Rakenteellisen teknisen velan ilmentyminen tietomallin vaatimusten muuttuessa

Diplomityö
Turun yliopisto
Tietotekniikan laitos
Ohjelmistotekniikka
2023
Tuukka Aro

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck-järjestelmällä.

TURUN YLIOPISTO
Tietotekniikan laitos

Tuukka Aro: Rakenteellisen teknisen velan ilmentyminen tietomallin vaatimusten muuttuessa

Diplomityö, 67 s., 4 liites.
Ohjelmistotekniikka
Kesäkuu 2023

Koodin laatu tunnustetaan yleisesti keskeiseksi uhaksi, mutta sitä tärkeämmiksi oletetaan esimerkiksi vaatimusten muuttuminen, arvioiden epätarkkuus ja tekijöiden tuottavuus, jotka käytännössä muodostuvat ongelmiksi vasta ryvettyneen koodin kautta, ja vain siten. Arviot ovat aina luonnostaan aluksi epätarkkoja ja vaatimukset muuttuvat, mutta laadukas ja kompakti koodi estää niiden muodostumisen ongelmaksi ja lienee usein se mahdollisten epäonnistumisten todellinen ydin, johon pitäisi tarttua. Tämän tutkimuksen motivaationa on ajatus siitä, että tietomallin toteutuksella moderneissa verkkosovelluksissa on parannettavaa näiden haasteiden torjumiseksi. Rakenteelliset piirteet kerrostuneessa full stack -arkkitehtuurissa vaikuttavat tarpeeseen toteuttaa sovelluksen tietomalli uudestaan ja uudestaan lähdekoodin eri osiin, jolloin kaikki tietomallin vaatimusten muutoksista kehkeytyvät koodin muutokset propagoituvat ympäri lähdekoodia. Tutkimuksen tavoitteena on kartoittaa tätä ongelmaa nykyään laajasti käytössä olevien verkkosovellusteknologioiden kannalta. Tutkimuksessa selviää, että minkälaiset teknologiapinot ovat suosittuja verkkosovelluksissa ja kuinka monta duplikoitunutta tietomallin toteutusta näiden teknologiapinojen rakenteeseen sisältyy.

Asiasanat: full stack, verkkosovellukset, tekninen velka, tietomalli, tietorakenne

Sisällys

Kuvat

Taulukot

1 Johdanto

Modernin ketterän ohjelmistokehityksen pohjana toimineessa manifestissa[1] koodin laatu tunnustetaan yleisesti keskeiseksi uhaksi, mutta sitä tärkeämmiksi oletetaan esim. vaatimusten muuttuminen, arvioiden epätarkkuus ja tekijöiden tuottavuus, jotka käytännössä muodostuvat ongelmiksi vasta ryvettyneen koodin kautta, ja vain siten. Arviot ovat aina luonnostaan aluksi epätarkkoja ja vaatimukset muuttuvat, mutta laadukas ja kompakti koodi estää niiden muodostumisen ongelmaksi ja on aina se mahdollisten epäonnistumisten todellinen ydin, johon pitäisi tarttua. Tässä suhteessa yksi selkeimmistä tavoista jolla koodin laatu ilmenee, on sen duplikoituminen.

Motivaationa tälle tutkimukselle on tietoisuuden laajentaminen laajalti käytössä olevien tietomalliratkaisujen mahdollisista rakenteellisista ongelmista. Tarkastelun kohteena on nimenomaan mahdollisten ongelmakohtien määrä eikä laatu. Työmääristä, tunneista, joustavuudesta, mahdollisuuksista jne. on turha teoretisoida, kun niitä ei saa samalla tavalla konkreettisiksi numeroiksi kuin koodirivien ja niihin sisältyvien kuvausten suhteiden määrää. Kysymys kuuluukin, missä määrin käytössä olevissa teknologioissa esiintyy tietomalliin liittyvää koodin duplikoitumista havainnoiltavalla tavalla, ja voitaisiinko perustella ongelmaksi.

Normimenettely full stack -toteutuksessa on se, että aloitetaan koodaamaan kuten kunkin toteutustason valittujen määritelmien ja kirjastojen dokumenteissa suositellaan. Siten päästään nopeasti alkuun ja tehdään periaatteessa kaikki oikein,

mutta jos sitä kaikkea koodia tarkastelee kokonaisuutena, sama tietorakenne toistuu ympäri lähdekoodia ja lisäksi kullakin tasolla tietomallin samankaltaiset piirteet duplikoituvat koodissa ellei niitä toteuteta ennakoivasti kutsurajapintojen sisään.

Tässä voi olla pari merkittävää ongelmaa. Ensinkin ne toistuvuudet koodissa nostavat lähdekoodin määrän moninkertaisesti suuremmaksi kuin saman toiminnallisuuden toteutus olisi täysin ilman duplikaatiota. Lähdekoodin määrä korreloi jokseenkin suoraan virheiden määrän kanssa ja käänteisesti ylläpidon tuottavuuden kanssa, eli laatu ja tuottavuus jäävät kauas ideaalista. Tästä päästään toiseen ongelmaan, eli kun tietomalli on projektia aloitettaessa alustava ja sitä pitää voida ylläpitää, ylläpidon tuottavuutta heikentää se, että se pitää tehdä kaikkiin kerroksiin missä kukin tietomallin osa näkyy ja verkkorajapinnan taaksepäin yhteensopivuus teettää vielä paljon ylimääräistä työtä. Ei siis pystytä kovin hyvin vastaamaan kehittyviin tarpeisiin ja tietomallin vaatimuksen muutoksiin.

Ongelman ratkaiseminen jää tämän tutkimuksen näkymän ulkopuolelle, mutta tavoitteena on kartoittaa sen laajuutta. Tämä toteutetaan ensin kartoittamalla niitä teknologioita ja ratkaisuja, jotka ovat yleisesti käytössä verkkosovelluksissa. Tämän kerätyn tiedon perusteella valitaan tarkasteltaviksi verkksovellusprojektit, jotka edustavat alan nykytilannetta. Näille valituille sovelluksille suoritettaan sitten tutkimus, jonka tarkoitus on havainnoida yllämainittuja haasteita sovellusten tietomallien toteutuksissa.

1.1 Tutkimuskysymykset

Tutkimus on jaettu kahteen erilliseen tutkimuskysymykseen. Näistä kysymyksistä ensimmäinen on tyypiltään deduktiivinen, joten sen tavoitteena on tiivistää kerättyä tietoa teoriaksi. Toinen kysymys on induktiivinen, joten sen tavoitteena on luoda yleistettävä teoria valittujen kohteiden tarkemmasta evaluoinnista.

K1: Minkälaisia teknologiapinoja on yleisessä käytössä verkkosovelluksien tietomallitoteutuksissa? Tähän kysymykseen saadun vastauksen perusteella valitaan ne verkkosovellukset, joiden kanssa seuravaa tutkimuskysymystä lähestytään.

K2: Missä kaikkialla tietomallin vaatimuksien muuttuminen ilmenee tyypillisissä verkkoprojekteissa? Lisää tietoa tutkimuksen menetelmistä löytyy kappaleesta 3 ja tutkimuskysymiksiin vastaamista koskevat käytännön asiat löytyvät kappaleista 4 ja 5.

Lisätavoite: Arvioida tutkimuskysymyksien vastauksien yhteyttä verkkosovelluksien rakenteelliseen tekniseen velkaan ja pohtia mahdollisuutta vähentää tätä velkaa.

1.2 Tutkimuksen rakenne

Tämän tutkumuksen rakenne koostuu johdannon sekä yhteenvedon lisäksi neljästä sisältökappaleesta. Jokaisen sisältökappaleen alusta löytyy lyhyt tiivistelmä sen tavoitteista ja havainnoista.

Kappale 2 keskittyy full stack kehitykseen liittyvään teoriaan. Tämän osion tarkoituksena on antaa lukijalle tarvittavat taustatiedot käsiteltävänä olevasta ongelmasta, jotta myöhemmissä osiosissa esiteltävät kokeet ja teoria ovat helppo ymmärtää. Käsiteltävänä olevia aiheita ovat tietomallit, rajapinnat ja kehitysprosessit. Tärkeänä osana myöhempien kappaleiden hyodyntämää teoriaa on keskustelu niistä tavoista, joilla sovelluksen vaatimukset muuttuvat kehityksen aikana.

Kappaleen 3 tarkoitus on käydä läpi tutkimuksessa käytettävät tutkimusmenetelmät ja tutkimusteoria. Tämä osio on edelleen jaettu pienempiin osiin, joiden aiheina ovat tutkimuksen lähestymistapa, käytetty tutkimusmetodi, sekä tutkimuksen osat.

Kappaleessa 4 vastataan tutkimuskysymykseen TK1 keräämällä kokoon GitHub:sta saatavilla olevaa dataa eri verkkosovellusten suosiosta. Kappale alkaa ly-

hyellä kuvauksella kappaleen tavoitteesta, jossa esitellään hieman missä muodossa tutkimuskysymykseen vastataan. Tämän jälkeen käydään läpi tulosten lajittelussa käytettävät termit ja menetelmät. Osion lopusta löytyy tutkimuksen toteutuksen läpikäynti, tutkimuksen tulokset sekä myös tulosten analyysi.

Kappale 5 sisältää tietomallin muutosta koskevan soveltavan tutkimuksen ja vastaa tutkimuskysymykseen TK2. Kappale alkaa lyhyellä kuvauksella kappaleen tavoitteesta, jossa esitellään hieman missä muodossa tutkimuskysymykseen vastataan. Alusta löytyy myös kuvaukset niistä neljästä projektista, jotka tähän tutkimukseen valittiin. Tämän jälkeen käydään läpi kohteille suoritettavan analyysin vaiheet. Osion lopusta löytyvät tämän tutkimuksen tulokset sekä näiden pohjalta johdetut johtopäätökset. Tähän tutkimuksen osaan valitut projektit perustuvat ensimmäisen tutkimuskysymyksen TK1 tuloksiin edellisessä kappaleessa.

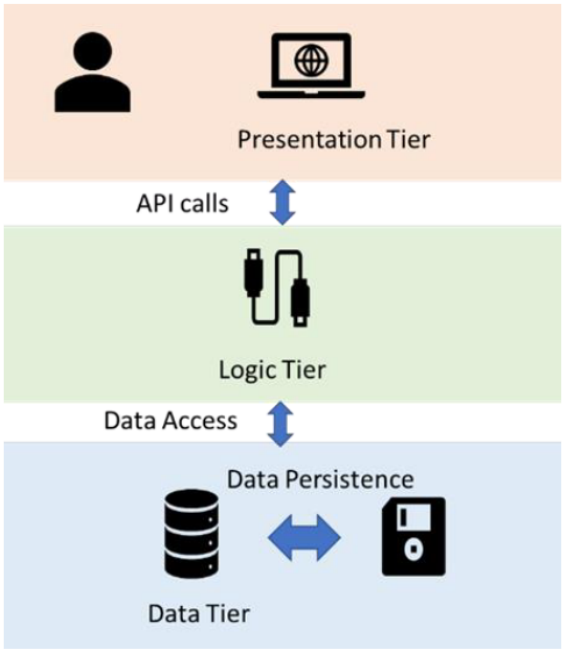
2 Verkkosovellusarkkitehtuuri

Tämän osion tarkoitus on kuvata ja taustoittaa verkkosovellusten rakenteellista ongelmaa, jonka mittaamiseen tämän tutkimuksen kokeellinen osuus keskittyy. Tätä varten esitellään tutkimuksen aiheeseen liittyviä käsitteitä siinä määrin, mikä on aiheen kannalta relevanttia. Nämä käsitteet ovat full stack -arkkitehtuuri, rajapinnat, vaatimukset ohjelmistokehityksessä ja tekninen velka. Lisäksi tämä osio kokoaa aikaisempien samaan aihealueeseen liittyvien tutkimuksien havaintoja.

Tässä tutkimuksessa tarkasteltavassa ja käsittelyssä olevassa lähdekoodin muutosten koodin muihin osiin vaikutuksessa ensisijaisena piirteinä lähdekoodin määrä kasvu ja duplikoituminen. Ongelman mahdollista realisoitumista tutkitaan vasta tutkimuksen myöhemmässä kokeellisessa osiossa, mutta tässä osiossa pohjustetaan sovellusten rakenteellisten piirteiden teoriaa, joka on myös osana myöhemmän kokeellisen osion hypoteesia. Jotta on mahdollista sisäistää tutkimuksessä myöhemmin käsiteltävien kysymyksien laatu sekä ne tavat, jolla koodin toistumista voi esiintyä laajasti käytössä olevia malleja toteuttavissa sovelluksissa, on ensin ymmärrettävä full stack -sovelluksissa esiintyvien käytäntöjen, rakenteiden ja protokollien kerrostuneisuus.

2.1 Full Stack -rakenne

Kaikkein yleisin korkealla tasolla abstrahoitu malli verkkosovelluksille on kolmikerroksinen esillepano-, logiikka- ja datakerroksista koostuva arkkitehtuuri (kuva 2.1).

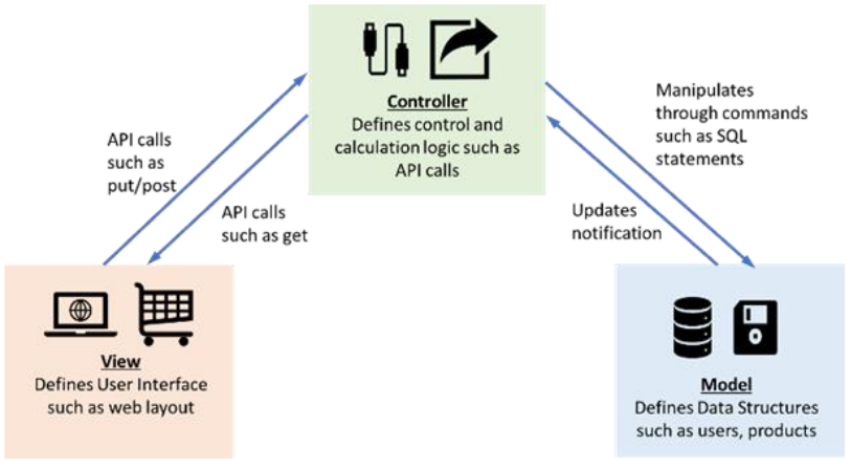


Kuva 2.1: Esillepano-, logiikka- ja datakerrokset[2]

[2] Tämän tutkimuksen kontekstissa englanninkielisellä termillä full stack viitataan nimenomaan tähän rakenteeseen, vaikka tällä termillä voidaan myös viitata erikseen sovelluskehittäjän ammattitaitoon tämän mallin perusteella rakennettujen sovelluksien kanssa työskentelyssä.

Kuvan 2.1 nuolet ilmaisevat interaktioita tietomallin toteutuksien välillä. Henkilön kuvake ilmaisee kerrosta, jonka kautta käyttäjä vaikuttaa tietomalliin. Levykkeen kuva ilmaisee, missä kerroksessa kiinteä tieto on tallennettuna. Nuolet kuvaavat verkkosovelluksen osia, joissa tätä sovellusarkkitehtuuria käyttävät ohjelmat sisältävät jonkinlaisia tietomallien toteutuksien vaikutuksia toisiinsa. Jokainen kerros siis vaatii jonkin tavan näyttää, muokata tai pitää tallessa tietomallin kuvaamaa rakennetta. Lisää tietomalleista kerrotaan seuraavassa kohdassa.

