KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutus

Markku Vaara

MUSIIKKITIETOKANTA JA KÄYTTÖLIITTYMÄ

Opinnäytetyö
Marraskuu 2021

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Marraskuu 2021
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä(t)
Markku Vaara

Nimeke
Musiikkitietokanta ja käyttöliittymä

Toimeksiantaja
Joensuun seutukirjasto

Tiivistelmä

Tämän opinnäytetyön tarkoituksena oli Joensuun seutukirjaston toimeksiannosta suunnitella ja toteuttaa pohjoiskarjalaisten musiikintekijöiden tietoja sisältävä musiikkitietokanta. Lisäksi oli tarkoituksena suunnitella ja rakentaa käyttöliittymä, jolla tämän musiikkitietokannan tietoja voidaan tarkastella ja käsitellä.

Opinnäytetyön teoreettisessa osassa käydään läpi toimeksiannon toteuttamiseen liittyvät työkalut ja menetelmät. Toteutusosiossa puolestaan käydään seikkaperäisesti läpi käytännön toteutukseen liittyvät työvaiheet. Sovellus toteutettiin perinteisenä verkkoselainsovelluksena. Käyttöliittymän ulkoasu suunniteltiin Balsamiq-mockup-ohjelmalla, ja käyttäjärajapinnan ulkoasu ja toiminnallisuudet rakennettiin Javascript-kielellä React-sovelluskehystä hyödyntäen. Palvelinpuoli puolestaan rakennettiin Javascriptillä Express-palvelinkehystä hyödyntäen, ja itse tietokanta toteutettiin MongoDB-tyypin dokumenttitietokantana. Tällaisen tietokannan käyttö vaati enemmän sovelluspuolen ohjelmointia, mutta toisaalta antoi lisää joustavuutta tietokantatietueiden luomiseen ja käsittelyyn.

Opinnäytetyöskentelyn lopputuloksena syntynyt sovellus toimi kohtuullisen hyvin. Kaikki vaatimusmäärittelyissä vaadittavat toiminnot oli saatu koodattua käyttöliittymään, aina luetteloinnista ja hakumekanismeista ylläpitopuolen toiminnallisuuksiin. Myös palvelintaso ehdittiin koodata, ja se toimi hyvin käyttöliittymän ja testitietokannan kanssa, joskin joitain tietoturvatoimintoja jäi ylläpitopuolella ajan puutteen vuoksi tekemättä. Myös itse varsinainen tietokanta jäi pystyttämättä, mutta koska koko sovellus on muuten enimmäkseen valmis, lopullinen musiikkitietokannan luominen ei tule olemaan suuritöinen projekti.

Kieli
suomi

Sivuja 49

Asiasanat
tietokanta, palvelin, verkko-ohjelmointi, Javascript, React

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
November 2021
Degree Programme in Business Information Technology

Tikkarinne 9
80200 JOENSUU FINLAND
+ 358 13 260 600 (switchboard)

Author (s)
Markku Vaara

Title
Music Database and its Interface Application

Commissioned by
Joensuu Regional Library

Abstract

The aim of this thesis was on behalf of the Joensuu Regional Library to design and create a music database containing information about musicians in the North Karelian region. A user interface that operates this music database through a server application was also developed.

The theoretical part of this thesis familiarizes the reader with the basics of the tools and methods used in the development of the application. The practical part of this thesis, on the other hand, explains in detail the development process of the same web application. The user interface was designed with the Balsamiq mock-up web application, and then it was created as a traditional web browser app with Javascript programming language utilizing the React framework. The server side of the application was created with Javascript using the Express framework as a server engine. The database itself was implemented as a MongoDB-type document database. This document model approach required more application-side programming to operate the database, but on the other hand provided more flexibility in creating and operating database records.

The result of this thesis was a fully functional single-page user interface application, and an almost fully functional server application. Due to the lack of time, the actual database was not implemented, but a test MongoDB database was used to test both the interface and the server. The overall application (the interface, the server, and the test database) worked quite nicely together and functioned as required. It also provided an excellent base for the actual music database implementation and further application development.

Language
Finnish

Pages 49

Keywords
database, server, web development, Javascript, React

Sisältö

Käytetyt termit ja lyhenteet

API	Application Programming Interface eli ohjelmointirajapinta, jonka avulla ohjelmat voivat tehdä pyyntöjä ja vaihtaa tietoja keskenään.
CSS	Cascading Style Sheet eli ohjelmointikieli/tekniikka, jolla määritellään ulkoasu HTML-kielellä kuvatulle rakenteelle.
Docker	Avoimen alustan sovellus, joka pystyy pakkaamaan sovelluksen kaikkien tarvittavien riippuvuuksien kanssa yhteen virtuaaliympäristöön, eli niin kutsuttuun konttiin.
DOM	Document Object Model eli dokumenttioliomalli on tapa kuvata rakenteinen ja/tai hierarkkinen dokumentti.
Express	Vapaan lähdekoodin Javascript-kirjasto palvelinsovelluksen luontia varten.
HTML	HyperText Markup Language eli standardoitu merkintäkieli, jolla voidaan kuvata hyperlinkkejä sisältävää tekstiä eli hypertekstiä.
HTTP	HyperText Transfer Protocol eli hypertekstin siirtoprotokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
JSON	JavaScript Object Notation eli avoimen standardin tiedostomuoto tiedonvälitykseen.
JSX	Javascript Syntax Extension eli HTML-kieltä muistuttava tapa kirjoittaa JavaScriptiä, jota käytetään React-komponenttien luonnissa.

Mockup	Malli kehitettävän sovelluksen ulkoasusta ja toiminnoista.
MongoDB	Ilmainen, avoimen lähdekoodin tietokantaohjelma.
NodeJS	Avoimen lähdekoodin alustariippumaton ajoympäristö JavaScript-koodin suorittamiseen palvelimella.
React	Facebookin kehittämä JavaScript-kirjasto käyttöliittymien ja käyttäjärajapintojen rakentamiseen.
Refaktorointi	Prosessi, jossa ohjelman lähdekoodia muutetaan niin, että sen toiminnallisuus säilyy vaikka rakenne muuttuu.
Renderöinti	Kuvantaminen eli tiettyjen ohjelmakomponenttien tai kuvien esittäminen näytöllä.
Route	Reitti, jota pitkin HTTP-pyynnöt kulkevat palvelimessa joko käyttöliittymästä tietokantaan tai tietokannasta käyttöliittymään.
SPA	Single Page Application eli selaimessa käytettävä ohjelmisto, jossa kaikki toiminnallisuudet ja tiedot ladataan kerralla selaimeen mutta kaikkea ei välttämättä näytetä kerralla.
SQL	Structured Query Language eli IBM:n kehittämä standardoitu kyselykieli, jolla relaatiotietokantaan voi tehdä erilaisia hakuja, muutoksia ja lisäyksiä.

1 Johdanto

1.1 Työn tavoite ja työkalut

Tässä toiminnallisessa opinnäytetyössä oli tarkoituksena toteuttaa toimeksiantona Joensuun seutukirjastolle pohjoiskarjalaisten musiikintekijöiden tietoja sisältävä tietokanta ja tämän tietokannan käyttöliittymä. Toimeksiannon pääasiallisena tavoitteena oli siis suunnitella ja luoda kyseinen tietokanta ja tietokannan sisältämiä tietoja kirjaston asiakkaille esittelevä verkkosovelluspohjainen käyttöliittymä. Käyttöliittymään oli lisäksi tarkoitus rakentaa tietokannan ylläpitoa varten erilaisia toiminnallisuuksia kuten tietueiden lisäys, muokkaus ja poisto. Autentikointi- ja kirjautumistoiminnot tietokannan ylläpitäjien tunnistamiseksi olivat niin ikään välttämättömiä.

Nykyaikaiset tietokannat toteutetaan yleensä erilaisia tietokantatyyppejä mallintaen. Eräät yleisimmistä nykyään mallinnetuista tietokantatyypeistä ovat relaatiotietokanta, joka järjestelee tiedot tiukasti järjesteltyihin tauluihin, sekä dokumenttitietokanta, joka järjestelee tietoja vähemmän organisoituihin dokumenttikokoelmiin. Molemmilla tietokantatyypeillä on omat hyvät ja huonot puolensa, ja näitä jo tiedossa olevia ominaisuuksia käytettiin hyväksi valittaessa tietokantatyyppiä, joka sopisi parhaiten tämän opinnäytetyötoimeksiannon käytännön toteutukseen. Tietokantatyypiksi valikoitui lopulta dokumenttitietokanta, joka sopii parhaiten Kaski-musiikkitietokannan tapaisen, useaa erilaista, erityyppistä ja eri kokoista tietuerakennetta sisältävän tietokannan toteuttamiseen.

Tietokannan käyttöliittymä oli opinnäytetyössä tarkoitus suunnitella ja toteuttaa oman harkinnan mukaan, toki toimeksiannon vaatimusmäärittelyjä mahdollisimman tarkasti noudattaen. Nykyaikaisessa sovelluskehityksessä suositaan SPA (Single Page Application) -lähestymistapaa, jossa verkkosovellus ja/tai käyttöliittymä toteutetaan ja tuodaan selaimeen yhtenä sivuna, joka sisältää erilaisia ominaisuuksia ja toiminnallisuuksia palvelimelta tai tietokannasta haettavasta

datasta ja käyttäjän toiminnasta riippuen. Musiikkitietokannan käyttöliittymä toteutettiin juuri tällaisena SPA-tyylisenä sovellusratkaisuna. Kooditasolla käyttöliittymän rakennuspalikoiksi valikoituivat Javascript-ohjelmointikieli ja React-käyttäjärajapintakehys, ja palvelinpuolen ohjelmoinnissa puolestaan hyödynnettiin Javascriptiä sekä Express-palvelinkirjastoa.

1.2 Työn rakenne

Opinnäytetyöraportti on jaoteltu Karelia-ammattikorkeakoulun opinnäytetyömallin mukaisiin lukuihin. Luvussa 1 käydään lyhyesti läpi työn tausta ja tavoitteet, käytettävät työkalut, sekä työn rakenne yleisesti. Luvussa 2 käsitellään opinnäytetyön teoriapuolta. Aluksi perehdytään tietokantojen yleispiirteisiin, mutta tämän jälkeen tutustutaan tarkemmin relaatio- ja dokumenttitietokantoihin. Lisäksi perehdytään verkko-ohjelmoinnin teoriaan, käyttöliittymä- ja palvelinohjelmoinnissa tarpeellisiin tekniikoihin, Javascriptiin kielenä, sekä React- ja Express-kirjastoihin.

Luvussa 3 perehdytään opinnäytetyön käytännön työskentelyssä apuna käytettäviin sovelluksiin ja menetelmiin. Luvussa 4 taas käydään läpi seikkaperäisesti työn käytännön toteutus, aina käyttöliittymän ja palvelintason suunnittelusta ja rakentamisesta tietokantapuolen suunnitteluun ja kehitettävän sovelluksen testaukseen. Käyttöliittymän ulkoasun suunnittelua ja toteutusta on tuotu esille kuvien avulla.

Luvussa 5 käydään läpi työn tulokset, eli millainen sovelluksesta tuli ja miten se vertautuu vaatimusmäärittelyissä mallinnetun sovelluksen ominaisuuksiin. Luvussa 6 eli pohdintaosiossa käsitellään sitä, mitä kehitetystä sovelluksesta jäi puuttumaan, mitä ylimääräistä saatiin aikaiseksi, ja millaiset jatkokehitysmahdollisuudet sovelluksella on. Lisäksi pohditaan, mitä uutta opinnäytetyön teoriatutkimuksen ja käytännön toteutuksen aikana opittiin, ja minkälaisia eväitä opinnäytetyö antoi oman osaamisen kehittymiseen.

2 Tietokannat ja verkkosovellukset

2.1 Tietokannat

2.1.1 Tietokantojen yleispiirteet

Tietokanta on yleisesti hyväksytyn määritelmän mukaan järjestetty tietokoneen tallentama tietojen ja informaation kooste. Tietokantoja käytetään hyväksi erilaisissa sovelluksissa tietojen säilömiseen, hakemiseen sekä käsittelyyn. Tietokanta voi olla sopiva ratkaisu yksittäisellekin käyttäjälle suunnatussa sovelluksessa (esimerkiksi sähköpostiohjelma), mutta erityisesti tietokantaa tarvitaan tiedon tallennusmekanismina silloin, kun tietovarastolta vaaditaan koordinaatiota usean eri käyttäjän välillä. (Encyclopedia Britannica 2021.)

Jotta tietokannasta saisi parhaan mahdollisen hyödyn irti, sen sisältämää tietoa pitää pystyä sekä lukemaan että muokkaamaan. Tietokannoista voidaan erilaisilla hakumenetelmillä saada näkyviin yksittäinen tai useampi tieto tai tietoyhdistelmä. Tietokantaan voidaan käyttäjän toimesta myös lisätä kokonaan uusi tieto tai poistaa vanha tieto. Myös jo olemassa olevaa tietoa voidaan korjata tai täydentää. Vaatimuksena kaikille näille operaatioille on, että tietokannan sisältämän tiedon tulee säilyä varsinaista operaation kohdetta lukuun ottamatta muuttumattomana ja kaikkien operaatioiden on toimittava halutulla tavalla ja ainoastaan halutulla tavalla.

Operaatioiden oikeellisuus varmistetaan sillä, että kaikki transaktiot eli tietokantaoperaatiot noudattavat niin kutsuttua ACID (Atomicity, Consistency, Isolation, Durability) -protokollaa. Protokollan mukaan transaktioiden tulee olla atomisia (jokainen operaatio suoritetaan kokonaan tai ei ollenkaan), johdonmukaisia (samanlainen operaatio antaa aina samanlaisen tuloksen, ja tietokannan tulee siirtyä operaation jälkeen eheästä tilasta käskyn mukaiseen uuteen eheään tilaan), eristettyjä (eri käyttäjien operaatiot eivät vaikuta toisiinsa, eivätkä ne näy toisil-

leen), sekä kestäviä (operaation jälkeiset muutokset eivät saa kadota järjestelmästä, eikä mahdollinen kaatuminenkaan voi estää onnistuneita operaatioita). (Haerder & Reuter 1983, 288–290.)

2.1.2 Tietokantatyypit

Tietokannat jaotellaan erilaisiin tyyppeihin riippuen siitä, millaisia tietokantamalleja ne tukevat. Tietokantamalli on kaavio tai käsitekonstruktio, joka määrittelee tietokannan loogisen rakenteen, ja sen, kuinka tietoa tallennetaan, organisoidaan ja käsitellään tietokannassa.

Varhaisimmat ja aikoinaan käytetyimmät tietokantamallit olivat niin kutsuttuja navigationaalisia malleja, kuten hierarkkinen tietokantamalli (jossa data organisoidaan puurakenteeksi), ja verkkotietokantamalli (jossa data järjestellään verkkorakenteeksi). Näissä malleissa käytetään osoittimia (usein fyysisiä levyosoitteita) sekä eri tietueiden välisiä linkkejä datan tallentamisessa, haussa ja seulonnassa. (Bachman 1973, 653–658.) Esimerkiksi Internetin nimipalvelujärjestelmä DNS ja hakemistoprotokolla LDAP pohjautuvat yhä edelleen hierarkkiseen tietokantamalliin.

