TAMPEREEN
AMMATTIKORKEAKOULU

LIFERAY-TEEMAPROJEKTIEN VIRTAVIIVAISTAMINEN

Henri Suominen

Opinnäytetyö
Toukokuu 2017
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

SUOMINEN, HENRI:
Liferay-teemaprojektien virtaviivaistaminen

Opinnäytetyö 38 sivua
Toukokuu 2017

Opinnäytteen tarkoituksena on tutkia, millä ohjelmistotuotannon keinoilla voidaan nopeuttaa ja yhdenmukaistaa Liferay 6.2 CE -portaalin teema-pluginien tuottamista asiakasprojekteihin. Teema-pluginit vastaavat portaaliympäristön ulkoasusta ja ovat tällöin osa miltei jokaista asiakastyönä tehtyä portaalitoimitusta.

Opinnäytetyössä tutkittiin tapoja laajentaa Liferay-portaalin toiminnallisuutta hyödyntäen sen tarjoamia laajennuspisteitä. Jotta teema-plugineille saataisiin yrityksen sisällä yhtenäinen tuoterunko, tutkittiin myös yhdenmukaisten pohjaprojektien tuottamiseen soveltuvia menetelmiä yrityksen uusissa toteutusprojekteissa.

Yhtenä tavoitteena oli tutkia, miten teema-plugineilla voitaisiin hyödyntää komponentteja eri toteutusprojektien välillä siten, että projekteissa usein toistuvia komponentteja voitaisiin yleistää ja käyttää tehokkaasti eri projektien välillä.

Työn tuloksena syntyi prototyyppi Liferay-portaaliin upotettavasta moduulista, johon sisällytettiin teema-plugineille yleisesti hyödyllisiä toiminnallisuuksia. Tämän avulla poistettiin tarve upottaa samat toiminnallisuudet sisältävä Java-koodi kuhunkin teema-pluginiin, mikä oli tapana aikaisemmissa projekteissa. Teema-pluginien rakennukseen sekä uudelleenkäytettävien komponenttien käyttöönottoon kehitettiin sovellus Yeoman-alustalla.

Asiasanat: Liferay, ohjelmistoarkkitehtuuri

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

SUOMINEN, HENRI:
Streamlining Liferay Theme Projects

Bachelor's thesis 38 pages
May 2017

The goal of this thesis was to study different software development methods to achieve a faster and more efficient way to create theme plugins for the Liferay 6.2 CE platform. Theme plugins are mainly responsible for the look and feel of portal pages, and as such are an essential part of most projects.

Different extension points available for extending the functionality of the platform were examined, so that common functionality earlier present in theme plugins themselves, could be unified so that they could be used by all theme plugins deployed in the same portal at runtime. This was achieved by comparing different ways of creating a plugin for the Liferay portal, and creating such a plugin to be used in future projects.

One of the main goals outlined in the project was achieving component reuse between theme plugin projects. To address this, a Yeoman application was created for both creating boilerplate projects for starting new assignments, as well as acting as a library of reusable and/or boilerplate components.

Key words: Liferay, software architecture

SISÄLLYS

LYHENTEET

BEM	Block-Element-Modifier, nimeämiskäytäntö CSS-tyyleissä käytetyille valitsimille.
CSS	Cascading Style Sheets, verkkosivujen visuaalisen tyylin määrittelyyn tarkoitettu täsmäkieli.
DDM	Dynamic Data Model, Liferayn käyttämä malli dynaamisten datarakenteiden hallintaan.
HTML	HyperText Markup Language, merkintäkieli verkkosivujen luontiin.
JSON	JavaScript Object Notation, JavaScriptin käyttämä tekstipohjainen tiedostoformaatti datan välittämistä varten.
NPM	Node Package Manager, Node.js –ympäristön käyttämä paketinhallintajärjestelmä.
POJO	Plain Old Java Object, nimi tavalliselle Java-luokalle. POJO-objekti ei käytä periytymistä ja sisältää usein vain getter- ja setter-metodit.
SCSS	Sassy CSS, syntaksiltaan kehittyneempi laajennus CSS-tyylitiedostojen kirjoittamiseen.
SMACSS	Smart and Modular Architecture for CSS, arkkitehtuurimalli CSS-tyylitiedostojen hallintaan.
WAR	Web Application Repository, tiedostomuoto Java-kielisten web-sovellusten paketointia ja jakelua varten.
XML	Extensible Markup Language, tekstipohjainen tiedostoformaatti rakenteellisen datan välittämistä varten.

1 JOHDANTO

Opinnäytetyön toimeksiannon taustalla ovat toimeksiantajan Visma Consultingin projektit joissa toteutetaan räätälöityjä Liferay-portaalitoimituksia. Lähes jokaisen portaalitoimituksen yhteydessä tuotetaan asiakkaalle myös sivuston ulkoasu. Tämä tehdään toteuttamalla räätälöity teema-plugin Liferay-alustalle. Räätälöidyn teema-pluginin toteutus on yrityksessä melko yleinen, elinkaareltaan lyhytkestoinen projekti, jota tavallisesti seuraa portaalin ylläpito sekä jatkokehitys.

Opinnäytetyön tarkoituksena on kehittää yrityksen sovelluskehitysprosessiin osakokonaisuuksia joiden avulla voidaan lyhentää Liferay-portaalitoimitusten teema-pluginien toteutuksessa ja ylläpidossa käytettyä aikaa. Tyypillisimpiä ongelmia ovat muun muassa teema-pluginien monimutkaisuus, hyväksi todettujen käytäntöjen dokumentaation puute, kehitysprosessiin liittyvät hidasteet kuten teema-pluginin kääntäminen sekä teema-pluginin käyttämien rakenteiden ja pohjien päivittäminen portaalisivulle. Näihin pyritään vaikuttamaan yhdenmukaistamalla toteutusten toimintatapoja, tuottamalla yrityksen sisäinen kirjasto uudelleenkäytettäville käyttöliittymäkomponenteille ja -työkaluille sekä kehittämällä tapoja joilla projektin alkuun saattaminen olisi mahdollisimman sujuvaa.

Opinnäytteessä tutkitaan, millä ohjelmistokehityksen menetelmillä yksittäisten teema-pluginien tuottamista voidaan viedä kohti yhtenäistä kehyssovellusta, ja miten näitä menetelmiä voidaan hyödyntää käytännössä. Tällaisia ovat esimerkiksi arkkitehtuurilliset ratkaisut sekä uudelleenkäytön mahdollistaminen. Syntyville käyttöliittymäkomponenteille luotava kirjasto tehdään yrityksen sisäiseksi resurssiksi ja sen ympärille rakennetaan kevyt rakenninsovellus, joka toimii tuoterunkona teema-plugineille.

Opinnäytetyön tilaaja on Visma Consulting Oy. Visma Consulting tarjoaa konsultointipalveluita niin yksityisen kuin julkisen sektorin asiakkaille. Visma Consulting on syntynyt Proactumin ja Prioriten yhteensulautumisen tuloksena, kun se samalla siirtyi pohjoismaisen Visma-konsernin alaisuuteen vuoden 2015 keväällä.

2 PROJEKTIN TAUSTAT JA SOVELLETTAVA TEORIA

Portaalitoteutusprojektien teema-pluginien alkuun saattamisessa on yleensä toimittu siten, että pluginin pohjaksi otetaan aikaisemmin toteutetun teema-pluginin lähdekoodi. Tällöin tuloksena on lähtökohta uudelle projektille, mutta myös paljon tarpeetonta koodia kuten CSS- ja JavaScript-tiedostoja, jotka ovat usein pitkälle räätälöityjä ja sen vuoksi epäkäytännöllisiä käyttää uudelleen uudessa projektissa.

Tämä voidaan nähdä anti-suunnittelumallina. Ohjelmistokehityksessä anti-suunnittelumalli on ilmiselvä, mutta vääränlainen ratkaisu toistuvaan ongelmaan. Se on vastakohta suunnittelumallille, joka on toimintamalli tai -menetelmä, jolla toistuva ongelma voidaan ratkaista. (Long 2001, 68.) Sekä anti-suunnittelumalleja että suunnittelumalleja on runsaasti, osan ollessa konkreettisia ratkaisuja ja osan ollessa yleisempiä määritelmiä ratkaisun tai ongelman luonteesta. Aikaisemmin mainittu menetelmä, jossa jo käytetty lähdekoodi kopioidaan uuden projektin pohjaksi on niin kutsutun ”leikkaa ja liimaa”-anti-suunnittelumallin mukainen.

Koodin suora kopiointi projektista toiseen on ohjelmakoodin uudelleenkäytön alkeellisimpia muotoja, mutta oikein toteutettuna uudelleenkäyttö nopeuttaa sovelluskehitystä, eikä hidasta sitä. Leikkaa ja liimaa-mallin seurauksena uusien projektien lähdekoodin karsimiseen kuluu aikaa, koska kehittäjä joutuu arvioimaan, mitkä osat alkuperäisestä lähdekoodista olisi syytä säilyttää uudessa projektissa.

Tämä ongelma pyritään ratkaisemaan toteuttamalla scaffolding- eli rakenninsovellus joka toteuttaa teema-plugineille suunniteltavan tuoterunkoarkkitehtuurin. Rakenninsovellus tarjoaa yksinkertaisen käyttöliittymän uuden projektipohjan luomiseen hyväksi todetuilla käytännöillä. Projektipohja on tällöin niin kutsuttua boilerplate-koodia. Boilerplate-koodi voi toimia aihiona projektin aloittamiselle, ja se sisältää usein projektin tiedosto- sekä hakemistorakenteen sekä yleiset projektissa tarvittavat tiedostot. Rakenninsovelluksen käyttö mahdollistaa yhtenäisten käytäntöjen kehittämisen ja vahvistaa niiden noudattamista.

Aikaisempien projektien kopioinnista seuraa myös niihin upotetun Java-koodin turhaa toistoa. Tämä on osoittautunut ongelmalliseksi varsinkin asiakasprojekteissa, jossa yhdelle Liferay-portaalille on asennettu teema-plugineja useampaa asiakasprojektia

varten. Tämän estämiseksi pyritään muokkaamaan teema-pluginien ympärille rakentuvaa arkkitehtuuria siten, että Java-luokat eriytetään erillisiin plugineihin, jolloin niiden toiminnot ovat yhteisesti käytettävissä kullakin teema-pluginilla ilman, että samaa Java-koodia tarvitsee kirjoittaa jokaisen pluginin sisälle.

Yleisimmille teemoissa käytettäville käyttöliittymäkomponenteille luodaan kirjasto, jonka kautta käyttöliittymäkomponentit ovat käytettävissä uusissa projekteissa. Tällöin minimoidaan tarve kirjoittaa ominaisuuksia alusta lähtien uudelleen kuitenkaan kopioimatta niitä aikaisempien projektien jo räätälöidystä lähdekoodista.

2.1 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuurilla pyritään kuvaamaan järjestelmän jakautumista osiin sekä näiden osien keskinäisiä suhteita. Osat voidaan jakaa eri näkökulmiin tilanteen mukaan, jolloin voidaan tarkastella esimerkiksi tiedostorakennetta tai järjestelmän loogista rakennetta. Osien suhteet taas käsittävät usein järjestelmän ajonaikaisen toiminnan. Arkkitehtuurin tarkoituksena on määritellä järjestelmän koostumukselle säännöt ja periaatteet, joiden avulla järjestelmää kehitetään arkkitehtuurin mukaan. (Koskimies & Mikkonen 2005, 18.)

