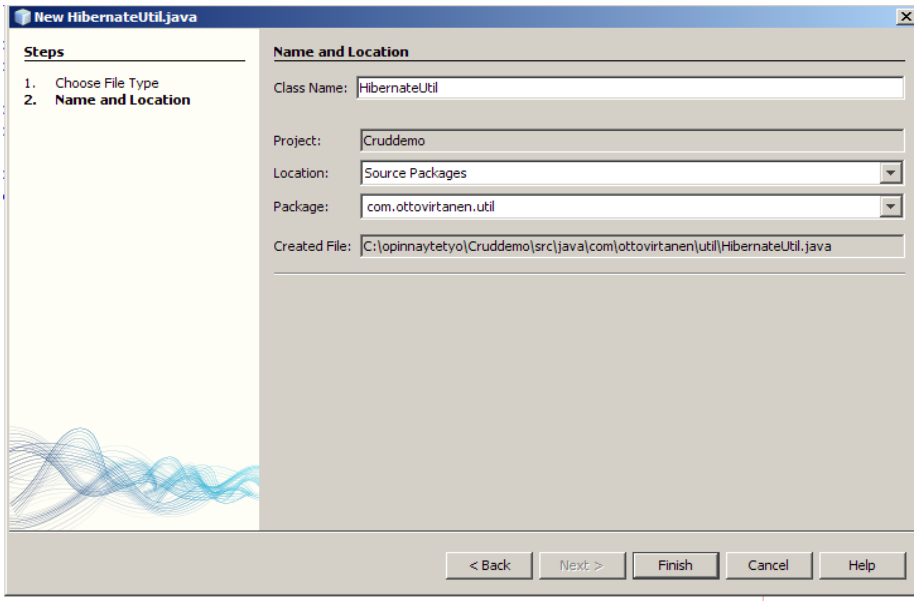
7.1 SessionFactory Util

Ennen kuin aletaan oikeasti käyttää Hibernaten ominaisuuksia ja metodeja tietokanta-hakuihin ja kyselyihin, on hyvä luoda erillinen HibernateUtil luokka, joka sisältää tietokantatransaktioiden käsittelyyn tarvittavan SessionFactory olion alustamisen. Netbeansistä löytyy kätevästi valmis toiminto tiedoston tekoon.

Valitaan New file – Hibenate – HibernateUtil.java



Nimetään tiedosto HibernateUtil.java :ksi ja laitetaan haluttuun polkuun.




```
package com.ottovirtanen.util;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try {
            sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Netbeansin generoima HibernateUtil luokka luo meille valmiiksi SessionFactory olion alustuksen virheenkäsittelijöineen ohjelman kääntämisen yhteydessä. Tätä samaa sessionFactory oliota kutsumme halutessamme tehdä tietokantakyselyitä ja hakuja. HibernateUtil toimii tällaisenaan, eli tähän tiedostoon tarvitse tehdä muutoksia.

7.2 DAO –rajapinnan luonti

Luodaan DAO mallin mukainen rajapinta(interface).

```
package com.ottovirtanen.dao;
import com.ottovirtanen.Person;
import com.ottovirtanen.Project;
import java.util.List;
public interface ProjectDAO{
   public abstract List listProjects();
   public abstract List getProjectById(int id);
   public abstract void addProject(Project project);
   public abstract void updateProject(Project project);
   public abstract void deleteProject(Project project);
}
```

DAO rajanpinnassa määritellään CRUD operaatioihin tarvittavat metodit.

7.3 DAO – rajapinnan toteteuttava luokka

Seuraavaksi luodaan DAO- luokka CRUD-operaatioilla(Create, read, update, delete). DAO-luokkien avulla voidaan erottaa alemman tason tietokantakäsittely liiketoiminta-logiikasta.

Luodaan uusi luokka nimeltä ProjectDAOImpl.java samaan pakettiin ProjectDAO:n kanssa

Liitetään luokkaan tarvittavat Javan ja Hibernaten kirjastoja:

```
//hibernate libraries
import org.hibernate.Session;
import org.hibernate.Query;
import org.hibernate.Transaction;
import org.hibernate.HibernateException;
import org.hibernate.Criteria;
import org.hibernate.criterion.*;
//sessionFactory util
import com.ottovirtanen.util.HibernateUtil;
//java libraries
import java.util.*;
import java.io.*;
```

Toteutetaan rajapinta lisäämällä luokan perään ”implement ProjectDAO”.

```
public class ProjectDAOImpl implements ProjectDAO{
   public ProjectDAOImpl() {}
```

```
    public List listProjects() {
      List projects = null;
     //Getting the sessionfactory from HibernateUtil.java
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      try {
         session.beginTransaction();
         //creating a list object from db by HQL query
         projects = session.createQuery("from Project order by projectId").list();
          //SQL way
         //projects = session.createSQLQuery("SELECT * FROM project").addEntity("project", Pro-
ject.class).list();
         //criteria way
        //Criteria crit = session.createCriteria(Project.class);
        // crit.addOrder(Order.asc("projectId"));
         //projects = crit.list();
         session.getTransaction().commit();
      } catch (HibernateException e) {
          session.getTransaction().rollback();
         e.printStackTrace();
       }
```

Seuraavaksi luodaan listProjects() metodi jolla haetaan kaikki Project luokan oliot Lista olioon.

```
public static List listProjects() {
  //Getting the sessionfactory from HibernateUtil.java
  Session session = HibernateUtil.getSessionFactory().getCurrentSession();
  List projects = null;
  try {
      session.beginTransaction();
     //creating a list object from db by HQL query
     projects = session.createQuery(" from Project").list()
    session.getTransaction().commit();
  } catch (HibernateException e) {
      session.getTransaction().rollback();
     e.printStackTrace();
   }
  return projects;
}}
```

Metodin alussa otetaan HibernateUtil.java luokasta tietokantakäsittelyihin tarvittava sessio olion ilmentymä. SessionFactory oliosta löytyy tarvittavat metodit hakuihin, transaktioihin jne.

Varsinainen tietokantatransaktio aloitetaan session.beginTransaction(); metodilla. Tietokantakysely on tehty Hibernaten omalla oliopohjaisella HQL-kyselykielellä:

```
projects = session.createQuery(" from Project").list()
```

”from project” hakulause hakee kaikki Project luokan oliot, sekä kaikki Project luokasta perityt luokat. Laiskan latauksen ollessa käytössä, alaluokat ladataan kannasta vasta kun niitä tarvitaan. List() metodi muuttaa haun listarakenteeksi.

Muita tapoja kyselyn tekemiseen on SQL ja Criteria Api.
Sama tulos saataisiin vastaavasti SQL:n haulla,

```
projects = session.createSQLQuery("SELECT * FROM project").addEntity("project", Project.class).list();
```

ja Criteria Api:n tarjoamalla rajapinnalla createCriteria:

```
Criteria crit = session.createCriteria(Project.class);
projects = crit.list();
```

Criteria Api tarjoaa myös kattavasti erilaisia metodeja kyselyiden rajaamiseen.

