Mika Kiiskinen

Verkkokauppa J2EE6-teknologialla toteutettuna

Metropolia Ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Insinöörityö
28.10.2010

ALKULAUSE

Tämä insinöörityö tehtiin Aleksi Lappalaisen toimeksiannosta hänen perustamalleen yritykselle. Haluan kiittää häntä sekä työn valvojaa Simo Silanderia yhteistyöstä. Erityiskiitos Juhani Rajamäelle hänen tietokantaosaamisestaan.

Helsingissä 04.10.2010

Mika Kiiskinen

TIIVISTELMÄ

Työn tekijä: Mika Kiiskinen

Työn nimi: Verkkokauppa J2EE6-teknologialla toteutettuna

Päivämäärä: 04.10.2010 **Sivumäärä:** 47 + 1 liite

Koulutusohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Työn ohjaaja: lehtori Simo Silander

Työn ohjaaja: Aleksi Lappalainen

Työssä toteutettiin verkkokauppasovellus työn tilanneen asiakkaan antamien lähtötietojen perusteella. Käytetyt teknologiat sain valita itse, ja ne on valittu palvelemaan sekä asiakkaan etua että omaa oppimistani. Asiakkaan vaatimukset valmiille sovellukselle olivat ulkonäön selkeys ja siisteys, helppo käytettävyys käyttäjille, ylläpitotehtävien suorittamisen mahdollisuus graafisen käyttöliittymän avulla, tietoturva sekä käytettävien työkalujen ja teknologioiden maksuttomuus (avoin lähdekoodi). Lisäksi valittujen teknologioiden tulisi toimia keskenään tehokkaasti valitulla palvelinalustalla ja palvelimella. Insinöörityön ohjaajan kanssa sovimme myös, että työ toteuttaa MVC-mallin (model-view-controller) ja että sovelluksen kieli on helposti vaihdettavissa. Lisäksi oppimistehtäväksi valittiin uuden J2EE6-teknologian tuomat muutokset verrattuna edeltäjäänsä J2EE5-spesifikaatioon.

Verkkokauppasovelluksen toteutuksessa käytetyt teknologiat ovat yleisiä verkkosovellusten tekijöiden suosimia teknologioita. Ohjelmointikielenä on laitteistoriippumaton Java, tietokannan hallintajärjestelmänä MySQL, palvelinpuolen komponenttiarkkitehtuurina Enterprise JavaBeans (EJB) 3.1, dynaamisten verkkosivujen rakentamiseen JavaServer Faces (JSF) 2.0 -sovelluskehys sekä selainpuolen skriptikielenä Asynchronous JavaScript and XML (AJAX).

Avainsanat: J2EE6, EJB3.1, JSF 2.0, AJAX, MySQL

ABSTRACT

Name: Mika Kiiskinen

Title: Web store implemented with J2EE6

Date: 04.10.2010 **Number of pages:** 47 p + 1 appendix

Department: Information technology **Study Programme:** Programming

Instructor: Simo Silander

Supervisor: Aleksi Lappalainen

The goal for this final thesis was to program a web store application based on basic information given by a client this project was made for. The idea was to choose well known and common technologies which would serve well both my client and my personal learning. The client's prerequisites included elaborate appearance, usability, easy maintenance through a graphical user interface, security and open source programs. The technologies were due to interoperate well on a chosen platform and server. We also agreed with the instructor of the project to make the application implement the MVC-model and make changing the applications language easily. In addition to that, I wanted to compare the new J2EE6-spesification with the older J2EE5-spesification.

The technologies chosen for the project are commonly known among web designers. The programming language is platform-independent Java, the database management system is MySQL, server side component architecture is Enterprise JavaBeans (EJB) 3.1, the web application framework is JSF 2.0 and client side script language is AJAX.

Keywords: J2EE6, EJB3.1, JSF 2.0, AJAX, MySQL

SISÄLLYS

1 JOHDANTO

1.1 Alexis Oy:n esittely

Alexis Oy on firma, jota ei verkkokauppaprojektin suunnittelu- ja luontivaiheessa ollut vielä olemassa. Sen perustajalla, Aleksi Lappalaisella, oli liikeidea olemassa, mutta varsinainen yritys perustettiin vasta projektin loppuvaiheessa. Tämä on sikäli luontevaa, että koska yrityksen koko toiminta keskittyy tuotteiden myyntiin verkossa, ei yrityksellä ole juurikaan virkaa ennen verkkokauppasovelluksen valmistumista. Lisäksi yrityksen alkuunlaittamiselle ei mm. projektin tekohetkellä vallinneen maailmantalouden alavireisen tilanteen vuoksi ollut erityistä kiirettä.

Kyseinen projekti Alexis Oy:lle on yrityksen rakenteen vuoksi kertaluonteinen, mutta se tullee jatkossakin työllistämään tämän insinöörityön tekijää, Mika Kiiskistä, ylläpito- ja jälleenkehitystehtävien parissa.

1.2 Verkkokauppaprojektin määrittely

Asiakkaan (Alexis Oy) projektille etukäteen ilmoittamat vaatimukset olivat

- ulkonäön selkeys ja siisteys. Sivuston tulee erottua massasta visuaalisella tyylikkyydellään ja olla komponenttien sijoittelultaan selkeä.
- helppokäyttöisyys. Kaikkien toimintojen logiikka yhdistettynä ulkoasuun tulee olla niin selkeätä, että kokematonkin käyttäjä kykenee vaivattomasti käyttämään sivuston toimintoja.
- helppo ylläpito. Ylläpitäjän pitää pystyä lisäämään, poistamaan ja päivittämään tietueita graafisen käyttöliittymän avulla ilman käytettyjen teknologioiden tuntemusta.
- tietoturva. Käyttäjien antamien tietojen tulee pysyä salassa ja heidät tulee vakuuttaa siitä, että näin myös tapahtuu.
- maksuttomuus. Käytettyjen ohjelmien ja teknologioiden lähdekoodien tulee olla avoimia (open source).
- tehokkuus. Valittujen teknologioiden sekä palvelimen tulee toimia tarpeeksi tehokkaasti, jotta sovelluksen käyttäminen on sujuvaa ja käyttäjien kokemat vasteajat lyhyitä.

Lisäksi sovimme insinöörityön ohjaajan Simo Silanderin kanssa, että työ tehdään MVC-mallin mukaisesti. Myös sovelluksen käyttökieli tuli olla vaihdettavissa helposti, kaikki sivuilla esitetty teksti tuli hakea tietokannasta ja sen tulee löytyä sekä englanniksi että suomeksi ja kielen vaihdon tulee onnistua käyttäjältä yhtä nappia painamalla.

Omaksi oppimistavoitteekseni määritin lisäksi uuden J2EE6-spesifikaation vertaamisen laajasti käytettyyn edeltäjäänsä J2EE5-spesifikaatioon.

2 PROJEKTISSA KÄYTETYT TEKNOLOGIAT

Tässä luvussa esitellään sovelluksen toteutuksessa käytetyt teknologiat pääpiirteittäin. Niitä täsmennetään myöhemmin, kun esitellään itse sovelluksen rakenne. Teknologiat on valittu projektin määrittelyn huomioimisen jälkeen kahdella kriteerillä: niiden tulee olla yleisesti käytettyjä, hyviksi ja toimiviksi havaittuja sekä niiden tuli tukea insinöörityön tekijän omaa oppimista.

2.1 Java 2 Platform Enterprise Edition 6 (J2EE6)

Projektin ohjelmointikieltä valitessa huomioitavia seikkoja olivat tuki olio-ohjelmoinnille sekä mahdollisuus web-pohjaisen käyttöliittymän luomiseen. Olio-ohjelmointi oli välttämättömyys, sillä perinteisillä, proseduaalisilla, ohjelmointikielillä sovelluksen toteuttaminen olisi ollut vähintään työlästä, ellei mahdotonta. Aikaisempaa ohjelmointikokemusta löytyi sekä Javasta että C++:sta ja niistä vaaka kallistui selkeästi Javan puolelle kielen käyttäjäystävällisyyden takia: C++:lla jää ohjelmoijan omalle vastuulle paljon sellaista, mistä Javan systeemi (käytännössä Java Virtual Machine, JVM) pitää ohjelmoijan puolesta huolen. Javalla ohjelmoituja sovelluksia on pidetty hitaampina kuin C++:lla ohjelmoituja, koska jälkimmäisellä monia asioita pystyy optimoimaan paremmin käyttäjäystävällisyyden kustannuksella. Palvelinten ja tietokoneiden teho ja nopeus ovat kuitenkin nykyisin niin suuria, ettei Javan hieman hitaamman suorituskyvyn katsottu olevan riittävä syy sen hylkäämiselle.

Ohjelmointikielen valinnan jälkeen kehitysalustan valinta oli helppo tehtävä. Olin jo aiemmin opiskellut J2EE:tä jonkin verran ja tiesin sen tukevan kaikkea sitä toiminnallisuutta, jota projektissani tulisin tarvitsemaan. Tuli siis loistava tilaisuus testata jo hankitut taidot käytännössä ja ennen kaikkea

parantaa niitä huomattavasti. Myös alan ammattilaisina työskentelevien tuttavien keskuudessa järjestetty kyselytutkimus puolsi valintaani. J2EE:stä julkaistiin uusi versio (J2EE6) joulukuussa 2009 [1.], ja koska halusin selvittää sen mukanaan tuomat muutokset vanhaan J2EE5:een [2.] verrattuna, päätin toteuttaa projektin käyttäen uudempaa versiota sekä omistaa näiden versioiden eroavaisuuksien tutkimustuloksille oman kappaleensa.

2.2 MySQL 5.5

Tietokantatyypiksi valittiin perinteinen relaatiotietokanta, jonka tyyppisiä valtaosa kaikista tietokannoista on. Relaatiotietokanta on tiedontallennusvarasto, jota hallinnoidaan tietokantojen hallintajärjestelmillä. Näitä hallintajärjestelmiä on olemassa lukuisia, niin maksuttomia kuin maksullisiakin. Ilmaisversioista tunnetuin ja suosituin on MySQL. Sitä käytetään yleisesti kaupallisissa sovelluksissa ja sen käyttöön on olemassa kattavat tuki- ja ohjesivustot. Lisäksi MySQL on hyvin skaalautuva, eli jos luodun sovelluksen käyttäjämäärät tulevaisuudessa kasvavat suuriksikin, kykenee se silloinkin hoitumaan tehtävistään. Näiden seikkojen vuoksi päädyttiin valitsemaan tietokannan hallintajärjestelmäksi MySQL ja siitä tuorein versio 5.5 [5.].

2.3 Enterprise JavaBeans 3.1 (EJB)

Seuraavaksi tarvittiin väline, jolla saataisiin tietokantojen sisältämä data muutettua Java-kielisiksi luokiksi ja joka tukisi verkkosovelluksen vaatimuksia. Ensiksi mainitulla prosessilla on myös englanninkielinen nimi ”object-relational mapping” (ORM, O/RM tai OR-mapping). Tarve ORM:lle syntyy siitä, että relaatiotietokannat ja niiden tietueet rakentuvat eri kielellä kuin niitä vastaavat Java-luokat (nk. impedance mismatch) ja näiden välille pitää luoda vastaavuussuhteet. Perinteisesti tämä ”mäppäys” on hoidettu J2EE:n tarjoaman JDBC-nimisen (Java Database Connectivity) sovellusohjelmarajapinnan (Application Programming Interface, API) kautta, jolloin tiedot viedään tietokantaan ja haetaan sieltä Java-kieleen upotettujen SQL-lauseiden avulla. Tämä on kuitenkin paitsi työlästä, niin myös varsin tehotonta; tietokantakutsut vievät aikaa ja on suurin yksittäinen pullonkaula sovelluslogiikassa. Tätä ongelmaa on yritetty ratkoa mm. erilaisten suunnittelumallien avulla, jotka pääpiirteissään kasaavat useamman

tietokantakutsun yhteen pakettiin ja lähettävät ne sitten kaikki kerralla tietokantaan. Nämä mallit ovat parantaneet tehokkuutta, mutteivat merkittävästi ohjelmoinnin työläyttä. [6.]

Verkkosovelluksen vaatimuksilla tarkoitetaan sitä, että tietokannoista tulevaa ja sinne menevää dataa pitää pystyä muokkaamaan, ts. sovelluksen käyttäjän näkemien verkkosivujen ja tietokannan välille tarvitaan sovelluslogiikkaa, joka määrää, mikä data menee minne, milloin ja miten.

Kaikkiin yllä mainittuihin ongelmiin ja vaatimuksiin on onneksi kehitetty ratkaisuksi erilaisia sovelluskehyksiä kuten Hibernate ja EJB. Niiden perusidea on se, että ne automatisoivat monet aikaisemmin ohjelmoijan vastuulle jääneet tehtävät, kuten edellä mainittujen tietokantakutsujen hienosäädön, mutta sen lisäksi lukuisan joukon muitakin tehtäviä. Sekä Hibernate että EJB perustuvat Java-kieleen ja käytettävän sovelluskehyksen valinta käytiinkin niiden välillä. Molemmat olivat entuudestaan melko tuntemattomia, ja niihin tutustuttuani pidin molempia varteenotettavina vaihtoehtoina. J2EE6 toi kuitenkin tullessaan uuden ja pitkään odotetun version myös EJB:stä versiolla EJB3.1, joten projektin uutta ja vanhempaa teknologiaa vertailevan luonteen vuoksi se tuli lopulta valituksi [8.]

Kaikki EJB:n tarjoama toiminnallisuus tapahtuu EJB container –nimisessä säiliössä. Kaikki asiakkaalta sovellukselle tulevat metodikutsut menevät tälle säiliölle, joka lähettää ne edelleen oikealle rajapintaluokalle; metodeita ei koskaan kutsuta suoraan. EJB-säiliö on perinteisesti ollut tietoinen sovelluksen sielunelämästä deployment descriptor –nimisten XML-tiedostojen avulla. Näissä tiedostoissa on määritelty kaikki sovelluksen OR-mäppäykseen, tietoturvaan, transaktioihin yms. liittyvät tiedot. Vähänkin laajemman sovelluksen deployment descriptor – tiedostoista tulee varsin pitkiä ja monimutkaisia ja niiden ylläpitäminen on puhtaaseen Java-koodiin tottuneelle työlästä. Yksi EJB 3.x -versioiden näkyvimmistä uudistuksista vanhempiin versioihinsa nähden ovat annotaatiot (annotation). Annotaatiot ovat selkokielisiä, Java-koodin sekaan kirjoitettavia merkintöjä, joilla voi korvata suurimman osan deployment descriptor -tiedostojen sisällöstä; EJB-säiliö osaa tulkita annotaatiot XML-tiedostojen sisältämän datan korvikkeina. [6; 7.] Ohessa esimerkki annotaation koodausta helpottavasta vaikutuksesta.

