TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Tietojenkäsittelyopin laboratorio

Yrjö Kari-Koskinen, yrjo.kari-koskinen@hut.fi
Henri Sivonen, hsivonen@iki.fi
Taavi Hupponen, taavi.hupponen@hut.fi
4. helmikuuta 2004

T-106.720 Ohjelmistotekniikan projekti WWW-käyttöliittymän rakentaminen

DOMin ja MVC:n käyttö WebUI:ssa

Sisältö

1 Johdanto **1**

2 WWW-käyttöliittymien ongelmia **1**

2.1 Taustaa 1

2.2 WWW-käyttöliittymien käyttämisen ongelmia 2

2.2.1 Sivupyyntö- ja vastaus -mallin asettamat rajoitukset . . . 2

2.2.2 Selain 3

2.3 WWW-käyttöliittymien rakentamisen ongelmia 3

2.3.1 Merkkauskielen käyttö 3

2.3.2 HTML-lomakkeet 4

2.3.3 Merkistöongelmat 6

2.3.4 Välimuistit 7

3 Ratkaisuja WWW-sovellusten ongelmiin **8**

3.1 Miksi juuri oliopohjainen ratkaisu? 8

3.1.1 Java 8

3.1.2 Servletit 10

3.1.3 XML-HTTP -pyyntö 10

3.2 DOMin käyttö WebUI:ssa 11

3.2.1 Apuri- ja työkaluluokat 11

3.2.2 XHTML-pohjat 13

3.2.3 Puuesityksen sarjallistus 13

3.2.4 DOMin ongelmat 14

3.3 MVC:n käyttö WebUI:ssa 15

3.3.1 Oma MVC-toteutus 15

3.3.2 Lomakkeiden ja tietokannan populointi modelilla 16

4 Suunnitteluperiaatteita **18**

4.1 Ulkoasu ja helppokäyttöisyys 18

4.2 Esteettömyys 19

4.3 Asiakaspään ohjelmointikielet 19

4.4 POST vai GET? 20

5 Yhteenveto **20**

5.1 Laajennettavuus 21

5.2 Asiakaskäyttöliittymän toiminta 22

1 Johdanto

Ryhmämme tavoitteena oli kehittää Ohjelmistotekniikan projekti -kurssilla hajautettu WWW-julkaisujärjestelmä, joka palvelee useita sisällöltään muuttuvia WWW-sivustoja ja jota rajattu käyttäjäkunta ylläpitää erillisen WWW-käyttöliittymän eli WebUI:n kautta.

WebUI koostuu sisällöntuottajan, asiakkaan ja operaattorin käyttöliittymistä. Sisällöntuottajan käyttöliittymällä järjestelmään syötetään sivupohjia eli templateja sekä tiedostoja, jotka julkaistaan sellaisenaan. Asiakaskäyttöliittymällä syötetään kuhunkin sivustoon liitettyjen asiakassovellusten – kuten uutispalvelu tai dokumenttipankki – yksittäisiä tietokanta-alkioita. Operaattorikäyttöliittymällä ylläpidetään järjestelmän parametreja sekä lisätään järjestelmään sivustoja, asiakassovelluksia ja niiden instansseja, jotka liittyvät aina johonkin sivustoon.

Tässä raportissa käsitellään WWW-käyttöliittymien ongelmia ja WebUI:n rakentamisessa käytettyjä ratkaisuja osaan ongelmista. Lisäksi olemme maininneet erityisiä huomion arvoisia suunnitteluperiaatteita kappalessa 4.

WebUI:n lisäksi WWW-julkaisujärjestelmään kuuluu tietokannan ja massamuistin abstraktio eli DBS (database/storage), sivupohjia hyödyntävä sivugeneraattori eli template-engine, asiakassovelluksia (application), edustapalvelimet (front) ja edustan riippuvuusseurain (dependency tracker). WebUI:n kannalta merkityksellisiä komponentteja ovat asiakassovellukset ja DBS.

2 WWW-käyttöliittymien ongelmia

2.1 Taustaa

Viime aikoina WWW-ympäristöön on siirretty monia sellaisia sovelluksia, jotka perinteisesti on toteutettu komentorivi- tai graafisilla käyttöliittymillä. WWW-ympäristö tarjoaa keskitetyn ratkaisun ja valmiin alustan verkkokäytölle. Näitä etuja ajatellessa unohtuu helposti WWW-mallin rajoitukset eli ne tosiseikat, että HTTP on suunniteltu yksittäisten resurssien tilattomaan palvelemiseen [3, sivut 1 ja 7] ja HTML taas dokumentin struktuurin, esityksen, semantiikan ja sisällön esittämiseen [13]. Kumpikaan näistä lähtökohdista ei suoranaisesti palvele käyttöliittymiä tai sovelluslogiikkaa. [5]

Komentorivisovelluksia on tehty huomattavasti pidempään kuin GUI- tai WWW-ohjelmia ja ne ovat melko hyvin siirrettäviä eri käyttöjärjestelmien välillä. Komentorivisovelluksia voi myös vaivattomasti etäkäyttää unix-käyttöjärjestelmissä esimerkiksi ssh:n avulla. Perinteiset graafiset käyttöliittymät sitovat taas ohjelman hyvin vahvasti kehitysalustaansa eivätkä tarjoa siirrettävyyttä – tästä poikkeuksena Java. Graafisten käyttöliittymien etäkäyttö on mahdollista sekä X-ikkunoinnilla että Windowsin Terminal Serverillä mutta vaatii todella hyvät verkkoyhteydet. Tekstipohjaisissa ja graafisissa käyttöliittymissä käyttäjälle pystytään tarjoamaan välitön vaste tämän toimista, mikä on välttämätöntä ihmisen ja tietokoneen sulavan interaktion takaamiseksi [11, sivu 122].

WWW:n läpimurron myötä 1990-luvun loppupuolella selaimet tulivat tutuksi vanhoille ja uusille käyttäjäryhmille ja WWW-ympäristöön ruvettiin rakentamaan monia näennäisesti helppokäyttöisiä sovelluksia.

2.2 WWW-käyttöliittymien käyttämisen ongelmia

2.2.1 Sivupyyntö- ja vastaus -mallin asettamat rajoitukset

WWW-sovelluksissa vastetta käyttäjän toimista ei voi aina tarjota välittömästi, sillä HTTP perustuu pyyntöön (request) ja vasteeseen (response). Osa käyttäjän toimista aiheuttaa muutoksia selaimen sisäisessä käyttöliittymässä (esimerkiksi lomakkeen täyttäminen) ja osa taas muuttaa koko sovelluksen ulkonäköä, mutta riippuu täysin vasteen sisällöstä (esimerkiksi lomakkeen lähettäminen). Jossain WWW-sovelluksissa on hitaasti latautuviin sivuihin jopa tarpeen rakentaa erillinen, latautumisen edistymistä esittävä sivu [11, sivu 123].

Pyynnöt muodostavat myös niin kutsutun selaushistorian, mikä ei sovelluksen käytön kannalta ole välttämättä järkevä. Selaushistorian avulla käyttäjä voi palata aikaisemmille sivuille eli sovelluksen näkökulmasta käyttäjä komentaa vanhoja komentoja. Osa komennoista voi olla epämääräisiä uudelleen ajettuna, jos esimerkiksi annettu komento käsittelee tietokantariviä, joka on poistettu komentojen välillä.

Selaushistorian käyttäminen sovelluksessa navigointiin on vaikeasti ennakoitavissa ja toistettavissa, se voi olla jopa mahdotonta. Tästä syystä käyttäjälle tulostetut viestit voivat olla myös epäjohdonmukaisia.

Juuri tästä syystä GET- ja HEAD-metodit on eritelty ns. turvallisiksi metodeiksi HTTP-spesifikaatiossa eli nillä ei pidä saada aikaiseksi mitään muutoksia sovelluksen tilaan. GET-pyynnöt tallentuvat selaushistoriaan kokonaan, eivätkä siis riko selaimen ”Back”-painikkeen käyttöä. Muutoksia varten on vastaavasti POST-, PUT- ja DELETE-metodit. Mikäli käyttäjä haluaa uusia selaushistoriassa olevan POST-pyynnön, tulisi selaimen esittää varoitus ”mahdollisesti turvattoman toimen suorittamisesta”[3, sivu 51]. Sen lisäksi, että käyttää POSTia väärässä yhteydessä, voi selaushistorian rikkoa tarpeettomasti myös avaamalla linkkejä uusiin selainikkunoihin, joiden selaushistoria on luonnollisesti tyhjä. [3, sivu 51] [11, sivu 37]

HTTP-spesifikaatiossa annetaan tarkat ohjeet eri metodien valitsemiseen ja selaimien käyttäytymiseen näitä pyyntöjä suoritettaessa, mutta se ei ikävä kyllä estä ohjelmoijia valitsemaan sovellukseensa vääriä metodeja. WWW-sovellusten kirjoittaminen on lisääntynyt niin räjähdysmäisesti, ettei näihin asioihin ole ehditty kiinnittää riittävästi huomiota. Siksi jokapäiväisessä WWW-selailussa törmää sivustoihin, joiden selaushistoria ei toimi järkevästi.

Selaushistorian lisäksi hyvä esimerkki HTTP:n ongelmista WWW-sovelluksissa on protokollan tilattomuus. Protokollan alkuperäisen käyttötarkoituksen laajentaminen nykymittoihin on vaatinut tilatiedon lisäämisen protokollaan evästeiden (cookie) avulla.

2.2.2 Selain

WWW-käyttöliittymän ulkoasu riippuu paljolti selaimesta ja käytettävästä käyttöjärjestelmästä, etenkin WWW-lomakkeissa käytettävien elementtien ulkoasu voi vaihdella hyvin paljon eri selaimissa. Nykyselaimien tukemilla tyylisivuilla (CSS) päästään säätelemään ulkoasua melko tarkasti ja selainriippumattomasti, mutta silti lopputulos näyttää usein erilaiselta varsinkin eri käyttöjärjestelmissä.

Jossain tapauksissa, esimerkiksi yrityksen sisäisessä WWW-ohjelmistossa, voidaan käytettävät käyttöjärjestelmät ja selaimet rajoittaa yhteen vaihtoehtoon. Julkiseen internetiin luotavissa sovelluksissa tällaisia oletuksia ei kuitenkaan voi tehdä. Arvioidun käyttäjäryhmän mukaan voidaan tehdä esimerkiksi suositus sellaisen selaimen käytöstä, joka tukee tyylisivuja. Parkkisen mukaan kannattaa pyrkiä ”avoimeen sunnitteluun, joka mukautuu selainten ominaisuuksiin, ennemmin kuin pyrkiä esityksiin, jotka toistuvat pikselintarkasti selaimelta toiselle siirryttäessä” [11, sivu 47].

