Opinnäytetyö (AMK)

Tietotekniikan koulutusohjelma

Sulautetut ohjelmistot

2017

Janne Virtanen

WEB-SOVELLUSKEHYKSEN EROTTAMINEN SULAUTETUN JÄRJESTELMÄN KÄYTTÖLIITTYMÄSTÄ

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautetut ohjelmistot

2017 | 61

Tiina Ferm, Jari-Pekka Paalassalo, Marko Vilola

Janne Virtanen

WEB-SOVELLUSKEHYKSEN EROTTAMINEN SULAUTETUN JÄRJESTELMÄN KÄYTTÖLIITTYMÄSTÄ

Sovelluskehykset nopeuttavat ohjelmistojen kehitystä, mutta niiden luominen on haastava ja riskialtis projekti. Kehysten suunnittelussa ennakoidaan tulevia käyttötarpeita, mikä on usein vaikeaa. Muutosten tekeminen ohjelmiin voi tulla kalliiksi, koska kehykset lukitsevat aina sovellusten arkkitehtuurin. Onnistuessaan kehyksillä voi kuitenkin luoda kustannustehokkaasti yhdenmukaisia, helposti ylläpidettäviä ohjelmia.

Tämän opinnäytetyön tutkimusosassa perehdyttiin sovelluskehyksiin ja joihinkin yleisiin web-arkkitehtuurien suunnittelumalleihin. Työn käytännön osassa toimeksiantajalle luotiin web-sovelluskehys. Kehystä ei kuitenkaan tehty alusta lähtien, vaan se erotettiin olemasta olevasta sulautetun järjestelmän käyttöliittymästä. Tutkimuksesta saatua tietoa sovellettiin käyttöliittymän arkkitehtuurin analyysissa, minkä jälkeen erotustyö suunniteltiin ja toteutettiin onnistuneesti. Suuria muutoksia arkkitehtuuriin ei lopulta tehty, mutta pienempiä parannuksia toteutettiin sovellusytimeen, istunnonhallintaan ja alustukseen. Uutena komponenttina kehykseen luotiin käännösohjelma, jolla kehys ja sovellus voidaan yhdistää. Kehykseen luotiin lisäksi testipalvelin käyttöliittymien testaamiseen.

Kehystä sovellettiin luomalla sen avulla käyttöliittymä kahteen tuotteeseen. Molemmissa tapauksissa kehys osoittautui toimivaksi työkaluksi. Muunneltavuuden hallinnassa havaittiin kuitenkin ongelmia, jotka hidastivat kehitystyötä. Myös suorituskyvyssä oli puutteita. Ymmärrettiin, että erotustyön aikana olisi kannattanut käyttää enemmän aikaa arkkitehtuurin ongelmien korjaamiseen. Käsitys kehysprojektien haastavuudesta osoittautui työn kohdalla paikkansa pitäväksi.

ASIASANAT:

sovelluskehykset, ohjelmistokehykset, web-sovellus, sulautettu tietotekniikka, ohjelmistoarkkitehtuuri, suunnittelumallit, JavaScript, Node.js

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Computer technology | Embedded software

2017 | 61

Tiina Ferm, Jari-Pekka Paalassalo, Marko Vilola

Janne Virtanen

SEPARATING WEB APPLICATION FRAMEWORK FROM THE USER INTERFACE OF AN EMBEDDED DEVICE

Designing an application framework is commonly regarded as a hard task. The process requires forward planning which always has risks involved. However, if the investment pays off, applications of similar structure can be built cost-efficiently with lower maintenance overhead.

This thesis presents a study on architecture design of application frameworks with emphasis on the web platform. The theory is put into practice and the work describes a process in which a web application framework was created for internal company use. The framework was not built from scratch; an existing user interface of an embedded device was used as a basis for the framework.

After an architectural analysis, separation of the code base was planned and successfully implemented. The work was completed quickly and the overall architecture was left mostly unchanged. Minor changes were implemented to the application initialization and the session management components. A new build system, which enables development on Windows and Linux platforms was implemented. A server platform that can be used to test the user interfaces was also created.

Two user interfaces were created with the framework. While the framework was considered a useful tool, some problems were found. While difficulty in making architectural changes was the most severe problem, performance and data synchronization had some issues as well.

A learning outcome of this developmental work was that reserving enough time and resources to make necessary architectural changes at the earliest is almost always the preferred approach to take. This thesis can also be considered a case study that confirms the initial expectation that application framework design is a difficult process.

KEYWORDS:

web application framework, embedded system, software architecture, design patterns, JavaScript, Node.js

SISÄLTÖ

KUVAT

TAULUKOT

KÄYTETYT LYHENTEET JA SANASTO

API	Ohjelmointirajapinta (Application Programming Interface)
Autotools	Unix-alustariippumaton kehitystyökalu, jolla ohjelma voidaan kääntää ja asentaa
CAM	Videosignaalin salaus- ja purkumoduuli (Conditional Access Module)
CATV	Kaapelitelevisio (Cable Television)
CMTS	Kaapelimodeemien päätelaite (Cable Modem Termination System)
CRUD	Pysyväistietoon suoritetut perusoperaatiot: luonti, luku, päivitys ja poisto (Create, Read, Update, Delete)
CSS	Web-dokumenttien esitysasun tyylien määritykseen käytetty merkkauskieli (Cascading Style Sheets)
DOM	Dokumenttioliomalli, kuvaa web-dokumentin dynaamisesti muokattavan elementtipuun (Document Object Model)
DVB	Digitaalisen videon siirtostandardi (Digital Video Broadcasting)
EPG	Sähköinen TV-ohjelmaopas (Electronic Program Guide)
Express.js	Node.js-kehys, jolla voi luoda web-palvelimia
Grunt	Node.js-ohjelma, jolla voi automatisoida ohjelmistokehityksen tehtäviä (JavaScript Task Runner)
HTML	Web-sivujen kuvauskieli (HyperText Markup Language)
HTTP	Tekstipohjainen tiedonsiirtoprotokolla (HyperText Transfer Protocol)
JSDoc	Merkintäkieli JavaScriptin dokumentointiin
JSON	Tiedon siirto- ja säilytysformaatti (JavaScript Object Notation)
Node(.js)	Palvelinpään järjestelmäriippumaton JavaScript-suoritusympäristö
npm	Node.js paketinhallintaohjelma, joka tulee Noden mukana
Refaktorointi	Lähdekoodin muokkaaminen helpommin ymmärrettävään tai yksinkertaisempaan muotoon ilman ulospäin näkyviä sivuvaikutuksia
REST tai RESTful	HTTP-protokollan verbeihin perustuva arkkitehtuurimalli sovellusrajapintojen toteutukseen (Representational State Transfer)
Tekninen velka	Nopeita ratkaisuja myöhemmin viiveellä seuraava kehitystyön hidastuminen ja vaikeutuminen
Toiminnallisuustestaus	Ohjelmiston toiminnallisuuden varmistaminen (Functional testing)
URI	Yksikäsitteinen merkkijonoviittaus web-resurssiin, esimerkiksi web-osoite (Uniform Resource Identifier)
XML-RPC	XML-koodattu etäproseduurikutsu HTTP:n yli
Yksikkötestaus	Lähdekoodin moduulien toiminnan varmistaminen (Unit testing)

1 JOHDANTO

1.1 Työn tausta ja tavoitteet

Sulautettujen järjestelmien käyttörajapintaan ei aina kiinnitetä tuotekehityksen alussa erityistä huomiota. Käyttöliittymän toteutukseen varataan vähän, jos lainkaan, resursseja. Saatetaan katsoa, että mahdollisuus laitteen asetusten muuttamiseen tai tilatietojen lukemiseen on riittävä määrä toiminnallisuutta käyttörajapinnassa. Tällöin etäyhteyden yli suoritettava komentoriviohjelma tai yksinkertainen verkkoprotokolla voi jo riittää rajapinnaksi. Monien sulautettujen järjestelmien kohdalla tällainen ratkaisu on hyvin tarkoituksenmukainen. Käyttäjältä vaaditaan kuitenkin paljon asiantuntemusta laitteen hallinnointiin, eikä helppokäyttöisyydestä voi puhua.

Laitteen asiakaskunnan kasvaessa voi syntyä painetta kehittää käyttörajapintaa. Komentoriviohjelman rinnalle aletaan ehkä kaavailla monipuolisempaa käyttöliittymää, mikä Internetin aikakaudella voi tarkoittaa, että käyttöliittymä toteutetaan selainpohjaisesti. Rajallisten laiteresurssien päälle ei ole jälkikäteen kuitenkaan välttämättä helppo rakentaa toimivaa kokonaisuutta. Käyttöliittymän toteutuksesta saattaa helposti tulla ad hoc -tyylinen ratkaisu. Yleiskäyttöisen toteutuksen tekemiseen tuskin uhrataan lainkaan aikaa.

Käyttöliittymän toteutukseen olisikin hyvä varautua jo varhaisessa vaiheessa laitteen suunnittelua. Parhaiten tämä onnistuu, jos kehittäjien saatavilla on konsepti sovellusalueella usein tarvituista toiminnoista ja konseptia vastaava osittainen toteutus. Tällainen yleiskäyttöinen, kehitystyötä nopeuttava *sovelluskehys* on aina mahdollista kehittää olemassa olevan tuotteen käyttöliittymän pohjalta tai luoda alusta lähtien. Tällöin tuotekehitysryhmän tulee soveltaa kokemuksen myötä kertyneitä ratkaisujaan ja näkemyksiään sovelluskehyksen suunnittelussa ja toteutuksessa.

Tämän opinnäytetyön toimeksiantajan tilanne oli ollut jokseenkin edellä kuvatun kaltainen, ja siitä oli syntynyt tarve luoda nykyisiä ja tulevia tuotteita varten *web-käyttöliittymäkehys*. Uusi kehys haluttiin erottaa olemassa olevan videosignaalien prosessointiin erikoistuneen laitteen[1] käyttöliittymästä. Lisätavoitteeksi asetettiin uuden käyttöliittymän luominen sovelluskehyksellä johonkin toiseen tuotteeseen.

1 Tuotenimi Luminato – laite ja sen käyttöliittymä esitellään luvussa 2.4.

1.2 Työn toteutuksesta ja sisällöstä

Työ toteutettiin toimeksiantona Teleste Oyj:lle[2]. Tuotekehitysprojektiin osallistui lisäkseni kaksi kokenutta ohjelmistosuunnittelijaa, joiden osuus kehyksen erotustyöstä oli merkittävä. Osuuteni kokonaistyömäärästä ei ollut lähelläkään puolta, ja suunnittelutyöstä osuus oli sitäkin pienempi. Kokonaan omaa työpanostani oli kuitenkin uuden käännösohjelman ja testipalvelimen toteutus.

Työn tekstiosuuden luvussa 3 on selostettu sovelluskehyksen erottamistyön suunnitteluprosessi ja varsinainen toteutus. Erotustyön lisäksi työn tutkimusosuudessa luvussa 2 on perehdytty sovelluskehyksiin ja joihinkin työn kannalta keskeisiin suunnittelu- ja arkkitehtuurimalleihin. Tutkimuksen painopiste on työn kannalta keskeisissä web-käyttöliittymissä.

Sovelluskehyksen varsinainen erotustyö alkoi vuonna 2014, ja opinnäytetyöhön rajattu osuus valmistui myöhemmin samana vuonna. Sovelluskehyksen kehitystyötä jatkettiin kuitenkin myös erotustyön valmistumisen jälkeen.

Opinnäytetyön tekstiä on työstetty sovelluskehyksen jatkokehityksen kanssa rinnakkain, toisinaan aktiivisemmin ja toisinaan useiden kuukausien taukojen jälkeen. Teksti on näin ollen saattanut saada joiltain osin vaikutteita kehyksen myöhemmistä vaiheista. Tekstin tarkoitus on kuitenkin selvittää työtä siinä muodossa, jossa se valmistui vuoden 2014 lopussa. Poikkeuksen tähän tekee kuitenkin lopputuloksia käsittelevä luku 4, joka sisältää retrospektiivisen tarkastelun kehyksen soveltamisesta kahden tuotteen osalta.

2 Teleste kehittää ja valmistaa video- ja laajakaistateknologisia laite-, ohjelmisto- ja palveluratkaisuja kansainvälisille markkinoille.

2 ANALYYSI

2.1 Sovelluskehykset

Sovelluskehyksellä tarkoitetaan oliokielissä joukkoa uudelleenkäytettäviä komponentteja, jotka yhdessä muodostavat arkkitehtuuriltaan yhdenmukaisen kokonaisuuden tietyllä sovellusalueella. Sovelluskehys määrittää komponenttien luokkien vastuualueet, niiden väliset riippuvuudet sekä ohjelmointirajapinnan, jota vasten erikoistunut sovellus luodaan sovelluskehystä laajentamalla. Komponenttien uudelleenkäyttö nopeuttaa kehitystyötä, koska suunnittelijoiden vastuulle jää parhaassa tapauksessa ainoastaan sovelluskohtainen toteutus. [1, ss. 26–28]

Uudelleenkäytettävien komponenttien lisäksi sovelluskehys edustaa ennen kaikkea joukkoa sovellusalueella hyväksi havaittuja suunnitteluratkaisuja. Sovelluskehys ohjaa sovelluksen kehitystyötä ja määrittää sen arkkitehtuurin perustan. Usein arkkitehtuuriratkaisujen uudelleenkäyttö on sovelluskehyksien hyötyjä ajatellen jopa tärkeämpää kuin komponenttien uudelleenkäyttö. [1]

Sovelluskehyksiä laajempi käsite on *tuoterunko* ja sitä vastaava *tuoterunko-arkkitehtuuri*. Tuoterungolla tarkoitetaan tiettyä tuotekategoriaa varten suunniteltua ohjelmistoalustaa, jonka arkkitehtuuri ohjaa yksittäisten tuotteiden kehitystyötä tiettyjen rajojen sisällä ja nopeuttaa tuotteiden kehitystyötä. Sovelluskehys on siis oikeastaan vain yksi suosittu tapa toteuttaa tuoterunkoarkkitehtuuri. Tässä tekstissä sovelluskehyksiä käsitellään olioparadigman näkökulmasta, jolloin puhutaan yleisesti *olioperustaisista ohjelmistokehyksistä*. Sovelluskehyksen käsite ei ole sidottu olioparadigmaan, mutta oliokielten ominaisuudet, kuten periytyminen ja dynaaminen sidonta soveltuvat niin hyvin tuoterunkoarkkitehtuurien tavoitteiden täyttämiseen, että yleensä sovelluskehyksillä tarkoitetaan nimenomaan oliopohjaisia kehyksiä. [2, Luku 7–8]

Sovelluskehyksellä on yleensä päävastuu ohjelman kontrollista [2, Luku 8.1.4]. Kehys delegoi tällöin sovelluskohtaiset tapahtumat rekisteröidyille käsittelijöille tai kutsuu rajapintojen kautta sovelluksen toteuttamia funktioita. Järjestelmätason tapahtuma esikäsitellään jo sovelluskehyksen omassa tapahtumasilmukassa. Jos tapahtuma on sovellusalueen kannalta keskeinen ja vaatii erikoistuneempaa käsittelyä, kehys luo tapahtumasta uuden, tarkoituksenmukaisemman abstraktion. Sovelluskehittäjä voi näin

käyttää abstraktia, sovellusalueessa hyvin määriteltyä rajapintaa haluamallaan tavalla. Erityisesti tällainen *käänteinen kontrolli* (engl. inversion of control) erottaa sovelluskehyksen tavanomaisesta uudelleenkäytettävästä ohjelmistokirjastosta[3]. [1]

Okanovićin ja Mateljan mukaan [3] sovelluskehysten keskeisimpiä hyötyjä ovat seuraavat:

– *Modulaarisuus*. Kehys enkapsuloi toteutuksen yksityiskohdat rajapintojen taakse. Ohjelmiston laatu paranee, koska kehyksen toteutusta on helpompi muokata koskematta sovelluksiin.

– *Uudelleenkäytettävyys*. Kehys sisältää rajapinnat ja toteutukset sovellusalueella yleisiin käyttötapauksiin. Sovelluskehittäjien ei tarvitse keskittyä jo ratkaistuihin ongelmiin, jolloin tuottavuus kasvaa ja sovellusten laatu paranee.

– *Laajennettavuus*. Kehys ohjaa kehittäjiä tarkoituksenmukaisiin ratkaisuihin, mutta tarjoaa myös mahdollisuuden laajentaa olemassa olevia ratkaisuja.

– *Käänteinen kontrolli*. Sovelluksen ohjaus on sovelluskehyksen vastuulla, mutta sovellus voi reagoida tapahtumiin takaisinkutsujen tai uudelleentoteutusten avulla.

Sovelluskehys voi olla toteutettu toisten erikoistuneempien kehysten avulla. Laajoissa kehyksissä avustavia kehyksiä voi olla lisäksi useita. Jokaisella kehyksellä on tällöin oma vastuualue, jossa se toteuttaa tähän alueeseen soveltuvan arkkitehtuurin ja laajennusrajapinnan. Kehykset voivat olla myös kerrosteisia tai luoda hierarkioita. Alustariippumaton käyttöliittymäkehys voi sisältää esimerkiksi ohjelmointikielten kääntämiseen, 2D- ja 3D-grafiikan piirtoon ja verkkokommunikointiin liittyviä sovelluskehyksiä. [2, Luku 8.3]

2.2 Sovelluskehyssuunnittelu

Uuden sovelluskehyksen luominen on aikaa ja resursseja kuluttava hanke. Suunnittelun vaikeusasteikossa se on kaikkein haastavin mahdollinen ohjelmistoprojekti, johon voidaan ryhtyä [1, s. 27]. Se on iteratiivinen prosessi, joka syntyy joskus yhden, mutta yleensä useamman yksittäisen valmiin sovelluksen pohjalta [3].

3 Kirjastot ainoastaan tarjoavat ratkaisuja ja palveluja tietyssä ongelmakentässä, eivätkä aseta sovellukselle erityisiä vaatimuksia.

Suunnittelijoilta edellytetään kykyä havaita sovellusalueella toistuvia säännönmukaisuuksia ja kykyä löytää niille toimivat rajapinnat kehyksen arkkitehtuurissa.

Hyvän sovelluskehyksen suunnittelun yksi avaintekijä onkin suunnittelutiimin kokemustaso. Jotta sovelluskehyksen yleisiä rajapintoja ja luokkia voi ylipäänsä löytää tehokkaasti ja yrityksen kannalta tuottavalla tavalla, edellytyksenä on suunnittelijoiden aikaisempi kokemus tuotteiden kehityksestä tietyllä sovellusalueella. [3]

Okanovićin ja Mateljan mukaan [3] sovelluskehyksen luominen edellyttää

– tarkkaa ymmärrystä käyttäjien tarpeista

– sovellusalueen asiantuntemusta

– asiantuntijuutta oliomallinnuksesta ja arkkitehtuurisuunnittelusta

– riittävää tukea johdolta kalliille, pitkäkestoiselle ja iteratiiviselle kehitystyölle.

Sovelluskehysten päälle toteutetut sovellukset ovat aina pysyvästi riippuvaisia kehyksestä. Toisaalta sovellusten toteutuksen joustavuus riippuu suoraan kehyksen laajennettavuudesta. Jos asiaa tarkastellaan toiseen suuntaan, jokainen sovellus lukitsee aina myös osan sovelluskehyksen rakenteesta. Kehysten suunnittelussa keskeinen haaste onkin muunneltavuuden hallinta [2, Luku 7.7]. Ei ole olemassa yksikäsitteistä sääntöä, miten paljon tai miten kehystä tulee voida mukauttaa sovelluksia varten. Suunnittelun lähtökohta on aina tilannekohtainen. On kuitenkin olemassa tekniikoita, joilla muunneltavuutta voidaan hallita myös sovelluskehysten suunnittelussa. *Suunnittelumalleilla* on tässä suhteessa keskeinen rooli, koska ne tukevat muunneltavuutta ja joustavuutta [2, Luku 8.1.3].

Sovelluskehys eriyttää sovelluskehityksen kahteen toisistaan eroavaan kokonaisuuteen. Koska toisiinsa liittyvien komponenttien kehitysvastuu ajautuu organisaatioissa usein yksittäiselle ryhmälle, kokonaisuuksiin eriytymisestä seuraa monesti myös kahden kehitysryhmän muodostuminen [2, Luku 1.5]. Tällöin *arkkitehtuuriryhmän* vastuulle tulee sovelluskehyksen kehitys- ja ylläpitotyö, ja *tuotekehitysryhmien* vastuulle siirtyy varsinaisten sovellusten kehitystyö. Ryhmien erikoistuminen edellyttää tehokasta kommunikaatiota ryhmien välillä [2, Luku 7.6]. Jos resursseja on niukasti saatavilla, on arkkitehtuuriryhmän erityisesti pystyttävä vastaamaan nopeasti juuri niihin tarpeisiin, joita tuotekehitysryhmillä kulloinkin on.

Tästä syystä arkkitehtuuriryhmälle onkin eduksi omakohtainen kokemus myös sovellusten kehitystyöstä ja sen haasteista.