Tietokannan Product-taulua vastaavan Product-luokan alun koodaaminen annotaatioiden avulla:

```
@Entity
@Table(name = "product")
public class Product implements Serializable {
    @Id
    @Basic(optional = false)
    @Column(name = "ID")
    protected Integer id;
```

Kuva 1. Product-luokan Java-kielistä määrittelyä

Sitä vastaava XML-määritys deployment descriptorissa olisi:

```
<entity-mappings>
    <entity class="Webstore.Entity.Product.Product" access="PROPERTY">
            <table name="Product">
               <attributes>
                <id name="id">
                column name="product_ID"
                nullable="false"
                column-definition="integer"/>
                </id>
        </attributes>
    </entity>
</entity-mappings>
```

Kuva 2. Product-luokan XML-kielistä määrittelyä

Enterprise Java Beans koostuu nimensä mukaisesti ”beaneista” eli pavuista. Pavut ovat tavallisia Java-luokkia (Plain Old Java Object, POJO) ja niitä on kahdenlaisia: tietokannan tietueita mallintavia tietokantapapuja (Entity beans) ja mm. niiden välisen sovelluslogiikan määritteleviä Enterprise-papuja (Enterprise beans).

2.3.1 Tietokantapavut

Tietokantapapujen avulla hoidetaan edellä mainittu OR-mäppäys. Entity-luokka vastaa yleensä yhtä tietokannan taulua ja sisältää taulun kenttiä vastaavat jäsenmuuttujat. Tietokantapavut toimivat siis datan varastoijina, mutta ne eivät varsinaisesti tee datalle mitään eivätkä toteuta mitään sovelluslogiikkaa. Entity-luokka merkitään @Entity-annotaatiolla.

2.3.2 Enterprise-pavut

Enterprise-papuja on kahta tyyppiä, jotka eroavat toisistaan merkittävästi: istunto-papu (session bean) ja viestipapu (message-driven bean, MDB). Istunto-pavut toteuttavat tietokantapapujen välille halutun toiminnallisuuden, joka määrittää sovelluksen logiikan; ne sisältävät nk. business-metodeja

(business methods), joita voidaan käyttää lukuisten erilaisten hajautettujen olioprotokollien (distributed object protocols) avulla. MDB:t prosessoivat asynkronisesti viestejä (messages), jotka tulevat systeemeistä kuten JMS (Java Message Service), perinnejärjestelmät (legacy system) ja Web Services (WS).

Vielä EJB 3.0 –spesifikaatio määräsi jokaisen istuntopavun toteuttamaan vähintään yhden rajapinnan, jonka kautta sen metodeihin pääsi käsiksi. EJB 3.1 -spesifikaation myötä tämä sääntö poistui paikallisen rajapinnan osalta ja koska useimmiten istuntopapuja käyttää samassa EE-säiliössä sijaitseva faceletti tai servletti, ei papujen tarvitse välttämättä toteuttaa yhtään rajapintaa (kuten ei tässäkään projektissa). Kyseiset rajapinnat ovat

- paikallinen rajapinta (local interface), joka määrittelee istuntopavun business-metodit, joita voivat käyttää muut istuntopavut saman EE-säiliön (EE container) sisällä, ts. saman JVM:n alla. Metodien suorittaminen on nopeata. Paikallinen rajapinta merkitään @Local-annotaatiolla, mutta se ei ole enää pakollinen EJB 3.1:n myötä.
- etärajapinta (remote interface), joka määrittelee istuntopavun business-metodit, joita voivat käyttää EJB-säiliön ulkopuoliset sovellukset. Sitä merkitään @Remote-annotaatiolla.
- päätepisterajapinta (endpoint interface), joka määrittelee istuntopavun business-metodit, joita voidaan kutsua EJB-säiliön ulkopuolelta SOAP:n (Simple Object Access Protocol) tai WSDL:n (Web Services Description Language) kautta. Sillä ei ole omaa annotaatiota.
- viestirajapinta (message interface), jonka toteuttavat MDB:t kuuntelevat muilta pavuilta tulevia viestejä ja vastailevat niihin. Tämä on vanhentunut teknologia, jota käytetään lähinnä perinnesysteemien kanssa. Viestirajapinnalle ei ole annotaatiota, MDB:t puolestaan merkitään @MessageDriven-annotaatiolla.

Istuntopapuja on puolestaan kahdenlaisia: tilaton (stateless) ja tilansa säilyttävä (stateful). Kuten nimet jo paljastavatkin, ne eroavat toisistaan siinä, kuinka ne säilyttävät tilansa (state).

Tilaton istuntopapu (stateless session bean)

Tilaton istuntopapu (ti) ei säilytä tilaansa kutsujen välillä. Tämä tarkoittaa sitä, että kun ti:tä kutsutaan, suorittaa se kutsutun metodin ja sen jälkeen

mahdolliset jäsenmuuttujien arvot eivät jää papu-ilmentymän (instance) muistiin vaan kyseinen ilmentymä on vapaa suorittamaan jonkun toisen asiakkaan (client) kutsun. Näin ollen ti soveltuu yksinkertaisten, yhdellä metodilla toimitettavien tehtävien hoitoon (esimerkkinä tietueen lisääminen tai poisto). Koska ti:t eivät ole millään lailla sidoksissa tiettyyn asiakkaaseen, voi ti-ilmentymiä olla huomattavasti vähemmän kuin asiakkaita. Tätä kutsutaan instance pooling –tekniikaksi ja EJB-säiliö käyttää sitä hyväkseen luodessaan ti-instansseja; instansseja luodaan tietty määrä reserviin ja niitä osoitetaan asiakkaiden käyttöön näiden kutsuttua ti-metodeja. Reservin kokoa voidaan vaihdella kutsujen määrän mukaan, jolloin reservi on aina optimaalisen kokoinen. Tilaton istuntopapuluokka merkitään @Stateless-annotaatiolla. [7.]

Tilansa säilyttävä istuntopapu (stateful session bean)

Tilansa säilyttävä istuntopapu (tsi) puolestaan säilyttää tilansa metodikutsujen välillä, jolloin siis tsi-ilmentymä on sitä kutsuneen asiakkaan käytössä, kunnes ilmentymä on suorittanut loppuun kaiken toiminnallisuutensa. Ne soveltuvat monimutkaisempiin, useita metodikutsuja sisältäviin toimintoihin (esimerkkinä ostoksen luominen, jolloin useilla eri metodeilla lisätään tietokantaan tiedot mm. asiakkaasta ja ostoksesta). Tsi:n osoittaminen asiakkaalle ei kuitenkaan tarkoita sitä, että sama tsi-ilmentymä välttämättä palvelisi tiettyä asiakasta äärettömiin, mikäli tämä esimerkiksi jättää tilauksensa kesken. Tsi:in tapauksessa EJB-säiliö ei käytä instance poolingia, koska tällöinhän ilmentymien sisältämä tieto menetettäisiin sen siirtyessä palvelemaan toista asiakasta. Sen sijaan säiliö käyttää nk. aktivointimekanismia (activation mechanism), jolloin ilmentymä passivoidaan jos sen toiminnallisuutta ei suoriteta loppuun tietyssä määräajassa ja aktivoidaan, kun toimintaa jatketaan. Käytännössä tämä tarkoittaa sitä, että kun ilmentymä passivoidaan, sen tila (jäsenmuuttujien arvot) serialisoidaan, ts. talletetaan tilapäismuistiin, ja itse ilmentymä poistetaan muistista eli tuhotaan. Tämä on aktivointi-mekanismin tapa säästää resursseja aivan kuten instance pooling tekee tsi:jen tapauksessa. Aktivoitaessa ilmentymä EJB-säiliö luo kokonaan uuden ilmentymän ja asettaa passivoitaessa serialisoidun tilan ilmentymän tilaksi.

Tilansa säilyttävä istuntopapuluokka merkitään @Stateful-annotaatiolla. [7.]

2.4 JSF ja MVC-arkkitehtuuri

Sovelluksen haluttiin toteuttavan MVC-arkkitehtuurin, joka tarkoittaa sitä, että ohjelmakoodi jaetaan toiminnallisuutensa perusteella useampaan kerrokseen (layer), tässä tapauksessa kolmeen. Mallikerros (model) huolehtii datan tallentamisesta tietokantaan ja hakemisesta tietokannasta ja se pitää sisällään EJB-luokat. Näkymäkerros (view) vastaa käyttäjän näkemästä näkymästä, eli palvelimen selaimille lähettämien verkkosivujen muodostamasta käyttöliittymästä. Kontrolleri-kerros (controller) huolehtii sovelluslogiikasta kahden edellä mainitun kerroksen välillä, ja se on perinteisesti koostunut servleteistä. Uuden JSF 2.0 –spesifikaation myötä tämä jako on kuitenkin hieman hämärtynyt; voidaan katsoa, että backing-bean–luokat ovat osa kontrollerikerrosta, koska sen ainoana tehtävänä on toimia rajapintana näkymän ja mallin välillä.

Menneinä vuosina MVC-arkkitehtuurin toteuttaminen kooditasolla jäi kokonaan sovelluksen kehittäjän harteille [2]. Sittemmin on kehitetty lukuisia web-sovellusten tekoon tarkoitettuja sovelluskehyksiä, jotka helpottavat MVC-arkkitehtuurin toteuttamista huomattavasti. Näistä tunnettuja ovat ainakin Struts, Tapestry, Velocity, Webwork, Wicket ja JSF [3]. Näistä Struts ja JSF päätyivät tässä työssä toiselle kierrokselle sovelluskehyksen valintaa tehdessä. Struts on kehyksistä vanhempi, mutta sitäkin on ajan kuluessa päivitetty ja sen viimeisin versio, Shale, on itse asiassa rakennettu JSF:n päälle. J2EE6-spesifikaatio toi mukanaan uuden version JSF:sta (JSF2.0), joten se tuoreempana sekä tarkasteltavaan spesifikaatioon kuuluvana tuli valituksi sovelluskehykseksi.

Siinä missä aiemman JSF 1.2 –version näkymät toteutettiin JavaServer Pages (JSP) –sivuilla, toteutetaan uuden JSF 2.0 –version mukaiset sivut oletusarvoisesti faceleteilla (facelet). Niiden avulla luodut XHTML-sivut ovat luojalleen ulkoasultaan selkeämpiä kuin aiemmat JSP + HTML –sivut. Ne ovat XML-pohjaisia, jonka vuoksi sivun elementit muodostavat tiukan puurakenteen eli DOM-mallin (domain object model). Tämän avulla sivujen elementteihin päästään helposti käsiksi XML:stä tuttujen mekanismien avulla. Vanhan JSF 1.2 –version käyttäminen oli myös luvattoman monimutkaista: erilaisia tiedostoja tarvittiin paljon, niiden luominen oli kömpelöhköä ja XML-kieliset deployment descriptorit olivat kookkaammissa sovelluksissa massiivisia. JSF 2.0:ssa ei XML-tiedostoihin tarvitse lisätä

halutessaan merkkiäkään, kaiken voi EJB:n tapaan hoitaa annotaatioiden ja oletusarvojen avulla.

Muuten JSF:n perustoiminta on kovasti entisen kaltaista. Perusideana on, että sovelluskehittäjä luo verkkosivut lisäämällä XHTML-sivuille valmiita JSF-tagikirjaston (JSF tag library, JSTL) määrittelemiä JSF-tageja. Kukin tagi hoitaa jonkin toiminnallisuuden, useimmiten ne vastaavat jotakin tiettyä HTML-komponenttia: tällöin kyseiset JSF-tagit muutetaankin vastaaviksi HTML-komponenteiksi käyttäjän lähetettyä sivun takaisin serverille. Tageilla hoidetaan myös monia muita käyttäjän antamiin syötteisiin ja sivujen ulkonäköön liittyviä toimintoja, kuten validoinnit ja lokalisoinnit. Mikäli valmiiden tagien tuoma toiminnallisuus ei sovelluskehittäjälle riitä, voi tämä melko vaivattomasti luoda myös omia kustomoituja tagejansa.

Kun käyttäjä lähettää sivun takaisin serverille esim. täytettyään jonkin lomakkeen, hoitaa EE-säiliö aina tietyt toimenpiteet saman syklin mukaan (life-cycle phases). Näiden aikana lähetetyn sivun tila tallennetaan, syöttökenttien arvot noudetaan, arvoille suoritetaan validoinnit, arvot mahdollisesti tallennetaan tietokantapapuihin, arvoilla suoritetaan mahdolliset metodit, päivitetään käyttäjälle lähetettävän sivun tila ja lopulta, jos missään vaiheessa ei esiintynyt virheitä, lähetetään se käyttäjälle. Virheen tapauksessa lähetetään luonnollisesti virheilmoitussivu.

Käyttäjän syöttämien tietojen tallennus tapahtuu backing-papujen avulla. Ne ovat tätä nimenomaista tarkoitusta varten luotuja papuluokkia, jotka paitsi ottavat vastaan syötteitä, myös suorittavat toimintoja niiden perusteella. Ne mm. vievät syötetyt tiedot tietokantaan ja määräävät, mikä sivu asiakkaalle kunkin toiminnon perusteella lähetetään (navigointi). JSF 2.0 toi mukanaan tukun backing-papujen käyttöä helpottavia annotaatioita sekä implisiittisen navigoinnin (implicit navigation), jonka avulla navigointi hoidetaan Java-luokissa määritetyin metodein XML-tiedostojen sijaan.

Kuten mainittua, kontrollerin tehtäviä hoitavat valituilla teknologioilla backing-papujen ohella servletit. Ohjelmoijan ei niille nykyisin tarvitse tehdä itse mitään. Kaikki näkymän ja mallin välinen toiminnallisuus hoidetaan oletusservletin eli facesServletin avulla, ja sen toiminta jää EE-säiliön harteille.

2.5 AJAX

Sovellukseen haluttiin mukaan näyttävyyttä ja käyttäjäystävällisyyttä, ja niitä toteuttamaan valittiin Asynchronous JavaScript and XML eli Ajax. Ajaxin toimintaperiaate on se, että sen avulla haluttuja sivun osia voidaan päivittää päivittämättä koko sivua, jolloin käyttäjä kokee kyseisten toimintojen tapahtuvan ns. ”lennossa”. Kuten koko nimensä paljastaa, kaikki toiminnallisuus tapahtuu asynkronisesti eli taustalla, jolloin käyttäjä voi esimerkiksi jatkaa lomakkeen täyttöä, vaikka yhteys tietokantaan olisi hetkellisesti jumiutunut. Ajaxin toiminnallisuus on perinteisesti saatu aikaan selainpuolella toimivien JavaScript-kielisten skriptien avulla, mutta JSF 2.0:llä on oma taginsa, jonka avulla Ajax-toiminnallisuus saadaan aikaiseksi ilman skriptejäkin.

3 VERKKOKAUPPASOVELLUKSEN RAKENNE

3.1 Yleinen toiminta

Verkkokauppasovelluksen käytännön toteutus koostuu XHTML-sivujen muodostamasta käyttöliittymästä sekä dataa käsittelevistä ja tietokantajärjestelmälle ja -järjestelmästä siirtävistä EJB-luokista ja backing-papu-luokista. Lisäksi datan varastona toimii MySQL-tietokanta.