Selaimen renderöimää WWW-käyttöliittymää ympäröi lisäksi selaimen oma käyttöliittymä, jota ei pysty täysin aukottomasti piilottamaankaan. Tämä on otettava huomioon WWW-sovellusta suunniteltaessa: WWW-UI:n eri osien on erotuttava selkeästi selaimen eri osista ja sen tarjoamien navigointielementtien kanssa on oltava tarkkana, ettei käyttäjä sotke niitä selaimen navigointipainikkeisiin.

2.3 WWW-käyttöliittymien rakentamisen ongelmia

2.3.1 Merkkauskielen käyttö

Graafisen käyttöliittymän rakentaminen merkkauskielen elementeistä, jotka etenevät lineaarisesti dokumentin alusta lähtien, on ehkä huomattavin ongelma WWW-käyttöliittymien rakentamisessa. Yhtäältä ongelmana on sivun rakenteen ja ulkoasun sekoittuminen merkkauskielessä, toisaalta taas HTML:n käsitteleminen merkkijonomanipulaation keinoin. Lähes kaikissa perinteisissä WWW-ohjelmointiympäristöissä – kuten aktiivisivuteknologiat (esimerkiksi PHP, ASP, JSP), CGI-ohjelmointi ja Java-servletit – merkkijonomanipulaatio tuottaa helposti virheitä tuotettuun merkkaukseen ja se on myös erittäin hidasta ohjelmoida, sillä mm. kaikki lainausmerkit täytyy eskapoida esitettäessä merkkauskieltä jonkin ohjelmointikielen merkkijonovakioina [2]. Template-engine -raportissa käsitellään lähemmin merkkijonomanipulaation ongelmia [14].

Tässä julkaisujärjestelmässä on päädytty käyttämään Document Object Model:ia (DOM) merkkauskielien käsittelyyn, mikä ratkaisee suoranaiset merkkijonomanipulaation aiheuttamat ongelmat. DOMin eduista ja ongelmista kerrotaan tarkemmin kappaleessa 3.2.

Tyylisivut tarjoavat ratkaisun rakenteen ja ulkoasun erotteluun [11, sivu 48]. Aivan kaikkia ongelmia CSS ei ratkaise, mutta joistain ongelmallisista rakenteista kuten ulkoasun taittamisesta table-elementillä päästään eroon. Tyylisivujen tuki eri selaimissa on kuitenkin vaihtelevaa, joten käyttäjälle pitää suositella

tyylisivut osaavaa selainta. Vielä tärkeämpää olisi testata sivustot myös ilman tyylisivuja, jotta tyylisivujen tuen puuttuminen ei varmasti estä sivujen toiminnallisuutta.

Vähimmäisvaatimus WWW-käyttöliittymää rakennettaessa on käytetyn merkkauskielen oikeellisuus. Juuri erilaiset aktiivisivuteknologiat ja muut HTML:ää merkijonona manipuolivat menetelmät aiheuttavat helposti syntaktisia virheitä tuotettuun merkkauskieleen. Lisäksi oikeellisuuden tarkistaminen on näillä menetelmillä usein hyvin työlästä, eikä oikeellisuustarkastimelle eli validaattorille ole ehkä mahdollista edes syöttää yhden resurssin kaikkia eri tulosteita, jos osa tulosteista saadaan vain POST-pyynnön vasteena.

Oikeellisuudesta pystyy pitämään parhaiten huolta, kun käyttää merkkauskielen tiukinta strict-muotoa, jota myös HTML-spesifikaatio suosittelee [13, kappale 4.1]. Tämän noudattamisessa hyvänä apuvälineenä ovat esimerkiksi `w3.org`:n ja `htmlhelp.com`:n validaattorit. HTML-määrittelyn noudattamiseen ei kuitenkaan riitä vain se, ettei validaattoriohjelma löydä virheitä. Esimerkiksi linkkien toimivuutta ja URLien muotosääntöjä ei pysty tarkistamaan pelkällä validoinnilla [7, sivu 158].

Vähimmäisvaatimus XML-dokumentin, kuten XHTML:n, jäsennettävyydelle onkin, että se on XML-spesifikaation mukaisesti hyvin muodostettu (well-formed) [1, kappale 2.1]. XML-dokumentin validius edellyttää lisäksi, että dokumentissä on document type declaration, jossa määritellään document type definition (DTD) ja että dokumentti noudattaa DTD:ssä julistettuja kielioppisääntöjä [10, sivu 40] [1, kappale 2.8]. XML:stä puhuttaessa validiutta olennaisempaa on juuri hyvin muodostettu rakenne, sillä XML:n määritelmän mukaan dataobjekti, joka ei ole hyvin muodostettu, ei ole XML-dokumentti.

DOMin puuesityksen sarjallistin, joka tuottaa julkaistavaa XHTML:ää, pitää huolen siitä, että XHTML-esitys on hyvin muodostettu eskapoimalla esimerkiksi '&'- ja '<'-merkit. DOMin käyttäjän huoleksi jää huolehtia elementtien asianmukaisista sisäkkäisyyksistä ja oikeiden attribuuttien käytöstä oikeissa elementeissä DTD:n mukaisesti sekä DOM-jäsentäjän XML-syötetiedostojen oikeellisuudesta. Näin ollen DOM ratkaisee alkeellisimmat hyvin muodostamiseen liittyvät ongelmat ja ohjelmoija voi keskittyä DOMille annettavan tiedon oikeellisuuteen sen sijaan että keskityttäisiin tulostettavan tavuvirran oikeellisuuteen kuten aktiivisivuteknologioissa.

Mikäli käytetään XML:ää, on lisäksi syytä huolehtia oikeista nimiavaruuksista kaikissa yhteyksissä. Pelkän XHTML:n mukaisten elementtien käyttö rajoittaa nimiavaruudet yhteen, mikä helpottaa huomattavasti nimiavaruuksiin liittyvää työmäärää.

2.3.2 HTML-lomakkeet

HTML-lomakkeiden avulla voidaan esittää WWW-sovelluksessa tarvittavia syöte-elementtejä kuten asetusnapit (check box) ja pudotusvalikot. Linkit ja niiden käyttämisestä aiheutuvat GET-pyynnöt ovat lomakkeiden lisäksi ainoa tapa luoda käyttäjän ja sovelluksen välistä interaktiota. POST-pyyntöjä onkin mahdollista saada selaimessa aikaiseksi vain lomakkeiden avulla.

Kenties tärkein lomakkeita koskeva valinta on lomakkeen lähettämiseen käytetty metodi: käytetäänkö GETtiä vai POSTia? Nyrkkisääntönä voidaan pitää POSTin käyttämistä sovelluksen tilaa muuttavissa sivupyynnöissä ja GETtiä kaikissa muissa. Tällöin ei turhaan rikota selaushistoriaa, kuten edellä kappaleessa 2.2.1 jo mainittiin. GETin kanssa on kuitenkin syytä olla hyvin varovainen, sillä HTML-lomakkeen kentän syöte voi paisuttaa pyyntö-URIn (request-URI) hyvin pitkäksi, mikä voi olla olla ongelma vanhemmille selaimille tai proxyille [3, kappale 3.2.1]. GET-pyyntöjä ei myöskään pidä käyttää arkaluontoisten tietojen kuten kirjautumistietojen välittämiseen, sillä pyynnössä esiintyvät parametrit tallentuvat palvelimen, välityspalvelimen ja selaimen logeihin, josta ne ovat luettavissa selkokielisinä [3, kappale 15.1.3].

HTTP:n tilattomuus aiheuttaa ongelmia WWW-sovelluksissa muiden kuin triviaalien lomakkeiden käsittelyyn: useimmiten lomakkeella syötetty data halutaan tarkistaa ennen sen varsinaista käsittelyä ja mikäli tarkastuksessa havaitaan virhe, pitää alkuperäinen lomake palauttaa käyttäjälle täytettynä valmiiksi käyttäjän antamilla syötteillä. Lisäksi palautetun lomakkeen yhteydessä pitäisi pystyä antamaan virheviesti, jonka perusteella käyttäjä ymmärtäisi, mitä kohtaa lomakkeessa hänen täytyy korjata päästäkseen tavoitteessaan eteenpäin.

Tämän tyyppistä saman lomakkeen palautusta takaisin käyttäjälle kutsutaan lomakkeen roundtripiksi. Palautetussa lomakkeessa käyttäjän jo syöttämät arvot on asetettu syötekenttien uusiksi alkuarvoiksi, kun alkuperäisessä lomakkeessa alkuarvot ovat yleensä loogisesti tyhjiä arvoja. Lomakkeen alkuarvojen muuttamisen seurauksena reset-painike ei toimi enää lomakkeen tyhjentävässä merkityksessä, eikä sitä kannata näyttää käyttäjälle, varsinkaan ”Tyhjennä”-tekstillä. Toisaalta tyhjennysnappulan tarpeellisuus HTML-lomakkeessa on hyvin kysenalainen, sillä kukapa haluaisi vahingossa tyhjentää kaikki syöttämänsä tiedot yhdellä napinpainalluksella siinä vaiheessa, kun kohtaa lomakkeen lopussa olevat ”Lähetä”- ja ”Tyhjennä”-painikkeet.

Syötekenttien alkuarvojen esittäminen input-elementin value-attribuutissa merkkijonona on herkkä eskapointivirheille. Tässä WWW-julkaisujärjestelmässä ongelmat korostuvat, sillä lomakkeiden syötteenä on merkkauskieltä useissa eri yhteyksissä. Jos syötteenä on merkkijono ”`<p class=\"foo\">`” täytyy se eskapoida muotoon ”`&lt;p class=\"foo\"&gt;`”. DOM kuitenkin eskapoi merkkijonot aina tarpeen vaatiessa eikä ohjelmoijan tarvitse huolehtia siitä, missä yhteydessä tekstiä milloinkin esitetään ja millä metodilla merkkijonot pitää näin ollen eskapoida. Merkkijonojen eskapoinnin lisäksi ongelmana on sovelluslogiikan sekä syötteen ja kenttien käsittelyn hoitavan ohjelmakoodin kirjoittaminen limittäin, mikä huonontaa yleensä koodin ylläpidettävyyttä.

Toinen HTTP:n tilattomuuden aiheuttama ongelma on moniosaisen lomakkeen eri osien välisen tilatiedon tallentaminen. Tiedot voidaan tallentaa joko lomakkeisiin piilokentiksi tai palvelimella käyttäjäsessioon, jota pidetään yllä joko evästeiden tai yhden automaattisen, kaikissa pyynnöissä lähetettävän piilokentän avulla. Piilokentät ovat sessioita luotettavampi metodi tähän tarkoitukseen. Moniosaisen lomakkeen eteneminen olisi myös syytä esittää käyttäjälle lomakkeen ohessa tekstinä, josta selviää jo käytyjen askelien määrä ja tavoitteeseen pääsemiseen tarvittavien askelien kokonaismäärä. Lisäksi tällaisesta lomakkeesta pitäisi tarjota ”selkeä tie eteenpäin, taaksepäin tai pois lomakesarjasta”. [11,

sivu 40]

Kun HTML-lomaketta käytetään tiedoston siirtämiseen palvelimelle, aiheutuu sekä moniosaisten lomakkeiden käytöstä että lomakkeen roundtrippauksesta väliaikaistiedostojen ongelma: jos siirretty tiedosto oli kunnossa, mutta jokin muu lomakkeen kenttä aiheutti lomakkeen palauttamisen, pitää tiedosto tallentaa väliaikaistaltioon, jotta käyttäjän ei tarvitse uudelleen etsiä tiedostoa ja suorittaa mahdollisesti hidasta siirtoa palvelimelle. Mutta miten jo siirretty tiedosto esitetään lomakkeessa ja miten kauan väliaikaistiedostoja pidetään tallessa? Tässä järjestelmässä päätettiin ratkaista jälkimmäinen ongelma ajamalla säännöllisesti cronilla putsausohjelma, joka tuhoaa esimerkiksi yli viikon vanhat väliaikaistiedostot.