Ohjelmistoarkkitehtuurin näkökulmasta ohjelmisto voidaan jakaa komponentteihin, jotka kommunikoivat keskenään rajapintojen kautta. Tällöin voidaan eriyttää komponenttien huolenaiheet siten, että yksittäisellä komponentilla ei tarvitse olla tarkkaa tietoa sitä ympäröivän ohjelmiston rakenteesta.

Opinnäytteeseen ohjelmistoarkkitehtuurit liittyvät kahdella tapaa: yhtenä tavoitteena on kehittää tulevissa projekteissa hyödynnettävä teema-pluginien tiedostorakennekuvaus, ja toisena tavoitteena on määritellä teema-plugineille yhteisesti käytettävissä oleva Java-koodi omiksi komponenteikseen. Ensin mainitussa voidaan hyödyntää tuoterunkoarkkitehtuurin periaatteita, ja jälkimmäisessä voidaan käyttää arkkitehtuurianalyysiä ja –suunnittelua.

Arkkitehtuurisuunnitelman toteutus alkaa ohjelmiston vaatimusten määrittelystä. Nämä voidaan jaotella laatuvaatimuksiin ja keskeisiin toiminnallisiin vaatimuksiin. Kun vaatimukset on selvitetty, voidaan niiden pohjalta laatia alustava

arkkitehtuurisuunnitelma. Suunnitelman pohjalta voidaan toteuttaa arkkitehtuurista eri versioita, joita iteroidaan kohti lopullista arkkitehtuurivalintaa vertaamalla arkkitehtuurisuunnitelmaa järjestelmän laatuvaatimuksiin. (Koskimies & Mikkonen 2005, 20-21.)

2.1.1 Tuoterunkoarkkitehtuuri

> ”Tuoterunkoarkkitehtuuria voidaan hyödyntää, kun halutaan uudelleenkäyttää ohjelmistokomponentteja. Sen tavoitteena on kehittää tuotekategorialle yhteinen ohjelmistoarkkitehtuuri eli tuoterunkoarkkitehtuuri, ja toteuttaa sitä tukeva ohjelmistoalusta eli tuoterunko. Tuoterunkoarkkitehtuurissa pyritään tällöin ottamaan huomioon lopullisten tuotteiden eroavaisuudet, jolloin tuotekohtaiset ominaisuudet voidaan toteuttaa olemassaolevan alustan päälle.” (Koskimies & Mikkonen 2005, 157.)

Tuoterunkoarkkitehtuurikeskeinen lähestymistapa on oivallinen teema-pluginien kohdalla, sillä yksittäinen teema ja sen sisältö voidaan mieltää kokonaiseksi tuotteeksi, joka tarjotaan asiakkaalle.

Tuoterunkoarkkitehtuurisuunnittelun ensimmäisenä tavoitteena on kartoittaa tuoteperheelle yhteiset piirteet, jotka voidaan sisällyttää tuoterunkoon sellaisenaan. Tärkeää on myös mahdollistaa eriävien ominaisuuksien toteuttaminen ja muokattavuus tuotekehitysvaiheessa. Arkkitehtuurisuunnittelun kannalta tuoterunkoarkkitehtuuri voi olla joko lopullista tuotetta selittävä, sen rakentamista ohjaava tai erilaisten tuotteiden rakentamisen mahdollistava. (Koskimies & Mikkonen 2005, 159, 163.)

Tuoterunkoa voidaan käytännössä suunnitella tutkimalla aikaisempia projekteja, ja tunnistamalla näistä kaikkiin teema-plugineihin mielekkäästi sisällytettävät kokonaisuudet. Tuoterunkoarkkitehtuurisuunnitelman valmistuttua voidaan kehittää varsinainen tuoterunko, minkä jälkeen se on käytettävissä tuoteperheen jäsenillä. Opinnäytteen näkökulmasta lopullinen tuoterunko on kokonaisuus, jonka keskiössä on itse teema-plugin. Sen sisäistä koostumusta pyritään vakiinnuttamaan tutkimalla eri toteutettuja käytäntöjä ja niiden toimivuutta, ja valitsemalla näistä toimivat menetelmät kuvaamaan teema-pluginin rakennetta. Lisäksi sitä tukee laajemman arkkitehtuurisuunnitelman tarjoama kokonaisuus, jossa teema-pluginin vaatimat yhteiset ominaisuudet on yleistetty käytettäväksi Liferayn tarjoamien rajapintojen kautta.

2.1.2 Arkkitehtuurin analysointi

Yleensä arkkitehtuurianalyysi toteutetaan, kun suunnitellaan uutta järjestelmää tai otetaan haltuun vanha, niin kutsuttu legacy-järjestelmä. Opinnäytetyön puitteissa Liferay-portaalin arkkitehtuuria analysoidaan, että voidaan määrittää eri keinoja Liferay-portaalin toiminnallisuuden laajentamiseen ja muokkaamiseen. Näitä keinoja hyödyntämällä pyritään luomaan näkemys siitä, mitkä käyttötapaukset ovat toteutettavissa milläkin Liferay-portaalin tarjoamalla laajennuspisteellä.

Arkkitehtuurin laatukriteereihin voidaan laskea mm. ajonaikaisia sekä ei-ajonaikaisia piirteitä. Ajonaikaisiin piirteisiin voidaan nähdä mm. Suorituskyky, tietoturvallisuus ja käytettävyys. Kehitys- ja evoluutioaikaisia laatuominaisuuksia ovat mm. muunneltavuus, siirrettävyys, ylläpidettävyys ja uudelleenkäytettävyys. Ohjelmistoarkkitehtuuri on tapa toteuttaa ohjelmiston laatuvaatimukset. Tällöin arkkitehtuuria voidaan analysoida vasten laatuvaatimuksia. (Koskimies & Mikkonen 2005, 221-222.)

Opinnäytetyössä keskitytään analysoimaan lähinnä Liferay-portaalin muunneltavuutta. Muunneltavuutta voidaan toteuttaa sovelluksiin laajennuspisteillä (engl. extension point), jotka mahdollistavat sovellusrungon toiminnallisuuden lisäämisen ja muunneltavuuden sen määrittelemän rajapinnan keinoin. Tällöin kyseessä on ns. plugin-kehys. (Koskimies & Mikkonen 2005, 199.) Eri laajennuspisteitä ja niiden toimintoja voidaan tutkia muunmuassa Liferayn tuottamasta dokumentaatiosta, siihen liittyvästä kirjallisuudesta ja tukimalla portaalin lähdekoodia.

2.2 Ohjelmakoodin uudelleenkäyttö

Koodin kirjoittaminen uudelleenkäytettäväksi on yleinen tavoite sitä kirjoitettaessa, mutta sen toteutuminen on yleensä epävarmaa. Yleinen muistisääntö on muun muassa DRY (engl. Don’t Repeat Yourself), jossa painotetaan ideaa siitä, että saman koodin kirjoittamista uudelleen kahteen paikkaan tulisi välttää. Tämän sijaan tulisi suosia koodin muokkaamista sellaiseksi, että eri kohdissa esiintyvä samankaltainen koodi voitaisiin yhtenäistää yhdeksi funktioksi.

Koodin uudelleenkäytön motiiveja ovat suunniteltu ja opportunistinen uudelleenkäyttö. Suunniteltu uudelleenkäyttö tarkoittaa etukäteen uudelleenkäytettäväksi rakennettujen komponenttien käyttöä, ja opportunistinen sitä, että aikaisemmista projekteista tunnistetaan osiot, joita voidaan käyttää uudelleen uudessa projektissa (Long 2001, 72). Uudelleenkäytetyn ohjelmakoodin rakenne voi olla joko referoitu tai forkattu (engl. fork). Referoitu uudelleenkäyttö tarkoittaa yleensä ulkoisen kirjaston käyttöä, jolloin projektilla ja sen referoimalla kirjastolla voi olla omat elinkaarensa. Forkattu uudelleenkäyttö perustuu lähdekoodin kopiointiin ulkoisesta kirjastosta. Sitä pidetään usein yhtenä ”leikkaa ja liimaa”-ohjelmoinnin muotona, mutta sen vahvuuksina ovat muun muassa eristetty elinkaari alkuperäisestä lähteestä, minkä etuna ovat muokattavuus, versiohallinnan helpottuminen ja yksinkertaisuus. (Colombo 2011.)

Optimaalisessa tilanteessa uudelleenkäytettäviä komponentteja voidaan abstraktoida tasolle, jolla niiden sisältämiä toimintoja ja ominaisuuksia voidaan soveltaa useampaan samankaltaiseen käyttötapaukseen.

Uudelleenkäytön tarjoavaa järjestelmää tuottaessa kehittäjän tulee ottaa huomioon seuraavat asiat:

- komponenttien löytäminen
- komponenttien ymmärtäminen
- komponenttien muokattavuus
- komponenttien rakenne

(Biggerstaff & Richter 1987).

Komponenttien löytämisellä tarkoitetaan uudelleenkäytettäväksi kelpaavien komponenttien tunnistamista, joko aikaisemmasta ohjelmakoodista tai suunnitellusti. Ymmärtäminen ja muokattavuus liittyvät mahdollisten projektista riippuvien ominaisuuksien tunnistamiseen ja eriyttämiseen komponentissa. Komponenttien rakenteissa tulee ottaa huomioon komponenttien koostumus ja eri komponenttien väliset relaatiot.

Uudelleenkäytettävien komponenttien suunnittelu linkittyy myös vahvasti tuoterunkoarkkitehtuuriin – niiden tulisi koostua noudattaen samoja periaatteita kuin mitkä tuoterungossa on määritelty.

2.3 Paketinhallinta ja projektirakennustyökalut

Projektin riippuvuudella tarkoitetaan kirjastoa, arkistoa tai muuta tiedostojen kokoelmaa, jonka ohjelmistoprojekti tarvitsee missä tahansa vaiheessa elinkaartaan (Lalou 2013, 24). Koska on tehokkaampaa ja usein turvallisempaa käyttää valmiina olevia projekteja jo ratkaistujen osa-alueiden toteuttamiseen, voi suuremmissa projekteissa sisällytettyjen kirjastojen määrä kasvaa ulottuvuuksiin joissa kirjastoja sisältävien tiedostojen päivittäminen ajantasaiseksi manuaalisesti olisi epäkäytännöllistä.

Riippuvuudenhallinnalla tarkoitetaan menetelmää tai työkalua, joka auttaa kehittäjää hallitsemaan projektin riippuvuuksia, niiden asentamista sekä niiden päivittämistä. Tällaisia työkaluja ovat mm. Java-projekteille yleiset Maven ja Gradle, sekä node.js -projekteissa Node Package Manager (npm). Usein projektin riippuvuuksia hallitaan tietynlaisen kuvaustiedoston avulla. Kuvaustiedosto voi olla esimerkiksi XML-muodossa, ja siihen voidaan määritellä projektin riippuvuudet, riippuvuuksien halutut versiot ja mahdolliset toiminnot kääntämisen aikana.

Riippuen siitä, missä kohtaa projektia riippuvuutta tarvitaan, voidaan määritellä kirjaston riippuvuusala (engl. scope). Projekteille tyypillisiä riippuvuusaloja ovat mm. kääntämisvaiheen, suorittamisvaiheen tai testaamisvaiheen riippuvuusalat (Lalou 2013, 50).

