Tampereen ammattikorkeakoulu

Uuden toiminnallisuuden luominen Liferay-projektiin Service Builderilla ja Reactilla

Sampsa Järvinen

OPINNÄYTETYÖ
Toukokuu 2024

Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

JÄRVINEN, SAMPSA:
Uuden toiminnallisuuden luominen Liferay-projektiin Service Builderilla ja Reactilla

Opinnäytetyö 35 sivua
Toukokuu 2024

Opinnäytetyön aiheena oli uuden toiminnallisuuden lisääminen Liferay-alustalla toimivalle suuren julkisen toimijan intra-sivustolle. Toiminnallisuus mahdollistaa sivuston sisäisten sivujen tallentamisen suosikeiksi ja niiden listaamisen käyttäjäkohtaisesti. Työssä esitellään ominaisuuden back- ja frontend kehitystä hyviä ohjelmointikäytänteitä noudattaen sekä Liferayn käyttöä alustana ja sen ohjelmistokehitykseen vaikuttavia erityispiirteitä.

Backend-puoli toteutetaan Liferay Service Builderin avulla, joka luo xml-pohjaisen määrittelyn pohjalta uudelle entiteetille Javalla toteutetun mallitason, persistanssi-tason tietokantayhteyksiä varten sekä palvelutason toimintalogiikalle. Työssä käsitellään myös tapoja, joilla näitä tasoja täydennetään ja kehitetään haluttujen ominaisuuksien luomiseksi.

Frontend-yhteyksiä varten esitellään tapa toteuttaa Liferay-ympäristöön REST-rajapinta entiteetin ominaisuuksien hyödyntämiseksi. Projektin frontend toteutetaan luomalla Liferayn sisällä ajettavat React-moduulit suosikkien lisäämiseksi, listaamiseksi ja hallinnoimiseksi.

Valmis työ on näin sekä toimeksiantaja Twoday Oy:n asiakkaan tarpeeseen luotu toiminnallisuus että hyviä ohjelmointikäytänteitä noudattava ja helposti yleistettävä esimerkki uuden ominaisuuden luomiseksi Liferay-projektiin.

Asiasanat: Java, Liferay, moduuli-kehitys, React, service builder, xml-määrittely

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

JÄRVINEN, SAMPSA:
Creating New Functionality to Liferay Project with Service Builder and React

Bachelor's thesis 35 pages
May 2024

The thesis reported on the addition of a new functionality to the intranet site of a large public entity operating on the Liferay platform. The functionality enables users to save internal pages of the site to a personal favourites listing. The work presented both backend and frontend development of the feature, following good programming practices, and Liferay as a platform with its characteristics affecting software development.

The backend implementation was carried out using Liferay Service Builder which generates, based on an XML-based definition, a Java-based model for the new entity, including the model layer, persistence layer for database connections, and service layer logic. The thesis also discussed the methods to complement and enhance these layers for creating the desired features.0

For frontend connections, a method for implementing a REST API in the Liferay environment to leverage the entity's features was presented. The project frontend was implemented by creating React modules within Liferay to add, list, and manage favourites.

The completed work serves as both a functionality created to meet the needs of the client of the thesis' commissioner, Twoday Oy, and an example adhering to good programming practices and easily generalizable for creating a new feature in a Liferay project.

Key words: Java, Liferay, module-development, React, service builder, xml-definition

SISÄLLYS

1 JOHDANTO 6
2 TOIMINNALLISUUDEN BACKEND-OSUUDEN LUOMINEN LIFERAY SERVICE BUILDERILLA 9
2.1 Ominaisuuden määrittely 9
2.2 Moduulin luonti 9
2.3 XML-määrittely 11
2.4 Tietokantataulujen luonti 14
3 SERVICE BUILDERIN LUOMAT LUOKAT JA RAJAPINNAT 17
3.1 Kansiorakenne 17
3.2 Userfavorites-service-moduuli 17
3.3 Userfavorites-api-moduuli 20
4 REST-RAJAPINTA FRONTENDILLE 21
4.1 Rest-rajapinnan toteutuksen rakenne 21
4.2 UserFavorite-resurssi 22
5 REACT FRONTEND 23
5.1 React-portlettien luominen Liferay-ympäristöön 23
5.2 React-frontend Liferayssä 24
5.3 Sovelluksessa käytettyjen React-ominaisuuksien teoriaa lyhyesti 25
5.4 Sovelluksen frontend portletit ja niiden logiikka 27
5.5 Saavutettavuus 30
6 POHDINTA 31
LÄHTEET 34

LYHENTEET JA TERMIT

Backend	Ohjelmiston palvelimella suoritettava osa.
Frontend	Ohjelmiston käyttäjän selaimessa suoritettava osa.
Gradle	Avoimen lähdekoodin automaatiotyökalu, joka tarjoaa tehokkaan tavan hallita projektien riippuvuuksia.
IDE	Integrated Development Environment, ohjelmistokehitystyökalu, joka tarjoaa ympäristön koodin kirjoittamiseen, kääntämiseen, testaamiseen ja virheiden paikannukseen yhdessä sovelluksessa.
Moduulikehitys	Ohjelmistokehitystapa, jossa ohjelma jaetaan pienempiin itsenäisiin osiin, mikä helpottaa ylläpidettävyyttä, uudelleenkäytettävyyttä ja projektinhallintaa.
OSGi	Open Service Gateway Initiative, modulaarinen Liferayn hyödyntämä ohjelmistokehitysstandardi, joka mahdollistaa Java-pohjaisten sovellusten jakamisen ja suorittamisen itsenäisinä osina, bundleina.
SQL	Structured Query Language, kyselykieli, jota käytetään relaatiotietokantojen hallintaan ja tietojen käsittelyyn.
XML	Extensible Markup Language, merkintäkieli tietojen tallentamiseen ja siirtämiseen. Käyttää hierarkkista rakennetta ja mukautettuja merkintöjä, kuten elementtejä ja attribuutteja.

1 JOHDANTO

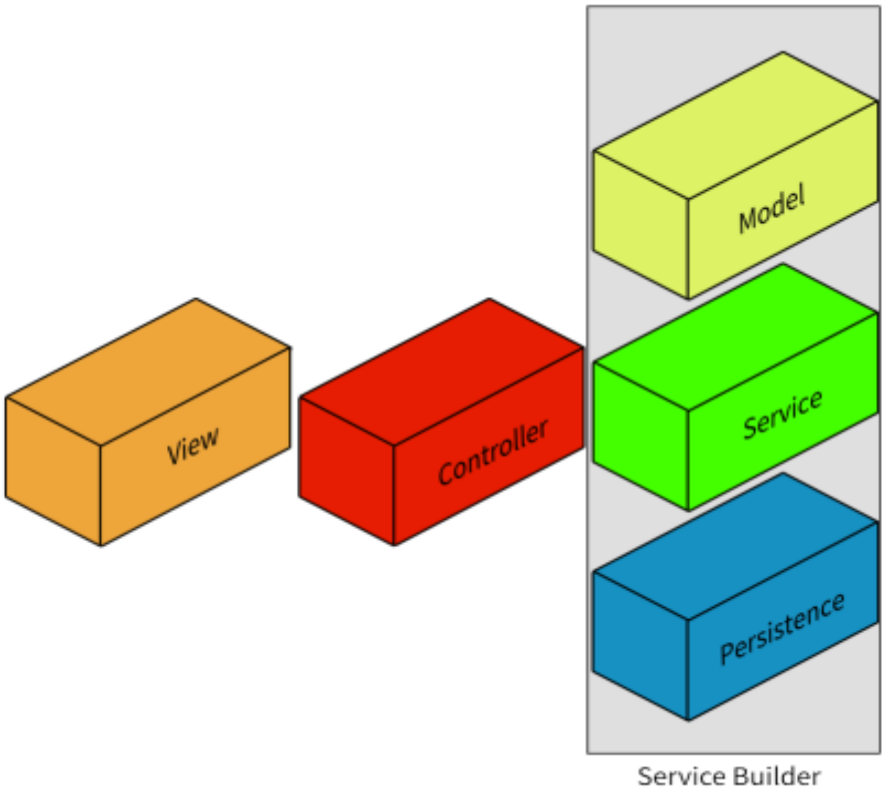
Opinnäytetyössä esitellään Liferay suuren julkisen toimijan intranet-sivuston toteutuksen alustana. Työn aiheen pohjana oli asiakkaan tarve kehittää sivuston käyttäjille mahdollisuus tallentaa laajan sivuston sisäisiä sivuja omaan helposti hallittavaan suosikkilistaukseen. Tämän esimerkin kautta esitellään, miten Liferay-alustaan voidaan verrattain nopeasti lisätä tämän tyyppisiä toiminnallisuuksia hyödyntäen Liferayn Service Builder automatiikkaa.

Liferay on avoimen lähdekoodin ohjelmisto (Liferay Help Center, Fundamentals n.d.). Se tarjoaa melko kattavan pohjan esimerkiksi verkkosivuston rakentamiseen ja sisältää valmiiksi järjestelmät sisällönhallintaan, käyttäjähallintaan ja esimerkiksi monikielisyyteen. Liferayn hyviä puolia onkin alustan pystyttämisen nopeus, ja se että sisällönhallinta on mahdollista täysin ilman ohjelmointiosaamista. Liferayn valinta kehitettävän ohjelmiston alustaksi kannattaa kuitenkin tarkoin harkita, koska vaikka Liferay on muokattavissa, on joidenkin valmiiden ominaisuuksien säätäminen muuten kuin ulkoasun osalta haastavaa tai jopa lähes mahdotonta. Yksinkertaiselle sivustolle ohjelmisto on todennäköisesti myös tarpeettoman monimutkainen ja raskas. Toisaalta sivustolle, joka hyödyntää monia Liferayn sisällönhallinnan valmiita ominaisuuksia, kuten esimerkiksi sisällön kommentointimahdollisuutta ja sähköposti-ilmoituksia, on Liferay todennäköisesti tarkoituksenmukainen valinta.