Lomakkeiden rakentamisessa on lukuisia käytettävyysongelmia; käyttäjän huomion ohjaaminen lomakkeissa oikeisiin asioihin ja tehokkaan lomakkeen täyttämisen mahdollistaminen eivät ole aivan yksikäsitteisiä asioita. Tärkeimpiä seikkoja lomakkeiden käytettävyydessä ovat

- pakollisten kenttien ilmoittaminen
- kaiken olemassaolevan tiedon antaminen oletusarvoiksi syötekenttiin eli käyttäjän ”tarkastettavaksi”
- painikkeen painamisen jälkeen palautteen antaminen käyttäjälle: onnistuiko suoritettu toimenpide, vai vaatiiko se vielä jatkotoimia
- samalla sivulla olevien erillisten lomakkeiden selkeä erottaminen esimerkiksi eri taustavärillä, jotta käyttäjä tietää, mitä kenttiä hänen täytyy täyttää [11, sivut 104 ja 45]. HTML:n fieldset-elementti sopii hyvin juuri tätä tarkoitusta varten.

2.3.3 Merkistöongelmat

HTML oli aluksi määritelty länsikeskeisesti käyttämään vain ISO-8859-1-merkkikoodausta. Vielä HTML 3.2 oli määritelty ISO-8859-1:n suhteen. Koska tarve muiden merkkikoodausten käyttöön on monien kirjoitusjärjestelmien kohdalla pakottava, ruvettiin muitakin merkkikoodauksia käyttämään. Villin käytön vuoksi selaimissa on vieläkin manuaalinen valinta merkkikoodaukselle, vaikka on määritelty, miten HTML-dokumentin merkkikoodaus ilmoitetaan selaimelle siten, ettei käyttäjän tarvitse säätää selaintaan sivukohtaisesti.

HTML 4.0:sta lähtien HTML on määritelty Unicoden (virallisesti merkeittäin Unicoden kanssa identtisen ISO-10646:n) suhteen [13, kappale 5.1]. Ajatellaan, että HTML-dokumentin merkit ovat syvimmältä luonteeltaan Unicode-merkkejä, vaikka ne olisikin koodattu siirtoa varten vaikkapa ISO-8859-1-koodauksella. Nykyselaimet myös todella käyttävät Unicodea sisäisenä esityksenä. Vaikka sivu olisikin ISO-8859-1-koodattu, voidaan numeeristen merkkiviittausten avulla käyttää myös ISO-8859-1:n ulkopuolelle jääviä merkkejä.

Lomakkeiden osalta tilanne on hankalampi. Teoriassa sivulla voidaan ilmoittaa selaimelle, mitä merkkikoodausta käyttäen lomakedata tulisi lähettää. Teorias-

sa selain voisi ilmoittaa POST-pyynnössä (muttei GETissä) palvelimelle käyttämänsä merkkikoodauksen. Käytännössä selaimet eivät toimi näin, koska maailmalla on käytössä lomakkeenkäsittelijöitä, jotka eivät siedä merkkikoodausparametria. [4]

Lomakedatan osalta todellisten selaimien menettelytapa on ”sama kuin sisääntuleva”. Samuuteen tosin liittyy erikoisuus: Joskus ISO-8859-*-koodaukset käsitellään todellisuudessa näistä koodauksista laajennettujen Windows-koodisivujen mukaan. [4]

Vaikka sivulla voidaan käyttää mitä tahansa Unicode-merkkejä merkkikoodauksesta riippumatta, selaimelta palvelimelle lähetettävässä lomakedatassa ei voida käyttää merkkejä käytetyn merkkikoodauksen salliman repertoaarin ulkopuolelta standardoidusti ja luotettavasti. Jotta mikä tahansa käyttäjän syöttämä merkki tulisi ehjänä perille, täytyy käyttää merkkikoodausta, jolla voidaan esittää koko Unicode-repertoaari. Käytännössä tämä tarkoittaa UTF-8:aa. ”Sama kuin sisääntuleva” -politiikan vuoksi haluttuun tulokseen päästään lähettämällä lomakkeen sisältävä sivu UTF-8-koodattuna ja sellaiseksi julistettuna selaimelle.

Selaimelle voitaisiin lähettää piilokentässä numeerisin merkkiviittauksin esitetty ASCII:hin kuulumaton merkkijono, jonka avulla nähtäisiin, lähettääkö selain todella odotettua koodausta. Piilokentän poikkeava sarjallistus olisi kuitenkin edellyttänyt erikoistapauksen ohjelmoimista sarjallistimeen. UTF-8 on nykyisin niin hyvin tuettu, ettei tällaista tarkistusta toteutettu.

2.3.4 Välimuistit

”Kaikki HTTP-yhteyden osapuolet, jotka eivät toimi tunnelina, voivat käyttää sisäistä välimuistia pyyntöjen käsittelyyn” [3, sivu 13]. Tällaisia osapuolia ovat siis yhteyden päätepisteet (asiakas ja palvelin) sekä mahdolliset päätepisteiden välillä olevat yhdyskäytävät ja proxyt. Välimuisteja käytetään nopeuttamaan pyyntöjä ja säästämään tietoliikennekaistan käyttöä WWW-selaimissa, -proxyissä ja -palvelimissa.

Jos haettavan kohteen tuoreus tiedetään, ei sivupyyntöä tarvitse tehdä edes palvelimelle asti välimuistin ansiosta. Muutoin suoritetaan konditionaalinen sivupyyntö, joka palauttaa uuden sisällön ainoastaan siinä tapauksessa, että se on muuttunut.

WWW-sovelluksissa tuotetut vasteet ovat lähes poikkeuksetta sisällöltään dynaamisia ja siten ongelmallisia välimuistin kannalta. Välimuistin käytön voi rikkoa hyvällä omallatunnolla sellaisessa WWW-sovelluksessa, jossa selaimelle ei tarjota enemmän tai vähemmän staattisia sisältödokumentteja vaan pelkästään sovelluksen käyttöliittymää. WebUI:ssa tämä on tehty käyttämällä Expires-otsaketta HTTP 1.0 -protokollaa varten ja Cache-Control -otsaketta HTTP 1.1:ä varten.

Joitain WWW-sovelluksen vasteita ei haluta esittää muuten kuin ainutkertaisena vasteena tietystä käyttäjän toimesta (esimerkiksi tietokantarivin poistaminen); sama vaste muissa tilanteissa olisi virheellinen ja johtaisi käyttäjää har-

haan. Tällaisten vasteiden tapauksessa on syytä käyttää esimerkiksi POST-, PUT- tai DELETE-pyyntöä, joiden vasteita ei normaalisti saa tallentaa välimuistiin [3, sivut 55-56]. Näiden pyyntöjen onnistuneet vasteet myös invalidoivat implisiittisesti vanhan kyseistä pyyntö-URI:a koskevan välimuistitallenteen, mikä on myös syytä ottaa huomioon WWW-sovelluksen suunnittelussa välimuistin aiheuttamien sivuvaikutusten suhteen [3, sivu 98]. Selaimet panevat alulle näistä kuitenkin vain POST-pyyntöjä.

3 Ratkaisuja WWW-sovellusten ongelmiin

Osaan edellä mainittuja ongelmia tarvittiin juuri tähän projektiin soveltuva ratkaisu. Oleellisia ja ratkaistavissa olevia ongelmia olivat merkkauskielen oikeellisuus, käyttöliittymän ulkoasu ja HTML-lomakkeiden käsittelyyn liittyvät ongelmat. Kolmannessa osiossa mainitaan erikseen yleisiä suunnitteluperiaatteita, joita noudattamalla vältettiin tai ainakin käsiteltiin monia yleisiä sudenkuoppia.

Näiden ratkaistujen ongelmien ja sudenkuoppien lisäksi WWW-sovelluksissa on hyvin paljon WWW:lle ominaisia ongelmia, joita ei tässä yhteydessä edes yritetä ratkoa. Tällaisia ongelmia olivat välittömän vasteen puuttuminen, täysin erilaisten selain- ja käyttöjärjestelmäympäristöjen tukeminen ja WWW-käyttöliittymän käytettävyysongelmat. Etenkin käytettävyydestä voisi kirjoittaa ihan oman raporttinsa aiheen laajuuden vuoksi. Näiden ongelmien vaikutuksia on kuitenkin syytä pohtia ja niiden vaarat tiedostaa, kun on suunnittelemassa mitä tahansa WWW-sovellusta.

3.1 Miksi juuri oliopohjainen ratkaisu?

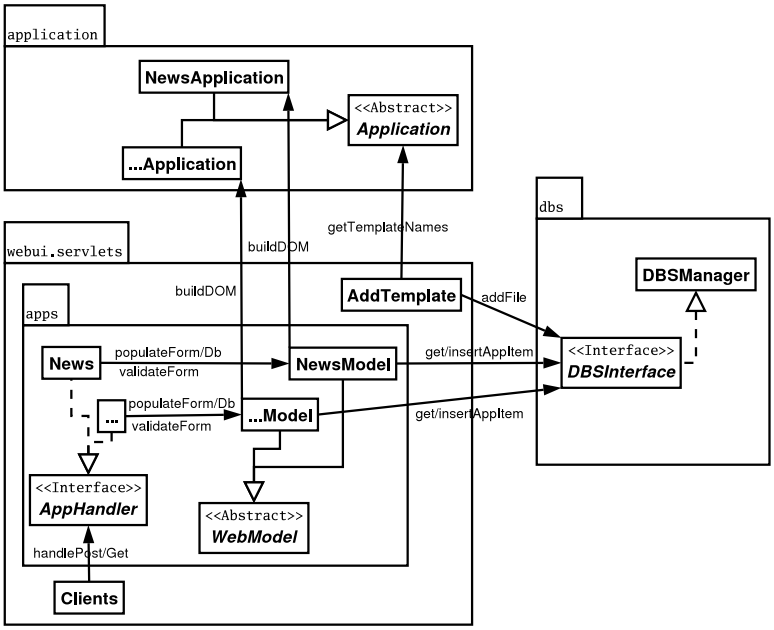
Manola mainitsee oliopohjaisten järjestelmien olevan asianmukainen ympäristö ”heterogeenisten, autonomisten ja hajautettujen järjestelmien” kehittämisessä [9, kappale 1.2]. Tässä järjestelmässä juuri erilaiset asiakassovellukset ovat sekä heterogeenisiä ja autonomisia olioita: heterogeenisiä, sillä itse sovellukset tekevät hyvin erilaisia toimenpiteitä (esimerkiksi uutissivut ja hakupalvelu), mutta myös autonomisia, koska ne voivat muuttua sisäisesti tämän vaikuttamatta niiden rajapintoihin [9, kappale 1.2].