3.2 Mallikerros

3.2.1 Tietokanta

Tietokannassa on yhteensä kolme taulua, jotka ovat Product-, Customer- ja Purchase-taulu, joiden välillä istuntopavut hallinnoivat sovelluslogiikkaa. Tuotteet (Product) on jaettu kahteen eri kategoriaan, vaatteisiin (clothes) ja välineisiin (equipment). Ne siis molemmat sisältyvät tuotteet-tauluun ja jako niiden välillä esim. verkkosivuilla esitettäessä hoidetaan ohjelmallisesti mm. kyselykielisten (query language, QL) lauseiden avulla; alakategoriat merkitään tuote-tauluun productType-nimisellä tietueella. Vaatteet jakautuvat vielä useampiin alakategorioihin, ja niiden välinen jako hoidetaan kuten alituotteidenkin. Asiakas-taulu (Customer) sisältää asiakkaan henkilötiedot ja ostos-taulu (Purchase) asiakkaiden tekemien ostosten tiedot.

Kolmen taulun taktiikkaan päädyttiin puhtaasti suorituskyvyllisistä syistä; se toimii nopeimmin. Varteenotettava vaihtoehto olisi ollut sijoittaa kaikki

alituotteet omiin tauluihinsa, jolloin kaikki osakokonaisuudet olisivat sijainneet siististi omissa paketeissaan (niin tietokannan tauluissa kuin Java-luokissakin), mutta mitä enemmän kannassa on tauluja, sitä hitaammaksi niiden välisten kyselyjen suorittaminen käy. Tässä sovelluksen toiminnan aikakriittisessä kohdassa päädyttiin siis arvostamaan enemmän tehoja kuin selkeyttä koodaajalle.

Ostos-taulu määrittää asiakkaan tekemän ostoksen, joka sisältää tämän verkkosivuilta valitsemat tuotteet. Tuotteita voi ostokseen sisältyä luonnollisesti useampiakin, joten ostoksen ja tuotteen välinen yhteys on muotoa yhden-suhde-moneen (1:N). Ostoksella puolestaan voi olla vain yksi asiakas, joten ostoksen ja asiakkaan välinen suhde on muotoa 1:1. Viitteet lähtevät ostos-taulusta, koska asiakas ja tuote voivat olla olemassa ilman ostostakin, mutta ostos ei ilman asiakasta ja tuotetta.

Tuotteita on tietokannassa yksi kutakin erilaista tuotetta kohden. Paino on sanalla erilainen; samaa mallia olevat verkkarihousut voivat jakaantua esimerkiksi koon ja värin mukaan lukuisiin erilaisiin yksilöihin, joista kukin saa tietokannassa oman pääavaimensa eli ID-numeron. Jokainen yksilö ei siis saa omaa ID-numeroa, joka olisi vaihtoehtoinen tapa toteuttaa tietokanta. Tästä seuraa, että kunkin tuotteen lukumäärää tietokannassa ei saada suoraan tekemällä count-tyyppistä tietokantakyselyä (sehän antaisi vastaukseksi aina 1), vaan tarvitaan jokin muu mekanismi seuraamaan varastosaldoa. Tämä on hoidettu antamalla tuotteelle inStock-niminen tietue, jota päivitetään aina, kun joko asiakas ostaa tuotteen (tai tarkemmin sanottuna lisää sen ostoskoriin) tai ylläpitäjä lisää tuotetta varastoon.

3.2.2 EJB-luokat

Entity-luokat

Entity-luokat ovat siis tietokannan tauluja mallintavia Java-luokkia. Niiden rakennetta ja toimintaa esitellään edellisessä kappaleessa esitettyjä tietokannan tauluja vastaavien luokkien avulla. [6; 7.]

Product-luokka

```
1   package Entity;
2
3   import java.io.Serializable;
4   import javax.persistence.Basic;
5   import javax.persistence.Column;
6   import javax.persistence.Entity;
7   import javax.persistence.GeneratedValue;
8   import javax.persistence.GenerationType;
9   import javax.persistence.Id;
10  import javax.persistence.NamedQueries;
11  import javax.persistence.NamedQuery;
12  import javax.persistence.Table;
```

Kuva 3 Product-luokan import-määreet

Kuvan 3 import-luokista nähdään, että tärkeimmät datan tietokantaan talletukseen eli persistointiin liittyvät luokat sijaitsevat javax.persistence-paketissa.

```
18  @Entity
19  @Table(name = "product")
20  @NamedQueries({
21      @NamedQuery(name = "Product.findAll", query = "SELECT p FROM Product p"),
22      @NamedQuery(name = "Product.findById", query = "SELECT p FROM Product p WHERE p.id = :id"),
23      @NamedQuery(name = "Product.findByProductType", query = "SELECT p FROM Product p WHERE p.productType = :productType"),
24      @NamedQuery(name = "Product.findByPrice", query = "SELECT p FROM Product p WHERE p.price = :price"),
25      @NamedQuery(name = "Product.findByModel", query = "SELECT p FROM Product p WHERE p.model = :model"),
26      @NamedQuery(name = "Product.findByManufName", query = "SELECT p FROM Product p WHERE p.manufName = :manufName"),
27      @NamedQuery(name = "Product.findByIsReduced", query = "SELECT p FROM Product p WHERE p.isReduced = :isReduced"),
28      @NamedQuery(name = "Product.findByInStock", query = "SELECT p FROM Product p WHERE p.inStock = :inStock"),
29      @NamedQuery(name = "Product.findByGender", query = "SELECT p FROM Product p WHERE p.gender = :gender"),
30      @NamedQuery(name = "Product.findByDescription", query = "SELECT p FROM Product p WHERE p.description = :description"),
31      @NamedQuery(name = "Product.findBySubProduct", query = "SELECT p FROM Product p WHERE p.subProduct = :subProduct"),
32      @NamedQuery(name = "Product.findByImageUrl", query = "SELECT p FROM Product p WHERE p.imageUrl = :imageUrl"),
33      @NamedQuery(name = "Product.findProductTypes", query = "SELECT DISTINCT prod.productType FROM Product prod ")})
34  public class Product implements Serializable {
35
```

Kuva 4. Product-luokan määritteitä

Kuvassa 4 luokan tyyppi määritetään @Entity-annotaatiolla. Annotaatio kertoo EJB-säiliölle, että kyseessä on Entity-luokka, joka halutaan liittää säiliöön. @Table-annotaatio mäppää Product-luokan product-nimiseen tauluun tietokannassa.

Riveillä 20-32 on IDE:n generoimia nimettyjä JPQL-kielisiä (Java Persistence Query language) tietokantakyselyjä, jotka määritellään @NamedQuery- ja @NamedQueries-annotaatioilla. Riville 33 on lisätty yksi kustomoitu nimetty tietokantakysely, jota tarvitaan sivustojen ylläpitäjälle tarkoitetun toiminnalisuuden luomisessa. Nimetyt kyselyt ovat käteviä silloin, kun tiettyä kyselyä tarvitaan useissa paikoissa sovelluksessa, koska tällöin ne voidaan suorittaa missä vain pelkästään viittaamalla kyselyn nimeen

ilman työlään kyselykielisen lauseen toistamista. Tämä lisää uudelleenkäytettävyyttä ja parantaa ylläpidettävyyttä. [7.]

```
36      private static final long serialVersionUID = 1L;
37      @Id
38      @GeneratedValue(strategy = GenerationType.IDENTITY)
39      @Basic(optional = false)
40      @Column(name = "ID")
41      private Integer id;
42      @Basic(optional = false)
43      @Column(name = "productType")
44      private String productType;
45      @Basic(optional = false)
46      @Column(name = "price")
47      private double price;
48      @Basic(optional = false)
49      @Column(name = "model")
50      private String model;
51      @Basic(optional = false)
52      @Column(name = "manufName")
53      private String manufName;
54      @Basic(optional = false)
55      @Column(name = "isReduced")
56      private boolean isReduced;
57      @Basic(optional = false)
58      @Column(name = "inStock")
59      private int inStock;
60      @Basic(optional = false)
61      @Column(name = "gender")
62      private String gender;
63      @Column(name = "description")
64      private String description;
65      @Basic(optional = false)
66      @Column(name = "subProduct")
67      private String subProduct;
```

Kuva 5. Product-luokan jäsenmuuttujien määrittelyä (1/2)

Kuvassa 5 määritellään luokan jäsenmuuttujia. Pääavain määritetään @ID-annotaatiolla ja @Column-annotaatiolla määritetään, kuinka muuttuja mäpätään tietokannan tauluun; name-atrubuutin lisäksi voidaan määritellä mm. unique-, nullable-, insertable- ja updatable-atribuutit. Pääavain on Integer-tyyppiä ja se luodaan automaattisesti uudelle Product-luokan ilmentymälle @GeneratedValue-annotaation avulla. @Basic-annotaatiolla voidaan määritellä muuttujalle kaksi ominaisuutta: Null-arvojen salliminen optional-määreellä sekä FetchType-nimisen määreen asettaminen (EAGER tai LAZY, oletuksena EAGER) [6; 7.]. Muut jäsenmuuttujat määritellään edellä mainituilla annotaatioilla. Kuvassa 6 esitellään loput Product-luokan jäsenmuuttujien määrittelyt.

```
68        @Basic(optional = false)
69        @Column(name = "imageUrl")
70        private String imageUrl;
71        @Basic(optional = true)
72        @Column(name = "color")
73        private String color;
74        @Basic(optional = true)
75        @Column(name = "sizes")
76        private String sizes;
77        @Basic(optional = true)
78        @Column(name = "clothesType")
79        private String clothesType;
80        @Basic(optional = true)
81        @Column(name = "blength")
82        private int blength;
83        @Basic(optional = true)
84        @Column(name = "width")
85        private int width;
86        @Basic(optional = true)
87        @Column(name = "material")
88        private String material;
89
```

Kuva 6. Product-luokan jäsenmuuttujien määrittelyä (2/2)

```
90      public Product() {
91      }
92
93      public Product(String productType, double price, String model,
94              String manufName, boolean isReduced, int inStock,
95              String gender, String desc, String subProduct, String imageUrl) {
96          this.productType = productType;
97          this.price = price;
98          this.model = model;
99          this.manufName = manufName;
100         this.isReduced = isReduced;
101         this.inStock = inStock;
102         this.gender = gender;
103         this.description = desc;
104         this.subProduct = subProduct;
105         this.imageUrl = imageUrl;
106     }
107
108     public Integer getId() {
109         return id;
110     }
111
112     public String getProductType() {
113         return productType;
114     }
115
116     public void setProductType(String productType) {
117         this.productType = productType;
118     }
```

Kuva 7. Product-luokan konstruktorit ja aksessorit

Yksi Entity-luokan neljästä pakollisesta metodista on parametriton konstruktori (kuva 7). Luokalle luotiin myös toinen konstruktori, jolla asetetaan arvot kaikille luokan jäsenmuuttujille lukuunottamatta id-muuttujaa, koska sen arvo generoidaan automaattisesti tietokannanhallintajärjestelmän toimesta eikä sitä näin ollen voida asettaa käyttäjän toimesta. Jokaiselle jäsenmuuttujalle on luotu myös aksessorit,

joista kuvassa 7 on esitetty id:n ja productTypen vastaavat (id:ltä puuttuu set-metodi samasta syystä kuin parametrillisen konstruktorin tapauksessa). Kaikille muillekin kuvissa 5 ja 6 esitellyille jäsenmuuttujille luodaan samanlaiset aksessorit.

Kuvassa 8 on esitetty Product-luokan metodit, joiden avulla päivitetään inStock-laskuria.

```
117      public void addInStock(int pcs) {
118          this.inStock = +pcs;
119      }
120
121      public void subtractInStock(int pcs) {
122          this.inStock = -pcs;
123      }
124
```

Kuva 8. inStock-laskuria päivittävät metodit

Kuvassa 9 on esitetty kolme muuta Entity-luokalle pakollista metodia. Ne ovat hashCode(), equals() ja toString() ja ne on merkitty @Override-annotaatiolla korvaamaan Object-yliluokan määrittelemät saman nimiset oletusmetodit. [6; 7.]

```
244      public void setWidth(int width) {
245          this.width = width;
246      }
247
248      @Override
249      public int hashCode() {
250          int hash = 0;
251          hash += (id != null ? id.hashCode() : 0);
252          return hash;
253      }
254
255      @Override
256      public boolean equals(Object object) {
257          // TODO: Warning - this method won't work in the case the id fields are not set
258          if (!(object instanceof Product)) {
259              return false;
260          }
261          Product other = (Product) object;
262          if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
263              return false;
264          }
265          return true;
266      }
267
268      @Override
269      public String toString() {
270          return "Final5.Entity.Product[id=" + id + "]";
271      }
272  }
273
```

Kuva 9. Entity-luokkien pakolliset metodit

Customer-luokka

```
42  public class Customer implements Serializable {
43
44      private static final long serialVersionUID = 1L;
45      @Id
46      @GeneratedValue(strategy = GenerationType.IDENTITY)
47      @Basic(optional = false)
48      @Column(name = "ID")
49      private Integer id;
50      @Basic(optional = false)
51      @Column(name = "firstName")
52      private String firstName;
53      @Basic(optional = false)
54      @Column(name = "lastName")
55      private String lastName;
56      @Column(name = "phoneNumber")
57      private String phoneNumber;
58      @Basic(optional = false)
59      @Column(name = "username_email")
60      private String usernameEmail;
61      @Basic(optional = false)
62      @Column(name = "password")
63      private String password;
64      @Column(name = "isDisabled")
65      private Boolean isDisabled;
```

Kuva 10 . Customer-luokan jäsenmuuttujien määrittelyä (1/2)

```
67      @Column(name = "homeStreet")
68      private String homeStreet;
69      @Basic(optional = false)
70      @Column(name = "homeCity")
71      private String homeCity;
72      @Basic(optional = false)
73      @Column(name = "homePCode")
74      private String homePCode;
75      @Column(name = "homeCountry")
76      private String homeCountry;
77      @Basic(optional = false)
78      @Column(name = "billStreet")
79      private String billStreet;
80      @Basic(optional = false)
81      @Column(name = "billCity")
82      private String billCity;
83      @Basic(optional = false)
84      @Column(name = "billPCode")
85      private String billPCode;
86      @Column(name = "billCountry")
87      private String billCountry;
88
```

Kuva 11. Customer-luokan kenttien määrittelyä (2/2)

Customer-luokka ei eroa määrittelytavaltaan Product-luokasta. Se määrittelee asiakkaan nimi-, salasana- ja osoitetiedot.