Liferayn Service Builder on työkalu, joka luo ohjelmistoon uusia ominaisuuksia kehittäjän toteuttaman määrittelytiedoston pohjalta. Määrittelytiedosto sisältää käytännössä tiedot uuden ominaisuuden tarvitsemiin tietokantatauluihin, niiden välisiin yhteyksiin sekä esimerkiksi tiedon millä kriteereillä taulun sisältöä voidaan hakea tietokannasta persistance eli pysyväistallennus-kerroksessa.

Service Builder luo SQL-kyselyt tietokantataulujen luomiseksi, entiteettien Java-luokat (model, eli mallikerros), service, eli palvelukerroksen sekä pysyväistallennuskerroksen (Kuvio 1). SQL-kyselyillä tarkoitetaan Structured Query Language kyselykieltä, jota käytetään relaatiotietokantojen hallintaan ja tietojen käsittelyyn. Kerroksittainen toteutus tukee MVC-arkkitehtuurin mukaista ratkaisua, eli Model-

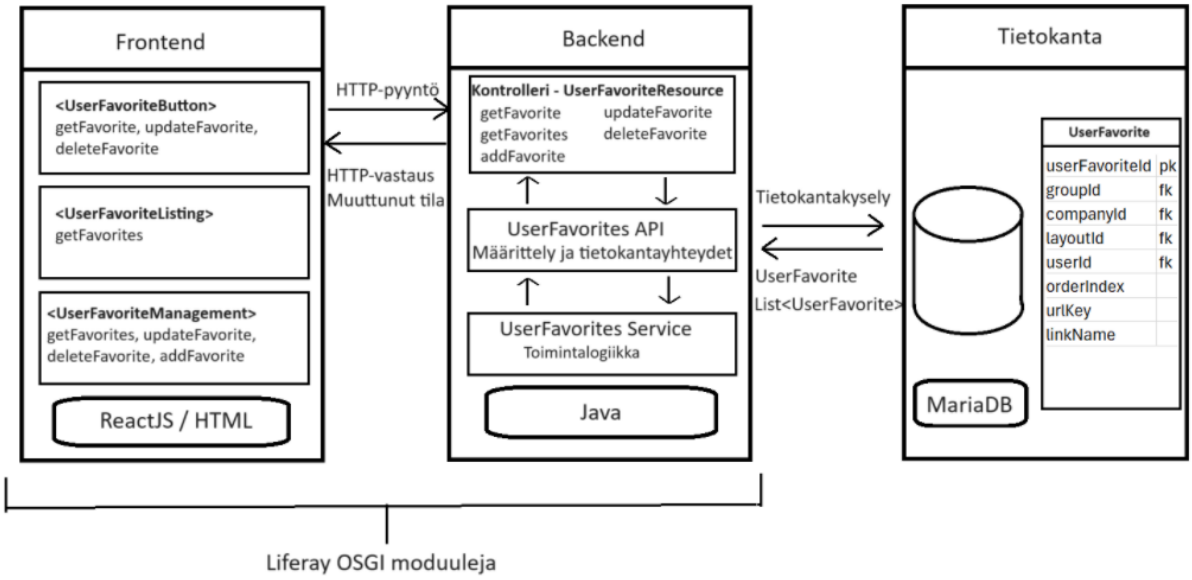
View-Controller-arkkitehtuuria, joka auttaa eriyttämään tietokantaan yhteydessä olevan palvelinpuolen ja käyttöliittymän logiikan. Kontrolleri- ja näkymätasot toteutetaan kuitenkin erikseen ilman automaatiota. Kontrolleri ottaa yhteyden palvelukerrokseen, joka puolestaan hallinnoi yhteyksiä pysyväistallennuskerrokseen ja mallikerrokseen. Pysyväistallennuskerros on vastuussa tietokantakyselyistä.



KUVIO 1. Service Builderin luoma ohjelmistorakenne. (Liferay Help Center, Generating the Back-end n.d.)

Kerrosten yhteyksille toteutetaan rajapinnat, jotka kerrosten metodien täytyy toteuttaa. Tämä edistää koodin uudelleenkäytettävyyttä ja hallittavuutta. Automatiikka myös vähentää inhimillisten virheiden mahdollisuutta toistuvassa koodiosuudessa ja esimerkiksi pysyväistallennuskerros toteuttaa automaattisesti transaktioperiaatteita, jotka auttavat varmistamaan tietojen eheyden ja konsistenssin. Kuviossa 2. esitellään tarkemmin toteutettavan sovelluksen eri osien komponentteja ja niiden välisiä yhteyksiä. Frontendin, eli ohjelmiston käyttäjän selaimella suoritettavan osuuden, pienoissovellukset ovat HTTP-pyyntöjen väli-

tyksellä yhteydessä backendin, eli ohjelmiston palvelimella suoritettavan osuuden, kontrolleriin. Kontrolleri on rajapinnan (API) välityksellä yhteydessä toimintalogiikkaan (Service) ja tietokantaan ja palauttaa sovelluksen muuttuneen tilan HTTP-muotoisena vastauksena frontendiin.



KUVIO 2. Sovelluksen keskeiset komponentit, metodit ja niiden väliset yhteydet.

Työn esimerkkitoteutuksessa frontend toteutetaan Liferayn sisällä ajettavalla React-moduulilla, joka on yhteydessä backendiin kontrolleritason välityksellä. Frontend puolella keskitytään Liferayn React-kehitykseen tuottamiin erityispiirteisiin ja huomioitaviin asioihin.

2 TOIMINNALLISUUDEN BACKEND-OSUUDEN LUOMINEN LIFERAY SERVICE BUILDERILLA

2.1 Ominaisuuden määrittely

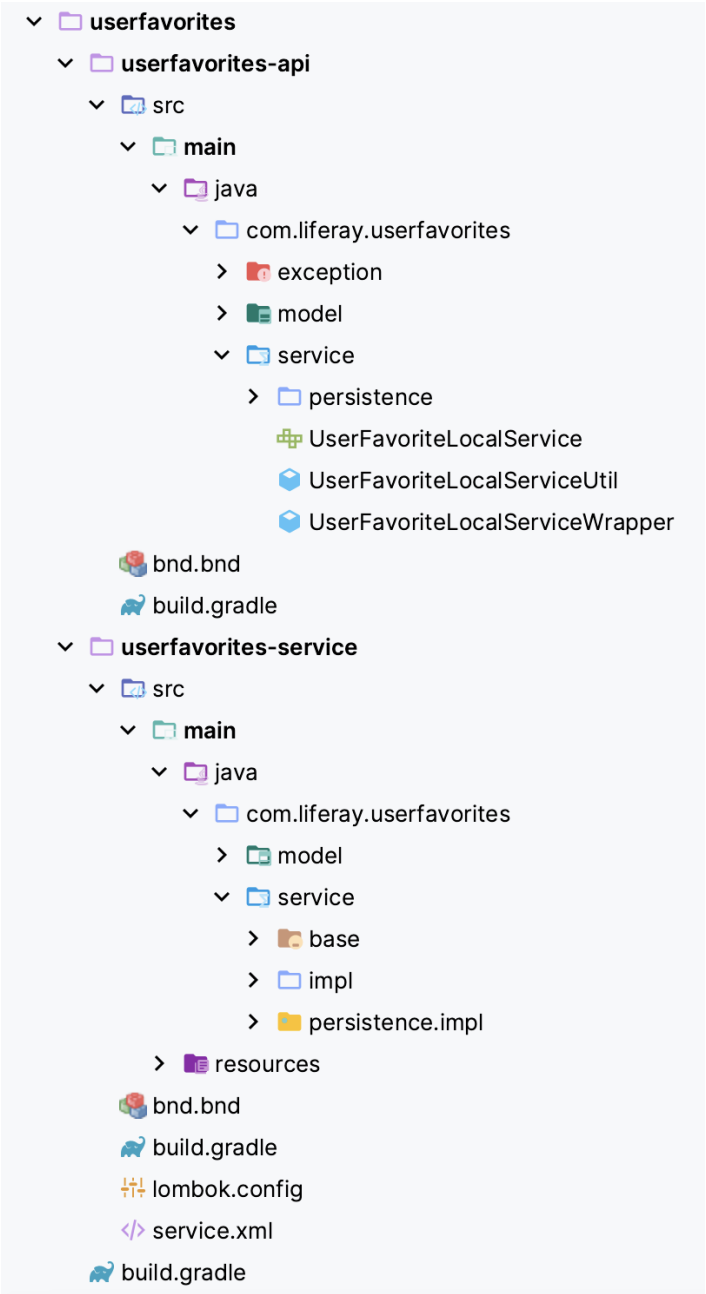
Asiakas tilasi toiminnallisuuden, jonka avulla intranet-sivuston käyttäjät voivat lisätä minkä tahansa sivuston sisäisen sivun omaan suosikkilistaukseen jokaisella sivulla näkyvän napin kautta. Suosikeille luodaan käyttäjäkohtaisesti kaksi näkymää: linkkilistaus ja hallinnointinäkymä. Hallinnointinäkymässä suosikkeja tulee voida järjestää, uudelleen nimetä ja poistaa.

2.2 Moduulin luonti

Liferay-kehitykseen on tarjolla useita eri työkaluja ja vaikka toiminnallisuudet ovat pääpiirteittäin samat ja Service Builder ohjeeni ovat todennäköisesti sovellettavat useilla työkaluilla ja versioilla, voi työnkulussa olla eroavaisuuksia. Epäselvyyksien välttämiseksi esittelen opinnäyteprojektissani käytettävät ohjelmointityökalut. Projektin ohjelmisto on Liferay Portal -julkaisun 7.4.3.68 version pohjalta luotu Liferay Workspace. Riippuvuuksien hallinnassa on käytössä Gradlen versio 6.6.1, käytetty Java-versio on OpenJDK 11.0.19 ja IDE:nä, eli ohjelmistokehitysohjelmistona, IntelliJ IDEA, jossa on asennettuna Liferay-projektinhallinta lisäosa.