Tyypillinen ja erittäin laajasti[3] verkkosovelluksien käyttöliittymissä käytössä oleva toteutuksen arkkitehtuuri on nimeltään MVC (Model View Controller), joka näkyy kuvassa 2.2. [4, p. 71] Tässä mallissa abstraktion taso voidaan tulkita yllä mainittuun esillepano-, logiikka- ja datakerros -arkkitehtuuriin verrattuna joko alem-



Kuva 2.2: Model View Controller -malli[2]

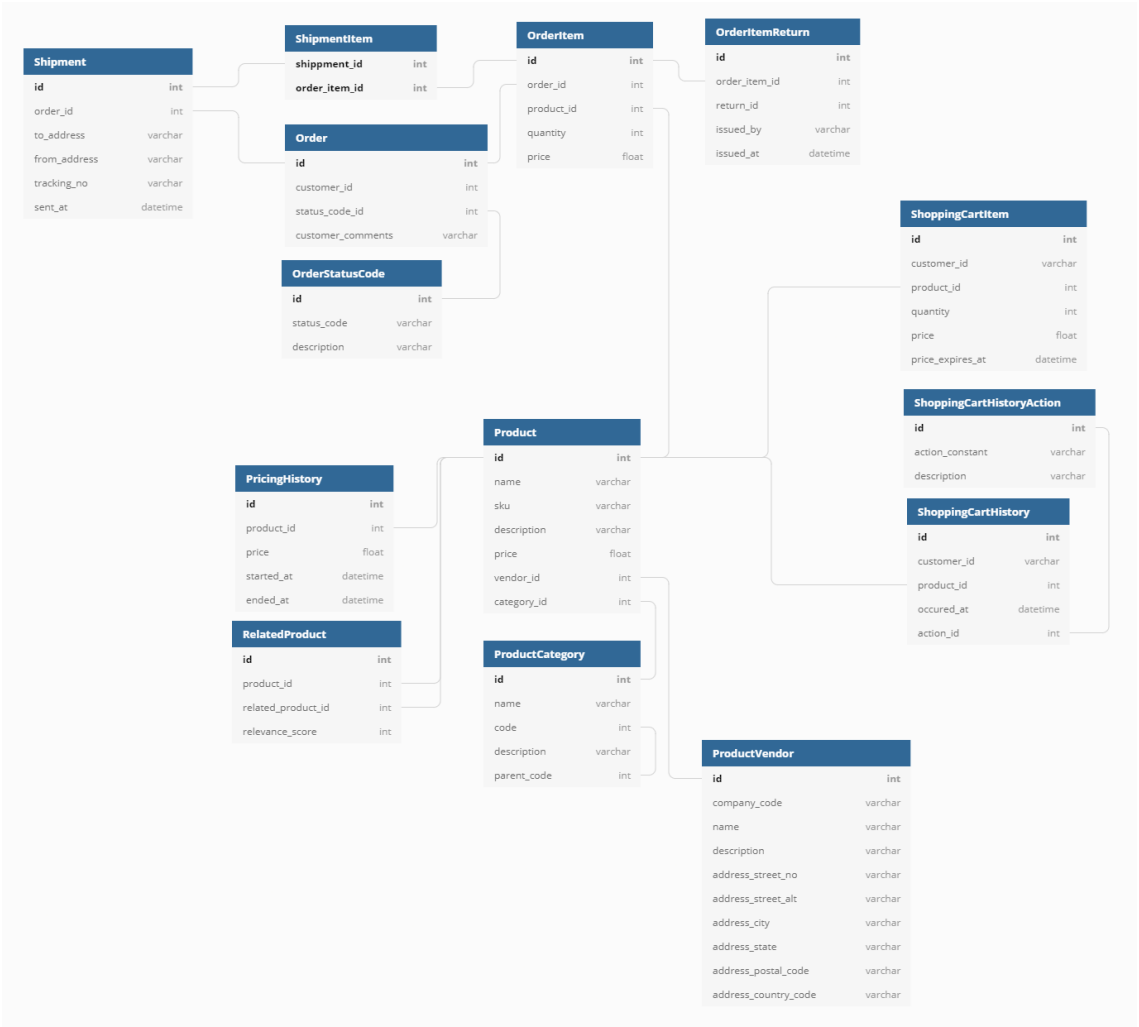
pana tai rinnakkaisena. Lyhyesti kuvailtuna ensimmäinen kerros toteuttaa suoraan eksplisiittisesti valitun tietomallin ja toinen kerros vastaa käyttöliittymän esittämisestä tietomallin sisältöön vaikuttamista varten. Kolmas kerros hoitaa interaktiot näiden kahden muun kerroksen välillä.[2]

Seuraavat kohdat käsittelevät tätä rakennetta tietomallin toteutuksien näkökulmasta. Tärkeää tämän osion kannalta on hahmottaa koko verkkosovelluksen koostuvan kerroksista.

2.1.1 Tietomallit

Aloitetaan ylläkuvatun sovellusarkkitehtuurin kohdasta tietomalli. Verkkosovelluksien tietotekniikan yhteydessä tietomallilla tarkoitetaan käsitteellistä näkemystä siitä, miten todellisuus mallinnetaan tietojärjestelmässä.[5]. Esimerkiksi kuvassa 2.3 on teoreettinen tietomallin esimerkki, jota voitaisiin käyttää lähtökohtana verkkokaupan rakenteen suunnittelussa. Tietomalli ilmenee verkkosovelluksen lähdekoodissa myös kohdissa, jotka eivät toteuta sitä suoraan.

Tässä tutkimuksessa tietomallin toteutuksella tai kuvauksella tarkoitetaan jotakin sovelluksen lähdekoodin osaa, joka kuvaa tietomallin mukaista rakennetta. Jär-



Kuva 2.3: Esimerkki tietomallista[6]

jestelmän arkkitehtuuri muodostuu täten sen komponenttien välisistä suhteista ja yhteentoimivuuden kuvauksista.[7] Esimerkiksi jos verkkosovellusta suunnittelussa käytettäväksi luotu tietomalli sisältää kohteen 'käyttäjä', jolla on lista siihen liittyviä kenttiä kuten 'nimi', niin jokainen osa koodia joka sisältää nämä sisäkkäiset rakenteet joko eksplisiitisesti tai implisiittisesti on tietomallin toteutus. Tietomallin toteutuksia eivät siis ole pelkät tietokantojen rakenteet.

Tietomallin toteutuksia on monenlaisia sovelluksen eri osissa, mutta tietomallia kuvataan ja toteutetaan kaikkein selkeimmin verkkosovelluksen tietokannassa ja sen tietorakenteissa. Kuvan 2.1 mallissa tietokantaa kuvataan levykkeellä ja sen hallinnosta vastaavaa ohjelmisto sylinterin muotoisella palvelimen kuvalla. Tietokantojen käyttö perustuu tiedon tallentamiseen organisoituun ja systemaattiseen muotoon ilman, että sovellus voi tallentaa tietokannan sääntöjen vastaisesti formatoitua tietoa.[8, p. 34] Tämän vuoksi tietomallin toteutus tietokannassa on oleellinen osa tietomallin kuvaaman todellisuuden mallintamista verkkosovelluksen sisällä. Tietomallin osien, suhteiden ja sääntöjen eksplisiittistä kuvausta tietokannassa kutsutaan tietokantamalliksi tai kaavioksi (englanniksi *schema*).

Tietokannan roolista johtuen se itsessään sisältää rajapintana toimivan tietokantakoodin muodossa muita tietomallin toteutuksia, mikä on aiheellista tässä tutkimuksessa käsiteltävän tietomallin vaatimusten muutosten kannalta. Rajapinnoista kerrotaan lisää kohdassa 2.1.2. Käytössä olevan tietokannan toteuttaman tietomallin muuttamista kutsutaan *datamigraatioksi* (englanniksi *schema migration*).

Yleisimmät tietokantaratkaisut toteuttavat relaatioihin perustuvaa SQL-standardia (Structured Query Language).[9] SQL-tietokantojen käytössä ilmenee haasteena se, että tietokannan sisällöllä ei välttämättä ole eksplisiittisä yhteyttä tietomallin pohjalta rakennetussa koodissa esiintyviin tyyppeihin tai malleihin. Lisää implisiittisyydestä on kohdassa 2.1.2. Tämän haasteen ratkaisemiseksi on luotu ORM -kirjastoja (Object Relational Mapping). Nämä kirjastot kääntävät tietokannan haut ja ha-

kutulokset koodissa käytetyiksi tyypeiksi ja olioiksi, sekä myös päinvastoin.[4, p. 258] ORM - työkaluilla on mahdollisuus vähentää tarvittavien tietomallin kuvausten määrää yhdistämällä enemmän ominaisuuksia yhteen tietomallin kuvaukseen, jota voidaan sitten käyttää sovelluksen lähdekoodin eri osissa.

Suoritusaikainen tilanne

Tähän mennessä mainitut tietomallin toteutukset keskittyvät verkkosovelluksen määrityspuoleen. Tällaiset toteutukset ovat staattisia, jolloin niihin vaikuttavat selvästi tässä tutkimuksessa käsiteltävät sovelluksien vaatimuksien muutokset. Tietomallit ovat kuitenkin myös ajonaikainen toiminnallinen osa verkkosovelluksen käytössä. Tietomallin toteutuksia suorituspuolella ovat esimerkiksi sovelluksen toiminnallisuuden kautta luodut tietorakenteet. Nämä rakenteet eivät aiheuta vastaavalla tavalla vaatimusten muutoksien propagoitumista sovelluksen eri kerroksiin, mutta ne otetaan silti huomioon tutkimusosiossa.

2.1.2 Rajapinnat

Verkkorajapinnat ovat oleellinen osa verkkopalveluiden arkkitehtuuria. Yksinkertaistettuna rajapinnat mahdollistavat informaation siirtämisen eri kielillä ja eri alustoille rakennettujen palveluiden välillä.[10] Tämän tutkimuksen kannalta tärkeää on hahmottaa rajapinnat interaktion välineenä erilaisia tietomalleja toteuttavien tietokantojen välillä. Tästä johtuen rajapinnan toteutuksen on myös toteutettava käytössä olevaa tietomallia.

Verkkorajapinnoissa on yleisessä käytössä kourallinen erilaisia protokollia. Kirjoittamisen hetkellä suosituimpia ovat WebSockets -protokolla ja GraphQL -spesifikaatio, sekä arkkitehtuurilliset tyylit REST ja gRPC. [11]

Verkkorajapintojen toteutus voidaan kerrostaa ja jakaa osiin monimutkaisuuden vähentämisen sekä tietoturvallisuuden nimissä. Tällöin edellämainitun verkkosovel-

lusrakenteen frontendin ja backendin välinen kommunikaatio hoidetaan eri rajapintatoteutuksella kuin palvelimen ja tietokannan välinen kommunikaatio.[4, p. 229]

Lisää kerrostuneisuutta tietomallien toteutuksille luo sovelluksen eri rajapintojen sekä kolmansien osapuolten kirjastojen rajapintojen välinen riippuvuus toisistaan.[12] Erityisesti modernit viimeisen reilun kymmenen vuoden sisällä kehitetyt verkkosovellukset voivat riippua ulkoisista palveluista, joiden rajapintojen kautta toteutetaan sovelluksen ja palvelun välinen interaktio.[13] Tästä esimerkkinä ovat tiedonhallintaan tarkoitetut pilvipalvelut, jotka vaativat oman tietokantamallin ja rajapinnan määrityksen. Valmiit tietokantapalveluratkaisut voivat käyttää geneerisiä rajapintoja, jotka toteuttavat tietomallia vain implisiittisesti.

Käyttöliittymä

Viimeinen kerros on kuvaajan 2.2 kohta *näkymä*. Näkymä määrittelee loppukäyttäjän käyttöliittymän, kuten nettisivun asettelun, mutta ei suoraan ”tunne” tietomallia eikä myöskään vaikuta tietomalliin tai tietokantamalliin.[2] Jotta käyttöliittymä voi kuitenkin toimia välineenä tietomalliin vaikuttamiseen, on sen myös osaltaan toteutettava tietomallia.

Kun tarkastellaan tämän tutkimuksen kannalta mainitsemisen arvoista tietomallin muuttumista verkkosovelluksen vaatimusten muutosten myötä, myös hyvin implisiittiset viittaukset tietomalliin, kuten kirjalliset ohjeet käyttöliittymässä, voidaan katsoa alttiiksi muutosten propagoitumiselle. Muutosten propagoitumisesta kerrotaan lisää kohdassa 2.2.3. Tällainen riippuvuus tietomallista ei kuitenkaan ei kuitenkaan ole tutkimuksen kohteena tässä työssä.

Eksplisiittiset ja implisiittiset kuvaukset

Projektin lähtökohtana toimiva alustava tietomalli on hyvä esimerkki tietomallin eksplisiittisestä kuvauksesta. Tällaisessa kuvauksessa tietomallin elementtien väli-

set interaktiot ja riippuvuudet kuvaavat vielä tarkasti sitä todellisuutta, jota kyseinen tietomalli kuvaa. Suunnittelussa laaditut tietorakenteen kaaviot ja kuvaajat ovat esimerkkejä eksplisiittisestä tietomallin toteutuksesta. Suunnitteluvaiheen eksplisiittiset tietomallikuvaukset muutetaan osittain toteutusvaiheessa implisiittisiksi säännöiksi ja oletuksiksi verkkosovelluksen toteutuksen sisällä.[7]

Tietomallin toteutuksia on erilaisia. Eksplisiittesti jäsennetty rakenne ilmenee esimerkiksi muuttujien ja luokkien nimistä yms. Koodiesimerkki 2.4 kuvaa tietomallin eksplisiittistä toteutusta rajapinnassa. Tässä esimerkissä tietomalliin kuuluu taulu User, jolla on kenttä Email. Kummatkin on esitetty eksplisiittisesti koodissa. Tämän sijaan koodiesimerkissä 2.5 on käyttöliittymän elementti, jolla voidaan esittää lista User -taulun sisältöä, mutta elementin oma tietorakenne toteuttaa vain implisiittisesti yllämainitun User ja Email nimet sisältävän tietomallin.

Rajapinnassa voi olla komentoja, joilla on vain implisiittinen yhteys tietorakenteeseen, kuten esimerkiksi tietomallin rakenteita rikkovat komennot. Tällöin rajanpinta toimii käytännössä tietomallin implisiittisenä toteutuksena. Tietomallin määrityksistä saatetaan myös generoida koodia, jolloin tietomallin pohjalta generoidussa koodissa on vain implisiittinen kuva tietorakenteesta. Tällainen on esimerkiksi geneerinen kattava rajapinta, jossa tietomallin jokaisella osalla on nimeä lukuunottamatta kaikki samat toiminnallisuudet. Koodiesimerkissä 2.6 on generoitua koodia taululle User.