Purchase-luokka

Purchase-luokka eroaa hieman määrittelytavaltaan kahdesta edellä esitellystä luokasta, koska siinä määritellään viitteet (tietokantaluokan

vierasavaimet) juuri edellä esiteltyihin luokkiin (kuvat 12 ja 13). Purchase-luokka muodostaa yksisuuntaisen 1:N-suhteen Product-luokan kanssa @oneToMany-annotaatiolla; ostos voi sisältää monta tuotetta, mutta kukin tuote voi esiintyä ostoskorissa vain kerran (asiakkaan ostaessa useamman kappaleen samaa tuotetta tuotteen id näkyy Purchase-taulussa vain kerran ja haluttu määrä hoidetaan inStock-muuttujan avulla). Clothes-luokan kanssa Purchase-luokka puolestaan muodostaa yksisuuntaisen N:1-suhteen @ManyToOne-annotaatiolla; ostoksella voi olla vain yksi asiakas, mutta asiakas voi tehdä monta ostosta. Yksisuuntaisuus tarkoittaa siis sitä, että suhteen toinen osapuoli ei ole tietoinen toisesta; tuote ja asiakas voivat olla olemassa ilman ostosta muttei päinvastoin.

```
5   package Entity;
6
7   import java.io.Serializable;
8   import java.util.Collection;
9   import java.util.Date;
10  import javax.persistence.Basic;
11  import javax.persistence.Column;
12  import javax.persistence.Entity;
13  import javax.persistence.GeneratedValue;
14  import javax.persistence.GenerationType;
15  import javax.persistence.Id;
16  import javax.persistence.JoinColumn;
17  import javax.persistence.ManyToOne;
18  import javax.persistence.NamedQueries;
19  import javax.persistence.NamedQuery;
20  import javax.persistence.OneToMany;
21  import javax.persistence.Table;
22  import javax.persistence.Temporal;
23  import javax.persistence.TemporalType;
24
```

Kuva 12. Purchase-luokan import-määreet

Purchase-luokassa (kuva 13) purchaseDate-kenttä ilmaisee päivämäärää ja se merkitään @Temporal-annotaatiolla. TemporalType-atribuutilla määritellään, missä muodossa päivämäärä talletetaan tietokantaan (DATE, TIME tai TIMESTAMP). [6.]

Viitteet merkitään @JoinColumn-annotaatiolla. Se määrittelee name-attribuutilla muuttujan nimen kyseisessä luokassa ja referencedColumnName-attribuutilla muuttujan nimen luokassa, johon viitataan ja näin yhdistää nämä kaksi toisiinsa eli luo niiden välille viitteen. Viitatun luokan nimen EJB-säiliö puolestaan päättelee muuttujan tyypistä

(esim. private Customer customer, joka luonnollisesti viittaa Customer-luokkaan).

```
38  public class Purchase implements Serializable {
39
40      private static final long serialVersionUID = 1L;
41      @Id
42      @GeneratedValue(strategy = GenerationType.IDENTITY)
43      @Basic(optional = false)
44      @Column(name = "ID")
45      private Integer id;
46      @Basic(optional = false)
47      @Column(name = "paymentType")
48      private String paymentType;
49      @Basic(optional = false)
50      @Column(name = "orderStatus")
51      private String orderStatus;
52      @Basic(optional = false)
53      @Column(name = "purchaseDate")
54      @Temporal(TemporalType.DATE)
55      private Date purchaseDate;
56      @Basic(optional = false)
57      @Column(name = "pSum")
58      private double pSum;
59      @JoinColumn(name = "productID", referencedColumnName = "ID")
60      @OneToMany
61      private Collection<Product> productCollection;
62      @JoinColumn(name = "customerID", referencedColumnName = "ID")
63      @ManyToOne(optional = false)
64      private Customer customer;
```

Kuva 13. Purchase-luokan jäsenmuuttujien määrittelyä

Istuntopapu-luokat

Toiminnallisuus tietokantapapujen välillä voidaan hoitaa kahdella tavalla: istuntopapujen ja backing-papujen avulla. Backing-papujen käyttäminen on suoraviivaisempaa. Niiden avulla XHTML-sivuilla olevien komponenttien arvot liitetään suoraan tietokantapapuihin. Instuntopapujen tapauksessa backing-papu käyttää istuntopapuja, jotka puolestaan muokkaavat tietokantapapuja. Istuntopapu-luokilla on kuitenkin merkittäviä ominaisuuksia, jotka backing-pavuilta puuttuvat: EJB-säiliö hoitaa käyttäjän puolesta mm. transaktiot. Istuntopapuja tarvitaankin silloin, kun mahdollisia samanaikaisia transaktioita on useita, eli tämän sovelluksen kontekstissa juurikin asiakkaille tarkoitettujen sivujen yhteydessä. Niissä ainoa kriittinen piste on samanaikainen pääsy samoihin tietokannan tietueisiin, eli silloin kun asiakas on tekemässä tilausta. Muut asiakkaan tietokantaan vaikuttavat toimet (rekisteröinti ja sisäänkirjautuminen) eivät kohdistu samoihin tietueisiin. Lisäksi tapahtumankulun (taskflow) toteuttaminen on selkeintä istuntopapu-luokan avulla, koska siihen ei sisälly muuta kuin juuri kyseisen tehtävän suorittamiseen tarvittavat metodit. Näiden syiden takia tilauksen tekemiseen liittyvä toiminnallisuus toteutetaan istuntopavun,

ShoppingCart:n, avulla. Muut asiakkaan toiminnot, kuten rekisteröityminen ja sisäänkirjautuminen, pystytään hoitamaan backing-papujen avulla. On myös huomattava, että ylläpitotoimia hoitavia luokkia varten ei ole muutenkaan mitään syytä käyttää istuntopapuja, ellei ylläpitäjien määrä ole suuri; samanaikaisten transaktioiden ongelmaa ei ole.

Ostoskoripapu (ShoppingCart)

Ostotapahtuman varsinainen sovelluslogiikka tapahtuu ostoskoripapu-luokassa (kuvat 14 ja 15). Siinä yhdistetään ostettavat tuotteet (Product) ja ne ostava asiakas (Customer) ostostapahtumaksi (Purchase), joka tapahtumaan liittyvien toimintojen jälkeen talletetaan tietokantaan. Asiakas saadaan selville hänen joko kirjauduttuaan sisään tai rekisteröidyttyään. Hänen valitsemansa tuotteet saadaan ostoskoriin addToCart- ja removeFromCart-metodeilla. Lisäksi asiakkaalla on maksutavan valinnan lisäksi mahdollisuus päivittää osoitetietojaan. OrderStatus on systeemin omaa kirjanpitoa varten luotu muuttuja, jolla pidetään kirjaa tilauksen kulloisestakin tilasta (tilattu, lähetetty, reklamoitu, ok). ExecutePurchase-metodi lopulta pistää kaiken tiedon kasaan ja luo tietokantaan uuden ostotapahtuman. Varsinainen maksunkäsittely sovelluksesta puuttuu, koska se on tarkoitus ulkoistaa maksupalveluja tarjoaville pankkilaitoksille.

Riveillä 18-21 (kuva 14) on @EJB-annotaatioilla tuotu luokkaan toisten luokkien ilmentymiä. @EJB edustaa yhtä monista tavoista toteuttaa resurssien lisääminen luokkaan, kyseessä on mekanismi nimeltään Depency Injection (DI) joka esiteltiin ensimmäisen kerran EJB 3.0 -spesifikaatiossa. Siinä missä aiemmin resurssit tuli hakea eksplisiittisesti JNDI:n (Java Naming and Directory Interface) Context-rajapinnan kautta resurssin tarkalla osoitteella (esim. "java:comp/env/CustomerFacade"), hoitaa EJB-säiliö haun nykyisin pelkän haettavan luokan nimellä.

Luokan metodit keräävät tarvittavat asiakas- ja tuotetiedot ostosta varten. @Remove-annotaatio tarkoittaa sitä, että sillä merkityn metodin suoritettuaan EJB-säiliö poistaa papu-ilmentymän pois muistista vapauttaen sen käytössä olevat resurssit.

```
1   package Session;
2
3   import Entity.Customer;
4   import Entity.Product;
5   import Entity.Purchase;
6   import Jsf.CustomerFacade;
7   import Jsf.PurchaseFacade;
8   import java.util.ArrayList;
9   import java.util.Collection;
10  import java.util.Date;
11  import javax.ejb.EJB;
12  import javax.ejb.Remove;
13  import javax.ejb.Stateful;
14
15  @Stateful
16  public class ShoppingCart {
17
18      @EJB CustomerFacade custFacade;
19      @EJB PurchaseFacade purcFacade;
20      @EJB Login login;
21      @EJB Register register;
22
23      Collection products = new ArrayList();
24      Customer customer = null;
25      int pSum = 0;
26      String paymentType;
27      String orderStatus;
28
29      public void Login(String email, String pw){
30          this.customer = login.login(email, pw);
31      }
32      public void Register(Customer cust){
33          register.register(cust);}
```

Kuva 14. Ostoskoripapu-luokan määritys (1/2).

```
34      public Collection<Product> addToCart(Product prod, int pcs) {
35          pSum += prod.getPrice() * pcs;
36          prod.subtractInStock(pcs);
37          products.add(prod);
38          return products;
39      }
40      public Collection<Product> removeFromCart(Product prod, int pcs) {
41          pSum -= prod.getPrice() * pcs;
42          prod.addInStock(pcs);
43          products.remove(prod);
44          return products;
45      }
46      public void setOrderStatus(String os) {
47          this.orderStatus = os;
48      }
49      public void setPaymentType(String pt) {
50          this.paymentType = pt;
51      }
52      public void updateHomeAddress(String street, String city, String code, String country) {
53          this.customer.setHomeAddress(street, city, code, country);
54          custFacade.edit(customer);
55      }
56      public void updateBillingAddress(String street, String city, String code, String country) {
57          this.customer.setBillingAddress(street, city, code, country);
58          custFacade.edit(customer);
59      }
60      @Remove
61      public void executePurchase() {
62          Purchase purchase = new Purchase(customer, products, paymentType, orderStatus, pSum, new Date());
63          purcFacade.create(purchase);
64      }
65  }
```

Kuva 15. Ostoskoripapu-luokan määrittely (2/2).

Facade-luokat

Tiedon tallettaminen (persistointi) tietokantaan on hoidettu fasaadi-suunnittelumallin (facade design pattern) avulla. Kullekin entity-luokalle on määritelty fasaadiluokka, joka toimii rajapintana tietokannan ja sovelluslogiikan välillä ja sen avulla hoidetaan tiedon talletus, päivitys, muokkaus ja haku tietokantaan/-kannasta. Tämän mallin ansiosta mahdolliset muutokset tietokantapalveluntoimituksessa vaikuttavat vain fasaadikerrokseen ilman tarvetta tehdä muutoksia varsinaiseen sovelluslogiikkaan. EJB 3.1 -spesifikaation mukainen toteutus määrittelee geneerisen AbstractFacade-yliluokan, jonka kullekin entity-luokalle määritelty fasaadiluokka perii (kuvat 16-19).

```
1  package Jsf;
2
3  import Entity.Customer;
4  import java.util.List;
5  import javax.persistence.EntityManager;
6  import javax.persistence.Query;
7  import javax.persistence.TypedQuery;
8
9  public abstract class AbstractFacade<T> {
10
11     private Class<T> entityClass;
12
13     public AbstractFacade(Class<T> entityClass) {
14         this.entityClass = entityClass;
15     }
16     protected abstract EntityManager getEntityManager();
17
18     public void create(T entity) {
19         getEntityManager().persist(entity);
20     }
21     public void edit(T entity) {
22         getEntityManager().merge(entity);
23     }
24     public void remove(T entity) {
25         getEntityManager().remove(getEntityManager().merge(entity));
26     }
27     public T find(Object id) {
28         return getEntityManager().find(entityClass, id);
29     }
```

Kuva 16. AbstractFacade-luokan määrittely (1/3).

```
30     public List<T> findAll() {
31         javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
32         cq.select(cq.from(entityClass));
33         return getEntityManager().createQuery(cq).getResultList();
34     }
35     public List<T> findRange(int[] range) {
36         javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
37         cq.select(cq.from(entityClass));
38         javax.persistence.Query q = getEntityManager().createQuery(cq);
39         q.setMaxResults(range[1] - range[0]);
40         q.setFirstResult(range[0]);
41         return q.getResultList();
42     }
43     public int count() {
44         javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
45         javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
46         cq.select(getEntityManager().getCriteriaBuilder().count(rt));
47         javax.persistence.Query q = getEntityManager().createQuery(cq);
48         return ((Long) q.getSingleResult()).intValue();
49     }
```

Kuva 17. AbstractFacade-luokan määrittely (2/3).

```
50      public List<String> findProductTypes() {
51          Query q = getEntityManager().createNamedQuery("Product.findProductTypes");
52          List<String> pTypes = q.getResultList();
53          return pTypes;
54      }
55      public Customer findByCredentials(String un, String pw) {
56
57          TypedQuery<Customer> q = getEntityManager().createQuery
58                  ("SELECT c FROM Customer c WHERE c.usernameEmail = :un AND c.password = :pw", Customer.class),
59          q.setParameter("un", un);
60          q.setParameter("pw", pw);
61          return q.getSingleResult();
62      }
63  }
64
```

Kuva 18. AbstractFacade-luokan määrittely (3/3).

Kuvassa 18 on määritetty kaksi kustomoitua metodia, joista ensimmäisen avulla käytetään aiemmin Product-luokan yhteydessä määriteltyä nimettyä tietokantakyselyä tuotetyyppien (productType) listaamiseksi tietokannasta verkkosivulle ja jälkimmäisen avulla testataan sisäänkirjautumisen yhteudessä, löytyykö annettua käyttäjätunnus-salasanaparia tietokannasta.

```
1   package Jsf;
2
3   import Entity.Product;
4   import javax.ejb.Stateless;
5   import javax.persistence.EntityManager;
6   import javax.persistence.PersistenceContext;
7
8   @Stateless
9   public class ProductFacade extends AbstractFacade<Product> {
10      @PersistenceContext(unitName = "Final5PU")
11      private EntityManager em;
12
13      protected EntityManager getEntityManager() {
14          return em;
15      }
16
17      public ProductFacade() {
18          super(Product.class);
19      }
20  }
21
```

Kuva 19. ProductFacade luokan määrittely

ProductaFacade-luokka perii yllä määritellyn AbstractFacade-luokan ja vastaavat fasaadit on siis luotu myös Customer- ja Purchase-luokille. Luokan tärkein toiminto on DI:n, tässä tapauksessa @PersistenceContext-annotaation avulla luokkaan tuotu resurssi EntityManager, jonka kautta EJB-säiliö hoitaa luokan ja tietokannan välisen tiedonsiirron ja kyselyt. PersistenceContextilla on parametri unitName, joka kertoo, minkä niminen luokan ja kannan välinen yhteys on. Tuon yhteyden käyttäjä määrittää itse

ottaessaan PersistenceContextin käyttöön; siinä käyttäjä liittää haluamansa tietokannan luomaansa PersistenceContextiin määrittämällä yhteydelle nimen ja yhteydelle käytettävän portin (johon yleensä käy käytetyn tietokannan tarjoama oletusportti). EntityManager on puolestaan EJB-säiliön tarjoama mekanismi, joka hoitaa O/R-mäppäyksen luokan jäsenmuuttujien ja tietokannan alkioiden välillä.

Kyseisten fasaadiluokkien metodit ovat vain ylläpitäjän käytettävissä, ts. vain ylläpitäjällä on pääsyoikeus verkkosivuille, joilla kyseisiä CRUD-toimintoja (create, read, update, destroy) käytetään.