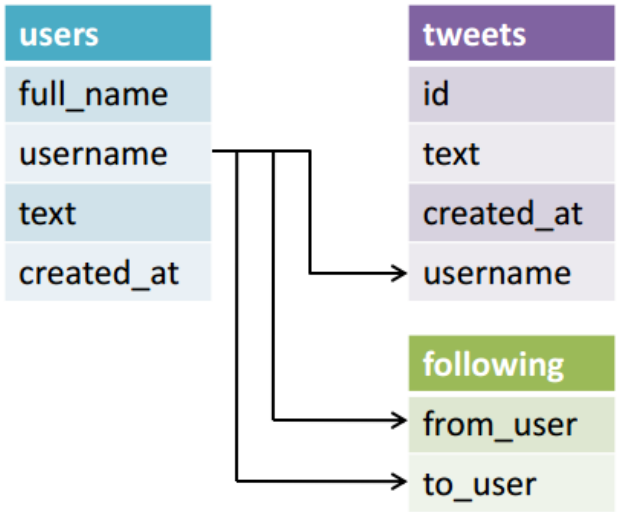
1970-luvulla kehitettiin niin sanottu relaatiotietokantamalli, jossa tietoa ei haeta tai organisoida erillisten osoittimien tai linkkien avulla, vaan itse datan tietosisällön mukaan. Tietokannan kokonaisrakenne voidaan myös vapaasti määrittää, eikä sitä tarvitse organisoida esimerkiksi puuksi tai verkoksi. (Codd 1970, 377–387.) Relaatiotietokannat korvasivat 1980-luvulle tultaessa suuremmaksi osaksi vanhemmat navigationaaliset tietokannat, paitsi joissain erityistapauksissa.

Noin vuodesta 2000 alkaen on kehitetty lisäksi niin sanottuja post-relationaalisia tietokantamalleja (muun muassa NoSQL-mallit), jotka korjaavat joitain relationaalisten mallien ongelmia. Näihin ongelmiin lukeutuvat muun muassa ei-järjestellyn tiedon tallentaminen ja horisontaalisen skaalautuvuuden puute. Kyseisiä

malleja kuvantaviin tietokantatyyppeihin kuuluvat esimerkiksi avain-arvo-tietokannat, oliotietokannat, joissa tieto tallennetaan ja haetaan olioina, sekä dokumenttitietokannat, joissa tieto tallennetaan dokumenttikokoelmiin. (Drake 2014; Leavitt 2010, 12–14.)

2.1.3 Relaatiotietokannat

Relaatiotietokanta koostuu tauluista, joista kukin sisältää määrätyn määrän kiinteitä sarakkeita. Tieto tallennetaan tauluihin riveinä, jotka voivat viitata toisiinsa, ja joilla jokaisella on oma tunnistamisessa käytetty id-numeronsa (eli niin sanottu pääavaimensa). Sarakkeet vastaavat tämän talletetun tietoelementin eli tietueen yksittäisiä attribuutteja eli ominaisuuksia. Jokaisessa taulussa on kokoelma tiettyyn asiaan liittyvää tietoa, ja taulujen ja niiden sisältämien tietojen välille voidaan luoda yhteyksiä toisen taulun avaimia (eli viiteavaimia) hyväksikäyttäen. Taulujen välille syntyy tällöin niin kutsuttu relaatio. (Codd 1970, 377–387.) Relaatiorakennetta havainnollistetaan kuvassa 1, jonka esittämässä tietokannassa kolme taulua sisältää kukin omanlaistaan tietoa. Tieto tauluissa on tallennettu riveille, jotka on järjestelty yllä mainittujen sarakkeiden mukaan. Kyseisten tietojen välille luodaan yhteyksiä eli relaatioita (nuolet).



Kuva 1: Yksinkertainen relaatiotietokanta (Kuva: code.tutsplus.com).

Relaatiotietokannan käyttäjä käsittelee tietokantadataa Structured Query Language (SQL) -kielellä. Kyseisessä kielessä on joukko standardoituja komentoja, joiden avulla tietokannan käyttäjä voi lisätä, hakea, muuttaa ja poistaa tietoa tietokannasta. Näitä komentoja ja niiden toteuttamia operaatioita kutsutaan kyselyiksi. (Chamberlin & Boyce 1974, 249–264.) Relaatiotietokannan sisällön ja käyttäjän välissä on tietokantajärjestelmä, jonka tehtävänä on käsitellä käyttäjän antamat SQL-komennot. Käyttäjän tarvitsee vain kuvailla SQL-kielellä, miten hän haluaa käsitellä tietokantaa, minkä jälkeen tietokantajärjestelmä suorittaa kyseisen operaation. Käyttäjän ei siis tarvitse tietää mitään tietokannan sisäisestä rakenteesta tai toiminnasta, vaan hän voi luottaa tietokantajärjestelmään, joka kätkee käyttäjältä tietokannan sisäisen toiminnan monimutkaiset yksityiskohdat. (Connolly & Begg 2014, 64.)

Relaatiotietokantoja käyttävät useimmiten yritysmaailman toimijat, ja SQL-kyselyiltä vaaditaankin usein suurta tarkkuutta. ACID-protokollaa noudatetaan siis yleensä hyvin tarkasti, mikä voi tehdä operaatioista hitaita, mutta samalla myös hyvin luotettavia ja toimintavarmoja. (Leavitt 2010, 12–14.) Yleisimmät tällä hetkellä käytetyistä kaupallisista relaatiotietokantajärjestelmistä ovat Oraclen tietokantajärjestelmä, Microsoftin SQL Server, ja IBM:n Db2. Vapaassa käytössä on myös useita ilmaisia relaatiotietokantajärjestelmiä, kuten MySQL/MariaDB, PostgreSQL, SQLite, sekä Hive. (DB-Engines 2021a.)

2.1.4 Dokumenttitietokannat

Relationaaliset tietokannat toimivat parhaiten, mikäli tieto, joka niihin syötetään, on jo valmiiksi järjesteltyä tietokannan sisältämien taulujen sarakerakenteen mukaisesti. Järjestelemättömän tiedon, kuten esimerkiksi tekstidokumenttien ja kuvien sisältämän datan, tallentaminen onkin vaikeampaa, ja tällaisen ei-järjestellyn datan tallentamisella relaatiotietokantaan onkin tapana tehdä tauluista kohtuuttoman monimutkaisia. SQL-kyselyt toimivat myös parhaiten, jos data on valmiiksi organisoitua. (Leavitt 2010, 12–14.)

Relationaalisiin tietokantamalleihin liittyvä toinen käytännön ongelma on horisontaalisen skaalautuvuuden puute, joka tarkoittaa sitä, että relaatiotietokannan jakaminen usean eri tietokoneen ja palvelimen kesken on hyvin vaikeaa. Näiden edellä mainittujen ongelmien takia onkin viime vuosikymmeninä kehitetty useita erilaisia niin kutsuttuja post-relationaalisia tietokantamalleja, joista avain-arvo-tietokannat, oliotietokannat, ja dokumenttitietokannat ovat yleisimpiä ja käytetyimpiä. (Leavitt 2010, 12–14.)

Dokumenttitietokannat organisoivat tiedot järjesteltyjen taulujen ja valmiiksi määriteltyjen kenttien sijasta dokumenttikokoelmiin, joissa sijaitseviin tietueisiin (olioihin) voidaan sijoittaa vapaavalintainen määrä kenttiä. Näihin kenttiin voidaan taas tallentaa mikä tahansa vapaavalintaisen kokoinen yksittäinen tieto, mikä tekee dokumenttitietokannasta hyvin joustavan tiedon tallennusmekanismin.

Dokumenttitietokannoille ei ole olemassa SQL:n kaltaisia kyselykieliä, joten kaikki tietokantaa operoivat toiminnot suoritetaan manuaalisesti suoraan sovelluksesta, yleensä palvelintasolta. Tämä on yksinkertaisilla operaatioilla toki helppoa, mutta suurilla tehtävämäärillä toteuttaminen saattaa viedä kohtuuttomasti aikaa. On myös muistettava, että dokumenttitietokantajärjestelmät eivät luonnostaan noudata ACID-protokollaa, eli ACID:n tarkempi noudattaminen dokumenttitietokannoissa vaatii ylimääräistä ohjelmointia. (Leavitt 2010, 12–14.) NoSQL-tietokannat yleensä ja dokumenttitietokannat erityisesti ovat vapaasti levitettäviä ilmaissovelluksia. Esimerkkejä käytetyimmistä dokumenttitietokantajärjestelmistä ovat MongoDB, CouchBase/CouchDB, ja Firebase Realtime Database (DB-Engines 2021b).

2.2 Verkko-ohjelmointi

2.2.1 Verkko-ohjelmoinnin teoriaa

Kaikkein perustavimmalla tasolla ohjelmointi tarkoittaa tietokoneelle tai vastaavalle laitteelle jollakin tavalla annettavia toimintaohjeita, tavallisesti käyttämällä tällaista tarkoitusta varten kehitettyä ohjelmointikieltä. Ohjelmistokehitys puolestaan sisältää ohjelman kehittämisen laajemmassa merkityksessä kuin pelkkä ohjelmointi, eli siihen kuuluu myös ohjelmistosovellusten vaatimusmäärittely ja testaaminen.

Hajautetuiksi järjestelmiksi kutsutaan fyysisesti eri paikoissa sijaitsevia tietokoneita, jotka vaihtavat toistensa kanssa tietoja ja toimivat koordinoidusti yhteisen päämäärän eteen. Hajautetuiksi sovelluksiksi puolestaan kutsutaan tällaisissa järjestelmissä toimivia ja esimerkiksi verkkoyhteyden kautta toisiinsa yhteydessä olevia ohjelmistoja. Tällaisiksi sovelluksiksi lasketaan myös perinteiset selaimessa toimivat verkkosovellukset. Hajautettujen sovellusten ohjelmointi ja koko ohjelmistokehitys on niin kutsuttua hajautettua ohjelmistokehitystä. (Tanenbaum & Steen 2002, 2–7.)

Hajautettujen sovellusten (eli samalla myös verkkosovellusten) toiminta ja ohjelmistokehitys tehdään tavallisesti käyttäen hyväksi niin sanottua asiakas-palvelin-arkkitehtuuria, jossa erilaiset tehtävät jaetaan joko palvelun tarjoajien eli palvelimien sekä palvelun pyytäjien eli asiakkaiden välille. Palvelimiin sijoitetaan yleensä tietokannat sekä tietokannan operointilogiikka, kun taas asiakaspuolella (eli tietokoneella, joka palveluja pyytää) sijaitsevat käyttöliittymän ja sen toiminnallisuudet tuottavat komentosarjat.

Toinen tapa tehdä sama asia on käyttää hyväksi vertaisverkkoarkkitehtuuria, jossa ei ole erikseen palvelun tarjoajia eli palvelimia ja palvelun käyttäjiä eli asiakkaita: Kaikki koneet voivat toimia sekä palvelun tarjoajina että käyttäjinä, ilman että on olemassa keskitettyä koordinaatiota tai tehtäväjakoa. Tämä hajautus lisää toiminnan joustavuutta ja luotettavuutta, ja tehostaa resurssien käyttöä,

mutta toisaalta voi myös altistaa yksittäisiä koneita hyökkäyksille ja saa aikaan tilanteita, joissa resursseja käytetään hyväksi, mutta mitään ei anneta vastineeksi. (Bandara & Jayasumana 2012, 257–276.)

Oli käytettävä asiakas-palvelin-arkkitehtuuri mikä hyvänsä, yleensä verkkosovelluksen sisältö, ulkoasu ja toiminnallisuudet säilötään palvelinkoneelle, josta käyttäjän oma verkkoselain hakee ne ja pyörittää niitä selaimen omalla moottorilla. Nykyaikaisessa verkkosovelluskehityksessä käytetään hyväksi erillisinä haettavien ja staattisten verkkosivujen sijaan niin kutsuttua Single Page Application (SPA) -lähestymistapaa, jossa koko sovellus haetaan yhtenä kokonaisena sivuna, jonka sisältö ja ominaisuudet vaihtelevat tietokannasta noudetun datan ja käyttäjän toiminnan mukaan.

2.2.2 Javascript

JavaScript on alun perin Netscapen kehittämä korkean tason komentosarjakieli. Se on dynaamisesti tyypitetty (toisin sanoen muuttujien tyypitys tehdään aina tapauskohtaisesti), ja se käännetään ajonaikaisesti. Siihen kuuluvat olennaisena osana sekä prototyyppi- että luokkapohjainen olioajattelu ja moniparadigmaisuus (usean eri ohjelmointityylin tukeminen). (Flanagan 2020, 1–10.) Yleensä Javascriptiä käytetään käyttöliittymälogiikan luomiseen, mutta joskus sillä koodataan myös palvelinpuolta. Jopa yli 97 % nettisivujen asiakaspuolen toiminnallisuuksista (W3Techs 2021a) ja huomattava osa palvelinsovelluksista (W3Techs 2021b) on koodattu Javascriptillä.

Yhdessä HyperText Markup Language (HTML) ja Cascading Style Sheets (CSS) -kielten kanssa Javascript muodostaa World Wide Web -ympäristön keskeisimmän teknologisen ytimen. HTML määrittelee verkkosivujen sisällön, CSS määrittelee niiden ulkoasun, ja Javascript määrittelee niiden dynaamiset toiminnallisuudet. (Flanagan 2020, 1.) Lähes kaikissa nettiselaimissa on sisäänrakennettu Javascript-moottori, joka mahdollistaa kyseisten koodien pyörittämisen

käyttäjän koneessa. Alun perin tämä oli ainoa tapa käyttää Javascript-toiminnallisuuksia, mutta nykyään Javascript-ajoympäristönä voi toimia myös selaimen ulkopuolinen moottorisovellus, kuten Node.js. Tällaiset sovellukset mahdollistavat palvelinpuolen ohjelmoinnin. (Mahemoff 2009.)

2.2.3 Käyttöliittymäohjelmointi Reactilla

Pyörää ei aina kannata keksiä uudestaan, kun on kyse sovelluskehityksestä. Monessa tapauksessa voidaan käyttää hyväksi aikaisempien ohjelmoijapolvien kokemusta ja osaamista. Ohjelmakehityksen ja ohjelmien suorittamisen tarpeita varten onkin kehitetty niin sanottuja kirjastoja, eli kokoelmia esikirjoitettua koodia, joiden on tarkoitus nopeuttaa paitsi kehitystyötä, myös ohjelmien toimintaa. Tällaisia eri tarkoituksiin käytettäviä koodikirjastoja on kehitetty erityisen paljon Javascriptille.