Kun IDE:ssa on käytössä Liferay-lisäosa, on uuden toiminnallisuuden toteuttavan projektirakenteeseen sopivan moduulin luonti helppoa. Valitaan IDE:ssa uuden moduulin luonti, valitaan generaattoriksi *Liferay* ja moduulin tyypiksi *Liferay Modules*. Nimetään moduuli, tässä tapauksessa *userfavorites*, asetetaan projektin tyypiksi *service-builder* ja asetetaan *Package Name*, jota hyödynnetään moduulin luokkien nimiavaruutena yksikäsitteisyyden takaamiseksi ja moduulin mahdollisten riippuvuuksien luomisessa. Lopputuloksena on kuvion 3 mukainen kansiorakenne, jossa näkyy sovelluksen jako rajapintamoduuliin userfavorites-api ja toimintalogiikkamoduuliin userfavorites-service. Build.gradle-tiedostot sisältävät ohjeet projektin koonnissa (build) tarvittavien riippuvuuksien hallintaan, `bnd.bnd`-tiedostot sisältävät ohjeet moduulien pakkaamiseen Liferayn käyttämiksi OSGi-

yhtensopiviksi bundleiksi. OSGi on Liferayn hyödyntämä ohjelmistokehitysstandardi, joka mahdollistaa Java-pohjaisten sovellusten jakamisen ja suorittamisen itsenäisinä osina.



KUVIO 3. Liferayn Service Builderin luoma projektirakenne, joka jakaa sovelluksen rajapinta- ja toimintalogiikkamoduuleihin.

2.3 XML-määrittely

XML muodostuu sanoista Extensible Markup Language ja tarkoittaa merkintäkieltä, jota käytetään tietojen tallentamiseen ja siirtämiseen. Luotu `service.xml`-tiedosto on uuden toiminnallisuuden rakentamisen pohjapiirros. XML-määrittelyä tehdessä on myös mietittävä mitä ominaisuuksia uudelle suosikkilistaus ominaisuudelle tarvitaan. Service Builder on tosin ajettavissa useaan kertaan, jos service.xml-tiedostoon tarvitaan muutoksia kesken kehitystyön. Suosikkiominaisuuden pohjan luontiin käytetty määrittely on esitetty kuvassa 1.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.4.0//EN"
3          "http://www.liferay.com/dtd/liferay-service-builder_7_4_0.dtd">
4  <service-builder dependency-injector="ds" package-path="com.liferay.userfavorites">
5      <namespace>USERFAVORITE</namespace>
6      <entity local-service="true" name="UserFavorite" remote-service="false">
7          <column name="userFavoriteId" primary="true" type="long" />
8          <column name="groupId" type="long" />
9          <column name="companyId" type="long" />
10         <column name="layoutId" type="long" />
11         <column name="userId" type="long" />
12         <column name="orderIndex" type="long" />
13         <column name="createDate" type="Date" />
14         <column name="modifiedDate" type="Date" />
15         <column name="urlKey" type="String" />
16         <column name="linkName" type="String" />
17         <!-- Order -->
18         <order by="asc">
19             <order-column name="linkName" />
20         </order>
21         <!-- Finder methods -->
22         <finder name="UserId" return-type="Collection">
23             <finder-column name="userId" />
24         </finder>
25         <finder name="LayoutId" return-type="Collection">
26             <finder-column name="layoutId" />
27         </finder>
28         <finder name="UrlKeyCompanyGroup" return-type="Collection">
29             <finder-column name="urlKey" />
30             <finder-column name="companyId" />
31             <finder-column name="groupId" />
32         </finder>
33         <finder name="GroupIdLayoutIdUserIdUrlKey" unique="true" return-type="UserFavorite">
34             <finder-column name="groupId" />
35             <finder-column name="layoutId" />
36             <finder-column name="userId" />
37             <finder-column name="urlKey" />
38         </finder>
39     </entity>
40 </service-builder>
```

KUVA 1. XML-määrittely Liferay Service Builderille projektin luomista varten.

- Riveillä 2 ja 3 haetaan ohjeet, joiden mukaan Service Builder lukee xml-määrittelyä.
- Rivi 4 kertoo Service Builderille mihin pakettiin entiteetin tarvitsemat luokat luodaan.
- Rivi 5 luo entiteetille nimiavaruuden, jotta tietokantataulujen nimeämisessä ei tule konflikteja.
- Rivi 6 määrittelee luotavan palvelun paikalliseksi ja nimeää entiteetin. Jokaiselle entiteetille luodaan oma taulu ja Service Builderillä voisi määrittää samaan moduuliin useamman toisistaan riippuvan entiteetin yhdellä xml-tiedostolla, mutta toiminnallisuutemme edellyttää vain yhtä. Service Builderillä voisi toteuttaa myös etäpalvelun, jolloin entiteetin toiminnallisuudet toteutettaisiin myös sovelluksen ulkopuolelle avoimiksi esimerkiksi JSON-pohjaisena rajapintana.
- Rivi 7 määrittelee entiteetin identifioivan pääavaimen ja sen tyypin.
- Rivit 8–16 määrittelevät muut entiteetin tarvitsemat ominaisuudet ja niiden tyypit, jotka siis toteutetaan myös tietokantataulun kolumneina.

Määrittelyä tehdessä on oleellista tuntea joitain Liferayn keskeisiä käsitteitä. *CompanyId* viittaa samassa Liferay-portaalissa mahdollisesti toimiviin sivustoihin, jotka ovat erillisiä toisistaan ja joilla on omat käyttäjät, sivut ja asetukset. *GroupId* puolestaan viittaa sivuston sisäisiin alaryhmiin, jotka voivat jakaa resursseja, kuten sivuja ja dokumentteja sekä käyttävät samoja asetuksia. Näillä alaryhmillä on samat käyttäjät, mutta esimerkiksi käyttöoikeudet voidaan määritellä erikseen. Yksikäsitteisyyden vuoksi nämä atribuutit on siis määriteltävä käytännössä jokaiselle entiteetille.

LayoutId vittaa Liferayssä käytettyihin sivupohjiin. Yksinkertaisimmillaan *layoutId* riittää määrittelemään sivunäkymän yksikäsitteisesti, mutta esimerkiksi saman rakenteen omaavia artikkeleita voidaan esittää niiden näyttösivumallin mukaan samalla sivupohjalla, jolloin näissä tapauksissa tarvitaan lisäksi jokin muu sivunäkymän yksilöivä tekijä. Pelkän URL-osoitteen käyttö suosikkeja tallennettaessa ei myöskään olisi kaikissa tilanteissa toimiva ratkaisu, koska sekä Liferayn automatiikka, että sivuston sisällöntuottajat voivat vaihtaa sivupohjan URL-osoitetta. Näiden seikkojen vuoksi päädyin käyttämään toteutuksessani sivupohjan

id:tä silloin kun se riitti yksilöimään sivunäkymän ja sivupohjan *id*:n ja URL-osoitteen (*urlKey*) yhdistelmää silloin kun se oli yksilöimisen takia tarpeen.

OrderIndex ominaisuutta käytetään sovelluksessa mahdollistamaan käyttäjille omien suosikkien järjestely ja *linkName* mahdollistaa linkin nimeämisen käyttäjän haluamalla tavalla.

- Riveillä 18–20 määritellään minkä perusteella suosikit järjestetään, kun niitä palautetaan tietokannasta kokoelmina. Tästä on jäänyt *orderIndex* pois, koska tämä ominaisuus toteutettiin kehityksen myöhemmässä vaiheessa ja järjestely suoritettiin tältä osalta itse toteutetussa palvelutason logiikassa. Tarvittaessa voitaisiin toteuttaa myös usean ominaisuuden mukaan järjestely lisäämällä useampi *order-column*-määrittely, jolloin suosikit järjestettäisiin ensisijaisesti ensimmäisen mukaan ja sen ollessa sama, seuraavan ominaisuuden mukaan.

- Riveillä 22–38 määritellään automaattisesti luotujen hakujen lisäksi tapoja, finder-metodeja, joilla persistanssitasolla haetaan tietokannasta suosikkeja ja palautetaan ne kokoelmina. Tyypillinen hakutapaus on tietysti palauttaa kaikki suosikit käyttäjäkohtaisesti *userId*:n mukaan. Viimeisessä määrittelyssä rivillä 33–38 toteutetaan *unique*-ominaisuus, joka määrää, että tietokannassa saa esiintyä vain yksi rivi, jolla tässä määritellyt arvot ovat samat. Näin tämä haku palauttaa aina korkeintaan yhden tuloksen.

Service Builderillä voisi myös luoda pohjan entiteettiin liittyville poikkeuksille (exceptions) seuraavasti:

```
<exceptions>
        <exception>UserFavorite</exception>
</exceptions>
```

Näin toteutettu `service.xml` on valmis käytettäväksi Service Builderin pohjana ja se sisältää jo paljon toiminnallisuudessa tarvitsemaamme informaatiota. Service Builderin voi ajaa esim. IDE:n Gradlen kautta modules → userfavorites → userfavorites-service → build → buildService.

2.4 Tietokantataulujen luonti

Service Builder luo määrittelyn pohjalta tiedostot entiteetin luontia varten tarvittaville SQL-kyselyille. `Tables.sql`-tiedosto sisältää kyselyt tietokantataulun tai taulujen luontia varten (Kuva 2.) ja `indexes.sql`-tiedosto (Kuva 3.) sisältää määrittelyn *finder*-metodien pohjaksi ja hakujen nopeuttamiseksi tietokantaindeksien luomista varten tarvittavat kyselyt. Myös `service.xml`-tiedostossa määritelty *unique*-ominaisuus toteutetaan tässä tiedostossa *create unique index* -kyselyllä.