3.3 Näkymäkerros

Faceletit

Käyttäjän näkemät näkymät eli verkkosivut ovat siis JSF 2.0 –spesifikaation mukaisesti faceletteja eli XHTML-sivuja. Ne jakaantuvat kahteen osioon: asiakas ja ylläpitäjä. Asiakkaita varten on etusivun lisäksi sivut tuotteiden selaamista, yksittäisen tuotteen esittämistä, ostoskoria, tuotteen tilaamista, asiakkaan rekisteröitymistä ja sisäänkirjautumista varten. Sivuja varten on luotu malli- eli templatesivu, joka määrittelee kaikille sivuille yhteisen ulkoasun. Sen avulla voidaan määritellä, mitkä elementit näkyvät sivuilla milläkin kohtaa ja sivuille voidaan lisäksi määrittää yhteisiä css-tiedostoja (cascading style sheets), joilla niiden ulkonäköä voidaan muokata (kuva 20). Kuvan esimerkin head-tagien sisällä on ilmoitettu käytetyt css-tiedostot. Varsinainen sivujen sisältö on määritelty body-tagin sisällä insert-tagien avulla.

```
<?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:h="http://java.sun.com/jsf/html">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <link href="./../resources/css/default.css" rel="stylesheet" type="text/css" />
        <link href="./../resources/css/cssLayout.css" rel="stylesheet" type="text/css" />
        <title>Facelets Template</title>
    </h:head>
        <h:body>
        <div id="top">
            <ui:insert name="top">Top</ui:insert>
        </div>
        <div>
            <div id="left">
                <ui:insert name="left">Left</ui:insert>
            </div>
            <div>
                <div id="right">
                    <ui:insert name="right">Right</ui:insert>
                </div>
                <div id="content" class="right_content">
                    <ui:insert name="content">Content</ui:insert>
                </div>
            </div>
        </div>
        <div id="bottom">
            <ui:insert name="bottom">Bottom</ui:insert>
        </div>   </h:body>  </html>
```

Kuva 20. Esimerkki template-sivusta

Asiakas näkee vain asiakkaille tarkoitetut sivut, olipa tämä kirjautunut sisään tai ei. Tilauksen tehdäkseen asiakkaan on kuitenkin kirjauduttava ensin sisään tunnuksillaan, ja jos sellaisia ei vielä ole, tulee asiakkaan rekisteröityä rekisteröitymissivulla. Tietokantaan on luotu myös käyttäjätunnus-salasanapari ylläpitäjää varten. Näillä tunnuksilla sisäänkirjauduttaessa tulevat näkyviin myös ylläpitäjää varten tarkoitetut sivut, joihin kuuluvat sivut fasaadiosiossa määriteltyjen CRUD-operaatioiden käyttöä varten. Niillä ylläpitäjä voi muokata, poistaa, selata ja luoda uusia tietueita kaikkiin kolmeen tauluun (Product, Customer, Purchase) liittyen. Näille sivuille on oma malliluokkansa, jolla voidaan määritellä sivuille asiakassivuja vaatimattomammat mutta nopean latautumisen takaavat tyylimääritykset. Kuvissa 21-23 on esitettynä yhden ylläpitäjälle näkyvän crud-sivun määrittely. Sen avulla luodaan uusia Product-luokan ilmentymiä eli uusia tuotteita. Vastaavat sivu löytyvät myös muillekin sovelluksen tietokantapavuille.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <ui:composition template="/template.xhtml">
        <ui:define name="title">
            <h:outputText value="#{bundle.CreateProductTitle}"></h:outputText>
        </ui:define>
        <ui:define name="body">
            <h:panelGroup id="messagePanel" layout="block">
                <h:messages errorStyle="color: red" infoStyle="color: green" layout="table"/>
            </h:panelGroup>
            <h:form>
                <h:panelGrid columns="2">
                    <h:outputLabel value="#{bundle.CreateProductLabel_productType}" for="productType" />
                    <h:selectOneMenu id="productType" value="#{productController.selected.productType}"
                                     title="#{bundle.CreateProductTitle_productType}" required="true"
                                     requiredMessage="#{bundle.CreateProductRequiredMessage_productType}">
                        <f:selectItem itemLabel="Leisure" itemValue="Leisure"/>
                        <f:selectItems value="#{productController.types}"/>
                    </h:selectOneMenu>
                    <h:outputLabel value="#{bundle.CreateProductLabel_price}" for="price" />
                    <h:inputText id="price" value="#{productController.selected.price}"
                                 title="#{bundle.CreateProductTitle_price}"
                                 required="true" requiredMessage="#{bundle.CreateProductRequiredMessage_price}"/>
                    <h:outputLabel value="#{bundle.CreateProductLabel_model}" for="model" />
                    <h:inputText id="model" value="#{productController.selected.model}"
                                 title="#{bundle.CreateProductTitle_model}" required="true"
                                 requiredMessage="#{bundle.CreateProductRequiredMessage_model}"/>
                    <h:outputLabel value="#{bundle.CreateProductLabel_manufName}" for="manufName" />
```

Kuva 21. Create.xhtml (1/3)

```
                    <h:inputText id="manufName" value="#{productController.selected.manufName}"
                                 title="#{bundle.CreateProductTitle_manufName}" required="true"
                                 requiredMessage="#{bundle.CreateProductRequiredMessage_manufName}"/>
                    <h:outputLabel value="#{bundle.CreateProductLabel_isReduced}" for="isReduced" />
                    <h:inputText id="isReduced" value="#{productController.selected.isReduced}"
                                 title="#{bundle.CreateProductTitle_isReduced}" required="true"
                                 requiredMessage="#{bundle.CreateProductRequiredMessage_isReduced}"/>
                    <h:outputLabel value="#{bundle.CreateProductLabel_inStock}" for="inStock" />
                    <h:inputText id="inStock" value="#{productController.selected.inStock}"
                                 title="#{bundle.CreateProductTitle_inStock}" required="true"
                                 requiredMessage="#{bundle.CreateProductRequiredMessage_inStock}"/>
                    <h:outputLabel value="#{bundle.CreateProductLabel_gender}" for="gender" />
                    <h:selectOneMenu id="gender" value="#{productController.selected.gender}"
                                     title="#{bundle.CreateProductTitle_gender}" required="true"
                                     requiredMessage="#{bundle.CreateProductRequiredMessage_gender}">
                        <f:selectItem itemLabel="M" itemValue="M" />
                        <f:selectItem itemLabel="F" itemValue="F"/>
                        <f:selectItem itemLabel="UNI" itemValue="UNI"/>
                    </h:selectOneMenu>
                    <h:outputLabel value="#{bundle.CreateProductLabel_description}" for="description" />
                    <h:inputText id="description" value="#{productController.selected.description}"
                                 title="#{bundle.CreateProductTitle_description}" />
                    <h:outputLabel value="#{bundle.CreateProductLabel_subProduct}" for="subProduct" />
                    <h:selectOneMenu id="subProduct" value="#{productController.selected.subProduct}"
                                     title="#{bundle.CreateProductTitle_subProduct}" required="true"
                                     requiredMessage="#{bundle.CreateProductRequiredMessage_subProduct}">
                        <f:selectItem itemLabel="CLO" itemValue="CLO" />
                        <f:selectItem itemLabel="EQU" itemValue="EQU"/>
                    </h:selectOneMenu>
                    <h:outputLabel value="#{bundle.CreateProductLabel_imageUrl}" for="imageUrl" />
                    <h:inputText id="imageUrl" value="#{productController.selected.imageUrl}"
                                 title="#{bundle.CreateProductTitle_imageUrl}" required="true"
                                 requiredMessage="#{bundle.CreateProductRequiredMessage_imageUrl}"/>
```

Kuva 22. Create.xhtml (2/3)

```
67                  <h:outputLabel value="#{bundle.CreateProductLabel_blength}" for="blength" />
68                  <h:inputText id="blength" value="#{productController.selected.blength}"
69                               title="#{bundle.CreateProductTitle_blength}" />
70                  <h:outputLabel value="#{bundle.CreateProductLabel_clothesType}" for="clothesType" />
71                  <h:inputText id="clothesType" value="#{productController.selected.clothesType}"
72                               title="#{bundle.CreateProductTitle_clothesType}" />
73                  <h:outputLabel value="#{bundle.CreateProductLabel_color}" for="color" />
74                  <h:inputText id="color" value="#{productController.selected.color}"
75                               title="#{bundle.CreateProductTitle_color}" />
76                  <h:outputLabel value="#{bundle.CreateProductLabel_material}" for="material" />
77                  <h:inputText id="material" value="#{productController.selected.material}"
78                               title="#{bundle.CreateProductTitle_material}" />
79                  <h:outputLabel value="#{bundle.CreateProductLabel_sizes}" for="sizes" />
80                  <h:inputText id="sizes" value="#{productController.selected.sizes}"
81                               title="#{bundle.CreateProductTitle_sizes}" />
82                  <h:outputLabel value="#{bundle.CreateProductLabel_width}" for="width" />
83                  <h:inputText id="width" value="#{productController.selected.width}"
84                               title="#{bundle.CreateProductTitle_width}" />
85              </h:panelGrid>
86              <h:commandLink action="#{productController.create}"
87                             value="#{bundle.CreateProductSaveLink}" />
88              <h:commandLink action="#{productController.prepareList}"
89                             value="#{bundle.CreateProductShowAllLink}" immediate="true"/>
90              <h:commandLink value="#{bundle.CreateProductIndexLink}" action="/index" immediate="true" />
91          </h:form>
92      </ui:define>
93  </ui:composition>
94  </html>
```

Kuva 23. Create.xhtml (3/3)

Kuvassa 21 heti xml-määritysten jälkeen nähdään tagi-kirjastojen määritykset nimiavaruuksineen (rivit 5-7). Ui-etuliitteellä merkityillä tageilla määritellään mallisivut, h-etuliitteillä merkityillä tageilla käyttöliittymäkomponentit (UIComponents) ja f-etuliittellä merkityillä tageilla JSF-toimintoja, jotka ovat riippuvaisia jostakin tietystä RenderKit:sta. Renderer on palvelu, joka tarjoaa mekanismin JSF-komponentin näyttämiseksi html-sivulla. RenderKit on kokoelma tällaisia Renderer-ilmentymiä.

Sivun DOM-mallin toiseksi korkeimmalla tasolla (heti html:n jälkeen) on ui:composition-tagi. Sen avulla ensinnäkin määritetään sivulla käytetty malli (template.xhtml). Lisäksi sen sisällä määritetään ui:define-tageilla, mitä elementtejä sijoitetaan mihinkin mallissa määriteltyyn sivun osaan; ui:define-tagin id-attribuutti merkitään samaksi kuin mallissa merkittiin sitä vastaavan osion ui:insert-tagin name-attribuutti (kuva 20).

Mallissa määritellyn runko-osan (body) sisällä määritellään lomake-elementti (form), joka puolestaan pitää sisällään h:panelgrid-elementin. Se näkyy verkkosivulla taulukkona, jonka sarakkeiden määrä määritetään columns-attribuutilla (kuva 21). Tällä sivulla sarakkeita on kaksi, jolloin kunkin rivin ensimmäisessä sarakkeessa näkyy ensin h:outputlabel-elementillä luotu,

tuotteen kentän nimen sisältävä tekstikenttä, ja toisessa sarakkeessa jokin syötettä vastaanottava elementti, johon ylläpitäjä syöttää haluamansa arvon. Jokaisen elementin tekstuaalisen arvon vaativa attribuutti saa arvonsa Bundle-nimisestä tekstitiedostosta. Tällä tavoin itse sivuille ei tarvitse kovakoodata lainkaan tekstiä, vaan kaikki teksti tulee ulkoisista tiedostoista tai tietokannasta. Kyseessä on JSF-spesifikaation mukainen tiedon esitystapa, ja sen avulla on helppo toteuttaa myös yksi sovellukselle asetetuista vaatimuksista eli kielen vaihtaminen.

Kielen vaihtamiseen käytetään hyväksi Locale-luokkaa, joka määrittää kaksikirjaimisen muutujan kaikille valtioille ja kielille. [18.] Pelkän kielen käyttäminen riittää tässä sovelluksessa, jossa käytettävät kielet ovat suomi ja englanti (lyhenteet fi ja en). Kieli-valtio–yhdistelmän avulla voidaan erottaa samaa kieltä puhuvat valtiot, kuten Iso-Britannia ja Yhdysvallat toisistaan (lyhenteet en-br ja en-us). Paitsi kieli, Localen avulla myös mm. päivämäärää ilmaisevat kentät saadaan esitetyksi niin kuin ne kussakin maassa on tapana esittää. [19.]

Locale voi saada arvonsa usealla eri tavalla. Niistä on tässä sovelluksessa käytetty hyväksi kahta. Jokaisella selaimella on asetuksissa automaatisesti määrätty oletuskieleksi sen maan kieli, jossa kyseisen selaimen sisältävä tietokone (verkkokortteineen) sijaitsee. Meillä oletuksena on fi, mutta tämän asetuksen voi halutessaan vaihtaa. Lisäksi kieliä voi listata useammankin, ja niiden preferenssijärjestys kulkee ylhäältä alas. Toisin sanoen, jos selaimessa on valittu halutuiksi sivun esityskieliksi fi, en ja sv tässä järjestyksessa, selain näyttää sivun suomenkielisenä, jos samankielinen toteutus löytyy. Jos ei löydy, pyritään näyttämään englanninkielinen toteutus, ja jos sitäkään ei löydy, pyritään näyttämään ruotsinkielinen toteutus. Mikäli mitään listatuista kielistä ei tueta, käydään sama kierto läpi faces-config–nimisessä xml-tiedostossa olevien Locale-määritetyin kielten kanssa (kuva 24). Jos senkään esittämiä kieliä ei tueta, näytetään sivu sen luojan määrittämällä oletuskielellä. [19.]

Bundle.properties-tiedosto sisältää kaiken sivuilla näytettävän tekstin. Tiedostoja luodaan niin monta kuin on haluttuja esityskieliä, ja ne nimetään spesifikaatiossa määrätyn tavan mukaan kielten nimet erotettuna alaviivoin: Bundle_en.properties ja Bundle_fi.properties. Bundle.properties määritellään faces-config -tiedostossa (kuva 24). Siinä määritellään perusnimeksi Bundle

ja sen aliakseksi var-attribuutilla bundle. Tätä aliasta käytetään sovelluksessa EL-kielisissä lauseissa (esim. kuva 21). [1.]