Sovelluskehys voidaan aluksi luoda yksittäisen sovelluksen tarpeita ajatellen, mutta tällöin voidaan harvoin varmistua tehtyjen yleistyksien toimivuudesta tai tarkoituksenmukaisuudesta. Yleensä vasta toisen tai kolmannen erilaisen sovelluksen luonnin jälkeen tiedetään tarkemmin, mitä komponentteja todella tarvitaan ja sovelluskehyksen toimivuutta voidaan todella arvioida. Hyvä ja onnistunut sovelluskehys maksaa yleensä kehitystyön investoinnit lopulta takaisin, mutta tämä edellyttää sitä todella käytettävän riittävän monen sovelluksen kehitystyöhön. [3]

2.3 Suunnittelu- ja arkkitehtuurimallit

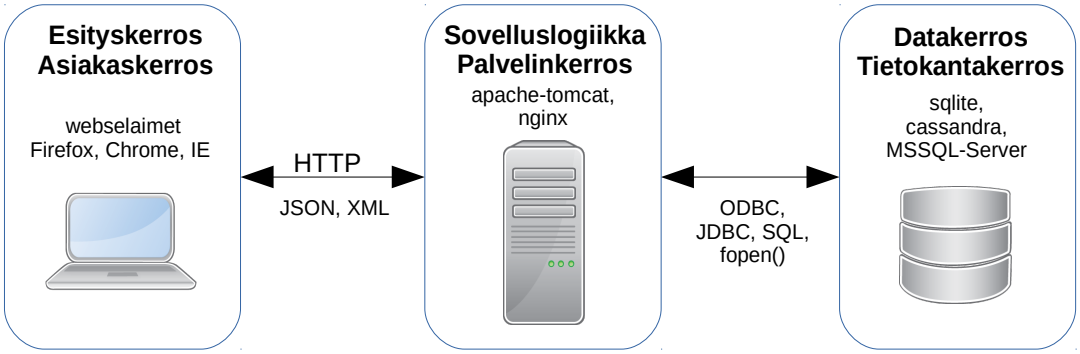
Suunnittelumallit ovat yleisesti tunnettuja, dokumentoituja, tarkkaan rajattuja, abstrakteja ratkaisumalleja usein toistuviin suunnitteluongelmiin tietyn viitekehyksen sisällä. [1, Luku 1]

Arkkitehtuurimallit tai *-tyylit* ovat pohjimmiltaan suunnittelumallin idean yleistyksiä sovelluksen kokonaisarkkitehtuuriin. Siinä missä yksittäistä suunnittelumallia voidaan soveltaa toistuvasti sovelluksen eri komponenteissa, arkkitehtuurimalli kuvaa tietyn toteutusratkaisun rakenteen koko sovelluksessa. Tietyn ratkaisun tunnistaminen suunnittelu- tai arkkitehtuurimalliksi ei välttämättä ole kuitenkaan aina niin itsestään selvää. [2, Luku 6]

Malleja käytetään paljon järjestelmien arkkitehtuurisuunnittelussa, eivätkä sovelluskehykset ole tässä poikkeus. Monet arkkitehtuureihin liittyvät ongelmat tunnetaan varsin hyvin ja niihin löytyykin runsaasti valmiita malleja. Tässä luvussa on esitelty joitakin tunnettuja arkkitehtuurimalleja, jotka ovat työn kannalta keskeisiä. Yksittäisiä suunnittelumalleja ei tässä tekstissä esitellä, mutta aiheesta löytyy erittäin paljon kirjallisuutta, josta hyvänä esimerkkinä käy viite [1].

2.3.1 Kolmikerros- ja asiakas-palvelin-arkkitehtuurimalli

Yleinen 3-kerrosarkkitehtuuri kuvataan erillisinä loogisina kerroksina: esityskerros, sovelluslogiikkakerros ja datakerros (kuva 1). Jokaisella kerroksella on oma tehtävä tai rooli. Kerrokset koostuvat komponenteista, joista osa tarjoaa rajapinnan ylemmille kerroksille, ja osa käyttää itse alempien kerrosten komponenttien tarjoamia palveluita. Kerroksiin jakamisella on useita etuja: arkkitehtuuria on helpompi ymmärtää, sovelluksen sisäiset riippuvuudet vähenevät ja komponenttien uudelleenkäyttö on helpompaa. [4, ss. 29–30, 118]



Kuva 1. Web-sovellusten 3-kerrosarkkitehtuurin kerrokset.
Kuvassa on listattu tyypillisiä kerrosten suoritusalustoja ja tiedonsiirtoprotokollia.

Loogisen kerroksen ohjelmakoodin suorituspaikka ei ole fyysisesti sidottu: kaikki kerrokset voivat olla samassa tietokoneessa, mutta yhtä hyvin kerrokset voivat olla täysin erillään toisistaan, kuten kuvassa 1 on havainnollistettu. Kerroksen toteutusta voi sijaita myös useammassa kuin yhdessä fyysisessä kohteessa. Englanninkielisessä kirjallisuudessa loogiset ja fyysiset kerrokset erotetaankin usein toisistaan käyttämällä termejä *layer* ja *tier*. [4, s. 29]

Seuraavassa on esitetty loogisten kerrosten tehtävät web-sovellusten viitekehyksessä.

Esityskerros sisältää sovelluksen käyttöliittymän. Kerros huolehtii sovellustiedon lataamisesta palvelimelta ja esittää tiedon näkymien kautta käyttäjälle. Esityskerros reagoi myös käyttäjän syötteisiin, joiden seurauksena kerros voi joutua kommunikoimaan alempien kerrosten kanssa. Koska selaimessa suoritettava ohjelmakoodi ladataan aina palvelimelta, esityskerroksen voi ymmärtää sijaitsevan fyysisesti sekä selaimessa että palvelimessa. Sovelluslogiikkaa ei yleensä ole sijoitettu esityskerrokseen lainkaan.

Sovelluslogiikkakerros sisältää sovellusaluekohtaisen toteutuksen. Se koostuu palvelukomponenteista ja muista sisäisistä prosesseista. Palvelukomponentit vastaavat esityskerroksesta tuleviin pyyntöihin, mutta toisaalta käyttävät itse datakerroksen palveluja.

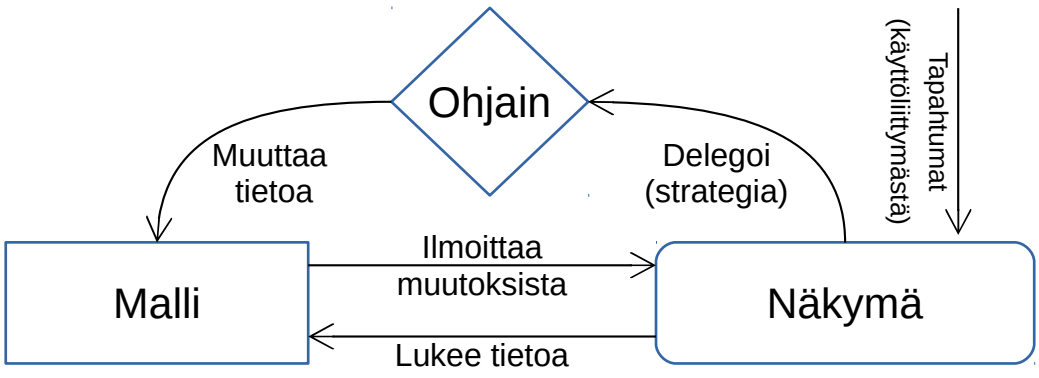
Datakerros toimii arkkitehtuurissa pysyväistiedon tallennuspaikkana. Kerroksen vastuulla on myös tiedonvarmennus.

Web-sovelluksissa looginen jako ei yleensä poikkea kovin paljon fyysisestä jaosta. Esityskerros on kuitenkin usein jaettu erillisiin selain- ja palvelinosuuksiin. Oikeastaan web-sovellusten kerrosarkkitehtuuri on samalla esimerkki *asiakas-palvelin-arkkitehtuurimallista*. Tällöin palvelimen roolina on yleensä tarjota eristettyjä palveluja asiakkaalle, tässä tapauksessa selaimessa suoritettavalle ohjelmalle [2, Luku 6.2.1].

2.3.2 Mallit, näkymät ja ohjaimet

Interaktiiviset sovellukset pohjautuvat usein standardiksi muodostuneeseen MVC (Model-View-Controller) -arkkitehtuurimalliin tai sen johdannaisiin. Seuraavassa esitetään malli sen kanonisessa muodossa, jossa se lähestulkoon sellaisenaan toteutettiin jo 1980-luvulla julkaistun Smalltalk-kielen[4] yhteydessä syntyneiden kehitystiimien työn tuloksena [5].

MVC perustuu käyttöliittymän ja sovelluslogiikan toteuttavien komponenttien selkeään eriyttämiseen ohjelman organisaation parantamiseksi (kuva 2). [5]



Kuva 2. MVC-arkkitehtuurimalli.

4 Smalltalk-80 on oliokieli, jolla on ollut huomattava merkitys ohjelmistokehityksen paradigmojen kehityksessä.

Malli edustaa sovelluksen käsittelemää tietoa ja tarjoaa rajapinnan, jolla tietoa voi lukea ja muokata sovellusaluesääntöjen rajoitteiden puitteissa. MVC ei ota kantaa mallin sisäiseen toteutukseen eikä siihen, miten tieto mahdollisesti esitetään käyttäjille. Mallin tilasta kiinnostuneet komponentit rekisteröityvät kuuntelemaan mallin muutoksia Tarkkailija-suunnittelumallin[5] mukaisesti. Mallin kuuntelijoita MVC:ssä ovat näkymät ja ohjaimet.

Näkymä vastaa mallin sisältämän tiedon esittämisestä sovelluksen kannalta sopivassa muodossa. Keskeinen MVC:n olettamus on, että näkymän tila vastaa aina[6] edustettavan mallin todellista tilaa. Jos siis mallin tila muuttuu, esimerkiksi asynkronisen HTTP-pyynnön seurauksena, on näkymän tilan myös muututtava. MVC:n yksi etu on, että samaa mallin tietoa voi yhtä aikaa edustaa useampi näkymä.

Ohjain toimii mallin ja näkymän välisenä liitoksena. Sen tehtävä on reagoida näkymistä tuleviin syötetapahtumiin ja tarvittaessa päivittää mallien tietoja. Ohjaimen vastuulla voi pienimmillään olla yksittäinen näkymä-malli-pari, mutta yhtä hyvin vastuulla voi olla myös paljon laajempi kokonaisuus. Ohjain luo tarvittaessa uusia näkymiä mukautuessaan vastaamaan sovelluksen kokonaistilaa. Ohjain tekee päätökset tilamuutoksista sisäisen tilansa, mallien tiedon ja syötetapahtumien perusteella. Ohjain rekisteröityy näkymien tapaan kuuntelemaan mallien tilamuutoksia.

MVC-mallin etuja ovat seuraavat [4, s. 194], [5]:

– Sama tieto on mahdollista esittää monella tavalla eri näkymissä.

– Ohjelman ylläpito helpottuu, koska vastuiden eriyttäminen selkeyttää arkkitehtuuria.

– Yksikkötestaus helpottuu, koska malleista ei ole suoria riippuvuuksia näkymiin.

– Näkymien ja sovelluslogiikan toteutusvastuun voi siirtää eri kehitystiimeille.

5 Perustuu komponentin liipaisemiin tapahtumiin, joita muut komponentit voivat rekisteröityä kuuntelemaan. Tapahtumia liipaisevan komponentin ei tarvitse tuntea yksityiskohtia kuuntelijoista. [1]

6 Näkymän tila voi hetkellisesti poiketa mallin tilasta. Arvon muokkaamisen salliva näkymä saattaa asettaa muutetun arvon malliin vasta, kun arvo täyttää tietyt muotovaatimukset.

2.3.3 SPA-arkkitehtuuri ja Ajax

Perinteisissä web-sovelluksissa HTML-sivut ladataan palvelimelta staattisina tiedostoina tai palvelin luo sivut dynaamisesti. JavaScriptiä käytetään vähän tai ei lainkaan. Sivun interaktiivisuus syntyy lähinnä lataamalla uusia staattisia web-sivuja tai lähettämällä syötetietoja HTML-lomakkeilla. [6]

SPA-arkkitehtuurimallissa (engl. Single-Page Application) web-sivun DOM-rakennetta muutetaan dynaamisesti ohjelman taustalla, lataamalla palvelimelta jatkuvasti uutta dataa. Nimensä mukaisesti malli poikkeaa perinteisestä web-sovelluksesta erityisesti siinä, että palvelimelta ladataan sovelluksen elinkaaren aikana vain yksi varsinainen HTML-dokumentti. Perinteisiin web-sovelluksiin verrattuna asiakaspäässä on huomattavan paljon enemmän sovelluslogiikkaa.

Arkkitehtuurimallin tietosisällön dynaamiset päivitykset tehdään Ajax-tekniikalla (Asynchronous JavaScript and XML). Siinä sovellus suorittaa palvelimelle asynkronisen pyynnön XMLHttpRequest-ilmentymällä, jota useimmat selaimet tukevat. Pyyntö tehdään sovelluksen taustalla, ilman uuden sivun lataamista. Vastauksen saatuaan sovellus muuttaa DOM-rakenteen vastaamaan uutta tilatietoa. Ajaxilla voi siirtää minkä tahansa formaatin mukaista dataa, mutta yleensä web-sovellukset käyttävät joko XML- tai JSON-enkoodausta. [7]

2.3.4 Refleksiivisyys ja metapiirteet

Sovelluskehyksellä luodaan uusi ohjelma kehystä laajentamalla, mihin oliokielten perintä ja monimuotoisuus tarjoavat tehokkaat staattiset työkalut. Ennakkoon tunnistetut rakenteet voi aina toteuttaa staattisesti, mutta kaikkia rakenteita ei ole mahdollista tunnistaa etukäteen. Olisikin eduksi, jos kehyksen olemassa olevia rakenteita voisi muokata, ja uusia luoda, dynaamisesti suoritusaikana. Seuraavassa tarkastellaan *refleksiivisyyttä* ja sen soveltuvuutta sovelluskehysten suunnitteluun.

Refleksiivisyydellä tarkoitetaan yleisesti järjestelmän tai ohjelman kykyä tarkastella ja muokata sisäistä rakennettaan suoritusaikana. Ohjelma erotetaan tällöin käsitteellisesti *perustasoon* ja vähintään yhteen *metatasoon*. Perustasoon kuuluu se ohjelman osa, joka suorittaa ohjelman tehtävää tai laskentaa. Metataso käsittää ohjelman ohjelmointikielestä riippuvan sisäisen rakenteen tai *metatiedon*. Perustason suoritus

mukautuu metatason tilaan. Perustasolla ja metatasolla on näin kausaalinen yhteys, missä muutos jälkimmäisessä kuvautuu perustasoon. Metatietoa luetaan ja muokataan ohjelmointikielen tarjoaman *metaprotokollan* avulla. Protokolla määrittelee, miten metatietoa voidaan käsitellä. Oliokielissä *metaolioprotokolla* on usein toteutettu perustason luokkien ilmentymiä vastaavilla metatason luokkien ilmentymillä eli *metaolioilla*. Monet ohjelmointikielet tukevat refleksiivisyyttä luonnostaan. Tällöin refleksiivisyyttä tukevia rakenteita kutsutaan *metapiirteiksi*. [4, Luku 16], [8]

Introspektiolla tarkoitetaan refleksiivisyyden sitä osuutta, jossa metatietoa pelkästään luetaan ja tarkkaillaan. Oliokielissä introspektio voi tarkoittaa esimerkiksi olion yliluokan nimen, attribuuttien lukumäärän, tai tietyn metodin parametrien tyyppien selvittämistä. [8, s. 3]

Refleksiivisyyden keskeinen etu on, että sillä voi vaikuttaa ohjelman toimintaan suoritusaikana, koskematta perustasoon. Sovelluskehyksissä tämä on erityisen hyödyllinen ominaisuus, koska sillä saadaan lisättyä kehyksen muokattavuutta. Refleksiivisyyden hyödyntäminen arkkitehtuurissa vaikuttaisikin olevan erittäin kannattavaa. Tilanne muuttuu entistä otollisemmaksi, kun ymmärretään, ettei refleksiivisyyden käyttö ole sidottu vain metapiirteisiin. Vaikka metapiirteitä ei siis olisi saatavilla, voi refleksiivisiä ominaisuuksia toteuttaa arkkitehtuuriin muilla keinoin. On kuitenkin syytä huomata, että metapiirteiden käyttö ei ole aina arkkitehtuurin kannalta suunnitteluratkaisu, vaikka metapiirteitä voikin käyttää suunnittelussa hyödyksi. Ohjelmointikielen valinta sen metapiirteiden takia on kuitenkin aina suunnittelupäätös.

Metapiirteiden yleisyys

Metapiirteiden käyttöpotentiaali riippuu ohjelmointikielestä. Dynaamisesti tyypitetyissä kielissä metapiirteitä on yleensä enemmän, koska ne viivyttävät tyyppitiedon käsittelyn osittain tai kokonaan suoritusaikaan. Skriptikielissä metapiirteitä onkin yleensä runsaasti. [8]

Oliokielissä luokkamalli tarjoaa tehokkaan tavan toteuttaa metapiirteitä kieleen. Tällöin metatasoa muokataan ja luetaan perustason kaltaisten luokkien metaolioiden avulla, mutta oliot sisältävät laskentaan liittyvän tiedon sijasta metatietoa. Viittaukset metaolioihin saadaan kielen tarjoaman metaolioprotokollan avulla.

Kaikki oliokielet eivät tue refleksiivisyyttä. Monissa vanhemmissa jäykän tyyppirakenteen omaavissa kielissä metapiirteitä on vähän, jos lainkaan. Esimerkiksi

C++ tukee refleksiivisyyttä heikosti, vaikka metaohjelmointi[7] onkin yksi kielen vahvuuksista [8, s. 5]. Uudemmassa Java-kielessä on melko kattavasti metapiirteitä ainakin introspektion muodossa. Vielä uudemmassa C#-kielessä on jo hyvät mahdollisuudet refleksiiviseen ohjelmointiin. Ruby on yksi esimerkki dynaamisesti tyypitetystä oliokielestä, jossa metapiirteitä on erittäin runsaasti.

JavaScriptin metapiirteet

Prototyyppikielenä JavaScript vaatii metapiirteiden suhteen erityismaininnan. Vaikka kielessä onkin metapiirteitä, lähinnä introspektiota, kieli ei kokonaisuudessaan ole erityisen refleksiivinen. Tämä on jossain määrin yllättävää, koska JavaScript on kuitenkin dynaamisesti tyypitetty skriptikieli. Osaltaan asiaa selittää puuttuva luokkamallin tuki, vaikka se onkin myöhemmissä versioissa lisätty kieleen.

Dynaamisesta luonteestaan johtuen kieli tarjoaa kuitenkin työkaluja toteuttaa sovellukseen refleksiivisiä ominaisuuksia ja jopa luokkamallin. Näin ollen metamallista on mahdollista tehdä niin kattava ja laaja kuin on tarpeen. Tällaisia ad hoc -tyylisiä ratkaisuja on kielen puitteissa tehty useita. Tässäkin työssä merkittävässä roolissa oleva Backbone.js-kehys sisältää yhden toteutuksen tällaisesta luokkamallista.

2.4 Luminato

Luminato on IP-päätelaite, joka prosessoi digitaalisia videosignaaleja. Laite vastaanottaa reaaliajassa eri lähteistä[8] DVB-standardin SD- ja HD-tarkkuuden TV-kanavavirtaa ja lähettää sitä eri signaalitekniikoita käyttäviin verkkoihin. TV-kanavia ja niiden alikomponentteja voi monipuolisesti suodattaa, muokata ja reitittää laitteeseen asennettujen moduulien lähtöihin. Tulosignaalin salauksen purku ja lähtösignaalin salaaminen onnistuu moduulikohtaisen CAM-liitännän ja sopivan älykortin avulla. Kokonaisuutta hallitaan web-käyttöliittymällä ja komentoriviohjelmalla.

Peruskäyttötapauksessa laite toimii CATV-verkkojen päätelaitteena. Kaapelioperaattori konfiguroi laitteen esimerkiksi niin, että laite vastaanottaa satelliitti- ja IP-lähteistä TV-kanavia, purkaa niiden mahdollisen salauksen ja ohjaa halutut kanavat

7 Metaohjelmointi on refleksiivisyyttä laajempi käsite, joka tarkoittaa ohjelmien kirjoittamista, jotka kuvaavat tai manipuloivat joko toisia ohjelmia tai itseään. Metaohjelmointi voi siis olla staattista. Refleksiivisyys on metaohjelmointia, jossa ohjelma manipuloi itseään suoritusaikana. [8, s. 5]

8 IP ja DVB-ASI/C/T/T2/S/S2 (kaapeli-, antenni- ja satelliittilähteet).

asiakasverkkoon. Operaattori voi lisätä kanavapaketin joukkoon myös metatietoa, esimerkiksi tekstitystä tai EPG-infoa.

Luminato soveltuu kuitenkin esimerkkiä huomattavan paljon mutkikkaampiin käyttötapauksiin. Viitteiden kautta kiinnostunut lukija voi tutustua tarkemmin laitteen tarjoamiin mahdollisuuksiin [9]. Laitetta kehitetään edelleen luomalla uusia ominaisuuksia sisältäviä moduuleja ja lisäämällä uusia ominaisuuksia vanhoihin moduuleihin.

2.4.1 Fyysinen rakenne

Luminato on rakenteeltaan modulaarinen. Laite koostuu kiinteästä rungosta, enintään kuudesta moduulista ja pakollisesta virtalähdemoduulista (kuva 3). Moduulit voi vapaasti valita käyttötarpeen mukaan, ja useimmat niistä voivat joko vastaanottaa tai lähettää moduloitua videosignaalia takapaneelin fyysisten porttien kautta. Moduulit kommunikoivat keskenään ja rungossa sijaitsevan hallintarajapinnan kanssa laitteen sisäisen IP-verkon avulla. Virtalähteessä sijaitsevat Ethernet-portit mahdollistavat kommunikoinnin myös ulkoisten IP-laitteiden kanssa.

Kuva 3. Luminaton runko, virtalähde ja moduulit. [9]

Laitteen rungossa ja jokaisessa moduulissa on yleissuoritin tai CPU, joka vastaa kunkin komponentin yleisestä toiminnallisuudesta. Rungossa on lisäksi kytkin, jonka kautta kaikki Ethernet-porttien, rungon ja moduulien IP-liikenne kulkee. CPU:n lisäksi moduuleissa on toinen, ohjelmoitava suoritin. Se prosessoi DVB-virtojen signaaleja ja reitittää tulokset kytkimelle tai moduulin lähtöihin. Prosessointiin ja reititykseen vaikuttaa olennaisesti moduulin asetukset, jotka moduulin CPU välittää ohjelmoitavalle

suorittimelle. Moduulin vastuulla on myös signaalien salauksen purku, jos moduulissa on CAM-liitäntä.

Toimintavarmuuden lisäämiseksi laite voidaan tarvittaessa kytkeä toiseen, isännän roolissa toimivaan Luminatoon. Kytketty laite ei tällöin normaalitilassa ole toiminnassa, vaan se kuuntelee isännän tilaa. Havaitessaan toimintahäiriön, kuuntelija aktivoituu korvaamaan isännän tai toimii sille vähintään varavirtalähteenä.