Projektiautomaatiolla tarkoitetaan eri automatisoituja prosesseja, joilla suoritetaan eri vaiheita projektin elinkaaressa, kuten kehitysympäristön valmistelua, riippuvuuksien hakemista ja niiden kääntämistä, tai testien suorittamista. Projektiautomaatiolla vältetään esimerkiksi manuaalisesti jokaisen lähdekooditiedoston kääntäminen ja niiden sisällyttäminen paketoituun ohjelmistoon.

Ominaisuuden tai kirjaston ulkoistaminen projektin riippuvuudeksi mahdollistaa sen päivittämisen julkaisemalla siitä uusia versioita. Tähän voidaan päätyä, mikäli kirjastossa on bugeja tai sitä päätetään jatkokehittää. Tällöin uusi versio voidaan luoda uuden paketin versionumeroa kasvattamalla, ja se voidaan haluttaessa päivittää riippuviin projekteihin määrittämällä ainoastaan uusi versionumero. Rakennusautomaatiolla varmistetaan, että käännettävä kokonaisuus voidaan tuottaa helposti, ja että projektin toteuttamiseen vaadittavan lähdekoodin jalanjälki pysyisi mahdollisimman pienenä.

3 MENETELMIEN VALINTA JA KÄYTETYT TEKNIIKAT

Tässä kappaleessa käsitellään Liferay-portaalia ja sen osia, sekä menetelmiä joilla työn eri osiot toteutetaan.

3.1 Liferay-portaali

Liferay-portaali (Liferay Portal) on verkkoportaaliympäristö joka sisältää muun muassa käyttäjähallinnan, sisällönhallinnan, sekä toteuttaa rajapinnan portlet-komponenteille, joilla voidaan laajentaa portaalin toiminnallisuutta. Liferayn CE (Community Edition) – versio on täysin avointa lähdekoodia. Liferay mahdollistaa useamman sivuston ylläpidon yhdellä portaaliasennuksella. Tällöin esimerkiksi eri domaineista, kuten esimerkki1.com ja esimerkki2.com voidaan ohjata eri sivustoille, joiden sisältö, käyttäjäryhmät ja ulkoasu voivat erota toisistaan täysin. Tämän takia on resurssien kannalta tehokkaampaa palvella useampia sivustoja yhdeltä portaaliasennukselta käsin.

3.1.1 Teema-plugin

Portaalin teema käsittää portaalin ulkoasun, näkymien ja sisältömallien toteuttamisen. Toteutus sisältää web-sivujen luontia, sisältäen HTML-, JavaScript-, ja CSS-koodia ja Liferayn sisäisesti käyttämiä template-kieliä kuten Velocity ja Freemarker. Näiden kielten avulla portaalisivuille voidaan luoda dynaamista sisältöä. Lisäksi niitä käytetään, kun määritellään portaalin dynaamisen sisällön rakenteille pohjia.

3.1.2 Liferay-pluginien arkkitehtuurin analysointi

Koska Liferay toteuttaa Java Servlet-spesifikaation, sitä voidaan ajaa millä tahansa palvelimella joka toteuttaa Java Servlet Container-spesifikaation. Tällaisia ovat mm. Tomcat, GlassFish ja Jetty. Näistä yleisin on Apachen julkaisema Tomcat, ja sitä käytetään myös Visma Consultingin tarjoamissa Liferay-portaalitoteutuksissa.

Java Servlet Containerin tehtävänä on hallita siihen asennettujen servletien elinkaarta ja ohjata HTTP-kutsut oikeille servleteille. Liferay hyödyntää tätä toiminnallisuutta siten, että siihen asennettavat pluginit toteutetaan servlet-komponenttina. Servlet- ja Servlet

Container-toteutuksen etuja ovat mm. servlet-komponentin asentaminen ja päivittäminen palvelimelle ilman palvelimen uudelleenkäynnistystä

Liferay määrittelee omat spesifikaationsa sille luotaville plugineille. Eri pluginit luokitellaan eri kategorioihin niiden käyttötarkoituksen mukaisesti. Teema-pluginit liittyvät suoraan portaalin ulkoasuun ja käyttökokemukseen. Hook-pluginit mahdollistavat koukuttamisen (engl. hooking), mikä tarkoittaa portaalin alkuperäisen toiminnallisuuden yliajamista tai räätälöintiä. Hook-pluginin yleisin käyttötarkoitus on yliajaa asetustiedostoja tai portaalin omia Java Server Pages-tiedostoja (Chen, Yuan & Yu 2013, 21). Koukkuja voi tehdä myös muihin plugineihin, kuten teema- tai web-plugineihin. Web-plugin mahdollistaa web-palvelun lisäämisen portaaliin. Käytännössä plugin-tyyppejä erottaa se, miten Liferay käsittelee niitä sisäisesti esimerkiksi asennuksen yhteydessä.

3.2 Teema-pluginin rakennusautomaatio

Teema-pluginin toteutuksessa on aikaisemmin suositeltu käytettäväksi Liferay Plugins SDK:ta, joka sisälsi toiminnallisuuden pluginien kääntämiseen. Plugins SDK mahdollistaa muunmuassa teema-pluginin periyttämisen toisesta teema-pluginista siten, että periytettävän pluginin tiedostot sisällytetään uuteen teema-pluginiin ja perivän pluginin omat tiedostot ylikirjoittavat aikaisemmat, samannimiset tiedostot. Alkuperäisten ja räätälöityjen tiedostojen erottamiseksi räätälöidyt tiedostot sijoitetaan pluginin /_diffs -hakemistoon. Tällöin esimerkiksi pluginin CSS-tiedostot sijoitetaan /_diffs/css –hakemistoon. (Chen ym. 2013, 25.)

Version 6.2 myötä Liferay alkoi kehittää erillistä kokonaisuutta “liferay-maven-plugins” joka mahdollistaa komponenttien kääntämisen Mavenilla. Liferay-maven-plugins sisältää automatisoidut ratkaisut eri Liferay-pluginien rakentamiseen. (Liferay, 2016.) Teema-pluginien tapauksessa Maven-pluginista käytetään goaleja eli päämääriä theme-merge ja build-css.

Theme-merge mahdollistaa teeman periyttämisen toisesta teema-pluginista samaan tapaan kuin liferay plugins sdk:ssa. Erona plugins sdk:hon theme-merge –päämäärä ei vaadi tiedostojen sijoittamista /_diffs/css –hakemistoon, vaan teema-pluginia voidaan

kirjoittaa suoraan webapp-hakemistoon. Build-css –päämäärä kääntää teema-pluginin sisältämät SCSS-tiedostot CSS-tiedostoiksi jotka ovat lopulta selaimen luettavissa.

Liferay 7 julkaisun myötä Liferayn teema-pluginien rakennusautomaatiota alettiin kehittää uudelta pohjalta, jolloin Liferay julkaisi npm-paketit generator-liferay-theme ja liferay-theme-tasks. Tällöin myös koko teema-pluginin rakennusautomaatio vaihdettiin Node.js:n ja npm:n yhdistelmään. (Cavanaugh 2016.) Liferay-theme-tasks tekee käytännössä samat asiat kuin aikaisemmat metodit, eli mahdollistaa teema-pluginin periyttämisen, kääntää SCSS -tiedostot CSS:äksi ja paketoi sen WAR-tiedostoksi. Lisäksi se mahdollistaa tiedostojen automaattisen päivityksen kehityspalvelimelle ilman erillistä WAR-tiedoston uudelleenasennusta. Generator-liferay-theme ja liferay-theme-tasks-pluginit ovat kirjoitushetkellä vielä kehityksensä alkuvaiheilla, joten niistä puuttuu vielä tärkeitä ominaisuuksia. Tämän takia niitä ei otettu käyttöön tuoterungon suunnittelussa, vaan päätettiin käyttää aikaisempaa tapaa kääntää plugin Mavenilla.

Projektien scaffolding-toiminnallisuuden toteuttamiseen vertailtiin myös generator-liferay-theme –pluginia sekä Maven-archetype-projektin käyttöä. Maven-archetype on Maven-projektityyppi, jolla voidaan luoda uudelle projektille tarvittava pohjarakenne. Kuten edellä mainittiin, ei generator-liferay-theme ollut tarpeeksi kypsä käytettäväksi asiakasprojekteissa. Samoin Maven-archetypellä uuden teema-pluginin luominen oli hidasta verrattuna Yeomaniin, ja Yeomanin ominaisuudet ovat kattavammat kuin Mavenin archetype-mallissa. Tämän takia päädyttiin lähteä toteuttamaan scaffolding-toiminnallisuus itse käyttäen Yeoman-sovelluskehystä.

3.3 Modulaarinen CSS

Yksi teema-pluginien pääasiallinen osa ovat tyylitiedostot. Liferay 6.2 sisältää komponentin tyylitiedostojen esikääntämiseen, jolloin tyylitiedostoja voidaan kirjoittaa perinteisen CSS:n lisäksi myös SCSS-syntaksilla.

Sass-kielellä on kaksi käytettävissä olevaa syntaksia. Sass on syntakseista vanhempi, ja se käyttää sisennyksiä koodilohkojen määrittelyyn. SCSS on uudempi syntaksi, joka käyttää CSS:lle ominaisempaa syntaksia, jossa koodilohkot erotellaan kaarisulkeilla. Tällöin jokainen CSS-tiedosto toteuttaa myös SCSS-syntaksin vaatimukset, ja tiedostoja voidaan käyttää tarvittaessa toistensa lomassa ongelmitta. (Sass reference 2016.)

Sassin etuja ovat kontrollilausekkeet, funktiokutsut, periyttäminen ja mixinit, joiden avulla voidaan kirjoittaa kompleksisempia rakenteita vaativia tyylitiedostoja. Erityisesti periyttämisellä ja mixineillä voidaan helpottaa tyylitiedojen modularisointia; useammalle elementille voi määritellä yhden pohjaluokan ominaisuuksia periyttämällä.

3.4 Responsiivisuus

Responsiivisuudella tarkoitetaan sivuston sisällön skaalautumista päätelaitteen kokoon verrattuna. Responsiivisuus on tullut yleiseksi vaatimukseksi mobiililaitteiden yleistyessä. Verkkosivut toteuttavat sekä desktop- että mobiiliversiot siten, että päätelaitteen ominaisuudet havainnoidaan näytön koosta tai selaimen lähettämistä user agent-tiedoista. Päätelaitteen kaventuessa voidaan sijoitella elementtejä uudelleen siten, että ne ovat luettavissa helpommin mobiililaitteella.

Liferay 6.2:ssa responsiivisuus tapahtuu käytännössä päätelaitteen leveyden perusteella. Liferay Portalin käyttämä AlloyUI-kirjasto pohjaa responsiivisuutensa Bootstrap-kirjastoon joka jakaa sivun näkymän riveihin ja sarakkeisiin. Kun näytön koko pienenee, sijoitellaan rivien solut allekkain. Tähän käytetään CSS:n mediakyselyjä, jolla voidaan määrittää arvot päätelaitteen näytön maksimi- ja minimileveydelle, jolloin sääntö on voimassa.

Liferay-portaalin 6.2-version responsiivisen ruudukon toteuttava Bootstrap 2.3.2 on versioltaan vanha ja Bootstrapin uudempiin versioihin verrattuna suhteellisen rajattu. Sen haittapuolena on esimerkiksi konfiguroinnin puute arvoille, joilla tavallinen näkymä muuttuu tablettinäkymäksi, ja tablettinäkymä mobiilinäkymäksi. Nämä voidaan muuttaa konfiguroitaviksi, mutta esimerkiksi portaalin teemasta riippumattomat globaalit tyylitiedostot käyttävät mediakyselyitä, joiden ylikirjoittaminen on hyvin vaivalloista.