Oliopohjaisen kehitysympäristön valintaa puoltaa myös parempi ylläpidettävyys, sillä Gaedken ja Gellersenin mukaan WWW-sovelluksia päivitetään ja muutetaan usein, mikä johtaa sovelluksia ylläpidettäessä kriittisiin ongelmiin [5, sivu 60].

3.1.1 Java

Koko projektin eli WWW-julkaisujärjestelmän toteutukseen valittiin Java-kieli sen hyvän laajennettavuuden ja DOM-tuen takia (katso kohta 3.2). Lisäksi Java oli ryhmälle entuudestaan tuttu: tiedettiin, että järjestelmälle asetetut tavoitteet saataisiin Javalla ylipäänsä toteutettua.

WebUI:n ja asiakassovellusten väliset kutsut saatiin toteutettua Application- ja AppHandler-rajapinnoilla. Rajapinnat ovat välttämättömiä WWW-julkaisujärjestelmän laajennettavuuden ja toiminnan kannalta. Laajennettavuus onkin yksi järjestelmän tärkeimmistä tavotteista.



Kuva 1: WebUI:n, asiakassovellusten ja tietokannan väliset rajapinnat

Kuvasta 1 selviää WebUI:n (webui.servlets-paketti), tietokannan (dbs-paketti) ja asiakassovellusten (application-paketti) välinen rajapintojen käyttö:

- Clients-servlet kutsuu AppHandler-rajapinnan kautta apps-paketin luokkia metodeilla `handlePost` ja `handleGet`, jotka käsittävät yhden asiakassovelluksen tietoalkion (esimerkiksi yksi uutinen) syöttämiseen ja editointiin liittyvän sovelluslogiikan.
- AddTemplate pyytää tarvittavien sivupohjien (template) tiedostonimet (metodilla `getTemplateNames`) asiakassovelluksilta Application-rajapinnan kautta.
- Tietokantaa käytetään DBSInterfacen kautta (`addFile`-, `insertAppItem`- ja `getAppItems`-metodit).
- Lisäksi kukin AppHandlerin lapsiluokista käyttää vastaavaa model-luokkaa.

Uusien asiakassovellusten luokkia (application- ja webui.servlets.apps-paketeissa) voidaan lisätä ajonaikaisesti järjestelmään Javan dynaamisen luokkien lataamisen ja ajonaikaisen linkityksen ansiosta. Käytännössä instantioitavan luokan nimi saadaan tietokannasta merkkijonona, josta pystytään instantioimaan varsinainen kättökelpoinen olio `java.lang.Class`-luokan reflektiivisillä metodeilla.

3.1.2 Servletit

Javan tarjoamat vaihtoehdot WWW-sovelluksien kirjoittamiseen ovat servletit ja Java Server Pages (JSP). Projektin muut merkkauskieltä käsittelevät osat eli asiakassovellukset ja template-engine käsittelevät merkkauskieliä DOM-rajapinnan kautta, joten WebUI:ssa joutuu myös käyttämään DOMia sivupohjien ja asiakassovellusten tietoalkioiden esikatselun liittämiseksi WebUI:n palvelemille WWW-sivuille. JSP tuo mukanaan samanlaisia merkkijonomanipulaation ongelmia kuin esimerkiksi PHP, joten valitsimme WebUI:n toteuttamisen servleteillä siten, että *kaikki* merkkauskielen käsittely toteutetaan DOMilla.

Javan servlettien (ja JSP:n) ylivoimainen etu PHP:hen nähden on käännetyn ohjelmakoodin pitäminen muistissa. Jokaisen sivupyynnön prosessointi alkaa PHP-tulkissa aina koodin kääntämisestä ja kun prosessointi päättyy, vapautetaan ohjelmakoodi muistista. C:llä tai C++:lla kirjoitettu Apache-moduuli voidaan myös kirjoittaa muistissa pysyväksi, mutta tällöin ongelmiksi muodostuvat vapauttamatta jääneen muistin vuotaminen ja ylivuotojen aiheuttamat tietoturvaongelmat.

Valmiiksi muistiin ladatun koodin suorittaminen on tehokkaampaa suurilla pyyntömäärillä. Tällöin palvelinohjelmisto käynnistää useita eri säikeitä ajamaan koodia samasta muistiosoitteesta (reentrant code), mikä vaatii ainoastaan lokaalimuuttujien käyttämistä servlettien metodeissa, jotta eri säikeet eivät pääse ylikirjoittamaan toistensa muuttujia. Käytännössä tämä hoidettin apuriluokalla (DOMDocument, katso kuva 2), johon tallennetaan kaikki sellaiset tiedot, jotka olisi muuten voinut tallentaa luokan kenttiin. Jokaisessa sivupyynnössä luodaan uusi tällainen apuriolio, joka välitetään kaikkiin tarvittaviin metodeihin.

3.1.3 XML-HTTP -pyyntö

WebUI:n rakentamiseen olisi voitu käyttää myös XML-HTTP -pyyntöä, joka mahdollistaa HTTP-pyyntöjen tekemisen JavaScriptillä WWW-selaimesta käsin [8]. Tällä teknologialla voidaan piilottaa palvelimelle tehtävät pyynnöt käyttäjältä ja esittää pyyntöjen seurauksena saatava data dokumenttiin ilmestyvänä merkkauksena. DOM-tuki löytyy myös JavaScriptistä, joten dokumentin käsittely asiakaspäässä ei olisi ollut ongelma. Sen sijaan XML-HTTP -pyyntö on tuettu vain MSIE:ssä ja Mozillassa – ja niissäkin eri tavalla, joten se ei ollut todellinen vaihtoehto, varsinkin kun WebUI:n esteettömyydestä valtaosalla selaimista haluttiin pitää kiinni.

Tämä teknologia menestyy varmasti vastaisuudessa paremmin, kunhan tuki eri selaimissa paranee, sillä asiakaspäähän ohjelmoidulla koodilla voidaan välttää turhia lomakkeiden round-trippejä ja parantaa käyttöliittymän käytettävyyttä nopean palautteen osalta.

Myös oliopohjaisen WWW-käyttöliittymäsuunnittelun lähdemateriaaleissa suositeltiin DOMin käyttöä asiakaspuolen ohjelmoinnissa lähinnä DHTML:n eli dynaamisen HTML:n korvaajana [9] [16]. Vaikka selaimeen kirjoitettavan ohjelmiston voisi toteuttaa myös palvelimella niitä selaimia varten, joista puuttuu

JavaScript, olisi tämän vaatima lisätyömäärä hyvin huomattava. WWW-käyttöliittymän ohjelmointi asiakaspuolella olisikin varteenotettava vaihtoehto, mikäli käyttäjäkunta ja sitä myöten käytettävät selaimet olisi hyvin rajatut.

3.2 DOMin käyttö WebUI:ssa

DOMin käyttö WebUI:ssa oli perusteltua, sillä se tarjosi erittäin käyttökelpoisen rajapinnan merkkauskieltä esittävien olioiden välittämiseen WebUI:n ja julkaisujärjestelmän muiden osien välillä. Sen ajateltiin myös ratkaisevan merkkauksen oikeellisuuteen ja käyttöliittymän rakentamiseen liittyviä ongelmia. Lisäksi ajateltiin, että DOM mahdollistaisi järjestelmän laajentamisen eri merkkauskielille. DOM ei ratkaissut näitä kaikkia – esimerkiksi muiden merkkauskielten sarjallistusta, mutta se edesauttoi säilyttämään hyvän rakenteellisuuden sekä WebUI:n luokkien Java-koodissa että tuotetussa merkkauksessa.

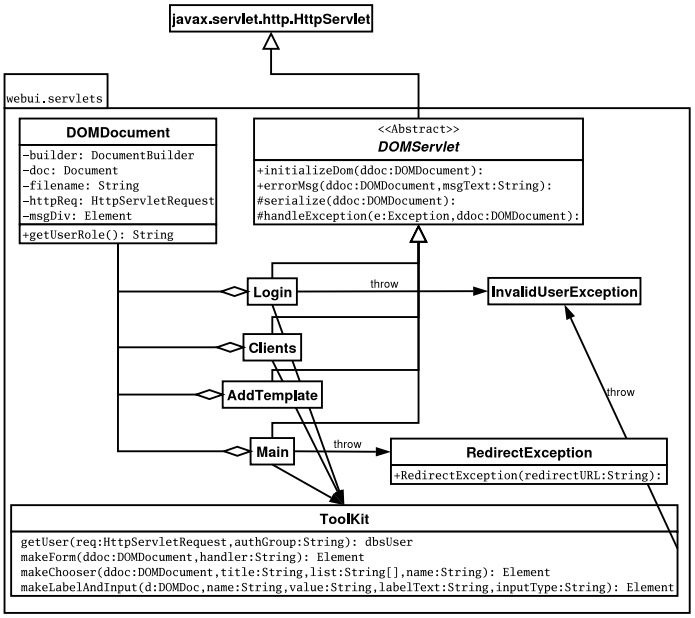
DOM-pohjaisten servlettien koodaaminen on verraten raskasta verrattuna JSP:n kaltaisiin aktiivisivuteknologioihin, joissa merkkauskieltä ripotellaan koodin välissä tulostevirtaan. Ohjelmistokehityksen alussa joutuu kirjoittamaan työkaluluokkia (luokat ToolKit ja DOMServlet kuvassa 2) DOMin käsittelyyn ja ratkomaan työkaluihin liittyviä ongelmia, kuten monisäikeisyys ja poikkeusten sujuva käsittely. Lopulta uusien servlettien luominen on kuitenkin nopeampaa ja hallitumpaa kuin puhtaalla JSP:llä.

DOM tarjoaa hyvät puitteet kaikissa servleteissä tarvittavien poikkeusten käsittelyn, virhe- ja infoviestien dokumenttiin liittämisen yms. yleistämiseen, sillä dokumentti on kokoelma puumaisesti järjestettyjä olioita, jotka sarjallistetaan vasta aivan lopuksi tekstiksi, joka sisältää merkkauskielen elementtejä. Kun kesken HTTP-pyynnön käsittelyn lentää poikkeus, voidaan poikkeuksen vaatimat sivuvaikutukset suorittaa ja vaikka initialisoida uusi dokumentti, joka kuvaa virhetilanteen käyttäjälle. Luotaessa merkkauskieltä merkkijonomanipulaation keinoin juuri tällaiset poikkeustilanteet ovat ongelmallisia, sillä yleensä osa sivusta on jo tulostettu tulostevirtaan, eikä uuden dokumentin muodostamista ole mahdollista aloittaa alusta.