```
crit.addOrder(Order.asc(""));
crit.setMaxResults(50);
crit.add(Restrictions.isNull(""));
```

Lisää Criteria API:n metodeja löydät osoitteesta:

http://www.redhat.com/docs/en-US/JBoss_Hibernate/3.2.4.sp01.cp03/api/hibernate-core/org/hibernate/Criteria.html

Seuraavaksi päätetään tietokantatransaktio kyselyn onnistuessa komennolla:

session.getTransaction().commit();

tai epäonnistuessa

session.getTransaction().rollback();

Viimeiseksi palautetaan kyselyn tuloksena saatu lista olio, joka sisältää kaikki Project luokan oliot.

return projects;

7.3.1 Projektin haku kannasta id:n perusteella

Id:n perusteella haettaessa välitetään metodille id:n arvo parametrinä ja rajataan HQL kyselyä id:n perusteella:

```
Query q = session.createQuery("from Project where projectId=:id ");
project.setInteger("id",id);
```

projectId:n arvo annetaan lauseelle tietoturvasyistä numerisena.

Koko metodi näyttää seuraavalta:

```
public List getProjectById(int id) {
      //Getting the sessionfactory from HibernateUtil.java
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      List project = null;
      try {
          session.beginTransaction();
         Query q = session.createQuery("from Project where projectId=:id ");
         q.setInteger("id",id);
         project = q.list();
        session.getTransaction().commit();
      } catch (HibernateException e) {
          session.getTransaction().rollback();
         e.printStackTrace();
       }
      return project;
   }
```

Mikäli sama haku toteutettaisiin Criteria Api:lla, tulisi id:n perusteella rajattaessa käyttää rajauksessa Restrictions rajapintaa.

```
Criteria crit = session.createCriteria(Project.class).add(Restrictions.idEq(id));
```

7.3.2 Projektin lisäys

Lisätään addProject() niminen metodi, jolle tulee välittäjäolion avulla arvot lisäykseen.

```
public void addProject(Project project) {
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
       session.save(project);
       session.getTransaction().commit();
    } catch (HibernateException e) {
        session.getTransaction().rollback();
       e.printStackTrace();
    }
```

Aloitettuamme listauksen tapaan sessiot ja transaktiot, tallennetaan Hibernaten metodilla save() uusi projekti tietokantaan ja lopetetaan transaktio.

7.3.3 Projektin muokkaus

Muokkaus ei eroa juurikaan lisäyksestä, mutta päivitettävä rivi on luonnollisesti haettava kannasta.

```
public void updateProject(Project project) {
    //hibernate session configurations
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
      Project p = (Project) session.get(Project.class, project.getProjectId());
       //sets
       p.setProjectName(project.getProjectName());
       p.setProjectCode(project.getProjectCode());
       p.setProjectDescription(project.getProjectDescription());
       //update
       session.update(p);
       session.getTransaction().commit();
    } catch (HibernateException e) {
        session.getTransaction().rollback();
       e.printStackTrace();
    }
```

Muokattava rivi haetaan parametrinä tulleen olion id:n avulla, session rajapinnan get metodia käyttäen. Sama olisi myös voitu tehdä Criteria:lla, SQL-lauseella tai HQL lauseella.

Alapuolen rivi hakee project olioon päivitettävän rivin tietokannasta.

```
Project project = (Project) session.get(Project.class, project.getProjectId());
```

Muilta osin lisäyksestä päivitys eroaa vain session.update() metodin osalta.

7.3.4 Projektin poistaminen

Tehtyämme olion muokkauksen on sen perusteella helppo tehdä poistaminenkin.

```
public void deleteProject(Project project) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
       //id
       Project p = (Project) session.load(Project.class, project.getProjectId());
       session.delete(p);
       session.getTransaction().commit();
    } catch (HibernateException e) {
       session.getTransaction().rollback();
       e.printStackTrace();
    }
 }//deleteProject
```

Poistossa haetaan parametrinä tulleen olion mukaan haluttu rivi ja poistetaan se *session.delete();* metodilla

8 KÄYTTÖLIITTYMÄN LUONTI ICEFACES:N AVULLA

8.1 Tapahtumankäsittelijäluokan luonti

Aloitetaan käyttöliittymän teko luomalla erillinen tapahtumankäsittelijäluokka ViewHandler.java, jossa tehdään ICEfacesin tarvitsemat tapahtumankäsittelyt, viestit ja tarkistukset. Tapahtumankäsittelijät ym. olisi voitu laittaa myös ProjectManager luokkaan, mutta silloin sovellus ei olisi enää MVC arkkitehtuurin mukainen. Nyt kun ICEfaces:n tarvitsemat osat ovat omassa luokassaan, voidaan käyttöliittymäkomponentit vaihtaa helposti, tekemättä muutoksia DAO-luokkaan.

Luodaan uusi Java luokka samaan polkuun muiden luokkien kanssa nimeltään ViewHandler.java.

Lisätään luokkaan seuraavat kirjastot:

```
//faces libraries
import javax.faces.event.ActionEvent;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.*;
//java libraries
import java.util.*;
```

Lisätään seuraavaksi tapahtumankäsittelyyn tarvittavia muuttujia:

Projekti olio, jota käytetään uuden projektin luonnin yhteydessä:

```
private Project newProject = new Project();
```

Projekti olio, jota käytetään mm. uuden projektin muokkauksessa

```
private Project currentProject = new Project();
```

Lista projektien nimistä, käytetään pudotusvalikoissa.

```
private List projectItems = new ArrayList();
```

Valittuna olevan projektin id

```
private String selectedProject = "";
```

Muuttujat tarvitsevat vielä getter ja setter metodit.

```
//--------- Project Setter and Getter methods ----------
   public Project getNewProject(){
      return newProject;
   }
   public void setNewProject(Project newProject){
      this.newProject = newProject;
   }
   public List getProjectItems(){
       return projectItems;
    }
   public Project getCurrentProject(){
      return currentProject;
   }
   public void setCurrentProject(Project currentProject){
      this.currentProject = currentProject;
   }
    public String getSelectedProject(){
      return selectedProject;
   }
   public void setSelectedProject(String selectedProject){
      this.selectedProject = selectedProject;
   }
```

Tehdään uusi olio ProjectDAOImpl luokasta

```
private ProjectDAOImpl projectDAOImpl = new ProjectDAOImpl();
```

Seuraavaksi lisätään DAO luokan metodien kutsuun tarvittavat metodit.