2.4.2 Sulautettu ohjelmistoarkkitehtuuri

Luminato on monien sulautettujen arkkitehtuurien tavoin toteutettu kerroksittain Linuxin päälle[9]. Alimpana sijaitsevan fyysisen kerroksen päällä ovat järjestelmä- ja ajurikerros. Niiden päällä on käyttöjärjestelmän palvelukerros, johon kuuluu tyypillisiä Linux-palveluita, kuten sshd, dhcpd tai ntpd. Seuraavina kerroksina ovat palvelu- ja hallintakerros, joiden taustaprosessien keskeisin tehtävä on hallinnoida laitteen asetuksia. Prosessien toteutuskielenä on järjestelmätasolla pääasiassa C++ ja alemmalla tasolla C-kieli.

Ylimmässä käyttöliittymäkerroksessa sijaitsee kaksi HTTP-palvelinta. Nginx[10] suorittaa pääasiallisesti käyttäjän tunnistukseen ja varmennukseen liittyviä toimenpiteitä ja ohjaa muut staattisen ja dynaamisen sisällön kyselyt joko Appweb-palvelimelle[11] tai palvelukerroksen prosesseille. Appweb on alun perin otettu käyttöön, koska sen avulla dynaamista sisältöä voi luoda JavaScriptillä. Molempien palvelinten keskeinen rooli on kuitenkin palvella Luminaton web-käyttöliittymää. Käyttöliittymäkerroksen toteutusta tarkastellaan luvussa 2.4.5.

Luminato tarjoaa HTTP:n lisäksi ulospäin palvelinrajapinnat DXI-protokollalle (luku 2.4.3), SSH-etäyhteyksille ja SNMP:lle[12].

9 UBM Tech:in suorittaman tutkimuksen mukaan Linux ja muut avoimen lähdekoodin sovellukset ovat jo usean vuoden ajan vallanneet tilaa suosituimpien (reaaliaika-)käyttöjärjestelmien joukosta [10].

10 Nginx on avoimeen lähdekoodiin perustuva kevyt ja monipuolinen web-palvelin [11].

11 Appweb2 oli ensimmäinen erityisesti sulautettuihin ympäristöihin suunnattu web-palvelin, jonka suunnittelussa tietoturvallisuus oli otettu huomioon alusta lähtien [12].

12 Simple Network Management Protocol on matalan tason protokolla, jonka avulla hallitaan IP-verkkolaitteita [13].

2.4.3 CATER-arkkitehtuuri ja DXI

Luminaton taustaprosessit ja niiden välinen kommunikointi rakentuu CATER-arkkitehtuurin päälle. CATER on yksityiseen käyttöön kehitetty C++ -pohjainen sulautettuihin järjestelmiin erikoistunut sovelluskehys. Se tarjoaa työkaluja taustaprosessien kehitystyöhön ja niiden hallinnointiin sovelluksen suoritusaikana. Lisäksi se tarjoaa Linuxissa toimivan standardin komponenttipohjaisen tuotteiden käännösjärjestelmän.

Taustaprosessit välittävät tietoa keskenään CATER-arkkitehtuuriin kuuluvan DXI-protokollan avulla. DXI tarjoaa kirjaston ja rajapinnan, jonka avulla prosessit voivat rekisteröidä tyypitettyjä avain-arvo pareja keskitetysti DXI:n hallintaprosessille. Toiset prosessit voivat lukea avainta vastaavan arvon tai asettaa sille uuden arvon käyttäen samaa rajapintaa. DXI-avain koostuu pisteillä erotetuista elementeistä, jotka viittaavat joko toiseen tyypitettyyn arvoon, avainhierarkian solmuun tai yhteen solmun lapsista. Esimerkiksi avain *device.0.module.2.output.3.ip.addr* voisi viitata laitteen 0 toisen moduulin kolmannen lähdön IP-osoitteeseen, jonka tyyppi voisi olla esimerkiksi kokonaisluku tai merkkijono.

Laitteen ulkopuolelta DXI:n rajapintaa kutsutaan XML-RPC:n avulla. Koska yksittäisten avainten kysely voi olla hidasta suurten tietomäärien kohdalla, Luminatoon on toteutettu optimoituja merkkijono-tyyppisiä erikoisavaimia, joiden arvot sisältävät koodattuna useiden avainten arvoja kerralla.

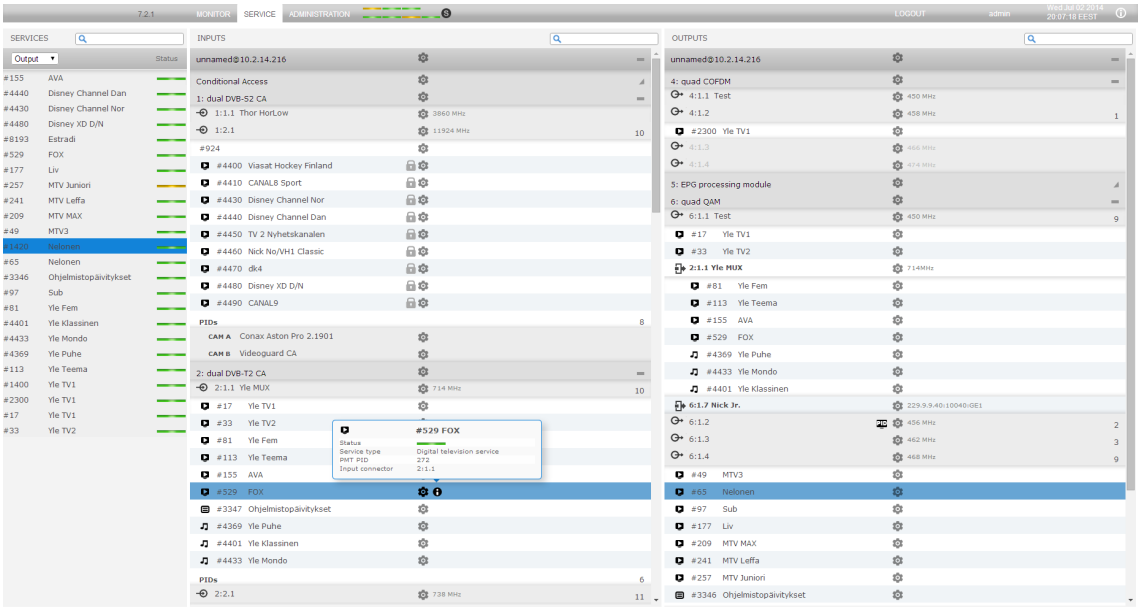
2.4.4 Web-käyttöliittymä

Luminatoa hallitaan monipuolisella web-käyttöliittymällä[13]. Laitteeseen yhteyden ottavalle käyttäjälle näytetään ensin sisäänkirjautumissivu, johon syötetään käyttäjätunnus ja salasana. Onnistuneen kirjautumisen jälkeen käyttäjälle avautuu päänäkymä, joka koostuu yläpaneelista ja kolmesta valittavasta työpöytänäkymästä. Haluttu työpöytänäkymä valitaan yläpaneelissa sijaitsevien välilehtien avulla. Sisäänkirjautuneen käyttäjän ryhmä määrää, minkä tyyppinen oikeus käyttäjälle on eri näkymiin.

13 Tuettuja selaimia ovat Google Chrome, Mozilla Firefox ja Microsoft Internet Explorer 8.

Monitorinäkymässä käyttäjä voi tarkastella laitteen ja moduulien tilaa yleiskuvan saamiseksi. Näkymässä ei kuitenkaan ole mahdollista muokata asetuksia.

Palvelunäkymä on käyttöliittymän ydin (kuva 4). Sen kautta hallitaan moduuleita ja niiden TV-kanavien ja palveluiden reitityksiä. Näkymä koostuu kolmesta sarakkeesta järjestyksessä vasemmalta oikealle: *palveluhakusarake* näyttää valinnan ja hakusanan perusteella listan saapuvista tai lähtevistä palveluista; *tulomoduulisarake* (kuvassa INPUTS) näyttää puuhierarkian tulomoduuleista, niiden porteista ja portteihin saapuvista palveluista; *lähtömoduulisarake* (kuvassa OUTPUTS) näyttää puuhierarkian lähtömoduuleista ja niiden porttien lähtevistä palveluista. Palveluiden reititys onnistuu yksinkertaisesti raahaamalla hiirellä halutut saapuvat palvelut lähtömoduuleiden portteihin. Jokaiselle moduulille, portille ja palvelulle saadaan avattua erillinen hallintadialogi, josta komponentin asetuksia pääsee tarkastelemaan ja muuttamaan.



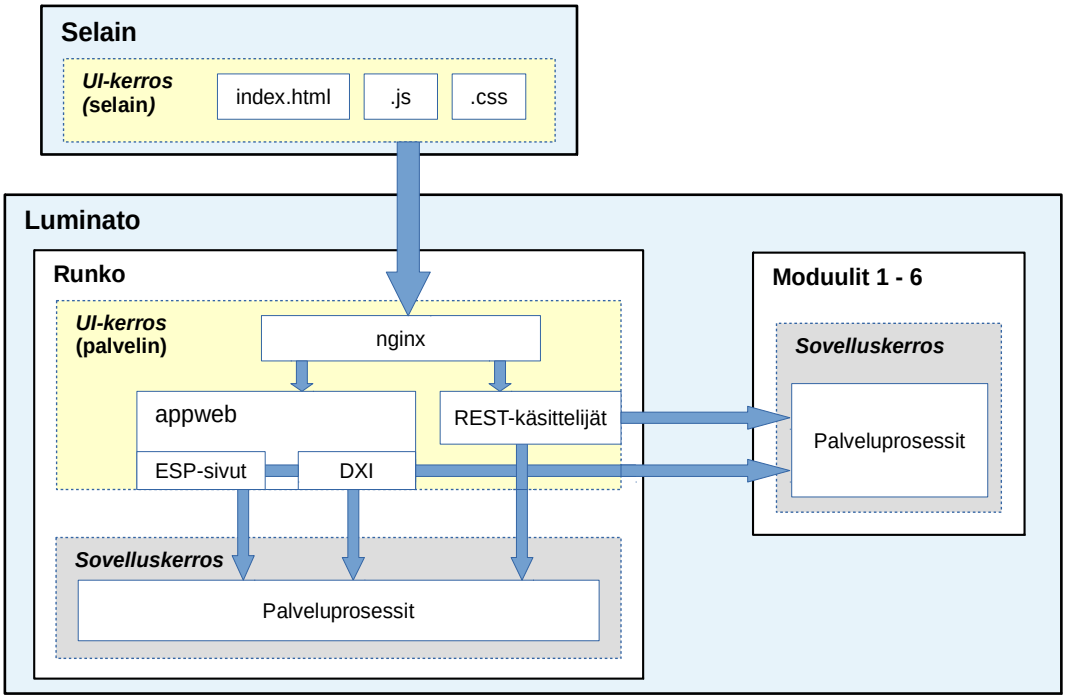
Kuva 4. Kuvakaappaus Luminaton käyttöliittymän palvelunäkymästä. Keskellä näkyvät tulomoduulit ja niiden portteihin saapuvat palvelut. Oikealla ovat lähtömoduulit ja niiden portteihin reititetyt palvelut. Lähes jokaisesta näkymän elementistä voi avata erillisen dialogin, jossa voi tarkastella elementtiä vastaavan fyysisen laitekomponentin tilaa ja mahdollisesti muokata sen asetuksia.

Administraationäkymässä käyttäjä voi muuttaa laitteen yleisiä asetuksia, joita Luminatossa on hyvin paljon. Näkymän kautta voi myös esimerkiksi lisätä uusia käyttäjiä, päivittää laitteen ohjelmiston, tarkastella laitteella olevia tiedostoja ja ladata aktiivisen laitekonfiguraation.

2.4.5 Käyttöliittymän ohjelmistoarkkitehtuuri

Luminaton käyttöliittymä on toteutettu SPA-mallilla. Laitteelta ladataan siis vain yksi HTML-dokumentti, joka määrittelee käyttöliittymän DOM-puun rungon ja sisältää viittaukset laitteelta ladattaviin JavaScript- ja CSS-tiedostoihin. Dokumentin lataamisen jälkeen laitteen kanssa kommunikoidaan pääasiassa Ajax-tekniikalla. Tietomallien siirtoon käytetään joko JSON-formaatin REST-rajapintaa tai DXI-protokollaa.

Käyttöliittymäkerros on osa laitteen 3-kerrosarkkitehtuuria (kuva 5). Luminaton laiteresurssien vähäisyyden takia käyttöliittymän logiikka on sijoitettu pääosin selaimessa suoritettavaksi. Näin ollen itse laitteessa on vain vähän esityskerrokseen kuuluvaa toteutusta. Itse asiassa osa sovelluslogiikasta on myös siirretty selaimen suoritettavaksi. Koska laitteelle tehdään pääasiassa tilakyselyitä ja lähetetään muuttuneita asetuksia, säilytetään selaimessa paljon tilatietoa. Kaikesta tästä johtuen selainta suorittavalta tietokoneelta vaaditaan hyvää suorituskykyä.



Kuva 5. Luminaton käyttöliittymän kerrosarkkitehtuuri.
Kuvaa on yksinkertaistettu ja esimerkiksi datakerrosta ei ole näkyvissä.

Datakerrokseen ei kuulu varsinaista ulkoista tietokantaa. Järjestelmän pysyväistiedot, kuten rungon ja moduulien asetukset, tallennetaan rungossa sijaitsevalle flash-piirille.

Ohjelmistokirjastot ja työkalut

Web-sovelluksien perustyökalujen lisäksi Luminatossa on käytetty joitakin avustavia ohjelmistoja (taulukko 1), joista keskeisimmät ovat jQuery [14], Backbone.js [15] ja Underscore.js [16].

Taulukko 1. Luminaton käyttöliittymän toteuttavia ohjelmistoja.

Ohjelmisto	Versio	Selite / käyttötarkoitus
JavaScript	ECMAScript 5	Sovelluksen interaktiivisuus ja logiikka.
HTML	4	DOM-elementit ja asemointi.
CSS	2	DOM-elementtien tyylit.
Less.js	2.5.x	Käännettävä CSS:n laajennuskieli.
jQuery	2.1.x	Yleiskäyttöinen kirjasto DOM-puun dynaamiseen muokkaamiseen. Sisältää ratkaisuja web-sovelluksissa usein esiintyviin ongelmiin.
jQuery-UI	1.11.x	Kokoelma yleishyödyllisiä käyttöliittymäelementtejä. Luminatossa erityisesti dialogit.
jsTree	3.2.x	Puunäkymien esittämiseen erikoistunut jQuery-liitännäinen.
Underscore.js	1.7.x	Yleishyödyllinen JavaScript-kirjasto.
Backbone.js	1.1.x	Tietomallit, kokoelmat ja CRUD (datan synkronointi). Näkymät ja tapahtumat.

Backbone.js

Backbone on minimaalinen JavaScript-kehys, jolla voi luoda rakenteellisesti yhtenäisiä web-sovelluksia luokkamallin päälle. Kehyksen filosofiana on tarjota perusluokat MVC:n malleja ja näkymiä vastaavista tietorakenteista, joita web-sovelluksissa usein tarvitaan. Perusluokkien lisäksi kehys tarjoaa kokoelmaluokan mallien joukkokäsittelyyn. Kokonaisuutta sitoo yhteen Tapahtuma-suunnittelumallin mukainen Events-rajapinta, jonka kaikki kehyksen luokat toteuttavat. Mallit synkronoidaan oletuksena Ajaxilla palvelimen toteuttamasta REST-rajapinnasta. [15]

Mallia käytetään erikoistamalla siitä uusi luokka. Erikoistuksen yhteydessä annettava JavaScript-objekti määrittää luokan alustusfunktion, metodit ja muut ilmentymän prototyypin oletusarvot. Backbone-mallin ilmentymällä on aina myös attributes-niminen objekti-ominaisuus, joka sisältää synkronoitavat avain-arvo-parit tai *Backbone-attribuutit*. Näitä attribuutteja ei tule kuitenkaan sekoittaa oliokielten yleiseen attribuutti-käsitteeseen, vaikka niillä nimen lisäksi yhteistä onkin. Backbone-attribuutit lisätään aina dynaamisesti, eikä niitä määritellä staattisten oliokielten tyyliin luokan määrittelyn yhteydessä. Käsitteiden välinen ero tosin hämärtyy, koska JavaScriptissä

objektien arvoja voi lisätä, poistaa ja muokata rajoituksetta. Jatkossa tässä tekstissä käytetään sekaannuksen välttämiseksi oliokielten attribuutista nimeä *ilmentymä-muuttuja* tai pelkästään muuttuja.

Backbone-mallin kantaluokka tarjoaa rajapinnan, jonka avulla ilmentymän attribuuttien arvoja voi lukea ja muuttaa. Toinen mallin kannalta tärkeä rajapinta liittyy attribuuttien ja siten koko mallin synkronointiin palvelimen kanssa. Synkronointirajapinnan keskeisimmät metodit liittyvät mallin *lukuun*, *tallennukseen* ja *tuhoamiseen*. Jokainen rajapinnan metodi suorittaa Ajax-kyselyn palvelimen REST-rajapintaan ja tarvittaessa päivittää attribuutit vastaamaan vastauksessa tullutta uutta dataa. Malleihin liittyy aina yksilöllinen tunniste, jonka perusteella palvelin voi tunnistaa metodin kohteen.

Kuten edellä mainittiin, myös mallin kantaluokka toteuttaa Events-rajapinnan. Malli tukee oletuksena useita tapahtumatyyppejä, joista yksi tärkeimmistä on ’change’-tapahtuma, joka liipaistuu yleensä mallin attribuuttien muuttuessa.

Näkymä sisältää aina viittauksen DOM-elementtiin, mutta usein myös malliin tai niiden kokoelmaan. Backbone ei kuitenkaan ota kantaa siihen, miten näkymää vastaava sisältö pitäisi esittää. Sovelluskehittäjä voi siis varsin vapaasti käyttää mitä tahansa tekniikkaa näkymän tietojen esittämiseen. Kuuntelijoiden avulla näkymän tila voidaan aina pitää ajan tasalla MVC:n mukaisesti. Näkymä voi esimerkiksi rekisteröityä kuuntelemaan edellä mainittua mallin ’change’-tapahtumaa ja päivittää DOM-rakenteen vastaamaan mallin uutta tilaa.

2.4.6 Käyttöliittymän arkkitehtuurin komponentit

Luminaton käyttöliittymän arkkitehtuurista voi tunnistaa useita loogisesti erillisiä komponentteja tai niiden kategorioita. Keskeisimmät niistä on listattu taulukossa 2.

Taulukko 2. Luminaton käyttöliittymän arkkitehtuurin komponentteja. Taulukossa on listattu esimerkkejä komponentteihin kuuluvista abstrakteista ja konkreeteista luokista, sekä komponenttien tärkeimmät vastuualueet.

Komponentti	Luokkia	Käyttötarkoitus tai vastuualueet
Sovellusydin	Application	Ohjelman alustus, päänäkymän hallinnointi ja paikallinen URL-reititys.
Istunnonhallinta	SessionManager	Sisään- ja uloskirjautuminen, käyttäjätunnistus, istunnon vanheneminen, laiteyhteyden varmennus.
Meta	ModelMeta, AttrMeta, EnumMeta	Mallien, attribuuttien ja enumeraatioiden metamääritysten muuttaminen luokkia vastaaviksi metaolioiksi.
Malli	BaseModel, DxiBaseModel, JsonBaseModel, Collection	Abstraktit ja konkreetit Backbone-pohjaiset mallien ja kokoelmien kantaluokat. Mallit ovat tiukasti sidottu Meta-komponenttiin.
Näkymä	BaseView, BaseAttributeView, LoginView, BaseDialog	Abstraktit ja konkreetit kokooma-, dialogi- ja attribuuttinäkymät käyttöliittymän asemointiin ja mallien datan esittämiseen.
Ohjain	BaseController	MVC-ohjaimet näkymien elinkaarten hallintaan.
Logopaneeli	Logopanel	Yläpaneelin yrityslogo, välilehdet, laitteen statustieto, uloskirjautuminen, päivämäärä yms.
Validointi	Validation	Validoi attribuuttinäkymien syötteet ennen kuin niitä vastaavat arvot asetetaan malleihin.
Toiminnot	BaseAction, BaseCommand	Luokkia, joilla voi luoda asynkronisia ja tapahtumapohjaisia taustaprosesseja.
Tietovarasto	LocalStorage SessionStorage	Abstraktoi selaimen LocalStorage- ja SessionStorage-ominaisuudet.
Engine	EngineApplication, EngineAPI, ModelAPI, ViewAPI	Lisää käyttöliittymään laajennuskohtia, joihin kolmannet osapuolet voivat luoda käyttöliittymäliitännäisiä. Tarjoaa rajoitetun rajapinnan käyttöliittymään, tietomalleihin ja näkymiin.
Synkronointi	Query, DxiUpdater, XmlRpc	Dxi-mallien attribuuttikohtainen CRUD DXI:n ja XmlRpc:n avulla. Komponentti sisältää myös Json-mallien synkronoinnin toteutuksen.
JSON	JsonBaseModel	Osa Json-mallien toteutuksesta. Json-mallien kantaluokka on määritelty Malli-komponentissa.
Jasmine	Testutils	Toiminnallisuus- ja yksikkötestaus.

Moduulit ja nimiavaruudet

Luminaton käyttöliittymän toteutus koostuu tiedostomoduuleista. Loogisesti tarkastellen toteutus koostuu myös puurakenteisesta joukosta nimiavaruuksia, ja jokainen moduuli vastaa pääsääntöisesti yhtä nimiavaruutta. Nimiavaruuden voi siis ymmärtää olevan yksilöllinen tunniste moduulille. Luminaton nimiavaruuspuun juuritunniste on **T**, ja sen alla sijaitsevat heti kaikki muut moduulit loogisesti nimettyinä.

Jokaisen taulukon 2 komponentin toteutus sijaitsee pääsääntöisesti yhdessä moduulissa. Joissakin suurissa komponenteissa, jotka koostuvat useammasta loogisesta alikokonaisuudesta, toteutus on kuitenkin jaettu useampaan moduuliin. Joissain tapauksissa jaon peruste on käytännöllinen, esimerkiksi kehitystyökalujen suorituskykyä kuormittava rivimäärä tiedostoissa. Toisaalta on myös komponentteja, jotka sisältyvät kokonaan yhteen moduuliin, mutta sama moduuli sisältää myös jonkin toisen komponentin toteutusta.