3.2.1 Apuri- ja työkaluluokat

Toteutimme abstraktin DOMServlet-luokan (katso kuva 2), joka perii servlet-api:n HttpServlet-luokan ja tarjoaa kaikille WebUI:n servleteille (mm. Login, Clients, AddTemplate ja Main) seuraavat metodit:

- `initializeDOM` jäsentää DOMDocument-oliossa määritellystä XHTML-tiedostosta DOM-puun ja lisää siihen mm. kaikille servleteille yhteiset tyylisivulinkin, otsikon ja linkkipalkin.
- `errorMsg` liittää annetun virheilmoituksen tai viestin ennalta määrättyyn paikkaan dokumentissa.
- `serialize` sarjallistaa dokumentin HTML-sarjallistimella käyttäen UTF-8 -merkistöä.



Kuva 2: WebUI:n servlettien periytyminen

- `handleException` käsittelee poikkeukset seuraavasti:
  - `InvalidUserException` aiheuttaa käyttäjän session invalidoimisen,
  - `RedirectException` aiheuttaa uudelleenohjauksen halutulle sivulle,
  - `IllegalArgumentException` aiheuttaa virheilmoituksen liittämisen dokumenttiin
  - muut poikkeukset heitetään `ServletException`:n sisällä ulospäin, mikä tuottaa vasteena HTTP-koodin 500: Internal server error ja vastaavan sivun.

ToolKit-luokkaan taas koottiin staattisia metodeja, joiden avulla eri servletit pystyvät tuottamaan vaivattomasti usein käytettyjä rakenteita dokumenttiin ja mm. tarkistamaan käyttäjäautentikoinnin. Jos vastaava työkaluabstraktio toteutetaan jollakin merkkijonomanipulaatioon perustuvalla tekniikalla, joudutaan tuotettu merkkauskieli joko tulostamaan suoraan tulostevirtaan tai palauttamaan yhtenä merkkijonona. Sekä tulostevirran että merkkijonon muokkaaminen jälkikäteen on joko hyvin vaikeaa tai mahdotonta merkkauskielen vaatiman monimutkaisen jäsentämisen vuoksi. [9, kappale 2.3.1]

HTML-rakenteita tuottavat ToolKitin metodit palauttavat DOM-rajapinnan mukaisen Element-olion, joka on mahdollista sijoittaa mihin tahansa kohtaan dokumenttia ja sitä pystyy käsittelemään vielä jälkeenpäin hakemalla viittaus siihen DOMin Document-oliossa olevilla puun läpikäyntimetodeilla. DOMin puuesityksen tarpeeton läpikäyminen voidaan kuitenkin välttää pitämällä tarpeelliset olioviittaukset tallessa esimerkiksi DOMDocument-oliossa aina dokumentin sarjallistukseen saakka.

3.2.2 XHTML-pohjat

Jokaisen WebUI-servletin DOM-esityksen runkona on XHTML-tiedosto, josta DOM-jäsentäjä muodostaa puuesityksen. Puuesitykseen lisätään dynaamista sisältöä DOM-rajapinnan metodeilla. XHTML-tiedostot tarjoavat mahdollisuuden lisätä WWW-käyttöliittymän tuottamiin sivuihin staattista sisältöä ilman, että servlettejä tarvitsee kääntää ja käynnistää uudelleen.

Manolan mukaan ”ohjelmakoodin liittäminen sivun eri komponentteihin on helppoa WWW-sivun oliesityksessä” [9, kappale 2.3.1]. Tämä toteutuu nimenomaan XHTML-pohjien kanssa, joskin tällöin tiedostojen sisältämän merkkauksen pitää olla tiukan rakenteellista, jotta sitä pystyy järkevästi käsittelemään DOM-olioilla servletin ohjelmakoodissa.

Tiukka rakenteellisuus onkin erittäin hyvä asia, sillä siihen ja rakenteen ja esityksen selkeään erottamiseen juuri kehoitetaan HTML-spesifikaatiossa. Tämä helpottaa dokumenttien päivittämistä ja eri esityksien muodostamista eri medioille tyylisivujen avulla. [13, kappale 2.4.1]

3.2.3 Puuesityksen sarjallistus

Viimeisenä DOMin eduista mainitsemme sarjallistuksen. Erillään dokumentin käsittelystä toteutettu sarjallistus auttaa välttämään merkkijonomanipulaatiolle tyypillisiä ongelmia kuten eskapointivirheitä ja puuttuvia tageja.

WebUI:ssa on käytetty XHTML:n mukaisia elementtejä DOMin käsittelyssä, joten DOM-puusta voidaan tuottaa DTD:tön XHTML-dokumentti yleiskäyttöisellä XML-sarjallistimella. Tällainen sarjallistus soveltuu lähetettäväksi mm. Mozillalle, Safarille ja Operalle application/xhtml+xml-mediatyypillä, jolloin dokumentti jäsennetään selaimessa XML-jäsentimellä. Tätä menettelyä käytettiin projektin alkuvaiheessa.

Järjestelmän testaamiseen haluttiin käyttää Lynxiä ja lisäksi yhteensopivuus Microsoft Internet Explorerin kanssa katsottiin järjestelmälle asetetuksi vaatimukseksi. Näiden selaiten vuoksi oli tarpeen lahettää dokumentit text/html-medityypillä. Yleiskäyttöisellä XML-sarjallistimella tuotettua merkkausta ei taas ole soveliasta väittää text/html:ksi. XHTML 1.0 -dokumentin saa kyllä merkitä text/html:ksi, jos sarjallistus on rajoitettu XHTML 1.0 -määrityksen liitteen C mukaisesti, koska selaimet eivät jäsennä text/html:ää XML-jäsentimellä [12].

Emme lähteneet noudattamaan liitettä C, vaan siirrymme tuottamaan HTML 4.01:tä, jonka väittäminen text/html:ksi on soveliasta eikä millään tapaa kiistanalaista. HTML:n tuottaminen XHTML:stä on helppoa, koska XHTML 1.0 on ainoastaan HTML 4.01:n uudelleenmuotoilu XML:ää pohjana käyttäen.

Henrin ja Taavin WWW-julkaisujärjestelmään implementoima HTML-sarjallistin pyrkii olemaan mahdollisimman lähellä hyvin muodostettua XML-mielessä (katso kappale 2.3.1) sikäli kuin se on HTML:ssä mahdollista. Käytännössä sarjallistin siis tuottaa merkkaukseen kaikki valinnaiset alkutagit (esimerkiksi `<html>`, `<head>` ja `<body>`) sekä myös valinnaiset lopputagit (esimerkiksi `</p>`).

Helppo HTML-sarjallistus ei kuitenkaan tarkoita sitä, että jotkin muut formaatit olisivat yhtä helposti tuotettavissa. XHTML:n ja HTML:n lisäksi muiden merkkauskielten sarjallistimien toteuttaminen WebUI:ssa voi olla hyvin hankalaa riippuen (X)HTML:n elementtien transformoitumisesta tarvittavalle merkkauskielelle. Vielä suurempi ongelma olisi eri merkkauskielien lomakkeiden käsittely.

Mikäli haluttaisiin oikeasti sarjallistaa esim WML:ää, pitäisi alkuperäinen puu muodostaa erillisellä, abstraktimmalla XML-kielellä, joka sitten olisi mahdollista transformoida joko (X)HTML:ksi tai WML:ksi. Tällaisessa abstraktissa kielessä määriteltäisiin WebUI:n tarvitsemia rakenteellisia elementtejä kuten `<linkkipalkki/>` tai `<templateuploadform/>`. Kielen tuottaminen tätä järjestelmää varten ei kuitenkaan ollut tarkoituksenmukaista varsinkin kun WML:n tai muiden merkkauskielien sarjallistus eivät olleet WebUI:n vaatimuksena.

Sen sijaan TE:n voisi helpommin laajentaa tuottamaan muitakin merkkauksia, sillä TE:ssä ei tarvita lomakkeiden käsittelyä. Toisaalta TE:ssä tapahtuva DOMin käsittely on sidottu (X)HTML-kieleen eli muita merkkauksia varten pitäisi toteuttaa TE:hen omat laajennukset. Laajamittaisempaan merkkausten transformointiin onkin olemassa hyvin monipuolinen ja toisaalta raskas työkalu nimeltä Extensible Stylesheet Language Transformations (XSLT).

Uusien sarjallistimien käyttöönotto aiheuttaa jatkossa muutoksia WebUI:ssa ainoastaan DOMServletin `serialize`-metodissa. Sitä muuttamalla WebUI on hyvin laajennettavissa ainakin uusiin HTML:n tai XHTML:n versioihin.

3.2.4 DOMin ongelmat

DOMin käsittelyyn tarvittavien työkaluluokkien kirjoittaminen tyhjältä pöydältä osoittautui hyvin työlääksi, varsinkin kun DOMin käytöstä ei ollut aikaisempaa kokemusta. Jo pelkästään sen perusteella voi todeta, että DOMin käyttö yksinkertaisiin tai pieniin WWW-käyttöliittymiin ei ole perusteltua, mikäli DOMista ei kyseisessä järjestelmässä ole jotain erityistä hyötynäkökohtaa kuten eri sarjallistimien tarve tai eri lähteistä saatavien merkkausten yhdistäminen.

Toinen hankalaksi kokemamme ominaisuus DOMissa ja servleteissä yleensä oli olioviittauksien vyöryttäminen jollekin sitä tarvitsevalle työkaluluokan metodille. Usein tuli vastaan tarve käyttää tietyn elementin, HTTP-pyynnön tai HTTP-vasteen olioviittausta, jonka saaminen käteen ei ollut ihan parin lisäparametrin takana. DOMDocument-olio ratkaisi lopulta useimmat tällaiset tapaukset.

Juuri HTTP-pyynnön ja -vasteen tapauksissa olioiden välittäminen parametreina tai toisen olion kenttinä tuntui turhauttavalta, sillä tahtomattaan käytäntöä vertasi PHP- tai Perl-tyyliseen globaalin muuttujan käsittelyyn. Toisaalta globaalit muuttujat ovat osoittautuneet tietoturvariskeiksi huolimattomasti toteutetuissa PHP-sovelluksissa.