Tästä huolimatta on luontevaa päätyä siihen, että tuoterungon perusta rakennetaan portaalissa käytettävän Bootstrapin 2.3.2-version päälle. Mikäli räätälöityjä responsiivisia ratkaisuja tarvitaan, voidaan ne toteuttaa erikseen Bootstrapista riippumatta.

3.5 Liferay-portaalin sisällönhallinnalliset komponentit

Tässä kappaleessa käydään yksityiskohtaisemmin läpi Liferay-portaalin yleisimpiä sisällönhallinnan komponentteja. Vaikka rakenteet ja pohjat eivät liity suoranaisesti teema-plugineihin, ne ovat usein vahvasti sidoksissa portaalin ulkoasuun niiden vaatiman räätälöinnin takia.

3.5.1 Portlet

Portletit ovat sovelluksia, joita voidaan sijoittaa portaalisivulle. Ne mahdollistavat dynaamisen sisällön näyttämisen, sekä niiden elinkaarta voidaan hallinnoida erikseen. Portletien elinkaarta ja niille ohjattua sisältöä hallitsee portlet-container, jonka portaaliympäristö toteuttaa. (O'Reilly 2017.) Liferay-portaalissa dynaamisen sisällön näkyville tuominen tehdään miltei poikkeuksetta portleteilla. Asiakasprojekteissa yleisimmin käytetyt portletit ovat Web-sisällön näyttö ja Asset Publisher. Nämä ovat asiakkaan perustyökalut sisällönluontiin ja sen ylläpitoon.

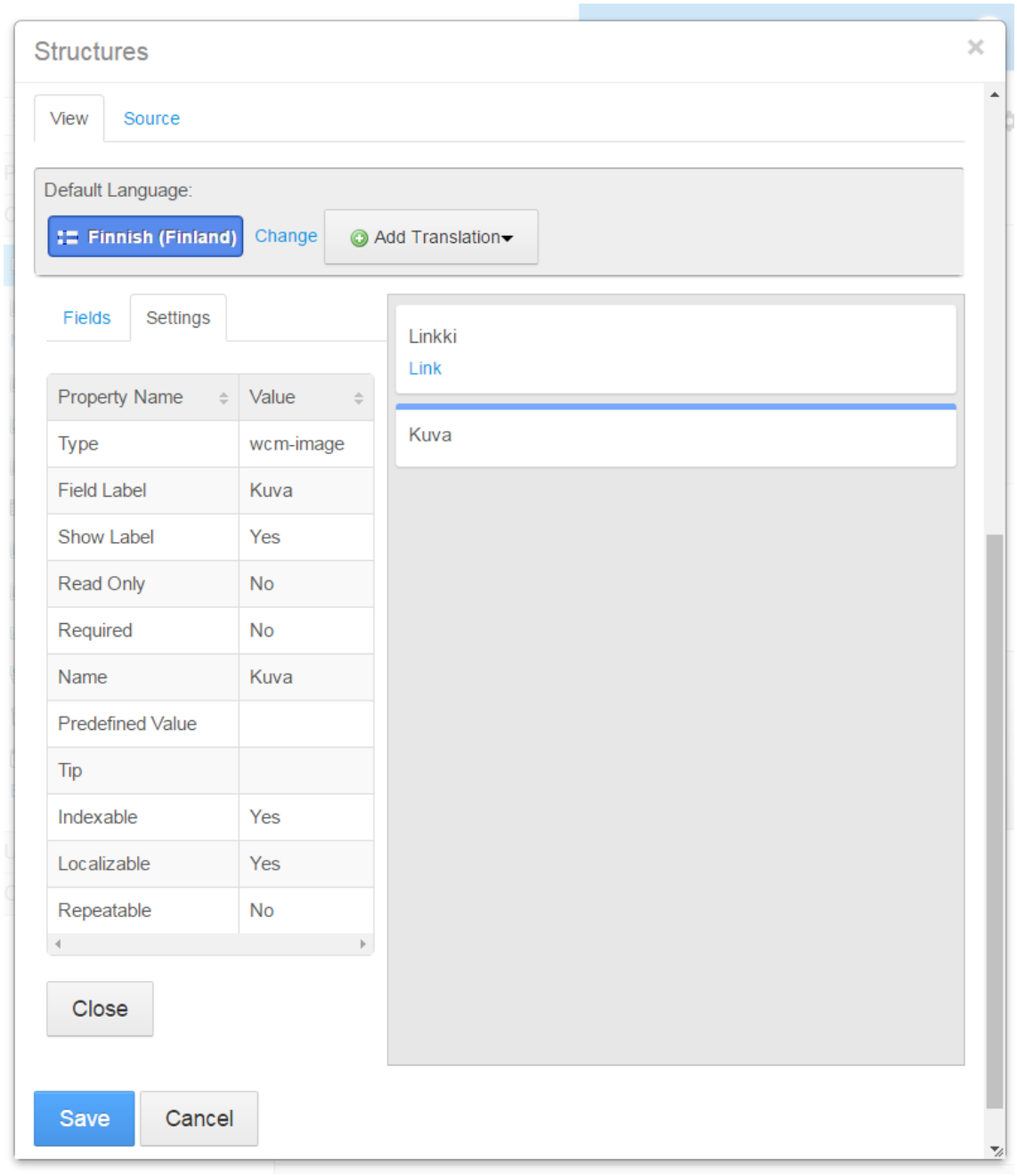
Web-sisällön näyttö-portletilla mahdollistetaan yksittäisen artikkelin näyttäminen sivulla. Web-sisältö pohjautuu aina tiettyyn rakenteeseen. Liferay-portaalin oletusrakenne web-sisällölle sisältää pelkän HTML-kentän. Rakenne voi olla myös käyttäjän määrittelemä. Kehittäjä voi muotoilla minkä tahansa rakenteen esitystä käyttäen pohjatiedostoa.

Asset Publisher on portlet, joka mahdollistaa web-sisältöjen tai dokumenttien näyttämisen sivulla. Sisältö voidaan määritellä näytettäväksi manuaalisesti, tai dynaamisesti tietyin parametrein. Dynaamista sisältöhakua voidaan käyttää muunmuassa viimeisimpien uutisten listaamisessa tai artikkeliin liittyvän sisällön näyttämisessä. Asset Publisher käsittelee web-sisältöä sekä dokumentteja eri Java-luokkakuvauksella kuin Web-sisällön näyttö-portlet, jolloin sen käytettävissä olevat ominaisuudet ovat rajoitetummat. Tämän takia Asset Publisherin näyttämää sisältöä räätälöidään erillisillä Asset Display Template-tiedostoilla.

3.5.2 Rakenne

Rakenteet (engl. structure) määrittävät dynaamisten datamallien (DDM) sisällön siten, että määritellyn datamallin elementit ovat saatavilla rakenteeseen liitetystä pohjasta.

Käytännössä rakenne määrittelee siis muuttujat, joihin sisältöä tuotettaessa voidaan asettaa arvoja. Käytettäviä muuttujatyyppejä ovat esimerkiksi tekstikentät, valintalistat ja HTML-kentät. (Chen ym. 2013, 100.) Rakenteilla mahdollistetaan sisällön räätälöiminen tilanteessa jossa siihen on tarve luoda tavallista tekstiä monimutkaisempia elementtejä, esimerkiksi kuvalinkkejä tai monistettavia kenttiä. Mikäli web-sisällöissä ei käytettäisi rakenteita, joutuisi loppukäyttäjä tuottamaan sisällön suoraan HTML-koodilla, ja tämä olisi vaivalloista. Rakenne määritellään XML-kielellä, mutta tämän lisäksi Liferay tarjoaa myös graafisen käyttöliittymän rakenteiden tuottamista varten (kuva 1).



KUVA 1. Rakenteen luontinäkymä

3.5.3 Pohja

Koko Liferay-portaalin sivun luominen perustuu joko Freemarker -tai Velocity -kielellä toteutettavaan sivupohjaan. Sekä Freemarker että Velocity ovat template-kieliä, jotka toteuttavat yksinkertaisia ohjausrakenteita kuten konditionaalilausekkeet ja silmukat. Kummatkin mahdollistavat dynaamisen sisällön upottamista HTML-koodiin siten, että tiedostoa luettaessa sen kontekstiin sijoitetaan sille määritellyt Java-muuttujat. Sivurakenteen parsiminen aloitetaan tiedostosta portal_normal.vm tai portal_normal.ftl, johon sisällytetään muut tarvittavat templatet. Tähän tiedostoon sisällytetään useimmiten sivun staattiset osiot kuten head-tagi sekä sivuston ylä- ja alatunnisteet (engl. header ja footer). Sivun keskiosa sisältää dynaamisen osion portleteille, johon voidaan erikseen valita ulkoasu. Tämä mahdollistaa loppukäyttäjän sisällöntuotannon vapauden siten, että haluttuja elementtejä kuten artikkeleita ja artikkelinostoja voidaan sijoittaa portleteilla mihin tahansa valitun ulkoasun määrittämään paikkaan.

Templateja käytetään myös rakenteiden näytössä sivulla, kuten esimerkiksi web-sisältöjen näyttöön. Yksinkertainen rakenne voi koostua esimerkiksi kuvasta ja linkistä, kuten aikaisemmin kuvassa 1. Rakenteen sisällöksi tulee tässä tapauksessa kuva ja kuvaa klikatessa avautuva linkki. Pohja on rakenteeseen sidottu template-tiedosto, jolla voidaan määritellä, miten rakenteesta muodostetaan HTML:ää joka lopulta lähetetään selaimelle. Rakennetiedostoon määritellään nimet kullekin sisällön elementille, jolloin niitä voidaan kutsua template-tiedostosta. Kuvalinkki voidaan esimerkiksi määritellä seuraavalla tavalla, jossa kuva on muuttujassa $Kuva, ja linkki on muuttujassa $Linkki (kuva 2).

```
<a class="image-link" href="$Linkki.URL">
    <img src="$Kuva.URL">
</a>
```

KUVA 2. Esimerkki Velocityllä toteutetusta pohjasta, joka sisältää kuvan sekä linkin

3.5.4 Ulkoasu

Ulkoasut (engl. layout-template) ovat Velocity-kielisiä template-tiedostoja, joiden avulla määritellään, miten sivulle asetetut portletit näytetään. Eri ulkoasut määrittelevät pääasiassa portletien rivittymisen tai niiden viemän tilan sivunäkymässä. (Chen ym. 2013, 65.) Liferay 6.2-versiossa layoutien sijoittelu perustuu oletusarvoisesti sen käyttämän Bootstrap 2.3.2:n ruudukkojärjestelmään.

3.6 Käytetyt työkalut

Tässä kappaleessa käydään läpi opinnäytteen toteutuksessa käytettyjä työkaluja.

3.6.1 Intellij IDEA

IntelliJ IDEA on osa tsekkiläisen JetBrains-nimisen yrityksen tuoteperhettä. Se on ohjelmointiympäristö, jonka tarkoituksena on sisällyttää sovelluskehityksessä yleisesti tarvittuja ominaisuuksia yhteen sovellukseen. IDEA on laajennettavissa plugineilla, ja sille on niitä myös saatavissa suuri määrä niin JetBrainsin kuin kehittäjäyhteisön tuottamina. IDEA on pääasiallisena työvälineenä Visma Consultingissa, minkä takia se oli oletettava valinta käytettäväksi sovelluskehityksessä.