```
create table USERFAVORITE_UserFavorite (
    userFavoriteId LONG not null primary key,
    groupId LONG,
    companyId LONG,
    layoutId LONG,
    userId LONG,
    orderIndex LONG,
    createDate DATE null,
    modifiedDate DATE null,
    urlKey VARCHAR(75) null,
    linkName VARCHAR(75) null
);
```

KUVA 2. Service Builderin luoman `tables.sql` tiedoston sisältämä SQL-kysely, jolla luodaan tietokantaan tietokantataulu määritellyin ominaisuuksin.

```
create unique index IX_205AC63C on USERFAVORITE_UserFavorite (groupId, layoutId, userId, urlKey[$COLUMN_LENGTH:75$]);
create index IX_96A4EEDE on USERFAVORITE_UserFavorite (layoutId);
create index IX_B2FB16B on USERFAVORITE_UserFavorite (urlKey[$COLUMN_LENGTH:75$], companyId, groupId);
create index IX_20FB3CBF on USERFAVORITE_UserFavorite (userId);
```

KUVA 3. Service Builderin luoman indexes.sql tiedoston sisältämät SQL-kyselyt, joilla luodaan tietokantaan hakuja nopeuttavat indeksit.

Kuten kuvasta 2 nähdään, toteutetaan `service.xml`:n *type="String"* -kentät oletusarvoisesti tietotyyppinä VARCHAR(75), joka ei sovelluksessamme riitä kaikkiin käyttötarkoituksiin. VARCHAR:n maksimipituuteen vaikuttaminen on Service Builderissä toteutettu hieman kömpelösti. Service Builder luo myös tiedoston `portlet-model-hints.xml`, joka sisältää kaikki tietokantataulussa esiintyvät

kentät. Täällä voidaan uudelleen määrittää VARCHAR-tietotyypin maksimipituus seuraavasti:

```
<field name="linkName" type="String">
     <hint name="max-length">255</hint>
</field>
```

Modernit tietokannat käyttävät jokaiseen VARCHAR-merkintään vain tarvittavan määrän muistia käyttämällä muutaman bitin merkinnän pituuden määrittämiseen (Gennick & Mishra 2001, 3.2. Datatypes). Tietotyypin tarvittava pituus kannattaa kuitenkin harkita sillä liian lyhyeksi määrittely voi aiheuttaa käytettävyysongelmia. Turha pituus puolestaan voi aiheuttaa haasteita syötteiden validoinneissa ja datan yhtenäisyydessä. Kehitysvaiheessa tietotyyppien muuttaminen onnistuu ajamalla Service Builder uudestaan `portlet-model-hints.xml` päivityksen jälkeen ja käynnistämällä Liferay-sovellus uudelleen.

Tuotannossa olevan sovelluksen ja tietokannan muuttaminen onnistuu turvallisesti ja verrattain helposti Liferayn tarjoamien *UpgradeStepRegistrator* Java-rajapinnan ja *UpgradeProcess* Java-luokan avulla. Kuvassa 4 nähtävä itse toteutettu luokka *UpgradeSchema_V101* periytyy luokasta *UpgradeProcess* ja toteuttaa metodin *doUpgrade*, jossa määritellään tietokantatauluun haluttavat muutokset. Linkin käyttäjän antama nimi määritettiin maksimissaan 255 merkin pituiseksi ja mahdollisen suosikin suoran url:n sisältämä *urlKey*-kenttä määritettiin termillä STRING, joka muuntuu tietokannassa hyvin pitkän maksimipituuden omaavaksi LONGTEXT tietotyypiksi.

```java
package com.liferay.userfavorites.model.upgrade;
import com.liferay.portal.kernel.upgrade.UpgradeProcess;

public class UpgradeSchema_V101 extends UpgradeProcess {
    private static final String TABLE = "USERFAVORITE_UserFavorite";

    @Override
    protected void doUpgrade() throws Exception {
        if (!hasColumnType(TABLE, columnName: "urlKey", columnType: "STRING NULL")) {
            alterColumnType(TABLE, columnName: "urlKey", newColumnType: "STRING NULL");
        }

        if (!hasColumnType(TABLE, columnName: "linkName", columnType: "VARCHAR(255) NULL")) {
            alterColumnType(TABLE, columnName: "linkName", newColumnType: "VARCHAR(255) NULL");
        }
    }
}
```

KUVA 4. Liferay-projektin tuotannossa olevien tietokantataulujen muokkaukseen käytetty Java-koodi.

Heti sovelluksen käynnistyessä ajettavassa Liferay-komponentissa *(@Component, immediate = true*) taas toteutetaan *UpgradeRegistrator*-rajapinta ja ohjeistetaan päivittämään tietokannan skeema Liferayn rekisterissä versioon 1.0.1 *doUpgrade*-metodissa määritellyn ohjeistuksen mukaisesti (Kuva 5).

```java
package com.liferay.userfavorites.model.upgrade;
import com.liferay.portal.upgrade.registry.UpgradeStepRegistrator;
import org.osgi.service.component.annotations.Component;

@Component(immediate = true, service = UpgradeStepRegistrator.class)
public class UserFavoritesUpgradeStepRegistrator implements UpgradeStepRegistrator {
    @Override
    public void register(Registry registry) {
        registry.register( fromSchemaVersionString: "1.0.0",
                toSchemaVersionString: "1.0.1", new UpgradeSchema_V101());
    }
}
```

KUVA 5. Liferayn tarjoaman *UpgradeStepRegistrator*-rajapinnan hyödyntäminen tietokantataulun päivittämisessä.

3 SERVICE BUILDERIN LUOMAT LUOKAT JA RAJAPINNAT

3.1 Kansiorakenne

Service Builder luo userfavorites moduulin sisälle kaksi erillistä moduulia: userfavorites-api ja userfavorites-service (Kuvio 3). Tämä jako nojaa ohjelmistokehityksessä yleisesti hyödynnettäviin abstrakion ja kapseloinnin periaatteisiin, joilla tässä tarkoitetaan, että api-moduuli toimii konnektorina, joka tarjoaa ulospäin selkeät metodit entiteetin kanssa toimimiseen ja service-moduuli sisältää monimutkaisemman sisäisen toimintalogiikan (Taylor, Medvidovic & Dashofy 2009, 5.2. Connector Foundations). Service Builder luo ajettaessa api-moduulin sisällön uudestaan service-moduulin pohjalta ja päivittää rajapinnan tiedot muutosten osalta. Kehitystyö tehdään siis pääasiassa service-moduulissa.

3.2 Userfavorites-service-moduuli

Moduuli jakautuu kahteen osaan: *model* ja *service*. *Model*-osa sisältää *UserFavoriteModelImpl*-luokan, joka edustaa *UserFavorite*-tietokantataulun riviä Java-luokkana. Tämä luokka tarjoaa pohjan tietokantarivin käsittelyyn, mukaan lukien getterit ja setterit, eli luokan atribuuttien arvojen hakuun ja asettamiseen käytettävät metodit, sekä *hashCode()* ja *equals()* -metodit.

Oma kehitystyöni suosikkilistauksen tarvitsemien backend-ominaisuuksien luomiseksi tapahtui pääasiallisesti *service*-osan UserFavoriteLocalServiceImpl-luokassa. Luokka on vastuussa suosikkien käsittelystä ja tarjoaa joukon metodeja suosikkien hakemiseen, päivittämiseen ja poistamiseen käyttäjän toimesta.

getUserFavoritesByUserId: Metodi palauttaa tietyn käyttäjän suosikkilistauksen. Se hakee ensin käyttäjän suosikit tietokannasta api-moduulista löytyvän *userFavoritePersistance*-luokan kautta käyttäjän *id*:n perusteella. Metodi muuntaa suosikit listaksi map-olioita, joissa suosikkien attribuutit ovat avain–arvo pareina. Lisäksi metodi rakentaa suosikkien URL:t hyödyntämällä apuna Liferayn tarjoamia valmiita luokkia *LayoutLocalServicea* ja *GroupLocalServicea*. Tämä

siksi, että Liferay luo sivupohjille (layout) käyttäjäystävällisen osoitteen, joka saattaa kuitenkin muuttua. Näin ollen sitä ei kannata tallentaa tietokantaan sellaisenaan, vaan hakea aina ajankohtainen tieto suosikin sivupohjan ja alaryhmän (group) mukaan. *GetUserFavoritesByUserId*-metodia käytetään myös siivoamisessa, kun käyttäjä poistetaan Liferay-sovelluksesta. Tässä hyödynnetään Liferayn valmiita luokkakohtaisia kuuntelijoita, joiden metodeja, kuten *onAfterRemove*, voidaan laajentaa, kun halutaan suorittaa toimia käyttäjän poiston yhteydessä. Kuvassa 6 esitetään, miten poistovaiheessa käyttäjän tietokantaan tallentamat suosikit käydään läpi ja poistetaan.