Käyttämässämme DOM-toteutuksessa eli GNU JAXPissa[1] huomattiin bugi, joka tuli ilmi template-enginessä. Tämän seurauksena otimme käyttöön Javan mukana tulevan DOM-implementaation, joka aiheutti ongelmia WebUI:ssa:

[1] Katso `http://www.gnu.org/software/classpathx/jaxp/`

`getElementById`-metodi ei palauttanut ajonaikaisesti lisättyjä elementtejä kuten GNU JAXPissa. Ongelma saatiin kierrettyä WebUI:ssa tallentamalla tarvittavia olioviittauksia DOMDocument-luokkaan. Tässä tuli kuitenkin ilmi DOM-spesifikaation väljyyden ongelmat. Eri ilmplementaatioiden eroista ja DOMin vaihtoehdoista kerrotaan tarkemmin template-engine -raportissa [**?**, 16]

3.3 MVC:n käyttö WebUI:ssa

WebUI:n toteutuksen olennaisimpiin osiin eli käyttöliittymän rakentamiseen ja HTML-lomakkeiden käsittelyyn harkittiin Model-View-Controller -paradigman (MVC) käyttöä. Javaan pohjautuvassa WWW-ohjelmoinnissa yleisin MVC-malli on niin kutsuttu Model 2, jossa controllerina toimivat servletit, view:n muodostavat JSP-sivut ja model käsitellään Javabean-luokilla [2]. WWW-ohjelmointia pelkällä JSP:llä tai servleteillä on kutsuttu Model 1:ksi. Model 2:n eduiksi mainitaan mm. ”eri käyttöliittymien rakentamisen helppous saman datan käsittelyyn, paremmat mahdollisuudet koodin uudelleenkäyttöön (code reuse) sekä eri modulien itsenäinen debuggaus” [15].

Ylivoimaisesti tunnetuin ja suosituin MVC-kehitysympäristö Javalla Web-sovelluksia ohjelmoitaessa on Apache Jakarta -projektin Struts[2]. Struts tarjoaa oman controller-komponenttinsa, jolloin model kirjoitetaan tietokantatoimintojen ympärille esimerkiksi JDBC:llä tai JavaBeans-teknologialla. View:n kirjoittamiseen Strutsin yhteydessä suositellaan esimerkiksi JSP:tä tai XSLT:tä.

DOMin ja servlettien tarjoamat edut katsottiin WebUI:ssa niin tärkeiksi, että emme halunneet vaihtaa tilalle JSP:tä Strutsia varten. Toisaalta view-komponentti hyötyisi Strutsin tarjoamasta monipuolisesta tagikirjastosta[3] ainoastaan mikäli view olisi toteutettu JSP:llä [2]. Tagikirjaston lisäksi eräs Strutsin tarjoamista voimakkaista abstraktioista olisi ollut WebUI:n lokalisaatio käyttämällä Strutsin `<message-resources>`:a, joka mahdollistaa mm. merkkijonoresurssien ulkoistamisen sovelluskoodista kansainvälistämisen (internationalisation, i18n) hoitavaan luokkaan [2]. Lokalisaatio ei kuitenkaan kuulunut WebUI:n vaatimuksiin.

WWW-julkaisujärjestelmässämme on myös monipuolinen tietokanta-abstraktio – jota tarvitaan joka tapauksessa järjestelmän muita osia varten, joten Strutsin model-komponentin hyödyt olisivat olleet vähäiset. Arvioimme, että Strutsin käyttöönotto ja sen sovittaminen tähän sovellukseen olisi vaatinut runsaasti lisätyötä ja -opettelua vielä DOMin tuoman työläyden lisäksi, emmekä siis valinneet Strutsia käytettäväksi työkaluksi.

3.3.1 Oma MVC-toteutus

Pohdimme myös oman MVC-toteutuksen rakentamista: Tarjoaisivatko DOMin käyttämät XHTML-pohjat view-komponentin, joka erottaisi sisällön toimin-

[2]Katso `http://jakarta.apache.org/struts/`

[3]Struts-JSP:ssä esimerkiksi virheilmoitukset laajennetaan `<html:errors/>`-tagin paikalle. WebUI:ssa jokaiseen palveltavaan sivuun liitetään div-elementti, johon virhe- tai infoviesti asetetaan DOMServletin `errorMsg`-metodilla.

nallisuudesta? Voisiko servlettien työn jakaa tämän jälkeen erikseen model- ja controller-osiin? Vastaukset ovat ei ja kyllä.

XHTML-pohjista puuttuu Strutsin JSP-tagikirjaston edut, joten pohjat täytyy kytkeä elementtien id-attribuuteilla tiukasti siiheen DOM-koodiin, jolla näkymää muutetaan; sisältöä ei siis pysty kunnolla erottamaan toiminnallisuudesta. Sen sijaan servleteissä päästään XHTML-pohjien avulla eroon merkkauskieltä tuottavasta koodista juuri yhteen tietokanta-alkioon (DBSItem) liittyvän HTML-lomakkeen käsittelyssä, ja tällöin servlettien muun koodin erittely onnistuu helpommin ja itse asiassa hyvin siististi.

Tästä jäljelle jäävästä koodista voitiin nimittäin erottaa sovelluslogiikka ja datan käsittely lomakkeessa tai tietokannassa. Näistä syntyivät vastaavasti controller ja model. WebUI:ssa asiakassovellusten tietokanta-alkioita käsittelevissä osissa (servlets.apps-paketti) AppHandler-rajapinnan toteuttava News-luokka toimii controllerina ja WebModel-isäluokan toteuttava NewsModel-luokka toimii modelina. Näistä controller käsittelee palvelimelle tulleen POST- tai GET-pyynnön, instantioi tarvitsemansa modelin ja kutsuu tarvittavia modelin metodeja.

Java-servletit sidotaan yhteen tai useampaan pyyntö-URIin (request URI) ServletHandler-luokan avulla, joten sama controller voi palvella useita eri pyyntöjä. Controller voi tällöin valita vastauksessa käytettävän view:n – eli XHTML-pohjan – asiakkaan servletille antamien parametrien sekä myös pyyntö-URI:n perusteella. Yksi controller voi siis ”palvella jokaista toisiinsa liittyvää toiminnallisuutta” [6, kappale 11.1.1].

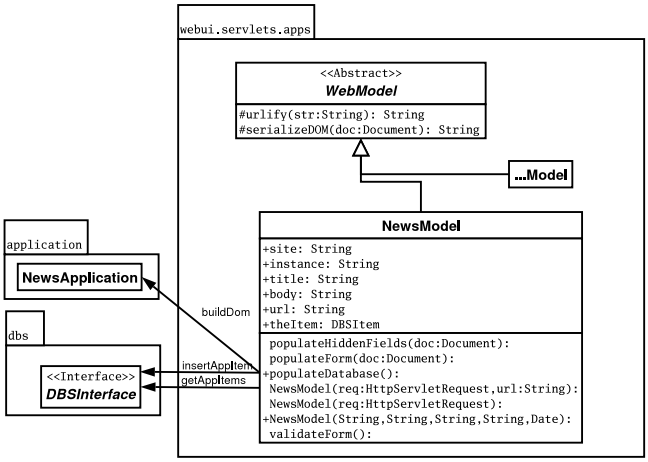
3.3.2 Lomakkeiden ja tietokannan populointi modelilla

Henri esitti ajatuksen edellä mainitun kaltaisesta model-oliosta, joka saisi datansa joko HTTP-pyynnön parametreista tai tietokannasta ja jolla voitaisiin populoida joko HTML-lomake tai yksi tietokantarivi. Tämän idean toteutus osoittautui voimakkaaksi abstraktioksi ja loi pohjan omalle MVC-toteutukselle osassa WebUI:ta. Poikkeuksena tiukasta MVC-mallista on kuitenkin XHTML-pohjien tarjoama view, joka ei osaa muuttaa itseään tarpeiden mukaan, vaan modelin pitää propagoida muutokset viewiin `populateForm`- ja `populateHiddenFields`-metodeilla. Tällaisella model-oliolla pystyttiin ratkaisemaan nimenomaan HTML-lomakkeiden käsittelyyn liittyviä ongelmia.

Model-luokat perivät abstraktin WebModel-luokan (katso kuva 3), jossa on `urlify`- ja `serializeDOM`-metodit sekä rajapinnat `validateForm`- ja populointimetodeille. `Urlify` muodostaa annetusta tietokanta-alkion nimestä uniikin ja normalisoidun[4] URI:n. `SerializeDOM` sarjallistaa asiakassovelluksen palauttaman DOM-puun tietokantaan tallennusta varten.

Jokaista model-luokkaa kutsutaan controllerista kahdessa eri yhteydessä: uuden tietokanta-alkion lisäyksessä ja olemassaolevan alkion muokkauksessa. Ensimmäisessä tapauksessa model saa datansa HttpServletRequest-oliosta ja jälkim-

[4]Tähän käytetään IBM:n Unicode normalization API:a ja NFKD-normalisaatiota, jolla saadaan mm. ääkkösistä pisteet kirjainten päältä pois. Lisäksi välilyönnit muutetaan tavuviivoiksi.



Kuva 3: WebUI:n model-luokat

mäisessä tapauksessa model hakee datansa tietokannasta annetun URLin perusteella. Controller-luokkia ovat siis AppHandler-rajapinnan toteuttavat luokat kuten luokka News kuvassa 1.

Controller-luokka kutsuu modelin `validateForm`-metodia syötteen oikeellisuuden tarkistamiseksi ja konstruoi tämän perusteella käyttäjälle vasteeksi DOM-työkaluilla halutun sivun. Syötetietojen tarkistaminen vaatii aina vastaavan Application-luokan `buildDOM`-metodin kutsumista syötetyn merkkauksen tarkistamiseksi. Mikäli lomake roundtrippaa (katso kappale 2.3.2), liitetään modelin data HTML-lomakkeeseen `populateForm`-metodissa DOMin avulla. Myös struts tarjoaa valmiit työkalut lomakkeen tietojen validointiin; käyttämämme menetelmä poikkeaa strutsista siten, että me käsittelemme HTML-esitystä DOMilla validoinnin jälkeen, kun taas strutsissa esityksen muuttaminen tapahtuu automaattisesti tagikirjaston tageilla määriteltyyn kohtaan merkkauksessa [2].

Mikäli lomakkeesta saatu syöte halutaan lisätä tietokantaan, kutsutaan `populateDatabase`-metodia. Metodin kutsuminen yhtä tietokanta-alkiota muokattaessa ei poikkea uuden alkion lisäämisestä mitenkään. Alkion muokkaus ja lisääminen on helppo kirjoittaa samaan controlleriin, sillä model huolehtii mm. muokattavan alkion yksilöivän kentän (URL) kuljettamisesta lomakkeen roundtrippauksessa. Tämä tapahtuu `populateHiddenFields`-metodissa.

Erillisten model-luokkien kirjoittaminen eriytti HTML-lomakkeiden käsittelyn kätevästi pois sovelluslogiikan seasta controllerin ohjelmakoodissa. Huomasimmekin, että WebUI:n debuggaaminen helpottui näiltä osin juuri koodin eriyttämisen takia, niin kuin MVC:n käytöstä edellä jo vihjailtiinkin. Modelin olisi tosin voinut viedä vielä pidemmälle luomalla lomakkeet DOMilla ajonaikaisesti. Tähän käytettiin kuitenkin XHTML-pohjia jotka eriyttivät kätevästi asiakassovelluskohtaisen lomakkeen merkkauksen omaan tiedostoonsa.