Yksi laajalti käytetyistä Javascript-kirjastoista on React, joka on tarkoitettu verkkosivujen käyttäjärajapintojen ja interaktiivisten ulkoasukomponenttien rakentamiseen (ReactJS 2021a). React muodostaa niin sanotun web-frameworkin eli verkko-ohjelmistokehyksen. Perinteisesti ohjelmistokehyksiksi on kutsuttu sovelluksia, joka muodostavat rungon niiden päälle rakennettaville ohjelmille, ja jotka tarjoavat valmiiksi rakennettuja komponentteja, joita ei tarvitse kirjoittaa uudelleen ohjelmistokehityksen aikana. Verkkokehykset (kuten React, Angular, Vue, ja vastaavat) ovat tällaisten ohjelmistokehysten erikoistapauksia, jotka monella tavalla tukevat verkkosovellusten kehitystyötä ja automatisoivat verkkokehityksen perustoiminnallisuuksia.

React-koodi koostuu komponenteiksi kutsutuista vuorovaikutteisista Javascript-elementeistä, jotka voidaan renderöidä (eli kuvantaa) haluttuun kohtaan HTML-koodia eli verkkosivun yleisarkkitehtuuria. Kuvan 2 esimerkissä App-komponentti tuodaan Javascript-tiedostosta ’App.js’ ja renderöidään HTML-elementtiin ’root’.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
    <App />,
    document.getElementById('root')
);
```

Kuva 2: Esimerkki React-koodista (Kuva: reactjs.org).

Komponenteilla voi olla myös lapsikomponentteja, joille voidaan välittää ylhäältä propseiksi (properties) kutsuttua informaatiota. Kuvan 3 esimerkissä HelloMessage-komponentille välitetään dataa (’propseja’) ylhäältäpäin. HelloMessage voi samalla tavalla välittää tietoelementtejä omille alikomponenteilleen. React-koodi on siis luonnostaan modulaarista, ja sillä voidaankin luontevasti rakentaa ohjelma tai sovellus, joka muodostuu sopivan kokoisista ja helposti käsiteltävistä moduuleista ja niiden alimoduuleista. (ReactJS 2021b.)

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  mountNode
);
```

Kuva 3: Datan (’propsien’) välitys HelloMessage-komponentille (Kuva: reactjs.org).

React on erittäin hyödyllinen ja helppokäyttöinen ohjelmistokehys myös siksi, että tietyn komponentin tilan muuttuessa sovellus renderöi automaattisesti kyseisen komponentin ja vain kyseisen komponentin, sekä tietenkin ne alikomponentit, joihin kyseinen komponentti vaikuttaa. Koko olemassa olevaa komponenttipuuta ei siis päivitetä ylhäältä alas asti. Tällainen toimintamalli tekee myös React-sovelluksen toiminnasta modulaarista, ja tämä modulaarinen toiminta tehostaa suoraan Reactilla luotujen verkkosivujen toimintaa. (ReactJS 2021c.)

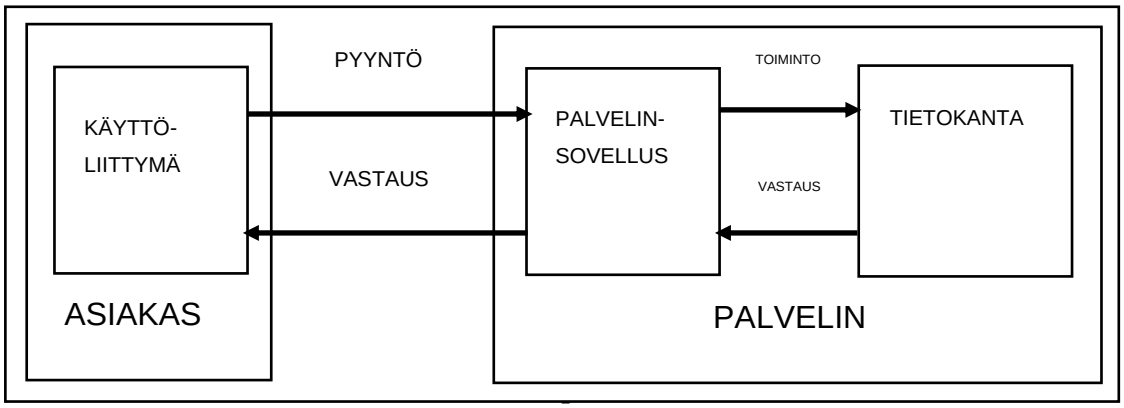
React-komponentit on mahdollista luoda kokonaan puhtaalla Javascriptillä, mutta lähes aina niiden koodauksessa hyödynnetään Javascriptin ja HTML:n koodauslogiikkaa yhdistävää Javascript Syntax Extension (JSX) -syntaksia, joka muistuttaa ulospäin HTML:ää, mutta on kuitenkin perustaltaan Javascriptiä (ReactJS 2021d). JSX mahdollistaa tietyllä tavalla koodaamisen näillä kahdella ohjelmointikielellä yhtä aikaa, ja silloin sekä komponentin ulkoasu että sen toiminnallisuudet voidaan vaivattomasti sijoittaa samaan tiedostoon. Tämä osaltaan helpottaa ja virtaviivaistaa React-komponenttien ja koko sovelluksen kehitystyötä.

Vuonna 2019 Reactiin lisättiin uusi ominaisuus nimeltä hook, joka mahdollistaa useiden erilaisten tilamuuttujien, tilanhallintafunktioiden ja renderöintitoiminnallisuuksien lisäämisen React-komponenttiin ilman, että komponenttia tarvitsee julistaa luokaksi. Hookien avulla komponentin tilaa ja tiloja pystytään helposti muuttamaan, ja sitä kautta myös komponentin uudelleen kuvantaminen yksinkertaistuu. React-hookit ovatkin suureksi osaksi syrjäyttäneet vanhat luokkamuotoiset tilatyypit ja niiden hallintalogiikat React-koodauksessa. (ReactJS 2021e.)

Reactia käyttävät verkkosovelluksissaan ohjelmistokehyksenä monet sellaiset yritykset, joiden kehittämille sovelluksille on tärkeää ulkoasun ja toimintojen modulaarisuus. Muun muassa Facebookin, Instagramin, Netflixin, Alibaban ja E-Bayn käyttöliittymät on luotu Reactia hyödyntäen.

2.2.4 Palvelinohjelmointi

Tietokantojen perimmäinen tarkoitus on säilöä tietoa myöhempää käyttöä varten. Tietokannan sisältämästä tiedosta ei ole kuitenkaan mitään hyötyä, jos käyttöliittymästä ei pääse käsiksi tietokantaan. Käyttöliittymästä ei myöskään ole mitään hyötyä, ellei tietokannan sisältämää dataa voi lähettää käyttöliittymään ja muuttaa käyttäjäystävälliseen muotoon. Tätä tarkoitusta varten näiden kahden toimintatason välille rakennetaan niin kutsuttu palvelintaso, joka asiakkaan eli käyttöliittymän käyttäjän pyynnöstä suorittaa niin kutsutun ’palvelun’. Tämä palvelu on verkkosovelluksen tapauksessa halutun ja sovellukselle tai tietokannalle sopivalla tavalla strukturoidun tiedon välittäminen joko käyttöliittymästä tietokantaan tai tietokannasta käyttöliittymään. Käyttöliittymän, palvelintason ja tietokannan muodostamaa toimintamallia kutsutaan asiakas-palvelin-malliksi (kuva 4).



Kuva 4: Asiakas-palvelin-malli ja sen eri osien välinen kommunikointi.

Tietokantaa hyväksi käyttävissä verkkosovelluksissa asiakaspuoli eli käyttöliittymä kommunikoi palvelinpuolen ja edelleen tietokannan kanssa niin kutsutun HyperText Transfer Protocol (HTTP) -protokollan avulla. Kun tietokantaan lähetetään käyttöliittymästä lomakkeellinen uutta dataa, suoritetaan haku kannasta, tai poistetaan tietoja, silloin lähetetään HTTP-pyyntö (post, get tai delete) palvelimelle. Palvelinsovellus vastaanottaa pyynnön, käsittelee sen asianmukaisella

tavalla, suorittaa halutun tietokantatransaktion, ja lähettää käyttöliittymään takaisin HTTP-vastauksen, joka sisältää paitsi tiedon siitä, miten kysely onnistui, myös tietokannasta vastaanotettua dataa, jos sellaista pyydettiin. (Fielding ym. 1999.)

Palvelintaso mahdollistaa myös niin sanotun istuntokohtaisen informaation keräämisen, eli palvelin voi kerätä tietoa tietystä käyttäjästä (tai käytännössä siis istunnosta) ja lähettää vastauksena kyselyihin nimenomaan tälle istunnolle räätälöityjä vastauksia. Tämä mahdollistaa paitsi tiedon salaamisen myös säästää resursseja ja helpottaa niiden kohdentamista.

Palvelintaso voidaan rakentaa useilla eri ohjelmointikielillä, kuten PHP:llä, C#:lla, tai Pythonilla. Myös Javascriptiä käytetään palvelinpuolen toiminnallisuuksien luomiseen: Ei ole ollenkaan harvinaista verkkosovelluksissa, että sekä asiakas- että palvelinpuoli on koodattu Javascriptillä. Palvelinpuolelle on tarjolla useita erilaisia ohjelmistokehyksiä, jotka nopeuttavat ja helpottavat palvelimen rakentamista ja käyttämistä. Esimerkiksi Javascript-palvelinohjelmointia varten ovat tarjolla muun muassa Express- (ExpressJS 2021) ja Deno- (InfoQ 2018) palvelinkehykset.

3 Työssä käytetyt menetelmät

Ohjelmistokehitystyössä on tärkeää pystyä hallitsemaan paitsi itse kehitystyötä kokonaisuutena, myös koodiin tehtäviä muutoksia (versionhallinta), sekä ajankäyttöä (ajanhallinta). Tämän vuoksi menetelmien valinta hallintatoimintoihin on tärkeä osa kehitystyöhön valmistautumista ja itse käytännön kehitystyötä.

Tässä opinnäytetyössä kehitystyön kokonaishallinnassa käytettiin apuna Trelloa, joka on yksinkertainen verkkosovellus projektinhallintaan ja projektin etenemisen seurantaan. Koska tässä projektissa oli ainoastaan yksi tekijä ja yksi kehitettävä sovellus, kehitystyön hallinnan ei tarvinnut olla tarpeettoman

monimutkaista ja yksityiskohtaista. Tässä tapauksessa riitti, että se antoi kehittäjälle kohtuullisen hyvän yleiskuvan siitä, kuinka projekti eteni.

Versionhallinnassa käytettiin hyväksi Git-versionhallintasovellusta sekä Gitlab-pilvipalvelua. Git otettiin käyttöön vasta aika myöhäisessä vaiheessa kehitystyötä eli palvelinpuolen koodaamisen ja tietokantatoimintojen tullessa ajankohtaisiksi. Tällöin sovellus alkoi käydä sen verran monimutkaiseksi, että versionhallinnointi katsottiin välttämättömäksi toteuttaa.

Ajanhallintaan käytettiin Toggl-ajanhallintasovellusta. Ajanhallinnassa ei menty edelleenkään tarpeettomiin yksityiskohtiin (esimerkiksi siihen, mitä milloinkin tehtiin), vaan ainoastaan hallinnoitiin sitä, kuinka paljon aikaa minäkin päivänä käytettiin koodaamiseen ja muuhun kehitystyöhön. Tämäkin auttoi pysymään aikatauluissa ja helpotti muutenkin projektin etenemisen hallintaa. Myös tehdyn työn dokumentointi auttoi osaltaan työn etenemisen seurannassa. Päiväkirjamuotoinen dokumentointi tehtiin käyttämällä hyväksi HackMD-verkkosovellusta.

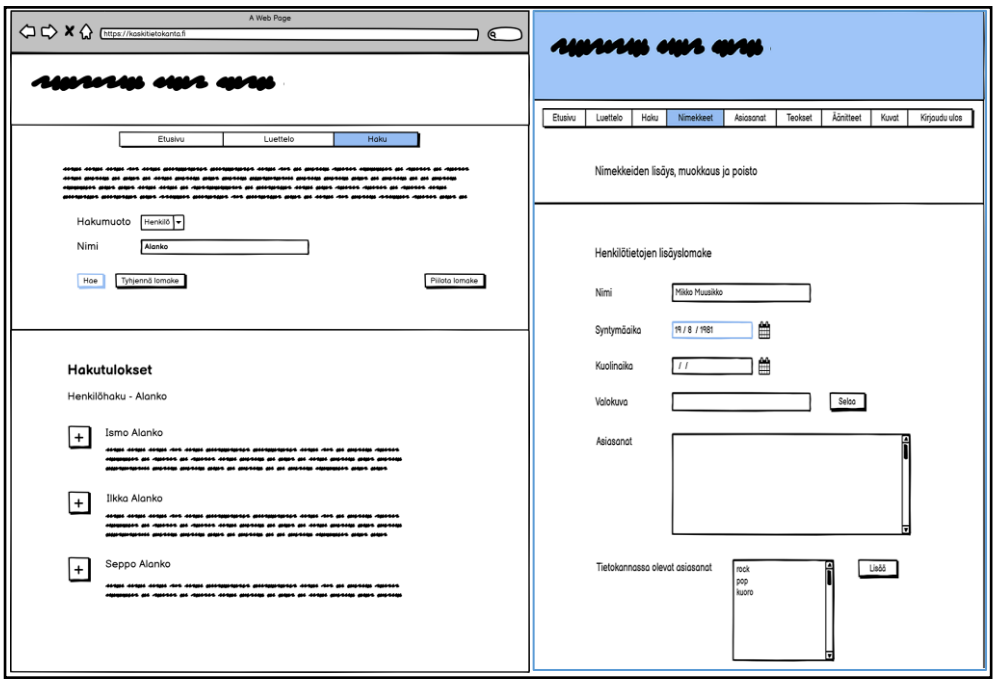
Sekä käyttöliittymän että palvelinpuolen ohjelmointiin käytettiin Microsoftin Visual Studio Codea, joka tarjoaa laajan valikoiman aputyökaluja ohjelmistokehitykseen. Visual Studio Code sisältää myös tuet tässä opinnäytetyössä hyödynnettyihin ohjelmointikieliin (HTML, Javascript ja CSS).

Käyttöliittymää ja palvelinpuolta testattiin myös sitä mukaa, kun kehitystyö eteni. Käyttäjärajapinnan testauksessa hyödynnettiin Google Chrome -verkkoselaimen kehitystyökaluja, ja palvelinpuolen testauksessa käytettiin apuna Postman-sovellusta, jolla pystytään lähettämään monimutkaisiakin HTTP-pyyntöjä kehitettävään palvelimeen. Testitietokantana palvelinkehityksen aikana ja sovelluksen lopulta toimiessa kokonaisuutenakin käytettiin MongoDB Atlas -pilvipalvelun tietokantaklusteriin luotua MongoDB- dokumenttitietokantaa.