```
<?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">
    <application>
        <resource-bundle>
            <base-name>/Bundle</base-name>
            <var>bundle</var>
        </resource-bundle>
        <locale-config>
            <default-locale>fi</default-locale>
            <supported-locale>en</supported-locale>
        </locale-config>
    </application>
</faces-config>
```

Kuva 24. Faces-config-tiedosto.

Kuvassa 25 on määritetty kielen vaihtava metodi, changeLocale, joka sijaitsee backing-pavussa. Metodi hakee JSF-säiliön FacesContext-luokasta nykyisen JSF-sivun ilmentymän ja tallettaa tiedon context-muuttujaan. Sen avulla saadaan yhteys kaikkien sivulla olevien elementtien juureen getViewRoot-metodilla ja sitä kautta kyseisen sivun Locale-ilmentymän arvo. Ehtolauseen avulla testataan, onko käytössä suomi vai englanti, ja vaihdetaan käytettävä kieli toiseksi. Mallisivu asettaa jokaisen sivun yläreunaan kaksi linkkinä toimivaa kuvaa (Suomen ja Britannian lippu), joiden action-attribuuttina on EL-kielinen viite changeLocale-metodiin ja näitä panamalla kieli vaihdetaan. Oletuskieleksi on valittu suomi, mutta ulkomailta sivuille tuntuaan kieli on automaattisesti selaimen oletusarvon mukainen; jos englanti löytyy selaimen kielilistalta, tulee se valituksi (mikäli siis suomi ei ole listalla preferoituna englantiin nähden). [20.]

```
31      private FacesContext context;
32      Locale currentLocale;
33
34      public Locale getCurrentLocale() {
35          return currentLocale;
36      }
37
38      public void setCurrentLocale(Locale currentLocale) {
39          this.currentLocale = currentLocale;
40      }
41
42      public void changeLocale() {
43          context = FacesContext.getCurrentInstance();
44          currentLocale = context.getViewRoot().getLocale();
45          Locale fi = new Locale("fi");
46          if (currentLocale.equals(fi)) {
47              context.getViewRoot().setLocale(Locale.ENGLISH);
48          } else {
49              context.getViewRoot().setLocale(fi);
50          }
51      }
```

Kuva 25. ChangeLocale-metodin määrittely muuttujineen.

Syötettä vastaanottavien kenttien arvot menevät backing-papuun, tässä tapauksessa ProductController-nimiseen backing-papuun. XHTML-sivulle pavun nimi kirjoitetaan muodossa, jossa se on backing-papuluokassa @ManagedBean-annotaation name-attribuuttiin merkitty, ja siksi se kirjoitetaan tässä tapauksessa pienellä alkukirjaimella. [23.] ProductController-pavulla on getSelected-niminen metodi, joka palauttaa arvonaan käsiteltävänä olevan tuote-pavun. Spesifikaation mukaisesti se merkitään XHTML-sivulle ilman get-etuliitettä ja sen avulla päästään käsiksi tuotteen kenttiin. [1.]

Niin syöttö- kuin esityskentissäkin (input/output) kentän arvoksi tuotava tai vietävä arvo saadaan joko sivulle tai backing-pavun arvoksi Expression Language (EL) –nimisen merkintäkielen avulla. Syntaksin mukaan EL-lause alkaa joko $- tai #-merkillä. Ensimmäistä käytettäessä tietoa voidaan vain lukea pavuista elementtiin, jälkimmäisellä syötettä voidaan myös kirjoittaa papuihin. Alkumerkin jälkeen merkitään aaltosulkeisiin minne syöte talletetaan tai esityselementin ollessa kysessä, mistä tieto haetaan. [1.]

Sivun alalaidassa on kolme painiketta, joita painamalla joko siirrytään toiselle sivulle, suoritetaan jokin toiminnallisuus tai molempia. Alimmainen h:commandLink-elementeistä suorittaa ensiksi mainitun toiminnallisuuden,

eli sitä painamalla pääsee takaisin pääsivulle ilman muita toimenpiteitä. Kaksi ylempää sen sijaan suorittavat lisäksi muuta toiminnallisuutta. Ensimmäinen näistä suorittaa ProductController-luokassa määritellyn create-metodin, joka vie annetut tuotetiedot tietokantaan ja sen jälkeen lähettää ylläpitäjälle uuden Create.xhtml-sivun. Jälkimmäinen puolestaan suorittaa ProductController-luokan metodin prepareList, joka palauttaa String-tyyppisen paluuarvon ”List”. Tämä perusteella JSF-säiliö osaa navigoida osoitteeseen List.xhtml ja lähettää ylläpitäjälle kyseisen sivun. [1.]

Kuvassa 26 on määritetty login.xhtml-sivu, jolla käyttäjä kirjautuu sisään antamalla käyttäjätunnuksen ja salasanan. Rivillä 20 on käytetty yhtä JSF:n tarjoamaa valmista validaattoria, joka tarkistaa, että syötetyn salasanan pituus on vähintään kahdeksan merkkiä pitkä. Valmiita validaattoreita löytyy enemmänkin, ja niitä pystyy myös halutessaan kustomoimaan. Kun on painettu Submit-painiketta suoritetaan CustomerController-nimisen backing-papuluokan validateLogin-metodi, joka validoi käyttäjän täyttämän lomakkeen. [22.]

```
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
8      <h:head>
9          <title>Login page</title>
10     </h:head>
11     <h:body>
12         <h:form id="loginForm" prependId="false">
13             <h:panelGrid columns="2">
14                 <h:outputLabel value="Email:" for="email"/>
15                 <h:inputText id="email" value="#{customerController.email}"
16                              required="true" title="Email"/>
17                 <h:outputLabel value="Password" for="password"/>
18                 <h:inputSecret id="password" value="#{customerController.password}"
19                                required="true" title="Password">
20                     <f:validateLength minimum="8"/>
21                 </h:inputSecret>
22             </h:panelGrid>
23             <h:commandButton id="submit" value="Submit"
24                              action="#{customerController.validateLogin}"/>
25             <br/>
26             <h:messages for="email"/>
27             <h:messages for="password"/>
28         </h:form>
29     </h:body>
30 </html>
```

Kuva 26. Login.xhtml-sivun määrittely

Ajax

Ajax-toiminnallisuus saadaan JSF 2.0 –spesifikaation myötä aikaan entistä helpommin. Monimutkaisempien, mm. xml-tiedostoja läpikäyvien metodien luomiseksi on edelleen käytettävä JavaScript-skriptitiedostoja halutun toiminnallisuuden saavuttamiseksi, mutta yksinkertaisemmat perusoperaatiot onnistuvat helposti f:ajax-tagin avulla. Kuvassa 27 on määritetty vaihtoehtoinen tapa esittää Product-luokan Create-sivun alituotteen syöttö. Siinä subProduct-syöttökentän sisään on liitetty f:ajax-tagi, jonka avulla käyttäjän syöte validoidaan. [24.] Event-attribuutti määrittelee, milloin validointi suoritetaan, tässä tapauksessa aina kun käyttäjä on syöttänyt kenttään uuden merkin. Render-attribuutti määrittelee, minkä elementtien arvot päivitetään; niitä voi olla useita pilkulla erotettuina. Tässä tapauksessa vain syöttökentän alapuolella sijaitseva subText-id:llä varustettu tulostekenttä päivitetään. SubProduct on siis ainoa kenttä, jonka muuttunut arvo viedään palvelimelle ja subText ainoa kenttä, jonka arvoa päivitetään; kaikki muut kentät pysyvät palvelimen käsittelemättöminä ja muuttumattomina. Viimeisenä attribuuttina on listener, joka määrittelee, mikä metodi suoritetaan aina käyttäjän syötettyä uuden merkin. Se on productController-luokan metodi subListener, joka on backing-papuluokassa määritetty kuvan 28 osoittamalla tavalla. SubListener-tapahtumankäsittelijämetodi saa parametrina AjaxBehaviorEvent-luokan ilmentymän, joka on JSF:n tapa laukaista ajax-tapahtuma. [24.] Muuten metodi testaa, onko käyttäjä syöttänyt sallitun arvon; jos on, subText-kenttä pysyy tyhjänä, muuten tulee virheviesti punaisella tekstillä. Ilman ehtolauseen viimeistä ehtoa virheviesti tulisi näkyviin heti sivun latauduttua, ennen kuin käyttäjä on ehtinyt syöttää kenttään mitään. Tätä samaa tekniikkaa on käytetty muuallakin sovelluksessa, joko kenttien validoimikseksi tai arvojen hakemiseksi tietokannasta alasvetovalikoihin ilman sivun päivitystä. [21.]

```
41                  <h:outputLabel value="#{bundle.EditProductLabel_subProduct}" for="subProduct" />
42                  <h:inputText id="subProduct" value="#{productController.selected.subProduct}"
43                               title="#{bundle.CreateProductTitle_subProduct}" required="true"
44                               requiredMessage="#{bundle.CreateProductRequiredMessage_subProduct}">
45                      <f:ajax event="keyup" render="subText" listener="#{productController.subListener}"/>
46                  </h:inputText>
47                  <h:outputText id="subText" value="#{productController.subText}" style="color:red" />
```

Kuva 27. Ajax-esimerkki, XHTML-sivu

```
58      private String subText;
59
60      public String getSubText() {
61          return subText;
62      }
63
64      public void setSubText(String subText) {
65          this.subText = subText;
66      }
67
68      public void subListener(AjaxBehaviorEvent event) {
69
70          if (current.getSubProduct().equals("CLO") ||
71                  current.getSubProduct().equals("EQU") ||  current.getSubProduct().equals("")){
72              subText = "";
73          } else {
74              subText = "Must be either \"CLO\" or \"EQU\"";
75          }
76      }
```

Kuva 28. Ajax-esimerkki, backing-papu.

3.4 Kontrollerikerros

Kontrollerikerroksen luominen sisältää sovelluskehittäjän näkökulmasta siis JSF 2.0 –spesifikaation mukaan enää vain backing-papujen kirjoittamisen. Niitä on luotu yksi kutakin tietokantaluokkaa varten, ja edellisessä näkymä-osiossa jo esiteltiinkin niistä yhden, ProductControllerin, yhteys näkymään. Kuvissa 29-34 nähdään sama toteutus ProductController-luokan kannalta sekä sen yhteydet malli-kerrokseen.

```
19  @ManagedBean(name = "productController")
20  @SessionScoped
21  public class ProductController {
22
23      private Product current;
24      private DataModel items = null;
25      @EJB
26      private Jsf.ProductFacade ejbFacade;
27      private List<String> types = null;
28      private PaginationHelper pagination;
29      private int selectedItemIndex;
30
31      public ProductController() {
32      }
33      public Product getSelected() {
34          if (current == null) {
35              current = new Product();
36              selectedItemIndex = -1;
37          }
38          return current;
39      }
40      private ProductFacade getFacade() {
41          return ejbFacade;
42      }
43      public List<String> getTypes() {
44          types = ejbFacade.findProductTypes();
45          return types;
46      }
47      public void setTypes(List<String> types) {
48          this.types = types;
49      }
```

Kuva 29. ProductController-luokan määrittelyä (1/6)

Luokan määrittely alkaa @managedBean-annotaatiolla. Tämä ilmaisee JSF-säiliölle, että kyseinen luokka on olemassa verkkosivuja varten; se luodaan kun jokin sivu viittaa siihen EL-viittauksella. @SessionScoped-annotaatio

määrittää pavun elinajan eli kuinka kauan sen tila säilytetään muistissa. [22.] Luokka sisältää seuraavat jäsenmuuttujat:

- Jäsenmuuttujista current on Product-luokan muuttuja.
- Items on Datamodel-luokan muuttuja. JSF käyttää sitä listattavan datan rivikohtaiseen prosessointiin.
- EjbFacade on ProductFacade-luokan muuttuja ja sen kautta päästään käsiksi tietokantatoimenpiteisiin. Muuttuja on tuotu luokkaan DI:n avulla, tässä tapauksessa @EJB-annotaation avulla.
- Types on List-luokan geneerinen String-tyyppinen muuttujalista, joka listaa jo aiemmin mainittuja erilaisia productType-arvoja.
- Pagination on sovellukseen luodun PaginationHelper-luokan muuttuja. Sen avulla määritetään mm. montako Product-luokan ilmentymää näytetään yhdellä sivulla niitä listatessa.
- SelectedItemIndex on int-tyyppinen apumuuttuja PaginationHelper-luokalle.

```
50      public PaginationHelper getPagination() {
51          if (pagination == null) {
52              pagination = new PaginationHelper(10) {
                    @Override
54                  public int getItemsCount() {
55                      return getFacade().count();
56                  }
                    @Override
58                  public DataModel createPageDataModel() {
59                      return new ListDataModel(getFacade().findRange(new int[]
60                      {getPageFirstItem(), getPageFirstItem() + getPageSize()}));
61                  }
62              };
63          }
64          return pagination;
65      }
66      public String prepareList() {
67          recreateModel();
68          return "List";
69      }
70      public String prepareView() {
71          current = (Product) getItems().getRowData();
72          selectedItemIndex = pagination.getPageFirstItem() + getItems().getRowIndex();
73          return "View";
74      }
75      public String prepareCreate() {
76          current = new Product();
77          selectedItemIndex = -1;
78          return "Create";
79      }
```

Kuva 30. ProductController-luokan määrittelyä (2/6)

```
80      public String create() {
81          try {
82              getFacade().create(current);
83              JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle").
84                      getString("ProductCreated"));
85              return prepareCreate();
86          } catch (Exception e) {
87              JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle").
88                      getString("PersistenceErrorOccured"));
89              return null;
90          }
91      }
92      public String prepareEdit() {
93          current = (Product) getItems().getRowData();
94          selectedItemIndex = pagination.getPageFirstItem() + getItems().getRowIndex();
95          return "Edit";
96      }
97      public String update() {
98          try {
99              getFacade().edit(current);
100             JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle").
101                     getString("ProductUpdated"));
102             return "View";
103         } catch (Exception e) {
104             JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle").
105                     getString("PersistenceErrorOccured"));
106             return null;
107         }
108     }
```

Kuva 31. ProductController-luokan määrittelyä (3/6)

```
109     public String destroy() {
110         current = (Product) getItems().getRowData();
111         selectedItemIndex = pagination.getPageFirstItem() + getItems().getRowIndex();
112         performDestroy();
113         recreateModel();
114         return "List";
115     }
116     public String destroyAndView() {
117         performDestroy();
118         recreateModel();
119         updateCurrentItem();
120         if (selectedItemIndex >= 0) {
121             return "View";
122         } else {
123             // all items were removed - go back to list
124             recreateModel();
125             return "List";
126         }
127     }
128     private void performDestroy() {
129         try {
130             getFacade().remove(current);
131             JsfUtil.addSuccessMessage(ResourceBundle.getBundle("/Bundle").
132                     getString("ProductDeleted"));
133         } catch (Exception e) {
134             JsfUtil.addErrorMessage(e, ResourceBundle.getBundle("/Bundle").
135                     getString("PersistenceErrorOccured"));
136         }
137     }
```

Kuva 32. ProductController-luokan määrittelyä (4/6)

```
138      private void updateCurrentItem() {
139          int count = getFacade().count();
140          if (selectedItemIndex >= count) {
141              // selected index cannot be bigger than number of items:
142              selectedItemIndex = count - 1;
143              // go to previous page if last page disappeared:
144              if (pagination.getPageFirstItem() >= count) {
145                  pagination.previousPage();
146              }
147          }
148          if (selectedItemIndex >= 0) {
149              current = getFacade().findRange(new int[]{selectedItemIndex,
150              selectedItemIndex + 1}).get(0);
151          }
152      }
153      public DataModel getItems() {
154          if (items == null) {
155              items = getPagination().createPageDataModel();
156          }
157          return items;
158      }
159      private void recreateModel() {
160          items = null;
161      }
162      public String next() {
163          getPagination().nextPage();
164          recreateModel();
165          return "List";
166      }
167      public String previous() {
168          getPagination().previousPage();
169          recreateModel();
170          return "List";
171      }
```

Kuva 33. ProductController-luokan määrittelyä (5/6)

```
172      public SelectItem[] getItemsAvailableSelectMany() {
173          return JsfUtil.getSelectItems(ejbFacade.findAll(), false);}
174      public SelectItem[] getItemsAvailableSelectOne() {
175          return JsfUtil.getSelectItems(ejbFacade.findAll(), true);}
176      @FacesConverter(forClass = Product.class)
177      public static class ProductControllerConverter implements Converter {
178          public Object getAsObject
179                  (FacesContext facesContext, UIComponent component, String value) {
180              if (value == null || value.length() == 0) {
181                  return null;}
182              ProductController controller = (ProductController) facesContext.getApplication().
183                      getELResolver().getValue(facesContext.getELContext(), null, "productController");
184              return controller.ejbFacade.find(getKey(value));}
185          java.lang.Integer getKey(String value) {
186              java.lang.Integer key;
187              key = Integer.valueOf(value);
188              return key;}
189          String getStringKey(java.lang.Integer value) {
190              StringBuffer sb = new StringBuffer();
191              sb.append(value);
192              return sb.toString();}
193          public String getAsString(FacesContext facesContext, UIComponent component, Object object) {
194              if (object == null) {
195                  return null;}
196              if (object instanceof Product) {
197                  Product o = (Product) object;
198                  return getStringKey(o.getId());} else {
199                  throw new IllegalArgumentException("object " + object + " is of type " + object.
200                          getClass().getName() + "; expected type: " + ProductController.class.getName());
201                  }
202          }
203      }
204  }
205
```

Kuva 34. ProductController-luokan määrittelyä (6/6)

```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">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <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>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Kuva 35. Sovelluksen web.xml-tiedosto.