Teema-pluginien kehityksessä IDEAn käytännöllisiä puolia ovat mm. Maven-sekä git-integraatiot riippuvuuksienhallintaan ja versionhallintaan, ja sen tuki kaikille projekteissa yleisimmin käytetyille kielille ja syntakseille, kuten HTML, CSS, SCSS ja Velocity (JetBrains 2017). Tuetuissa kielissä ominaisuudet kuten syntaksikorostus, syntaksin tarkistus ja automaattinen täyttö helpottavat kehitystä huomattavasti.

3.6.2 Yeoman

Yeoman on JavaScript-pohjainen avoimen lähdekoodin projekti. Sen toiminta perustuu esimerkiksi Ruby on rails- kehyssovelluksessa käytettyyn scaffolding-periaatteeseen. Sillä voidaan tuottaa erityisiä generaattoreita, joiden avulla uuden projektin perusta voidaan tuottaa interaktiivisesti ja ohjelmallisesti (Yeoman 2017a).

Yeoman-generaattori ajetaan oletuksena komentoriviltä, ja generaattorille voidaan asettaa komentoriviargumentteja tai eri parametreja, jotka käyttäjältä kysytään luomisen yhteydessä (Yeoman 2017b). Komentorivi-interaktioihin Yeoman käyttää sisäisesti kirjastoa Inquirer.js, joka tarjoaa eri kyselytyyppejä kuten lista, monivalinta ja tekstisyöte (Inquirer.js 2017). Nämä toiminnallisuudet luovat selkeän ja laajennettavan järjestelmän uusien teema-pluginien luontiin. Eri kyselytyyppejä voidaan hyödyntää eri tilanteissa, kuten listaa valittaessa asennettava komponentti, tai tekstisyötettä valittaessa teema-pluginin nimi ja versio.

3.6.3 Maven

Maven on työkalu Java-projektien riippuvuuksienhallintaan ja projektien kääntämiseen. Maven mahdollistaa helposti konfiguroitavat kääntämisen- testaamisen- ja ajonaikaisten kirjastojen hallinnan, eri päämäärät (engl. goal), ja kääntöprofiiilit, jolloin sovellus voidaan konfiguroida eri tavalla riippuen siitä, käännetäänkö se kehitys-, testaus- tai tuotantoympäristöön.

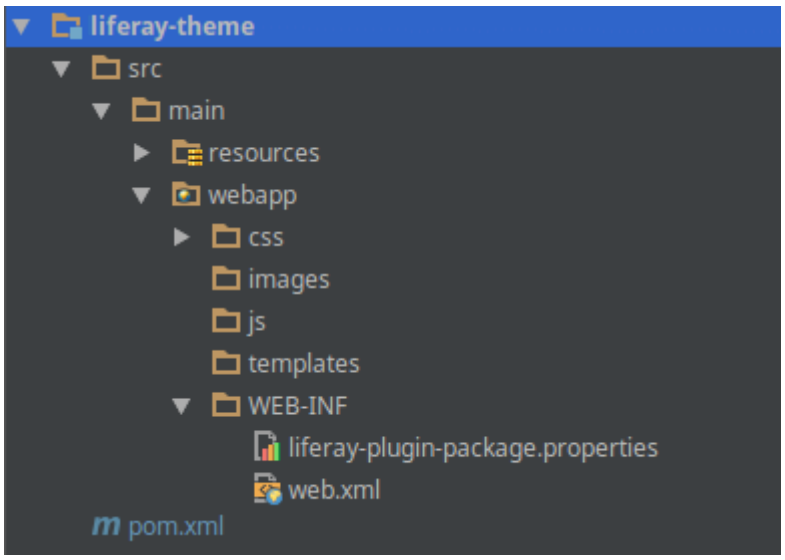
4 TOTEUTUS

Toimeksiantona saatu tehtävä kehittää menetelmiä teema-pluginien kehittämiseen jakautui seuraaviin osakokonaisuuksiin:

- arkkitehtuurimuutosten vaatimusten määrittely ja kuvaus
- teema-pluginien sekä niiden riippuvuuksien eriyttäminen erillisiksi Maven-moduuleiksi arkkitehtuurin vaatimusten mukaisesti
- teema-pluginin tuotearkkitehtuurin määrittely ja scaffolding-applikaation tuottaminen kehityksen käynnistämistä varten
- uudelleenkäytettävien komponenttien kehitys, kerääminen ja muotoilu scaffolding-applikaation kautta käytettäväksi

4.1 Java-arkkitehtuurin muutokset

Esimerkiksi Maven-arkkityypistä luotu teema-plugin sisältää normaalisti vain staattisia tiedostoja, kuten CSS:ää, JavaScriptiä sekä teema-pluginin kuvaustiedostoja (kuva 3).



KUVA 3. Maven-arkkityypistä luodun teema-pluginin rakenne

CSS-, JavaScript- ja template-tiedostot ovat staattisia tiedostoja jotka portaali lataa teeman mukana valitulle sivulle. Lisäksi WEB-INF-hakemistossa on kuvaustiedostoja kuten liferay-plugin-package.properties, joista portaali lukee pluginin asennusvaiheessa projektin metatietoja, kuten esimerkiksi pluginin nimen ja tekijän.

Yrityksen sisäisissä projekteissa teemoihin on usein sisällytetty Java-luokkia, joita portaali kutsuu ajonaikaisesti. Näiden luokkien vastuulla on muunmuassa teema-pluginiin kuuluvien dynaamisten sisältöjen, kuten pohjien ja rakenteiden päivittäminen palvelimelle. Lisäksi osa luokista sisällyttää portaaliin räätälöityjä Velocity-muuttujia, joita voidaan kutsua esimerkiksi portal_normal.vm –tiedostosta.

Ongelmalliseksi muodostuvat tilanteet, jossa yhteen portaaliasennukseen asennetaan useampi teema-plugin, jotka sisällyttävät Velocity-muuttujia samaan nimiavaruuteen, jolloin toinen teema-plugin saattaa ylikirjoittaa toisen pluginin määrittelemät muuttujat. Tämä voidaan kiertää nimeämällä eri teema-pluginien sisällyttämät muuttujat eri nimiavaruuksiin, mutta tämä ratkaisu ei ole kovin järkevä, koska tällöin samat toiminnallisuudet sisällytetään jokaiseen portaalikutsuun useamman kerran.

Samoin tarpeetonta on sisällyttää luokkia, jotka tuovat dynaamista sisältöä (DDM-rakenteita ja pohjia) portaaliympäristöön. Nämä pysyvät projektista toiseen melko muuttumattomana, joten ne ovat yleistettävissä. Ainoana erona on luokka, jolla luodaan expando-kenttiä ohjelmallisesti. Expando on yleinen termi, jota käytetään jonkin objektin ominaisuuksien laajentamiseen. Liferay-portaalin tapauksessa laajennuksen kohteena ovat tietokantatauluihin linkittyvät luokat kuten käyttäjä (User), sivusto (Group) tai sivu (Layout) (Liferay 2017). Tällöin expando-kenttien kautta voidaan luoda näille luokille laajennuksia, jossa esimerkiksi jokaiselle sivulle voidaan määritellä kyllä/ei –arvo sille, näytetäänkö sivulla murupolku vai ei. Expando-kenttien luominen tehtiin aikaisemmin kirjoittamalla luokkaan tarvittava Java-koodi, jolla kentät luotiin portaaliympäristöön.

Aikaisemmin mainittujen ongelmien ratkaisemiseksi voidaan suunnitella uusi arkkitehtuuri jossa pohjien, rakenteiden ja muiden ei-staattisten tiedostojen palvelimelle päivitys tapahtuu erillisen, yhteisen pluginin kautta. Samalla plugin voi suorittaa Velocity-muuttujien sisällyttämisen portaalikutsuihin, jolloin yleisesti muuttumattomat funktiot ja Velocity-makrot ovat teemojen kutsuttavissa. Uuden arkkitehtuurin merkittävimmät toiminnalliset vaatimukset voidaan koostaa seuraavalla tavalla:

1. eri teema-pluginien yhteisesti käyttämät ominaisuudet tulee eriyttää siten, että nämä ovat sijoitettuna vain yhteen lähteeseen
2. teema-pluginin ei tule kutsua itse portaalia tuodakseen sen ympäristöön sisältämänsä rakenteet, pohjat ja räätälöidyt kentät

3. teema-pluginin sisältämät rakenteet, pohjat ja räätälöidyt kentät voidaan tuoda portaaliympäristöön hallitusti vaikuttaen siihen, mille portaali-instanssille ja portaalin sivustolle sisältö tuodaan

Uudessa arkkitehtuurisuunnitelmassa pyritään eriyttämään yleiset komponentit jotta niiden ylläpidettävyys ja päivittäminen helpottuisi. Arkkitehtuuri voidaan toteuttaa tekemällä näistä Java-luokista erillinen plugin, jota yksi tai useampi teema-plugin voi kutsua. Riippuen luokkien toiminnasta on valittava Liferayn kautta käytettävät rajapinnat pluginien väliselle kommunikoinnille.

Portaaliympäristöön Velocity-muuttujia injektoivat luokat toimivat aikaisemmin siten, että asennettavassa teema-pluginissa määriteltiin koukku portal.properties-tiedostoon. Koukku luo takaisinkutsun portaalin tapahtumaan servlet.services.events.pre, jonka avulla voidaan kutsua haluttuja luokkia ennen Java-servletin palvelemia HTTP-pyyntöjä. Kutsuttavien luokkien on toteutettava Action-rajapinta, joka kuvaa ainoastaan metodin run(). Metodin parametreissä olevaan HttpServletRequestiin sisällytetään tässä vaiheessa halutut muuttujat. Liferay-portaalissa arvo haetaan MainServlet-luokassa. MainServlet toteuttaa Servlet-rajapinnan, jolloin joka kutsun yhteydessä kutsutaan luokan service()-metodia. Ennen varsinaista kutsun prosessointia kutsutaan processServicePre()-metodia jokaiselle palvelimen hook-pluginille (kuva 4).

```
protected boolean processServicePre(HttpServletRequest request,
    HttpServletResponse response, long userId)
    throws IOException, ServletException {

    try {
        EventsProcessorUtil.process(
            PropsKeys.SERVLET_SERVICE_EVENTS_PRE,
            PropsValues.SERVLET_SERVICE_EVENTS_PRE, request, response);
    }
    catch (Exception e) { ...
```

KUVA 4. Liferayn MainServlet-luokan metodi processServicePre.

Lähdekoodia analysoidessa voidaan huomata, että processServicePre-metodissa haetaan arvot avaimelle servlet.service.events.pre, ja siinä määritellyt luokat kutsutaan yksitellen läpi. Tällöin toteutettavassa action-luokassa julistetut Velocity-muuttujat ovat saatavilla jokaisessa portaaliin saapuvassa kyselyssä, riippumatta siitä, mikä plugin määrittelee ennen palvelua kutsuttavat toiminnot. Tällöin aikaisempaa toimintoa voidaan suoraan

hyödyntää uudessa arkkitehtuurissa – muuttujat ovat joka tapauksessa saatavissa sivuilla, joilla teema-plugin on käytössä. Tämä tuo myös mukanaan haittapuolia: kutsut toteutetaan myös monessa tapauksessa, joissa itse muuttujia ei tarvita, kuten hakiessa staattisia tiedostoja. Tämän ei pitäisi kuitenkaan haitata, mikäli tehty operaatio ei vaadi paljolti tehoa. Lopullisena hyötynä on kuitenkin se, että kaikkia toiminnallisuuksia ei tarvitse kirjoittaa erikseen kuhunkin teema-pluginiin.