4 Musiikkitietokannan ja käyttöliittymän toteutus

4.1 Käyttöliittymän suunnittelu

Ohjelmistokehityksessä toimivan ja helppokäyttöisen käyttöliittymän rakentaminen on ensiarvoisen tärkeää. Loppukäyttäjä on vuorovaikutuksessa ohjelman kaikkien osien ja tietokannan sisältämien tietojen kanssa juuri käyttöliittymän kautta, ja jos liittymä on huonosti suunniteltu ja sekava, se antaa välittömästi huonon vaikutelman koko sovelluksen toiminnasta. Käyttöliittymä pyritäänkin alusta alkaen luomaan sellaiseksi, että se on rakenteeltaan selkeä, johdonmukaisesti toimiva, ja antaa hyvää palautetta käyttäjän toiminnasta. Myös käytön helppous ja ennustettavuus on hyvissä käyttöliittymissä oleellista.



Kuva 5: Mockup -mallit käyttäjäpuolen hakulomakkeesta ja hakutuloksista (vasemmalla) sekä ylläpitopuolen henkilön lisäyslomakkeesta (oikealla).

Kaski-musiikkitietokannan käyttöliittymän suunnittelussa noudatettiin kaikkia edellä mainittuja periaatteita. Suunnittelu aloitettiin luomalla Balsamiq -verkkosovelluksella graafisia malleja siitä, miltä käyttöliittymän on tarkoitus näyttää (kuva 5). Balsamiqia ja muita sen kaltaisia niin kutsuttuja mockup-sovelluksia

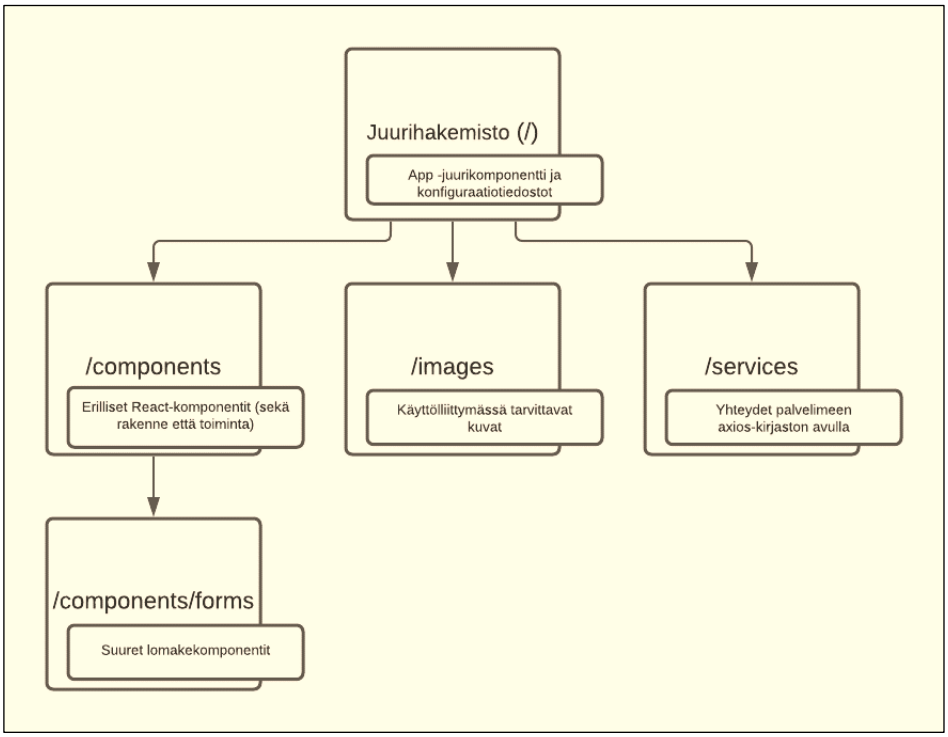
käytetään yleisesti apuna käyttäjärajapintojen ulkoasun suunnittelussa. Mockupeiksi kutsutut mallinnuskuvat helpottavat paitsi kehittäjän työskentelyä käyttöliittymän parissa, myös tarjoavat toimeksiantajalle/työnantajalle mahdollisuuden vaikuttaa sovelluksen ulkoasuun jo ennen kuin koodia on kirjoitettu riviäkään.

Toimeksiantajan kanssa oltiinkin tiiviissä yhteistyössä jo käyttöliittymän ulkoasun suunnitteluvaiheessa, ja pienten säätöjen ja muutoksien jälkeen kaikki osapuolet olivat liittymän tulevaan ulkonäköön tyytyväisiä. Ulkoasun perusteella oli toki mahdollista suunnitella myös itse toiminnallisuuksia, ja alustavia suunnitelmia tehtiinkin. Suunnitteluvaiheen jälkeen työskentelyssä kuitenkin edettiin nopeasti käyttöliittymän toteutusvaiheeseen, koska siihen kuluisi joka tapauksessa paljon aikaa ja voimavaroja.

4.2 Käyttöliittymän toteutus

4.2.1 Yleistä käyttöliittymästä

Kaski-musiikkitietokannan käyttöliittymä koodattiin Javascript-kielellä React-ohjelmistokehystä hyödyntäen. Liittymän ulkoasun luonnissa noudatettiin hyvin pitkälle mockup -malleja, ja vain muutamissa yksityiskohdissa poikettiin alkuperäisistä suunnitelmista. Toiminnallisuudet käyttäjärajapintaan luotiin Reactin tiloja ja hookeja hyödyntäen, ja käyttöliittymän yhteydet palvelimeen rakennettiin axios-kirjaston avulla perinteisiä REST-API- käytänteitä noudattaen.



Kuva 6: Käyttöliittymäsovelluksen hakemistorakenne.

Käyttöliittymäsovelluksen hakemistorakenne (kuva 6) luotiin React-koodauksen yleisiä konventioita seuraten niin, että juurihakemisto sisälsi käyttöliittymän App-juurikomponentin sekä apu- ja konfiguraatiotiedostot. Lisäksi kuville oli oma images -alihakemisto, palvelinyhteyksille oma services -alihakemisto, ja sellaisille rakenteellisille ja toiminnallisille React-komponenteille, joita on kätevin käsitellä juurikomponentin ulkopuolella, components -alihakemisto. Lisäksi lomakekomponenteille, jotka olivat liian suuria järkevästi käsiteltäväksi juurikomponentissa, oli oma alihakemistonsa.

4.2.2 Käyttöliittymän ulkoasu ja yleistoiminnot

Ensimmäinen vaihe käyttöliittymän rakentamisessa oli kokonaisulkoasun toteutus mockup-mallien mukaan. Aluksi rakennettiin yläpalkki, navigointipalkki ja sen painikkeet, sekä alasivut. Sovelluksesta oli tarkoitus tulla niin sanottu yhden sivun sovellus (SPA, single page application), joka hakee kaiken sovelluksen toiminnalle tärkeän tiedon kerralla palvelimelta selaimen muistiin ja kuvantaa koko käyttöliittymän ja kaikki sen alasivut sen jälkeen samaan HTML-osioon

(’root’) ja sitä kautta selainikkunaan. Navigointi alasivuilla ja alasivujen näyttäminen hoidetaan tämän jälkeen usean erillisen sivuhaun sijasta sovelluksen yhden React-tilan (’page’) muutoksella.

Seuraavaksi liittymäsovellukseen luotiin useita hook-tiloja, joihin tietokannan sisältämät tiedot ladataan heti sovelluksen käynnistyessä. Tiloja ja vastaavia tietorakenteita oli kaikkiaan kuusi: Henkilöt, yhteisöt, asiasanat, teokset, äänitteet ja kuvat. Koska tietokantana tullaan käyttämään MongoDB:tä, johon tiedot talletetaan avain-arvo-pareja sisältävinä dokumentteina, paras toimintatapa on muuttaa kyseiset tietorakenteet palvelintasolla standardisoituun JSON-muotoon. JSON-data voidaan sitten viedä suoraan tiloihin ja käsitellä taulukkona, joka sisältää useita erillisiä ja erilaisia olioita (yksi olio vastaa siis yhtä dokumenttia eli tietuetta).

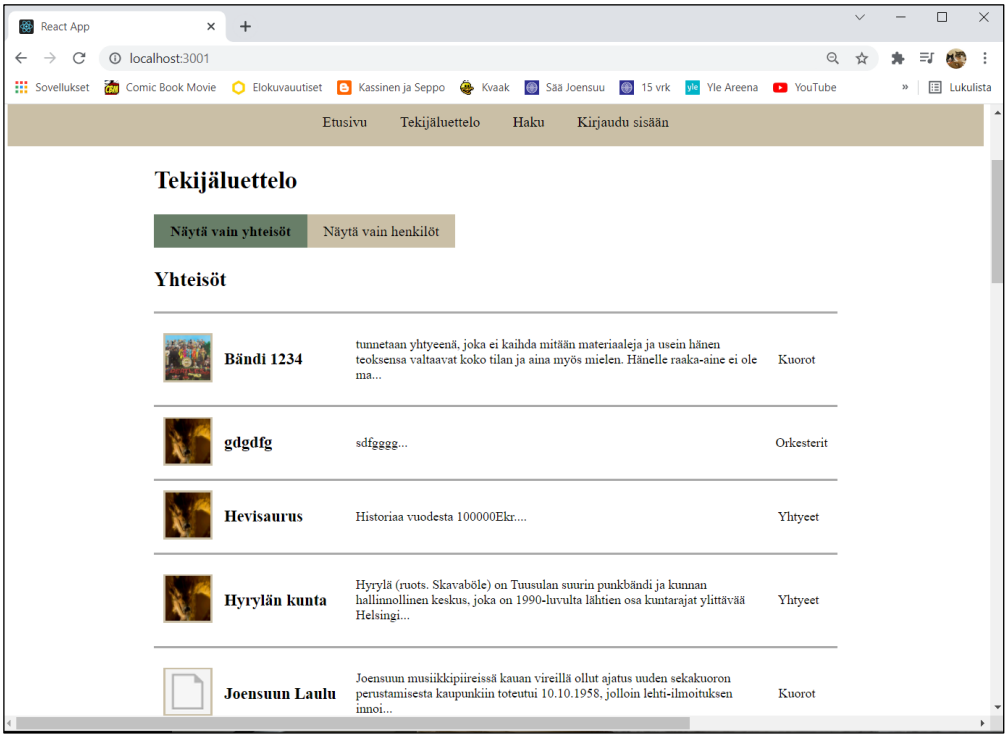
Koska tietokantaa ja palvelinpuolta ei ollut tässä vaiheessa kehitystyötä vielä olemassa, piti kaikki sovelluksen testauksessa käytettävä data aluksi koodata taulukkomuodossa suoraan käyttöliittymään (niin kutsuttu kovakoodaus). Muuten käyttöliittymän toimintoja ei olisi pystytty alusta lähtien järkevästi ja ajantasaisesti testaamaan.

4.2.3 Luettelosivu ja hakusivu

Kun koko sovelluksen yleinen ulkoasurakenne ja perustoiminnot olivat valmiita, jatkettiin käyttöliittymän alasivujen ulkoasun ja yksityiskohtaisempien toiminnallisuuksien rakentamisella. Koodaaminen aloitettiin etusivusta, luettelosivusta ja hakusivusta, koska nämä tulevat olemaan näkyvissä sekä käyttäjille että ylläpitäjille, ja ovat muutenkin tärkein osa musiikkitietokantasovelluksen käyttöliittymää. Myöhemmin työtä jatkettiin ylläpitopuolen alasivuilla, kuten tekijä-, asiasana- ja kuvasivuilla. Kaikkia sivuja tehtiin enemmän tai vähemmän rinnakkain, aina henkilökohtaisen mielenkiinnon ja työvaiheen sen hetkisen tärkeyden mukaan.

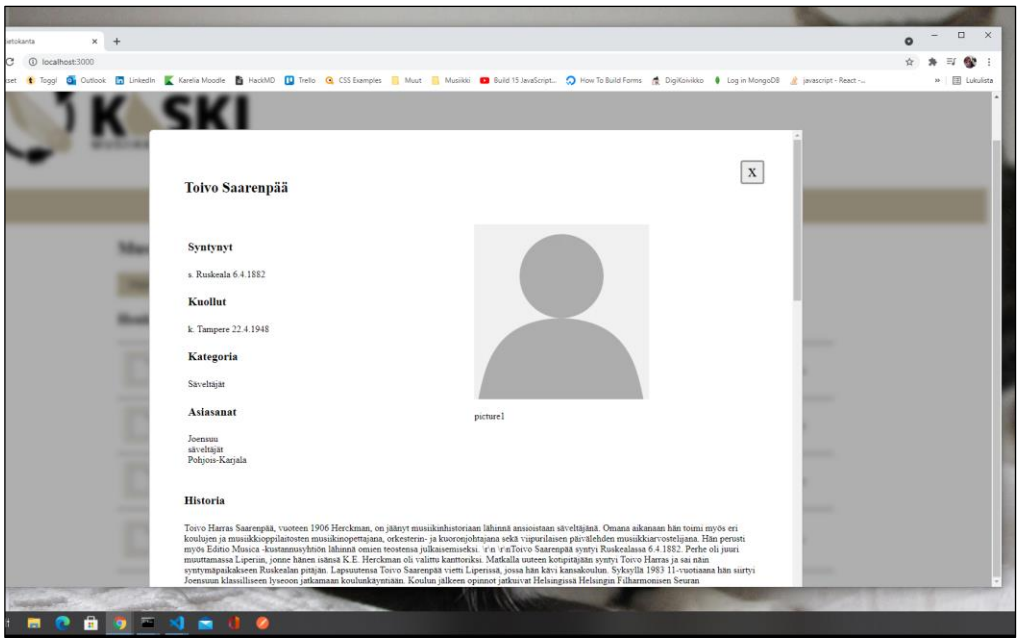
Luetteloalasivulle päätettiin listata yhdelle sivulle näkyviin kaikki tietokannan sisältämät musiikintekijät (sekä yhteisöt että henkilöt) nimen mukaan aakkosjärjestyksessä. Luettelo oli mahdollista ja jopa järkevää luoda tällä tavalla, koska musiikintekijätietueita ei loppujen lopuksi tule olemaan suurta määrää (yhteensä noin 150 kappaletta). Tälläkin määrällä luettelon manuaalinen selaaminen on kohtuullisen helppoa, varsinkin kun kyseinen luettelo on aakkosjärjestetty.

Luettelo rakennettiin niin, että jokaisen rivin eli listatun tietueen kohdalla näkyy tekijän nimi, lyhyt alkunäyte tekijähistoriasta, tekijäkategoria, sekä painike, josta avautuu ikkuna, johon tulee näkyviin kaikki kyseisen tietueen sisältämät tiedot (esimerkki luettelolistauksesta kuvassa 7). Esimerkkikuvassa sivun ylälaidassa on painettuna ’Näytä vain yhteisöt’-painike, jolloin tekijäluettelossa on nähtävillä vain musiikintekijäyhteisöt, jotka on lueteltu nimen mukaan aakkosjärjestyksessä. Tietueikkunan avauspainike, joka on jokaisen yhteisörivin vasemmassa laidassa, sisältää yhteisön kuvan, mikäli sellainen on olemassa.