```
@Component(immediate = true, service = ModelListener.class)
public class UserDeletionListener extends BaseModelListener<User> {

    private static Log _log = LogFactoryUtil.getLog(UserDeletionListener.class);

    @Reference
    private UserFavoriteLocalService userFavoriteLocalService;

    @Override
    public void onAfterRemove(User user) throws ModelListenerException {
        // Lisätään tänne metodit, joilla poistetaan käyttäjän omiin toimiin liittyvä data, kun käyttäjä poistetaan.
        try {
            List<UserFavorite> userFavorites = userFavoriteLocalService.getUserFavoritesByUserId(user.getUserId());
            for (UserFavorite userFavorite : userFavorites) {
                userFavoriteLocalService.deleteUserFavorite(userFavorite);
            }

        } catch (Exception e) {
            _log.error( msg: "Error deleting subscriptions from deleted user " + e);
            throw new ModelListenerException(e);
        }
        super.onAfterRemove(user);
    }
}
```

KUVA 6. Liferayn tarjoaman kuuntelijaluokan laajennus, jolla poistetaan poistettavan käyttäjän tallentamat omat suosikit.

Osoitteiden haussa käytetystä lähestymistavasta aiheutui haasteita, kun työn tilaaja halusi mahdollistaa myös yksittäisten uutisten ja vastaavien sisältöjen lisäämisen suosikkilistaukseen. Liferay näyttää eri uutiset samalla sivupohjalla, joten näiden yksilöimiseksi suosikeille tarvittiin kuitenkin erillinen URL:n sisältävä kenttä *urlKey*.

updateUserFavoritesOrder: Tämä metodi päivittää käyttäjän kaikkien suosikkien järjestysindeksin. Sitä kutsutaan, kun käyttäjä uudelleen järjestelee listaus-

tastaan. Tällöin metodi saa frontendistä parametrina yhden suosikin, jonka järjestysindeksi on muuttunut. Tämän perusteella metodi osaa antaa kaikille käyttäjän suosikeille uuden oikean indeksin.

getUserFavorite: Metodi palauttaa persistanssitason kautta tehtävän tietokantahaun perusteella käyttäjän suosikin sivupohjan *id*:n ja URL:n perusteella. Metodia käytetään näyttämään, onko sivu suosikeissa, kun käyttäjä saapuu sivulle. Jos on, niin suosikki on valmiiksi tallessa mahdollista päivityskutsua, tai poistoa varten.

addUserFavorite: Metodi lisää uuden suosikin käyttäjän suosikkilistaukseen. Se ottaa parametreina käyttäjän suosikille antaman nimen, sivupohjan *id*:n, URL:n ja *ThemeDisplay*-olion. Tämä olio tarjoaa lukuisia tietoja käyttäjän istunnon tilasta rajapintakutsun hetkellä. Tässä sitä käytetään yksinkertaisesti hakemaan käyttäjän *id*. Metodi luo uuden *UserFavorite*-olion, asettaa sille tarvittavat attribuutit ja tallentaa sen tietokantaan.

deleteUserFavorite: Tämä metodi poistaa suosikin käyttäjän suosikkilistauksesta. Se ottaa parametrina suosikin *id*:n ja istunnon tiedot *ThemeDisplayn* muodossa. Metodi tarkistaa, että kutsun lähettäjä on suosikin omistaja, ja poistaa sitten suosikin tietokannasta.

deleteUserFavoritesByUrlKey: Metodi poistaa kaikki käyttäjän suosikit tietyn URL-avaimen perusteella. Se hakee ensin kaikki suosikit, jotka vastaavat annettua URL-avainta, ja poistaa ne sitten tietokannasta. Tätä metodia käytetään, kun uutissisällön tila vaihtuu julkaistusta joksikin muuksi (eli se ei ole enää käyttäjälle esillä), jolloin suosikkilinkki jäisi orvoksi.

getUserFavoritesByLayoutId: Metodi palauttaa kaikkien käyttäjien suosikit tietyn sivupohjan:n perusteella. Tätä hyödynnetään, kun kyseinen sivupohja poistetaan Liferay-sovelluksesta, orpojen suosikkien muodostumisen ehkäisemiseksi.

3.3 Userfavorites-api-moduuli

Userfavorites-api-moduuli tarjoaa rajapinnan sovelluslogiikan ja käyttöliittymän väliseen kommunikaatioon. Tämä moduuli sisältää pääasiassa eri rajapintaluokkia (Java interface), jotka määrittelevät metodit ja toiminnot sekä niihin tarvittavat parametrit, joita käyttöliittymä voi käyttää vuorovaikutukseen backendin kanssa. Liferayn Service Builder luo rajapintaluokat tarvittaessa uudestaan service-moduulin muutosten jälkeen.

Userfavorites-api-moduuli on abstraktio backendin toteutuksesta ja sen avulla käyttöliittymää ja sovelluslogiikkaa voidaan kehittää tehokkaammin erikseen varmistamalla, että oleellinen arkkitehtuurikerrosten välissä tarvittava informaatio on tarkoin määritelty. Tämä mahdollistaa myös modulaarisen ja suurempaan mittakaavaan skaalautuvan kehitystyön. (Malaska & Seldman 2018, 4. Interface Design)

4 REST-RAJAPINTA FRONTENDILLE

4.1 Rest-rajapinnan toteutuksen rakenne

Rajapinnan pohja toteutettiin seuraavilla Java-luokilla, jotka lisättiin Liferayn komponenteiksi *@Component* annotaatiolla:

***IntraRestApplication*:** Luokka määrittelee REST-palvelun ja sen juuripolun */intra-rest/api* sekä rajapinnan käyttäjien varmentamisessa käytettyä autorisaatiota ja autentikointia.

***ThemeDisplayContextProvider*:** Luokka tarjoaa kulloisenkin HTTP-kutsun kontekstista *ThemeDisplay*-olion, jonka kautta backend saa tietoa kutsun aikaisesta-frontend-istunnon tilasta.

***LiferayEventProcessorFilter*:** Luokka tarjoaa mahdollisuuden reagoida kutsuun ennen kuin se etenee varsinaiseen REST-rajapintaan. Tätä voisi hyödyntää muun muassa pääsynvalvonnassa tai lokitietojen tallentamisessa ja tarkkailemisessa.

***IntraRestExceptionMapper*:** Luokka käsittelee määritellyt virheet ja poikkeukset REST-kutsuissa, joita ei ole erikseen määritelty rajapinnan päätepisteiden toiminnallisuuksien yhteydessä.

***UserFavoriteResource*:** Luokka on varsinainen REST-resurssin toteutus. Se tarjoaa frontendille päätepisteet, joiden kautta päästään käsiksi backendin toimintalogiikkaan.

Näin toteutettuna rajapinnan pohja mahdollistaa toiminnallisuuden helpon laajentamisen muihin tarvittaviin rajapintoihin ja välttää toisteisen koodin kirjoittamista.

4.2 UserFavorite-resurssi

UserFavorite-resurssi toteutettiin melko puhtaana rajapintana, varsinaisen toimintalogiikan sijaitessa userFavoriteLocalService-luokassa. Luokan yleinen polku määriteltiin *@Path("user-favorites")* -annotaatiolla ja sitä tarkennettiin tarvittaessa yksittäisten päätepisteiden määrityksissä. Frontendista tarvittavat tiedot pyrittiin hankkimaan pääasiallisesti kutsun kontekstista saatavasta *ThemeDisplay*-luokasta, mutta myös polkuparametreja hyödynnettiin tässä tarkoituksessa. Näin esimerkiksi suosikin käyttäjältä poistavan päätepisteen koodiksi muodostui kuvassa 7 nähtävä toteutus

```
@DELETE
@Path("/{id}")
public Response deleteFavorite(@PathParam("id") long userFavoriteId,
                               @Context ThemeDisplay themeDisplay) throws PortalException {
    try {
        userFavoriteLocalService.deleteUserFavorite(userFavoriteId, themeDisplay);
        return Response.status(Response.Status.NO_CONTENT).build();
    } catch (NoSuchUserFavoriteException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

KUVA 7. UserFavorite-resurssin päätepiste omien suosikkien poistamiseksi tietokannasta.

Polkuparametrina otetaan vastaan suosikin *id*. Service-luokalle lähetetään parametrina lisäksi *themeDisplay*-olio, josta tarkastetaan, onko kutsun suorittaja sama kuin poistettavan suosikin käyttäjä. Jos poisto service-luokassa onnistuu, palautetaan vastauksena *status 204 – ei sisältöä*, joka on tyypillinen palautus poiston yhteydessä (Sharma S 2023, Exploring HTTP methods and status codes). Jos suosikkia ei kutsutulla *id*:llä löydy, tai käyttäjä on väärä, palautetaan *status 404*. Tässä hyödynnetään Service Builderin automaattisesti luomaa valmista poikkeusluokkaa *NoSuchUserFavoriteException*.

Muut tarvittavat päätepisteet ovat *getFavorite*, *getFavorites*, *addFavorite* ja *updateFavorite*. Ne on toteutettu hyvin samankaltaisella logiikalla kuin esimerkin päätepiste ja niiden käyttötavat tulevat esille työn frontend-osiossa.

5 REACT FRONTEND

5.1 React-portlettien luominen Liferay-ympäristöön

Toiminnallisuuden frontend toteutettiin kolmella React-portletilla, eli kolmella erillisellä Liferayn sisällä ajettavalla React-sovelluksella. Sovellukset olivat jokaiselle sivuston sivulle lisättävä nappi, jolla sivun voi lisätä omiin suosikkisivustoihin, suosikkisivujen listausnäkymä sekä suosikkisivujen hallinnointinäkymä. Sovellukset toteutettiin Liferay-projektin yhteyteen siten, että ne voi ottaa admin-oikeuksilla käyttöön sivuston hallintanäkymästä ja sijoittaa mille tahansa sivulle (layouttiin) itsenäisesti toimivana portlettina.

React sovellukset luotiin Liferay JS Generator -sovelluksella Liferay Help Center sivuston ohjeistuksen mukaan (Liferay Help Center, Developing a React Application). Sovelluksen voi asentaa npm:llä, eli JavaScript-projektien ohjelmistopakettien hallintaan tarkoitetulla Node Package Managerilla, ja ajaa komennoilla:

```
npm install -g yo@4.3.1 generator-liferay-js
```

ja

```
yo liferay-js
```

Sovellus kysyy seuraavat asiat:

```
? What type of project do you want to create? React Widget
? What name shall I give to the folder hosting your pro-
ject?
reactWidgets/quickLinksButton
? What is the human readable description of your project?
Quick Links Button
? Do you want to add localization support? Yes
? Do you want to add configuration support?
 No
? Under which category should your widget be listed?
category.reactWidgets
? Do you have a local installation of Liferay for develop-
ment? Yes
? Where is your local installation of Liferay placed?
../../bundles
? Do you want to generate sample code? Yes
```

Kysymysten tämän projektin sovellusten luomisessa käytetyt vastaukset on kirjoitettu punaisella värillä kysymyksen perään.