2.1.3 Yhteenveto rakenteesta

Tässä osiossa käsitelty verkkosovellusten rakenne koostuu siis kerroksista ja osista, jotka ovat toisistaan keskenään riippuvaisia. Jokaiseen osaan sisältyy jonkinlaisia tietomallin toteutuksia, joten nämä toteutukset muodostavat sovelluksen rakenteen tavoin myös keskenäisiä riippuvuuksia. Lähes kaikki osat käyttävät jossain määrin samaa tietomallia täyttääkseen sovelluksen tarkoituksen, mutta ovat samalla enem-

```
case "POST":
    const { email_address, first_name } = req.body;
    result = await prisma.customer.create({
        data: {
            first_name: first_name,
            email_address: email_address,
        },
    });
    res.json({ customer: result });
    break;
```

Kuva 2.4: Eksplisiittinen tietomallin toteutus rajapinnassa

```
export interface AccordionProps
    extends React.HTMLAttributes<HTMLDivElement> {
    data: {
        title: string;
        summary: string;
        content: any;
        customPreview?: React.ReactNode;
    }[];
}
```

Kuva 2.5: Implisiittinen tietomallin toteutus oliorakenteessa

```
export const CustomerFindManyQuery = queryField('findManyCustomer', {
    type: nonNull(list(nonNull('Customer'))),
    args: {
        where: 'CustomerWhereInput',
        orderBy: list('CustomerOrderByWithRelationInput'),
        cursor: 'CustomerWhereUniqueInput',
        take: 'Int',
        skip: 'Int',
        distinct: list('CustomerScalarFieldEnum'),
    },
    resolve(_parent, args, { prisma, select }) {
        return prisma.Customer.findMany({
            ...args,
            ...select,
        })
    },
})
```

Kuva 2.6: Implisiittinen tietomallin toteutus generoidussa koodissa

män tai vähemmän modulaarisia, mikä vaatii näiden osien sisältävän toiminnallisuuden ja rajapinnat toistensa välisen interaktion mahdollistamiseksi. Toisin sanoen kyseisessä arkkitehtuurissa näiden päällekkäin pinoksi rakennettujen moduulien on jossain määrin tiedettävä, minkälaisia tietomalliin liittyviä ohjeita ne voivat lähettää muille moduuleille.

Tulevassa kohdassa 2.2.3 kerrotaan lisää toisistaan riippuvaisten verkkosovelluksen osien vaikutuksesta sovelluksen kehityksen yhteydessä.

2.2 Kehityksessä hallittavat asiat

Normikäytäntönä on, että aloitetaan uusi tietojärjestelmäprojekti, jossa on lähtökohtana alustava tietomalli, visio tavoitellusta käyttäjäkokemuksesta sekä hahmotelma taustajärjestelmästä, joka ensi vaiheessa lähinnä taltioi tietoa. Näiden pohjalta valitaan esim. tietokanta, joka on palvelimella tai pilvipalvelussa ja määritellään sille verkkorajapinta, jolloin tietokannan määritys ja/tai sen päälle tehty kutsurajapinta sekä verkkorajapinta toteuttavat tietomallia. [4, p. 18]

2.2.1 Tekninen velka

Tekninen velka on ohjelmistokehityksessä käytössä oleva termi, joka viittaa tapaan, jolla aikaisemmin ohjelmistoprojektissa käytetyt oikotiet ilmaantuvat myöhemmin erilaisina kustannuksina. Verkkosovellusta rakentaessa kehittäjät törmäävät mahdollisiin haasteisiin, kuten tiukkoihin määräaikoihin tai asiakkaan vaikeisiin vaatimuksiin, jolloin he joutuvat tekemään suboptimaalisia väliaikaisia tai pysyviä ratkaisuja.[12] Resurssien kuten ajan säästämiseksi tehdyt valinnat mahdollistavat projektin määräajoissa pysymisen lyhyellä aikavälillä, mutta käyttäytyvät ikään kuin velan tavoin pitkällä aikavälillä.

Tämän lisäksi teknisellä velalla on myös toinen määritelmä, joka osittain kil-

pailee yllämainitun määritelmän kanssa, ja myös osittain leikkaa tätä määritelmää. Sovellusta rakentaessa koodilla on käytännössä aina jokin käyttötarkoitus tai mallinnettava asia, josta poikkeaminen tulkitaan tällöin tekniseksi velaksi. Myös se tarve tai mallinnettava asia, johon koodi liittyy, voi muuttua, jolloin sitä simuloiva koodi erkanee ajan myötä kauemmaksi eksplisiittisestä tarkoituksestaan. Koodin vanheneminen on tästä syystä myös tämän määritelmän perusteella teknistä velkaa.

Velka voi esiintyä monella tavoin, mutta ensisijaisesti se tuottaa lisää työmärää projektin myöhemmässä vaiheessa. Teknistä velkaa tuottavat valinnat voidaan tehdä joko tietoisesti tai vahingossa. Tietoisesti tehdyissä teknistä velkaa tuottavissa valinnoissa pyritään arvioimaan, onko velan kerryttämä työmäärä tarpeeksi suurta perustellakseen kalliimman, vähemmän teknistä velkaa tuottavan valinnan.[13]

Teknisellä velalla on todistetusti haitallinen vaikutus koodin ja sovelluksen laatuun. Haitallisia vaikutuksia ovat ainakin ohjelmointivirheet, sovelluksen vääränlainen toiminta ja tarve kirjoittaa aikaisempaa koodia uudestaan.[14][12] Seuraavassa kohdassa käsitellään tarkemmin aiempaa tutkimustietoa koodin määrän, laadun ja teknisen velan suhteesta.

Tässä tutkimuksessa sovelluksissa käytössä olevien arkkitehtuurien vaikutusta pyritään tutkimaan teknisen velan tavoin. Tässä yhteydessä teknisellä velalla tarkoitetaan mahdollisia rakenteellisia haasteita yleisesti käytössä olevassa verkkosovellusteknologiassa, jotka eivät seuraa tietoisesti tarkoituksella valituista oikoteistä, vaan yleisesti suosituista käytännöistä. Edellisissä kappaleissa esiteltyjen rakenteiden käyttämistä voidaan tämän tutkimuksen näkökulmasta ajatella teknistä velkaa kerryttävänä valintana, jonka kerryttämän velan määrä tutkimuksessa mitataan.

2.2.2 Koodin määrä

Tässä yhteydessä koodin määrällä ei viitata ominaisuuksien tai vaatimusten määrään, jotka puolestaan vaativat luonnostaan enemmän pidemmän lähdekoodin. Tä-

män tutkimuksen kontekstissa koodin määrällä viitataan tarpeettoman suuren määrään koodia tilanteissa, jossa vähemmällä, mutta ei välttämättä aina oikeammalla määrällä koodia pystyttäisiin täyttämään samat vaatimukset. Tarpeettoman pitkä lähdekoodi tunnetaan yleisesti haitalliseksi.[15] Haittavaikutuksiin kuulu yllä mainittu tekninen velka.

On useita tekijöitä, jotka voivat kasvattaa koodin määrää tällä tavalla. Todennäköisesti selkein näistä tekijöistä on duplikoitunut koodi, eli koodi joka esiintyy lähdekoodissa useamman kuin yhden kerran. Myös toiminnallisuudeltaan identtisiä koodin osia voidaan tulkita duplikoituneeksi koodiksi. Duplikoitunut koodi lisää sovelluksen virhealttiutta ja on haastavampaa lukea ja kehittää.[15] Yksi tämän tutkimuksen myöhemmän kokeellisen osion ensisijaisista motivaatioista on selvittää, muistuttavatko yleisesti käytössä olevat verkkosovellusten tietomalliratkaisut vaikutuksiltaan tekniseen velkaan tätä koodin duplikoitumisen ilmiötä.

2.2.3 Propagoituminen

Ohjelmistojen kehityksen ja elinkaaren aikana tapahtuu jatkuvasti muutoksia uusien ja muuttuvien sovelluksen toiminnalle asetettujen vaatimusten täyttämiseksi. Nämä muutokset voivat vaikuttaa sovelluksen muihin osiin, mikä vuorostaan voi tuottaa haitallisia vaikutuksia sekä kasvattaa teknistä velkaa. Tätä ilmiötä kutsutaan muutoksien propagoitumiseksi (englanniksi *change propagation* tai *ripple effects*).[16]

Aiemmassa osiossa pohjustettiin verkkosovellusten tyypillistä rakennetta ja sitä, miten verkkosovellusten eri osat toteuttavat sovelluksen tietomallin. Tästä johdettiin yhteys muutoksien propagoitumiseen, mikä puolestaan johti tähän tutkimukseen. Tyypilliset verkkosovellusten arkkitehtuurilliset periaatteet painottavat modulaarisia pinomaisia rakenteita. Tähän malliin sisältyy rakenteellista altiutta muutosten propagoitumiselle koko sovelluksen läpi, kun sovelluksen osat toteuttavat samaa tietomallia.

3 Tutkimusmenetelmä

Tässä kappaleessa käydään lyhyesti läpi ne tutkimusmetodit ja tutkimusteoria, joita tullaan hyödyntämään seuraavissa kappaleissa 4 ja 5.

3.1 Lähestymistapa

Koska tämän tutkimuksen tarkoitus on tutkia ja kerätä aiheena olevasta ongelmasta uutta tietoa, eikä niinkään ratkaista olemassa olevaa käytännön ongelmaa, on yleisesti lähestymistavaksi valittu positivistinen induktiivinen tutkimus. Tämä tarkoittaa, että tulosten tavoitellaan olevan yleistettävissä tämän tutkimuksen kontekstin ja tutkimuskohteiden ulkopuolelle tavalla, joka on toistettavissa soveltamalla samoja tutkimusmetodeja muille kun tähän tutkimukseen valituille tutkimuskohteille.[17]

Tiedon keräykseen käytetään kokeellista metodia. Tähän sisältyy kvantitatiivinen mittaus tekemällä valituille sovelluksille jokin vaatimuksien muutos, ja laskemalla kuinka moneen kohtaan se vaikuttaa lähdekoodissa.

Kerätyn tiedon käsittelyssä käytetään empiirisiä menetelmiä, mutta sen analyysi on kuitenkin induktiivista, koska tarkoitus on kokeen perusteella arvioida ilmiötä kyseisten sovelluksien ulkopuolella. Tutkimusten tulosten analyysiin sisältyy temaattista analyysiä, eli erilaisten muutosten luonteen tutkintaa ja pohdinta siitä, vaikuttavatko jotkin niistä koodin laatuun tai työmäärään enemmän kuin muut. Analyysin metodina käytettään myös osin statistista analyysia, koska tutkimuksessa nimenomaan tutkitaan määrällisesti kuinka monta lähdekoodin eri moduulia kokees-

sa joudutaan muokkaamaan.

3.2 Tutkimusmetodi

Tutkimus jakautuu karkealla tasolla kahteen osaan. Ensimmäinen osa on deduktiivinen laadullinen tutkimus ja toinen osa induktiivinen laadullinen evaluaatio. Ensimmäisessä osassa kartoitetaan nykytilannetta verkkosovelluksissa käytössä olevista teknologioista, jotta toisessa osassa voidaan valita nykytilannetta edustavat verkkosovellukset rakenteellisen teknisen velan tarkastelun kohteiksi. Näitä kumpaakin ennen kuitenkin kerättiin pohjatietoa kokeellisia osia edeltävää teoriakappaletta varten.

3.2.1 Pohjatiedon keräys

Teoriapohjan luomista varten kerätään ennen koetta analysoitavaksi olemessaoleva aiheeseen liittyvää tutkimusaineistoa. Tämä kerätty tieto pohjatieto löytyy edellisestä kappaleesta 2. Tiedon keräykseen käytetty metodi on kirjallisuuskatsaus ja lähteinä toimineet aikaisemmat tutkimukset ja artikkelit on kerätty Google Scholar -palvelusta.

3.2.2 Deduktiivinen laadullinen tutkimus

Tiedonkeruumenetelmänä kätetään arkistoidun tiedon louhintaa.[13] Tiedon lähteenä toimii GitHub, koska tämän oletetaan olevan yleistettävissä oleva ja luonteeltaan kattava lähde sovellusten ja sovelluksissa käytettävien teknologioiden suosion kannalta. GitHub:n tarjoaman hakurajapinnan tulokset kirjataan sellaisenaan ylös siinä muodossa, mitä ne ovat tutkimuksen kirjoitusprosessin aikana. Tutkimuksen tavoitteena on vastata tutkimuskysymykseen TK1.

Tulosten analyysiin käytetty metodi on statistinen sekä temaattinen analyysi.

Projektien uusiksi projekteiksi haarautumisten määrän oletetaan toimivan indikaattorina alkuperäisen projektin teknologisten ratkaisujen suosiolle. Tutkimus tuottaa listan sovelluksia ja näissä käytössä olevia teknologioita, joiden pohjalta syntetisoidaan temaattisen analyysin avulla yleiskuvat verkkosovellusten teknologiapinoista.

3.2.3 Induktiivinen laadullinen evaluaatio

Tutkimuksen kokeen induktiivinen osa suoritetaan simulaationa. Tutkimustrategiana käytetään toimintatutkimusta, jotta tutkittavana oleville verkkosovelluksille voidaan synteettisesti tuottaa keskenään vastaavat simuloidut vaatimuksien muutokset. Kokeen tavoitteena on vastata tutkimuskysymykseen TK2.

Kokeessa on yhteensä 3 vaihetta, jotka ovat tutkittavien kohteiden altistaminen tietomallin muutokselle, muutoksen vaikutuksen alaisten moduulien havainnointi sekä muutoksen vaikutuksen evaluaatio teknisen velan kontekstissa. Koetta varten on valittu neljä (4) full stack -verkkosovellusta, joiden arkkitehtuurissa käytetään deduktiivisen osion perusteella suosituiksi todettuja viitekehyksiä.

Datan laadullista analyysimetodia valittaessa oli kaksi selkeää vaihtoehtoa. Yksi on temaattinen analyysi ja toinen on grounded theory. Grounded theory painottaa datan kategorisointia enemmän teorian näkökulmasta, kun taas temaattinen analyysi painottaa kategorisointia tulosten mukaan. Metodiksi valittiin temaattinen analyysi lähdekoodien moduuleille, koska teoria aiheesta on nykytilanteessa puutteellista ja koska analyysin tavoitteena on johtaa tuloksista karkeat laskut ongelman laajuudesta.

4 Suositut teknologiapinot

Tässä osiossa vastataan johdannossa esiteltyyn tutkimuskysymykseen TK1, eli minkälaisia teknologiapinoja on yleisessä käytössä verkkosovelluksien tietomallitoteutuksissa. Tähän kysymykseen vastataan ennen tutkimuskysymystä TK2, jotta voidaan valita järkevät ja nykytilannetta edustavat projektit analysoitavaksi seuraavan osion TK2:sta käsittelevään soveltavaan tutkimukseen.

Osio alkaa lyhyellä johdannolla, jossa esitellään missä muodossa kysymykseen vastataan. Tämän jälkeen käydään läpi tulosten lajittelussa käytetyt termit ja kategoriat. Osion lopusta löytyvät tutkimuksen toteutuksen läpikäynti, tutkimuksen tulokset sekä tulosten graafinen analyysi.

4.1 Tutkimuksessa huomioitavat kohteet

Ensiksi on määriteltävä, miltä vastaus tutkimuskysymykseen tulee näyttämään. Toisin sanoen on valittava ne datapisteet, jotka kirjataan tarkastelun alla olevista kohteista ylös. Tätä varten ensiksi suunnitellaan taulukko, jonka kentät myöhemmin täytetään tutkimuksen tuloksilla.

Aluksi voidaan koittaa hyödyntää olemassa olevia lähteitä näiden kenttien soveltamiseksi. Suosituista full stack -teknologioista ei löytynyt tuoreita tutkimuksia tämän tutkimuksen taustalla tehdyn tutkimustyön myötä. Tästä syystä kenttien laatiminen tapahtuu jokseenkin sokeasti turvautuen suosittujen kansainvälisten konsultointiyhtiöiden julkisiin arvioihin. Esimerkiksi w3schools.com (ei yhteyttä W3C:ään,

eli *The World Wide Web Consortium*:iin) sivustolla on oma lista suosituimmista full stack -teknologioista otsikolla *Popular Stacks*. Listalle[18] ei ole merkitty lähdettä, mutta se toimii riittävän hyvin taulukon pohjana.