Kuva 7: Luetteloalasivu.

Painikkeesta avautuva tietueikkuna puolestaan muotoiltiin niin, että se muistuttaa vanhanaikaista luettelokorttia, johon tulevat näkyviin kaikki tekijätietueen sisältämät tiedot johdonmukaisessa järjestyksessä. Tietueikkunan koko mukautuu pienemmäksi selainikkunan pienentyessä, mutta ei kasva selainikkunan suurentuessa tiettyä maksimikokoa isommaksi. Ikkunan tausta sumennettiin, jotta taustalla näkyvät yksityiskohdat eivät häiritsisi ja veisi käyttäjän huomiota oleellisesta eli itse tietueikkunan sisältämistä tiedoista. Esimerkki tietueikkunasta ja edellä mainituista ominaisuuksista on nähtävissä kuvassa 8.



Kuva 8: Tietueikkuna.

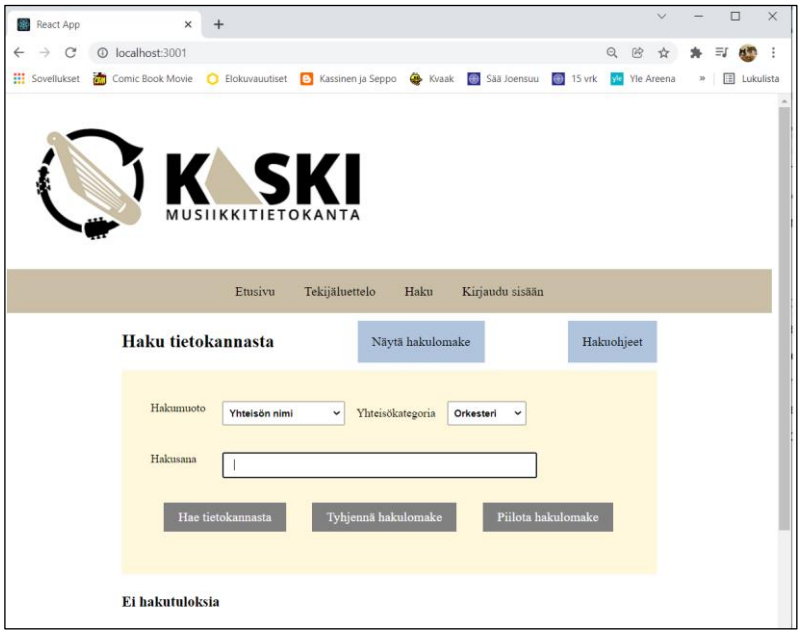
Tietueen sisältämät tiedot sijoitettiin ikkunaan aina samaan järjestykseen, eli kolmeen päälohkoon. Ensimmäisessä eli ylimmässä lohkossa sijaitsevat henkilön tai yhteisön nimi otsikkotekstinä sekä ikkunansulkemispainike. Tätä seuraavassa kakkoslohkossa sijaitsevat syntymä- tai perustamisaika, kuolin- tai hajoamisaika, kategoria ja asiasanat, ja näiden vieressä kuva ja kuvateksti, mikäli sellainen on kyseiseen tekijään liitettynä. Jos kuvaa ei ole, niin tässä kohdassa on pelkkä yleisluontoinen hahmokuva. Kolmoslohko sisältää allekkain kaikki muut tiedot: Henkilön tai yhteisön historiatiedot, henkilöön liitetyt teokset, henkilöön tai yhteisöön liitetyt äänitteet, ja loput vaatimusmäärittelyn mukaiset

tietuekentät säännönmukaisessa järjestyksessä. Huomionarvoista on, että huomautus- ja lisätietokenttien tekstit tulevat lopullisessa sovelluksessa näkyviin ainoastaan ylläpitäjälle eli sisään kirjautuneelle käyttäjälle.

Kuten kaikissa muissakin osissa käyttöliittymän kehitystyötä, myös tietueikkunan rakentamisessa pidettiin mielessä se, että tietojen tulisi näkyä selkeästi, aina samassa järjestyksessä ja niin, että yksittäiset tiedot eivät katoa tekstimassaan tai muun tiedon sekaan. Erityisen tärkeitä nämä suunnitteluperiaatteet ovat nimenomaan tietueikkunan kohdalla, koska kyseiseen osaan sovellusta tulevat näkyviin tietokannan sisältämät tärkeät ja loppukäyttäjän kannalta mielenkiintoiset tiedot.

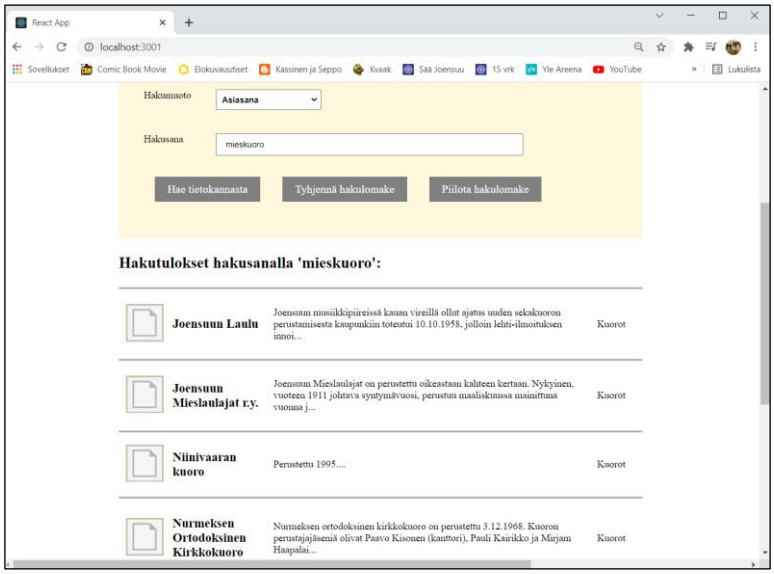
Seuraavassa kehitysvaiheessa keskityttiin hakualasivun ja hakulomakkeen ulkoasun ja toiminnallisuuksien rakentamiseen. Vaatimusmäärittelyjen mukaan hakuja tietokannasta on pystyttävä tekemään henkilön nimellä, yhteisön nimellä, ja asiasanalla. Lisäksi käyttöliittymästä käsin on pystyttävä tekemään vapaasanahakuja, joissa haku kohdistetaan kaikkien henkilöiden ja yhteisöjen kaikkiin kenttiin. Näihin kenttiin luetaan myös henkilöön ja yhteisöön liitettyjen teoksien ja äänitteiden kaikki kentät, joten hausta on tarkoitus tulla hyvin laaja-alainen.

Aluksi kehitettiin yksinkertainen hakulomake, josta on helposti valittavissa hakumuoto ja johon pystyttiin kirjoittamaan näkyvälle paikalle hakusana. Kuvan 9 esimerkissä hakumuodoksi voidaan valita henkilön nimi, yhteisön nimi, asiasana tai vapaasana. Lisähakuoptioina ovat myös henkilö- ja yhteisökategoriat. Haku käynnistyy ’Hae tietokannasta’ -painikkeesta ja hakuohjeet saa näkyville painikkeesta yläoikealla.



Kuva 9: Hakulomake.

Kaikki hakualgoritmit onnistuttiin rakentamaan käyttöliittymään toimiviksi ja halutunlaisiksi. Jopa vapaasanahaku, jonka rakentaminen tuntui alussa aika vaikealta hahmottaa, toimi loppujen lopuksi täysin moitteettomasti. Hakuja testattiin useassa eri vaiheessa ja useaan kertaan erilaisilla merkkijonoilla, ja aina algoritmit onnistuivat löytämään ne tekijätietueet ja vain ne tekijätietueet, joiden jokin kenttä tai tietueeseen liitettyjen teoksien tai äänitteiden jokin kenttä sisälsivät hakumerkkijonon.



Kuva 10: Hakulomake ja hakutulosluettelo.

Kuvassa 10 on nähtävillä esimerkki hausta ja hakutuloksista. Haku on tehty asiasanalla ’mieskuoro’ ja haun tuloksissa näkyvät ne tekijätietueet, joihin on liitetty asiasana tai asiasanan osa ’mieskuoro’. Käytettävä hakualgoritmi on niin kutsuttu katkaisuhakualgoritmi, joka hakee kaikki ne tietueet, jossa hakumerkkijono esiintyy, olipa kyseessä pelkkä sana tai sanan osa.

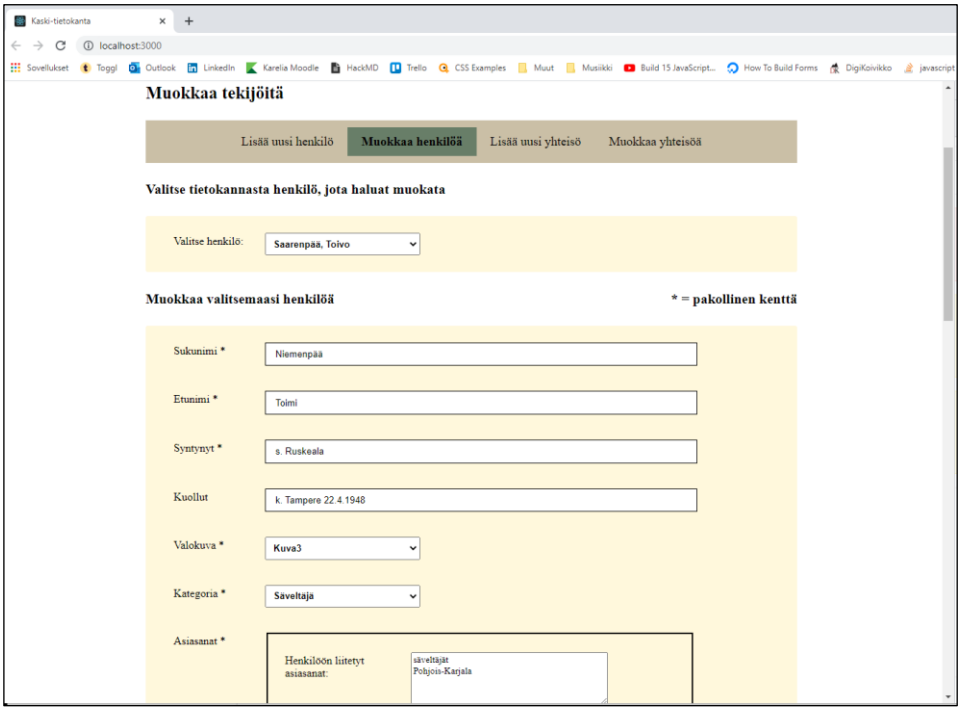
4.2.4 Ylläpitopuoli ja sen toiminnot

Seuraava vaihe käyttöliittymän koodaamisessa oli ylläpitopuolen alasivujen ulkonäön ja toimintojen rakentaminen. Jotta ylläpitosivujen luominen olisi mahdollisimman helppoa, kirjautumistoimintojen koodaaminen jätettiin viimeiseksi eli kaikkien muiden ylläpitopuolen toiminnallisuuksien jälkeen toteutettavaksi. Kehitettävän sovelluksen testaajan ei siten tarvitsisi jatkuvasti uudelleenkirjautua sisään sovellukseen pienienkin ylläpitopuolen toimintojen testaamiseksi.

Tärkeimmät toiminnot ylläpitopuolella tulevat olemaan uusien musiikintekijöiden luominen tietokantaan ja tietokannassa jo olemassa olevien tekijöiden muokkaus. Musiikintekijäkokoelmat (näihin kokoelmiin lasketaan sekä tietokannassa olevat yhteisökokoelmat että henkilökokoelmat) ovat isoimmat ja keskeisimmät tietorakenteet Kaski-tietokannassa. Niiden sisältämä tieto tulee myös ensimmäisenä näkyviin loppukäyttäjälle eli kirjaston asiakkaalle joko luettelon tai hakutulosten muodossa.

Koska uusien tekijöiden luominen ja vanhojen muokkaus vaatii toimiakseen paljon sovelluksen sisäistä dataa (asiasanavaihtoehdot, kuvavaihtoehdot, teosvaihtoehdot, äänitevaihtoehdot, ja suuriakin merkkijonoja sisältävät yksittäiset kentät) ja monia pieniä erillistoimintoja (asiasanojen liitos, teosten liitos, äänitteiden liitos, vastaavat poistot ja muokkaustoiminnallisuuksissa henkilön tai yhteisön valinta), kyseisiä toimintoja hallinnoivat lomakkeet päätettiin sijoittaa kokonaisuudessaan omiin komponentteihinsa ja tiedostoihinsa. Tämän on tarkoitus paitsi selkeyttää koodia, myös helpottaa lomakkeiden testausta niiden ollessa

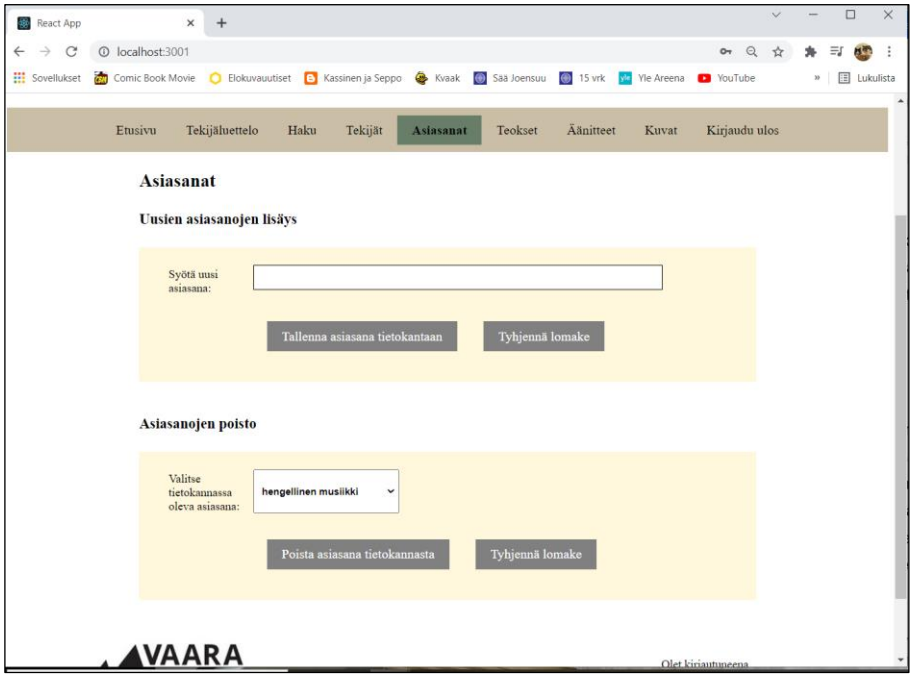
täysin omia moduulejaan. Lisäksi, koska henkilötietorakenteissa ja yhteisötietorakenteissa on jonkin verran eroavaisuuksia (sekä kenttien lukumäärässä että niiden ominaisuuksissa), myös henkilöiden ja yhteisöjen luominen sekä henkilöiden ja yhteisöjen muokkaus päätettiin sijoittaa kokonaisuudessaan omille erillisille lomakkeilleen.