Teemaan sisällytettyjen DDM-rakenteiden ja pohjien tuominen portaaliin oli aiemmin toteutettu lisäämällä takaisinkutsu palvelimen application.startup.events -tapahtumiin. Application.startup.events -tapahtumaa kutsutaan aina palvelimen käynnistyttyä, sekä pluginin asennuksen yhteydessä. Tuotavat elementit on teema-pluginissa määritelty aiemmin yksinkertaisella kansiorakenteella: tiettyyn hakemistoon sijoitetut tiedostot tuodaan jokaisen palvelin-instanssin globaalille sivustolle.

Jotta ohjelmakoodia ei tarvitsisi kutsua teema-pluginista, voidaan sille luoda kuvaustiedosto, jossa voidaan määritellä asennusprosessin yksityiskohdat. Tällöin vastuu sisällön käsittelystä voidaan siirtää erilliselle pluginille, joka hakee tarvittavat tiedot teema-pluginin kuvaustiedostosta. Ollakseen helposti käsiteltävissä Java-ohjelman lähdekoodissa, tulee kuvaustiedoston noudattaa tietynlaista rakenteellista mallia. Tähän päädyttiin käyttämään JSON-formaattia. JSON-tiedoston rakenne voidaan määritellä JSON-skeemalla (kuva 5).

KUVA 5. Osa kuvaustiedoston JSON-skeemasta

JSON-tiedostojen käyttämiseen Java-ympäristön kanssa valittiin jsonschema2pojo-kirjasto, joka mahdollistaa JSON-skeeman muuntamisen yksinkertaisiksi Java-luokiksi, jotka noudattavat POJO –eli Plain Old Java Object-mallia. (kuva 6).

```
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder({
    "webId",
    "sites"
})
public class Instance {

    /**
     *
     * (Required)
     *
     */
    @JsonProperty("webId")
    private String webId;
    @JsonProperty("sites")
    @JsonDeserialize(as = java.util.LinkedHashSet.class)
    private Set<String> sites = null;
    @JsonIgnore
    private Map<String, Object> additionalProperties = new HashMap<~>();

    /**
     *
     * (Required)
     *
     */
    @JsonProperty("webId")
    public String getWebId() {
        return webId;
    }
```

KUVA 6. JSON-skeemasta generoitu Java-koodi

Kun Java-luokat on luotu, voidaan skeemaa noudattava JSON-tiedosto lukea Java-objekteiksi. Tämän seurauksena voidaan ensinnäkin validoida saatu JSON-tiedosto: ohjelma antaa virheilmoituksen, mikäli tiedosto ei ole skeeman mukainen. Toisekseen JSON-tiedostoa voidaan käsitellä kuten tavallista Java-objektihierarkiaa, mikä on yksinkertaisempaa kuin listan läpikäynti JSON-objektin kenttiä lukemalla.

Uudessa arkkitehtuurissa päädyttiin käyttämään Liferay-portaalin sisäistä viestitysjärjestelmää, jossa portaalin tapahtumat voivat viestittää toiminnastaan muille plugineille. Viestintä tapahtuu nimetyn väylän kautta. Tämä ominaisuutta käyttää myös Liferayn osana oleva resources-importer-web –plugin, joten se todettiin hyväksi käytännöksi. Kun uusi plugin asennetaan portaaliympäristöön, se lähettää viestin kanavalla ”liferay/hot_deploy”. Tällöin viestejä kuuntelemaan konfiguroitu luokka ajaa ”onDeploy” –metodin, joka saa parametrikseen portaalin lähettämän viestin. Viestin

parametreissä on muunmuassa asennetun pluginin kontekstin nimi, jonka avulla sen hakemistorakenteesta voidaan lukea sekä kuvaustiedosto että DDM-tiedostot.

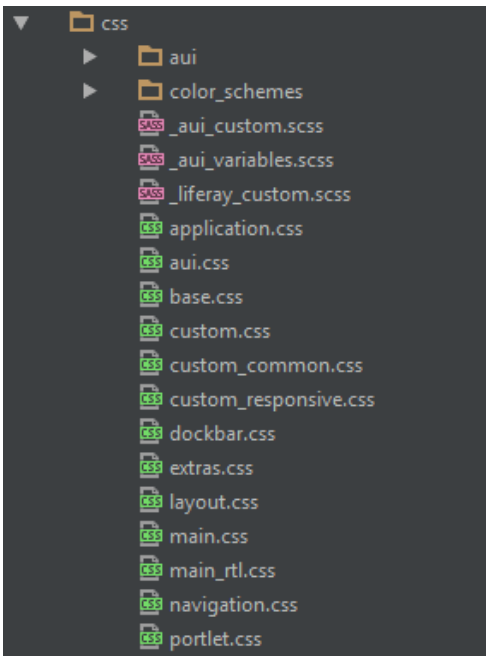
Syntynyt kokonaisuus paketointiin siten, että kaikki ominaisuudet sisällytettiin yhteen JAR-arkistoon, mahdollistaen sen asentaminen portaalin juurihakemistoon. Tämän seurauksena esimerkiksi aikaisemmasta ideasta portaalin ominaisuuksien koukuttamisesta hook-pluginilla jouduttiin luopumaan. Sen sijaan, että portaalin ominaisuuksia yliajetaan koukuttamalla, JAR-arkiston juureen sijoitettiin oma portal-ext.properties-tiedosto, jonka arvot portaali lukee käynnistyessään.

4.2 Teema-pluginin tuoterunkoarkkitehtuuri

Teema-pluginien mallin yhtenäistämiseksi selvitettiin aiemmin hyväksi todettuja käytäntöjä, ja toteutettiin pohjateema, joka sisältää tarvittavan tiedostohierarkian, jonka avulla pluginin rakenne pysyy selkeänä ja helposti hahmotettavana.

4.2.1 Tyylitiedostojen modularisointi

Liferay-portaali mahdollistaa CSS:n kirjoittamisen hallintapaneelissa, mutta siitä huolimatta suurin osa tyylitiedostoista kannattavaa sijoittaa itse teema-pluginiin. Tällöin mahdollistetaan versionhallinnan tehokas käyttö, ja varmistetaan että kaikki tarvittavat tiedostot ovat saatavilla yhdestä paikasta jolloin data ei katoa mikäli esimerkiksi portaalipalvelin joudutaan asentamaan uudestaan tai loppukäyttäjä tai kehittäjä poistaa huomaamattaan CSS-koodia portaalin hallintapaneelin kautta. Pohjana toimivan “styled” -tai “classic” –teema-pluginin tiedostot sisältävät valmiiksi tietynlaisen rakenteen, joka on eroteltu tiedosto kerrallaan. Yleisimpänä lähtökohtana yrityksen projekteille on classic-teema (kuva 7).



Kuva 7. ”classic” teema-pluginin CSS-koodin rakenne.

Liferayn tarjoamien tutoriaalien mukaan käyttäjän tulisi muokata olemassaolevaa teemaa tiedostosta custom.css. Tämä tiedosto kuitenkin sisältää CSS-koodia esimerkiksi “classic” -teema-pluginissa, jolloin siitä periytettäessä alkuperäiset muutokset ylikirjoitetaan. Tästä johtuen on päädytty aiemmin standardisoimaan uusien tyylitiedostojen lisäys siten, että main.css-tiedostoon luodaan import-lauseke tiedostolle “custom_vismaconsulting.css”, jota käytetään itse räätälöidyn CSS:n sisääntulopisteenä. Tällöin voidaan periyttää teemaprojekteja sekä “styled” - että “classic” –teema-plugineista.

Kun kiintopiste räätälöidyille tyylitiedostoille on määritelty, on koodin luettavuuden ja selattavuuden kannalta tärkeää että ne modularisoidaan. Tällä saavutetaan myös koodin uudelleenkäytettävyys käyttäen SCSS:än @include -ja @extend-funktioita. Jotta tavoite portaalitoteutuksien yhdenmukaistamisesta täytetään, voidaan valita kokoelma käytäntöjä, joita voidaan käyttää ohjesääntönä portaaliteemojen toteutuksessa. Tämä koskee käytännössä tyylitiedostojen hakemistorakennetta niiden vastuualueiden mukaan ja CSS-valitsinten nimeämistä.

Tyylitiedostot päätettiin hajauttaa eri hakemistoihin siten, että konkreettiset tyylitiedostot (kuten _header.scss) sijaitsevat ”partials”-nimisessä hakemistossa. Tyylitiedostoista kutsuttavat mixin-tiedostot sijoitetaan hakemistoon ”partials/mixins”, jolloin selkenee

ero räätälöidyn koodin ja Liferayn kääntämisvaiheessa injektoiman ”mixins”-objektin välille.

Osa olemassaolevista CSS-arkkitehtuureista kuten SMACSS perustuvat eri CSS-valitsimien kategorisointiin. SMACSS määrittelee kategoriat kuten base, layout, module ja theme, joilla kullakin on oma vastuualueensa. Tätä ideologiaa hyödyntäen voidaan kategorisoida teema-pluginille tyypilliset valitsimet ja määritellä niille omat hakemistonsa. Esimerkiksi olemassaolevien portletien, kuten web-sisällön julkaisu-portletin tyylejä määrittelevät tiedostot voidaan sijoittaa ”portlet”-alihakemistoon. Tällöin kehittäjä voi esimerkiksi muutoksia tehdessään heti hahmottaa käytetyn luokan kontekstista sen sijainnin pluginin tiedostohierarkiassa. Sovellusnäyttömallikohtaiset tiedostot sijoitetaan puolestaan ”asset”-hakemistoon.

Portletien sekä layoutien tyylien kirjoittaminen on erilaista kuin räätälöityjen komponenttien rakentaminen – Liferay-portaali määrittää hyvin suuren osan portletien HTML-rakenteesta ja CSS-valitsimista. Näiden ulkopuolisten komponenttien valitsinhierarkiaan kehittäjä pystyy kuitenkin usein vaikuttamaan. Aiempien toteutusprojektien pohjalta pystyttiin vertailemaan eri tyylejä kirjoittaa HTML:ää räätälöidyille plugineille, ja tämän pohjalta selkeimmäksi havaittiin niin kutsuttu BEM-malli. BEM, eli Block-Element-Modifier –nimeämiskäytäntö kehottaa käyttämään luokkavalitsimia jakaen HTML-koodi lohkoihin (block), elementteihin (element) ja määritteisiin (modifier), mikä muun muassa estää liiallista valitsinten kirjoittamista sisäkkäin, mikä on ominaista SCSS-koodille. Valitsinten kirjoittaminen sisäkkäin ei ole haitallista, mutta saattaa vaikuttaa koodin luettavuuteen.

4.2.2 Rakenninsovellus

Jotta teema-pluginien alkuun saattaminen olisi mahdollisimman nopeaa, on teemaprojekteille mielekästä olla jonkinlainen pohja, joka sisältää sovitun hakemistorakenteen, projektin tarvittavat Maven-riippuvuudet sekä projektin aloittamiseen tarvittavia tiedostoja.

Tällainen toiminnallisuus voidaan saavuttaa scaffolding –menetelmällä. Kuten aikaisemmin mainittu, voidaan scaffolding-operaatio toteuttaa esimerkiksi Maven-archetype-pluginina. Modernimmat ohjelmistokehykset, kuten Yeoman, tarjoavat

kuitenkin Maven-archetypeä paremmat ominaisuudet, joten se valikoitui toteutusalustaksi.