Sovelluksella voisi luoda myös mm. paljaan JavaScript- tai Angular-sovelluksen, mutta valitaan tässä tapauksessa *React Widget*, jolloin sovellus rakentaa `package.json` tiedoston, joka sisältää ohjeet Liferayssä ajettavan React-sovelluksen tarvitsemista riippuvuuksista ja versionumeroista, jotka ovat oleellisia kun otetaan käyttöön esimerkiksi valmiita ulkopuolisia kirjastoja. *Localization support* luo valmiit tiedostot kielistystä varten ja konfiguraatiotuki loisi tiedostot, joilla voisi luoda portletille Liferayn graafisen admin-käyttöliittymän kautta hallittavia konfiguraatiovaihtoehtoja. *Category.reactWidgets* valinta luo samaisen käyttöliittymän valikkoon kansion, jonka alta portletin lisääminen sivustolle on mahdollista. Tieto Liferayn installaation sijainnista on oleellinen että *npm build* -komento osaa sijoittaa React-sovelluksesta luodun paketin Liferayssa ajettaviin OSGi-moduuleihin. Uutta sovellusta luotaessa esimerkkikoodin luominen auttaa ymmärtämään Liferayssä ajettavan React-sovelluksen harvoja erityispiirteitä.

5.2 React-frontend Liferayssä

Jotta React-sovellus osataan sijoittaa oikein Liferay-sivustolla, annetaan render-funktiolle parametrina tyypillisen root-elementin sijaan haluttu portlet-elementti, jolloin index.js-tiedostosta tulee yksinkertaisimmillaan seuraava:

```
Import React ym…
Export default function main({portletElementId}){
ReactDOM.render(<AppComponent />, document.getEle-
mentById(PortletElementId));}
```

Jossa `<AppComponent />` pitää sisällään koko sovelluksen, eikä tarvitse enää mitään pakollisia viitteitä Liferayhin.

Saman Liferay-sovelluksen backendiin tehdyt api-kutsut autorisoidaan frontendistä löytyvän Liferay-objektin *authTokenin* avulla, jolloin kutsut rajapintaan ovat yksinkertaisimmillaan seuraavanlaisia:

```
Fetch('/o/intra-rest/api/user-favorites?p_auth=${Lif-
eray.authToken}')
```

Backendiin voi toki lähettää kutsun mukana muita tietoja frontendistä esimerkiksi *query*- tai *path*-parametreina. Yksinkertaisimmin backend kuitenkin saa suurimman osan tarvituista tiedoista, kuten kirjautuneen käyttäjän *id*:n, kutsun yhteydessä *request atribuuttina* haettavasta *ThemeDisplay*-objektista, joka pitää sisällään suuren määrän käyttäjäkohtaista tietoa frontend-istunnon tilasta kutsun hetkellä. Samaan *ThemeDisplay*-objektiin pääsee käsiksi myös frontend-koodin puolella kutsulla *Liferay.ThemeDisplay*.

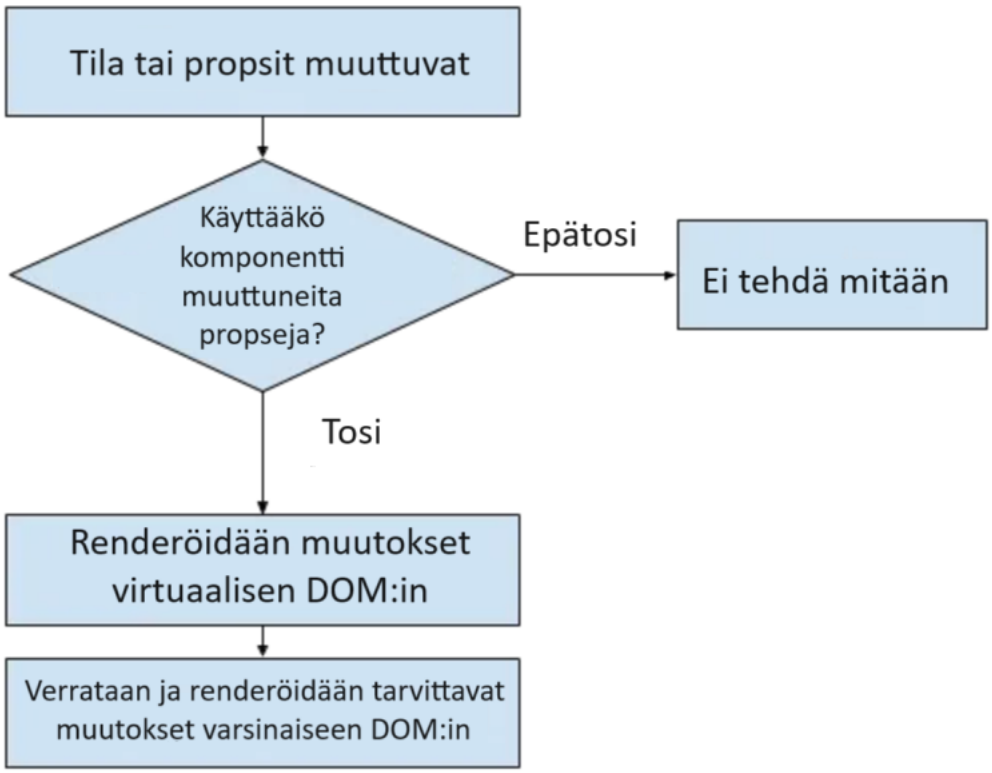
5.3 Sovelluksessa käytettyjen React-ominaisuuksien teoriaa lyhyesti

Projektissa toteutetut React-sovellukset olivat kaikki verrattain yksinkertaisia, mutta niissä hyödynnettiin kuitenkin muutamaa Reactin toiminnalle keskeistä ominaisuutta. React-sovelluksessa informaatiota voidaan välittää ylemmältä komponentilta alemmille komponenteille propseina. Propsit ovat React-komponenttien ominaisuuksia, jotka välitetään niille ylemmän tason komponenteilta. Ne mahdollistavat esimerkiksi samojen komponenttien uudelleen käytön eri tietosisällöillä. Lisäksi, kun propsit muuttuvat, React havaitsee nämä muutokset ja pakottaa uudelleen renderöinnin kaikille komponenteille, jotka käyttävät näitä muuttuneita props-arvoja. (Banks & Porcello 2017, 4. Pure React, DOM Rendering.) Tässä projektissa data haettiin backendistä ylemmän tason komponenteista ja välitettiin propseina alemman tason komponenteille. Näin kun backendistä haetaan data muutosten takia uudelleen, propseina käytettyjen muuttujien sisältö muuttuu, mikä puolestaan aiheuttaa myös alempien komponenttifunktioiden uudelleen suorittamisen ja renderöinnin uudella datalla. Näin kaikkien komponenttien tila on yhteydessä oikealla tavalla ja pysyy ajantasaisena. Kuviossa 4 on esitetty Reactin toimintalogiikka DOM:in päivittämiseen muuttuneen tilan ja propsien suhteen. DOM (Document Object Model) tarkoittaa ohjelmointirajapintaa, jonka

kautta verkkosivun näkymään voidaan tehdä muutoksia JavaScriptillä (MDN Web Docs 2023, Document Object Model).

Projektin React-sovelluksissa käytettiin kolmea React-koukkua. *UseEffect*-koukkua hyödynnettiin esimerkiksi seuraavasti:

```
useEffect(() => {
    accordionActive && getQuickLinks();
}, [accordionActive]);
```



KUVIO 4. Kun komponentin tilaksi määritellyt muuttujat tai sen käyttämät propsit muuttuvat, React uudelleenarvioi komponentit ja renderöi ne Reactin virtuaalisen DOM:in. Tämän jälkeen virtuaalista DOM:ia verrataan varsinaiseen DOM:in, jonne päivitetään tarvittavat muutokset.

Tässä listauksessa käytetyn haitarinäkymän tilasta kertova *accordionActive* tilamuuttuja toimii sekä koukun riippuvuutena, että koukun sisältämän funktion logiikassa. *getQuickLinks*-funktio hakee omat suosikit backendistä. Tässä *useEffect*-koukku toimii kuuntelijan tavoin ja sen sisältämä funktio suoritetaan vain ja ainoastaan kun koukun riippuvuus *accordionActive* muuttuu (Sakhniuk & Boduch

2024, 3. Understanding React Components And Hooks, React Hooks). Suoritettaessa kyseinen *useEffect*-koukku tarkistaa ensin onko haitari aktiivinen (*accordionActive)* ja jos on, hakee se omat suosikit backendista (*getQuickLinks()*).