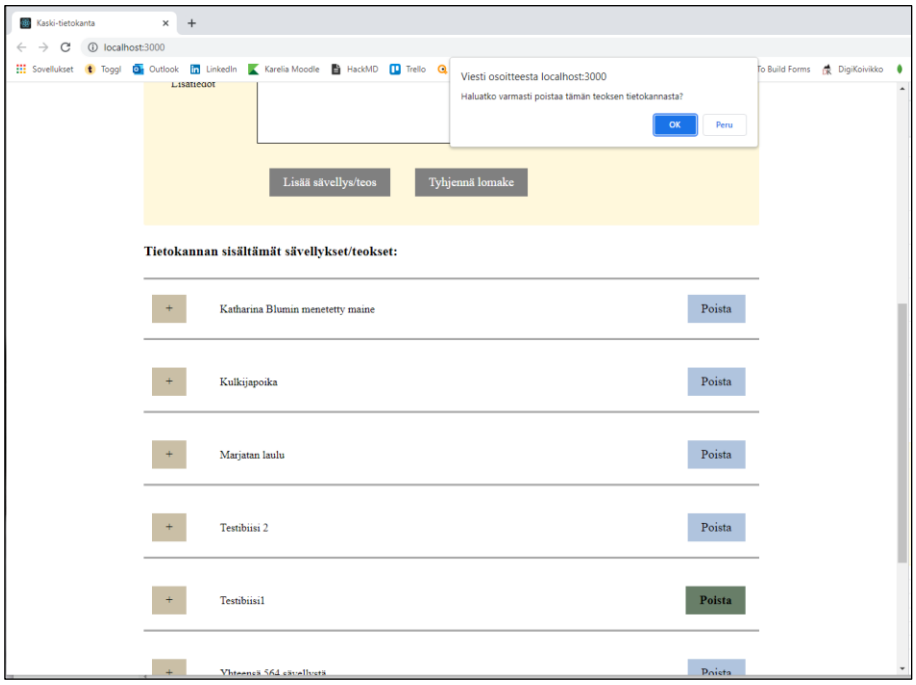
Kuva 11: Henkilön muokkauslomake.

Kuvassa 11 on nähtävillä osa henkilön muokkauslomakkeesta. Henkilön muokkaustoiminto valitaan ylhäällä olevasta palkista ja muokattava henkilö tämän alapuolella olevasta valikosta. Kyseisen henkilön nykyiset ominaisuudet tulevat näkyviin muokkauslomakkeen asianomaisiin kenttiin ja niitä voi muokata haluamallaan tavalla. Lopuksi muokattu henkilö tallennetaan tietokantaan.

Kun uusien tekijöiden luonti ja jo olemassa olevien tekijöiden muokkaus oli saatu toimimaan, siirryttiin asiasanasivun ulkoasun ja toimintojen rakentamiseen. Asiasanojen lisäykset ja poistot vastaavasta tietorakenteesta toteutettiin asiasana-alasivulle yksinkertaisina peruslomakkeina (nähtävillä kuvassa 12). Sivulla on kaksi yksinkertaista lomaketta, joista ylempi lisää asiasanan tietokantaan, ja alempi poistaa asiasanan tietokannasta.



Kuva 12: Asiasanojen käsittelysivu.

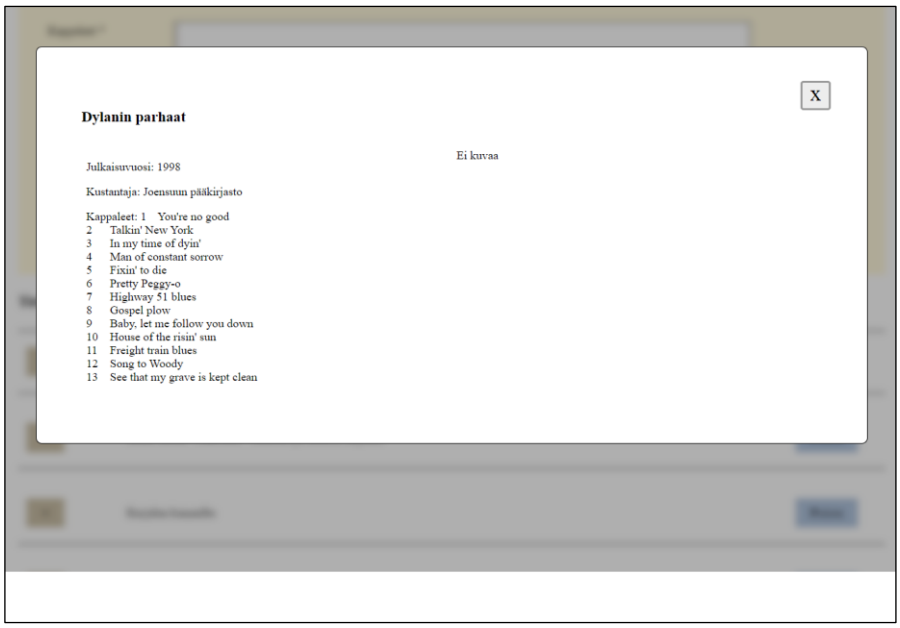


Kuva 13: Teosalasivu ja sivulla sijaitseva teosluettelo.

Seuraava vaihe ylläpitopuolella oli teosalasivun ulkoasun ja toiminnallisuuksien rakentaminen. Alasivulla jo olemassa olevat teokset eli sävellykset luetteloi-

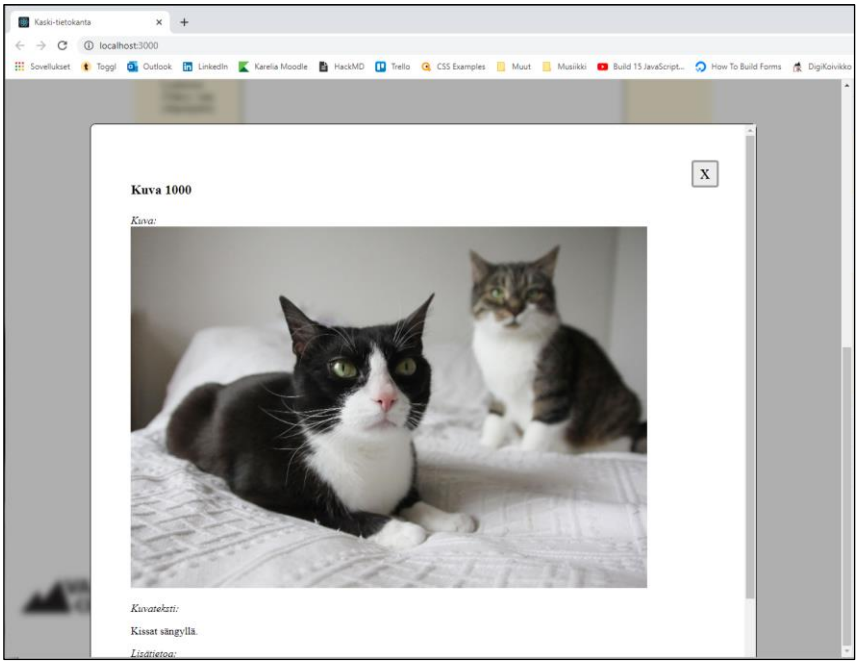
daan, uusia teoksia luodaan lomakkeen avulla, ja vanhoja teoksia poistetaan luettelossa sijaitsevalla erillispainikkeella. Esimerkkikuvassa 13 näkyy listauksen yläpuolella uuden teoksen luontilomake. Jokaisen teoksen kohdalla on ikkunanavauspainike (vasemmalla), josta tulee näkyviin teoksen kaikki tiedot sisältävä tietueikkuna, ja poistopainike (oikealla), joista yhtä on painettu.

Myös äänitealasivulle luotiin samanlainen ulkoasu ja samanlaiset listaukset ja toiminnallisuudet, aina tietueikkunasta poistopainikkeeseen. Kuvassa 14 on nähtävillä esimerkki avatusta äänitetietueikkunasta. Teos- ja ääniteluettelot eivät näy tietenkään suoraan kirjaston asiakkaalle, mutta ylläpitäjälle on tärkeää, että kyseiset luettelot ovat nähtävillä ja että teoksia ja äänitteitä on helppo selata ja hallinnoida. Tällöin niitä voidaan valita liitettäväksi henkilöihin tai yhteisöihin pelkästään nimen perusteella.



Kuva 14: Äänitteen tiedot nähtävillä tietueikkunassa.

Viimeinen ylläpitoalasivu, joka piti vaatimusmäärittelyjen mukaan rakentaa, oli kuva-alasivu. Kyseisellä alasivulla luetteloidaan edellisten sivujen tapaan tietokannassa olevat kuvat, tarjotaan mahdollisuus lisätä uusi kuva ja kuvatietue lomakkeella, sekä poistaa jo olemassa oleva kuva luettelossa sijaitsevalla erillispainikkeella.



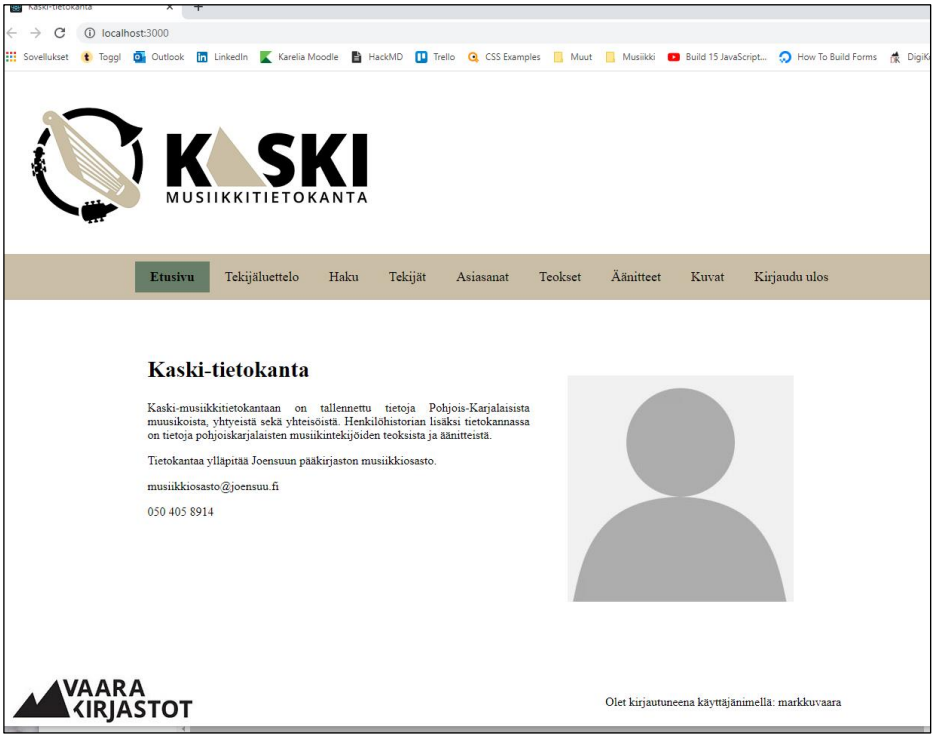
Kuva 15: Kuvatietueen esittely omassa ikkunassaan kuva-alasivulla.

Kuvalistauksessa on teos- ja äänitelistauksien tapaan painike, jonka avaamaan ikkunaan tulee näkyviin kyseinen kuva ja kaikki kuvaan liitetyt tiedot (esimerkki kuvassa 15). Huomionarvoista on, että kuvateksti tulee näkyviin kaikille käyttäjille, kun taas lisätiedot näkyvät vain ylläpitäjälle.

Kuvien hallinnointi oli kaikkien hankalin vaihe käyttöliittymän rakentamisessa, koska kuvan lisääminen tietokantaan ja hakeminen tietokannasta asettivat monenlaisia vaatimuksia toiminnallisuuksille. Kuva piti tallettaa merkkijonoksi muutettuna luotavaan kuvaolioon/tietueeseen niin, että se voitiin lähettää tietokantaan, edelleen hakea tietokannasta, ja lopuksi muuttaa lennosta takaisin kuvamuotoon selaimen näkymään. Lisäksi, toisin kuin muiden tietorakenteiden ja yksittäisten tietueiden kohdalla, kuvadataa käsitellessä palvelinpuolta ja tietokantaa oli rakennettava yhtä aikaa käyttöliittymän kanssa. Tämä siksi, että kuvan käsittelyä pystyttäisiin kokonaisvaltaisesti testaamaan ja samalla varmistamaan, ettei kuvainformaatio häviä missään vaiheessa sovelluksen toimintoja.

4.2.5 Kirjautuminen ja viimeistely

Kun kaikki ylläpitopuolen alasivut olivat valmiita ja toimivat vaatimusmäärittelyjen edellyttämällä tavalla, rakennettiin viimeinen osa käyttöliittymää eli sisäänkirjautumis-, uloskirjautumis- ja autentikointimekanismit. Myös nämä toiminnot luotiin hyödyntäen React-tiloja, joihin talletettiin syötetty käyttäjätunnus, syötetty salasana, ja käyttäjätunnusta ja salasanaa vastaavan, tietokannasta haetun, käyttäjän tiedot. Koska itse käyttäjätietorakenteita ei tietoturvasyistä kokonaisuudessaan haettu tietokannasta ja talletettu käyttöliittymään, vaan autentikointi tehtiin yksittäisenä pyyntönä palvelimeen, myös sisäänkirjautumisen koodaaminen ja testaaminen vaati toiminnallisuuksien luomista yhtä aikaa käyttöliittymään, palvelimeen ja tietokantaan.



Kuva 16: Kaski-musiikkitietokannan käyttöliittymän etusivu lähes lopullisessa muodossaan.