- LAMP stack: JavaScript - Linux - Apache - MySQL - PHP
- LEMP stack: JavaScript - Linux - Nginx - MySQL - PHP
- MEAN stack: JavaScript - MongoDB - Express - AngularJS - Node.js
- Django stack: JavaScript - Python - Django - MySQL
- Ruby on Rails: JavaScript - Ruby - SQLite - Rails[18]

Sovelletaan tästä listasta ensimmäiset kentät taulukkoon. Ne ovat verkkoympäristön ohjelmointikieli, tietokantaohjelmisto ja palvelinympäristön ohjelmointikieli. Käyttöjärjestelmälle ei tarvita kenttää, koska voidaan olettaa että kentän sisältö on sama lähes jokaiselle tutkittavalle kohteelle. Palvelinohjelmalla (esimerkiksi *Apache* tai *Nginx*) on myös turha olla oma kenttä, koska se ei tyypillisesti riipu valitusta repositoriosta, joten kenttä jää vapaaksi lähes kaikille kohteille.

Tässä tutkimuksessa näiden kohteiden lisäksi on tarpeellista valita myös osion 2 teoriaan perustuvia kenttiä, jotka liittyvät tietomallien propagoitumiseen ja tekevät tuloksista olennaisempia liittyen tutkimuskysymykseen TK2. Nämä kentät ovat tietokannan tyyppi sekä verkkorajapintojen protokollat. Lopuksi taulukkoon täytyy lisätä vielä datan käsittelyyn liittyvät kentät, eli nimi, kategoria, haarautumisten määrä riippuvuuksien määrä ja repositorion koko. Taulukossa 2 on listattuna kaikki kentät ja näiden kenttien tarkat selitykset ovat seuraavassa osiossa 4.1.1.

4.1.1 Taulukon kenttien selitykset

Tästä osiosta löytyy tarkemmat selitykset niistä kentistä, joita käytetään tutkimuksen tämän osion tulosten taulukossa.

Taulukko 4.1: Tutkimuksen datapisteet

nimi	haarautumisten määrä	kategoria
käyttöliittymän kieli	tietokantaohjelmisto	p:ympäristön kieli
tietokannan tyyppi	verkkorajapinta	riippuvuudet

Nimi: Kohteen GitHub -repositorion nimi. Kohteet ovat joko verkkosovelluksia tai verkkosovelluksien tekemiseen käytettäviä viitekehyksiä. Nimet ovat muodossa [repositorion tarjoajan nimi]/[repositorion nimi]

Haarautumisten määrä: Kuinka monta kertaa kohdeprojektin pohjalta on GitHub:ssa luotu uusi haara. Tämä on ensisijainen mittari sille, kuinka laajasti kohteen teknologia on käytössä.

Kategoria: Tämän kenttä määrittelee onko tarkasteltavan kohteen tyyppi staattinen, dynaaminen, verkkokauppa, portaali vai sisällönhallinta. Katso osio 4.1.2.

Käyttöliittymän kieli: Verkkoympäristössä ensisijaisesti käytössä oleva ohjelmointikieli (englanniksi *frontend*). Tämä tarkoittaa kieltä, jolla suurin osa käyttäjälle esittettävistä osista on kirjoitettu. Esimerkiksi JavaScript.

Tietokantaohjelmisto: Ohjelmisto, joka vastaa tietokannan jäsentelystä ja hakemisesta. Kohteissa, joiden kanssa voi käyttää useampaa tietokantaohjelmistoa, on kenttään merkitty suositeltu ohjelmisto. Mikäli tietokantaohjelmistolle ei ole suositusta, niin kenttään on laitettu ensimmäisen vaihtoehto listatuista ohjelmistoista, sekä asteriski (*).

Palvelinympäristön kieli: Palvelinympäristössä ensisijaisesti käytössä oleva ohjelmointikieli (englanniksi *backend*). Tämä tarkoittaa kieltä, jolla määritellään tietokannan logiikka ja tietokannan rajapintojen käyttö. Esimerkiksi PHP.

Tietokannan tyyppi: Tietokantamallin arkkitehtuuri. Kohteissa, joiden kanssa voi käyttää useampaa tietokantatyyppiä, on kenttään merkitty suositeltu tyyppi. Mikäli tietokannan tyypille ei ole suositusta, niin kenttään on laitettu ensimmäisen

vaihtoehto listatuista tyypeistä, sekä asteriski (*). Esimerkiksi SQL.

Verkkorajapinta: Rajapinnassa käytössä oleva arkkitehtuuri ja protokolla. Mikäli kentässä on listattuna pelkkä kohteen rajapinta-arkkitehtuurin tyyli (esimerkiksi tyypillinen REST), niin näissä tapauksissa käytettävää protokollaa (esimeriksi *GraphQL*) ei ole tarkemmin määritelty.

Riippuvuudet: Kuinka monta muuta projektia listaa riippuvuuksissaan kyseisen kohteen. Tämä on toissijainen mittari sille, kuinka laajasti kohteen teknologia on käytössä. GitHub -sivuilla tämä arvo löytyy linkin github.com/[repositorion koko nimi]/network/dependents?dependent_type=REPOSITORY takaa.

Riippuvuuksien määrä ei ole luotettava ensisijaisena suosion indikaattorina, sillä jotkin projekteista eivät sisällä rajapintoja, jotka mahdollistavat niiden käytön riippuvuuksina tai riippuvuuksien laskemista. tällaisten projektien kohdalle on tuloksissa merkitty tähän kenttään *n/a*.

Koko: Kuinka ison tilan projetki vie muistista kilotavuina (kB). GitHubin rajapinnasta tämä arvo löytyy osoitteen api.github.com/repos/[repositorion koko nimi] takaa. Tämä arvo ei kuitenkaan kerro tarkasteltavan projektin oikeaa kokoa, koska siihen ei lasketa muihin repositorioihin hajautettujen komponenttien tai riippuvuuksien kokoja.

Tässä tutkimuksessa repositorion vaatiman levytilan on tarkoitus toimia pinnallisena mutta tarpeeksi luotettavana indikaattorina sille, onko tarkasteltava kohde niin kutsuttu valmis sovellus, vai sovelluksien kehittämisen lähtökohtana käytettävä viitekehys tai pohja. Repositorion koon ja tietomallin kompleksisuuden välistä suhdetta ei myöskään tutkita.

4.1.2 Kategoriden määrittely

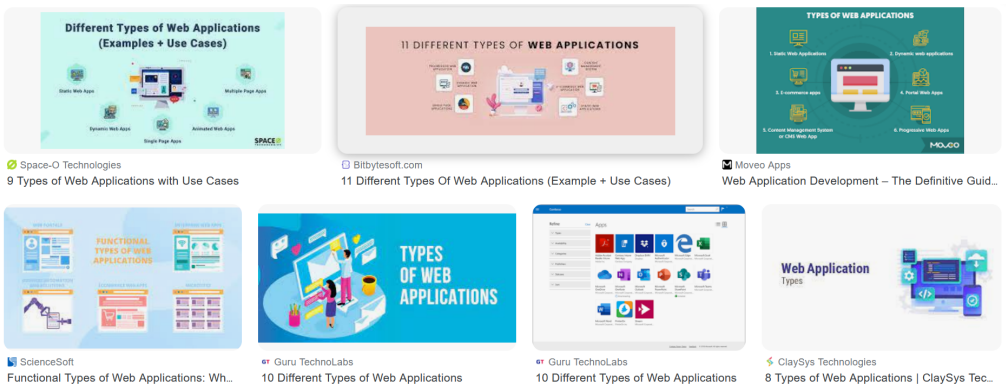
Voidaan olettaa, että verkkosovelluksen rakenne riippuu sen käyttötarkoituksesta. Tämän vuoksi on tärkeää kartoittaa erilaisia toimialueita ja luoda näiden perusteella

temaattiset kategoriat, joista jokaisesta valitaan tarkasteltavaksi sovellus. Esimerkki tällaisen kategoirsoinnin pohjasta löytyy aikaisemmasta laajasti siteeratusta verkkosovelluksia kategorioineesta tutkimuksesta *The content and design of web sites: an empirical study* käytetty jako viiteen.[19]

- Computers
- Information
- Finance & Insurance
- Services
- Products

Tämä jako ei kuitenkaan vielä riitä, sillä lähteenä käytetty tutkimus on 24 vuotta vanha ja eikä ota huomioon erilaisten verkkosovellusten teknisiä vaatimuksia. Toisin sanoen nämä kategoriat eivät ota kantaa siihen, miltä käytössä oleva tietomalli ja sen toteutus voisivat näyttää. Kategorioita täytyy tämän tutkimuksen osalta muokata niin, että ne heijastavat paremmin erilaisia full stack -teknologian vaatimuksia eri kategorien välillä.

On olemassa laajasti käytetty jako verkkosovelluksille, joka perustuu eri teollisuudenalojen erilaisiin vaatimuksiin. Tämä on ensimmäinen jako, joka tulee vastaa hakukoneiden tuloksissa, kun haetaan hakutermeillä kuten esimerkiksi "types of web applications". Kuvassa 4.1 näkyy tällaisen kuvahaun tulos. Näillä katergorisoinneilla on keskenään pinnallisia eroja, mutta kuitenkin lähes kaikki sisältävät ainakin kategoriat *static*, *dynamic*, *single page*, *content managament system* ja *portal*. Muita usein sisällettyjä kategorioita ovat *multiple page* ja *e-commerce*. Tämän tutkimustyön taustalla tehdyssä tutkimustyössä tälle jaolle ei löytynyt yhteistä lähdettä, mutta sitä voidaan kuitenkin käyttää inspiraationa tässä tutkimuksessa. Määrittelemällä näille katergorioille tarkat määritykset, niitä voidaan käyttää sivujen valinnassa



Kuva 4.1: ”different types of online applications” -Google kuvahakutulokset

4.1.3 Kategorioiden määritelmät

Staattinen: Tällainen verkkosovellus ei mahdollista tietomallin sisältöön vaikuttamista käyttöliittymän kautta. Käyttäjä ei pysty tallentamaan tällaiseen dataa, kuten esimerkiksi käyttäjätiliä. Tämän vuoksi esitettävä sisältö ei riipu käyttäjästä, vaan on sama kaikille.

Dynaaminen: Dynaamisia verkkosovelluksia ovat kaikki sovellukset, jotka staattisista sovelluksista poiketen mahdollistavat sovellukseen tallennettuun tietoon vaikuttamisen käyttöliittymän kautta. Tämä määritelmä sisältää valtaosan kaikista verkkosovelluksista, minkä vuoksi tässä tutkimuksessa tämä kategoria on seuraavien kategorioiden kattokategoria, johon laitetaan vain ne tutkinnan kohteet, jotka eivät sovi verkkokauppojen, portaalien, tai sisällönhallintajärjestelmien kategorioihin.

Verkkokauppa: Tähän kategoriaan kuuluvat kaikki ne verkkosovellukset, joiden toiminnallisuuteen kuuluu tässä tutkimuksessa sekä maksujen käsittely maksurajapinnan kautta, että mahdollisuus ostaa useampi kohde kerralla. Tästä syystä esimkerkiksi mahdollisuus lahjoittaa rahaa tai aktivoida maksullinen tilaus eivät yksinään riitä tekemään sovelluksesta verkkokauppaa.

Portaali: Portaalit keräävät tietoa yhteen useasta lähteestä. Tässä tutkimuksessa tähän kategoriaan kuuluvat ne verkkosovellukset, joissa yksittäinen haku tieto-

kannasta tapahtuu useamman hajautetun ja eri tietomallia käyttävän tietokannan välillä. Esimerkiksi hakukäyttöliittymä, joka on yhdistetty useamman hakukoneen rajapintaan.

Sisällönhallinta: Englanniksi *content management system* (CMS). Sisällönhallintajärjestelmän toiminnallisuuteen sisältyy uuden sisällön luominen käyttäjän toimesta, tämän sisällön muokkaaminen ja sen julkaiseminen. Kategorian kannalta huomioon otettavaksi sisällöksi lasketaan tässä tutkimuksessa ainoastaan verkkosivut, asiakirjat, audiovisuaalinen sisältö ja rekisterit.[20] Esimerkiksi verkkosivujen hallintajärjestelmä WordPress.

Mikäli jokin verkkosovellus täyttää useamman kategorian vaatimukset, se tulkitaan tässä tutkimuksessa priorisointisäännön mukaan merkittävimpään kategoriaan. Suurin painoarvo ensin, tämä priorisointisääntö on verkkokauppa > sisällönhallinta > portaali > staattinen > dynaaminen.

4.2 Käytössä olevat teknologiat

Siis itse asiaan. Tästä aiheesta oli haastavaa löytää kerättyä tietoa muualta, joten tässä kohdassa tiedonkeruumenetelmänä on käytetty avoimen lähdekoodin kirjastojen haarautumisten määriä. Tietomalli- ja tietokantaratkaisuissa voidaan git -repositorioiden haarukoitumista analysoimalla havaita kourallinen yleisiä trendejä, joista löytyy kuvaajat seuraavassa kohdassa 4.3.

Hakutulokset, joiden pohjalta tulostaulukko on täytetty perustuvat seuraaviin GitHub:n hakurajapintaan syötettyihin hakuasetuksiin.

- *s=forks*, eli lajitellaan haarautaumisten määrän mukaan.
- *o=desc*, eli suurin määrä haarautumisia tulosten alkuun.
- *type=Repositories*, eli rajataan tulos vain repositorioihin.

Taulukkoon valittiin kaikki hakutuloksista löytyneet kohteet, jotka täyttivät osiossa 2 käsitellyn full stack -rakenteen määritelmän.

4.3 Tulosten lyhyt analyysi

Lopullinen tarkasteltujen kohteiden määrä on 40 kappaletta. Tulokset riittävät varmistamaan, että tutkimuksen seuraavassa osiossa analysoitavaksi valitut kohteet heijastavat tosimaailmaa ja käytössä olevia ratkaisuja. Tässä osiossa on kuvaajia, jotka auttavat havainnollistamaan tämän tutkimuksen taulukkojen tuloksia. Kaikki kuvaajat sisältävät vain absoluuttisia lukuja, joita ei ole kerrottu esimerkiksi kyseisiä ratkaisuja sisältäneiden repositorioiden haarautumisten määrällä.

4.3.1 Numeerinen anaalyysi

Ensinnäkin kategorioiden valinta onnistui siinä mielessä, eikä mikään kategoria jäänyt täysin ilman kohteita. Valittujen kategoridoiden kontekstissa *dynaaminen* vastaa kaikkein eniten hypoteettista muu kategoriaa, mikä heijastuu myös tuloksista. Alla olevassa kuvaajassa 4.2 näkyy, että suurin osa kohteista ovat dynaamisia verkkosovelluksia, joita oli yhteensä 17 kappaletta. Verkkokauppojen vähäisestä määrästä avoimen lähdekoodin verkkosovellusten joukossa ei ole tässä tutkimuksessa luotettavaa teoriaa, mutta arvauksena on, että kilpailun ja tietoturvan nimissä yksityiset ratkaisut ovat suositumpia.

Valtaosa viitekehyksistä tutkittujen kohteiden joukossa olivat nimenomaan JavaScript -kehyksiä, eikä tästä syystä tunnu ollenkaan todellisuuden kanssa ristiriitaiselta, että ylivoimaisesti suosituin ohjelmointikieli käyttöliittymässä on juuri JavaScript. Kielten jakauma näkyy kuvaajasta 4.3.

Erilaisten SQL -ratkaisujen suuri määrä vastaa odotuksia. Alla olevasta taulukosta 4.4 näkyy, että myös noSQL -tyyppinen Mongo DB on suosittu valinta.