UseState-koukkua hyödynnettiin esimerkiksi listausten suodattamisessa hakutekstin mukaan. Hakuteksti asetettiin tilaksi:

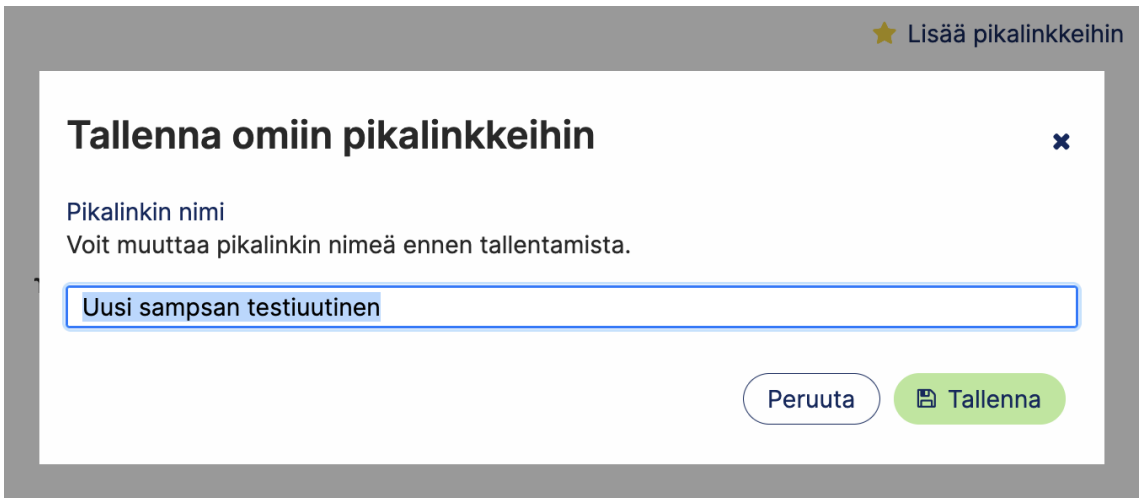
```
const [filterValue, setFilterValue] = useState("");
```

jolloin aina hakukentän syötteen muuttuessa *filterValue*-tila asetetaan uudestaan *useState* koukun yhteydessä aina määriteltävällä set-funktiolla *setFilterValue*. Tilan muuttuessa komponenttifunktio suoritetaan uudelleen (Sakhniuk & Boduch 2024, 3. Understanding React Components And Hooks, React Hooks). Näin komponentti renderöidään uudelleen käyttäen uutta *filterValue*-arvoa ja vain halutut omat suosikit tulevat loppukäyttäjälle näkyviin.

UseRef-koukku tarjoaa tavan säilyttää tietoa komponentin uudelleen evaluointien välillä. Sitä käytetään tyypillisesti, kuten tässäkin sovelluksessa, viittamaan käyttäjän tekstisyötekenttään syöttämään arvoon. Tämän referenssimuuttujan muutokset eivät aiheuta komponentin uudelleen suorittamista, mutta se mahdollistaa tarpeen tullen esimerkiksi ajantasaisen tekstisyötetiedon lähettämisen backendiin. (Larsen 2021, 5. Managing component state with the useRef hook.)

5.4 Sovelluksen frontend portletit ja niiden logiikka

Ensimmäinen toteutettu portletti oli määritelty asiakkaan toimesta seuraavasti: Jokaisella sivulla näytetään nappi, jolla sivun voi lisätä omiin suosikeihin tai poistaa sen. Napin kuvake ja teksti muuttuu suoritettavan toimenpiteen mukaan ja lisätessä aukeaa modaali, jossa varmistetaan lisäys ja voidaan halutessa syöttää suosikille oma nimi, oletusarvona dokumentin *title*-objektista saatavan nimitiedon sijaan (Kuva 8).



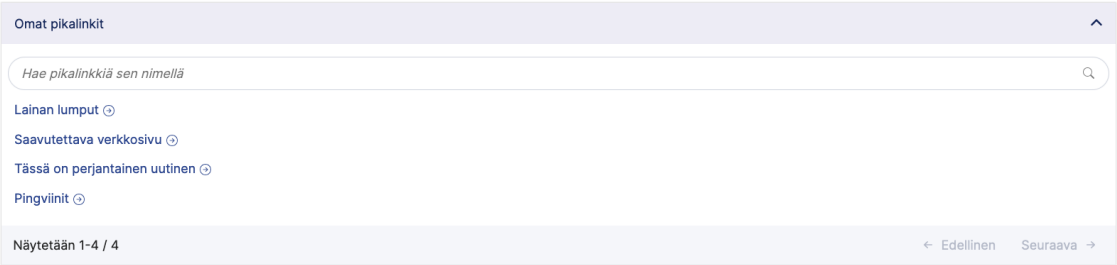
KUVA 8. *Lisää pikalinkkeihin* nappi ja sen painamisen seurauksena avautuva tallennusikkuna.

Pienoissovelluksen logiikka toteutettiin seuraavasti: Sovellus hakee latautuessaan *ThemeDisplay*-objektia hyödyntäen backendistä tiedon löytyykö kyseinen sivu jo käyttäjän suosikeista (rajapinnan *getFavorite* päätepiste). Jos löytyy, niin suosikin tiedot, käyttäjän antama suosikin nimi ja sen *id*, otetaan talteen sovelluksen tilaan. Suosikin poisto onnistuu tällöin lähettämällä backendiin DELETE-kutsu parametrinaan suosikin *id*. Modaalin tekstikentän tieto säilytettiin useRef-koukun avulla backend-kutsun mukana lähetettävässä muuttujassa. Suosikkia lisättäessä backendiin lähetetään POST-kutsu, joka tarvitsee parametrinaan vain käyttäjän antaman suosikin nimen, muut tiedot saadaan jälleen *ThemeDisplay*-objektista.

Toinen portletti oli määritelty omien suosikkien käyttäjäkohtaiseksi listaukseksi, joka olisi sivutettu ja joka mahdollistaisi listauksen suodattamisen hakutekstin perusteella. Testiympäristöön asennettu valmis pienoissovellus on esitetty kuvassa 9.

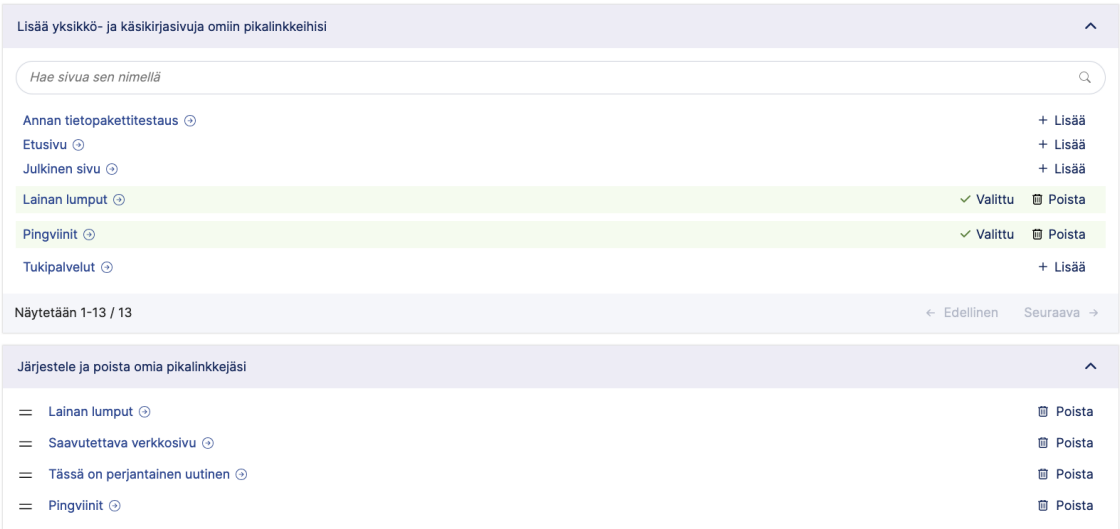
Sovellus oli kohtalaisen yksinkertainen toteuttaa ja React toimii tehokkaasti haluttujen kaltaisten toiminnallisuuksien toteuttamisessa. Huomionarvoista on, että sovellus toteutettiin siten, että omat suosikit haetaan vain kerran backendistä komponentin *mountin* yhteydessä. Tämän voi tehdä esimerkiksi suorittamalla GET-kutsu ylemmän tason komponentissa ja välittämällä data propsina rende-

röintilogiikkaa suorittavalle komponentille. Suodatus ja sivutus toteutettiin kahdella *useState*-tilalla, joiden muuttuessa komponentti uudelleen renderöidään uusi tila huomioiden.



KUVA 9. Näkymä omien suosikkien listaukseen käytetystä Reactilla toteutetusta pienoissovelluksesta.

Kolmas portletti määriteltiin omien suosikkien hallintaa varten. Se koostuisi kahdesta komponentista, joista ylempi sisältäisi tietyllä tavalla Liferay-sovelluksessa kategorisoidut sivut, jotka voisi näkymän kautta suoraan lisätä omiin suosikeihin tai poistaa niistä. Toinen komponentti sisältäisi kaikki omat suosikit näyttävän listauksen, mutta tässä näkymässä suosikkien järjestystä listaus-portletissa voisi muokata *vedä ja pudota* -tavalla. Kuvassa 10 on esitetty valmis pienoissovellus asennettuna testiympäristöön.



KUVA 10. Näkymä omien suosikkien lisäämiseen, poistamiseen ja muokkaamisen tarkoitetusta pienoissovelluksesta.

Koska kahden komponentin tilat liittyivät kiinteästi toisiinsa, esim. ylemmässä komponentissa suosikin lisäyksen tulee näkyä heti myös alemmassa komponentissa ja toisinpäin, oli tehokkainta hallita molempien komponenttien yhteistä tilaa ylemmän tason komponentissa. Näin muutokset saadaan tarvittaessa näkymiin molempiin komponentteihin propsien välityksellä. Portletin backendiin yhteydessä olevat metodit toteutettiinkin yhtä lukuun ottamatta ylemmän tason komponentissa, koska ne olivat suurilta osin samoja molemmilla komponenteilla. Suosikkien uudelleenjärjestely toteutettiin valmiilla *react-beautiful-dnd* kirjastolla. Jokaisen siirron päätteeksi backendiin lähetetään PUT-kutsu (rajapinnan *updateFavorite* päätepiste), joka sisältää *dnd*-kirjaston laskeman uuden järjestysindeksin suosikille. Sovelluksen backend huolehtii logiikasta muiden suosikkien muuttuvien indeksien osalta.