```
// ------------PROJECT CRUD OPERATIONS ------------------>
   public List listProjects(){
     List projects;
     projects = projectDAOImpl.listProjects();
     currentProject.clear();
     return projects;
   }
```

```
public void createProject(ActionEvent event){
        projectDAOImpl.addProject(newProject);
        newProject.clear();
    }
    public void updateProject(ActionEvent event){
        projectDAOImpl.updateProject(currentProject);
    }
    public void deleteProject(ActionEvent event){
        projectDAOImpl.deleteProject(currentProject);
       //Clearing the currentProject values from input fields
       currentProject.clear();
       getProjectMenuItems();
    }
```

ActionEvent on JSF:n tapahtumienkäsittelyluokka, jolla voidaan “kuunnella” Iceface-sin submit painikkeiden painalluksia. currentProject.clear() metodilla kutsutaan Project.java luokan clear() metodia, jolla pyyhitään valittuna oleva projekti. getProjectMenuItems() metodilla haetaan arvot pudotusvalikkoon.

Seuraavaksi esitellään metodit pudotusvalikon arvojen tulostusta ja pudotusvalikon tapahtumakäsittelyä varten.

```
//---- Selection values to edit.xhtml page
    public void getProjectMenuItems() {
       //project items
       if(projectItems.size() >=1){
           projectItems.clear();
        }
       List projectResult = projectDAOImpl.listProjects();
       projectItems.add(new SelectItem(“”));
       for (Iterator it = projectResult.iterator(); it.hasNext();) {
           Project project = (Project) it.next();
           projectItems.add(new SelectItem(project.getProjectId(), project.getProjectName()));
        }
```

getProjectMenuItems() metodilla tulostetaan projektien nimet ja id:t pudotusvalikkoon. Ensiksi metodissa tarkistetaan onko pudotusvalikossa jo arvoja, ettei pudotusvalikkoon tule arvoja useaan otteeseen. Seuraavaksi haetaan kaikki projektit DAO luokasta, laitetaan tyhjä arvo pudotusvalikkoon ja iteraatiolla asetetaan projektien nimet ja id:t pudotusvalikkoon.

Seuraavaksi tarvitaan kuuntelija muutoksille pudotusvalikossa.

```
public void projectValueChanged(ValueChangeEvent event){
    if(event.getNewValue() != null){
        int id = Integer.parseInt((String)event.getNewValue());
        List projectResult =  projectDAOImpl.getProjectById(id);
        currentProject = (Project)projectResult.get(0);
    }
}
```

Metodissa ”kuunnellaan” muutoksia pudotusvalikossa ja laitetaan sen perusteella currentProject:n arvoksi valittu projekti.

8.2 Icefaces sivujen luonti

JSF 2 version isoin uudistus on, että JSP sivut on korvattu facelets tekniikalla. Icefaces 2.0 ei siis luonnollisesti myöskään tue enää JSP:tä. Facelet-tekniikkaa sivujen tulee olla .xhtml päätteisiä valideja XML-dokumentteja.

8.2.1 Mallisivu Template.xhtml

Aloitetaan sivujen luonti luomalla erillinen Template.xhtml tiedosto, joka toimii sivujen mallina. Template tiedostoon voidaan siis tehdä esimerkiksi sivujen navigaatiot ja sivujen yhteiset

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ice="http://www.icesoft.com/icefaces/component">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
        <link href="./xmlhttp/css/rime/rime.css"
rel="stylesheet" type="text/css"/>
         <h:outputScript name="jsf.js" library="javax.faces"/>
        <title>Project crud</title>
    </h:head>
    <h:body>
        <div id="top" class="top">
            <h1>Project management CRUD</h1> <span>Made with
Hibernate 3.2.5 and ICEfaces 2.0 Beta 1</span>
            <ul>
                <li><a href="read.iface">read</a></li>
                <li><a href="create.iface">create</a></li>
                 <li><a href="update.iface">Update</a></li>
            </ul>
            <ui:insert name="top"></ui:insert>
        </div>
```

```
        <div id="content" style="max-width:800px;"
class="center_content">
            <ui:insert name="content"></ui:insert>
        </div>
    </h:body>
</html>
```

8.2.2 Projektien lisäyssivu create.xhtml

Tehdään seuraavaksi projektien lisäystä varten Template.xhtml:n client sivu. Lisätään uusi Icefaces Facelets Client tiedosto(New File, ICEFaces, ICEFaces Facelets Client). Valitse Templateksi Template.xhtml sivu tiedostoa luodessasi. Lisää käyttöliittymäkomponentit seuraavasti sivun sisältöosaan (ui:define name=”content”)

```
        <ui:define name="content">
          <ice:form id="iceForm">
            <ice:panelTabSet>
              <!-- CREATE -->
              <ice:panelTab label="Create Project">
                <ice:panelGrid columns="1">
                  <ice:outputText value="Name:"/>
                  <ice:inputText value="#{ViewHandler.newProject.projectName}" re-
quired="true"/>
                  <ice:outputText value="Code:"/>
                  <ice:inputText value="#{ViewHandler.newProject.projectCode}" re-
quired="true"/>
                  <ice:outputText value="Description:"/>
                  <ice:inputTextarea rows="10" cols="50"
value="#{ViewHandler.newProject.projectDescription}" required="true"/>
                  <ice:commandButton id="button" type="submit" value="Add Project"
                                actionListener="#{ViewHandler.createProject}" />
                </ice:panelGrid>
              </ice:panelTab>
            </ice:panelTabSet>
          </ice:form>
        </ui:define>
      </ui:composition>
```

Kaikki ICEFacesin tarvitsemat komponentit tulee sijoittaa **<ice:form>** lomakkeen sisälle, koska ICEFacesin komponenttien toiminta perustuu lomakkeiden käyttöön.

panelTabSet puolestaan luo välilehtikehyksen, johon voidaan liittää uusia välilehtiä panelTab tagilla. **<ice:panelGrid/>** :n avulla luodaan taulukko, joka osaa tehdä tarvittavat taulukon rivit tagin sisällä olevien komponenttien perusteella automaattisesti.

<ice:inputText/> komponenttien arvoina on newProject olion attribuutit ja **required=”true”** arvolla asetetaan tieto pakolliseksi.

<ice:commandButton/> komponentin **actionListener** arvolla viitataan tapahtumankäsittelijäluokkan createProject metodiin. Tapahtumakäsittelijäluokat tulee ilmoittaa Icefacelle erillisessä faces-config.xml tiedostossa seuraavasti:

```
<managed-bean>
  <managed-bean-name>ViewHandler</managed-bean-name>
    <managed-bean-class>com.ottovirtanen.ui.ViewHandler</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Ensiksi annetaan nimi, mihin viitataan Facelet sivulla, toiseksi polku ja kolmanneksi näkyvyysalue.

8.2.3 Selaussivu read.xhtml

Aloita luomalla edellisen sivun kaltainen Client –sivu. Lisää ice:form lomakkeen sisälle seuraavat komponentit ja arvot:

```
<ice:panelTab label="View projects">
  <ice:panelGrid columns="1">
    <ice:dataTable value="#{ViewHandler.listProjects()}"  title="taulu" width="100%" rows="10"
id="projectTable" var="item">
      <ice:column>
        <f:facet name="header">
          <ice:outputText value="Code"/>
        </f:facet>
        <ice:outputText value="#{item.projectCode}"/>
      </ice:column>
      <ice:column>
        <f:facet name="header">
          <ice:outputText value="Name"/>
        </f:facet>
        <ice:outputText value="#{item.projectName}"/>
      </ice:column>
      <ice:column>
```

```
            <f:facet name="header">
               <ice:outputText value="Description"/>
            </f:facet>
            <ice:outputText  value="#{item.projectDescription}" />
         </ice:column>
      </ice:dataTable>
</ice:panelTab>
```

Icefacesissä taulukko luodaan **<ice:dataTable>** tagilla, joka osaa lukea listaolion suoraan taulukkoon, kun haluttu tulostus on esitetty. **<f:facet name="header">** tagilla ilmoitetaan dataTabelin otsikkorivit, seuraavat rivit dataTable osaa automaattisesti tulostaa normaaleiksi.

8.2.4 Muokkaus- ja poistosivun luonti update.xhtml

Muokkauksessa tarvitsemme nyt aiemmin luotuja getProjectMenuItems() ja projectValueChanged() metodeja.