Kaikki JSF-säiliön vaatimat määrittelyt voidaan hoitaa nykyisin annotaatioilla. Näin ollen web.xml-tiedostoon, johon vielä JSF 1.2 –spesifikaation mukaan tuli merkitä kaikki kyseiset muutokset, ei nykyisen 2.0-spesifikaation mukaan tarvitse lisätä mitään. Web.xml-tiedosto pysyy varsin kevyenä. Kuvassa 35 on esitetty tämän sovelluksen tarvitsema vastaava tiedosto.

Luokka sisältää yllä mainittujen muuttujien aksessorien (kuva 27) lisäksi metodeita, joilla hoidetaan

- CRUD-toimenpiteet, esim create() (kuva 31) ja destroy() (kuva 32)
- navigointi, kaikki String-tyyppiset metodit (kuvat 29-34)
- konvertointi, joka tarkoittaa verkkosivujen tekstityyppisen datan muuttamista papujen olio-muotoiseksi dataksi ja päin vastoin. Tätä tehtävää varten on luotu @FacesConverter-annotaatiolla ProductControllerConverter-niminen sisäluokka, jonka metodit hoitavat konvertoimisen.

Osa metodeista toimii varsinaisten metodien apumetodeina, esim. prepare-metodit (kuva 30). Kaikilla pavun jäsenmuuttujilla, jotka saavat arvonsa XHTML-sivulla olevien EL-lauseiden avulla, on oltava papuluokassa aksessorit siitäkin huolimatta, että ohjelmoija itse ei niitä koodissaan käytä. Säiliö tarvitsee niitä, jotta ohjelma toimii oikein. [23.]

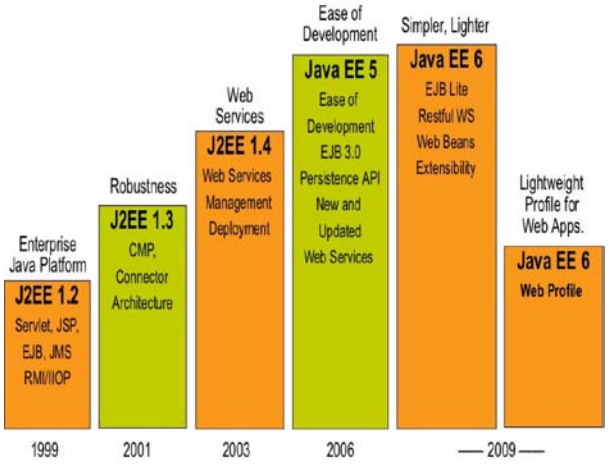
CustomerController toimii vastaavalla tavalla tarjoten metodit sekä ylläpitäjälle käyttäjätietoja muokkaavin toiminnoin että käyttäjille sisäänkirjautumista ja rekisteröitymistä varten. Näkymäosiossa esiteltiin login.xhtml-sivun määrittely, kuvassa 36 esitellään sitä vastaava osuus CustomerController-luokasta, validateLogin-metodi. Siinä tarkistetaan AbstractFactory-luokassa määritellyn kustomoidun findByCredentials-metodin avulla, löytyykö tietokannasta syötetty käyttäjätunnus-salasanapari. Jos löytyy, kirjataan käyttäjä sisään ja näytetään tälle etusivu; jos ei löydy, ohjataan tämä rekisteröinti-sivulle.

```
110     public String validateLogin() {
111         try {
112             Customer c = ejbFacade.findByCredentials(email, password);
113             return "/index";
114         } catch (Exception e) {
115             e.printStackTrace();
116             return "register";
117         }
118     }
```

Kuva 36. ValidateLogin-metodin määrittely

4 JAVA EE6 VS. JAVA EE5

J2EE6:n pääasiallinen sisältö liittyy web-ohjelmoimisen sekä –sovellusten helpottamiseen ja nopeuttamiseen. Konfiguroimista on pyritty vähentämään siirtämällä toimintoja ohjelmoijan harteilta EE-serverille mm. annotaatioiden käyttömahdollisuuksia lisäämällä. Lisäksi vanhentuneita teknologioita on jätetty pois ja tilalle on tuotu uusia, monipuolisempia ja helppokäyttöisempiä API:ja. [1.]

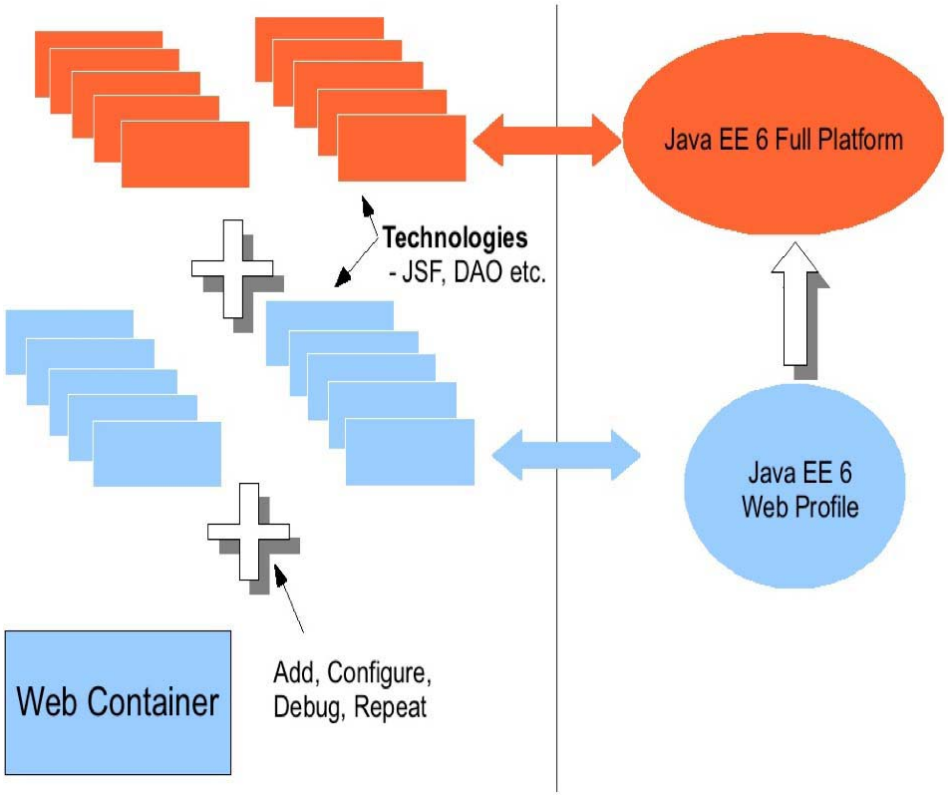


Kuva 37 JavaEE:n kehityskaari kohti kevyempää EE6:tta

Seuraavassa uudistukset pääpiirteittäin:

Profiilit (profiles)

Profiilit eivät itsessään ole mikään uusi keksintö, vaan niitä on käytetty mm. JavaME-maailmassa. Ideana on, että kaikkiin sovelluksiin ei tarvita koko EE-alustan raskasta API-arsenaalia, vaan ne voidaan touteuttaa kevennetyin API:in. Siinä missä aiemmin web-sovelluksen luonti aloitettiin web-säiliöstä johon lisätään teknologia teknologian perään (EJB, JSF jne.) aina välissä konfiguroiden ja debugaten, J2EE6 esittelee uutuutena Web Profilen, joka tarjoaa valmiin, tiiviin ja standardoidun API-paketin pienten ja keskisuurten sovellusten kehittämiseen. Siitä on helppo siirtyä tarpeen mukaan käyttämään koko alustaa. Kuva 38 havainnollistaa Web Profilen olemusta.



Kuva 38 Web Profile tarjoaa tarpeelliset työkalut tietokantapohjaisten web-sovellusten kehittämiseen

Vanhentuneet/vanhenevat teknologiat

Seuraavaksi mainitut teknologiat ovat vielä käytössä, mutta niistä ollaan luopumassa tulevissa spesifikaatioissa joko teknologian vanhentumisen tai vähäisen käytön vuoksi.

- Java API XML-pohjaiselle RPC:lle (JAX-RPC, JSR 101). Tämä API korvattiin JAX-WS:llä EE5-spesifikaatiossa.
- EJB tietokantapavut (määritelty osana JSR 153:a: Enterprise JavaBeans 2.0 ja sitä aikaisemmat). Korvattu Java Persistence API:lla (JPA).
- Java API XML Registryille (JAXR, JSR 93). Ei korvattu millään, käyttö vähäistä.
- Java EE Application Deployment (JSR 88). Ei korvattu millään, käyttö vähäistä.

Servlet 3.0

Uuden spesifikaation mukaan web.xml-tiedostoa ei enää tarvitse konfiguroida, vaan se voidaan korvata annotaatioilla. Lisäksi Servlet 3.0 tarjoaa Web Fragments –nimisen toiminnon, jolla web.xml-tiedosto voidaan jakaa loogisiin osiin niin että jokainen käytetty teknologia (esim. JSF, Atmosphere) lisää omat tietonsa jar-tiedostoon. Käyttäjän ei tarvitse itse konfiguroida xml-tiedostoja, vaan ne muodostetaan automaattisesti sovelluskehyksessä annettujen tietojen pohjalta. Tämä looginen ositus mahdollistaa web-kehyksien automaattisen rekisteröinnin web-säiliön toimesta.

Servlet 3.0 tukee myös asynkronista prosessointia (asynchronous prosessing), jonka avulla servletin ei tarvitse enää odottaa vastausta esim. tietokantakutsuun vaan sen säie voi jatkaa prosessiaan. Servletit voidaan merkitä asynkronisiksi annotaation avulla. Tämä tehostaa toimintaa erityisesti AJAX:n avulla sekä silloin, kun käytössä on minimaalisesti resursseja.

JAX-RS 1.1

JAX-RS 1.1 on RESTful Web Services 1.1 –teknologian käyttämä API. Se tarjoaa keinot käyttää URI:ja verkkoresurssien (Web resources) tavoittamiseksi ja manipuloimiseksi mistä tahansa verkossa. Se on suosittu vaihtoehto SOAP:lle.

EJB 3.1

Java EE6 -spesifikaatioon kuuluva EJB3.1 tuo mukanaan seuraavat uudistukset verrattuna aikaisempaan, J2EE5:een kuuluvaan EJB3.0:aan:

Singletonit

Istuntopapu, joka voidaan instantioida kerran per sovellusohjelma ja joka voidaan jakaa usean kyseiseen sovellukseen kuuluvan komponentin kesken. Ne tarjoavat helpon tavan jakaa tila (state) usean enterprise-papukomponentin välillä.

Ei rajapintoja

Pavulle ei tarvitse enää kirjoittaa paikallisia rajapintoja (local interface), vaan asiakkaat voivat käyttää niiden business-metodeita suoraan toiminnallisuuden (esim. transaktioiden ja tietoturvan hallinnan) pysyessä ennallaan.

Java Naming and Directory Interface (JNDI).

JNDI on teknologia, joka tarjoaa mahdollisuuden paikallistaa EJB-komponentit globaalisti määrittäen niille globaalit JNDI-nimet. Tämä parantaa EJB-komponenttien laajennettavuutta useille eri toteutuksille.

Asynkroninen istuntopapu (asynchronous session bean)

Istuntopapu voidaan annotaatiolla merkitä tukemaan asynkronisia metodikutsuja. Aiemmin kaikki istuntopapuihin kohdistuneet metodikutsut ovat olleet synkronisia.

Embeddable API

Embeddable API on sulautettu (embedded) EJB-säiliö ja sille API, jota voidaan käyttää varsinaisen EE-säiliön ulkopuolella (esim. JavaSE-ympäristössä) mutta kuitenkin saman JVM:n alla. Tämä helpottaa EE-säiliön ulkopuolisten EJB-komponenttien testaamista huomattavasti, kun säiliön tarjoamia palveluja ei tarvitse simuloida – ainakin teoriassa. Tämän projektin aikana kyseistä ominaisuutta ei saatu toimimaan lähdekoodivirheiden vuoksi. Toimiessaan olisi/tulee olemaan lyömätön testaustyökalu.

Ajastin-palvelu (Timer service)

Uusi ajastinpalvelu on parannettu versio vanhan spesifikaation mukaisesta ajastinpalvelusta. Se mahdollistaa kalenterityyppisen (cron-like) aikataulutuksen kaikentyyppisille pavuille (paitsi tilansa säilyttäville istuntopavuille) annotaation avulla.

EJB Lite

EJB Lite on kevennetty ja standardoitu versio EJB 3.1:stä, joka sisältää vain ydintoiminnot. Se on osa aiemmin esiteltyä Web Profilea.