Käytännössä rakenninsovellus kopioi projektipohjan tiedostot luomaansa hakemistoon, sijoittaa sille annetut parametrit projektipohjassa määritellyille kohdille, ja asentaa tarvittavat npm-riippuvuudet.

Rakenninsovellukseen integroitiin myös toiminnallisuus, jolla voidaan asentaa käyttöliittymäkomponentteja kehityksen alla olevaan teema-pluginiin. Tämä toteutettiin siten, että jokaisella komponentilla on juurihakemisto, johon pluginin sisältämät tiedostot voidaan sijoittaa. Asennuksen lisätoiminnot kuten komponentin vaatimien lisäparametrien kysyminen mahdollistettiin komponentin juurihakemistoon sijoitettavan JSON-konfiguraatiotiedoston avulla.

Yhdeksi ongelmaksi muodostui komponenttien keskenäisen rakenteen muodostuminen. Yksittäinen komponentti saattaa käyttää tiettyä funktiota, ja toinen komponentti saattaa tarvita samaa funktiota. Jotta samaa funktiota ei tarvitsisi kopioida komponentista toiseen, ratkaisuksi mahdollistettiin konfiguroida komponentin juureen asetettavaan JSON-tiedostoon sen omat riippuvuudet.

4.3 Yleistettävien käyttöliittymäkomponenttien toteutus

Uudelleenkäytettävien komponenttien löytämiseksi havainnoitiin aikasemmissa projekteissa toistuvia elementtejä. Tällaisia ovat muunmuassa navigaatiot, murupolut ja kuvakarusellit. Esimerkiksi navigaatio sisältää usein puurakenteessa listan linkkejä, ja kuvakaruselli sisältää listan kuvia, joita liikuttaa JavaScript-plugin. Osa kirjastoon sisällytettävistä komponenteista määriteltiin toimeksiantajan kanssa, ja osa omien havaintojeni pohjalta sen perusteella, minkätyyppiset komponentit ilmenivät usein aiemmin toteutetuissa projekteissa.

Komponenttien tarjoaman kehyksen tarkoituksena on tällöin vähentää tarvetta kopioida toteutuksia aikaisemmista projektista, mikä aiheuttaa usein ylimääräistä ja tarpeetonta koodia. Samalla hyvin ylläpidetty komponenttikirjasto ohjaa kehittäjää käyttämään hyväksi todettuja ratkaisuja ja menetelmiä. Sisällytetty komponentti pyritään pitämään

mahdollisimman yksinkertaisena ja laajennettavana, jotta sen muokkaaminen olisi helppoa.

Rakentimen ratkaisu perustuu ylhäältä alaspäin-menetelmään, jossa komponentit tuodaan projektiympäristöön. Tavallisesti tuotearkkitehtuuri on malliltaan alhaalta ylöspäin, jossa luotu koodi kutsuu funktioita jotka toteuttavat halutun toiminnallisuuden. Yhteen teema-pluginin komponenttiin voi kuitenkin kuulua esimerkiksi sekä rakennetiedosto, pohjatiedosto että tyylitiedostoja, jotka kaikki on toteutettu käyttäen eri kieliä. Tällöin niihin ei voida soveltaa esimerkiksi olio-ohjelmoinnin kaltaista kompositiota, vaan käyttöliittymäkomponentit toteutetaan tarpeeksi yksinkertaisina, perustoiminnallisuuden tarjoavina versioina. Tällöin luodut uudelleenkäytettävät osiot ovat kopioitua uudelleenkäyttöä. Sen etuina ovat muokattavuus ja versiohallinnan yksinkertaisuus, mutta se kärsii päivityksien puuttumisesta. Toisaalta yksi lähde kopioitavalle koodille helpottaa komponenttikirjaston kypsyessä yhä virheettömämpiä lähtökohtia komponenteille.

Navigaatio

Jokaiseen sivustoon kuuluu yleensä navigaatio, jonka avulla käyttäjä voi selata sivuston sivurakennetta. Liferayssä sivurakenne ilmentyy hierarkkisena rakenteena Layout-objekteja, jotka muodostavat navigaatiopuun. Liferay tarjoaa navigaatiorakenteen Velocity-muuttujassa navItems, joka sisältää NavItem-objekteja samanlaisessa puurakenteessa. Navigaatio voidaan tällöin tuoda sivulle käymällä läpi navItems-muuttujan sisältämät elementit silmukassa. Navigaatiotyypeistä valittiin alustavasti toteutettavaksi mm. pudotusvalikko, vertikaalinen avautuva valikko sekä kasattu valikko.

Ohjelmistonäyttömallit

Ohjelmistonäyttömallit (engl. asset display template) ovat Liferayn käyttämiä template-kielisiä tiedostoja, joiden avulla näytetään Asset Publisher-portletin sisältöä. Ne vastaavat tavallisia DDM-pohja-tiedostoja, mutta niihin sidotaan eri muuttujat kuin DDM-pohjiin. Ohjelmistonäyttömalleista uudelleenkäytettäviksi valittiin Rich Asset-komponentti, joka on käytännössä korjattu versio Liferay-dokumentaatiossa esitellystä Rich Asset-komponentista. Tällöin tiedostoa voidaan käyttää boilerplate-koodina uuden ohjelmistonäyttömallin luonnissa – se sisältää makrot miltei kaikkien AssetEntry-tiedoston sisältämän tiedon näyttämiseen, ja kehittäjä voi muokata tarvittavan mallin itse template-tiedostoon.

JavaScript-pluginit

Aikaisempien teema-pluginien JavaScriptillä toteutetuista toiminnallisuuksista yleistettiin osa siten, että ne eristettiin kukin yksittäiseen tiedostoon. Toiminnallisuuksien viemisessä modulaariseen muotoon pyrittiin suosimaan AlloyUI-kehyksen plugin-mallia. Osa jo modularisoiduista komponenteista sisällytettiin kirjastoon sellaisenaan, niiden ollessa esim. jQuery-plugineja, jolloin koodin konvertointi kehykseltä toiselle olisi ollut turhan aikaavievää ja mahdollistanut virheiden ilmenemisen.

4.4 Kehityssyklin nopeuttaminen

Opinnäytteen alkuvaiheessa oli tarkoituksena toteuttaa kehityksen ajonaikainen uudelleenlataus käyttäen jRebel-työkalua. Tämä lähestymistapa oli vakiintunut aikaisemmissa projekteissa, ja se oli osoittautunut pääosin käyttökelpoiseksi.

JRebel mahdollistaa ohjelmakoodin uudelleenkääntämisen ja injektoinnin käynnissä olevaan ohjelmaan (tässä tapauksessa teema-pluginiin) siten, ettei koko ohjelmaa tarvitse kääntää ja käynnistää uudestaan. Tämän tarkoitus on vähentää usein aikaa vievien käännöskertojen määrää kehitysvaiheessa. JRebel toimii Java-koodin ajonaikaisessa uudelleenlatauksessa tehokkaasti, mutta staattisten CSS- ja JavaScript-tiedostojen uudelleenlataamiseen se on raskas. Lisäksi sen kanssa kohdattuja ongelmia olivat muunmuassa teema-pluginin asennus kehityspalvelimelle JRebelin ollessa käytössä, jolloin palvelin ei käynnistä normaalia asennusprosessia tai kutsu portaalin HotDeployListener – tai StartupAction-luokkia. JRebel käynnistetään Liferayn tapauksessa aina portaalin sisältävän tomcat-palvelimen käynnistyksen yhteydessä, joten asennusprosessin käyttämiseksi tai testaamiseksi normaaliin tapaan joudutaan ajamaan palvelin alas ja käynnistämään uudestaan ilman JRebeliä. Tämä voi viedä useampia minuutteja riippuen palvelimelle asennettujen moduulien määrästä.

Kehityssyklin virtaviivaistaminen toteutettiin käyttäen gulp.js-nimistä JavaScript-kirjastoa. Tämä mahdollistaa jRebelin tavoin ajonaikaisen tiedostojen uudelleenlatauksen tiedoston muuttuessa. Gulp on JavaScript-kirjasto, joka erikoistuu rakennusautomaatioon, ja sen avulla on mahdollista tarkkailla tiedostojen muutoksia projektihakemistossa, ja päivittää muuttuneet tiedostot kehityspalvelimelle.

Gulpia ajettaessa sen sisääntulopisteenä on oletusarvoisesti tiedosto ”gulpfile.js”. Tähän tiedostoon voidaan luoda eri tehtäviä, ja kirjoittaa niiden toteutukset. Teeman päivitykseen vaadittavalle gulp-tiedostolle luotiin oma npm-pakettinsa siten, että se on asennettavissa kehitettävään teema-pluginiin. Alkuperäinen tavoite oli tehdä gulp-skriptistä globaali npm-paketti, joka asennettaisiin kerran ja voitaisiin kutsua missä tahansa projektissa ilman erillistä asennusta. Tämä menetelmä vaikeuttaisi kuitenkin versiointia siinä tapauksessa, jossa kaksi eri teemaa vaativat eri versiot gulp-skripteistä. Lopullinen ratkaisu oli sisällyttää gulp-skriptit riippuvuudeksi teema-pluginiin luotavaan package.json-tiedostoon. Silloin kehittäjä voi asentaa määritellyn version komennolla “npm install”. Kun rakenninsovelluksen tuottamien teema-pluginien package.json-tiedostoon määritellään riippuvuus gulp-skriptille, asentaa Yeoman sen automaattisesti operaation loppuvaiheessa.

Gulp-moduuli oli alunperin kopio Emil Obergin tuottamasta MIT-lisenssin alaisesta “Liferay-Instant-Deploy-Theme-Changes-Gulp-Script”-gulp-skriptistä. Kehityksen aikana tähän liitettiin muutamia ominaisuuksia. Alkuperäisessä moduulissa kehitettävän teema-pluginin lähdekoodihakemistot ja palvelimen sijaintihakemisto sijoitettiin suoraan gulpfile.js-tiedostoon. Lisäksi livereload-ominaisuuden käyttö vaati erillisen selainlaajennuksen asentamisen. Livereload on kirjasto, jolla selaimen sivu voidaan ladata uudelleen automaattisesti, yleensä heti sen jälkeen kun tiedostot on päivitetty kehityspalvelimelle. Jotta gulp-skriptin asennus saatiin mahdollisimman yksinkertaiseksi, toteutettiin moduuli siten, että se osaa päätellä kehittäjän senhetkisen hakemistorakenteen perusteella, mitä teema-pluginia uudelleenladataan automaattisesti, sekä se myös injektoi automaattisesti livereload-skriptin jotta erillistä selain-pluginia ei tarvittaisi.

4.5 Selainkohtaiset rajoitukset

Sekä CSS että JavaScript ovat jatkuvasti kehittyviä kieliä, minkä takia eri selaimet toteuttavat ominaisuuksia eri tavalla, tai jättävät osan niistä toteuttamatta. CSS on kirjoitushetkellä versiossa 3, jonka päämääränä on toteuttaa yksittäiset tyylielementit vapaasti implementoitavina kokonaisuuksina ja lisätä niitä jatkuvasti, toisin kuin aikaisemmat CSS1 sekä CSS2-speksit, jotka sisälsivät käytännössä muuttumattomat säännöstöt eri tyyliattribuuteille ja niiden arvoille.