Moduulien ja nimiavaruuksien alustus tehdään latausjärjestyksessä anonyymin funktion sisällä[14]. Alustuksen yhteydessä luodaan aina ensin nimiavaruutta vastaava JavaScript-objekti, jonka ominaisuuksiksi määritellään staattiset funktiot, moduulin luokat ja ilmentymämuuttujat. Osa moduuleista aloittaa alustuksen yhteydessä asynkronisia operaatioita, joiden tulokset saadaan vasta myöhemmin.

Sovellusydin ja istunnonhallinta

Nimiavaruuksien alustuksen jälkeen kontrolli siirtyy sovellusytimelle, joka luo ja alustaa sovelluksen keskeiset komponentit ja käyttöliittymän perusrakenteet. Ytimen voi ymmärtää sovellusta hallinnoivaksi ohjaimeksi, jolla on aina viimeinen sana koko sovellusta koskevissa tapahtumissa.

Istunnonhallinta on yksi tärkeimmistä sovellusytimen hallinnoimista komponenteista. Se suorittaa laitteelle alustuksensa yhteydessä ensimmäisen datakyselyn, ja jää sitten odottamaan käyttäjän sisäänkirjautumista. Jos käyttäjällä on jo voimassaoleva istunto, sovellus siirtyy suoraan päänäkymään ja jatkaa aiempaa istuntoa. Datakyselyn lisäksi istunnonhallinta tekee kyselyn näkymätemplaateista[15], jos ne eivät ole jo valmiiksi

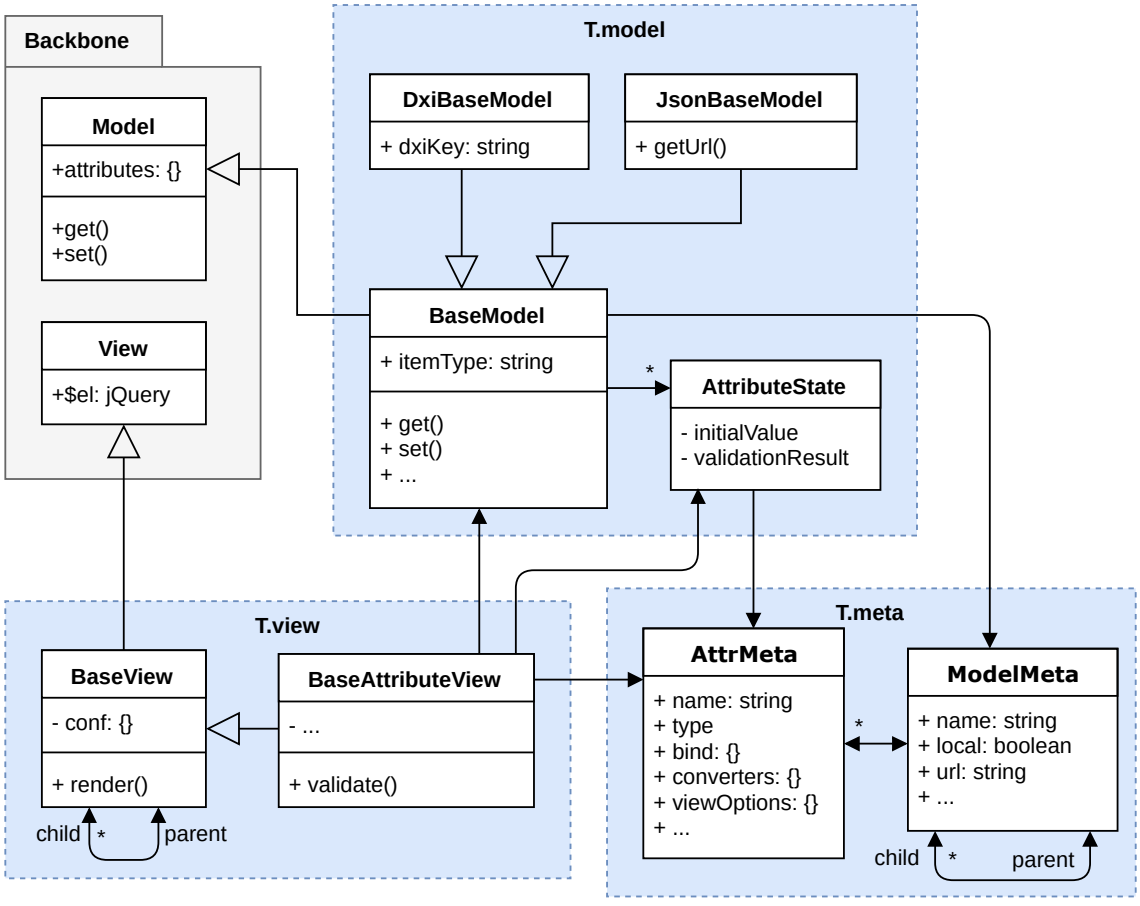
14 JavaScript-idiooma, jossa muuttujat määritellään moduulissa välittömästi suoritetun anonyymin funktion paikallisiksi muuttujiksi. Normaalisti globaalit muuttujat voidaan näin piilottaa moduulin sisälle.

15 Luminaton templaateissa käytetään Underscore.js -kirjaston tukemaa formaattia, jossa tekstin korvattavat osat kirjoitetaan aloitustunnisteen ’<%=’ ja lopetustunnisteen ’%>’ väliin.

selaimen paikallisessa tietovarastossa tai jos varastossa olevien templaattien versio poikkeaa laitteen ohjelmistoversiosta. Edellisten kyselyiden lisäksi istunnonhallinta tekee ohjelman normaalin suorituksen aikana myös säännöllisiä kyselyitä, joilla varmistutaan laiteyhteyden toimivuudesta ja käyttäjän istunnon autenttisuudesta.

Tietomallit ja metadata

Luminaton tietomallit periytyvät T.model.DxiBaseModel- ja T.model.JsonBaseModel-luokista, jotka taas pohjimmiltaan periytyvät Backbonen mallista (kuva 6). Luminaton kaikki mallit rakentuvat siis Backbonen attribuuttien ympärille. Luokkien toteutukset poikkeavat toisistaan olennaisesti niiden synkronointitekniikan osalta: DxiBaseModel päivittää kaikki tai osan attribuuteistaan DXI-protokollan ja attribuutteihin sidotun DXI-avaimen avulla. JsonBaseModel päivittää kaikki attribuuttinsa yhdellä kertaa, suorittamalla Ajax-pyynnön laiteen REST-rajapintaan.



Kuva 6. Malli-, näkymä- ja metaluokkien assosiaatiot.
Metamallit vaikuttavat mallien ja näkymien käyttäytymiseen. Kuvassa ei ole esitetty enumeraatio-metaluokkaa, joka on kolmas käytetty metatyyppi.

Luminaton malliluokkiin ja niiden attribuutteihin sitoutuu luokkamäärittelyn lisäksi metatietoa eksplisiittisesti annettavien *metamääritysten* muodossa. Määrittelyillä on aina *metatyyppi*, joista malli- ja attribuutti-tyyppi ovat tärkeimmät[16]. Metamäärittely rekisteröidään sovelluksen alustuksen yhteydessä moduulin T.meta rajapinnan kautta *metakonfiguraatiolla*, joka on käytännössä JavaScript-objekti. Jokaista rekisteröityä määrittelyä vasten luodaan metaolio, johon jokainen myöhemmin luotu mallin tai attribuutin ilmentymä saa viittauksen. Metaolio vaikuttaa siis jokaisen sitä vastaavan ilmentymän käyttäytymiseen.

Sovelluskehittäjän näkökulmasta edellisestä seuraa, että yksittäisen luokan määrittelyä löytyy kahdesta eri tiedostosta. Vaikka tällä on omat haittapuolensa, erottelu on kuitenkin looginen. Moduuli T.model sisältää luokan toteutuksen, eli metodit ja muuttujat, samaan tapaan kuin muissakin oliokielissä. Moduuli T.meta sisältää metamääritykset. Tilannetta voi tarkastella vertaamalla rakennetta Java-kieleen. Tällöin Luminaton mallien metaoliot ovat suoraan rinnastettavissa Javan Class-luokan ilmentymiin. Siinä missä Javan metapiirteet luovat ilmentymien määritelmän implisiittisesti, Luminatossa määritelmät annetaan eksplisiittisesti. Molemmissa tapauksissa on kyse refleksiivisestä mekanismista.

Luminaton mallien ja attribuuttien metamääritykset tukevat suurta joukkoa ominaisuuksia. Osa ominaisuuksista on yhteisiä kaikille metatyypeille[17], mutta suurin osa riippuu metatyypistä. Ominaisuuksien vaikutusalue ei kuitenkaan rajoitu MVC:n malliin. Erityisesti attribuutteihin liittyy ominaisuuksia, jotka vaikuttavat suoraan attribuuttia vastaavan näkymän toimintaan. Itse asiassa juuri attribuutin metamäärityksen perusteella metaoliolle asetetaan näkymäluokka, jolla attribuutti esitetään käyttäjälle.

Näkymät

Kaikki Luminaton näkymät periytyvät T.view.BaseView-kantaluokasta, joka taas periytyy Backbonen näkymästä (kuva 6). Näkymien koko perintähierarkia on varsin mittava, mutta sen ymmärtäminen helpottuu heti, kun huomataan perustyyppejä olevan vain muutama. Perustyyppien lisäksi Luminatossa on myös joitakin erikoistuneempia

16 Mallien ja attribuuttien lisäksi Luminatossa on mahdollista rekisteröidä myös *enumeraatio*-tyyppejä, joiden ilmentymät mallintavat staattisesti tai dynaamisesti koostettua joukkoa arvo-teksti-pareja. Enumeraatioita käytetään erityisesti attribuuttinäkymien pudotusvalikoissa.

17 Esimerkiksi määritystä vastaavan luokan nimi, DXI-avain ja ilmentymän rakentajafunktio.

näkymiä, joita on vaikeampi sijoittaa mihinkään tiettyyn kategoriaan. Palvelunäkymän toteuttava puunäkymä on yksi esimerkki.

Kokoomanäkymät koostuvat tavalla tai toisella toisista näkymistä. Ne eivät itse välttämättä vastaa DOM-puussa muusta kuin juurielementistään. Niiden keskeinen rooli on helpottaa näkymähierarkian hallintaa. Konkreettisen näkymätyypin ohella kokoomanäkymän voi ymmärtää myös näkymiin liittyväksi ominaisuudeksi.

Attribuuttinäkymät vastaavat mallien attribuuttien arvojen esittämisestä ja tarjoavat mahdollisesti mekanismin arvojen muuttamiseen. Attribuuttinäkymät hallinnoivat suhteellisen pientä osaa DOM-puusta. Näkymien päivitys perustuu joko HTML:ää generoiviin templaatteihin tai DOM-rakenteen ohjelmalliseen muokkaamiseen jQueryn avulla. Yleensä käytetään kuitenkin näiden tekniikoiden yhdistelmää. Tallennettavien attribuuttien tapauksessa näkymä tarjoaa käyttäjälle yleensä jonkinlaisen tavan muuttaa arvoa. Lähes poikkeuksetta näkymä tällöin myös validoi syötteen Validointi-komponentin avulla, ennen sen asettamista hyväksyttynä mallille.

Juurinäkymä ei varsinaisesti ole konkreettinen näkymätyyppi, vaan enemmänkin tiettyihin näkymiin liittyvä ominaisuus. Keskeinen juurinäkymän tehtävä on ylläpitää *transaktiota*, johon keskitetysti kootaan kaikki juurinäkymän näkymähierarkiassa tehdyt muutokset tallennuskelpoisiin malleihin. Jos transaktio sisältää muutoksia, ja näkymissä ei ole virheitä, näkymä yleensä välittää tämän informaation käyttäjälle ja sallii tietojen tallennuksen jollain näkymälle luonnollisella tavalla. Transaktion tallennus tarkoittaa käytännössä muuttuneiden tietojen synkronointia laitteen kanssa.

Dialoginäkymät ovat siirrettäviä, hetken aikaa näytettäviä modaalisia ikkunoita päänäkymässä. Vaikka dialogit ovat toisaalta aina juurinäkymiä ja kokoomanäkymiä, ne sisältävät myös niin paljon muuta toiminnallisuutta, että ne on syytä erottaa omaksi tyypiksi. Juurinäkymän ominaisuudessa dialogi sallii yleensä käyttäjän painaa dialogin ”Save”-painiketta, joka käynnistää transaktion tallennuksen. Sen lisäksi, että transaktion tallennus on estetty, jos attribuuttinäkymissä on virheitä, näyttää dialogi käyttäjälle myös virheiden lukumäärän.

Ohjaimet

Ohjainten rooli on kanoniseen MVC-malliin varsin suppea. Ohjaimet lähinnä hallinnoivat näkymiensä elinkaarta ja suorittavat ensimmäisen datakyselyn, jossa haetaan näkymiin liittyvät tietomallit. Ohjainluokat rekisteröidään sovellusytimen tarjoaman rajapinnan kautta yksilöllisen tunnisteen alle.

Ohjaimet jakautuvat kahteen pääluokkaan. **Työpöytä**-ohjaimien ilmentymiä voi olla olemassa kerrallaan vain yksi tunnistetta kohden. Niitä käytetään ohjaamaan käyttöliittymän kiinteitä näkymiä, kuten Logopaneelia (kuvassa 4 ylimpänä) tai sen kautta avattavia työpöytänäkymiä.

Toiseen pääluokkaan kuuluvat **dialogi**-ohjaimet, joiden ilmentymien määrä ei ole rajattu. Nimensä mukaisesti ohjaimet hallinnoivat dialoginäkymiä.

Engine

Luminaton käyttöliittymän perustoteutuksen voi ajatella olevan itsenäinen isäntäsovellus. Tällöin Engine-komponentti tarjoaa kolmansille osapuolille rajapinnan, jota vasten isäntäsovellukseen voi luoda Engine-sovellusliitännäisen. Tällainen sovellus sisältää isännästä kokonaan erilliset tietomallit, metamääritykset, toteutuslogiikan ja XML-pohjaisen käyttöliittymämäärityksen. Engine-sovellus voi kuitenkin tarvittaessa käyttää isäntäsovelluksen malleja. Tämän työn puitteissa Enginen yksityiskohtiin ei syvennytä tarkemmin. Todetaan kuitenkin, että aiheesta riittäisi kerrottavaa tutkielmaan jos toiseenkin.

2.4.7 Käännösympäristö ja muutosten testaaminen

Luminaton CATER-käännösjärjestelmä perustuu Linux-skripteihin ja Autotoolsiin. Tuotejulkaisun yhteydessä käännösprosessi noutaa tuotekonfiguraation mukaiset versiot tuotekomponenteista, kääntää ja asentaa ne kohdehakemistoon, ja generoi lopuksi tuotteesta julkaistavan levykuvan.

Käyttöliittymäkomponentti ei kääntämisen suhteen eroa muista tuotteen osakomponenteista. Vaikka varsinaista käännöstä binäärikoodiksi ei tehdäkään, komponentin asennuksen yhteydessä suoritetaan tiedostojen kopioinnin lisäksi myös tiedostojen yhdistämistä ja muita raskaampia lähdekoodien muokkausoperaatioita.

Käyttöliittymäkomponentissa on komento, jolla tuotekehityksen aikana käyttöliittymään tehtyjä muutoksia voi testata oikeassa laiteympäristössä. Tällöin käyttöliittymä käännetään ja asennetaan ensin johonkin työkoneen hakemistoon, josta se pakkaamisen jälkeen siirretään SSH:n avulla laitteelle, ja puretaan siellä kohdehakemistoihin. Laitteen HTTP-palvelimen uudelleenkäynnistyksen jälkeen käyttöliittymää voi välittömästi testata. Optimointina laitteeseen siirretään vain oikeasti muuttuneet tiedostot, koska laiteresurssien rajoitteiden takia siirto kestää suhteellisen kauan.

2.5 Muut kehitystyökalut

2.5.1 Node.js ja npm-ekosysteemi

Web-palvelinten toteutukseen käytetään monia erilaisia tekniikoita. Erikoistuneet natiivisovellukset lukeutuvat suorituskykyisimpiin vaihtoehtoihin, mutta niiden ylläpito voi olla työlästä ja toteuttaminen hidasta. Suositumpi vaihtoehto onkin käyttää yleiskäyttöistä web-palvelinta, joka ohjaa pyynnöt dynaamisille käsittelijöille. Käsittelijöiden toteutustekniikoiden joukko on mittava. Esimerkkeinä voi mainita palvelimen lapsiprosesseina suoritettavat ohjelmat, palvelimen omassa prosessissaan suoritettavat laajennukset ja kokonaiset web-sovellusarkkitehtuurin muodostavat laajat sovelluspalvelimet, kuten Java EE.

Node.js on käyttöjärjestelmäriippumaton[18] teknologia, jolla web-sovelluksen palvelinosuuden voi toteuttaa nopeasti. Tekniikaltaan Node on tapahtumapohjainen, asynkroninen ohjelmointiympäristö, jossa kielenä käytetään ECMA-262-standardin JavaScriptiä. Vaikka Noden ensisijaisia käyttökohteita ovatkin web-palvelimet, voi Nodea käyttää myös muihin tarkoituksiin. [17]

Node-ohjelma suoritetaan Googlen V8-moottorilla. Tällöin V8 kääntää lähdekoodin aina alustavaksi natiivikoodiksi. Jos V8:n profiloija havaitsee ajonaikana proseduuria suoritettavan usein, proseduuri käännetään optimoidummaksi versioksi. Moottori sisältää myös muita optimointitekniikoita, kuten objektien piilotetut luokat, joilla päästään tietyissä tilanteissa hyvinkin nopeaan natiivikoodiin. Googlen V8-kääntäjä on

18 Tuettuja käyttöjärjestelmiä ovat ainakin OS X, Microsoft Windows, Linux, Solaris, FreeBSD ja OpenBSD.

monesti tehokkaampi kuin pelkästään tulkkaukseen perustuvat JavaScript-moottorit. [18]

Noden asennuksen mukana tulee kokoelma perusmoduuleja, mutta niiden oheen on ehtinyt syntyä mittava avoimen lähdekoodin moduuliekosysteemi. Moduuleja asennetaan Noden mukana tulevan **npm-**paketinhallintaohjelman avulla. Moduulit asentuvat joko ohjelmistoprojektin alle node_modules-hakemistoon tai globaalisti järjestelmäriippuvaan hakemistoon. Sovelluksen avulla hallinnoidaan myös jo asennettuja moduuleja. Toimintansa tueksi npm lukee yleensä projektin juurihakemistossa sijaitsevasta package.json-tiedostosta projektiin liittyvää metadataa, kuten projektin version, nimen tai riippuvuudet muihin Node-moduuleihin. [19]

2.5.2 Grunt

Grunt on Node-ekosysteemin ohjelma, jolla voi automatisoida ohjelmistoprojektien kehitysaikaisia tehtäviä, joita ovat esimerkiksi sovelluksen kääntäminen, asennus tai käyttöönotto. Gruntin ympärille on muodostunut suuri liitännäisten ekosysteemi, johon kuuluvat moduulit helpottavat yleisimpien tehtävien luontia. [20]

Grunt perustuu projektihakemistossa sijaitsevaan Gruntfile.js-tiedostoon, johon annetaan deklaratiivisesti tehtävien tarvitsemat konfiguraatiot. Konfiguraatiot ovat tehtäväkohtaisia, mutta niihin kuuluu joitakin standardiosia, joista keskeisimmät ovat useimpien tehtävien tarvitsemat lähde- ja kohdetiedostomääritykset. Omat tehtävät rekisteröidään tiedostoa suoritettaessa antamalla niiden nimi ja toteutusfunktio Gruntin rajapintaa edustavan instanssin metodille. Tehtävä suoritetaan antamalla sen nimi ja optiot argumentteina Grunt-ohjelmalle, jolloin Grunt yhdistää tehtävän ja sen konfiguraation nimen perusteella. Tehtävän toteuttava funktio suoritetaan konfiguraation esikäsittelyn jälkeen.

2.5.3 Express.js

Express(.js) on minimalistinen Node-pohjainen web-sovelluskehys, jolla voi toteuttaa HTTP-palvelimen. Siihen on koostettu web-palvelimissa usein tarvittavia ominaisuuksia korkeamman tason abstraktioiksi, jolloin ohjelmoija voi keskittyä olennaiseen. [21]

Yleinen web-palvelimen ominaisuus on HTTP-kutsupolun (URI) käsittely. Express tukee tällaisten käsittelijöiden toteuttamista jo ydinmoduulissa. Pyynnön käsittelijä, joka voi yksinkertaisimmillaan olla mikä tahansa JavaScript-funktio, rekisteröidään joko suoraan sovelluksen juurireitittimeen tai johonkin erikseen luotuun alireitittimeen. Rekisteröinnin yhteydessä annetaan polkumäärittely[19], jonka perusteella Express osaa valita oikean käsittelijän. On myös mahdollista rekisteröidä käsittelijä ilman polkumäärittelyä, jolloin käsittelijä suoritetaan jokaisen pyynnön yhteydessä.

Funktioiden lisäksi käsittelijäksi voidaan asettaa myös alireititin. Alireititin tulkitsee siihen rekisteröidyt polut suhteessa siihen polkuun, johon reititin on itse rekisteröity. Tästä seuraa, että reitittimen ilmentymän voi rekisteröidä useaan eri polkuun, mahdollisesti eri reititinten alle. Reitittimien avulla voi näin luoda hyvinkin monipuolisen käsittelijöiden hierarkian. Tällöin on tosin riski, että hierarkia tuo mukanaan turhaa monimutkaisuutta palvelimen toteutukseen.

Reitittimen ja käyttäjän omien funktioiden lisäksi Express-liitännäismoduulit tarjoavat runsaasti valmiita käsittelijöitä eri käyttötarkoituksiin. Esimerkiksi JSON- tai FormData-tyyppisen datan käsittelyyn on saatavilla monia valmiita moduuleita.

19 Polkumäärittely voi olla kiinteä tai avoin merkkijono. Se voi olla myös säännöllinen lauseke, jonka avulla määrittelyn saa täsmäämään pyynnön polkuun juuri niin tarkkaan kuin halutaan.