Taulukko 4.2: Tulokset 1/3

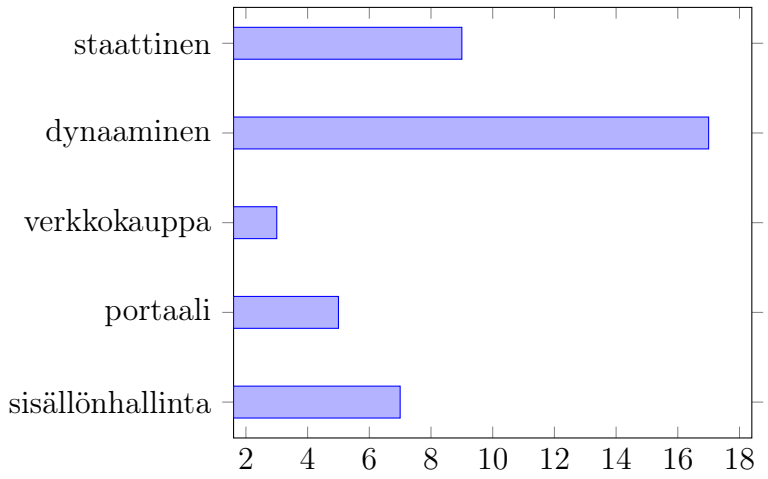
nro.	nimi	haarautumiset	kategoria
#0	spring-projects/spring-boot	38.7k	dynaaminen
#1	barryclark/jekyll-now	35.1k	staattinen
#2	django/django	29.2k	dynaaminen
#3	angular/angular.js	28.1k	dynaaminen
#4	macrozheng/mall	26.8k	verkkokauppa
#5	facebook/create-react-app	26.4k	staattinen
#6	Yidadaa/ChatGPT-Next-Web	25.2k	portaali
#7	vercel/next.js	23.9k	staattinen
#8	laravel/laravel	23.7k	dynaaminen
#9	angular/angular	23.6k	dynaaminen
#10	rails/rails	21.1k	dynaaminen
#11	spring-projects/spring-petclinic	19.8k	dynaaminen
#12	odoo/odoo	18.7k	portaali
#13	pallets/flask	15.7k	dynaaminen
#14	daattali/beautiful-jekyll	14.5k	staattinen
#15	ionic-team/ionic-framework	13.6k	dynaaminen
#16	jeecgboot/jeecg-boot	13.6k	dynaaminen
#17	kubernetes/website	12.7k	staattinen
#18	bailicangdu/vue2-elm	12.5k	portaali
#19	WordPress/WordPress	12.3k	sisällönhallinta
#20	woocommerce/woocommerce	10.8k	verkkokauppa
#21	gatsbyjs/gatsby	10.5k	staattinen
#22	expressjs/express	10.5k	staattinen
#23	lenve/vhr	10.3k	sisällönhallinta
#24	jekyll/jekyll	10.1k	sisällönhallinta
#25	dotnet-architecture/eShopOnContainers	10k	verkkokauppa
#26	TryGhost/Ghost	9.4k	sisällönhallinta
#27	magento/magento2	9.2k	verkkokauppa
#28	desktop/desktop	9.1k	dynaaminen
#29	symfony/symfony	9.1k	staattinen
#30	dotnet/aspnetcore	9.1k	dynaaminen
#31	halo-dev/halo	8.6k	sisällönhallinta
#32	RocketChat/Rocket.Chat	8.6k	dynaaminen
#33	sahat/hackathon-starter	8.1k	dynaaminen
#34	discourse/discourse	8k	sisällönhallinta
#35	vnpy/vnpy	7.9k	portaali
#36	bcit-ci/CodeIgniter	7.7k	dynaaminen
#37	shuzheng/zheng	7.5k	portaali
#38	DrKLO/Telegram	7.4k	dynaaminen
#39	dockersamples/example-voting-app	7.4k	dynaaminen
#40	gohugoio/hugo	7.2k	staattinen

Taulukko 4.3: Tulokset 2/3

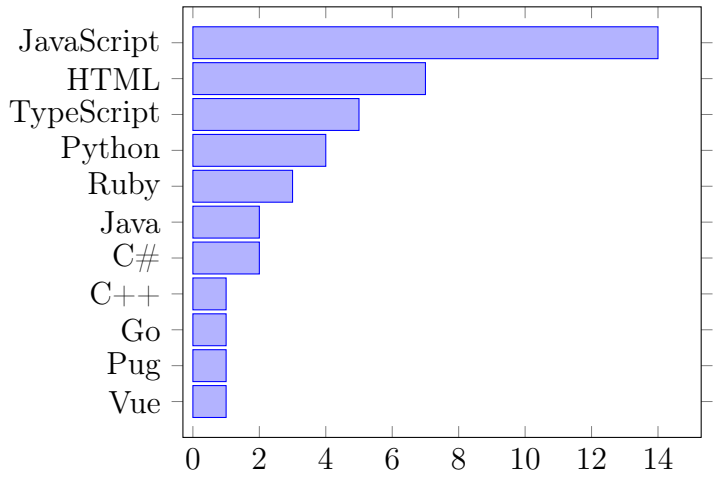
nro.	käyttöliittymän kieli	tietokantaohjelmisto	p:ympäristön kieli
#0	HTML	Mongo DB	Java
#1	HTML	GitHub	xml
#2	Python	SQLite	Python
#3	JavaScript	Mongo DB*	JavaScript
#4	JavaScript	Mongo DB	Java
#5	JavaScript	text	JavaScript
#6	TypeScript	vercel	TypeScript
#7	TypeScript	vercel	JavaScript
#8	HTML	MariaDB	PHP
#9	TypeScript	Mongo DB*	TypeScript
#10	Ruby	SQLite3	Ruby
#11	Java	MySQL	Java
#12	JavaScript	PostgreSQL	Python
#13	Python	SQLite3	Python
#14	HTML	GitHub	JavaScript
#15	JavaScript	SQLite	TypeScript
#16	TypeScript	MySQL	Java
#17	JavaScript	Google Cloud	Python
#18	Vue	MySQL*	JavaScript
#19	JavaScript	MySQL/MariaDB	PHP
#20	JavaScript	MySQL/MariaDB	PHP
#21	JavaScript	PostgreSQL	TypeScript
#22	JavaScript	Cassandra	JavaScript
#23	Java	MySQL	Java
#24	HTML	GitHub	Ruby
#25	C#	Azure SQL	TypeScript
#26	JavaScript	MySQL	JavaScript
#27	JavaScript	MySQL/MariaDB	PHP
#28	Ruby	MySQL	TypeScript
#29	HTML	MySQL/MongoDB	PHP
#30	C#	Azure SQL	C#
#31	TypeScript	PostgreSQL	Java
#32	JavaScript	Mongo DB	TypeScript
#33	Pug	Mongo DB	JavaScript
#34	Ruby	PostgreSQL	JavaScript
#35	Python	SQLite*	Python
#36	HTML	MySQL	PHP
#37	JavaScript	MySQL	Java
#38	C++	SQLite	Java
#39	Python	PostgreSQL	C#
#40	Go	Azure SQL	Go

Taulukko 4.4: Tulokset 3/3

nro.	tietokannan tyyppi	verkkorajapinta	riippuvuudet	koko (kB)
#0	NoSQL	REST	n/a	167387
#1	YAML + JSON	REST	n/a	8406
#2	ORM	REST	1,280,567	238394
#3	NoSQL*	REST	135,970	101744
#4	NoSQL + ORM	REST	n/a	57980
#5	JSON	GraphQL	29,114	23405
#6	JSON	REST	n/a	4536
#7	JSON	REST	40,706	1839341
#8	SQL + ORM	REST	n/a	10635
#9	NoSQL*	REST	2,446,547	460653
#10	SQL	REST	2,028,942	260768
#11	SQL	REST	n/a	7926
#12	SQL	REST	n/a	6133624
#13	SQL	REST	1,458,225	10210
#14	n/a	REST	203	4872
#15	SQL	REST	n/a	1090067
#16	SQL + ORM	Swagger	n/a	52017
#17	Object storage	Swagger	n/a	395496
#18	SQL	REST	n/a	34429
#19	SQL	REST	n/a	399713
#20	SQL	REST	84	437758
#21	SQL	GraphQL	484,024	1116000
#22	ORM	REST	20,655,303	9096
#23	SQL	WebSocket	n/a	8495
#24	YAML + JSON	REST	n/a	55898
#25	SQL	REST	12	659922
#26	SQL	REST	1,498	236368
#27	SQL	REST	n/a	703497
#28	SQL	REST	3	90118
#29	ORM	REST	n/a	253794
#30	SQL	REST	16	413282
#31	SQL	REST	n/a	65609
#32	NoSQL	WebRTC	n/a	748967
#33	NoSQL	REST	n/a	13555
#34	SQL	REST	n/a	584081
#35	SQL*	Websocket	n/a	253900
#36	SQL	REST	2,290	54786
#37	SQL + ORM	REST	n/a	41267
#38	SQL	gRPC	n/a	406409
#39	SQL	REST	n/a	1191
#40	SQL	REST	n/a	115520



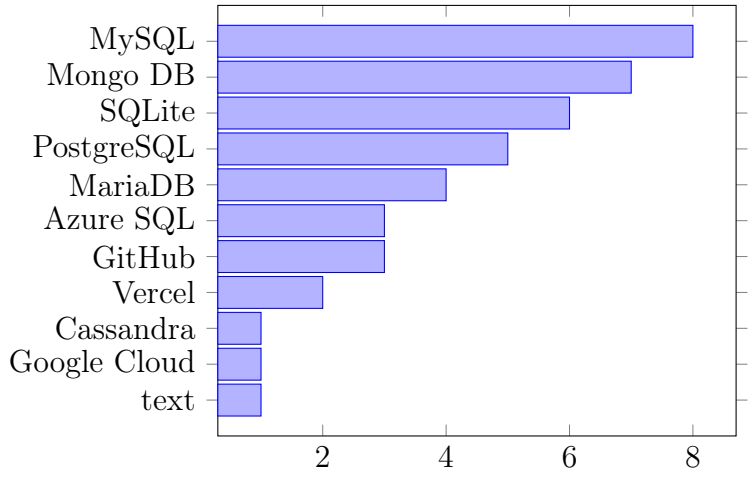
Kuva 4.2: Repositorioiden kategoriat



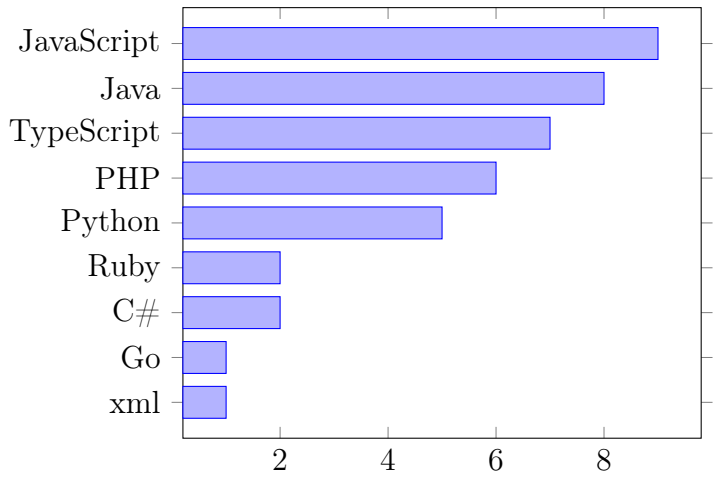
Kuva 4.3: Käyttöliittymäympäristön kielet

Palvelinympäristössä ohjelmointikielet ovat jakaantuneet tasaisemmin verrattuna käyttöliittymäpuoleen. Taulukosta 4.5 näkyy, että kielten jakauma ei ole yhtälailla painottunut vain yhteen kieleen verrattuna JavaScriptin asemaan käyttöliittymäpuolella.

Rajapintaratkaisuissa jakauma oli erittäin painottunut erilaisten yksinkertaisten REST -rajapintojen suuntaan. Ennakkoluuloihin verrattuna rajapintakehysten kuten GraphQL vähäinen ilmentyminen on hieman yllättävää. Yksi mahdollinen selitys tälle voi olla monen taulukon kohteen luonne aloituspisteeksi tarkoitetuun



Kuva 4.4: Tietokantaratkaisut



Kuva 4.5: Palvelinympäristön kielet

viitekehyksenä. Näissä tilanteissa rajapinnan hyvin yksinkertainen toteutus saattaa olla tarkoituksellista sillä perustelulla, että viitekehystä soveltaville kehittäjille jää vapaus itse valita haluamansa rajapintakehys.

4.3.2 Syntetisoidut teknologiapinot

Tulosten perusteella voidaan myös syntetisoida esimerkit teknologiapinoista jokaista kategoriaa varten, joilla vastataan ensimmäiseen tutkimuskysymykseen TK1. Nämä on ensisijaisesti tarkoitettu esimerkeiksi, mutta ne otettiin myös huomioon seuraa-

vassa kappaleessa 5, kun oli aika etsiä tutkittavia projekteja analyysin kohteiksi.

- **Staattinen:** JavaScript - next.js - TypeScript
- **Dynaaminen:** JavaScript - Mongo DB - angular.js - JavaScript
- **Verkkokauppa:** JavaScript - MySQL - SpringBoot - Java
- **Portaali:** JavaScript - MySQL - Vue - PHP
- **Sisällönhallinta:** JavaScript - MySQL - WordPress - PHP

5 Soveltava tutkimus tietomallin muutosten propagoitumisesta

Tässä osiossa vastataan johdannossa esiteltyyn tutkimuskysymykseen TK2, eli missä kaikkialla tietomallin vaatimuksien muuttuminen ilmenee tyypillisissä verkkoprojekteissa. Tähän tutkimuksen osaan valitut projektit perustuvat ensimmäisen tutkimuskysymyksen TK1 tuloksiin edellisessä osiossa.

Osio alkaa lyhyellä kuvauksella tavoitteesta, jossa esitellään hieman missä muodossa tähän kysymykseen vastataan. Alusta löytyy myös kuvaukset niistä neljästä projektista, jotka tähän tutkimukseen valittiin. Tämän jälkeen käydään läpi kohteille suoritettavan analyysin vaiheet. Osion lopusta löytyvät tämän tutkimuksen tulokset sekä näiden pohjalta johdetut johtopäätökset.

5.1 Tavoite

Vastauksen muoto tässä tutkimuksen osiossa on lukumäärä siitä, kuinka monta kertaa tutkittava kohde sisältää kattavan kuvauksen omasta tietomallista, sekä tähän liittyvä laadullinen analyysi siitä, kuinka helppoa näille kohteille on suorittaa tietomallin vaatimusten muutos. Tuloksena on kaksi taulukkoa, joista ensimmäinen on vain lista tietomallin muutoksen vaikutuksen alaisista moduuleista kokeen aikana. Toinen taulukko sisältää muutoksen jalanjäljen lähdekoodissa koodirivien muodossa sekä arviot rakenteellisten tekijöiden vaikutuksesta tekniseen velkaan kertoimen

muodossa. Kertoimen tekijät esitellään kohdassa 5.9.2.

5.2 Tutkittavat kohteet

Tässä tutkimuksen osiossa tarkempaa analyysia varten on valittu tarkasteltaviksi kohteiksi neljä avoimen lähdekoodin verkkosovellusta.

5.2.1 Valintakriteerit

Valinta perustuu edellisessä osiossa 4 kerättyihin tuloksiin, joiden perusteella on tehty arvio laajasti verkkosovelluksissa käytössä olevista teknologioista. Tarkemmat valintakriteerit vaihtelevat hieman valittujen kohteiden välillä, joten niistä kerrotaan tarkemmin kohdissa 5.2.2, 5.2.3, 5.2.4 ja 5.2.5.