Modelin tarjoama yhden tietokanta-alkion käsittelyn abstraktio osoittautui hyväksi ideaksi myös ohjelmiston rakennusvaiheessa: model-luokkiin kirjoitettiin oma, julkinen konstruktori, jolla pystyttiin helposti luomaan komentoriviohjel-

massa tietokanta-alkoita testausta varten ja siirtämään ne tietokantaan julkisella `populateDatabase`-metodilla. Modelia hyödyntämällä varmistuttiin siitä, että testidata tallentui varmasti samalla tavalla tietokantaan kuin WebUI:n kautta syötetty data.

4 Suunnitteluperiaatteita

Tässä kerrotaan muutamista yleisistä suunnitteluperiaatteista, joita WebUI:ssa on noudatettu.

4.1 Ulkoasu ja helppokäyttöisyys

WebUI:n käyttäjäryhmä on rajattu: sitä käyttävät vain WWW-sivuja ylläpitävät henkilöt, joilta voidaan odottaa HTML-lomakkeiden käytön perustuntemus. Mikäli käyttäjä syöttää merkkauskieltä lomakkeeseen, oletetaan tämän myös tuntevan merkkauksen oikeellisuuden tarkastamiseen kuuluvan perussanaston (alkavat ja sulkevat tagit jne.). Tekstejä voi syöttää myös ilman merkkausta, jolloin asiakassovellus liittää tarvittavan merkkauksen eli `<p>`-tagit kappaleiden alkuun ja loppuun.

WebUI:ssa on pyritty konsistenttiin ulkoasuun ja pitämään samanlaiset toiminnot saman näköisinä, mikä rakentaa käyttäjän luottamusta järjestelmään [11, sivu 43]. Ulkoasun ja tyylin yhtenäiseen määrittelyyn on käytetty tyylisivuja ja rakenteellisesti vahvaa merkkausta, minkä seurauksena palvelu näyttää todennäköisesti hieman erilaiselta eri selaimissa. Parkkisen mukaan tärkeämpää kuitenkin on ”palvelun toimiminen järkevästi siinä ympäristössä, jossa käyttäjä käyttää muitakin palveluja” [11, sivu 41].

Yksittäisten WebUI:n sivujen eri osat (esimerkiksi merkkauksen syöttölomake ja esikatselu) on pyritty selkeästi erottamaan. Tähän tarkoitukseen on käytännöllinen div-elementti, jonka sisälle voidaan sijoittaa myös muita lohkotason elementtejä, kuten esimerkiksi julkaistavan uutisen leipäteksti (eli `<body>`:n lapsielementit) merkkauskielen esikatselussa. Uloimalle div:lle voidaan määritellä css:ssä esimerkiksi näkyvä reunus sen sisällön erottamiseksi sivun muusta tekstistä.

Lomakkeissa pyritään hyödyntämään fieldset-elementtiä samaan tarkoitukseen. Lisäksi lomakkeiden rakenteellisuutta lisättiin kenttien otsikkoja merkkaavilla label-elementeillä, jotka antavat selaimelle mahdollisuuden muuttaa kentän tilaa (esimerkiksi asetusnapeissa) tai kohdistaa kursorin syötekenttään labelin tekstiä klikattaessa. Labelia käytettiin myös lomakkeen ulkoasun rakentamiseen yhteistyössä CSS:n kanssa, millä saatiin aikaan siististi tasatut syötekentät ilman taulukolla taittamista. Lomakkeiden täytön helpottamiseksi kenttien labelit – textareaa lukuunottamatta – sijoitettiin kenttien vasemmalle puolelle, sillä se nopeuttaa lukemista [11, sivu 113].

WebUI:n vaatimuksissa ei ollut aina näkyvissä olevaa navigointipalkkia, joten emme edes lähteneet harkitsemaan kehyksien käyttöä niiden tuomien käytet-

tävyysongelmien vuoksi. Samaan lopputulokseen päästäisiin myös CSS2:n kiinteällä asettelulla (fixed positioning), mutta emme halunneet käyttää siteä, sillä esimerkiksi Windowsin MSIE ei vielä tue sen käyttöä.

4.2 Esteettömyys

HTML-sarjallistin kehitettiin osaksi järjestelmää jo järjestelmän rakentamisen alkuvaiheessa. Syynä tähän olivat Lynxin käyttö testaamisessa ja MSIE:n ongelmat palvellun XHTML:n kanssa kuten kappaleessa 3.2.3 jo mainittiin. Voidaankin sanoa, että HTML:n käyttö XHTML:n sijaan on vielä nykyisillä selaimilla välttämätöntä esteettömyyden kannalta.

WebUI:ssa pyrittiin käytettämään tyylisivuja siten, että css-tuen puuttuminen selaimesta ei estä WebUI:n toiminnallisuutta. Tyylisivujen lisäksi JavaScriptiä käytettiin hyvin varovaisesti eli ainoastaan helpottamaan käyttöä (katso kappale 4.3 alla). XML-HTTP -pyyntöjä (katso kappale 3.1.3) emme käyttäneet ollenkaan pitääksemme kiinni esteettömyydestä.

4.3 Asiakaspään ohjelmointikielet

Useimmissa WWW-selaimissa on tuki niin kutsutuille selainskripteille kuten esimerkiksi JavaScript, VBScript ja ActiveX. JavaScriptillä voidaan ohjelmoida vaikka koko WWW-käyttöliittymä muokkaamalla sivun sisältöä DOM-rajapinnan avulla ja tekemällä HTTP-pyyntöjä XML-HTTP:llä (katso kappale 3.1.3). Näiden selainskriptien lisäksi Java-sovelmilla (applet) sekä Flash-esityksillä voidaan luoda selaimiin oikeasti interaktiivisia ohjelmia, jotka antavat käyttäjälle kappaleessa 2.1 kaipaamani välittömän vasteen.

Edellä mainittuja teknologioita ei kuitenkaan tueta läheskään kaikissa selaimissa ja joissain selaimissa skriptit saattavat toimia vain valinnaisesti. Toisaalta selainskriptit ovat usein tarkoituksella kytketty pois päältä selaimista niiden tietoturvaongelmien takia [7, sivu 298]. Java-sovelmien ja Flashin ongelmana ovat myös vaikeasti asennettavat selain-pluginit ja koneelta vaadittava suorituskyky.

Päädyimme käyttämään selainskriptejä vain apukeinona sulavamman käyttöliittymän rakentamiseen ja valitsimme tähän JavaScript-kielen, joka on eri selainskripteistä parhaiten tuettu ja jota myös Korpela ja Linjama suosittelevat [7, sivu 299]. Pelkästään selainskripteihin ei WebUI:n toiminnoissa voida luottaa esteettömyyden säilyttämisen vuoksi. JavaScriptiä käytettiin WebUI:ssa mm. kursorin kohdistamiseen täytettävään lomakekenttään ja valintalistalomakkeen (option) automaattiseen lähettämiseen sen arvon muuttuessa. Jälkimmäisessä tapauksessa listan vieressä piti myös tarjota ”Lähetä”-painike, jotta lomake toimii myös ilman JavaScriptiä.

Emme halunneet tarkistaa lomakekenttien arvoja JavaScriptillä, sillä siitä olisi seurannut kaksinkertainen työmäärä: tarkistukset pitää kuitenkin suorittaa myös palvelinpäässä niitä selaimia varten, joissa JavaScript on pois päältä. Selaimessa tapahtuva lomakkeiden tarkistaminen onkin hyvä esimerkki välittömän

vasteen tarjoamisesta käyttäjälle, mutta kuitenkin hankala toteuttaa käytännössä täysin ongelmitta.

JavaScriptin vahvistusdialogit ovat myös ongelmallisia esteettömyyden kannalta, sillä toiminnallisuus ei saa nojautua niiden käyttöön. Emme käyttäneet niitäkään. Vahvistuksia kysytään yleensä vain tilanteissa, joissa pitäisi muutenkin käyttää POSTia (esimerkiksi tietokanta-alkion poistaminen). Tällaisessa tilanteessa JavaScriptin käytöstä aiheutuva GET-pyyntö on HTTP-spesifikaation vastainen (katso kohta 2.2.1).

4.4 POST vai GET?

WebUI:n kehityksessä haluttiin kiinnittää erityistä huomiota HTML-lomakkeiden oikean pyyntömetodin valitsemiseen kappaleessa 2.2.1 esitettyjen ongelmien – kuten selaushistorian tarpeettoman rikkomisen – ja eräiden varoittavien esimerkkien vuoksi. Kun käytetty metodi valitaan oikein, ei sovellukseen tarvitse rakentaa omaa, HTTP:n vastaista navigointilogiikkaa eikä pelätä, että selaimen ”Back”-painikkeen painamisella on arvaamattomia vaikutuksia. Esimerkiksi pääkaupunkiseudun uuden kirjastojärjestelmän Helmetin[5] asiakaspäätteitä käytetään WWW-käyttöliittymällä, jossa selaushistorian käyttö on estetty piilottamalla selaimen GUI-elementit ja estämällä muiden selainikkunoiden käyttämiänen. Estot voidaan kuitenkin kiertää, mikä johtaa uusiin käytettävyys- ja tietoturvaongelmiin.

WebUI:ssa käytetään GET-pyyntöä aina, kun mahdollista eli mikäli a) pyyntö-URI ei kasva ongelmallisen pitkäksi, b) sen parametreissa ei ole arkaluontoista tietoa ja c) lomakkeen lähetys ei voi aiheuttaa tietokantatransaktiota, joka muuttaisi sovelluksen tilaa. Muissa tapauksissa käytetään POSTia. Erityisesti ”Kyllä”/”Ei” -vahvistuksen kysymiseen käytetään POST-pyyntöä tarkoitusta varten abstrahoidussa omassa servletissään.

5 Yhteenveto

WebUI täytti sille asetetut vaatimukset asiakassovellusten laajennettavuuden sekä tietokanta-alkioiden syötön helppokäyttöisyyden ja esikatselun osalta. Myös eri käyttöliittymien laajennettavuus on helppoa WebUI:n nykyisellä koodipohjalla, sillä DOMServlet-, DOMDocument- ja ToolKit-luokat tarjoavat paljon valmista koodia uusien servlettien rakentamista varten.

DOMin käyttö WebUI:ssa osoittautui erittäin hyväksi ideaksi, vaikka aluksi sen käyttö arvelutti mahdollisesti suuren työmäärän tai huonomman suorituskyvyn vuoksi. Merkkijonojen käsittelyä pelkillä servleteillä kokeiltiin hieman, mutta eskapoinnin työläys ja merkkijonomanipulaation ongelmat antoivat heti hyvän motivaation ainakin DOMin kokeilua varten. Ajateltiin myös, että DOMin käyttö pelkän WebUI:n tarpeisiin on yliampuvaa, mutta toisaalta DOMista olisi

[5] Katso `http://www.helmet.fi`

suuri hyöty integroitaessa WebUI:ta WWW-julkaisujärjestelmän muiden osien kanssa.