3 SOVELLUSKEHYKSEN EROTUSPROSESSI

Tässä luvussa käydään läpi yrityksen sisäiseen käyttöön tarkoitetun web-sovelluskehyksen erotustyö Luminaton käyttöliittymästä. Erotustyön taustamotiivien jälkeen esitellään vaatimusmäärittely ja sen pohjalta vaatimusanalyysi. Työn toteutuksesta esitetään pääpiirteittäin suunnitelma ja käytännön työvaiheet.

3.1 Motivaatio sovelluskehykselle

Päätös sovelluskehyksen erottamiselle Luminaton web-käyttöliittymästä on tullut tuotteen elinkaaren myöhemmässä vaiheessa. Käyttöliittymä oli ollut osana tuotetta jo vuodesta 2007, mistä lähtien se on ollut aktiivisen kehitystyön kohteena. Vuonna 2012 käyttöliittymään oli toteutettu merkittävä käyttökokemusta parantanut uudistus, jolloin käyttöliittymä oli keskeisiltä osin saanut sen muodon, jossa se tämän erotusprojektin aloituksen aikaan oli. Useimmat alkuperäisistä kehittäjistä eivät enää työskennelleet projektissa ja käyttöliittymän kehitysvastuu oli pääosin kahden henkilön vastuulla.

Luminaton käyttöliittymään oli ehtinyt syntyä tässä vaiheessa merkittävässä määrin yleiskäyttöistä toiminnallisuutta, ja uusia ominaisuuksia oli suunnitteilla. Esimerkiksi Engine-ominaisuuden myötä kolmansien osapuolten haluttiin voivan luoda käyttöliittymiin sisältöä liitännäisinä. Samaan aikaan oli alkanut syntyä tarvetta toteuttaa selainkäyttöliittymä myös muihin tuotteisiin. Yksi vaihtoehto olisi ollut toteuttaa jokaiseen uuteen tuotteeseen käyttöliittymä alusta saakka, mahdollisesti eri tekniikoilla. Tämä vaihtoehto olisi varmasti tuonut joustavuutta kehitystyöhön.

Yrityksen tunnistettavuuden kannalta katsottiin kuitenkin edulliseksi, että tuotteiden ulkoasu olisi samanlainen. Myös käyttökokemuksen kannalta samanlainen toiminnallisuus nähtiin hyödylliseksi. Kolmantena tekijänä oli resurssointi: eri tekniikoiden tutkiminen jokaisen tuotteen kohdalla olisi välttämättä vienyt enemmän aikaa. Näiden syiden nojalla ja olennaisesti myös siksi, että yrityksellä ei ollut toista valmista web-käyttöliittymätoteutusta saatavilla, sovelluskehys päätettiin toteuttaa nimenomaan erottamalla se Luminaton käyttöliittymästä.

3.2 Vaatimusmäärittely

Sovelluskehyksellä ja projektilla tuli olla nimi. Tässä tapauksessa päädyttiin suoraviivaisesti nimeen WebUI Core tai lyhyesti *Core*.

Seuraavassa on listattu Corelle ja erotustyölle asetettuja keskeisiä vaatimuksia. Koska työn lähtökohtana oli Luminaton käyttöliittymä, kaikki Luminaton alkuperäiset vaatimukset olisivat luonnostaan mukana myös Coressa, niiltä osin kuin Luminaton toteutusta siirrettäisiin Coreen. Seuraavassa ei kuitenkaan ole, joitakin teknisiä vaatimuksia lukuun ottamatta, listattu Luminaton vaatimuksia:

– Luminatosta tunnistetaan ja erotetaan yleiskäyttöiset osat, joiden pohjalta luodaan web-sovelluskehys yrityksen sisäiseen käyttöön. Kehystä tullaan käyttämään pääasiassa IP- ja kaapeliverkkolaitteiden käyttöliittymien kehitystyössä.

– Luminatoon ei erotustyön takia tehdä toiminnallisia muutoksia.

– Coren selainkoodi on ECMAScript 5 -standardin ja CSS-tyylit version 2 mukaista. HTML on version 4 mukaista.

– Core tukee uusimpia Chrome ja Firefox -selaimia ja Internet Explorer -selainta alkaen versiosta 9.

– Core-käyttöliittymien kehitystyötä tulee voida tehdä myös Windows-ympäristössä.

– Core tukee tietomallien synkronoinnissa REST-rajapintaa ja JSON-formaattia sekä DXI-protokollaa. Näistä jälkimmäistä tuetaan yksinomaan Luminatoa varten ja jatkossa keskitytään tukemaan lähinnä RESTiä.

– Core ei tietomallien synkronointirajapintaa lukuun ottamatta aseta vaatimuksia laitteiden toteutukselle.

– Corella tulee voida aloittaa uuden käyttöliittymän kehitystyö nopeasti ja vaivattomasti uudessa tuotteessa.

3.3 Vaatimusanalyysi

3.3.1 Coren arkkitehtuuri

Core on sovelluskehys. Se ei ole siis ohjelmistokirjasto sen perinteisessä merkityksessä. Kehyksen avulla luodut käyttöliittymät ovat ennen muuta sovelluskehyksen laajennuksia. Näin ollen Coren arkkitehtuuri asettaa jo sovelluskehysten määritelmän perusteella vaatimuksia ja rajoituksia sovelluksille. Vaatimusten tulee kuitenkin olla tarkoituksenmukaisia siten, että joustavuudesta tinkimällä saadaan lisähyötyä suhteessa enemmän muilla osa-alueilla.

Se, että sovellusalue on selkeästi rajattu, helpottaa kehyksen suunnittelua, koska kaikkia mahdollisia käyttötapauksia ei tarvitse ottaa huomioon. Järkevillä rajoituksilla ja suunnittelupäätöksillä käyttöliittymien kehitystyötä voi ohjata hyviksi havaittuihin toimintamalleihin sulautettujen verkkolaitteiden käyttöliittymien osalta.

On kuitenkin huomattava, että Luminatossa on jo valmiiksi ratkaistu monia sovellusalueen käyttöliittymiin yleisesti liittyviä ongelmia. Näin ollen monet Luminaton suunnitteluratkaisut periytyvät lähes suoraan Coreen. Koska erotusprojektin käytössä olevat resurssit ovat rajalliset, suuria arkkitehtuurimuutoksia tehdään vain, jos ne ovat välttämättömiä tai ne tuovat selkeää lisähyötyä. Coren arkkitehtuuriin uusia suunnitteluratkaisuja ei siis uskota tehtävän merkittävästi. Tärkeässä roolissa on kuitenkin Luminaton arkkitehtuurin suunnitteluratkaisujen tunnistaminen, joista keskeisimmät esiteltiin luvussa 2.4.

3.3.2 Luminaton arkkitehtuurin soveltuminen erotustyöhön

Sovelluskehyksen erottaminen suuresta koodikannasta ei ole helppo tehtävä, jos asiaa ei ole huomioitu riittävästi alusta alkaen. Usein voi olla niin, että käyttöliittymän kehitystyön alkuvaiheessa halutaan saada nopeasti näkyvää valmiiksi. Erotustyötä, tai yleensä muutostyötä helpottavien tekniikoiden käyttöä ei ehkä ole haluttu tai ehditty hyödyntää. Erotustyö on kuitenkin helpompaa, jos kehitystyössä on noudatettu olio-ohjelmoinnin periaatteita, kuten modulaarisuutta, abstraktointia ja enkapsulointia. Myös suunnittelumallien johdonmukainen soveltaminen heti kehitystyön alusta lähtien auttaa tässä suhteessa.

Vaikka Luminaton käyttöliittymän arkkitehtuuri on modulaarinen, ei sovellusta kokonaisuutena ole suunniteltu yleiskäyttöisyyden näkökulmasta. Tämän voi ymmärtää jo sovelluksen kehityshistoriasta, mutta sen näkee myös koodikantaa analysoidessa joidenkin komponenttien välisissä kehäriippuvuuksissa ja modulaarisuutta rikkovissa ratkaisuissa.

Suurin kehäriippuvuutta sisältävä komponenttikokonaisuus on sovelluksen ytimessä sijaitseva MVC-kokonaisuus. Sen lisäksi, että metamallit, tietomallit ja attribuutit ovat tiukasti sidoksissa toisiinsa, sitoutuu kokonaisuuteen muussa kuin kuuntelijan roolissa myös näkymä-, transaktio- ja synkronointikomponentit. Riippuvuuksia on toki vaikea välttää sovelluksen keskeisten komponenttien osalta, mutta ongelma tuleekin ymmärtää enemmän abstraktien rajapintaluokkien puuttumisena, minkä voi ennakoida vaikeuttavan Coressa mahdollisesti myöhemmin tarvittavia muutoksia. Yksi keskeinen haaste tulisi olemaan näiden ratkaisuiden tunnistaminen ja mahdollinen korjaaminen.

Toinen ongelmallinen kohta Luminatossa on synkronointikomponentin voimakas riippuvuus tietomalleihin. Tämän voi ymmärtää sillä, että aikaisemmin Luminato on käyttänyt pelkästään DXI-protokollaa attribuuttikohtaiseen synkronointiin. Synkronointi-komponentin sitominen suoraan tietomalleihin on siis sinällään ollut perusteltua. Coren vaatimusten kannalta ratkaisu on kuitenkin hyvin ongelmallinen, koska jo nykyinen toteutus REST-synkronoinnille on enemmän DXI-toteutuksen päälle liimattu kuin kunnollinen modulaarinen komponentti, joka ideaalisesti toimisi DXI:n rinnalla yhteistä synkronointirajapintaa vasten. Vaikka synkronoinnin refaktorointi olisi suureksi eduksi, ei tähän kuitenkaan uskota riittävän resursseja ainakaan tässä vaiheessa.

Yksinkertaisimpia tapauksia lukuun ottamatta, analyysissa on vaikea löytää komponentteja, joilla ei olisi riippuvuuksia toisiin komponentteihin. Riippuvuudet eivät tietenkään sinällään ole ongelma, vaan enemmänkin se, että komponentit käsittelevät yleensä suoraan konkreetteja luokkia tai pahimmillaan käyttävät suoraan ilmentymien muuttujia. Toisaalta JavaScriptin luonteen ja tehokkuusnäkökohdat huomioiden rajanveto enkapsuloinnin rikkomisen ja pragmaattisen lähestymistavan välillä voi olla vaikeaa. Koodista löytyy myös paljon toisistaan riippumattomia osuuksia, mutta näissä on kyse enemmän avustavasta toiminnallisuudesta.

Ongelmista huolimatta voidaan todeta Luminaton arkkitehtuurin olevan varsin toimiva kokonaisuus, eikä sovelluskehyksen erottamisen pitäisi kaatua ainakaan koodikannan rakenteeseen. Muutostyön hallintaa helpottavien tekniikoiden käyttö toisi lisätyötä,

mutta on pitkän päälle kannattavaa. Erityisen kannattavaa se on sovelluskehysten tapauksessa, koska kehyksen elinvoimaisuus piilee ennen muuta sen joustavuudessa muutoksille. Tästä syystä Coressa viimeistään on tarkoitus ottaa nämä tekniikat käyttöön.

3.3.3 Perustelu Coren erottamiselle Luminatosta

Voidaan esittää kysymys, olisiko Core järkevämpi toteuttaa kokonaan alusta saakka erillisenä projektina. Uudelleentoteutus voi olla hyvinkin järkevää tilanteissa, joissa koodikannan mukauttaminen todetaan vaikeaksi, esimerkiksi kehitystekniikoiden tai dokumentoinnin puutteellisuuden takia tai jos sovellusalue on erittäin spesifinen. Tai pelkästään siitä syystä, ettei tuotteen aikaisempia kehittäjiä ole enää saatavilla. Edellä tarkasteltiin Luminaton koodikannan soveltuvuutta muutos- ja erottamistyöhön ja todettiin, ettei tämän suhteen ollut suuria ongelmia.

Myös laadulliset seikat voivat vaikuttaa päätökseen. Uuden tuotteen luominen alusta antaa ainakin teoriassa mahdollisuuden tehdä suunnittelu ja toteutus huolella. Onhan hyvän sovelluskehyksen edellytyskin jo, että näin tapahtuu. Aikataulutekijät ja muut resurssipaineet ovat kuitenkin yritysmaailmassa aina läsnä, jolloin alkuperäinen hyvä idea saattaa kaatua lopulta täysin tekijöistä riippumattomiin syihin.

Luminaton tapauksessa on melko perusteltua nimenomaan erottaa yleiskäyttöiset osat ja mukauttaa ne kehykseksi, eikä tehdä kaikkea alusta saakka. Tämä on itse asiassa hyvin yleinen tapa toimia, vaikka suositus onkin yleensä luoda kehys vasta, kun tuotteita on kaksi tai useampia, jolloin nähdään selkeämmin, mitä ominaisuuksia kehyksessä todella tarvitaan. Tässä tapauksessa valmiita sovelluksia on vain yksi, minkä voi nähdä perustelua heikentävänä tekijänä.

3.3.4 Uuden sovelluksen luominen Coren avulla

Uuden käyttöliittymän luominen Corella tulee olla nopeaa ja helppoa. Kehitystyön nopeuden arvioinnissa yksi mittari on uuden projektin alkuun saamisen helppous.

Ihanteellisesti Coressa olisi toteutettuna parametrisoidut generaattorit uuden projektin luomiseksi. Koska Core ei kuitenkaan ole koko maailman käyttöön tarkoitettu kehys, on

syytä miettiä tarkkaan, kuinka paljon generaattorien toteutukseen voidaan käyttää resursseja. Jos kehyksen käyttäjät tuntevat koodin erittäin hyvin, kuten tässä tapauksessa on odotettavissa, ei käsin kopiointi uuteen hakemistoon ole paljon hitaampi ratkaisu. Tällöin kannattaa vähintäänkin luoda malli, joka voidaan tarvittaessa suoraan kopioida uudeksi projektiksi. Ongelmaksi tässä voi tulla kuitenkin mallien ylläpito, koodin eläessä.

Helpoin ja suoraviivaisin ratkaisu voisi olla lopulta kokonaisen, valmiin käyttöliittymätoteutuksen kopiointi uudeksi projektiksi ja poistaa siitä kaikki ylimääräinen. Kyseessä olisi edelleen eräänlainen malli, joka ei kuitenkaan vaadi ylläpitotyötä.

Sovelluskehysten näkökulmasta vähimmäistavoitteeksi on järkevää asettaa mallipohjainen toteutus, koska varsinaisen tarkoituksensa lisäksi se toimii myös hyvänä dokumentaationa.

3.3.5 Komponenttien arkkitehtuurimuutokset

Coren kannalta olisi edullista, jos erotustyön aikana voitaisiin yleistää Luminaton komponentteja paremmin kehyksen käyttötarkoitusta vastaavaksi. Seuraavassa on listattu komponentteja, joihin ainakin tehdään muutoksia.

Moduulien latausjärjestys, alustus ja konfigurointi

Luminaton moduulien latausjärjestyksen merkitsevyys on ohjelman alustuksen kannalta jossain määrin ongelmallinen. Sen lisäksi, että järjestysriippuvuus on jo itsessään ongelmallista, on alustuksessa helppo tehdä vaikeasti havaittavia loogisia ohjelmointivirheitä, jotka pahimmillaan estävät koko sovelluksen käynnistymisen.

Luminatossa ei ole ollut tarvetta konfiguroida moduuleja ohjelman käännöksen tai suorituksen aikana. Coressa tällainen tarve kuitenkin on, koska mahdollisuus jättää moduuleita pois käännöksestä ja mahdollisuus mukauttaa niiden käyttäytymistä konfiguraation perusteella on joustavuuden kannalta tärkeä ominaisuus. Vaihtoehtoisten moduulitoteutusten käyttö saattaisi olla tietyissä tilanteissa myös hyödyllinen ominaisuus. Käytössä olevien resurssien määrän huomioiden on kuitenkin hyvin kyseenalaista, tultaisiinko vaihtoehtoisia moduulitoteutuksia laajasti koskaan tekemään.

Edelliseen perustuen moduulien alustukseen tehdään seuraavat muutokset:

– Moduulien latausjärjestyksen merkitsevyyttä vähennetään.

– Moduuleihin lisätään sovelluskohtainen konfigurointimahdollisuus.

Istunnonhallinta

Istunnonhallinta on ollut Luminatossa kiinteä osa sovellusydintä. Koska kyse on kuitenkin selkeästi erillisestä, rajattua tehtävää hoitavasta komponentista, erotetaan se Coressa kokonaan erilliseen moduuliin. Koska istunnonhallintaan liittyvät yksityiskohdat riippuvat tilannekohtaisesti sovelluksesta, istunnonhallinta toteutetaan niin, että sitä on helpompi erikoistaa ja laajentaa.

Metamallit ja -oliot

Analyysin perusteella ymmärretään Luminaton tietomallien sisältävän huomattavasti refleksiivisiä piirteitä ja potentiaalia niiden soveltamiseen. Koska Luminaton metamäärittely ja näin ollen metaoliot luodaan ajonaikaisesti, olisi jo tässä vaiheessa mahdollista vaikuttaa mallien ilmentymien käyttäytymiseen muokkaamalla metamäärittelyä. Luminatossa tätä potentiaalia ei erityisemmin ole käytetty, eikä jo luotuja metaolioita pääsääntöisesti muokata luonnin jälkeen. Sen sijaan metaolioiden introspektion voi hyvin sanoa olevan tietomallien toiminnallisuuden ydintä, koska kaikki malleja käsittelevät komponentit tavalla tai toisella lukevat metaolioita.

Mallien osalta tuki refleksiivisyydelle pidetään vähintään sellaisena kuin se on Luminatossa. Käytännössä muutoksia joudutaan kuitenkin tekemään, koska Coressa luokka- ja metamäärittelyt tulevat sijaitsemaan valtaosin kehystä käyttävässä sovelluksessa. Core itsessään tulee sisältämään alkuvaiheessa vain vähän tietomalliluokkia.

Coren moduuleihin lisätään mahdollisuus konfiguroida niitä ja malleihin liittyvät komponentit ovat yksi potentiaalinen konfiguroinnin kohde. Refleksiivisyyden rinnalle lisätään siis lisäksi staattista muokattavuutta. Käytännössä tämä tarkoittaa, ettei metakomponentille annettava metakonfiguraatio ole jatkossa yksikäsitteinen.

3.3.6 Windows-kehitysympäristö

Luminaton CATER-käännösjärjestelmä toimii ainoastaan Linux-ympäristöissä. Keskeiset web-kehitystyökalut, kuten IDE[20], ovat kuitenkin Windows-pohjaisia, koska tuotekehitys käyttää pääsääntöisesti Windows-koneita. Luminaton projektihakemistoa on jouduttu jakamaan virtuaalikoneessa ajettavasta Linuxista, mikä on ollut merkittävä kehitystyön hidaste.

Kehitystyön tehostamiseksi Coreen halutaan Windowsia tukeva käännöstyökalu. Tuki make-ohjelmalle kuitenkin säilytetään, koska tuotteiden julkaisuversioiden käännökset tehdään useimmiten Linuxissa. Käytännössä tämä tarkoittaa, että Makefile delegoi käännöskomennot uudelle käännöstyökalulle.

Nykyinen kehitysaikainen käyttöönotto -ominaisuus säilytetään. Mahdollisuuksien mukaan sitä parannetaan, koska käännetyt tuotanto- ja kehitysjulkaisut tulee voida siirtää nopeasti todelliseen laiteympäristöön testattavaksi.

JavaScript on web-maailmassa ainoa vaihtoehto ohjelmointikieleksi. Olisikin eduksi, jos myös käännösympäristön voisi toteuttaa samalla kielellä. Node-teknologia tarjoaa juuri tämän mahdollisuuden. Sen lisäksi, että Nodea käytetään yhä enemmän web-sovellusten taustaosuudessa, sitä voi hyvin käyttää myös skriptien ja laajempien ohjelmien toteutuksessa. Tässä mielessä Node voi helposti korvata Autotoolsin kaltaiset järjestelmäriippuvaiset työkalut.

Käännöstyökalu toteutetaan Node-sovelluksena, ja valmiita moduuleita käytetään hyväksi niin paljon kuin mahdollista. Muilta osin toteutuksen suhteen ei ole asetettu yksityiskohtaisia vaatimuksia, joten työkalun luomisen voi nähdä myös eräänlaisena opiskeluprojektina Node-tekniikkaan.

3.3.7 Haasteet ja riskit

Resursointi on erotusprojektin suurin riski. Kahden kokeneen suunnittelijan rinnalla projektissa on mukana web-sovellusten suhteen noviisi harjoittelija. Luminaton rinnakkain jatkuva kehitystyö vie osan näistäkin resursseista. On siis selvää, että erotusprojektin aikana jokainen päätös arkkitehtuurin muuttamisesta on jonkinasteinen

20 JetBrains WebStorm.

riski. Osittain tästä syystä suuria muutoksia komponentteihin ei edes tehdä, jos ne eivät ole välttämättömiä. Pienempiin muutoksiin liittyy aina myös riskejä, mutta niiden ei uskota kaatavan koko projektia.