Nämä projektit vastaavat edellisessä osiossa 4 esiteltyjä ja käytettyjä kategorioita *dynaaminen*, *verkkokauppa*, *portaali* ja *sisällönhallinta*. Kategoriaa *staattinen* ei oteta mukaan tutkimuksen tähän osaan kyseisen kategorian sovellusten tietomallin interaktiivisuuden puutteen johdosta.

5.2.2 Kohde 1: Mastodon

Kohteen sijainti: https://github.com/mastodon/mastodon

Mastodon on peer2peer -verkkosovellus, joka toimii sosiaalisena media-alustana. Edellisen osion mukaisessa kategorisoinnissa tämä sovellus kuuluu ryhmään *dynaaminen*. Lyhyen selaamisen jälkeen tämä projekti osoittautui yhdeksi eniten GitHub -haarautumisia omaavaksi ”valmiiksi” projektiksi. Haarautumisten määrän lisäksi tämä projekti valittiin myös koska se edustaa tutkimuksen edellisessä osiossa suosituksi osoittautunutta Ruby on Rails -viitekehystä. Sovelluksen tietokantana toimii PostgreSQL 15.

5.2.3 Kohde 2: PrestaShop

Kohteen sijainti: https://github.com/PrestaShop

Valmiita verkkokauppoja löytyy avoimella lähdekoodilla suhteessa vähän. Mahdollisia syitä lienee tietoturvallisuus tai kilpailu. PrestaShop verkkokauppakehys on yksi harvoista lähes täysin käyttöönottovalmiista avoimen lähdekoodin verkkokauppasovelluksista, joka on myös samaan aikaan GitHub:n haarautumisten perusteella suosittu. Kirjoittamisen hetkellä haarautumisia on 4,6 tuhatta. Edellisen osion mukaisessa kategorisoinnissa tämä sovellus kuuluu arvattavasti ryhmään *verkkokauppa*. Sovellus käyttää tietokantanaan MySQL:lää ja tätä simulaatiota varten valittu versio on 8.0.

5.2.4 Kohde 3: Laravel.io Community Portal

Kohteen sijainti: https://github.com/laravelio/laravel.io

Haarautumisten määrän käyttäminen GitHub -haussa antoi tuloksia, jotka olivat viitekehyksiä ja alustoja oman portaaliverkkosovelluksen rakentamiseen, eikä niinkään valmiita projekteja analysoitavaksi. Tämä kohde on valittu analysoitavaksi sillä perusteella, että se on portaaliverkkosovellus, jolla on eniten GitHub -tähtimerkintöjä. Edellisen osion mukaisessa kategorisoinnissa tämä sovellus kuuluu arvattavasti ryhmään *portaali*. Sovellus käyttää tietokantanaan MySQL:lää ja tätä tutkimusta varten valittu versio on 8.0.

5.2.5 Kohde 4: WordPress

Kohteen sijainti: https://github.com/WordPress/WordPress

Sisällönhallintasovellus. Tämä projekti valittiin analysoitavaksi kohteeksi, koska se on kaikkein suosituin sisällönhallinta-alusta tutkimuksen edellisen osion tulosten perusteella.

WordPress on rakennettu ajettavaksi PHP:tä suorittavalla verkkopalvelimella ja käyttää tietokantanaan joko MySQL tai MariaDB:tä. Sen palvelinohjelmisto on kirjoitettu kokonaan PHP:llä ja käyttöliittymä on yhdistelmä PHP:tä ja JavaScriptia. Edellisen osion mukaisessa kategorisoinnissa tämä sovellus kuuluu ryhmään *sisällönhallinta*. Tietokannan kanssa kommunikointiin käytetään REST -rajapintaa ja Valittu versio on tutkumuksen hetkellä uusin 6.2.2 ja tietokannaksi on valittu MySQL 8.0.

5.3 Tutkimuksen vaiheet

Tässä kokeessa on tarkoitus simuloida muutosta tutkittavien kohteiden tietomalleissa.

5.3.1 Tietomallin muutos

Tutkimuksen prosessi alkaa projektin tietokantamallin (englanniksi *schema*) tunnistamisesta. Kun projekti alustetaan käyttöönottoa varten, tietokannan taulut ja relaatiot määräytyvät tietokantamallin pohjalta, mikä tekee siitä sopivan lähtökohdan tietomallin muutoksen seurausten kartoittamiselle. Tietokantamalli tarjoaa kattavan tietomallitoteutuksen, minkä vuoksi tässä tutkimuksessa sitä käytetään lähtökohtana ja pohjana muiden kattavien tietomallin toteutuksien etsintään lähdekoodista.

Tietomallin eri osat voivat ilmentyä eri tavoin sovelluksen muissa osissa, kuten esimerkiksi käyttämättä jääneet taulut. Tämän vuoksi on tärkeä valita simuloitava tietomallin muutos oikein. Tässä tutkimuksessa tietomallin vaatimusten muutosta simuloidaan tekemällä tietomalliin muutoksia, jotka näkyvät vähintään kaikissa niissä osissa sovellusta, kuin tietomallin olemassa olevat taulut. Toisin sanoen tietomallin valinnassa tavoitteena on, että tietomallin simulaatiossa lisätyllä osalla on oltava kaikki samat ominaisuudet ja toiminnallisuudet kuin jo olemassa olevilla tietomallin

osilla.

Lisäksi tulosten määrän lisäämiseksi simuloitavia tietomallin muutoksia tehdään kaksi jokaista projektia kohden. Tämä toteutetaan luomalla sekä uusi taulua ja muokkaamalla olemassa olevaa taulua. Muokattava olemassa oleva taulu valitaan taulun kenttien lukumäärän perusteella. Perusteluna tälle on, että laajin taulu kattaa todennäköisimmin suurimman osan sovelluksen ominaisuuksista.

Esimerkki

Muutoksen prosessi jakautuu kolmeen osaan. Ensiksi etsitään tietomallin isoin taulu ja lisätään siihen yksi uusi kenttä. Sen jälkeen luodaan kokonaan uusi taulu. Lopuksi uuteen tauluun luodaan kenttä, jonka avaimena aiemmin luotu kenttä toimii. Taulussa 5.1 on esimerkki erittäin yksinkertaisesta tietomallista ja taulussa 5.2 on esimerkki siitä, miltä tämä yksinkertainen tietomalli näyttäisi tutkimuksessa suoritettavien muutosten jälkeen. Tutkimustaululle lisätään myös ylimääräinen kenttä *arvo*, sillä monet tietojärjestelmät eivät salli tauluja, joilla on pelkkä identifikaatiokenttä.

Taulukko 5.1: Esimerkkitietomalli ennen muokkauksia

Käyttäjä	**Artikkeli**
käyttäjä_id	artikkeli_id
email	sisältö
salasana	Käyttäjä:käyttäjä_id

Tutkimusmetodin oletuksena on, että sovelluksen kääntäminen käyttövalmiiksi näiden muutosten jälkeen vaatii lähdekoodille suoritettavaksi sellaiset toimenpiteet, jotka paljastavat missä kaikkialla lähdekoodista löytyy tietomallin kuvauksia.

Taulukko 5.2: Esimerkkitietomalli muokkausten jälkeen

Käyttäjä	**Artikkeli**	***Tutkimus**
käyttäjä_id	artikkeli_id	*tutkimus_id
email	sisältö	*arvo
salasana	Käyttäjä:käyttäjä_id	
*Tutkimus:tutkimus_id		

5.3.2 Moduulien analyysi

Verkkosovellusten jako moduuleihin tätä tutkimusta varten perustuu osiossa 2 käsiteltyyn teoriaan liittyen sovellusten rakenteeseen ja arkkitehtuurillisiin käsitteisiin. Analyysin ensisijaisena tavoitteena on reduntattien tietomallin kuvausten määrän ja laadun havainnointi, joten tämän heijastuu myös tuloksissa. Muita tietomallin vaatimusten muutokseen vaikuttavia tekijöitä, kuten tietokantatyökalut tai olemassa olevan tiedon liittäminen uuteen tietomalliin, ei oteta huomioon, vaikkakin ne saatetaan mainita tutkimuksen kulun kuvauksissa.

Jokaiselle tarkasteltavalle kohteelle luodaan moduulien analyysin pohjalta kaksi taulukkoa. Ensimmäisessä taulukossa on listattuna tietomallin olemassa olevien objektien muutosten kannalta relevantit moduulit, lyhet kuvaus näistä moduuleista sekä niiden suurpiirteinen sijainti lähdekoodissa. Toisesta taulukosta löytyy samat kentät, mutta tällä kertaa tietomalliin uuden objektin lisäämisen kontekstissa.

5.3.3 Muutoksen jalanjäljen määrittely

Kaikkia tietomallin ilmentymiä ei aina tarvitse manuaalisesti päivittää tietomallin muuttumisen jälkeen. Tästä johtuen analyysissa otetaan huomioon tämän muutosprosessin automatisoidut osat, sekä myös hyvin pintapuoleinen käsitys siitä, kuinka käyttökelpoisia automaattisesti generoidut moduulit ovat.

5.4 Sovellettavat työkalut

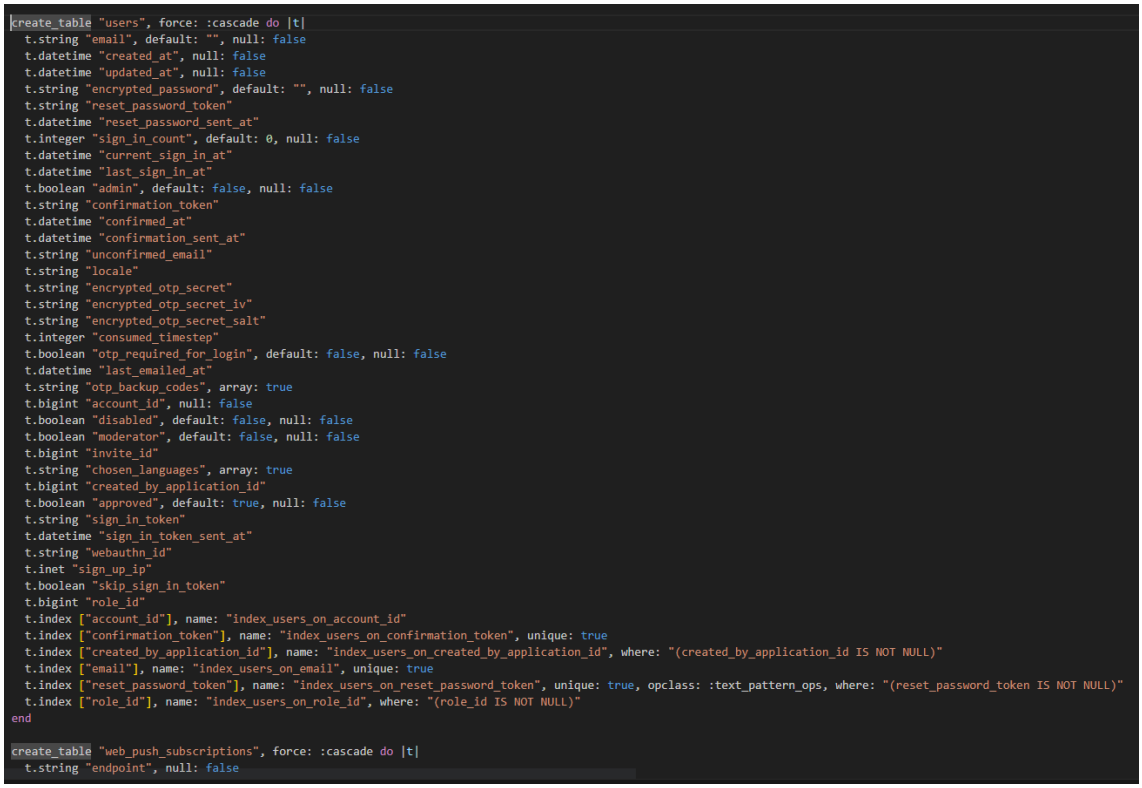
Tietomallien toteutuksien analyysin yhteydessä tutkittiin myös mahdollisuuksia automatisoida analyysiprosessia soveltamalla koodin laatua diagnosoivia työkaluja, mutta tämä ei osoittautunut ainakaan lyhyen kokeilun myötä toimivaksi ratkaisuksi. Kahta tällaista työkalua harkittiin hyödynnettäväksi tutkittavien kohteiden lähdekoodin analysointiin.

Yksi työkaluista oli SonarLint[21], joka on jatkuvaa integrointia tukevan lähdeanalyysityökalu SonarQube:n IDE osa. Tämä työkalu kuitenkin osoittautui erikoistuvan liiaksi yksittäisten tiedostojen koodirivien tyyliongelmiin, eikä tässä tutkimuksessa vaadittavaan projektin laajuiseen analyysiin.

Toinen työkaluista oli OpenTelemetry[22], joka on yksi suosituimmista koodin toiminnan seuraamiseen käytetyistä standardeista, jota käytetään sovellusten hajautettujen osien havainnointiin. Työkalun toiminta riippuu kuitenkin sovelluksen koodin instrumentoinnista, eli tarvittavien merkkien sisällyttämisestä lähdekoodiin. Tämä prosessi on mahdollista automatisoida, mutta automatisoitu instrumentointi tarjoaa pelkästään tämän tutkimuksen kannalta liian pinnallista diagnostiikkaa, kuten osoitteiden latenssi, tietokannan hakujen nopeus tai liikenteen määrä. Manuaalinen instrumentointi on toki mahdollista, mutta tämä vaatisi jo yksinään tutkimuksen tässä osiossa tehtävän analyysin suorittamisen instrumentointia varten, joten työkalusta ei sinänsä ole tässä tilanteesta hyötyä muuten kun esimerkiksi graafien luontiin, mikä ei tämän tutkimuksen kontekstissa ole kohtuullinen etu suhteessa vaadittuun työmäärään.

5.5 Mastodon

Mastodon perustuu verkkosovelluksissa yleiseen Ruby -ohjelmointikieleen pohjautuvaan Ruby on Rails viitekehykseen. Tähän viitekehykseen kuulu mukaan ORM



```
create_table "users", force: :cascade do |t|
  t.string "email", default: "", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.string "encrypted_password", default: "", null: false
  t.string "reset_password_token"
  t.datetime "reset_password_sent_at"
  t.integer "sign_in_count", default: 0, null: false
  t.datetime "current_sign_in_at"
  t.datetime "last_sign_in_at"
  t.boolean "admin", default: false, null: false
  t.string "confirmation_token"
  t.datetime "confirmed_at"
  t.datetime "confirmation_sent_at"
  t.string "unconfirmed_email"
  t.string "locale"
  t.string "encrypted_otp_secret"
  t.string "encrypted_otp_secret_iv"
  t.string "encrypted_otp_secret_salt"
  t.integer "consumed_timestep"
  t.boolean "otp_required_for_login", default: false, null: false
  t.datetime "last_emailed_at"
  t.string "otp_backup_codes", array: true
  t.bigint "account_id", null: false
  t.boolean "disabled", default: false, null: false
  t.boolean "moderator", default: false, null: false
  t.bigint "invite_id"
  t.string "chosen_languages", array: true
  t.bigint "created_by_application_id"
  t.boolean "approved", default: true, null: false
  t.string "sign_in_token"
  t.datetime "sign_in_token_sent_at"
  t.string "webauthn_id"
  t.inet "sign_up_ip"
  t.boolean "skip_sign_in_token"
  t.bigint "role_id"
  t.index ["account_id"], name: "index_users_on_account_id"
  t.index ["confirmation_token"], name: "index_users_on_confirmation_token", unique: true
  t.index ["created_by_application_id"], name: "index_users_on_created_by_application_id", where: "(created_by_application_id IS NOT NULL)"
  t.index ["email"], name: "index_users_on_email", unique: true
  t.index ["reset_password_token"], name: "index_users_on_reset_password_token", unique: true, opclass: :text_pattern_ops, where: "(reset_password_token IS NOT NULL)"
  t.index ["role_id"], name: "index_users_on_role_id", where: "(role_id IS NOT NULL)"
end

create_table "web_push_subscriptions", force: :cascade do |t|
  t.string "endpoint", null: false
```

Kuva 5.1: Mastodon tietokantamalli

-toiminnallisuus nimeltä Active Record. Active Record yksinkertaistaa tietomallitoteutusta mahdollistamalla interaktion tietokannan välillä tietokantamallin pohjalta generoitujen Ruby -luokkien kautta. Sovelluksen tietomallin kuvaukset sisältävät moduulit löytyvät taulokoista 5.3 ja 5.4. Tietomallin muutoksen jalanjälki koodissa sekä muokattujen koodirivien määrä löytyvät kohdasta 5.9.1 taulukosta 5.11.