Asiakaskäyttöliittymää lukuunottamatta WebUI:ta oli mahdollista kehittää melko itsenäisesti puuttumatta julkaisujärjestelmän muihin osiin. Tämän mahdollisti paljolti onnistunut DBS-rajapinta, johon ei tarvittu kovin suuria muutoksia. Kuitenkin asiakaskäyttöliittymää ja sen laajennettavuutta suunniteltaessa jouduimme tekemään tiivistä yhteistyötä asiakassovellusten kehittäjän kanssa. Asiakassovellusten kehittäminen vaatii taas yhteistyötä template-enginen kehittäjän kanssa, jotta template-kieltä tulee käytettyä oikein.

Tämän seurauksena varsinkin suurempiin asiakaskäyttöliittymää koskeviin kehitysaskeliin tarvittiin WebUI:n, asiakassovelluksen ja TE:n kehittäjiä, mikä teki etenemisestä ajoittain raskasta. Model-abstraktio kuitenkin vähensi servlettikoodin riippuvuutta asiakassovelluksista ja helpotti tilannetta. Lisäksi asiakassovelluksiin liittyvät osat erotettiin muista WebUI:n osista eri pakettiin (servlets.apps).

Nykyisen MVC-toteutuksen voisi myös mainita yhtenä WebUI:n huonona puolena, sillä se ei tarjoa yhtä kaunista abstraktiota eri osiensa välillä kuin Struts ja se koskee vain asiakassovellusten käyttöliittymää. Strutsin mukauttamiseen tämän projektin tarpeisiin ei kuitenkaan ollut aikaa.

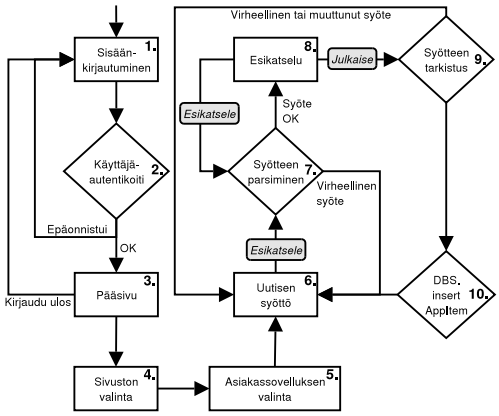
5.1 Laajennettavuus

WWW-julkaisujärjestelmän laajennettavuudelle uusien asiakassovellusten osalta luotiin hyvät puitteet. Asiakassovelluksen Foo lisäämiseksi järjestelmään kirjoitetetaan kolme uutta luokkaa:

- Asiakassovelluksen vaatima DOMin käsittely TE:tä ja WebUI:ta varten kirjoitetaan application-pakettiin:
  `FooApplication extends Application`.
- Model-luokka, joka lukee asiakassovelluksen tietokanta-alkion datan tietokannasta tai HTML-lomakkeesta ja populoi tarvittaessa lomakkeen tai tietokantarivin, luodaan webui.servlets.apps-pakettiin:
  `FooModel implements WebModel`.
- Controller-luokka, joka palvelee lomakkeen, ottaa lomakkeen vastaan ja käsittelee sen modelin avulla, kirjoitetaan webui.servlets.apps-pakettiin:
  `Foo extends FooHandler`
- Lisäksi HTML-lomakkeen merkkaus kirjoitetaan XHTML-pohjaan `foo.xml`, joka sijoitetetaan apps-hakemistoon.

Valmiiden kolmen luokan .class-tiedostot voidaan asettaa pakettinimen määrämään hakemistoon ajonaikaisesti ja tämän jälkeen tieto uudesta asiakassovelluksesta voidaan lisätä tietokantaan WebUI:n operaattorikäyttöliittymän kautta. Vasta tämän jälkeen voidaan luoda asiakassovellusinstansseja jollekin järjestelmässä olevalle domainille.

5.2 Asiakaskäyttöliittymän toiminta



Kuva 4: Vuokaavio yhden uuden uutisen lisäämisestä

Kuvassa 4 on esitetty hieman yksinkertaistettuna yhden uuden uutisen lisääminen WebUI:lla. Suorakaiteet kuvaavat yksittäisiä servletin tarjoamia sivuja ja salmiakit HTTP-pyyntöjen käsittelyä.

Vaiheessa 2. autentikoidaan käyttäjä annetun nimen ja salasanan perusteella DBSAuthenticator-luokan avulla. Mikäli autentikointi onnistuu, tallennetaan saatu DBSUser-olio sessioon talteen. DBSUser-oliolta saadaan niiden sivustojen nimet, joihin käyttäjällä on oikeus puuttua. Nämä sivustot esitetään pudotusvalikkona kohdassa 4.

Kohdassa 5. haetaan valittuun sivustoon liitettyjen asiakassovellusinstanssien nimet tietokannasta ja esitetään käyttäjälle pudotusvalikko, josta tämä valitsee instanssin, johon uusi uutinen halutaan lisätä. Tässä esimerkissä instanssin nimi on ”uutiset”. Kohdissa 6.-10. lähettää selain pyynnöt clients-servletille, joka instantioi aina asiakassovelluksen controllerin. Ensin clients-servletti kysyy tietokannasta, minkä asiakassovelluksen instanssi ”uutiset” on ja saa vastaukseksi asiakassovelluksen ”news”. Tämän perusteella Clients instantioi News-olion: `AppHandler handler=(AppHandler)Class.forName("News").newInstance();`

Tämä News-olio lukee kohdassa 6. uutisen lisäys -lomakkeen news.xml-tiedostosta ja sarjallistaa sen selaimelle. Käyttäjä syöttää lomakkeeseen uuden uutisen ja painaa Esikatsele-painiketta. Tämän seurauksena News-olio luo uuden NewsModel-olion, joka lukee datansa HTTP-pyynnön parametreista. Kohdassa 7. NewsModel luo uuden NewsApplication-olion ja kutsuu tämän `buildDOM`-metodia, joka jäsentää lomakkeella syötetyn datan ja heittää SAXExceptionin, jos syötteessä on virhe. Virhetilanteessa poikkeuksen syy esitetään käyttäjälle ja lomake roundtrippaa siten, että edellä syötetty data on lpjdam 6. lomakkeen kenttien alkuarvoina.

Vasta kun syöte on todettu oikeelliseksi päästään kohtaan 8., jossa controller littää uutisen leipätekstiä vastaavan DOM-puun vasteena annettavan sivun puuesityksessä ennalta luodun div-elementin lapseksi. Myös uutisen syöttölomake

on täytetty edellä annetulla datalla. Käyttäjä voi halutessaan muokata lomaketta ja valita joko Esikatsele- tai Julkaise-komennon. Julkaise-komennon seurauksena tarkistetaan kohdassa 9., onko lomakkeessa oleva leipäteksti sama kuin esikatseltu leipäteksti. Käyttäjän on siis pakko esikatsella syöttämänsä uutinen ennen kuin se voidaan julkaista.

Kun uutinen on esikatseltu ja sen leipätekstin merkkaus on vielä tarkistettu oikeelliseksi, kutsuu controller kohdassa 10. NewsModelin `populateDatabase`-metodia, joka luo datasta DBSItem-olion ja lisää sen tietokantaan DBSInterfacen `insertAppItem`-metodilla. Onnistuneesta uutisen julkaisusta näytetään käyttäjälle infoviesti uuden, tyhjän uutisen syöttölomakkeen ohessa kohdassa 6.

Viitteet

[1] Bray Tim, Paoli Jean, Sperberg-McQueen C. M. & Maler Eve. Extensible Markup Language (XML) 1.0 (Second Edition). Lokakuu 2000. [viitattu 20.1.2004]. URL: http://www.w3.org/TR/REC-xml.

[2] Dalton Sam. Complete the MVC Puzzle with Struts. SitePoint. Marraskuu 2003. [viitattu 22.1.2004]. URL: http://www.sitepoint.com/print/1244.

[3] Fielding Roy T, Gettys James, Mogul Jeffrey C., Nielsen Henrik Frystyk, Masinter Larry, Leach Paul J. & Berners-Lee Tim. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. Kesäkuu 1999. [viitattu 4.1.2004]. URL: http://www.ietf.org/rfc/rfc2616.txt.

[4] Flavell Alan. FORM submission and i18n. Joulukuu 2003. [viitattu 20.1.2004]. URL: http://ppewww.ph.gla.ac.uk/~flavell/charset/form-i18n.html.

[5] Gaedke Martin & Gellersen Hans-W. Object-oriented Web Application Development. IEEE Internet Computing. Tammikuu/helmikuu 1999 (Vol. 3, No. 1). pp. 60-68. [viitattu 18.1.2004]. URL: http://www.teco.edu/~gaedke/paper/1999-ieee-ic.pdf. ISSN 1089-7801.

[6] Inderjeet Singh, Beth Stearns, Mark Johnson & the Enterprise Team. Kesäkuu 2002. Designing Enterprise Applications with the J2EE™ Platform. 2. painos. Addison-Wesley Pub Co. 352 sivua. ISBN 0201787903. [viitattu 25.1.2004]. URL: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/.

[7] Korpela Jukka K. & Linjama Tero. 2003. Web-suunnittelu. 1. painos. Docendo Finland Oy. 457 sivua. ISBN 951-846-172-4.

[8] Ley Jim. Using the XML HTTP Request object. Huhtikuu 2002. [viitattu 20.1.2004]. URL: http://jibbering.com/2002/4/httprequest.html.

[9] Manola Frank. Towards a Web Object Model. Helmikuu 1998. [viitattu 4.1.2004]. URL: http://www.objs.com/OSA/wom.htm.

[10] Manola Frank. Technologies For a Web Object Model. IEEE Internet Computing. Tammikuu/helmikuu 1999 (Vol. 3, No. 1). pp. 38-47. [viitattu 19.1.2004]. URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=16134. ISSN 1089-7801. (artikkeli luettavissa hut-domainissa).

[11] Parkkinen Jarmo. 2002. Hyvään verkkopalveluun!. 1. painos. Inforviestintä Oy. 163 sivua. ISBN 952-5123-42-1.

[12] Pemberton Steven ym. XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). Elokuu 2002. [viitattu 30.1.2004]. URL: http://www.w3.org/TR/xhtml1/.

[13] Ragget Dave, Le Hors Arnaud & Jacobs Ian. HTML 4.01 Specification. Joulukuu 1999. [viitattu 4.1.2004]. URL: http://www.w3.org/TR/html4/.

[14] Sivonen Henri, Kari-Koskinen Yrjö & Koskela Jussi. Assembling Web Pages Using Document Trees. Huhtikuu 2004. URL: http://iki.fi/hsivonen/cms/te.html

[15] Williams Al. Design Patterns for Web Programming, Do you need MVC?. New Architect. Kesäkuu 2002. CMP Media LLC. [viitattu 24.1.2004]. URL: http://www.newarchitectmag.com/documents/new1020217692284/.

[16] Wood Lauren. Programming the Web: The W3C DOM Specification. IEEE Internet Computing. Tammikuu/helmikuu 1999 (Vol. 3, No. 1). pp. 48-54. [viitattu 18.1.2004]. URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=16134. ISSN 1089-7801. (artikkeli luettavissa hut-domainissa).