```
      <ice:outputText value="Project:"/>
      <ice:setEventPhase events="ValueChangeEvent" phase="INVOKE_APPLICATION">
         <ice:selectOneMenu  style="width:300px" label="menu"
value="#{ViewHandler.selectedProject}"
                        partialSubmit="true"
                        valueChangeListener="#{ViewHandler.projectValueChanged}">
            <f:selectItems value="#{ViewHandler.projectItems}" />
         </ice:selectOneMenu>
      </ice:setEventPhase>
      <ice:outputText value="Project Name:"/>
      <ice:inputText style="width:290px" value="#{ViewHandler.currentProject.projectName}"/>
      <ice:outputText value="Project Code:"/>
      <ice:inputText  style="width:290px" value="#{ViewHandler.currentProject.projectCode}"/>
      <ice:outputText value="Project Description:"/>
      <ice:inputTextarea rows="20" cols="55"
value="#{ViewHandler.currentProject.projectDescription}"/>
      <ice:commandButton type="submit"  value="Save Changes" actionLis-
tener="#{ViewHandler.updateProject}"/>
      <ice:commandButton type="submit" value="Delete Project" actionLis-
tener="#{ViewHandler.deleteProject}"/>

      <ice:setEventPhase events="ValueChangeEvent" phase="INVOKE_APPLICATION">
```

```
        <ice:selectOneMenu  style="width:300px" label="menu"
value="#{ViewHandler.selectedProject}"
                        partialSubmit="true"
                        valueChangeListener="#{ViewHandler.projectValueChanged}">
            <f:selectItems value="#{ViewHandler.projectItems}" />
        </ice:selectOneMenu>
    </ice:setEventPhase>
```

setEventPhase komponentilla kerrotaan ICEFacelle, että näkymän pitää päivittyä pudotusvalikon arvon vaihtuessa. **partialSubmit** arvolla määritellään, että vain tämän osan sivusta tulee päivittyä. **valueChangeListener** arvolla määritellään luokka, joka kuuntelee pudotusvalikon muutoksia. Tällä otetaan siis yhteys aiemmin luomaamme **projectValueChanged** metodiin, joka hakee kannasta **currentProject**:n id:n perusteella halutun käyttäjän. **f:selectIems** komponentilla tulostetaan **getProjectMenuItems()** metodissa määritellyt selectItemit.

9 PERSONS LUOKAN CRUD-OPERAATIOT(MONEN-SUHDE-MONEEN)

Tarkastellaan seuraavaksi tilannetta, jossa kahden luokan välillä on monen suhde moneen assosiaatio. Projektin voi osallistua monta henkilöä ja sama henkilö voi olla monessa projektissa. Kuten jo tietokantaa luodessa huomasit, skriptissä oli erillinen person ja project_person välitaulu.

9.1 POJO-luokkien määrittelyt

Seuraavaksi määritellään Person POJO-luokka

```
package com.ottovirtanen;
import java.util.HashSet;
import java.util.Set;
public class Person {
   private int personId = 0;
   private String firstName = "";
   private String lastName = "";
   private String phone = "";
   private String email = "";
   private Set project = new HashSet();
}
```

ja generoidaan sille setter ja getter metodit Netbeansin aputyökaluilla(kts. Luku 5).

```
POJO –luokkaan liitetään myös uusi
private Set projects = new HashSet();
```

projects olio, jolla voidaan hakea henkilöiden projektit kätevästi.

Lisätään vielä luokan loppuun clear() metodi.

```
public String clear() {
      firstName = "";
      lastName= "";
      phone = "";
      email = "";
      return "clear";
   }
```

Samanlainen olio tulee myös luoda toisinpäin Project.java luokaan, jotta voidaan hakea projektiin kuuluvat henkilöt.

Lisää siis rivi,

private Set persons = new HashSet();

Project luokkaan ja sille setter ja getter metodit yllämainitulla tavalla, sekä tuo luokkaan Javan kirjastot:

import java.util.HashSet;

import java.util.Set;

9.2 Mallinnustiedostojen määrittelyt

Tehdään uusi Person.hbm.xml tiedosto, johon määritellään Project.hbm.xml tiedoston tavoin miten yhdistytään person tietokannan taulun kanssa

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 29.8.2010 15:34:52 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping default-lazy="false">
 <class catalog="cruddemo" name="com.ottovirtanen.Person" lazy="false" table="person">
  <id name="personId" type="java.lang.Integer">
   <column name="person_id"/>
   <generator class="identity"/>
  </id>
  <property name="firstName" type="string">
   <column name="first_name"/>
  </property>
  <property name="lastName" type="string">
   <column name="last_name"/>
  </property>
  <property name="phone" type="string">
   <column name="phone"/>
  </property>
  <property name="email" type="string">
   <column name="email"/>
  </property>
  <set inverse="true" lazy="false" name="projects" table="project_person">
   <key>
    <column name="person_id" not-null="true"/>
   </key>
   <many-to-many lazy=”false” entity-name="com.ottovirtanen.Project">
    <column name="project_id" not-null="true"/>
   </many-to-many>
  </set>
 </class>
```

</hibernate-mapping>

Mallinnustiedosto on muuten samankaltainen Project:n kanssa, mutta loppuun on lisätty <set> tagien sisälle luokkien väliset assosiaatiomäärittelyt.

inverse="true"

Inverse arvolla määrittellään kumpi puoli on vastuussa assosaatioiden hallinnasta ja kumpi puoli hoitaa tarvittavat SQL-kyselyt. Koska Person.hbm.xml tiedoston set:ssä on määritelty inversen arvoksi ”true”, assosiaatioiden hallinta tapahtuu Project –päässä.

lazy="false"

Laiskan latauksen käyttäminen tietokantahauissa. Ollessa true Hibernate hakee Person luokan tietokantarivit ainoastaan kun niitä tarvitaan. Tämä on esimerkissä otettu pois, transactioiden ja sessioiden luontiin liittyvien ongelmien takia. lue lisää viimeisessä luvussa

name="projects"

Nimi assosaatiolle. Tulee olla samanniminen Person POJO-luokassa olevan Set:n kanssa.

table="project_person"

Monen suhde moneen tapauksessa tarvittavan liitostaulun nimi.