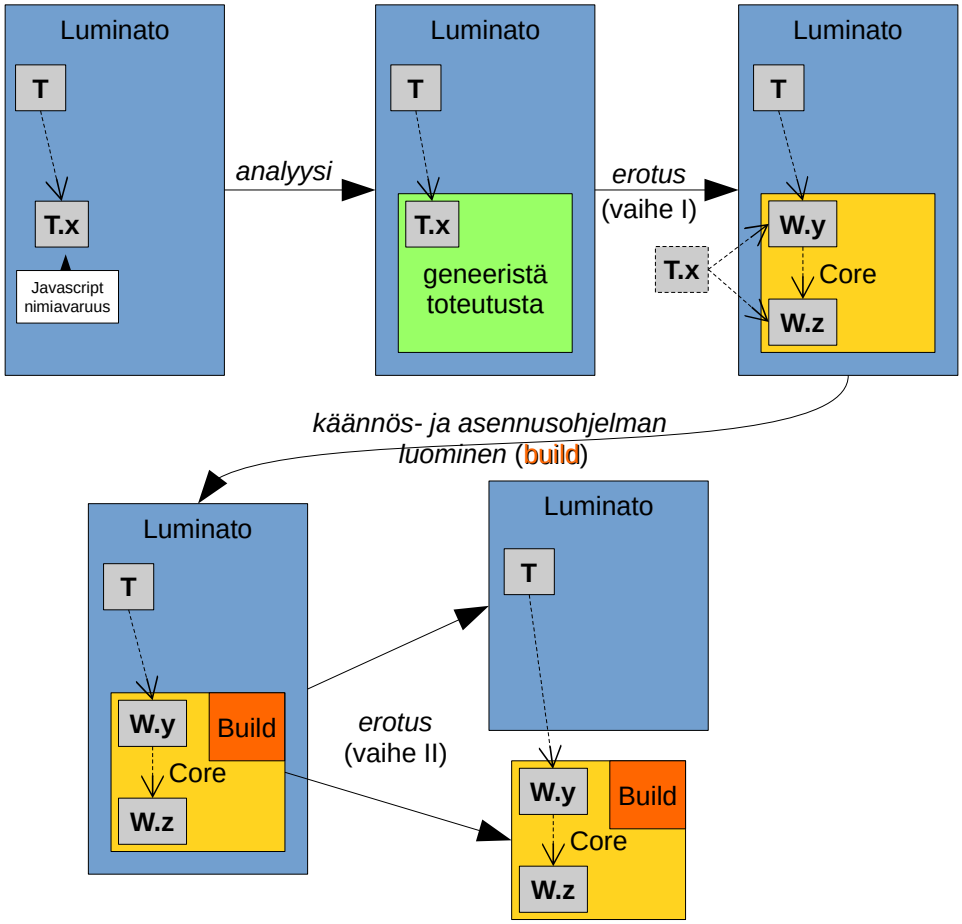
Toinen riskitekijä, ja myös haaste, on käännösympäristön toteuttaminen entuudestaan vieraassa ohjelmointiympäristössä. Riskiä lisää se, ettei ole täysin selvää, miten eri projektihakemistoissa sijaitsevat Core ja sitä laajentava sovellus pitäisi käännössovelluksella yhdistää. Luminatossa tätä ongelmaa ei ole ollut, koska koko käyttöliittymän lähdekoodi on sijainnut yhdessä projektissa.

Kolmas riski on Luminaton Backbone-kehys, jonka käytöstä haluttaisiin luopua, koska jo nyt Luminato korvaa monet sen valmiit toteutukset. Backbonesta luopuminen edellyttäisi kuitenkin niin suuria muutoksia, ettei siihen katsota olevan varaa. Riski syntyy, koska Core tulee olemaan myös jatkossa tiukasti sidottu Backboneen, ja usean käyttöliittymän sidonnaisuus Coreen voi tehdä Backbonesta irrottautumisen erittäin vaikeaksi.

3.4 Toteutus

3.4.1 Suunnitellut työvaiheet

Coren erottaminen suunniteltiin toteutettavaksi selkeissä työvaiheissa, jotka on esitelty tässä luvussa. Kuvan 7 mukaisesti Luminaton lähdekoodi jakautuu lopputuloksena kahteen itsenäiseen komponenttiin.



Kuva 7. Coren erotusprosessin työvaiheet.

Lähdekoodien erottaminen

Vaiheen tärkein tavoite on siirtää Coren toteutus nimiavaruuden **W** alle erillisiin tiedostoihin Luminato-projektin sisällä. Refaktorointia tehdään niiltä osin, kuin suorat siirrot eivät onnistu tai muutokset ovat mielekkäitä ja aikataulu antaa myöden. Työvaiheen lopputuloksena Luminaton käyttöliittymä näyttää ja toimii täsmälleen samalla tavoin kuin ennen erotustyötä. Kaikki muutokset ovat pinnan alla. Tähän

vaiheeseen odotetaan kuluvan eniten aikaa koko projektissa, ja aikaa työhön on varattu muutama kuukausi.

Jos on epäselvää mihin komponenttiin toteutus kuuluu, painopisteeksi valitaan näissä tapauksissa Luminato. Coreen ei haluta ylimääräistä koodia tässä vaiheessa. Toiminnallisuutta voidaan siirtää helposti myöhemmin.

Uuden käännöstyökalun toteuttaminen

Käännöstyökalun toteuttamiseen varataan aikaa muutamia viikkoja. Tavoitteena on toteuttaa ohjelma, jolla Core ja sitä laajentava sovellus voidaan yhdistää, ja yhdistämisen lopputulos asentaa haluttuun kohdehakemistoon.

Komponenttien erotus

Coren osuus siirretään Luminaton lähdekoodin joukosta erilliseen versiohallittuun projektiin. Aikaa ei työhön odoteta kuluvan paljon, mutta vähintään tarvitaan uusi säilö versionhallintapalvelimelta.

Lopputuloksena Luminato ja Core ovat erilliset projektit. Luminaton käyttöliittymä toimii ja näyttää edelleen samalta kuin ennen erotusta.

Coren dokumentointi ja esimerkkisovelluksen toteuttaminen

Core dokumentoidaan kattavasti. Dokumentointia pidetään erityisen tärkeänä, koska yleiskäyttöisenä komponenttina sen käytön esteenä ei saa olla vaikeakäyttöisyys. Tavoitteena on, että tuotteiden edusosan kehitystyöhön voidaan tarpeen tullen lisätä resursseja, ilman kohtuuttoman korkeaa oppimiskynnystä.

Uuden kehyksen avulla luodaan myös pienimuotoinen sovellus, joka tulisi toimimaan esimerkkinä ja testiympäristönä Coren ominaisuuksille.

3.4.2 Lähdekoodien erottaminen

Lähdekoodien erottamista tehtiin suunnitelman mukaisesti Luminaton kehitystyön rinnalla saman projektihakemiston sisällä. Corea varten luotiin omat alihakemistot, joihin Coreen kuuluva toteutus siirrettiin ja luotiin. Luminaton moduulien tiedostonimet pidettiin valtaosin samoina, mutta refaktoroinnin ja uudelleentoteutusten myötä uusia

tiedostoja syntyi myös Luminatoon. Tiedostojen siirron kaltaisiin helppoihin muutoksiin lukeutuivat myös muuttujien uudelleennimeämiset ja objektiviittausten vaihtaminen osoittamaan Coren **W-**nimiavaruuteen.

Tiedostot voitiin jakaa neljään kategoriaan niiden sisältämän Luminaton ja Coren toteutuksen suhteen perusteella. Yleensä oli helppo tehdä valinta sen suhteen, kumpaan projektiin yksittäinen komponentin toteutus kuului (taulukko 3).

Taulukko 3. Komponenttien toteutuksen jakautuminen Luminaton ja Coren välillä. Taulukkoon on kirjattu myös arvio työmäärästä. Vaikka osa komponenteista siirtyi kokonaan Coreen, työmäärä saattoi olla silti suuri komponenttiin tehtyjen sisäisten muutosten takia.

Komponentti	Luminato	Core	Työmäärä
Sovellusydin	20,0%	80,0%	++++++
Sessionhallinta	50,0%	50,0%	+++++
Malli	75,0%	25,0%	+++
Meta	85,0%	15,0%	++
Näkymä	75,0%	25,0%	++++
Ohjain	75,0%	25,0%	+++++
Validointi	25,0%	75,0%	++
Logopaneeli	50,0%	50,0%	++
Toiminnot	75,0%	25,0%	++
Engine		100,0%	+
DXI		100,0%	+++++
JSON		100,0%	+
Jasmine	90,0%	10,0%	++

Ensimmäiseen luokkaan kuuluivat tiedostot, jotka sisälsivät ainoastaan Luminaton toteutusta. Näihin tiedostoihin tehtiin hyvin yksinkertaisia muutoksia, kuten objektiviittausten muutoksia Coren nimiavaruuteen.

Toiseen luokkaan sijoittuivat tiedostot, jotka sisälsivät pelkästään Coreen kuuluvaa toteutusta. Näissä täytyi vähintäänkin muuttaa nimiavaruus ja monesti myös muuttujia piti nimetä uudelleen.

Kolmannessa luokassa olivat tiedostot, jotka sisälsivät merkittävissä määrin sekä Luminaton että Coren toteutusta. Tällaisten tiedostojen mukauttaminen vaati yleensä tarkempaa koodin analyysia ja valintaa. Esimerkiksi tietomalleihin liittyvissä tiedostoissa ei riittänyt pelkästään osien siirto. Mallien luokkarakenteen määrityksiä jouduttiin myös muuttamaan.

Viimeisenä ja haastavimpana luokkana olivat tiedostot, joihin jouduttiin tekemään suuria muutoksia, kuten luokkien yleistämistä tai osittaista uudelleentoteuttamista. Työmääriä arvioiden sovellusytimeen ja sessionhallintaan liittyvä toteutus tuotti erityisesti päänvaivaa.

Luminaton toteutus oli yleensä jo valmiiksi varsin hyvin eroteltu kantaluokista erikoistuneisiin luokkiin, joten käänteistä yleistystä ei katsottu usein tarpeelliseksi. Koska koodi oli valmiiksi modulaarista, tiedostoja voitiin monissa tapauksissa siirtää ilman monimutkaisia muutoksia niiden sisältöihin. Toisaalta tiedostot sisälsivät usein sekä kantaluokan, että siitä erikoistetut aliluokat. Osa tiedostoista olikin huomattavan suuria ja kuormittivat kehitystyökaluja siinä määrin, että olisi ollut edullista suorittaa luokkien pilkkomista erillisiin tiedostoihin. Useimmiten tällaiseen optimointiin ei katsottu olevan aikaa.

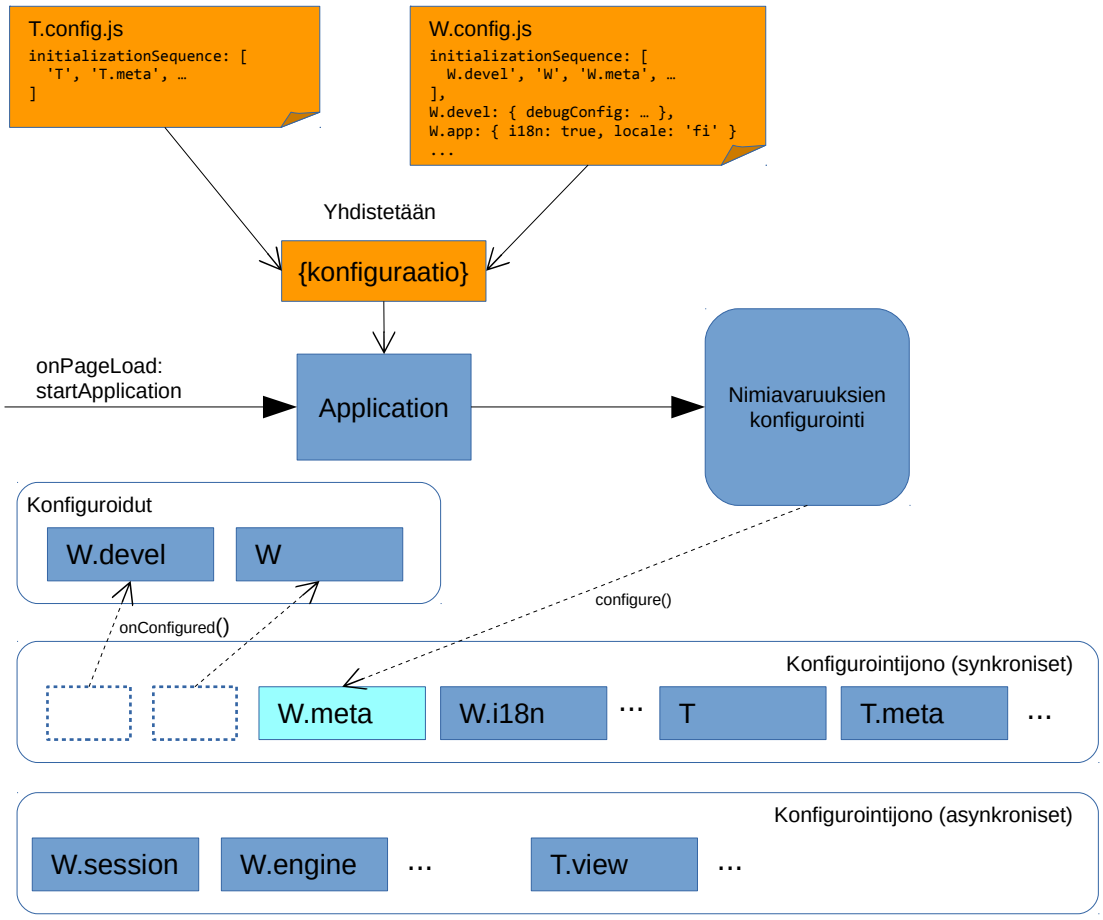
Luonteeltaan vaihe oli toisaalta mekaanista huolellisuutta ja tarkkaavaisuutta vaativaa työtä, mutta toisaalta olio-ohjelmoinnin periaatteita noudattavaa suunnittelua ja toteuttamista sekä tilannekohtaista analyysia. Erityismaininta voidaan antaa käytössä olleen IDE:n monipuolisille ja havainnollisille lähdekoodin muokkaustoiminnoille. Erityisesti tekstin korvaaminen oli näppärä ja nopea suorittaa, ainakin niiltä osin kuin merkkijonojen, muuttujien ja funktioiden nimet olivat yksikäsitteisiä. Suurena etuna olikin, ettei Luminatossa merkkijonoja yleensä oltu koostettu paloista.

Mittavia arkkitehtuurimuutoksia ei lopulta toteutettu lainkaan. Tämä ei tarkoita, etteikö merkittäviä muutoksia olisi tehty, vaan pikemmin niiden luonne oli enemmän lisäyksen kaltaista tai sellaista, minkä seuraukset eivät ulottuneet laajalle muuhun toteutukseen. Seuraavassa on selvitetty joitakin haasteellisempia muutoskohtia.

Nimiavaruudet ja sovelluksen konfigurointi

Luminatossa komponenttia vastasi tyypillisesti yksittäinen tiedosto tai moduuli, jossa välittömästi alustettiin luokat ja muu toteutus. Moduulien latausjärjestys oli merkitsevä, koska moduulien toteutus sisälsi aina viittauksen vähintään **T-**nimiavaruuteen, mutta yleensä myös muihin nimiavaruuksiin.

Joustavuuden lisäämiseksi Coreen tehtiin nimiavaruutta mallintava Namespace-luokka, joka tukee komponentin suoritusaikaista konfigurointia ja asynkronista alustamista. Komponenttien moduulit ladataan edelleen järjestyksessä, mutta nyt jokaiselle nimiavaruudelle kutsutaan myöhemmin sovellusytimen alustuksen yhteydessä metodia, jossa nimiavaruus voi suorittaa synkronisia tai asynkronisia alustustoimenpiteitä. Kutsun aikana nimiavaruus lukee konfigurointiparametrinsa ja mukauttaa itsensä niiden pohjalta. Kuvassa 8 on esitetty nimiavaruuksien konfigurointiprosessi. Vaikka nimiavaruusviittausten osalta moduulien latausjärjestys on edelleen merkitsevä, ratkaisulla saavutettiin kuitenkin huomattavaa joustavuutta muun alustuksen osalta.



Kuva 8. Nimiavaruuksien suoritusaikainen konfigurointi.
Sovelluksen konfiguraatio yhdistetään Coren oletuskonfiguraatioon. Yhdistämisen tulos annetaan sovellusytimelle, joka suorittaa nimiavaruuksien konfiguroinnin osana omaa alustustaan.

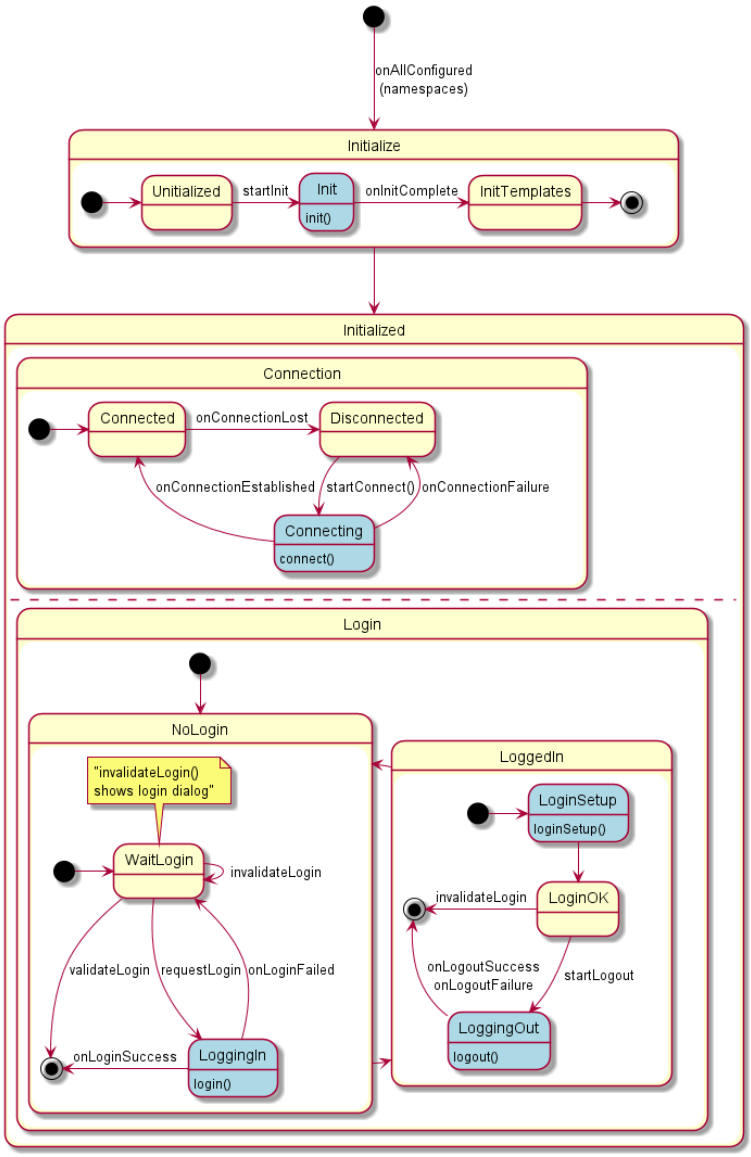
Sovellusydin ja sessionhallinta

Luminatossa sovellusytimen ja istunnonhallinnan luokat oli määritelty samassa moduulissa. Coressa molemmat toteutettiin uudelleen ja sijoitettiin erillisiin moduuleihin ja komponentteihin, koska niillä katsottiin olevan riittävän erilaiset vastuualueet. Molemmat luokat suunniteltiin ainokais-ilmentyminä[21] ja sovelluksessa erikoistettaviksi.

Luokkien vastuualueet olivat suurimmaksi osaksi selkeät, mutta joidenkin ominaisuuksien kohdalla tilannetta jouduttiin miettimään tarkemmin. Yksi tällainen liittyi säännöllisesti suoritettavaan laiteyhteyden tarkistukseen. Toisaalta yhteyden tarkistuksen voi ajatella kuuluvan sovellusytimelle, koska kyseinen toiminto saatetaan haluta tehdä myös ilman voimassaolevaa istuntoa. Toisaalta yhteydenhallinnan voi ymmärtää olennaisesti kuuluvan istuntoon, koska istunnonhallinta muutenkin suorittaa säännöllisesti istunnon oikeellisuuteen liittyviä kyselyitä. Vaikka kyse on oikeasti eri toiminnallisuuksista, Coressa päädyttiin pragmaattiseen päätökseen pitää molempien vastuut istunnonhallinnalla, vieläpä niin, ettei molempia kyselytyyppejä koskaan tehdä samanaikaisesti.

Vaikka luokat toteutettiin erikoistamista silmällä pitäen, sovellusydin ei käytännössä vaadi sovellukselta omaa toteutusta. Istunnonhallinnan kohdalla tilanne on kuitenkin täysin toinen. Yhteyden varmistus, käyttäjän sisäänkirjautuminen, istunnon oikeellisuuden varmennus ja uloskirjautuminen täytyy kaikissa sovelluksissa toteuttaa erikseen. Istunnonhallinta onkin oikeastaan tilakone (kuva 9, seuraava sivu), jolle sovellus ilmoittaa aloittamiensa toimintojen lopputulokset. Istunnonhallinta varmistaa, että operaatiot tehdään loogisesti oikeassa järjestyksessä. Sovellusydin kuuntelee istunnonhallinnan lähettämiä tilamuutosten tapahtumia ja reagoi niihin näyttämällä esimerkiksi päänäkymän, kirjautumisnäkymän tai viestidialogin.

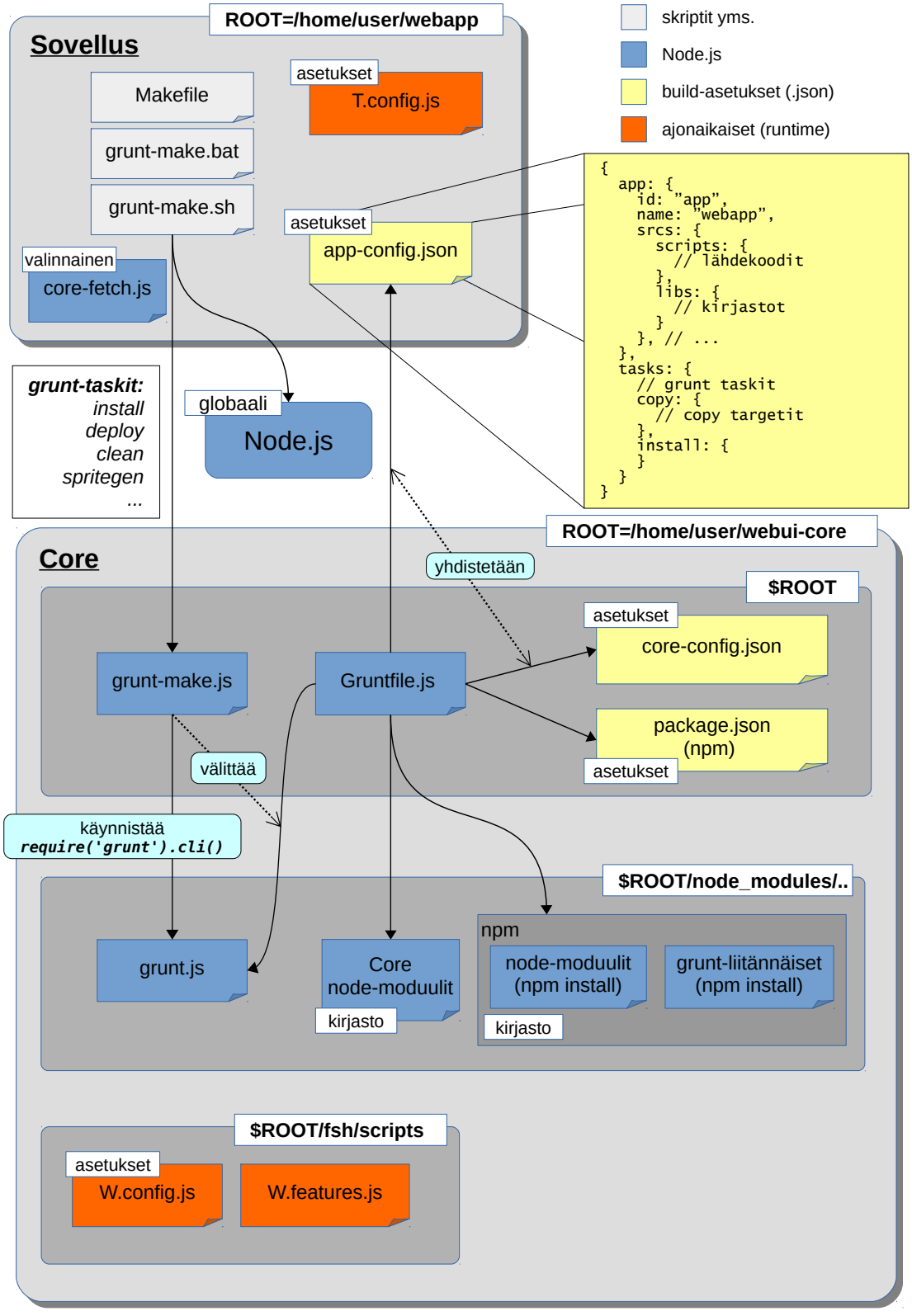
21 Suunnittelumalli, jossa luokasta luodaan sovelluksen aikana korkeintaan yksi ilmentymä.