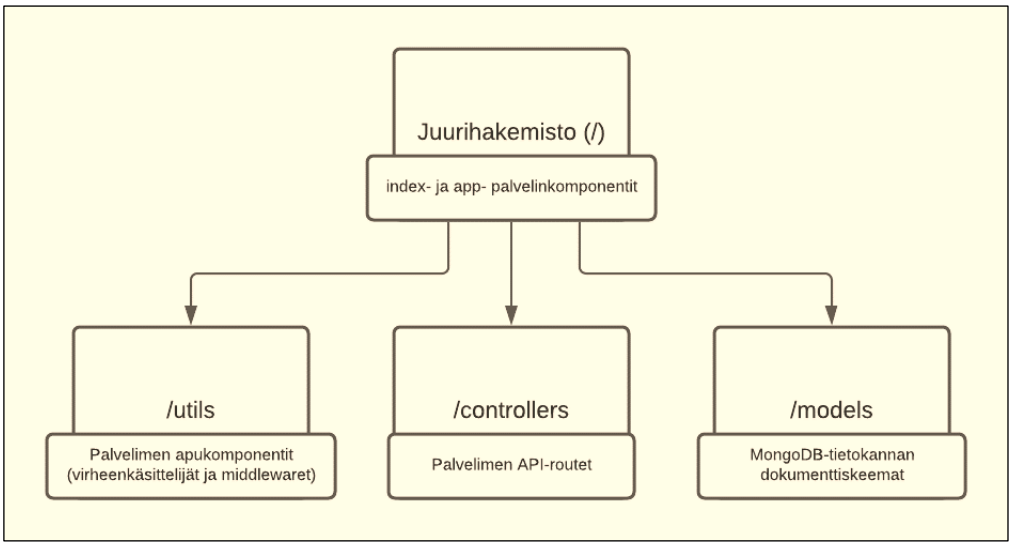
Viimeinen osa käyttöliittymän rakentamista oli ulkoasun viimeistely (esitetty kuvassa 16). Yleisulkoasu, navigointipalkki, yksittäiset painikkeet ja muut pienemmät yksityiskohdat muutettiin vastaamaan Joensuun Seutukirjastojen Digikoivikko-tietokannan käyttöliittymän ulkoasua, joka on toiminut yleismallina

muillekin pienimuotoisimmille Seutukirjastojen käyttöliittymäratkaisuille. Yläpalkkiin asetettiin myös Joensuun kaupungin graafikon sovellusta varten piirtämä Kaski-musiikkitietokannan logo. Logon ja itse sovelluksen värimaailma säädettiin vielä yhtenäiseksi. Esimerkkikuvassa käyttäjä on kirjautuneena sisään käyttäjätunnuksella ’markkuvaara’.

4.3 Palvelinpuolen toteutus

4.3.1 Yleistä palvelinpuolesta

Musiikkitietokantaa operoiva ja tietokannan sisältämiä tietoja siirtelevä palvelinpuoli rakennettiin käyttöliittymän tapaan Javascript-kielellä. Palvelintason koodauksessa hyödynnettiin Express-ohjelmistokehystä, joka on suosittu ja usein käytetty palvelinkirjasto. Yhteydet MongoDB-dokumenttitietokantaan hoidettiin puolestaan mongoose -kirjaston avulla. Salasanojen kryptaamiseen käytettiin bcrypt-kirjastoa, ja autentikointitokenin luomisessa ja sen käyttämisessä käyttäjätietojen tarkistuksessa hyödynnettiin suosittua ja luotettavaa jsonwebtoken-kirjastoa.



Kuva 17: Palvelimen hakemistorakenne.

Palvelintason hakemistorakenne (kuva 17) luotiin niin, että juurihakemistossa sijaitsevat palvelintiedostot (index.js ja app.js), ja utils -alihakemistossa erilaiset palvelimen apukomponentit eli niin kutsutut middlewaret (esimerkiksi virheenkäsittelijä on tällainen aputoiminnallisuus). HTTP-pyyntöjä käsittelevät API-reitit ja tietokannassa käytettäviä tietorakenteita mallintavat dokumenttimallit eriytettiin omiin tiedostoihinsa ja hakemistoihinsa, jotta niitä voitaisiin kätevästi käyttää ja operoida erillään toisistaan. Palvelinpyyntöreitit sijoitettiin controllers -alihakemistoon, ja models -alihakemistoon puolestaan varastoitiin edellä mainitut MongoDB-tietokannan operoinnissa käytettävät dokumenttimallit.

4.3.2 Dokumenttimallit ja reitittäminen

Palvelinkehitystyön alkuvaiheessa rakennettiin yksinkertainen palvelin, joka käytti toiminnassaan hyväksi vain kahta dokumenttimallia (person ja organization), jotka mallintavat tietokannassa säilytettäviä henkilö- ja yhteisötietueita, sekä kahta API-reittiä, edellisten mallien mukaisia henkilöitä ja yhteisöjä tietokannasta hakevia get-reittejä. Kun nämä mallit ja reitit toimivat eli kykenivät hakemaan testitietokannasta halutun mallin mukaista dokumenttidataa palvelintasoon, lähdettiin monimutkaistamaan palvelimen rakennetta lisäämällä uusia dokumenttimalleja, kuten asiasanamalli, teosmalli, äänitemalli, ja niin edelleen. Samalla lisättiin uusia hakureittejä, kuten asiasanahaku, teoshaku, äänitehaku, ja niin edelleen. Kun näidenkin havaittiin toimivan halutulla tavalla, lisättiin lopulta hakureittien oheen lisäysreittejä (post), poistoreittejä (delete), ja muutosreittejä (put), jokaiselle dokumenttimallille luonnollisesti erikseen.

Jokaisessa kehitysvaiheessa palvelinpuolta testattiin toimintojen virheettömyyden varmistamiseksi, ja kun kaikki reititykset ja mallit alkoivat olla valmiita, näitä toimintoja alettiin siirtää selkeyden vuoksi omiin tiedostoihinsa ja hakemistoihinsa (niin kutsuttu refaktorointi). Testauksia jatkettiin myös pitkin siirtovaihetta, jotta mitään vaikeasti havaittavaa ja korjattavaa virhettä ei pääsisi koodiin pesiytymään.

4.3.3 Käyttäjämalli ja autentikointimekanismit

Kun hakemistomallin refaktorointi oli saatu viimein tehtyä, luotiin viimeinen dokumenttimalli. Käyttäjän dokumenttimalli (user) mallintaa sovellukseen kirjautuneen käyttäjän eli ylläpitäjän tietoja, ja on tärkeä komponentti paitsi luotaessa uusi ylläpitäjä tietokantaan, myös varmistettaessa yksittäisen ylläpitäjän tietoja, kuten käyttäjätunnuksia ja salasanoja. Lisäksi luotiin myös kaksi viimeistä API-reittiä. Käyttäjäreitti mahdollistaa uusien ylläpitäjien luomisen, ja yksittäisen ylläpitäjän tietojen hakemisen ja tarkistamisen sisäänkirjautumisen yhteydessä. Login-reitti puolestaan sisältää itse sisäänkirjautumisen mekanismit.

Sovelluksen sisäänkirjautumis- ja autentikointitoiminnot vaativat palvelinpuolella paitsi edellä mainitun käyttäjämallin ja käyttäjä- ja login-reittien koodaamisen, myös tokeniksi kutsutun tunnistautumisavaimen luomisen ja tarkistamisen. Avain luodaan, kun sovellukseen kirjaudutaan sisään tietokannassa jo olevilla käyttäjätunnuksella ja salasanalla. Avain varmennetaan tämän jälkeen joka kerta, kun tehdään joitain vain ylläpitäjälle sallittuja tietokantaoperaatioita, esimerkiksi lisäyksiä, muutoksia tai poistoja.

Tokenin luominen ja sen tarkistaminen operaatioiden yhteydessä tehtiin hyödyntämällä jsonwebtoken-kirjastoa, joka luo ja varmentaa standardoituja JSON-muotoisia tokeneita. Lisäksi salasanan liittäminen käyttäjään uuden ylläpitäjän luomisen yhteydessä vaati bcrypt-kirjaston olemassaoloa. Bcrypt luo tiivistyssalatun (hashatun) salasanan ja suorittaa myös tämän salasanan tarkastuksen aina sovellukseen kirjautuessa.

4.4 Tietokannan suunnittelu

Musiikkitietokannan tietokantatyypiksi oli tarjolla kaksi vaihtoehtoa: SQL-kyselykieltä hyödyntävä relaatiotietokanta ja noSQL-tyyppinen dokumenttitietokanta. Molemmilla tietokantatyypeillä on omat hyvät ja huonot puolensa, joita käsitellään laajasti tämän opinnäytetyön teoriaosuudessa.

Kokonaisvaltaisen pohdinnan jälkeen päätettiin käyttää sovelluksen tietokantana dokumenttityyppistä tietokantaa. Kaski-musiikkitietokanta tulee sisältämään useita erilaisia tietokokoelmia (esimerkiksi yhteisöt, asiasanat ja kuvat), joista jokaisella on omat rakenteensa ja erityispiirteensä. Lisäksi tietueiden informaatiosisällön vaihtelu yksittäisenkin tietokokoelman sisällä voi olla suurta. Tämän tyyppisten tietorakenteiden rakentaminen ja käsitteleminen relaatiokantoina pitäisi siis tehdä jokaiselle kokoelmalle erikseen ja olisi aikaa vievä ja työläs prosessi.

Dokumenttitietokanta puolestaan on hyvin joustava tietokantatyyppi, eikä dokumenttikokoelmia käsiteltäessä vaadita syvempää tietämystä kokoelman rakenteesta ja toiminnasta, vaan tietokantamoottori osaa operoida niitä automaattisesti niiden sisäisestä koostumuksesta huolimatta. Toki dokumenttitietokannan käsittely vaatii hieman enemmän koodattuja toiminnallisuuksia itse käyttöliittymässä ja palvelinpuolella, mutta tämä on pieni hinta käytön vaivattomuudesta.

Dokumenttitietokantana päätettiin lopulta käyttää MongoDB-tietokantaa. MongoDB on nopea ja helppokäyttöinen, sen sisältöön päästään käsiksi kaikilla suosituilla ohjelmointikielillä (myös Javascriptillä) ja sen käyttämä datamalli muistuttaa standardoitua JSON-tiedostomuotoa, mikä helpottaa tiedon muuntamista käyttöliittymän ymmärtämään muotoon. MongoDB valikoitui käytettäväksi myös siitä syystä, että se on hyväksi havaittu ja yleisesti käytetty dokumenttitietokantamoottori ja koska sen käytöstä oli jo aikaisempaa kokemusta.

4.5 Tietokannan toteutus

Kun oli tehty päätös siitä, minkä tyyppinen tietokanta Kaski-tietokannasta oli tarkoitus tulla, ja palvelinpuolen kehittämisessä oli edetty niin pitkälle, että palvelintoimintoja oli tarpeen ryhtyä testaamaan, pystytettiin niin sanottu testitietokanta. Testikanta luotiin MongoDB Atlas:iin, joka on pienimuotoisia MongoDB-dokumenttitietokantoja ylläpitävä ilmainen verkkosovellus. Kyseinen testitietokanta

tarvittiin paitsi testaamaan rakennettavan palvelimen toimintoja, myös ylläpitämään testidataa, jotta sitä ei tarvitsisi luoda jokaista testauskertaa varten uudelleen.

Varsinainen musiikkitietokanta oli sovelluksen valmistuttua tarkoitus sijoittaa Joensuun Seutukirjastojen tietokantoja ylläpitävän Meita Oy:n palvelimelle. Palvelimen sisällä tietokanta ja lopulta koko sovellus oli lisäksi tarkoitus asentaa niin kutsuttuun Docker-konttiin, joka muodostaa standardoidun ajo- ja siirtoympäristön siihen sijoitetulle sovellukselle. Valitettavasti aika kuitenkin loppui kesken ja itse lopullista tietokantaa ja sen dockerointia ei ehditty toteuttaa.

Musiikkitietokannan luominen olisi vaatinut vielä paitsi itse tietokannan pystyttämistä, myös palvelinpuolen säätöä niin, että se olisi pystynyt tietoturvallisesti keskustelemaan Docker-kontissa sijaitsevan, muusta maailmasta eristetyn dokumenttitietokannan kanssa. Tämä säätäminen olisi ollut oma taiteenlajinsa, ja vaatinut, kuten edellä mainittiin, enemmän aikaa ja työpanosta kuin lopulta oli käytettävissä. Kyseinen varsinaisen tietokannan luominen ja sovelluksen palvelinpuolen muokkaaminen halutulla tavalla jäivätkin lopulta tämän opinnäytetyön ulkopuolelle ja tässä vaiheessa vain jatkokehitysmahdollisuuksiksi.

4.6 Testaus

Testaaminen on hyvin tärkeä osa ohjelmistokehitystä. Testaus käsitteenä jaetaan usein neljään eri tasoon tai kokonaisuuteen: Yksikkö-, integraatio-, järjestelmä- tai hyväksymistestaukseen. Yksikkötestaus keskittyy yksittäisiin toiminnallisuuksiin, esimerkiksi uuden henkilön tallentamiseen käyttöliittymän muistiin, sivun vaihtumiseen sovelluksessa tai muihin vastaaviin funktioihin tai komponentteihin. Integraatiotestaus käsittää puolestaan useamman tällaisen komponentin toiminnan yhdessä tai kokonaisen sovelluksen osan, kuten käyttöliittymän, testauksen.

Järjestelmätestaus taas testaa koko sovelluksen toimivuutta, aina käyttöliittymästä ja palvelimesta tietokantaan asti. Tällaisessa testauksessa voidaan usein kokeilla, miten sovelluksen eri osat toimivat yhdessä, esimerkiksi kuva voidaan tallettaa tietokantaan, hakea sieltä takaisin ja samalla tarkastella keskustelevatko sovelluksen eri osat kunnolla keskenään, jotta tämä toiminnallisuus tekee sen minkä lupaa. Hyväksymistestaus taas on viimeinen kokonaisvaltainen testaus, joka tehdään, kun sovellus on täysin valmis, ja jossa varmennetaan sovelluksen toimivuus ennen sen lähettämistä asiakkaalle tai muuten yleiseen käyttöön.

Musiikkitietokantasovelluksen testausta tehtiin yksikkö-, integraatio- ja järjestelmätasolla. Useita erilaisia yksikkötestejä tehtiin käyttöliittymän toimintojen testaamiseksi pelkästään käynnistämällä kehitysversio ja kokeilemalla yksittäisiä ominaisuuksia. Testauksen onnistuminen voitiin varmentaa joko välittömästi käyttöliittymässä tai Google Chromen kehitystyökaluja hyödyntäen. Palvelimen yksikkötestaaminen toimi samalla tavalla, tosin tässä oli apuna Postman-sovellus, joka mahdollisti HTTP-pyyntöjen lähettämisen ja vastaanottamisen suoraan palvelintasoon. Testitietokanta piti tietenkin tässä vaiheessa olla toiminnassa, koska muuten palvelinkomponenttien toimintaa ei olisi voinut täysin varmentaa.

Integraatiotestauksia tehtiin myös yllä mainituilla tavoilla sekä käyttöliittymässä että palvelinpuolella. Kaikkia käyttöliittymän ja palvelintason toiminnallisuuksien testauksia ei toki voitu tehdä, ennen kuin koko sovellus oli toiminnassa ja kaikista sovelluksen osista oli toimivat yhteydet muihin osiin, missä vaiheessa siirryttiin kokonaisvaltaiseen järjestelmätestaukseen. Tässäkin vaiheessa hyödynnettiin käyttöliittymää, Google Chromen kehitystyökaluja, Postman-sovellusta ja itse testitietokantaa, jotta mahdollisen virheen sattuessa tai sovelluksen toimiessa oudosti pystyttäisiin välittömästi paikantamaan virhelähde ja korjaamaan se nopeasti ja mahdollisimman pienellä vaivalla.