<key column="person_id"/>

Elementillä key määritellään viittaava avain project_person taulussa.

<many-to-many class="com.ottovirtanen.Project" column="project_id"/>

Many-to-many elementillä ilmoitetaan mihinkä luokkaan ja tietokantataulun riviin liitytään.

Vastaavasti set elementti tulee lisätä Project.hbm.xml tiedostoon seuraavasti:

```
<set lazy="false" inverse=”false” name="persons" table="project_person">
    <key column="project_id"/>
    <many-to-many class="com.ottovirtanen.Person" column="person_id"/>
</set>
```

Huomaa **inverse=”false”**, tällä määritellään että tämä luokka ei ole vastuussa olioden hausta kannasta. Hibernate.cfg.xml tiedostoon tulee myös lisätä uusi lisätty luokka mapping elementillä.

<mapping resource="com/ottovirtanen/Person.hbm.xml"/>

9.3 Uusi POJO-luokka

Luo edellä kuvailtujen määrityksien kaltainen POJO-luokka samalla tavalla, kuin Project.java luokka tehtiin oppaan alussa.

```
package com.ottovirtanen;
import java.util.HashSet;
import java.util.Set;
public class Person{
    private Integer personId;
    private String firstName;
    private String lastName;
    private String phone;
    private String email;
    private Set projects = new HashSet(0);
   public Person() {
   }
   public Person(String firstName, String lastName, String phone, String email) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.phone = phone;
      this.email = email;
   }
   public Integer getPersonId() {
      return this.personId;
   }
   public void setPersonId(Integer personId) {
      this.personId = personId;
   }
   public String getFirstName() {
      return this.firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public String getLastName() {
      return this.lastName;
   }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getPhone() {
        return this.phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public String getEmail() {
        return this.email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public void clear(){
        personId = null;
        firstName = null;
        lastName = null;
        phone = null;
        email = null;
    }
}//class
```

Tässä tiedostossa ei liene epäselvyyksiä, mikäli haluttaisiin saada myös tiettyä henkilöä koskevat projektit helposti selville, tulisi tähän luokkaan laittaa Project luokan tavoin Set-tyyppinen Projects olio get- ja settereineen, sekä mallinnuksissa tulisi Person.hbm.xml tiedostossa olla Set määritelty päinvastaisesti, kuten mallinnuskohdassa jo selostettiin.

9.4 Crud operaatiot Facelet sivuille

Lisätään aluksi henkilöiden lisäystä varten create.xhtml sivulle uusi välilehti.

```
<ice:panelTab label="Create Person">
    <ice:panelGrid columns="2">
        <ice:outputText value="Firstname:"/>
        <ice:inputText value="#{ViewHandler.newPerson.firstName}" required="true">
        </ice:inputText>
        <ice:outputText value="Lastname:"/>
        <ice:inputText value="#{ViewHandler.newPerson.lastName}" required="true">
```

```
    </ice:inputText>
    <ice:outputText value="Phone number:"/>
    <ice:inputText id="phoneNumber" value="#{ViewHandler.newPerson.phone}"  re-
quired="true">
    </ice:inputText>
    <ice:outputText value="Email:"/>
    <ice:inputText  value="#{ViewHandler.newPerson.email}" required="true"/>
    <ice:commandButton id="button2" type="submit" partialSubmit="true" value="Add"
            actionListener="#{ViewHandler.createPerson}" />
  </ice:panelGrid>
</ice:panelTab>
```

Tämä on tietysti simppeliä, muutetaan ainoastaan kentät kuvaamaan Person luokan attribuutteja. Seuraavaksi vielä päivityssivu samalla tapaa:

```
<ice:panelTab label="Update/delete persons">
  <ice:panelGrid columns="2">
    <ice:outputText value="Person:"/>
    <ice:setEventPhase events="ValueChangeEvent"
            phase="INVOKE_APPLICATION">
      <ice:selectOneMenu style="width:300px" value="#{ViewHandler.selectedPerson}"
          partialSubmit="true"
          valueChangeListener="#{ViewHandler.personValueChanged}">
        <f:selectItems value="#{ViewHandler.personItems}"/>
      </ice:selectOneMenu>
    </ice:setEventPhase>
    <ice:outputText id="firstname" value="First name:"/>
    <ice:inputText style="width:290px" value="#{ViewHandler.currentPerson.firstName}"/>
    <ice:outputText value="Last name:"/>
    <ice:inputText  style="width:290px" value="#{ViewHandler.currentPerson.lastName}"/>
    <ice:outputText value="Phone:"/>
    <ice:inputText value="#{ViewHandler.currentPerson.phone}"/>
    <ice:outputText value="Email:"/>
    <ice:inputText value="#{ViewHandler.currentPerson.email}"/>
    <ice:outputText value="Department:"/>
    <ice:selectOneMenu style="width:300px" value="#{ViewHandler.selectedDepartment}"
            partialSubmit="true"
            valueChangeListener="#{ViewHandler.departmentValueChanged}">
      <f:selectItem itemLabel="#{ViewHandler.currentPerson.department.departmentName}"/>
      <f:selectItems  value="#{ViewHandler.departmentItems}"/>
    </ice:selectOneMenu>
```

```
        <ice:commandButton type="submit" value="Save Changes" actionLis-
tener="#{ViewHandler.updatePerson}"/>
        <ice:commandButton type="submit" value="Delete Person" actionLis-
tener="#{ViewHandler.deletePerson}"/>
    </ice:panelGrid>
</ice:panelTab>
```

9.5 PersonDAO luokkien teko

Luo PersonDAO luokan rajapinta ja sen toteuttava PersonDAOImpl luokka, Project-DAO ja ProjectDAOImpl kaltaisiksi.

PersonDAO:

```
package com.ottovirtanen.dao;
import com.ottovirtanen.Person;
import com.ottovirtanen.Project;
import java.util.*;
public interface PersonDAO {
   public abstract List listPersons();
   public abstract List getPersonById(int id);
   public abstract void addPerson(Person person);
   public abstract void updatePerson(Person person, Department department);
   public abstract void deletePerson(Person person);
}//interface
```