Sovelluksen tietokantamallia muokataaan yksittäisinä migraatioina käyttäen viitekehyksen tarjoamia työkaluja. Koko tietokantamalli on kuitenkin luettavissa yhdessä nykyisen tietorakenteen pohjalta automaattisesti generoidussa tiedostossa db/schema.rb. Tietomallille tehdään kohdassa 5.3.1 esitellyt muutokset. Samassa kohdassa määriteltyjen kriteerien mukaan muokattavaksi valittava taulu on *users*. Kyseinen taulu näkyy kuvan 5.1 tiedostossa db/schema.rb.

Yllä mainittu ORM Active Record mahdollistaa tietokannan taulujen käytön

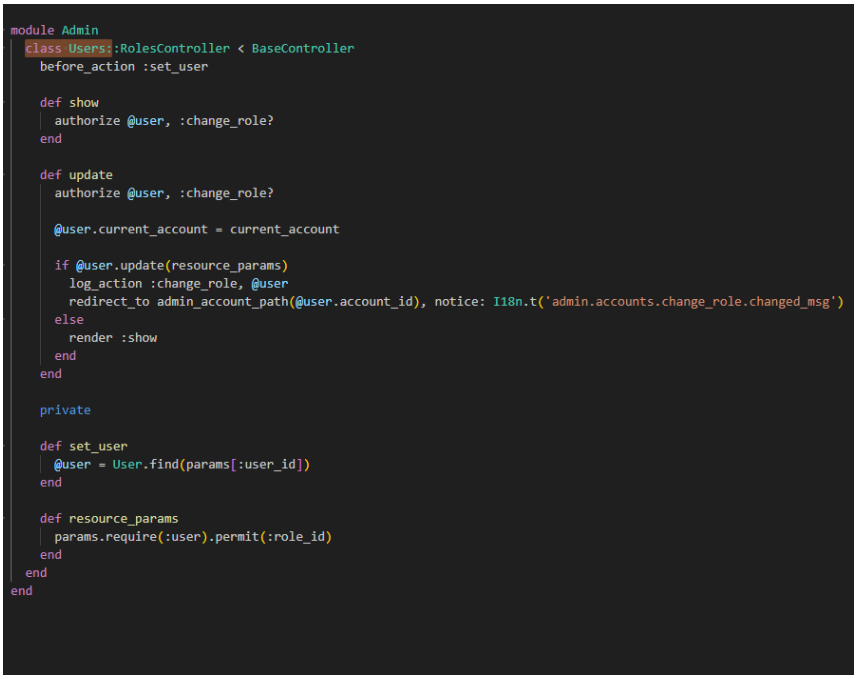
```ruby
class User < ApplicationRecord
  self.ignored_columns = %w(
    remember_created_at
    remember_token
    current_sign_in_ip
    last_sign_in_ip
    skip_sign_in_token
    filtered_languages
  )

  include Settings::Extend
  include Redisable
  include LanguagesHelper

  # The home and list feeds will be stored in Redis for this amount
  # of time, and status fan-out to followers will include only people
  # within this time frame. Lowering the duration may improve performance
  # if lots of people sign up, but not a lot of them check their feed
  # every day. Raising the duration reduces the amount of expensive
  # RegenerationWorker jobs that need to be run when those people come
  # to check their feed
  ACTIVE_DURATION = ENV.fetch('USER_ACTIVE_DAYS', 7).to_i.days.freeze

  devise :two_factor_authenticatable,
         otp_secret_encryption_key: Rails.configuration.x.otp_secret

  devise :two_factor_backupable,
         otp_number_of_backup_codes: 10

  devise :registerable, :recoverable, :validatable,
         :confirmable

  include Omniauthable
  include PamAuthenticable
  include LdapAuthenticable

  belongs_to :account, inverse_of: :user
  belongs_to :invite, counter_cache: :uses, optional: true
  belongs_to :created_by_application, class_name: 'Doorkeeper::Application', optional: true
  belongs_to :role, class_name: 'UserRole', optional: true
  accepts_nested_attributes_for :account

  has_many :applications, class_name: 'Doorkeeper::Application', as: :owner
  has_many :backups, inverse_of: :user
  has_many :invites, inverse_of: :user
  has_many :markers, inverse_of: :user, dependent: :destroy
  has_many :webauthn_credentials, dependent: :destroy
  has_many :ips, class_name: 'UserIp', inverse_of: :user
```

Kuva 5.2: Mastodon Object Relational Mapper

luokkina muualla koodissa. Tätä varten on kuitenkin luotava uusi kattava tietomallin toteutus, sillä näitä luokkia ei ole mahdollista generoida automaattisesti. Uuden taulun luokka luodaan kohteeseen app/models/. ORM -luokan määrittely näkyy kuvan 5.2 tiedostossa app/models/user.rb.

Käyttöliittymän, palvelimella ajetun koodin ja tietokannan välisenä rajapintana toimivat *controller* -polut. Nämä polut täytyy generoida yksitellen, mutta ne käyttävät ORM:n luokkia, joten tietomallin vaatimusten muuttuminen ei tuota merkittävästi huomioitavaa näissä tiedostoissa. ORM -luokan esimerkki rajapintaa ohjaavasta tiedostosta app/controllers/admin/users/roles_controller.rb näkyy kuvassa 5.3.

Tietomallin osille määriteltyjen toimintojen rajapintakoodi määritellään palveluina hakemistoon app/services/. Näitä tiedostoja ei voi generoida automaattisesti,



```ruby
module Admin
  class Users::RolesController < BaseController
    before_action :set_user

    def show
      authorize @user, :change_role?
    end

    def update
      authorize @user, :change_role?

      @user.current_account = current_account

      if @user.update(resource_params)
        log_action :change_role, @user
        redirect_to admin_account_path(@user.account_id), notice: I18n.t('admin.accounts.change_role.changed_msg')
      else
        render :show
      end
    end

    private

    def set_user
      @user = User.find(params[:user_id])
    end

    def resource_params
      params.require(:user).permit(:role_id)
    end
  end
end
```

Kuva 5.3: Mastodon rajapinta

mutta myös nämä tiedostot käyttävät samoja ORM -malleja. Tietomallin vaatimusten muutos ei myöskään välttämättä vaadi muutoksia näihin palveluihin, koska palveluiden ei ole pakko olla liitettynä mihinkään luokkien osaan.

Taulukko 5.3: Mastodon moduulit tietomallia muuttaessa

moduuli	*kuvaus*	*sijainti*
tietokantamalli	PostgreSQL. Migraatio tehdään manuaalisesti Ruby -koodilla.	db/schema.rb
ORM	Active Record. Luokkien kentät määriteltävä manuaalisesti.	app/models/
API route	Hyödyntävät ORM luokkia.	app/controllers/

Taulukko 5.4: Mastodon moduulit tietomalliin lisättäessä

moduuli	*kuvaus*	*sijainti*
tietokantamalli	PostgreSQL. Taulujen lisäys hoituu alkuun komentoriviltä.	db/schema.rb
ORM	Active Record. Uudet kentät määriteltävä manuaalisesti.	app/models/
API route	Generoitava yksitellen. Käyttää ORM luokkaa.	app/controllers/

5.6 PrestaShop

PrestaShop -verkkosovellus perustuu Symfony -viitekehykseen. Sovelluksen arkkitehtuuri koostuu kahdesta rinnakkaisesta MVC -pinosta. Niin kutsuttu *Front Office* -pino koostuu käyttöliittymää ohjaavasta *controllers/front* -hakemistosta, verkkokauppatoimintoja ohjaavasta classes hakemistosta ja käyttöliittymän tyylit sisältävästä *themes* -kansiosta. Järjestelmänvalvojan sovelluksen *Front Office* -pino koostuu Symfony -hallintasovelluksesta hakemistossa src/PrestaShopBundle/Controller/Admin. Sovelluksen tietomallin kuvaukset sisältävät moduulit löytyvät taulokoista 5.5 ja 5.6. Tietomallin muutoksen jalanjälki koodissa sekä muokattujen koodirivien määrä löytyvät kohdasta 5.9.1 taulukosta 5.11.

Tietokantamalli tietokannan rakenteen määrittämiseksi löytyy tiedostosta install-dev/data/db_structure.sql. Tietokantamallin muutokset toteutetaan migraatioina, jotka kirjoitetaan puhtaina .sql tiedostoina ilman rajapintana toimivaa kehystä. Alkuperäisen asunnuksen jälkeen lisätyt taulut tarvitsevat myös omat toteutukset kohteeseen /upgrade/php/. Tietomallille tehdään kohdassa 5.3.1 esitellyt muutokset. Samassa kohdassa määriteltyjen kriteerien mukaan muokattavaksi valittava taulu on *products*. Kyseinen taulu näkyy kuvan 5.4 tiedostossa install-dev/data/db_structure.sql.

Sovellus käyttää ORM -työkalua nimeltä *Doctrine*, mutta tämä on käytössä vain joillakin tietokannan tauluilla. ORM -luokan määrittely näkyy kuvan 5.5 tiedostossa src/PrestaShopBundle/Entity/ProductDownload.php.

PrestaShop sisältää kaksi ulospäin suunnattua rajapintaa. Sovelluksen arkkitehtuuri toteuttaa *adapter* -tyyliä. Tämän mukaisesti käyttöliittymän rajapinta tarvitsee oman luokkarakenteen, jotka toteutetaan kuvan 5.6 mukaan src/Adapter/ hakemistossa. Esimerkki Adapter -tiedostosta näkyy kuvan 5.6 tiedostossa src/Adapter/Product/QueryHandler/GetProductForEditingHandler.php.

Doctrine ORM ei määrittele kaikkia tietomallin olioita, joten osa pitää määrittää luokkarakenteessa. Tällainen määritelmä näkyy kuvan 5.7 tiedostossa classes/Pro-

```
/* list of products */
CREATE TABLE `PREFIX_product` (
  `id_product` int(10) unsigned NOT NULL auto_increment,
  `id_supplier` int(10) unsigned DEFAULT NULL,
  `id_manufacturer` int(10) unsigned DEFAULT NULL,
  `id_category_default` int(10) unsigned DEFAULT NULL,
  `id_shop_default` int(10) unsigned NOT NULL DEFAULT 1,
  `id_tax_rules_group` INT(11) UNSIGNED NOT NULL,
  `on_sale` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `online_only` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `ean13` varchar(13) DEFAULT NULL,
  `isbn` varchar(32) DEFAULT NULL,
  `upc` varchar(12) DEFAULT NULL,
  `mpn` varchar(40) DEFAULT NULL,
  `ecotax` decimal(17, 6) NOT NULL DEFAULT '0.00',
  `quantity` int(10) NOT NULL DEFAULT '0',
  `minimal_quantity` int(10) unsigned NOT NULL DEFAULT '1',
  `low_stock_threshold` int(10) NULL DEFAULT NULL,
  `low_stock_alert` TINYINT(1) NOT NULL DEFAULT 0,
  `price` decimal(20, 6) NOT NULL DEFAULT '0.000000',
  `wholesale_price` decimal(20, 6) NOT NULL DEFAULT '0.000000',
  `unity` varchar(255) DEFAULT NULL,
  `unit_price` decimal(20, 6) NOT NULL DEFAULT '0.000000',
  `unit_price_ratio` decimal(20, 6) NOT NULL DEFAULT '0.000000',
  `additional_shipping_cost` decimal(20, 6) NOT NULL DEFAULT '0.000000',
  `reference` varchar(64) DEFAULT NULL,
  `supplier_reference` varchar(64) DEFAULT NULL,
  `location` varchar(255) NOT NULL DEFAULT '',
  `width` DECIMAL(20, 6) NOT NULL DEFAULT '0',
  `height` DECIMAL(20, 6) NOT NULL DEFAULT '0',
  `depth` DECIMAL(20, 6) NOT NULL DEFAULT '0',
  `weight` DECIMAL(20, 6) NOT NULL DEFAULT '0',
  `out_of_stock` int(10) unsigned NOT NULL DEFAULT '2',
  `additional_delivery_times` tinyint(1) unsigned NOT NULL DEFAULT '1',
  `quantity_discount` tinyint(1) DEFAULT '0',
  `customizable` tinyint(2) NOT NULL DEFAULT '0',
  `uploadable_files` tinyint(4) NOT NULL DEFAULT '0',
  `text_fields` tinyint(4) NOT NULL DEFAULT '0',
  `active` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `redirect_type` ENUM(
    '', '404', '410', '301-product', '302-product',
    '301-category', '302-category', '200-displayed',
    '404-displayed', '410-displayed', 'default'
  ) NOT NULL DEFAULT 'default',
  `id_type_redirected` int(10) unsigned NOT NULL DEFAULT '0',
  `available_for_order` tinyint(1) NOT NULL DEFAULT '1',
  `available_date` date DEFAULT NULL,
```

Kuva 5.4: PrestaShop tietokantamalli

```php
namespace PrestaShopBundle\Entity;

use DateTime;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 * @ORM\Entity
 */
class ProductDownload
{
    /**
     * @var int
     *
     * @ORM\Id
     * @ORM\Column(name="id_product_download", type="integer", options={"unsigned"=true})
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var int
     *
     * @ORM\Column(name="id_product", type="integer", options={"unsigned"=true})
     */
    private $idProduct;

    /**
     * @var string
     *
     * @ORM\Column(name="display_filename", type="string", length=255, nullable=true)
     */
    private $displayFilename;

    /**
     * @var string
     *
     * @ORM\Column(name="filename", type="string", length=255, nullable=true)
     */
    private $filename;

    /**
     * @var DateTime
     *
     * @ORM\Column(name="date_add", type="datetime")
     */
```

Kuva 5.5: PrestaShop Object Relational Mapper

```php
/**
 * {@inheritdoc}
 */
public function handle(GetProductForEditing $query): ProductForEditing
{
    $product = $this->productRepository->getByShopConstraint(
        $query->getProductId(),
        $query->getShopConstraint()
    );

    return new ProductForEditing(
        (int) $product->id,
        $product->getProductType(),
        (bool) $product->active,
        $this->getCustomizationOptions($product),
        $this->getBasicInformation($product),
        $this->getCategoriesInformation($product, $query->getDisplayLanguageId()),
        $this->getPricesInformation($product, $query->getShopConstraint()),
        $this->getOptions($product),
        $this->getDetails($product),
        $this->getShippingInformation($product),
        $this->getSeoOptions($product),
        $this->getAttachments($query->getProductId()),
        $this->getProductStockInformation($product),
        $this->getVirtualProductFile($product),
        $this->getCover($query->getProductId(), $product->getShopId())
    );
}

/**
 * @param ProductId $productId
 *
 * @return AttachmentInformation[]
 */
private function getAttachments(ProductId $productId): array
{
    $attachments = $this->attachmentRepository->getProductAttachments($productId);

    $attachmentsInfo = [];
    foreach ($attachments as $attachment) {
        $attachmentsInfo[] = new AttachmentInformation(
            (int) $attachment['id_attachment'],
            $attachment['name'],
            $attachment['description'],
            $attachment['file_name'],
            $attachment['mime'],
            (int) $attachment['file_size']
        );
```

Kuva 5.6: PrestaShop rajapinta

```php
class ProductCore extends ObjectModel
{
    /**
     * @var string Tax name
     *
     * @deprecated Since 1.4
     */
    public $tax_name;

    /** @var float Tax rate */
    public $tax_rate;

    /** @var int Manufacturer identifier */
    public $id_manufacturer;

    /** @var int Supplier identifier */
    public $id_supplier;

    /** @var int default Category identifier */
    public $id_category_default;

    /** @var int default Shop identifier */
    public $id_shop_default;

    /** @var string Manufacturer name */
    public $manufacturer_name;

    /** @var string Supplier name */
    public $supplier_name;

    /** @var string|array Name or array of names by id_lang */
    public $name;

    /** @var string|array Long description or array of long description by id_lang */
    public $description;

    /** @var string|array Short description or array of short description by id_lang */
    public $description_short;

    /**
     * @deprecated since 1.7.8
     * @see StockAvailable::$quantity instead
     *
     * @var int Quantity available
     */
    public $quantity = 0;
```

Kuva 5.7: PrestaShop luokat

duct.php.