Alati muuttuvan speksin takia useat selaimet toteuttavat osan CSS3-perheen kokeellisista piirteistä käyttäen eri toimittajien etuliitteitä (engl. vendor prefix). Esimerkiksi Webkit-selainmoottorin käyttämä etuliite on -webkit, ja Internet Explorerissa etuliite on usein -ms. Yksittäisen kokeellisen piirteen standardisoiduttua etuliite saattaa poistua kokonaan.

Selaintuen ja toimittajien etuliitteiden perässä pysyminen jatkuvasti kehittyvässä ekosysteemissä on vaivalloista kehittäjälle, mutta ongelmaan löytyy muutamia työkaluja. Teema-plugineja toteuttaessa todettiin hyödylliseksi mm. caniuse.com-sivusto, josta eri CSS-attribuuttien senhetkistä selaintukitilannetta voi seurata. Tällöin voidaan tarkistaa onko käytetty attribuutti tuettu projektin vaatimusten mukaisissa selaimissa, ja voidaanko sitä tällöin käyttää.

4.6 Responsiivisuuden toteutus

Yrityksessä toteutettujen teema-pluginien responsiivisuus voidaan jakaa kolmeen eri lähtökohtaan; Bootstrap-kirjaston oletusarvoisesti toteuttama responsiivisuus, räätälöity responsiivisuus ja selaimen user-agent –attribuuttiin perustuva responsiivisuus.

Bootstrap-kirjaston oletusarvoisesti toteuttaman responsiivisuuden käyttö on projekteissa kaikista yksinkertaisinta: teema-pluginiin sisältyvän Bootstrap-kirjaston arvoja ei tarvitse ylikirjoittaa. Räätälöity responsiivisuus tarkoittaa tässä sitä, että Bootstrap-kirjaston määrittelemät pikseliarvot responsiivisten näkymien vaihtumiselle ylikirjoitetaan. User-agent –attribuuttiin perustuva responsiivisuus tarkoittaa sitä, että mobiilinäkymään siirrytään, kun portaali havaitsee päätelaitteen olevan mobiililaite, ja sijoittaa portaalin html-juurielementtiin CSS-luokan ”mobile”.

Räätälöity responsiivisuus toteutettiin siten, että teema-pluginin CSS-hakemiston juureen luotiin tiedosto ”variables_vismaconsulting.css”, jossa voidaan määrittää raja-arvot näytön leveydelle, joilla näkymät muuttuvat. Bootstrap 2.3.2 määrittää tablettinäkymän leveyden 768 ja 979 pikselin välille, horisontaalisen puhelimen näkymän 480 ja 767 pikselin välille, ja vertikaalisen puhelimen alle 480 pikselin. Nämä pikseliarvot voidaan ylikirjoittaa suoraan siten, että tarkasti määriteltyjen pikseliarvojen tilalle sijoitetaan SCSS-syntaksin mukainen muuttuja. Esimerkiksi tablettinäkymän ylärajalle annetaan tällöin muuttuja $breakpoint-tablet. Tämän jälkeen tiedostossa

”variables_vismaconsulting.css” tähän muuttujaan voidaan määritellä arvo, jonka jälkeen breakpointeja voidaan muokata suoraan muuttujatiedostosta.

Tämä vaatii toisaalta myös korjauksen Liferayn sisäisesti käyttämään ”respond-to” SCSS-mixiniin, joka luo media-attribuutin mixinin sisälle sijoitetun CSS-koodin ympärille. Tätä mixiniä käyttävät muunmuassa periytettävät teemat kuten ”classic” ja ”styled”, jolloin responsiivisuuden raja-arvot eivät ole yhtenäiset. Oletusarvoisesti ”respond-to”-mixin käyttää samoja arvoja kuin Bootstrap, jolloin ylikirjoitetut arvot eivät päde niihin. Ensimmäisenä ratkaisuna tähän käytettiin vain uudelleenkirjoitettua ”mixins.scss” –tiedostoa, jota periytettävän teeman tiedostot hakevat. Tämä toimi käännettäessä pluginia Maven-goalilla ”build-css”, mutta ei silloin, kun teema ajettiin portaaliin. Lopullisena ratkaisuna jouduttiin käyttämään erillistä Maven-pluginia, joka korvaa jokaisesta periytetystä teematiedostosta lausekkeen ”@import ’mixins’” muotoon ”@import ’mixins-override’”. Tällöin teema-pluginiin sijoitettu tiedosto mixins-override.scss on käytössä myös kaikilla teeman periyttämillä tiedostoilla, ja responsiivisuuden raja-arvot pysyvät yhtenäisinä.

Sen sijaan, että käytettäisiin vain päätelaitteen leveyttä responsiivisuuden toteuttamisessa, päädyttiin kahden asiakastyön puitteissa toteuttamaan responsiivisuus käyttäen myös user-agent-tunnistusta. Liferay-portaalialusta on tässä vaiheessa jo luonut hyvät puitteet kyseiselle totetutustavalle, sillä se osaa automaattisesti päätellä mm. selaimen käyttöjärjestelmän ja sen ominaisuudet. Kun portaalialusta havainnoi selaimen olevan mobiililaitteesta, portal_normal.vm-tiedoston <html> -juurielementtiin sijoitetaan CSS-valitsin “mobile”. Tämän jälkeen voidaan tarkistaa mobiililaitteen leveys, ja tehdä ainoastaan mobiiliulkoasu responsiiviseksi tavalliseen tapaan. Tämä muun muassa helpottaa teeman CSS-tiedostorakenteen modulaarisuutta siten, että responsiiviset tyylitiedostot voidaan sisällyttää tarvittaessa eri tiedostoihin, ja media-queryjen määrää voidaan vähentää.

Näiden kahden projektin perusteella rakenninsovelluksen pohjateemoihin lisättiin myös ‘separate-mobile-theme’, joka sisältää lähtökohdat teemojen luontiin erikseen desktop- ja mobiililaitteille.

5 TULOKSET JA POHDINTA

Kokonaisuudessaan opinnäytetyön lopputuotokset onnistuttiin toteuttamaan tavalla, joka on potentiaalisesti kehitystyötä nopeuttava. Merkittävimpiä muutoksia toi teema-pluginien Java-koodin ulkoistaminen erilliseen moduuliin, jonka seurauksena teemaan sisällytettyjen DDM-rakenteiden sekä template-tiedostojen vientiä palvelimelle voidaan yksinkertaistaa. Myös rakenninsovelluksen, gulp-kehitysskriptin sekä uudelleenkäytettävien komponenttien yhdistelmä on jo kirjoitusvaiheessa todettu toimivaksi uuden asiakasprojektin yhteydessä.

Ennen opinnäytetyötä olin tehnyt muutamia teema-plugineja osana yrityksen portaalitoimituksia. Opinnäytetyön tekemisen myötä käsitykseni Liferayn toiminnasta ja sen kehittämisestä syveni entisestään. Työn toteutus viivästyi, koska tutkimuskohde oli melko haastava ja sisälsi useampia osa-alueita. Varsinkin uudelleenkäytettävien komponenttien kehittäminen ja testaaminen vei arvioitua enemmän aikaa. Jälkikäteen tarkasteltuna varsinaisen toteutusalueen olisi voinut rajata kapeammaksi.

Työn tutkimusmenetelminä tutustuin kirjalliseen materiaaliin, sekä havainnoin aikaisemmin tehtyjä projekteja ja niissä käytettyjä toimintamalleja. Havaintojen perusteella kehitettyjä ratkaisuja vertailin keskenään testaamalla niitä käytännössä. Liferayn pitkäaikaisen kehityskaaren vuoksi pyrin sen osalta käyttämään lähteinä mahdollisimman ajantasaista kirjallisuutta ja dokumentaatiota. Osa dokumentaatiosta oli kuitenkin puutteellista tai vanhentunutta, minkä takia välillä oli tarpeellista selvittää portaalin yksityiskohtia suoraan lähdekoodista. Ongelmatilanteissa sain myös korvaamatonta apua kollegoiltani, ja sain heiltä myös kehitysvaiheessa hyviä ehdotuksia toteutuksen toiminnallisuuksiin liittyen.

Uudelleenkäytettäviä komponentteja toteutettaessa kävi ilmi, että useamman eri täsmäkielen muodostama komponentti on hyvin vaikea toteuttaa muulla tavoin kuin käyttäen kopiointia uudelleenkäyttöstrategiana. Tällaisia olivat muun muassa prototyyppeinä toteutetut navigaatio-templatet, jotka vaativat sekä JavaScript-, CSS-, ja Velocity-tiedostoja. Strategia ei ole huono, mutta se ei ole myöskään aivan ongelmaton, mikä on harmillista.

Teema-pluginin tuotearkkitehtuuria varten määriteltiin odotettua vähemmän sääntöjä, sisältäen vain muutamia nimeämis- sekä hakemistorakennekäytäntöjä, jotka helpottavat koodin luettavuutta ja hallintaa. Säännöstön pitäminen melko väljänä kuitenkin helpottanee mallin käyttöönottoa muissa projekteissa.

Jatkossa uudelleenkäytettävien komponenttien kirjastoa voidaan laajentaa uusilla komponenteilla, joiden toistuvuus on runsasta eri projektien välillä ja joiden uudelleenkäytöstä voidaan saada täten hyötyä yritykselle.

LÄHTEET

Biggerstaff, T & Richter, C. 1987. Reusability Framework, Assessment, and Directions. IEEE Software 4 (2), 41-49.

Cavanaugh, N. 2016. The status and direction of the frontend infrastructure in Liferay 7 & DXP. Liferay. Luettu 9.2.2017 https://web.liferay.com/web/nathan.cavanaugh/blog/-/blogs/the-status-and-direction-of-the-frontend-infrastructure-in-liferay-7-dxp

Chen, X & Yuan, J & Yu, F. 2013. Liferay 6.2 User Interface Development. 1. painos. Birmingham: Packt Publishing Ltd.

Colombo, F. 2011. It's not just reuse. Luettu 5.3.2017 http://sharednow.blogspot.fi/2011/05/its-not-just-reuse.html

Inquirer.js, 2017. Prompt types. Luettu 25.3.2017. https://github.com/SBoudrias/Inquirer.js

JetBrains, 2017. Supported languages. Luettu 25.3.2017. https://www.jetbrains.com/help/idea/2016.3/supported-languages.html

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. 1. painos. Jyväkylä: Talentum Oy.

Lalou, J. 2013. Apache Maven Dependency Management. 1. painos. Birmingham: Packt Publishing Ltd.

Liferay, 2016. Developing Liferay Theme Plugins with Maven. Luettu 25.3.2017. https://dev.liferay.com/develop/tutorials/-/knowledge_base/6-2/developing-liferay-theme-plugins-with-maven

Liferay, 2017. Expando. Luettu 2.4.2017 https://web.liferay.com/community/wiki/-/wiki/Main/Expando

Long, J. 2001. Software Reuse Antipatterns. Software Engineering Notes 26 (4), 68-76.

O'Reilly, 2017. What is a portlet?. Luettu 27.3. 2017. http://archive.oreilly.com/pub/a/java/archive/what-is-a-portlet.html?page=2

Sass reference. 2016. Luettu 20.3.2017 http://sass-lang.com/documentation/file.SASS_REFERENCE.html#syntax

Yeoman, 2017a. What's Yeoman? Luettu 25.3.2017. http://yeoman.io/

Yeoman, 2017b. Interacting with the user. Luettu 25.3.2017. http://yeoman.io/authoring/user-interactions.html