Java Persistence API 2.0 (JPA)

JPA 2.0 sisältää parannuksia ensi kertaa Java EE5:ssä esiteltyyn tietokannanhallintaprotokollaan, kuten

- O/R-mäppäys
  tukee annotaatioiden avulla perustyyppiä (mm. Integer, String) ja embeddable-tyyppiä olevien kokoelmien (collection) mäppäystä.
- Java Persistence Query Language (JPQL)
  mm. uusia operaattoreita, kuten NULLIF ja VALUE.
- Criteria API
  tyyppi-turvallinen (type safe) kyselymekanismi, joka perustuu metatieto-konseptiin. Turvallisempi kuin perinteinen kirjainjonopohjainen (string-based) mekanismi.

Contexts and Dependency Injection (CDI)

CDI sisältää standardoidun laajennuksen EJB 3.0:ssa esiteltyyn Dependency Injection (DI) -mekanismiin, jonka avulla resurssien liittäminen ohjelmakoodiin on helppoa. Uusina ominaisuuksina mm. näkyvyysalueiden (scopes) liittäminen papuihin sekä mahdollisuus käyttää enterprise-papuja suoraan JSF-sivujen backing-papuina (backing bean).

Java ServerFaces 2.0 (JSF)

JSF 2.0 –spesifikaation suurin uudistus on aiemmin oletussivuina olleiden JSP-sivujen korvaaminen selkeämmillä XML-pohjaisilla (XHTML) Facelets-sivuilla. Lisäksi web.xml-tiedoston voi korvata kokonaan annotaatioilla.

Bean Validation 1.0

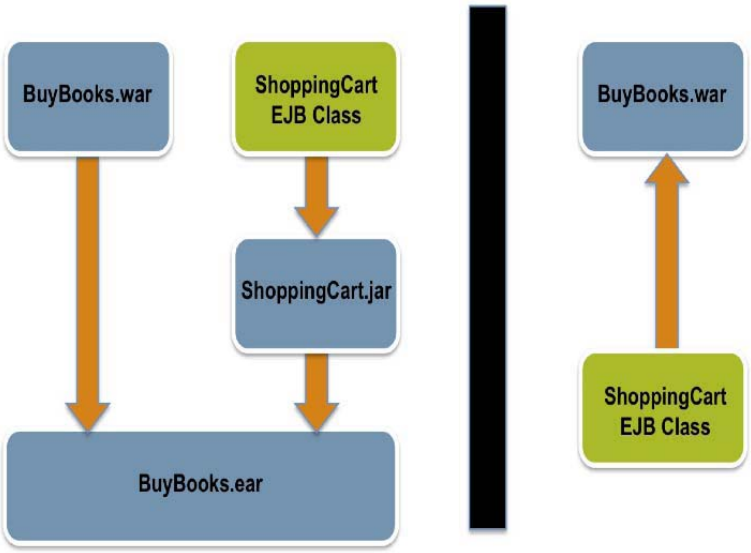
Bean Validation 1.0 on API, joka helpottaa papujen validointia. Sitä voidaan käyttää missä kerroksessa tahansa (esityskerros, datakerros jne.) annotaatioiden avulla, mahdollisuus myös luoda omia kustomoituja validaattoreita.

Connector Architecture 1.6

Connector Architecture 1.6 on isomman kaliiberin korporaatioille tarkoitettu teknologia helpottamaan molemminpuolista yhteydenpitoa enterprise-sovellusten ja Enterprise Informaatiosysteemien (Enterprise Information System, EIS) välillä. Sen avulla data liikkuu haarakonttoreilta pääkonttorille ja takaisin.

Pakkaaminen

Tiedostojen pakkaaminen on tehty helpommaksi mahdollistamalla EJB-tiedostojen pakkaaminen suoraan .war-muotoon, kun ne vielä J2EE5:n aikaan piti ensin pakata .jar-muotoon (kuva 39).



Kuva 39. Tiedostojen pakkaamisen on helpompaa J2EE6:lla (oikealla). Vasemmalla perinteinen tapa, jota vielä kuitenkin tuetaan.

Sun GlassFish™ Enterprise Server v3

GlassFish-serverin tuorein versio, v3.1 tarjoaa edeltäjiään nopeammat käynnistys- ja vasteajat. Parannuksia on tehty myös yhteensopivuuteen

kahden suosituimman IDE:n, NetBeansin ja Eclipsen kanssa. Tuki löytyy myös uudemmille, dynaamisille skriptikielille kuten Jruby ja Groovy.

5 KEHITYS- JA TESTIYMPÄRISTÖ

5.1 Kehitysympäristö

Kehitysympäristön valinta tapahtui samoilla kriteereillä kuin käytettävien teknologioiden valinta, eli IDE:n täytyi olla avoimen lähdekoodin ohjelma ja sen tuli olla yleisesti käytetty, ts. toimivaksi tiedetty ja hyvin dokumentoitu. Vaihtoehtoja oli kaksi: Eclipse ja Netbeans. Valinta kohdistui jälkimmäiseen hyvän dokumentoinnin ja erityisesti siihen integroidun ja paljon kehutun GlassFish-sovellusserverin takia. Netbeans pitää sisällään hyvät työkalut mm. tietokantatyöskentelyyn sql-editorin avulla, joten muita ohjelmia ei sovelluksen kehittämiseen tarvittu.

GlassFish-serveri on puolestaan varusteltu varsin kätevällä ja toimivalla graafisella admin-työkalulla, jonka avulla sovelluspalvelimen konfigurointi ja käyttö on helppoa. Lisäksi niin Netbeansin kuin GlassFishinkin käyttöön löytyy kattava dokumentaatio sekä esimerkkisovelluksia joiden avulla aloittelijakin pystyy perehdyttämään itsensä niiden pariin. Netbeansin viimeisin versio on 6.9 ja Glassfishin v3.1, joten käyttöön otettiin ne.

5.2 Testiympäristö

Yksikkötestaus päätettiin suorittaa yleisesti käytössä olevalla, Java-kielisille ohjelmille tarkoitetulla JUnit-ohjelmistokehyksellä. [9; 10; 11.] Sen toiminta-ajatus on yksinkertainen: luodaan Entity-luokan ilmentymä, kutsutaan sen avulla yksitellen kaikkia luokan metodeita ja testataan, antaako sovellus odotettuja arvoja. Entity-luokkien testaus on suoraviivaista ja helppoa, koska kyseiset luokat ovat tavallisia Java-luokkia (POJO-luokka) eivätkä ne ole sidoksissa mahdollisiin yläpuolisiin konstruktioihin (kuten EJB-säiliöön tai tietokantaan).

Integraatiotestaus eli istuntopapuluokkien testaaminen on hieman monimutkaisempaa, koska ne vaativat toimiakseen yleensä vähintään yhteyden tietokantaan sekä EJB-säiliön, joka siis käytännössä hoitaa kaiken

EJB:hen liittyvän toiminnallisuuden (kuten yhteyden tietokantoihin ja transaktioiden hoitamisen). Nämä toiminnot eivät kuitenkaan monesti liity varsinaiseen testattavaan ohjelmalogiikkaan, joten niiden toiminta halutaan poistaa testeistä. Tätä tarkoitusta varten on kehitetty JMock-niminen testauskehys, kuten myös samansukuiset jälkeläisensä Jmockit ja Mockito. [12; 13; 15.] Kaikkien edellämainittujen perusideana on tarjota metodeja, joiden avulla haluttu ylemmän tason toiminnallisuus korvataan ns. mock-objektilla, joka imitoi alkuperäistä toimintoa. Tällä tavoin vaikkapa tietokantakutsua varten tarvittava ja EJB-säiliön toimittama entityManager-luokan ilmentymä voidaan korvata mock-objektilla, joka tietokantaa kutsuttaessa antaa käyttäjän määräämän vastauksen. Samalla tekniikalla voidaan siis korvata kaikki säiliöistä riippuvat toiminnot, jolloin testiä varten ei tarvitse rakentaa esim. monimutkaisia testitietokantoja.

Tämän projektin integraatiotestauksessa käytettiin kehyksistä uusinta ja kevyimmän oloista eli Mockitoa. [14; 15; 16; 17.] Sen käyttöönottoa hankaloittaa kunnon dokumentoinnin puuttuminen; API [16.] ei kerro selkeästi, miten kutakin metodia käytetään. Esimerkkejäkin löytyy netistä huonosti, joten suurin työ Mockiton oppimisessa liittyy tiedon etsimiseen; itse kehys on varsin suoraviivaisen helppo käyttää kun vain keksii, mitä mikin toiminnallisuus tekee.

6 YHTEENVETO

Tie tämän projektin alkuvaiheista loppuun saakka oli pitkä ja kivinen. Työ valmistui kausiluonteisesti pienissä erissä kerrallaan. Kyseistä työskentelytapaa ei voi suositella lämpimästi kenellekään, sillä asiat tuppaavat unohtumaan matkan varrella ja asioita joutuu siksi tekemään pahimmillaan useampaan kertaan. Jos vastaava projekti pitäisi tehdä uudelleen, varaisin riittävän pitkän yhtenäisen pätkän aikaa ja tekisin selkeän aikataulun itselleni – ja pysyisin siinä.

Sovelluksen luomisessa käytettiin mahdollisuuksien mukaan hyväksi Netbeans-sovelluskehittimen tarjoamia, automaattisia koodingeneroimisominaisuuksia. Ne ovat hyödyllisiä työkaluja sovelluksen rungon rakentamiseen, mutta sellaisenaan tuotantoon kelpaavaa koodia ne eivät generoi. Koodaajan harteille jää aina koodin hiominen haluamansa laiseksi, automaattisesti generoitu koodi on aina puutteellista, toisinaan

suorastaan virheellistä. Toisaalta taas jotkin ominaisuudet eivät toimi lainkaan, ellei koodia ole generoitu automaattisesti. Kyseessä on lähdekoodivirhe, joita Netbeans sisältää valitettavan paljon. Näiden virheiden työtä hidastava vaikutus oli tämän projektin yhteydessä todella huomattava: projektin työmäärä kasvoi usealla viikolla erilaisten, ohjelmoijasta riippumattomien vikatilanteiden selvittämisen ja korjaamisen tai kiertämisen takia. Vaikka Netbeans muuten varsin näppärä sovelluskehitin onkin, suosittelen ainakin aloitteleville koodareille selvittämään, mikä on vastaava tilanne Eclipse-sovelluskehittimen kanssa.

Toinen erityisen paljon aikaa vaatinut osuus oli testaaminen. Testausyökaluja kyllä löytyy, mutta niiden käyttöohjeet olivat monesti puutteellisia ja toisinaan suorastaan virheellisiä. Toivottavasti testaustyökalujen valmistajat panostavat tulevaisuudessa edemmän tuotteidensa dokumentointiin.

Valitut teknologiat (erityisesti JSF) olivat melko työläitä opetella, mutta kun ne on oppinut niiden käyttäminen melko suoraviivaista ja vaivatonta. Aloitin projektin lukemalla suuren määrän kirjallisuutta ja aloitin vasta sitten varsinaisen koodaustyön. Koodaamisen alkuun olisi päässyt vähäisemmälläkin valmistautumisella ja niin olisi kannattanut tehdäkin: omaksuttavia asioita oli liikaa kerralla opittaviksi ja lisäksi ne alkoivat ajan kuluessa unohtumaan. Projekti olisi edennyt nopeammin ja tehokkaammin etenemällä pienin harppauksin eli ns. refaktorointimenetelmällä [9.]. Samaa menetelmää suosittelen itse koodaamiseenkin, sillä näin virhetilanteet pysyvät kontrollissa ja työ etenee jatkuvasti pienin harppauksin eteenpäin.

Näkisin oppimiseen liittyneet tavoitteeni täyttyneen melko hyvin; täydelliseksi ei tule koskaan mutta tämä projekti antoi mielestäni hyvän pohjan hyväksi ohjelmoijaksi kehittymiselle. Sovellus ei ole tämän lopputyöraportin kirjoitushetkellä vielä täysin valmis eikä näin ollen tuotantokäytössä. Sen käyttöliittymä vaatii vielä pientä hiomista ja sen vuoksi käyttöliittymäosuutta ei tässä lopputyöraportissa olla käsitelty lainkaan.

VIITELUETTELO

[1] Java EE6 Specification (WWW-dokumentti). http://jcp.org/aboutJava/communityprocess/pfd/jsr316/index.html.

[2] Java EE5 Specification (WWW-dokumentti). http://java.sun.com/javaee/5/docs/tutorial/doc/.

[3] EJB 3.0 advanced, University of Helsinki (Tieturi 2006 kurssimateriaali).

[4] (WWW-dokumentti). http://static.raibledesigns.com/repository/presentations/ComparingJavaWebFrameworks-ApacheConUS2007.pdf

[5] MySQL 5.5 Manual (WWW-dokumentti). http://dev.mysql.com/doc/refman/5.5/en/tutorial.html.

[6] Burke, Bill; Monson-Haefel, Richard: Enterprise Java Beans 3.0 (O'Reilly Media Inc. 2006).

[7] Panda, Debu; Rahman, Reza; Lane, Derek: EJB3 in action (Manning Publications 2007).

[8] EJB 3.1 -spesifikaatio (WWW-dokumentti). http://jcp.org/aboutJava/communityprocess/final/jsr318/index.html.

[9] Koskela, Lasse: Test Driven (Manning Publications 2007)

[10] Netbeans JUnit4 tutorial (WWW-dokumentti). http://netbeans.org/kb/docs/java/junit-intro.html.

[11] JUnit kotisivu (WWW-dokumentti). http://www.junit.org/.

[12] JMock kotisivu (WWW-dokumentti). http://www.jmock.org/.

[13] JMockit Tutorial (WWW-dokumentti). https://jmockit.dev.java.net/tutorial.html.

[14] Ohjeistusta yksikkötestaukseen (WWW-dokumentit). http://spagettikoodi.wordpress.com/2009/12/15/ejb-3-yksikkotestauksesta-1/, http://spagettikoodi.wordpress.com/2010/01/19/ejb-3-yksikkotestauksesta-2/ ja http://spagettikoodi.wordpress.com/2010/01/19/ejb-3-yksikkotestauksesta-3/.

[15] Mockito kotisivu (WWW-dokumentti). http://mockito.org/.

[16] Mockiton API (WWW-dokumentti). http://mockito.googlecode.com/svn/branches/1.5/javadoc/org/mockito/Mockito.html.

[17] Mockito vs. MockRunner (WWW-dokumentti). http://blog.frankel.ch/two-different-mocking-approaches.

[18] Locale-listaus (WWW-dokumentti). http://www.roseindia.net/tutorials/I18N/locales-list.shtml.

[19] i18n (WWW-dokumentti). http://jsflessons.blogspot.com/p/internationalization.html.

[20] i18n (WWW-dokumentti). http://bdn.backbase.com/java/jsf/examples/bjsf-components-i18n.

[21] Holzner, Steven: Ajax bible (Wiley publishing 2007).

[22] Bergsten, Hans: JavaServer Faces (O'reilly 2004).

[23] Schalk, Chris; Burns, Ed: JavaServer Faces - the complete reference (McGraw Hill 2007)

[24] JSF-tagi (WWW-dokumentti). http://mkblog.exadel.com/2010/04/learning-jsf-2-ajax-in-jsf-using-fajax-tag/.

LIITTEET

Sovelluksen tiedostojen hierarkiapuu