5.5 Saavutettavuus

Koska kyseessä oli intranet-sivusto, ei saavutettavuutta pidetty projektin prioriteettilistan kärjessä, mutta se täytyi kuitenkin oleellisilta osin huomioida frontend-kehityksessä. Portletteja oli pystyttävä käyttämään näppäimistön avulla. Tämä aiheutti lisätyötä esimerkiksi pikalinkkinapin modaalin toteutuksessa, kun kohdistus tuli modaalin auetessa siirtää sen ensimmäiseen kohdennettavaan elementtiin ja suljettaessa takaisin siihen, mihin kohdistus oli jäänyt. Erityisesti näytönlukijoiden toimivuuden varmistamiseksi oikeaoppisesti muodostettu html-koodi oli oleellista. Tämä tarkoittaa esimerkiksi otsikkoelementtien käyttöä niin, että otsikkotasot h1, h2, jne. seuraavat loogisesti toisiaan. Tämä siksi, että ruudunlukijoiden käyttäjät voivat käyttää otsikkotasoja sivulla navigointiin (Cunningham 2012, Creating Accessible Sites). Lisäksi suunnitteluvaiheessa otettiin huomioon, että tekstin ja taustan kontrasti on riittävä laajalle käyttäjäkunnalle ja että käytetyt fontit ja kuvakkeet ovat riittävän suuria ja selkeitä.

6 POHDINTA

Projektin tullessa päätökseen on tärkeää pohtia onnistumisia, haasteita ja kehitysideoita seuraavia projekteja varten. Suurimmat haasteet tässä projektissa aiheutuivat omien suunnitteluratkaisujen yhteensovittamisesta Liferayn toiminnallisuuksien kanssa. Myös Liferayn käyttämä React-versio 16 on kohtalaisen vanha ja aiheutti ongelmia valmiiden React-kirjastojen hyödyntämisessä.

Suurin yksittäinen hidaste oli yksittäisten uutissisältöjen lisääminen suosikeiksi. Suunnitteluratkaisu oli alun perin tallentaa suosikki tietokantaan sivupohjan yksilöivän *id*:n perusteella ja hakea suosikkia frontendiin palauttaessa sivupohjan ajantasainen URL Liferayn valmiita metodeja käyttäen. Uutissisällöt käyttivät kuitenkin samaa sivupohjaa, uutisten näyttämiseen luotua näyttösivumallia, jolloin näitä varten täytyi erikseen lisätä suosikin tallennus URL:n perusteella (Liferay Learn n.d., Using Display Page Templates). Tämä vaati lisätöitä frontendin, backendin sekä tietokannan tasolla. Vaihtoehtoinen toteutustapa uutissisältöjen osalta olisi ollut suosikkia tallentaessa selvittää backendissä URL:n perusteella mikä uutissisältö sivulla on ja yksilöidä suosikki sen perusteella. Palauttaessa suosikkia, URL voitaisiin rakentaa samalla periaatteella kuin sivupohjan tapauksessa, jolloin toiminto ei olisi altis Liferay-kontekstissa mahdollisesti muuttuville URL:lle. Joka tapauksessa ominaisuuden toteutus olisi ollut tehokkaampaa ja yhtenäisempää jos tämä erityispiirre olisi osattu ottaa huomioon jo suunnitteluvaiheessa (Indeema n.d., 3.4 A good plan helps you manage risks better.)

Suosikkilistaus tuli asiakkaan sivustolla käyttöön sivulle, jossa oli esillä muitakin käyttäjäkohtaisia sisältöjä. Tällöin sivun latauksessa esiintyi hitautta, jota varten lisätyönä toteutettiin listaussovelluksen optimointia siten, että jo ennestään haitarinäkymässä olevat suosikit ladattiin backendistä vasta haitarinäkymää avattaessa. Tämä sekä nopeutti sivun yleisnäkymän latautumista, että vähensi alustan kuormitusta, kun mahdollisesti turhia backend-käsittelyjä voitiin näin välttää. React-sovellusta kehittäessä kannattaa kiinnittää huomiota koukkujen käyttöön, koska ne aiheuttavat usein komponentin ja näkymän uudelleen evaluointeja ja renderöintejä. Vaikka React onkin tehokas laskiessaan mitä muutoksia näkymään tarvitsee tehdä, kannattaa esimerkiksi useState-koukkujen sijaan käyttää

tavallisia muuttujia silloin kun se on mahdollista, eli kun muuttujan muutoksen seurauksena näkymää ei tarvitse päivittää (Chavan 2020).

Esimerkiksi haitareiden ja modaalien toteuttamisessa olisi mahdollisesti voinut nopeuttaa kehitystyötä käyttämällä oman toteutuksen sijaan valmiita komponentteja esimerkiksi Material UI:lta (MUI Core n.d.). Liferayn React-version aiheuttamat yhteensopivuusongelmat kuitenkin tekivät tämän lähestymistavan haastavaksi ja tällaiset komponentit olivat lopulta melko yksinkertaisia toteuttaa käsityönä. Suosikkien uudelleenjärjestely *raahaamalla ja tiputtamalla* pystyttiin onneksi toteuttamaan valmiilla react-beautiful-dnd kirjastolla, koska sen toteutus itse olisi ollut huomattavasti työläämpi.

React-portlettien toteutuksessa olisi ollut mahdollista käyttää JavaScriptin sijaan TypeScriptiä. Sen käyttöönotto JavaScriptin sijaan vaatii kehittäjiltä mahdollisesti hieman opiskelua, mutta tuo mukanaan hyötyjä kuten staattisen tyypityksen JavaScriptin dynaamisen sijaan, mahdollisten null-arvojen tarkastuksen jo IDE:ssä sekä esimerkiksi rajapintoihin nojaavan toteutuksen tuoman refaktoroinnin helppouden (Jansen, Vane & de Wolff 2016, 1. Introducing TypeScript). Kaiken kaikkiaan TypeScriptin lisäominaisuudet ryhdittävät kehitystyötä ja vähentävät ajonaikaisten virheiden mahdollisuutta. Tämän kokoluokan projektissa JavaScript voi olla myös hyvä valinta, mutta React-kehitystyötä aloittaessa tämä pohdinta on aina hyvä käydä läpi.

Kaupallisen työn aiheuttamat aikataulupaineet vaikuttivat kaikkeen tekemiseen, ja työskentelytavoissa oli tärkeää löytää tasapaino tehokkuuden ja laadun välillä. Tässä auttavat tarkoista määrittelyvaatimuksista huolehtiminen sekä selkeä vastuunjako testaamistyössä ja laadunvarmistuksessa asiakkaan ja toimittajan välillä.

Ominaisuuden toteutus noudattaa MVC-arkkitehtuurin mukaisia käytänteitä, jolloin sovelluksen osat ovat teoriassa mahdollista vaihtaa esimerkiksi kokonaan toisella ohjelmointikielellä toteutetuiksi. Tätä tukemaan tarvittaisiin hyvä dokumentaatio siitä, mitä tietoa rajapinnoilla otetaan vastaan ja missä muodossa sen on oltava. Esimerkiksi, ottavatko frontendin metodit datan vastaan backendin REST-rajapinnasta JSON- tai XML-muodossa jne.

Tietoturva on tärkeä osa jokaisen verkkosovelluksen kehitystä, ja Liferay tarjoaa monia automaattisia ratkaisuja tähän liittyviin kysymyksiin. Esimerkiksi REST-rajapinnan tietoturva voitiin hoitaa Liferayn omilla tietoturvamäärityksillä. Kuitenkin kehittäessä sovelluksen sisäistä sovellusta oli tärkeää huolehtia mm. syötteiden validoinnista oikeiksi ja tietoturvallisiksi sekä pääsynhallinnasta niin, että käyttäjä näkee vain hänelle kuuluvia tai hänen nähtäväkseen tarkoitettuja tietoja.

Kaiken kaikkiaan uuden ominaisuuden toteutus onnistui hyvin ja se saatiin toimitettua asiakkaalle käyttöön kohtuullisessa ajassa ja vaatimusmäärittelyn mukaisena.

LÄHTEET

Banks, A. Porcello, E. 2017. Learning React: Functional Web Development with React and Redux. United States: O'Reilly Media, Inc.

Chavan, Y. 2020. When it's not good to use state for storing data in React. Medium. Viitattu 25.4.2024. https://blog.devgenius.io/when-its-not-good-to-use-state-for-storing-data-in-react-adcf261e8467

Cunningham, K. 2012. Accessibility Handbook. United States. O'Reilly Media, Inc.

Gennick, J. Mishra, S. 2001. Oracle SQL*Loader: the definitive guide. United States: O'Reilly Media, Inc.

Indeema. 10 Points of Importance of Project Planning & Steps to Build a Plan. Viitattu 25.4.2024. https://indeema.com/blog/the-importance-of-project-planning

Jansen, R. Vane, V. de Wolff I. 2016. TypeScript: Modern JavaScript Development. United Kingdom: Packt Publishing.

Larsen, J. 2021. React Hooks in Action. United States: Manning Publications.

Liferay Help Center. Developing a React Application. Viitattu 25.4.2024. https://help.liferay.com/hc/en-us/articles/360029028051-Developing-a-React-Application

Liferay Help Center. DXP 7.2 Developer Tutorials. Generating the Back-end. Verkkosivu. Viitattu 19.4.2024. https://help.liferay.com/hc/en-us/articles/360032887592-Generating-the-Back-end

Liferay Help Center. Fundamentals. Verkkosivu. Viitattu 19.4.2024. https://help.liferay.com/hc/en-us/articles/360018163311-Fundamentals

Liferay Learn. Using Display Page Templates. Viitattu 25.4.2024. https://learn.liferay.com/w/dxp/site-building/displaying-content/using-display-page-templates

Malaska, T. Seldman, J. 2018. Foundations for Architecting Data Solutions. United States: O'Reilly Media, Inc.

MDN Web Docs. Document Object Model, Introduction to the DOM. Viitattu 5.5.2024. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

MUI Core. Material UI – Overview. Viitattu 25.4.2024. https://mui.com/material-ui/getting-started/

Sakhniuk, M. Boduch, A. 2024. React and React Native – Fifth Edition. United Kingdom: Packt Publishing.

Sharma, S. 2023. Modern API Development with Spring 6 and Spring Boot 3 - Second Edition. UK: Packt Publishing.

Taylor, R. Medvidovic, N. Dashofy, E. 2009. Software Architecture: Foundations, Theory, and Practice. USA: Wiley.