Järjestelmätestausta suorittivat osaltaan myös kirjaston henkilökunnan edustajat, eli tulevat musiikkitietokannan ylläpitäjät. Onkin ehdottoman välttämätöntä,

että kehitystestausta suorittaa useampi kuin yksi henkilö, ja varsinkin joku muukin kuin kehitystyöstä päävastuussa oleva kehittäjä. Tällöin vältytään virheiltä, joita tulee, kun kehittäjä tulee niin kutsutusti sokeaksi omalle työlleen. Tietokannan tulevat ylläpitäjät voivat myös huomata kehitettävästä sovelluksesta virheitä ja outoja tai keskeneräisiä toiminnallisuuksia, jotka voivat jäädä korjaamatta sellaiselta, jonka ei ole tarkoitus käyttää sovellusta jatkuvasti tai useaan otteeseen omassa työssään.

Javascriptiin on saatavilla useita testikirjastoja, esimerkiksi Jest, Mocha ja Ava. Jestiä käytetään usein React-komponenttien ja Reactilla koodattujen käyttöliittymien yksikkö- ja integraatiotestauksessa, ja sitä olisi voinut tässäkin opinnäytetyössä hyödyntää. Aika oli kuitenkin lopulta Kaski-tietokannan sovelluskehityksessä kortilla, ja Jestin käyttö olisi vaatinut jonkin verran opettelua, joten kaikki testaukset päätettiin suorittaa ilman tätä testikirjastoa. Epävarmaa toki on, olisiko Jest-testaus ollut sittenkin hyödyllisempi tai tehokkaampi testaustapa, mutta kaikki tässä osiossa mainitut testaukset pystyttiin kyllä suorittamaan halutulla tavalla ja halutussa mittakaavassa.

5 Työn tulokset

5.1 Työn lähtökohdat

Opinnäytetyön toimeksiannon mukaan työssä oli tarkoituksena pystyttää pohjoiskarjalaisten musiikintekijöiden tietoja sisältävä tietokanta ja luoda tämän tietokannan tietoja tavalla tai toisella käsittelevä käyttöliittymä. Tietokannan luomiseen ja käyttöliittymän rakentamiseen toimeksianto antoi suhteellisen vapaat kädet, ja ohjelmointikieliksi sekä tietokantarakenteiksi valikoituivat aikaisemmista opiskeluprojekteista tutut ja omaan kokemuspohjaan parhaiten soveltuvat Javascript, React, sekä MongoDB-dokumenttitietokanta.

Kokemukset näistä kielistä ja tietorakenteista eivät tietenkään tarkoittaneet, että kaikki näihin liittyvät kehitystyötavat olisivat olleet ennestään tuttuja. Esimerkiksi palvelintason koodaamisesta Javascriptillä oli vähemmän kokemusta kuin käyttöliittymäpuolesta, mutta tämäkin puoli tuli paremmin tutuksi opinnäytetyön aikana.

5.2 Käyttöliittymä

Käyttöliittymä saatiin rakennettua lähes täysin vaatimusmäärittelyjen mukaan. Vaatimusmäärittelyissä piti ei-kirjautuneen käyttäjän (käytännössä siis kirjaston asiakkaan) pystyä selaamaan käyttöliittymän kautta tietokannan sisältämiä tietoja, olivat ne sitten henkilötietoja, yhteisötietoja, henkilöihin/yhteisöihin liitettyjä teostietoja, henkilöihin/yhteisöihin liitettyä äänitedataa tai henkilöihin/yhteisöihin liitettyjä kuvia. Teokset, äänitteet ja kuvat, vaikkakaan eivät olleet sellaisenaan selattavissa, olivat aina liitettyinä henkilöihin ja yhteisöihin, joten tietystä näkökulmasta ne olivat myös suoraan kirjaston asiakkaan nähtävillä. Joka tapauksessa käyttöliittymä oli lähes täysin vaatimusmäärittelyjen mukainen, ja kaikki data oli nähtävissä selkeästi ja yksityiskohtaisesti tekijäluettelon kautta.

Edellisen toiminnallisuuden lisäksi ei-kirjautuneen käyttäjän piti kyetä suorittamaan hakuja tietokannasta, joko yhteisön nimen, henkilön nimen, tai asiasanan mukaan. Kaikki nämä haut pystyttiin suorittamaan käyttöliittymän kautta katkaisuhaulla, eli nimen tai sanan osakin riitti siihen, että tietty tekijä oli osa hakutuloksia. Nämäkin vaatimusmäärittelyjen mukaiset toiminnot toimivat siis moitteettomasti.

Lopuksi ei-kirjautuneen käyttäjän piti kyetä suorittamaan hakuja vapaasanalla, eli käytännössä sovelluksen piti löytää nimellä, sanalla, tai sanan osalla kaikki ne tekijät, joihin sisältyvästä datasta tuo merkkijono löytyi, oli data joko mistä tahansa tekijäkentästä, tai mistä tahansa tekijään liitetyn teoksen kentästä, tai

mistä tahansa tekijään liitetyn äänitteen kentästä, tai mistä tahansa tekijään liitetyn kuvan kentästä. Vapaasanahaku oli siis kaikkein kattavin kaikista hakutoiminnoista ja työläs toteuttaa, mutta sekin toimi lopulta täysin halutulla tavalla.

Edellisten, ei-kirjautuneelle käyttäjälle näkyvien toiminnallisuuksien lisäksi käyttöliittymään piti vaatimusmäärittelyjen mukaan rakentaa myös ylläpitopuoli sisään kirjautuneelle käyttäjälle (eli kirjaston henkilökunnalle). Ylläpitopuolen toimintoihin sisältyivät henkilöiden lisäys ja muokkaus, yhteisöjen lisäys ja muokkaus, asiasanojen lisäys ja poisto, teoksien lisäys ja poisto, äänitteiden lisäys ja poisto, ja kuvien lisäys ja poisto. Nämä kaikki saatiin rakennettua käyttöliittymään täysin vaatimusmäärittelyjen mukaisella tavalla.

Vaatimusmäärittelyssä mainittiin myös henkilöiden ja yhteisöjen poistot, mutta ne jätettiin pois käyttäjärajapinnasta, koska aika loppui kesken ja ne eivät olleet täysin välttämättömiä: Kun pohjoiskarjalaisia henkilöitä tai musiikkiyhteisöjä talletetaan kyseiseen tietokantaan, harvemmin käy niin, että niitä pitäisi syystä tai toisesta poistaa. Poisto onnistuu tarvittaessa valmiissa sovelluksessa niin, että musiikintekijä muokataan muokkaustoiminnallisuudella täysin toiseksi, tai sitten poisto tehdään suoraan palvelintasoon (vaikka näitä poistotoiminnallisuuksia ei ehditty koodata käyttöliittymään, ne ovat kuitenkin olemassa palvelinpuolella).

5.3 Palvelin

Palvelinpuolta ei mainittu vaatimusmäärittelyissä millään tavalla, mutta se piti toki rakentaa sillä tavalla, että se tukisi täysin kaikkia käyttöliittymän sisältämiä toiminnallisuuksia ja sisältäisi vastaavat yhteydet tietokantaan. Lisäksi kaikkien toimintojen pitäisi tehdä tietokannassa kaikki se, minkä ne käyttöliittymässä lupasivat. Nämäkin palvelinpuolen vastaavat toiminnot koodattiin ja saatiin toimimaan halutulla tavalla.

Koska vaatimusmäärittelyissä mainittiin alun alkaenkin, että käyttöliittymässä oli oltava erikseen tietokantatietojen muokkaamisen mahdollistava ylläpitopuoli,

käyttöliittymään ja palvelintasoon piti rakentaa kaiken edellä mainitun lisäksi toiminnallisuudet ylläpitäjien luomiselle, sisäänkirjautumiselle, uloskirjautumiselle, ja kaikkien ylläpitotoimintojen autentikoinnille. Sisään- ja uloskirjautumistoiminnot rakennettiin täysin toimiviksi sekä käyttöliittymä- että palvelinpuolelle. Lisäksi palvelinpuolelle rakennettiin ylläpitäjän luomismekanismit: Näitä ei ajan puutteen vuoksi ehditty toteuttaa itse käyttöliittymään, mutta asiansa osaava ja oikeita sovelluksia käyttävä autentikoitu loppukäyttäjä osaa kyllä luoda uuden ylläpitäjän pelkästään lähettämällä ylläpitäjän tiedot suoraan palvelintason kautta tietokantaan. Kaikkien ylläpitotoimintojen autentikointi sisäänkirjautumisen yhteydessä luotavien tokenien eli varmistusavainten avulla puolestaan jäi vielä kokonaisuudessaan tekemättä, ja tältä osalta sovelluksen tietoturvavarmuus ei ole vielä hyvää tasoa.

6 Pohdinta

Kuten edellä mainittiin, sovelluksen käyttöliittymä ja palvelinpuoli saatiin opinnäytetyön aikana lähes täysin valmiiksi. Ainoa isompi toiminnallisuus, joka jäi ajanpuutteen vuoksi tekemättä, oli varmistusavaimen (tokenin) käyttö ylläpitotoimintojen varmistamisessa. Vaikka ylläpitotoimintoja ei käyttöliittymän kautta edelleenkään pysty käyttämään kirjautumatta sovellukseen tietokantaan tallennetuilla käyttäjätunnuksella ja salasanalla, niiden operointi suoraan palvelintason kautta sopivalla HTTP-pyyntöjä generoivalla sovelluksella on edelleen mahdollista. Tämä on vakava tietoturva-aukko, joka on syytä korjata välittömästi jatkokehityksen aikana.

Toinen suuri puute on Docker-toiminnallisuuksien puuttuminen palvelinpuolen Javascript-koodista. Kyseiset toiminnat vaaditaan siihen, että Docker-ajoympäristöön pystytetty sovellus pystyy kommunikoimaan toiseen samanlaiseen konttiin pystytetyn tietokannan kanssa. Dockeroinnin puuttuminen koodista ei johtunut siitä, että itse opinnäytetyössä aika olisi loppunut kesken, vaan siitä, ettei

Joensuun Seutu-kirjastojen tietokantoja ylläpitävä Meita Oy ehtinyt luoda rakennettavalle sovellukselle ja dokumenttitietokannalle Docker-ajoympäristöjä opinnäytetyölle tarkoitetun aikaikkunan puitteissa. Tällöin myös tietokannan pystytys ja koodin Docker-yhteensopivuuden rakentaminen jäi kokonaan tekemättä ja testaamatta.

Sovelluksen testaaminen yhdessä tietokannan kanssa onnistui sinänsä, koska testitietokantana pystyttiin käyttämään MongoDB Atlas -pilvipalveluun luotua dokumenttitietokantaa, joka oli ominaisuuksiltaan lähes identtinen varsinaisen lopputuotteeseen yhdistettäväksi tarkoitetun tietokannan kanssa. Konttiajoympäristön vaatimat toiminnallisuudet jäivät kuitenkin puuttumaan, ja nämä ovatkin luonnollisesti ensimmäiset oleelliset jatkokehityksen aikana luotavat toiminnot, heti tietokannan pystytyksen ja varmistusavaintoimintojen jälkeen.

Tämän opinnäytetyön aikana tuli opittua paljon verkkosovelluksen kehitystyöstä, sekä teoriasta että käytännön tekemisestä. Käyttöliittymän koodaaminen Reactilla oli toki jo ennestään tuttua, mutta lisäkokemus aiheesta on aina hyödyllistä ja joitain uusia näkökulmia ja koodaustapoja tuli opittua. Palvelinpuolen luominen Javascriptillä oli taas ennen tuntematonta, ja siitä tuli opittua aika kattavat perusteet. Myös kokonaisen verkkosovelluksen kehitystyö eli niin kutsuttu full stack -kehitys (käyttöliittymän, palvelinpuolen ja tietokannan integraatio) tuli tutuksi ja paitsi avasi täysin uusia näkökulmia sovelluksien luomisprosesseihin, antoi myös hyvät eväät oman osaamisen kehittymiseen.

Lähteet

Bachman, CW. 1973. The Programmer as Navigator. Communications of the ACM 16 (11), 653–658.

Bandara, HMND. & Jayasumana, AP. 2012. Collaborative Applications over Peer-to-Peer Systems – Challenges and Solutions. Peer-to-Peer Networking and Applications. 6 (3), 257–276.

Chamberlin, DD. & Boyce, RF. 1974. SEQUEL: A Structured English Query Language. Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control. Association for Computing Machinery, 249–264.Tekijänoikeuslaki 404/1961.

Codd, EF. 1970. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13 (6), 377–387.

Connolly, TM. & Begg, CE. 2014. Database Systems – A Practical Approach to Design Implementation and Management (6th ed.). Pearson. p. 64.

DB-Engines. 2021a. DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms. 26.10.2021.

DB-Engines. 2021b. DB-Engines Ranking of Document Stores. https://db-engines.com/en/ranking/document+store. 26.10.2021.

Drake, M. 2014. A Comparison of NoSQL Database Management Systems and Models. The DigitalOcean Community. 21.2.2014. https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models. 28.10.2021.

Encyclopaedia Britannica. 2021. Database. https://www.britannica.com/technology/database. 30.10.2021.

ExpressJS. 2021. Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/. 30.10.2021.

Fielding, RT., Gettys, J., Mogul, JC., Nielsen, HF., Masinter, L., Leach, PJ. & Berners-Lee, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. IETF. 26.10.2021.

Flanagan, D. 2020. JavaScript - The definitive guide (7 ed.). p. 1.

Haerder, T. & Reuter, A. 1983. Principles of transaction-oriented database recovery. ACM Computing Surveys 15 (4), 287-317.

InfoQ. 2018. Deno: Secure V8 TypeScript Runtime from Original Node.js Creator. 26.12.2018. https://infoq.com/news/2018/12/deno-v8-typescript/. 30.10.2021.

Leavitt, N. 2010. Will NoSQL Databases Live Up to Their Promise? IEEE Computer 43 (2), 12–14.

Mahemoff, M. 2009. Server-Side JavaScript, Back with a Vengeance. Readwrite. 17.12.2009. https://readwrite.com/2009/12/17/server-side_javascript_back_with_a_vengeance/. 28.10.2021.

ReactJS. 2021a. React. A JavaScript library for building user interfaces. https://reactjs.org/. 27.10.2021.

ReactJS. 2021b. Components and Props. https://reactjs.org/docs/components-and-props.html/. 27.10.2021.

ReactJS. 2021c. Rendering Elements. https://reactjs.org/docs/rendering-elements.html/. 27.10.2021.

ReactJS. 2021d. Introducing JSX. https://reactjs.org/docs/introducing-jsx.html/. 27.10.2021.
ReactJS. 2021e. Introducing Hooks. https://reactjs.org/docs/hooks-intro.html/. 27.10.2021.
Tanenbaum, AS. & Steen, M. 2002. Distributed systems: Principles and paradigms. Upper Saddle River, NJ: Pearson Prentice Hall.
W3Techs. 2021a. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript/. 26.10.2021.
W3Techs. 2021b. Usage statistics of JavaScript for websites. 26.10.2021.