PersonDAOImpl

```
package com.ottovirtanen.dao;

//hibernate libraries
import com.ottovirtanen.Person;
import com.ottovirtanen.Project;
import org.hibernate.Session;
import org.hibernate.Query;
import org.hibernate.HibernateException;
import org.hibernate.Criteria;
import org.hibernate.criterion.*;

//sessionFactory util
import com.ottovirtanen.util.HibernateUtil;
//java libraries
import java.util.*;
import java.io.*;
import java.util.Set;

import javax.faces.event.ActionEvent;

public class PersonDAOImpl implements PersonDAO {
```

```
    public PersonDAOImpl() {}

    public List listPersons() {
      //Getting the sessionfactory from HibernateUtil.java
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      List persons = null;
      try {
         session.beginTransaction();
         //creating a list object from db by HQL query
          persons = session.createQuery("from Person").list();
          //SQL way
         //persons = session.createSQLQuery("SELECT * FROM person").addEntity("person", Per-
son.class).list();
          session.getTransaction().commit();
      } catch (HibernateException e) {
           session.getTransaction().rollback();
         e.printStackTrace();
       }
      return persons;
    }//listPersons
    public List getPersonById(int id) {
      //Getting the sessionfactory from HibernateUtil.java
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      List person = null;
      try {
         session.beginTransaction();
         Query q = session.createQuery("from Person where personId=:id");
         q.setInteger("id",id);
         person = q.list();
          session.getTransaction().commit();
      } catch (HibernateException e) {
           session.getTransaction().rollback();
         e.printStackTrace();
       }
      return person;
    }//getPersonById
     public void addPerson(Person person) {
      //Getting the sessionfactory from HibernateUtil.java
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      try {
        session.beginTransaction();
```

```
        session.save(person);
         session.getTransaction().commit();
    } catch (HibernateException e) {
          session.getTransaction().rollback();
        e.printStackTrace();
    }
}//addProject

public void updatePerson(Person person, Department department) {
    //hibernate session configurations
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
   try {
       session.beginTransaction();
        //getting the right row from db
        Person p = (Person) session.get(Person.class, person.getPersonId());
        //sets
        p.setFirstName(person.getFirstName());
        p.setLastName(person.getLastName());
        p.setPhone(person.getPhone());
        p.setEmail(person.getEmail());
        p.setDepartment(department);
        //update
        session.update(p);
           session.getTransaction().commit();
    } catch (HibernateException e) {
          session.getTransaction().rollback();
        e.printStackTrace();
    }
}//updatePerson

 public void deletePerson(Person person) {
    //Getting the sessionfactory from HibernateUtil.java
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
        //id
        Person p = (Person) session.load(Person.class, person.getPersonId());
        session.delete(p);
         session.getTransaction().commit();
    } catch (HibernateException e) {
         session.getTransaction().rollback();
```

```
            e.printStackTrace();
        }
    }//deletePerson

        public void addProjectToPerson(Person person, Project project){
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        try {
            session.beginTransaction();
            Project p = (Project)session.load(Project.class,project.getProjectId());
            Person pe = (Person)session.load(Person.class,person.getPersonId());
           pe.getProjects().add(p);
           session.save(pe);

        } catch (HibernateException e) {
           session.getTransaction().rollback();
            e.printStackTrace();
        }
}
}
```

9.6 ProjectDAO luokan muutokset

ProjectDAO luokkaan tarvitaan nyt Projektiin liittyvien henkilöiden lisäys- ja poisto-toiminnot.

ProjectDAO:

Lisää seuraavat rivit rajapintaan:

```
public abstract void addPersonToProject(Project project, Person person);
public abstract void removePersonFromProject(Project project, Person person);
```

ProjectDAOImpl:

```
public void addPersonToProject(Project project, Person person){
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
       Project p = (Project)session.load(Project.class,project.getProjectId());
       Person pe = (Person)session.load(Person.class,person.getPersonId());
      p.getPersons().add(pe);
      session.save(p);
       session.getTransaction().commit();
    } catch (HibernateException e) {
      session.getTransaction().rollback();
       e.printStackTrace();
    }
 }
 public void removePersonFromProject(Project project, Person person){
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    try {
       session.beginTransaction();
       Project p = (Project)session.load(Project.class,project.getProjectId());
       Person pe = (Person)session.load(Person.class,person.getPersonId());
       p.getPersons().remove(pe);
       session.getTransaction().commit();
    } catch (HibernateException e) {
        session.getTransaction().rollback();
       e.printStackTrace();
    }
 }
```

Molemmille metodeille tulee parametreinä Project ja Person oliot, joista haetaan session.load() metodeillä oikeat arvot tietokannasta. Seuraavaksi haetaan p.getPersons()

metodilla Projektiin liittyvät henkilöt ja remove() tai add() funktiolla poistetaan tai lisätään se hashSet rakenteesta, jolloin Hibernate osaa sen myös automaattisesti päivittää tietokantaan.

9.7 Henkilöiden lisäys projektiin projektien muokkaussivulla

Seuraavaksi lisätään update.xhtml sivulle, projektien muokkaus välilehdelle ominaisuus, jonka avulla voidaan liittää henkilöitä projekteihin.
Aluksi lisätään dataTable taulu, johon tulostetaan kaikki projektiin liitetyt henkilöt:

```
<h3>Participants </h3>
<ice:panelGrid columns="1" style="float:left; padding-right:30px;">
  <ice:dataTable var="person" value="#{ViewHandler.currentProject.persons}">
    <ice:column>
      <f:facet name="header">
        <ice:outputText value="Person ID:"/>
      </f:facet>
      <ice:outputText value="#{person.personId}"/>
    </ice:column>
    <ice:column>
      <f:facet name="header">
        <ice:outputText value="First name:"/>
      </f:facet>
      <ice:outputText value="#{person.firstName}"/>
    </ice:column>
    <ice:column>
      <f:facet name="header">
        <ice:outputText value="Last name:"/>
      </f:facet>
      <ice:outputText value="#{person.lastName}"/>
    </ice:column>
    <ice:column>
      <f:facet name="header">
        <ice:outputText value="Phone:"/>
      </f:facet>
      <ice:outputText value="#{person.phone}"/>
    </ice:column>
    <ice:column>
      <f:facet name="header">
        <ice:outputText value="Email:"/>
```

```
        </f:facet>
        <ice:outputText value="#{person.email}"/>
      </ice:column>
      <ice:column>
        <f:facet name="header">
          <ice:outputText value="Department"/>
        </f:facet>
        <ice:outputText value="#{person.department.departmentName}"/>
      </ice:column>
      <ice:column>
        <f:facet name="header">
          <ice:outputText value="Remove"/>
        </f:facet>
        <ice:commandButton type="submit" value="Remove"
                           partialSubmit="true"
                           actionListener="#{ViewHandler.removePersonFromProject(person)}">
        </ice:commandButton>
      </ice:column>
    </ice:dataTable>
```

Kuten jo aiemmin mainittiin listProjects() metodi hakee ”from Project” HQL haullaan kaikki projektin alitaulut myös mukaan List rakenteeseen. Laiskan latauksen ollessa päällä (lazy=”true”), Hibernate hakee vain sillon aliluokkia tietokannasta kun niitä tulostetaan, eli tässä tapauksessa se hakisi ne jälkikäteen.

<ice:dataTable var="person" value="#{ViewHandler.currentProject.persons}">

Valitun projektin listasta voidaan siis hakea suoraan persons hashSet ja purkaa sen dataTable:n.

<ice:commandButton type="submit" value="Remove" partialSubmit="true" actionListener="#{ViewHandler.removePersonFromProject(person)}">

Poistonapille annetaan parametrinä dataTablen loopissa oleva person olio.
Valitun projektin olio menee removePersonFromProject() metodille currentProject oliossa.