Kuva 9. Istunnonhallinnan tilat ja niiden väliset siirtymät.
Sinisellä esitetyt tilat ovat sovelluksen toteuttamia osuuksia.

3.4.3 Käännös- ja asennustyökalun toteutus

Uusi käännös- ja asennustyökalu, josta käytetään jatkossa nimeä *build*, toteutettiin Grunt-ohjelman avulla. Build perustuu eri hakemistoissa sijaitseviin sovelluksen ja Coren lähdekoodeihin. Buildin tärkein tehtävä on yhdistää hakemistojen sisällöt yhdeksi käyttöliittymäsovellukseksi. Kuvassa 10 (seuraava sivu) on esitetty buildin arkkitehtuuri.



Kuva 10. Käännösohjelman komponenttien keskinäiset riippuvuudet.
Suoritusaikaisista asennustiedostoista on esitetty vain muutama konfiguraatiotiedosto.

Buildissa sovelluksen juurihakemisto toimii työhakemistona, josta käsin kutsutaan Coren juurihakemistossa sijaitsevaa grunt-make.js-käynnistysskriptiä. Skriptin voi suorittaa joko suoraan Nodella tai apuskripteillä[22]. Käynnistysskriptille annetaan argumentteina ne Grunt-tehtävät, jotka halutaan suorittaa. Skripti käynnistää Coren node_modules-hakemiston alla sijaitsevan varsinaisen Grunt-ohjelman. Grunt lukee Coren juurihakemistosta Gruntfile.js-tiedoston, jossa on varsinainen komennon suorittava logiikka.

Buildin konfiguraatio sijaitsee tyypillisestä Gruntista poiketen sovelluksen app-config.json ja Coren core-config.json -tiedostoissa. Gruntfile.js lukee molemmat tiedostot, ja konfiguroi suoritettavat tehtävät dynaamisesti yhdistämällä sovelluksen ja Coren konfiguraatiot.

Buildin tärkeimmät tehtävät ovat:

– **install**: kääntää ja asentaa sovelluksen kohdehakemistoon. Asennettavien tiedostojen määrään vaikuttaa build-moodi: julkaisu, testijulkaisu tai kehitysversio. Moodi vaikuttaa asennettavien tiedostojen määrän lisäksi suoritusaikana tehtyjen virhetarkistusten, lokiviestien ja kehitysaikaisten apumoduulien määrään.

– **deploy**: julkaisee sovelluksen kohdelaitteeseen testaamista varten. Muuttuneet tiedostot tiivistetään arkistotiedostoksi, joka siirretään SSH:lla laitteeseen. Laitteessa suoritetaan lopuksi etäkomennot arkiston purkamiseksi ja tiedostomoodien muuttamiseksi.

– **node-server-install/run**: asentaa/suorittaa paikallisen testipalvelimen.

– **spritegen**: luo sovelluksen sprite-kuvan automaattisesti yksittäisistä ikoneista.

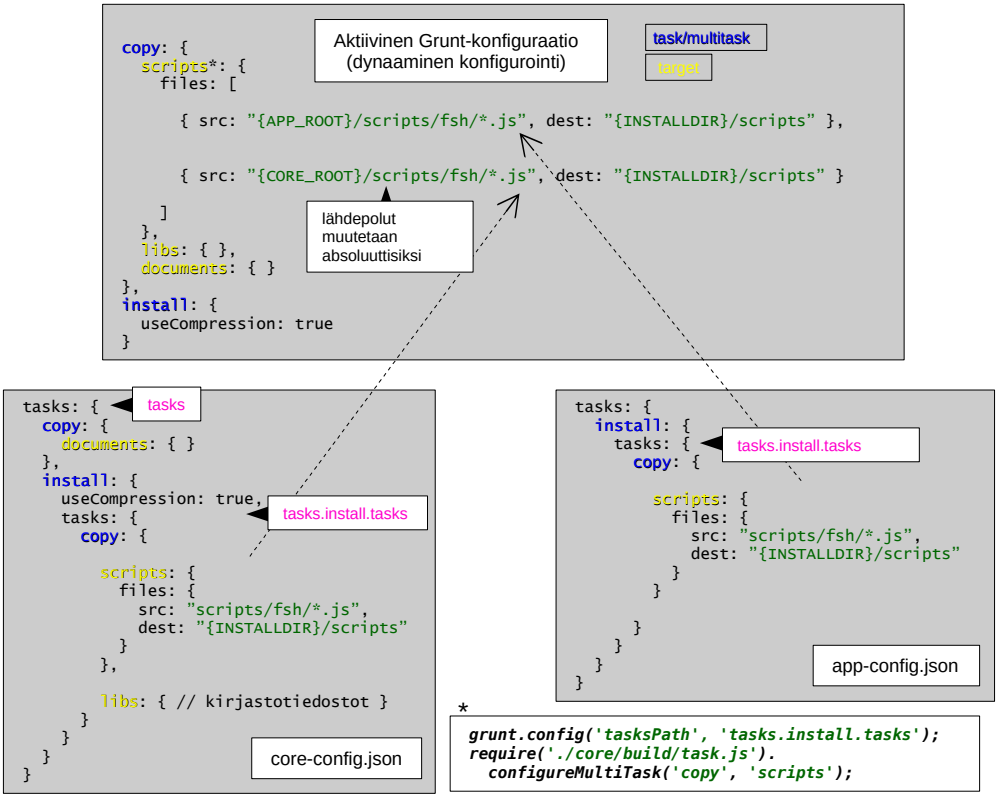
– **language_generate**: luo sovelluksen käyttämät kieliresurssit automaattisesti.

– **clean**: poistaa tiedostoja.

Kaikista tehtävistä install on selkeästi suurin, sisältäen monia alitehtäviä: tiedostojen kopiointia, yhdistämistä, minimointia ja mahdollisesti myös tiivistämistä.

22 Apuskriptit on suunniteltu tuotekehityksen aikaiseen käyttöön ja käyttäjäkohtaisesti muokattaviksi.

Useimmat buildin tehtävät tehtiin Grunt-liitännäisten avulla. Koska liitännäisten tarjoamia tehtäviä haluttiin voida suorittaa myös itsenäisesti, laajennettiin Gruntia *konfigurointikontekstin* käsitteellä. Konteksti tarkoittaa sitä JavaScript-objektipolkua, josta dynaamisesti konfiguroitavan tehtävän konfiguraatiomäärityksiä etsitään sovelluksen ja Coren konfiguraatiotiedostoista. Kuvassa 11 on havainnollistettu kontekstia install-tehtävän yhteydessä. Kuvassa on myös esitetty, miten dynaaminen konfigurointi toimii.



Kuva 11. Käännösohjelman dynaaminen konfigurointi.
Build etsii asetuksia kulloinkin asetetun kontekstipolun alta (oletuksena 'tasks'). Kuvassa install-tehtävä asettaa kontekstiksi *tasks.install.tasks* ja kutsuu buildin konfigurointifunktiota. Coren ja sovelluksen asetuksiin on määritelty tässä kontekstissa tehtävä *copy.scripts*. Löydetyt asetukset yhdistetään Grunt-konfiguraatioksi.

Build erityishuomioita

Käännösympäristön polusta odotetaan löytyvän Node-ohjelma, mutta muita vaatimuksia tai riippuvuuksia ei ole[23]. Sovelluksen kääntämiseen ei tarvita edes verkkoyhteyttä, koska kaikki Grunt ja Node -moduuliriippuvuudet ovat Coressa valmiiksi mukana. Koska moduuleita on paljon ja npm-ekosysteemi on melko

23 Tuotekehityksen aikana vaatimuksia on hieman enemmän. Esimerkiksi sprite-ikoni luodaan ImageMagick-ohjelmalla.

välinpitämätön levytilan käytön suhteen, on Core-komponentin tavuissa mitattu koko kasvanut huomattavasti Luminatoon verrattuna. Vaikka buildin suhteellinen osuus on suuri, siitä ei ole kuitenkaan ollut merkittäviä haittavaikutuksia. Tosin Windowsissa ongelmia on aiheuttanut npm:n piirre luoda syviä moduulien hakemistorakenteita: Windows rajoittaa polkujen pituuden 260 merkkiin tietyissä tilanteissa.

Buildin dynaamisella konfiguroinnilla haettiin suorituskykyä. Build ei kuitenkaan hitaimmillaankaan ole pitkäkestoinen prosessi, joten jälkikäteen voi kyseenalaistaa dynaamisuudella saavutetun hyödyn ylläpidon ja muokattavuuden kustannuksella. Suorituskykyä on tarvittaessa helpompi ja tehokkaampi parantaa vähentämällä suoritettavia operaatioita. Build toimiikin myös näin. Useimmat tehtävät ajetaan grunt-newer liitännäisen kautta, mikä toimii varsin hyvin suurimmassa osassa tapauksista.

Kaksi buildin pitkäkestoisinta tehtävää ovat asennuksen yhteydessä suoritettava JavaScript-lähdekoodien minimointi, joka tehdään aina kun yksikin lähdekooditiedosto on muuttunut, sekä asennushakemiston ja sen sisällön siirto kohdelaitteeseen. Jälkimmäisen tehtävän kesto riippuu laitteen verkkokapasiteetista ja sen suorittimen kyvystä purkaa tiivistetty arkistotiedosto laitteen tiedostojärjestelmään. Julkaisua on optimoitu siten, että vain muuttuneet asennustiedostot siirretään laitteelle. Nopeutta voisi edelleen parantaa esimerkiksi rsync-protokollalla, mutta sen tukeminen pelkästään testauksen aikaisen julkaisun tehostamiseksi ei ole kannattavaa.

Grunt-ohjelmaa ei ole suunniteltu toimivaksi kahdessa rinnakkaisessa hakemistossa samanaikaisesti. Konfigurointiasetusten yhdistäminen ei tästä syystä ollut triviaalia toteuttaa, sillä käytännössä kaikki asetusten lähdetiedostojen polut täytyy esikäsittelyssä muuttaa absoluuttisiksi. Lisähaastetta tähän aiheutti Gruntin lähdetiedostojen määrittelytapojen monimuotoisuus erinäisten optioiden muodossa. Kaikkia optioyhdistelmiä build ei edes tue, ja asetusten määritysmuoto poikkeaa muutenkin dokumentoidusta Gruntista. Tämä ei ole kovin ideaalinen tilanne, mutta nykyisessä muodossaan tilannetta ei voi helposti muuttaa ilman huomattavaa työpanosta.

3.4.4 Luminaton ja Coren erottaminen erillisiksi projekteiksi

Coren lähdekoodien siirto omaan versionhallinnan alaiseen komponenttiin tehtiin varsin nopeasti, eikä työvaiheessa esiintynyt erityisiä ongelmia. Lähdekoodien osalta riitti lähestulkoon yksittäisen hakemiston siirto Luminatosta. Uutta käännösohjelmaa varten molempiin komponentteihin piti kuitenkin luoda uudet konfiguraatiotiedostot.

Luminaton lähdekoodia ei tarvinnut tässä vaiheessa refaktoroida, koska viittaukset Coren nimiavaruuksiin oli muutettu jo ensimmäisessä erotusvaiheessa. Luminaton Makefile päivitettiin delegoimaan käännös- ja asennuskomennot uudelle käännösohjelmalle.

Tuote käännettiin lopuksi uudessa ympäristössä ja todettiin toimivaksi. Luminatosta oli tullut Core-sovellus.

3.4.5 Dokumentointi

Erotusprosessin aikana lähdekoodien dokumentointia tehtiin lähinnä JSDoc-kommenttien muodossa, joista niistäkin valtaosa liittyi metodien ja luokkien määrittelyihin. Tarkempaa selittävää kommentointia ei lisätty joitakin isompia komponentteja lukuun ottamatta. Muussa kuin lähdekoodiin muodossa dokumentaatiota lisättiin yleensäkin vähän tai se jäi keskeneräiseksi, eikä pysynyt ajan tasalla. Käännösohjelmaan lisättiin kuitenkin komento, joka luo erillisten markdown-dokumenttien[24] ja lähdekoodin kommenttien pohjalta yhtenäisen HTML-dokumentti-kokonaisuuden.

Missään vaiheessa ei siis päästy suunnitelmassa tavoiteltuun dokumentoinnin määrään. Aikatauluarvioissa dokumentoinnille varattiin runsaasti päiviä, mutta muut tehtävät menivät tärkeysjärjestyksessä edelle, eikä tähän työvaiheeseen koskaan päästy. Kokonaisuudessaan Coren dokumentaatio oli projektin lopussa hyvin keskeneräinen, eikä siitä olisi ollut suurta hyötyä kolmansille osapuolille Core-käyttöliittymien kehitystyössä.

24 Markdown on ihmisen luettavaksi tarkoitettu tekstiformaatti, joka sisältää normaalin tekstin lisäksi merkkausinformaatiota. Markdown-tiedosto käännetään HTML-dokumentiksi.

3.4.6 Esimerkkisovelluksen toteutus sovelluskehyksellä

Erotustyön jälkeen Coren avulla luotiin esimerkkisovellus, jonka roolina oli toimia dokumentaationa ja testiympäristönä. Sovelluksen pohjaksi valittiin Luminaton uusi Corea käyttävä toteutus, josta karsittiin kaikki epäoleellinen pois. Tässä vaiheessa esimerkkisovellus sisälsi minimaalisen määrän toiminnallisuutta: sisäänkirjautuminen, istunnonhallinta ja ohjelman alustus.

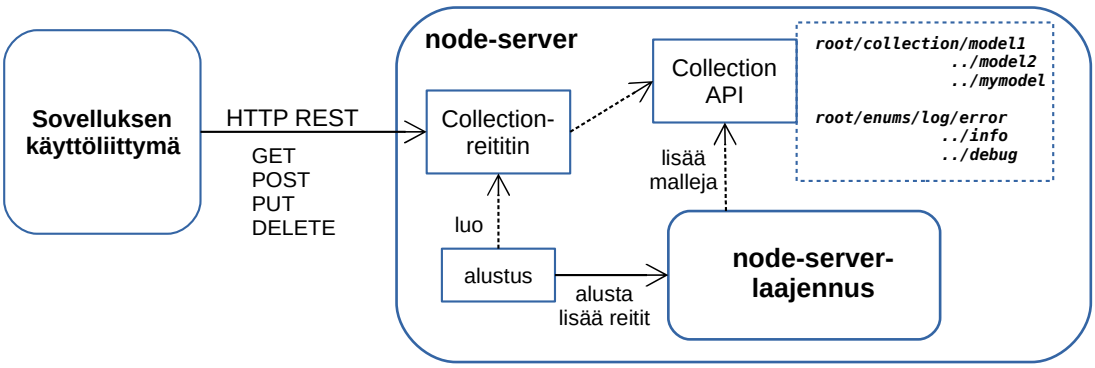
Esimerkkisovelluksen varsinainen sisältö toteutettiin Engine-sovelluksena, koska Engineä haluttiin kehittää muita tuotteita varten, ja esimerkkisovellus antoi tähän hyvät puitteet. Sovellus koostui näin ollen lähes pelkästään niistä Coren ominaisuuksista, joihin Engine tarjosi rajapinnan. Tämä tarkoitti kuitenkin myös sitä, ettei esimerkkisovelluksen avulla voinut kunnolla arvioida Coren soveltuvuutta Luminaton kaltaisen, omia näkymätemplaatteja sisältävän, ja Coren komponentteja suoraan käyttävän sovelluksen luomiseen.

Node-Server

Esimerkkisovelluksen palvelinosuutta varten Coreen tehtiin Express.js-kehyksen päälle testipalvelimen runkototeutus. Testipalvelinta ei kuitenkaan ole pakko käyttää Core-sovelluksissa. Tavoitteena oli, että tuotekohtaisen palvelintoteutuksen rinnalle voi aina tarpeen tullen luoda eräänlaisen palvelinsimulaation, jota vasten käyttöliittymää voi testata. Parhaassa tapauksessa käyttöliittymää voi siis testata jo ennen kuin varsinaista palvelintoteutusta on vielä olemassa. Testipalvelin sai kehitystyön aikana epävirallisen nimen: node-server[25].

25 Parempi nimi olisi ollut test-server, koska se kuvaa palvelimen käyttötarkoituksen ja nimi node-server oli käytössä jo toisessa projektissa.

Koska Coren mallit ja niiden kokoelmat vaativat palvelimelta REST-rajapinnan toimiakseen kitkattomasti, toteutettiin palvelimeen yleinen Collection-API ja Express.js-reititinmoduuli. Yhdessä nämä muodostavat Collection-kokonaisuuden. Kuvassa 12 on esitetty node-serverin rakennetta.



Kuva 12. Testipalvelimen rakenne.
Testipalvelin pystyy käsittelemään tyypilliset tietomallien REST-operaatiot. Collection-reititin käsittelee tulevat pyynnöt ja päivittää tai lukee mallien datan Collection-rajapinnan kautta.

Collection-API koostuu joukosta elementtipuita, joiden solmut tai elementit vastaavat aina tiettyä osapolkua, johon REST-verbit kohdistuvat. Elementin tyypistä riippuu, mitä operaatioita siihen voi tehdä.

Collection-reitittimen voi asettaa käsittelemään REST-pyyntöjä mihin tahansa juuripolkuun, jota vastaavalle elementtipuulle se osaa käsitellä tietomallien REST-operaatiot. Reititin ohjaa toiminnot aina lopulta Collection-rajapinnan tehtäväksi ja palauttaa siltä saamansa tulokset REST-pyynnön vastauksessa.

Collection-kokonaisuus on mahdollista konfiguroida suoritusaikana melko joustavasti erilaisia simuloituja palvelinskenaarioita varten. Lähtevän mallidatan voi esimerkiksi asettaa kääriytymään ylimääräisen JSON-objektin sisälle, ja käänteisesti purkamaan tulevan datan kääreen. Jokaiselle tietomalleja sisältävälle elementille voi myös määrittää sen JavaScript-ominaisuuden nimen, jota vastaava arvo mallin datan joukossa määrittää kyseisen mallin Backbonen edellyttämän tunnisteen tai id:n.

4 LOPPUTULOKSET JA PÄÄTELMÄT

4.1 Valmis sovelluskehys: WebUI Core

Työ vietiin onnistuneesti loppuun suhteellisen nopeassa aikataulussa ja tuloksena saatiin käyttöön web-sovelluskehys: WebUI Core tai lyhyesti Core. Kehyksellä yritys voi luoda MVC- ja SPA-arkkitehtuurimallin käyttöliittymiä eri tuotteisiin. Ensisijaisia käyttökohteita ovat sulautetut järjestelmät, mutta kehys soveltuu käytettäväksi myös muissa ohjelmistotuotteissa. Työn perustana käytettiin olemassa olevaa Luminato-laitteen käyttöliittymätoteutusta, josta siirrettiin yleiskäyttöiset osat soveltuvin osin Coreen.

Core on puhtaasti asiakaspään kehys: se ei itsessään sisällä palvelinpuolen toteutusta. Kehyksen käyttöönotto edellyttää kuitenkin, että laitteeseen toteutetaan Backbone.js-malleja tukeva REST-synkronointirajapinta. Sulautettujen järjestelmien kannalta ratkaisu on joustava, vaikka rajapinta itsessään onkin rajoittava tekijä.

4.2 Coren soveltaminen: kaksi esimerkkitapausta

Web-sovelluskehys on hyödytön, jos sillä ei luoda käyttöliittymiä. Coren erottaminen Luminatosta onnistui sujuvasti ja molemmat komponentit alkoivat elää omaa elämäänsä, mutta tämä ei vielä kerro paljon kehyksen soveltuvuudesta.

Corea testattiin jo työn puitteissa toteuttamalla pienimuotoinen esimerkkisovellus. Sovelluksesta muodostui kuitenkin lähinnä Engine-ominaisuuden esittely- ja kokeilualusta. Näin ollen esimerkkisovelluksen ei voi katsoa olevan erityisen arvokas analyysin kohde kehyksen kokonaisvaltaista soveltuvuutta arvioitaessa. Esimerkki-sovellus oli kuitenkin varsin nopea saattaa alkuun, ja tältä osin kehyksen vaatimuksien voi katsoa toteutuneen.

Coren kehitystyötä on jatkettu erotustyön valmistumisen jälkeen. Koska työn kirjoitusprosessi on vienyt oman aikansa, on nyt mahdollista retrospektiivisesti tarkastella kehyksen käyttöönottoa kahdessa tuotteessa, joihin on toteutettu asiakaskäyttöön julkaistu käyttöliittymä. Tuotteiden yksityiskohtiin seuraavassa ei kuitenkaan pureuduta syvemmin.

EMS-hallinnointisovellus

Coren ensimmäisenä kohteena toimi IP-verkkolaitteiden monitorointiin ja etähallintaan erikoistunut EMS-ohjelmisto (Element Management System), johon luotiin admin-käyttöliittymä. Käyttöliittymä toteutettiin Engine-sovelluksena, joten oli varsin luonnollinen ratkaisu käyttää esimerkkisovellusta työn pohjana. Näin ollen myös admin-käyttöliittymä toimi Enginen kehitystyön testipenkkinä.

Työajasta valtaosan voi arvioida kuluneen Coren kehitystyöhön. Tämä ei sinällään yllätä, kun ottaa huomioon erotusprosessin tavoitteet ja päätöksen jättää suuret arkkitehtuurimuutokset tekemättä. Harmillista tässä on kuitenkin se, että Coren kehitystyöhön käytetystä ajasta suurin osa kului Coren synnyttämien hämäräperäisten ongelmien ratkomiseen. Ongelmien syyt yleensä löydettiin, mutta niihin tehdyt korjaukset olivat luonteeltaan enemmän paikallisia kuin koko arkkitehtuurin läpi vaikuttavia muutoksia, joilla olisi estetty ongelmien uusiutuminen.

Toinen ongelma liittyi synkronointirajapinnan yhteensovittamiseen EMS-palvelimen kanssa. Palvelimen rajapintatoteutusta tästä ei voi kuitenkaan syyttää, koska palvelimen kehitystyö oli aloitettu ennen kuin Coren erotustyö valmistui. Ongelma olikin tässä tapauksessa Coren joustamattomuus: tuki hierarkkisille REST-pyyntöpoluille ja pyyntöjen parametreille puuttui lähes kokonaan. Koska EMS-palvelimessa oli käytetty molempia, ongelmia jouduttiin kiertämään mitä erikoisemmin keinoin, mihin kului paljon aikaa.

DAH monitorointi- ja hallinnointisovellus

Tuoreempi Coren sovelluskohde oli minimaalinen kaapelimodeemien päätelaite tai mini-CMTS – DAH (DOCSIS Access Hub). Sen käyttöliittymään haluttiin näkymät laitteen ja modeemien tilojen tarkastelua varten sekä hallintapaneelit laitteen asetusten muuttamiseen. Kuten EMS:ssä, tuotteen kehitystyö oli edennyt varsin pitkälle, ennen kuin siihen päätettiin tehdä web-käyttöliittymä. Myös tässä tapauksessa käyttöliittymä tehtiin Engine-sovelluksena.

Käyttöliittymän kehitystyö eteni varsin nopeasti. Tässä auttoi osaltaan se, että Engine alkoi olla tässä vaiheessa ominaisuuksiensa puolesta varsin kattava. Toinen kehitystä nopeuttava tekijä oli, että DAH oli laitepuolen arkkitehtuuriltaan hyvin paljon samankaltainen kuin Luminato – jopa niin paljon, että järjestelmätason koodia oli voitu

kopioida lähes sellaisenaan Luminatosta. DAH ei kuitenkaan tarjoa käyttöliittymälle DXI-synkronointirajapintaa.

Käyttöliittymän tietomallien synkronoinnin kannalta edullinen seikka oli, että laitteeseen oli jo toteutettu kattava JSON-formaattia tukeva REST-rajapinta. Rajapinta oli kuitenkin tehty lähinnä laitteen komentorivisovellusta varten ja myös tässä tapauksessa yhteensovitus tuotti ongelmia. Suurimpana ongelmana oli rajapinnan epäjohdonmukaiset dataformaatit, joihin käyttöliittymää ei ollut aina suoraviivaista mukauttaa. Esimerkiksi JSON-formaatin osalta useimmat kutsut palauttivat ja ymmärsivät pelkästään merkkijono-tyyppisiä arvoja, vaikka JSON tukee myös muita tyyppejä.

DAH tukee maksimissaan 500:aa laitteeseen rekisteröitynyttä kaapelimodeemia. Coren suorituskyvyssä havaittiin kuitenkin ongelmia tällä datamäärällä. Jo 100 modeemia riitti aiheuttamaan hidastumista modeemien listanäkymässä, joka oli käyttöliittymää eniten kuormittava osa. Käyttökokemuksen kannalta tämä oli suuri ongelma. Optimoimalla tilannetta saatiin parannettua, mutta siitä huolimatta suorituskyky jäi toivottua huonommaksi.

Kehitystyön aikana Engineen tehtyjen muutosten määrä väheni merkittävästi. Voikin sanoa, että DAH:n kehitystyö vei Enginen tilaan, jossa toiminnallisuutta alkoi olemaan kattavasti. On ironista, että samalla kun käyttöliittymän ominaisuuksien määrä kasvoi, kasvoi myös ymmärrys, että päätös käyttöliittymän toteuttamisesta Enginellä oli ollut väärä. Käyttöliittymä alkoi nimittäin kasvaa liian suureksi Enginen alkuperäiseen ideaan nähden. Toteutuksen siirtäminen kokonaan isäntäsovellukseen olisi jatkokehityksen kannalta ollut järkevää, mutta aikaa tähän työhön olisi kulunut liian paljon.

4.3 Coren soveltuvuus käyttöliittymien kehitystyöhön

Esimerkkitapausten perusteella on jokseenkin vaikea arvioida Coren kokonaisvaltaista soveltuvuutta käyttöliittymien kehitystyöhön, koska kaikki esimerkit toteutettiin Engine-sovelluksina. Jotain viitteitä kehyksen mahdollisista ongelmista ja eduista on kuitenkin mahdollista saada, koska Engine on pohjimmiltaan Coren abstraktoiva rajapintakerros.

Esimerkkitapausten perusteella voi jo sanoa, ettei Core ole täysin valmis kokonaisuus, erityisesti arvioitaessa käyttöliittymien kehitystyön nopeutta. Suurimpana ongelmana

oli, että aikaa kului liikaa kehyksen synnyttämien ongelmien selvittelyyn ja korjaamiseen. Toisena ongelmana oli synkronointirajapinta, josta puuttui joustavuutta. Kolmantena kehyksen suorituskyvyssä oli puutteita – jo suhteellisen pienet datamäärät aiheuttivat selvää hidastumista.

Koska käyttöliittymätoteutuksia täytyi viedä aikataulussa eteenpäin, ei aikaa yleensä jäänyt ongelmakohtien korjaamiseen muuten kuin pintatasolla. Koska missään vaiheessa ei tietoisesti varattu aikaa ongelmakohtien määrätietoiseen korjaamiseen, seurauksena oli teknisen velan jatkuva lisääntyminen Coren arkkitehtuurissa.

Ongelma 1: kompleksisuus

Ongelmien ytimessä voi nähdä olevan Coren sisäisen rakenteen kompleksisuuden ja sitä myöten muutostenhallinnan vaikeus. Muutosten tekeminen erityisesti mallien, näkymien, syötteiden validoinnin, ja synkronoinnin osalta ovat työläitä, koska niiden keskinäiset riippuvuudet ovat erittäin monimutkaisia. Komponenttien välillä syntyy suoritusaikana usein syviä ja moniin suuntiin polveilevia kutsuketjuja, joiden seuraaminen on työlästä. Kokonaisuuden hahmottaminen on, jos ei mahdotonta, vähintäänkin erittäin haasteellista. Näin ollen tarpeellistakaan muutostyötä ei tule hevin aloittaneeksi ja kiusaus ongelmien kiertämiseen muilla tavoin on suuri. Toisaalta uusien ominaisuuksien ja tyyppien lisääminen ei sinällään ole vaikeaa ja toisinaan jopa hyvinkin suoraviivaista, koska Coren komponenttien rajapinnat ovat pääsääntöisesti hyvin tarkoituksenmukaisia.

Ydinongelmien ensisijaisena lähteenä on Coren riippuvuus Luminatosta. Riippuvuus on toisaalta luonnollinen, koska Luminato on suurin kehykseen sidottu sovellus. Toisaalta on muistettava, että Core perustuu Luminaton arkkitehtuuriin.

Toisena ongelmien lähteenä voi nähdä kehykseen luodut ominaisuudet uusia käyttöliittymiä varten. Toisinaan uusia ominaisuuksia luotiin, vaikkei näille olisikaan ollut vielä varsinaista käyttöä. Uudet ominaisuudet synnyttivät osaltaan lisää kompleksisuutta, koska vanhoihin ongelmallisiin komponentteihin luotiin lisää riippuvuuksia. Tällainen ominaisuuksien ujuttautuminen (engl. feature creep) on hyvin tunnettu ongelma sovelluskehityksessä.

Ongelma 2: synkronointirajapinta

Kummassakin esimerkkitapauksessa ongelmia esiintyi synkronointirajapinnan kanssa ja molemmissa tapauksissa ongelman todettiin olevan osittain tai kokonaan Coren joustamattomuudessa.

Esimerkkien perusteella voidaan todeta, että jos päätös käyttöliittymätoteutuksesta tehdään tuotteen kehitystyön myöhemmässä vaiheessa, on tärkeää, että kehyksestä löytyy riittävästi joustavuutta mukautua olemassa olevaan toteutukseen. Uusien tuotteiden tapauksessa asiaa voi miettiä myös käänteiseen suuntaan. Jos nimittäin tavoitteena on web-sovelluskehyksen laajamittainen hyödyntäminen, on ehdottoman tärkeää, että kehyksen rajapinnan mahdollisuuksista ja rajoituksista on olemassa selkeä dokumentaatio. Tärkeää on myös, että dokumentaatio on helposti löydettävissä ja keskeisten sidosryhmien tiedossa. Vain näin on mahdollista tehdä päätöksiä tuotteiden käyttöliittymien toteutuksesta mahdollisimman varhaisessa vaiheessa. Coren tapauksessa tällaista dokumentaatiota ei ollut, mutta tällä ei sinällään ollut merkitystä esimerkkitapausten kohdalla, koska Core valmistui vasta, kun molempien tuotteiden kehitystyö oli edennyt jo pitkälle.

Ongelma 3: suorituskyky

Ensimmäisen esimerkkitapauksen kohdalla Coren suorituskyvyssä ei havaittu erityisiä ongelmia. Tämän selittää osittain se, että EMS-käyttöliittymä oli varsin pienimuotoinen ja datamäärät vähäisiä. Toisen esimerkin tapauksessa oli kuitenkin selkeitä ongelmia, joiden syyksi löydettiin profiloinnin jälkeen listamuotoisten näkymien DOM-päivitysten hitaus.

Corea ei ole optimoitu suuria datamääriä varten. Yksittäisen mallin näkymän lisäys ja poisto on suhteellisen hidas operaatio, koska se voi synnyttää pitkän synkronisen kutsuketjun, jonka aikana monet osallistujat tekevät kyselyitä tai päivityksiä DOM-puuhun. Ketjujen pituus selittyy osittain sillä, että kutsujen aikana liipaistaan useita Backbone-tapahtumia, joiden kuuntelijat voivat taas synnyttää uusia tapahtumia. Tällaista porrasteista kutsuketjua on vaikea optimoida, koska Tapahtuma-suunnittelumallin idea perustuu sen yleisyyteen. Backbonen Events-rajapinnan toteutus vie myös itsessään osan suoritusajasta.

Mallien näkymien lisäys- ja poisto-operaatiot tehdään aina yksi kerrallaan loppuun asti. Coresta puuttuu siis mallien ja näkymien osalta tuki suurten mallierien optimaaliseen

päivitykseen. Jos mallit ovat kevyitä – niiden attribuuteissa ei ole monimutkaisia riippuvuuksia toisiin attribuutteihin – suorituskyvyn ongelmat eivät välttämättä näy kovin nopeasti. DAH:n tapauksessa mallit olivat kuitenkin erittäin monimutkaisia, mikä korosti ongelmaa.

Suorituskyvyn osalta Coreen tehtiin erinäisiä optimointeja. Varaa lisäoptimointiin on, mutta nykyisen arkkitehtuurin suorituskyvyn käytännöllisen rajan voi arvioida tulevan vastaan nopeasti. Suorituskykyyn ei voi tehdä merkittäviä parannuksia ilman huomattavia arkkitehtuurimuutoksia.

Soveltuvuus

Ongelmista huolimatta Core on suhteellisen toimiva ja monipuolinen kokonaisuus ja tarjoaa erittäin paljon yleistä toiminnallisuutta ja hyödyllisiä ominaisuuksia. Jos laitetoteutus noudattaa kehyksen synkronointirajapinnan vaatimuksia, ja datamäärät ovat suhteellisen pieniä, on Corella varsin suoraviivaista toteuttaa käyttöliittymä tyypilliseen sulautettuun järjestelmään. Jatkokehityksen aikana joitakin ongelmia saatiin korjattua tai lievennettyä, vaikka samalla arkkitehtuurin kompleksisuus edelleen lisääntyi. Suurten muutostöiden aloittaminen ilman riittävää ajallista panostusta ja harkintaa ei ole kuitenkaan järkevää.

4.4 Vaihtoehtoinen toteutustapa sovelluskehykselle

Sovelluskehys olisi voitu vaihtoehtoisesti toteuttaa kokonaan alusta. Kehyksen uudelleensuunnittelu olisi voinut olla edullista, koska Luminaton ongelmalliset osat olisi voitu jättää heti pois ja tehdä uusi toteutus kunnolla. Luminatosta olisi voitu kopioida yleisesti hyviksi havaitut rakenteet, vaikka nekin olisi toteutettu osittain tai kokonaan uudelleen.

Tietomallit on optimaalisinta synkronoida attribuuttikohtaisesti, koska näin minimoidaan siirrettävän datan määrä. Luminaton DXI-pohjainen synkronointi toimii juuri näin. Coressa olisi haluttu vähentää DXI:n osuutta tai poistaa se kokonaan. Projektin alussa ei kuitenkaan tiedetty, tarvittaisiinko DXI:tä jatkossa. Luminatossa se tarvittiin, joten päätöksestä erottaa Core Luminatosta seurasi suoraan, että DXI:n oli oltava myös Coressa. Toteutusanalyysissa todettiin, että Luminaton synkronointitoteutuksen muokkaaminen yleisempään muotoon olisi ollut erittäin vaikeaa johtuen komponentin

voimakkaasta sidoksesta tietomalleihin. Jos sovelluskehys olisi tehty kokonaan alusta, DXI:n pudottaminen pois olisi ollut edullinen ratkaisu, vaikka JSON-mallien synkronointi olisi jouduttukin toteuttamaan osittain uudestaan.

Suurimpana esteenä uudelleentoteutukselle olisi lopulta ollut kehitysresurssien rajallisuus, koska Luminaton ja Coren kehittäminen rinnakkain olisi vaatinut merkittävästi enemmän resursseja. Kuitenkin jo erottamisprosessin pohjalta voi todeta, että resurssien vähäisyyden takia monia tarpeellisiakaan muutoksia ei ehditty tehdä.

Toisena ongelmana lähestymistavassa olisi ollut, ettei erotustyön alkuvaiheessa ollut täysin selvää käsitystä, missä tuotteissa kehystä tultaisiin soveltamaan. Olisi siis ollut vaikea tietää, mitä ominaisuuksia kehykseen olisi pitänyt sisällyttää ja mitä jättää pois. Tehdyn erottamispäätöksen seurauksena työ oli verrattain suoraviivainen tehdä.

4.5 Coren jatkokehitys

Coren ja sen sovellusten kehitystyön aikana kerääntyneen kokemuksen pohjalta olisi varmasti helppo lähteä toteuttamaan kokonaan uutta sovelluskehystä, ainakin jos tavoitteena olisi tehdä laajuudeltaan Corea vastaava kehys. Tällöin olisi ehdottoman tärkeää varmistua siitä, etteivät nykyiset tunnistetut ongelmat uusiudu. Erityisesti siis arkkitehtuurin muunnosten hallintaan, synkronointiin ja suorituskykyyn tulisi panostaa. Toisaalta on odotettavissa, että kehitysresurssien rajallisuus välttämättä pakottaisi tekemään kompromisseja.

5 YHTEENVETO

Tämän työn tavoitteena oli luoda web-sovelluskehys yrityksen sisäiseen käyttöön olemassa olevan käyttöliittymän pohjalta. Työssä tutkittiin myös yleisemmin sovelluskehyksiä ja niiden arkkitehtuurien suunnittelua. Tutkimuksen perusteella muodostui käsitys, että kehyksen luomisprosessi on yleensä haastava hanke. Jotta kehysprojektin voisi ennakoida onnistuvan, tekijöiltä edellytetään perusteellista ymmärrystä sovellusalueesta ja kykyä luoda helposti muutettavia komponentteja. Siinä missä muunneltavuuden hallinnan työkaluja ovat arkkitehtuuri- ja suunnittelumallit, ei sovellusalueen ymmärrykseen käytännössä auta muu kuin kokemus.

Sovelluskehys luotiin erottamalla yleiskäyttöiset osat olemassa olevasta käyttöliittymästä. Erotustyön aikana ei kuitenkaan tehty merkittäviä arkkitehtuurimuutoksia, ja siksi työ valmistui nopeassa aikataulussa. Valmistunutta kehystä sovellettiin luomalla sen avulla muutama käyttöliittymä, joiden käyttökokemus ja ulkonäkö olivat keskenään lähes identtisiä. Käyttöliittymäprojektien aloittaminen oli nopeaa ja uusi käännösohjelma toi niiden kehitystyöhön joustavuutta. Tässä mielessä kehyksen voi siis sanoa onnistuneen. Tutkimuksen aikana syntynyt ennakkokäsitys sai kuitenkin vahvistuksen jatkokehityksen aikana: periytynyt ja myöhemmin otettu tekninen velka selvästi hidasti kehyksen ja sovellusten kehitystyötä.

Työ osoittaa suunnittelun ja muunneltavuuden hallinnan tärkeyden sovelluskehysten kehitystyössä. Jos kehys erotetaan olemassa olevasta tuotteesta, on tärkeää varmistaa, etteivät arkkitehtuurin aiemmat ongelmat periydy. Yhden tuotteen pohjalta ongelmakohtia voi olla kuitenkin vaikea tunnistaa. Vaihtoehtoisesti kehys voidaan tehdä kokonaan alusta, jolloin ongelmat on mahdollista välttää. Tällöin suunnitteluun täytyy kuitenkin käyttää enemmän aikaa. Jos on nähtävissä, ettei kunnolliseen toteutukseen ole resursseja tai aikaa, on syytä harkita uuden kehysprojektin aloittamista hyvin tarkkaan.

Vastaavia töitä voisi olettaa tehtävän paljon, mutta kirjallisia selvityksiä niistä on vaikea löytää. Jos vastaavia töitä tehdäänkin, niitä ei siis julkisteta laajalle yleisölle. Toisaalta monet julkiset sovelluskehykset ovat syntyneet vastaavista lähtökohdista. Kiinnostava tutkimuksen kohde olisikin selvittää keskeiset tekijät, jotka saavat ohjelmistokehittäjät laajalti ottamaan käyttöön uuden kehyksen, jos sellainen julkaistaan.

LÄHTEET

[1] E. Gamma, R. Helm, Ralph Johnson, ja John Vlissides, *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995.

[2] K. Koskimies ja T. Mikkonen, *Ohjelmistoarkkitehtuurit*. Helsinki: Talentum, 2005.

[3] V. Okanović, T. Mateljan, ”Designing a New Web Application Framework”, esitetty tilaisuudessa 2011 Proceedings of the 34th International Convention MIPRO, ss. 1315–1318.

[4] Robert S. Hanmer, *Pattern-Oriented Software Architecture For Dummies*. Hoboken, N.J. : Chichester: Wiley, 2013.

[5] A. Osmani, *Learning JavaScript Design Patterns*. O’Reilly Media, 2015 [Online]. Saatavissa: https://addyosmani.com/resources/essentialjsdesignpatterns/book

[6] Aija Kaakinen, ”Single-Page Application -arkkitehtuurin käyttö verrattuna perinteiseen web-sovellukseen”, Opinnäytetyö, Haaga-Helia ammattikorkeakoulu, 2014.

[7] ”Ajax”, *Wikipedia*. 02-touko-2017 [Online]. Saatavissa: https://en.wikipedia.org/w/index.php?title=Ajax_(programming)&oldid=778326458

[8] E. Vehmanen, ”Skriptikielten metapiirteet”, Pro gradu -seminaariesitelmä, Helsingin yliopisto, 2012.

[9] Teleste, ”Luminato”. [Online]. Saatavissa: https://www.teleste.com/video-headend/products/teleste-luminato. [Viitattu: 28-touko-2017]

[10] UBM Tech, ”2014 Embedded Market Study”. [Online]. Saatavissa: http://cms.edn.com/ContentEETimes/Documents/Embedded.com/MarketStudy/2014-embedded-market-study-then-now-whats-next.pdf. [Viitattu: 28-touko-2017]

[11] *Nginx OSS*. Nginx Inc [Online]. Saatavissa: https://www.nginx.com. [Viitattu: 28-touko-2017]

[12] *Appweb2*. Embedthis [Online]. Saatavissa: https://embedthis.com/appweb/doc-2/product/index.html. [Viitattu: 28-touko-2017]

[13] J. D. Case, M. Fedor, M. L. Schoffstall, ja J. Davin, ”Simple Network Management Protocol (SNMP)”, RFC Editor, RFC1157, touko 1990 [Online]. Saatavissa: https://www.rfc-editor.org/info/rfc1157. [Viitattu: 28-touko-2017]

[14] jQuery Foundation, ”jQuery”. [Online]. Saatavissa: https://jquery.com. [Viitattu: 28-touko-2017]

[15] Jeremy Ashkenas, ”Backbone.js”. [Online]. Saatavissa: http://backbonejs.org. [Viitattu: 28-touko-2017]

[16] Jeremy Ashkenas, ”Underscore.js”. [Online]. Saatavissa: http://underscorejs.org. [Viitattu: 28-touko-2017]

[17] *Node.js*. Node.js Foundation [Online]. Saatavissa: https://nodejs.org

[18] Google, ”V8”, *Chrome V8*. [Online]. Saatavissa: https://developers.google.com/v8. [Viitattu: 28-touko-2017]

[19] *npm*. npm, Inc. [Online]. Saatavissa: https://www.npmjs.com

[20] Ben Alman, *Grunt*. github.com/gruntjs [Online]. Saatavissa: https://gruntjs.com

[21] *Express*. Node.js Foundation [Online]. Saatavissa: http://expressjs.com