10 DEPARTMENT LUOKAN LISÄYS (YHDEN-SUHDE-MONEEN)

Viimeisessä osiossa lisätään vielä yhden-suhde-moneen assosaatio, tarkoituksena liittää henkilöille osastot, joissa he työskentelevät. Yksi henkilö voi olla töissä ainoastaan yhdellä osastolla, mutta osastoilla on töissä useita eri henkilöitä. Assosaatiosuhde on siis yhden-suhde-moneen.

10.1 Mallinnustiedosto

Määritellään seuraavanlainen Department.hbm.xml mallinnustiedosto:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 29.8.2010 15:34:52 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping default-lazy="false">
  <class catalog="cruddemo" name="com.ottovirtanen.Department" table="Department">
    <id name="departmentId" type="java.lang.Integer">
      <column name="department_id"/>
      <generator class="identity"/>
    </id>
    <property name="departmentName" type="text">
      <column name="department_name" not-null="true"/>
    </property>
  </class>
</hibernate-mapping>
```

default-lazy

Tällä arvolla määritellään, että oletusarvoisesti luokka ei käytä laiskaa latausta.

Lisätään Person.hbm.xml tiedostoon seuraavanlainen rivi kuvaamaan assosaatiota:
<many-to-one update="true" class="com.ottovirtanen.Department" column="department_id" lazy="false" name="department"/>

Tällä määrityksellä Hibernate luo automaattisesti uuden department_id rivin person tauluun, jos Hibernaten konfiguraatiossa on seuraava:
<property name="hibernate.hbm2ddl.auto">update</property>
update=”true”

Tällä arvolla annetaan oikeus päivittää Department luokkaa Person luokkaa päivittäessä.

POJO –luokka

Department POJO –luokka ei ole sisällä assosaatiomäärittelyjä, joten se on yksinkertaisesti seuraavanlainen:

```
package com.ottovirtanen;

public class Department {
    private Integer departmentId;
    private String departmentName;

   public Department() {}

   public Department(String departmentName) {
     this.departmentName = departmentName;
   }
   public Integer getDepartmentId() {
      return this.departmentId;
   }
   public void setDepartmentId(Integer departmentId) {
      this.departmentId = departmentId;
   }
   public String getDepartmentName() {
      return this.departmentName;
   }
   public void setDepartmentName(String departmentName) {
      this.departmentName = departmentName;
   }
   public void clear(){
      departmentId = null;
      departmentName = null;
   }
}
```

10.2 DAO luokat

Lisää seuraavanlainen DAO –rajapinta:

```
package com.ottovirtanen.dao;
import com.ottovirtanen.Department;
import java.util.List;

public interface DepartmentDAO {
   public abstract List listDepartments();
   public abstract List getDepartmentById(int id);
   public abstract void addDepartment(Department derpartment);
   public abstract void updateDepartment(Department department);
   public abstract void deleteDepartment(Department department);
}
```

ja sen toteuttava luokka:

```
package com.ottovirtanen.dao;
//hibernate libraries
import com.ottovirtanen.Department;
import org.hibernate.Session;
import org.hibernate.Query;
import org.hibernate.Transaction;
import org.hibernate.HibernateException;
import org.hibernate.Criteria;
import org.hibernate.criterion.*;
//sessionFactory util
import com.ottovirtanen.util.HibernateUtil;
//java libraries
import java.util.*;
import java.io.*;

public class DepartmentDAOImpl implements DepartmentDAO {

   public DepartmentDAOImpl() {}

     public List listDepartments() {
       //Getting the sessionfactory from HibernateUtil.java
       Session session = HibernateUtil.getSessionFactory().getCurrentSession();
       List department = null;
       try {
          session.beginTransaction();
```

```
            //creating a list object from db by HQL query, only department class
             department = session.createQuery("select department from Department department").list();
             //SQL way
            //persons = session.createSQLQuery("SELECT * FROM person").addEntity("person", Per-
son.class).list();
             session.getTransaction().commit();
        } catch (HibernateException e) {
             session.getTransaction().rollback();
            e.printStackTrace();
         }
        return department;
    }//listPersons
     public List getDepartmentById(int id) {
        //Getting the sessionfactory from HibernateUtil.java
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        List department = null;
        try {
            session.beginTransaction();
            Query q = session.createQuery("from Department where departmentId=:id");
            q.setInteger("id",id);
            department = q.list();
             session.getTransaction().commit();
        } catch (HibernateException e) {
             session.getTransaction().rollback();
            e.printStackTrace();
         }
        return department;
    }//getPersonById

      public void addDepartment(Department department) {
       //Getting the sessionfactory from HibernateUtil.java
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        try {
          session.beginTransaction();
            session.save(department);
             session.getTransaction().commit();
        } catch (HibernateException e) {
             session.getTransaction().rollback();
            e.printStackTrace();
        }
```

```
  }//addDepartment

  public void updateDepartment(Department department) {
    //hibernate session configurations
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
   try {
      session.beginTransaction();
       //getting the right row from db
       Department d = (Department) session.get(Department.class, department.getDepartmentId());
       //sets
       d.setDepartmentName(department.getDepartmentName());
       //update
         session.update(d);
        session.getTransaction().commit();
    } catch (HibernateException e) {
         session.getTransaction().rollback();
       e.printStackTrace();
    }
  }//updatePerson
   public void deleteDepartment(Department department) {
    //Getting the sessionfactory from HibernateUtil.java
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    try {
      session.beginTransaction();
       //id
       Department d = (Department) session.load(Department.class, department.getDepartmentId());
       session.delete(d);
        session.getTransaction().commit();
    } catch (HibernateException e) {
         session.getTransaction().rollback();
       e.printStackTrace();
    }
  }//deleteDepartment
}
```

Näiden toteutuksessa ei ollut mitään uutta, luokka on samankaltainen ensimmäisen osion projektiluokkien kanssa, jolloin ei käytetty assosaatioita.

10.3 Tapahtumankäsitteljän muutokset

Alustetaan aluksi tarvittavat muuttujat:

```
//----------------- Department variables
   private Department newDepartment = new Department();
   private Department currentDepartment = new Department();
   // List of derpartments
   private List departmentItems = new ArrayList();
   // Currently selected department
   private String selectedDepartment;
```

ja luodaan uusi ilmentymä DAO-luokasta:

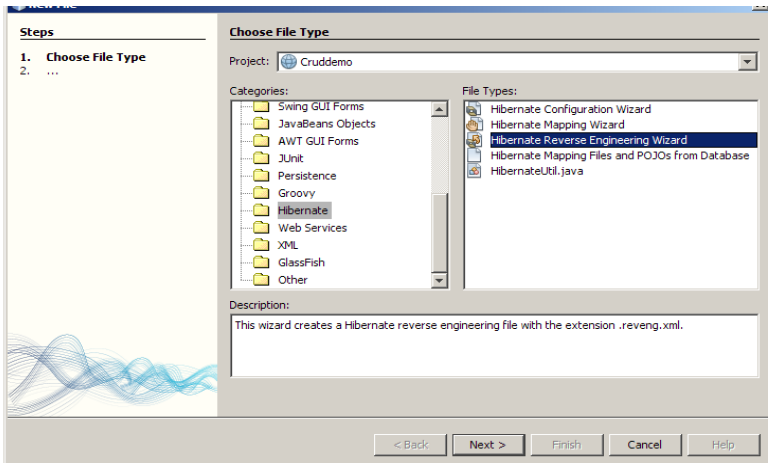
```
private DepartmentDAOImpl departmentDAOImpl = new DepartmentDAOImpl();
```

Seuraavaksi tehdään pudotusvalikon arvojen muodostaja ja pudotusvalikon muutoksien kuuntelija:

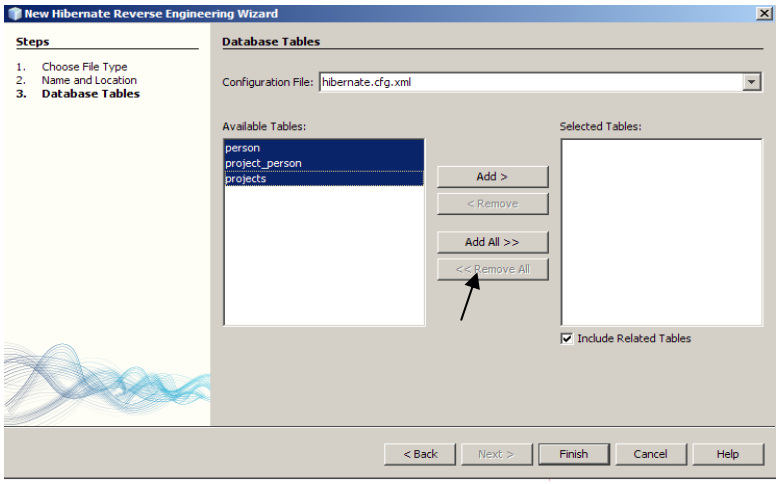
11 KUVAUSTIEDOSTOJEN JA POJO-LUOKKIEN KÄÄNTEINEN GENEROINTI NETBEANSIN HIBERNATE TYÖKALUJEN AVULLA.

Mikäli tietokanta on jo olemassa, Hibernaten tarvitsemat POJO-luokat ja mallinnustiedostot on helppo luoda Netbeansin työkaluilla.

Ensiksi tulee olla hibernate.cfg.xml tiedoston asetukset kunnossa ja hibernate.reveng.xml tiedosto luotuna seuraavalla tavalla:
Valitaan *Hibernate Reverse Engineering Wizard* Hibernate-valikosta

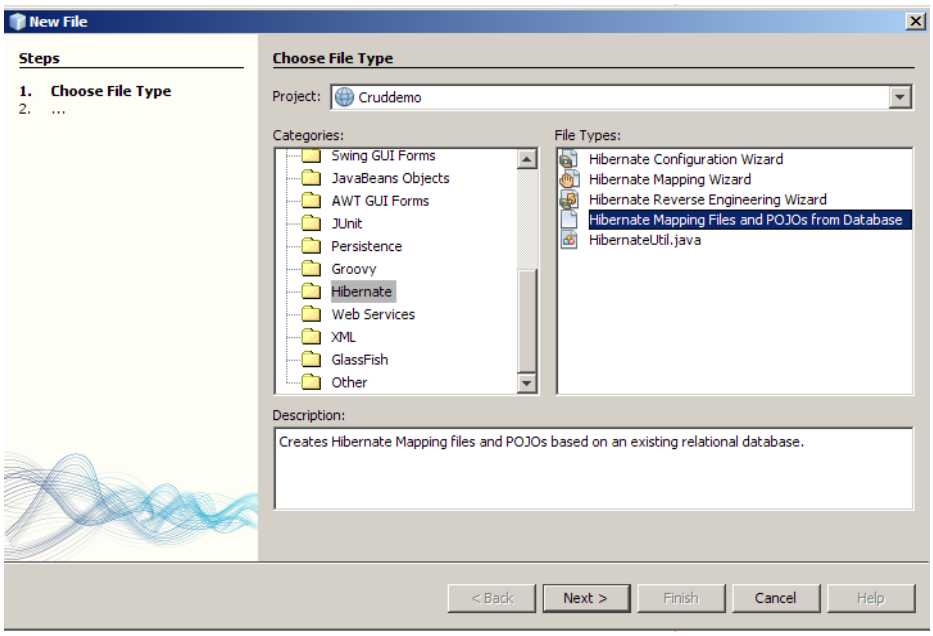


ja halutut tietokantakentät joista reveng.xml koostetaan.

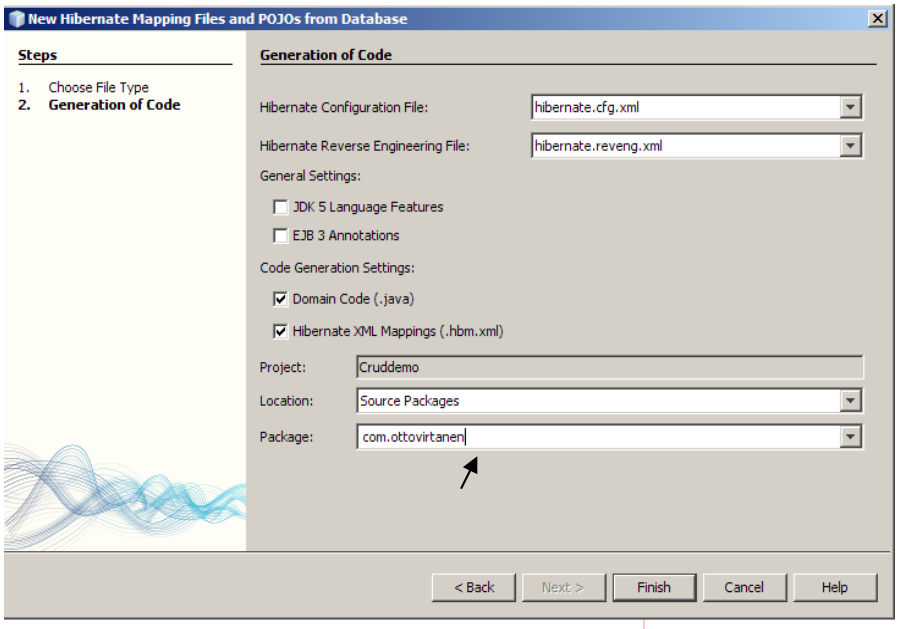


Tämän tuloksena saadaan Reverse Engineering XML aputiedosto, jonka avulla Netbeansin sisäänrakennettu Hibernate työkalu osaa koostaa POJO-luokat ja Hibernaten mallinnustiedostot suoraan tietokannan riveistä. Tämä on selostettuna seuraavassa:

Valitaan *Hibernate Mapping files and POJOS from Database* Hibernate-valikosta,



Ilmoitetaan luokkatiedostojen haluttu polku ja hibernate.cfg.xml ja hibernate.reveng.xml tiedostojen sijainti,



Tämän jälkeen Hibernaten luo tietokannan tauluista mallinnustiedostot ja POJO-luokat.