5.7 Laravel.io Community Portal

Nimensä mukaisesti Laravel.io -verkkosovellus käyttää suosittua Laravel PHP -verkkosovellusviit
Sovelluksen yksinkertainen rakenne koostuu *app* -hakemistossa olevasta palvelinpuolen koodista ja *resources* -hakemistossa olevasta käyttöliittymän koodista. Sovelluksen tietomallin kuvaukset sisältävät moduulit löytyvät taulokoista 5.7 ja 5.8. Tietomallin muutoksen jalanjälki koodissa sekä muokattujen koodirivien määrä löytyvät kohdasta 5.9.1 taulukosta 5.11.

Taulukko 5.5: PrestaShop moduulit tietomallia muuttaessa

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Muutokset tietomalliin puhtaalla sql koodilla.	install-dev/data/db_structure.sql
ORM	Doctrine. Ei pakollinen riippuen luokan toiminnallisuuksista.	src/PrestaShopBundle/Entity/
API route	Adapter design pattern	src/Adapter/
luokat	Luokkien manualinen määrittely.	classes/

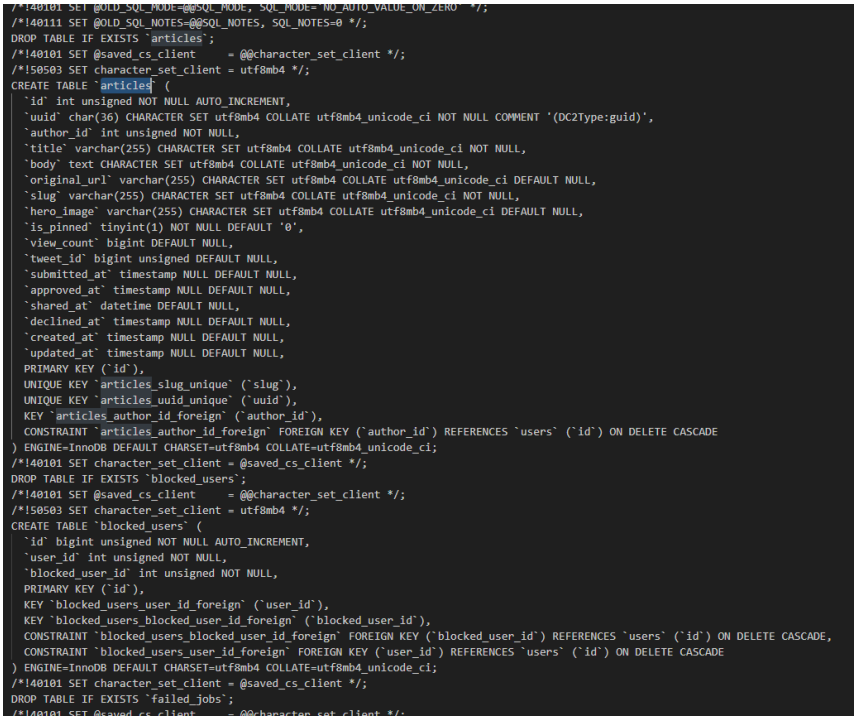
Taulukko 5.6: PrestaShop moduulit tietomalliin lisättäessä

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Uusien taulujen lisäys puhtaalla sql koodilla.	install-dev/data/db_structure.sql
API query	Rajapinta pelkästään asennuksen jälkeen lisätyille tauluille.	/upgrade/php/
ORM	Doctrine. Ei pakollinen riippuen luokan toiminnallisuuksista.	src/PrestaShopBundle/Entity/
luokat	Luokkien manualinen määrittely.	classes/

Tässä projektissa tietokannan taulut eivät ole muokattavissa yhdessä tiedostossa ohjelmoijan määrittämässä tietokantamallissa(englanniksi *schema*). Sen sijaan tietokannan rakenteeseen tehdään muokkauksia Laravel:n työkaluilla, jotka tallentuvat erillisinä .php tiedostoina kansioon *database/migrations*. Tiedostojen pohjalta voi myös generoida yhden tiedoston, joka tiivistää migraatiot yhteen. Tietomallille tehdään kohdassa 5.3.1 esitellyt muutokset. Samassa kohdassa määriteltyjen kriteerien mukaan muokattavaksi valittava taulu on *articles*. Kyseinen taulu näkyy kuvan 5.8 tiedostossa database/schema/mysql-schema.dump.

Laravel viitekehys sisältää ORM -ratkaisun nimeltä *Eloquent*. Tämä ominaisuus mahdollistaa geneeristen luokkien automaattisen generoinnin tietokannan rakenteen pohjalta. Tietomallin muutosten propagoitumisen kannalta ratkaisu on hyvin toiminva, sillä valtaosa sovelluksen muista osista tarvitsee ainoastaan näitä luokkia rajapinnakseen tietomallin kannsa. ORM -luokan määrittely näkyy kuvan 5.9 tiedostossa app/Models/Article.php.

Tietokannan elementteihin liittyvä rajapinnan logiikka ja CRUD -operaatiot on kerätty *Controllers* -tiedostoihin. Laravel pystyy generoimaan yksinkertaiset versiot näistä. Näiden Controller -luokkien asynkroniset käsittelijäfunktiot pystytään



```
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
DROP TABLE IF EXISTS `articles`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `articles` (
  `id` int unsigned NOT NULL AUTO_INCREMENT,
  `uuid` char(36) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '(DC2Type:guid)',
  `author_id` int unsigned NOT NULL,
  `title` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
  `body` text CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
  `original_url` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `slug` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL,
  `hero_image` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `is_pinned` tinyint(1) NOT NULL DEFAULT '0',
  `view_count` bigint DEFAULT NULL,
  `tweet_id` bigint unsigned DEFAULT NULL,
  `submitted_at` timestamp NULL DEFAULT NULL,
  `approved_at` timestamp NULL DEFAULT NULL,
  `shared_at` datetime DEFAULT NULL,
  `declined_at` timestamp NULL DEFAULT NULL,
  `created_at` timestamp NULL DEFAULT NULL,
  `updated_at` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `articles_slug_unique` (`slug`),
  UNIQUE KEY `articles_uuid_unique` (`uuid`),
  KEY `articles_author_id_foreign` (`author_id`),
  CONSTRAINT `articles_author_id_foreign` FOREIGN KEY (`author_id`) REFERENCES `users` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
DROP TABLE IF EXISTS `blocked_users`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `blocked_users` (
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int unsigned NOT NULL,
  `blocked_user_id` int unsigned NOT NULL,
  PRIMARY KEY (`id`),
  KEY `blocked_users_user_id_foreign` (`user_id`),
  KEY `blocked_users_blocked_user_id_foreign` (`blocked_user_id`),
  CONSTRAINT `blocked_users_blocked_user_id_foreign` FOREIGN KEY (`blocked_user_id`) REFERENCES `users` (`id`) ON DELETE CASCADE,
  CONSTRAINT `blocked_users_user_id_foreign` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
DROP TABLE IF EXISTS `failed_jobs`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
```

Kuva 5.8: Laravel.io tietokantamalli

myös generoimaan automaattisesti, jolloin ne löytyvät eri kansiosta nimeltä *Jobs*. Rajapintatiedostot käyttävät *Eloquent* luokkia, mutta hakutoiminnallisuudet riippuvat näiden omasta tietomallin kuvauksesta. Esimerkki Controller -luokasta näkyy kuvan 5.10 tiedostossa app/Http/Controllers/Articles/ArticlesController.php. Esimerkki rajapinnasta näkyy kuvan 5.11 tiedostossa app/Http/Resources/ArticleResource.php.

Taulukko 5.7: Laravel.io moduulit tietomallia muuttaessa

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Tietokannan nykyisen mallin tarkastelu vaati erillisen ohjelmiston.	database/migrations
ORM	Eloquent. Kaikki kentät määriteltävä manuualisesti.	app/models/
API route	Kentät tulosten hakemista varten määriteltävä manuualisesti.	app/Http/Controllers/
käyttöliittymä	Riippuu luokan toiminnallisuuksista.	app/jobs/

```php
final class Article extends Model implements Feedable
{
    use HasFactory;
    use HasAuthor;
    use HasSlug;
    use HasLikes;
    use HasTimestamps;
    use HasTags;
    use HasUuid;
    use PreparesSearch;
    use Searchable;

    const TABLE = 'articles';

    const FEED_PAGE_SIZE = 20;

    /**
     * {@inheritdoc}
     */
    protected $fillable = [
        'uuid',
        'title',
        'body',
        'original_url',
        'slug',
        'hero_image',
        'is_pinned',
        'view_count',
        'tweet_id',
        'submitted_at',
        'approved_at',
        'declined_at',
        'shared_at',
    ];

    /**
     * {@inheritdoc}
     */
    protected $casts = [
        'submitted_at' => 'datetime',
        'approved_at' => 'datetime',
        'shared_at' => 'datetime',
    ];

    /**
     * {@inheritdoc}
     */
    protected $with = [
```

Kuva 5.9: Laravel.io Object Relational Mapper

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

/**
 * @mixin \App\Models\Article
 */
class ArticleResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
            'id' => $this->getKey(),
            'url' => route('articles.show', $this->slug()),
            'title' => $this->title(),
            'body' => $this->body(),
            'original_url' => $this->originalUrl(),
            'author' => AuthorResource::make($this->author()),
            'tags' => TagResource::collection($this->tags()),
            'is_submitted' => $this->isSubmitted(),
            'submitted_at' => $this->submittedAt(),
            'created_at' => $this->createdAt(),
            'updated_at' => $this->updatedAt(),
        ];
    }
}
```

Kuva 5.10: Laravel.io Controller -luokka

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

/**
 * @mixin \App\Models\Article
 */
class ArticleResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
            'id' => $this->getKey(),
            'url' => route('articles.show', $this->slug()),
            'title' => $this->title(),
            'body' => $this->body(),
            'original_url' => $this->originalUrl(),
            'author' => AuthorResource::make($this->author()),
            'tags' => TagResource::collection($this->tags()),
            'is_submitted' => $this->isSubmitted(),
            'submitted_at' => $this->submittedAt(),
            'created_at' => $this->createdAt(),
            'updated_at' => $this->updatedAt(),
        ];
    }
}
```

Kuva 5.11: Laravel.io rajapinta

Taulukko 5.8: Laravel.io moduulit tietomalliin lisättäessä

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Lisääminen manuaalisesti tai Artisan komentorivityökalulla.	database/migrations
ORM	Eloquent. Kaikki kentät määriteltävä manuualisesti.	app/models/

5.8 WordPress

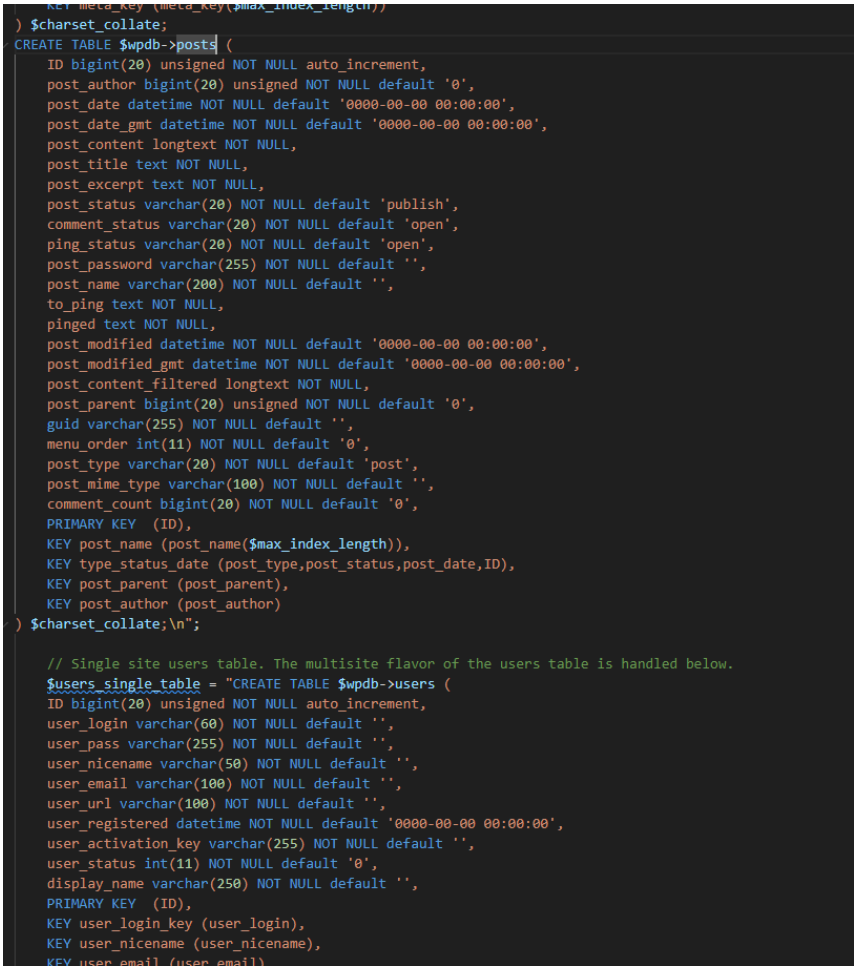
WordPressin projektirakenne koostuu kolmesta kokonaisuudesta. Kansio *wp-admin* keskityy sovelluksen käyttöliittymän ja sen työkalujen toimintaan. Kansio *wp-includes* hoitaa sovelluksen taustaprosesseja ja mahdollistaa kommunikaation rajapintojen ja tietokannan välillä. Kansio *wp-content* mahdollistaa omien lisäosien liittämisen sovellukseen. Sovelluksen tietomallin kuvaukset sisältävät moduulit löytyvät taulukoista 5.9 ja 5.10. Tietomallin muutoksen jalanjälki koodissa sekä muokattujen koodirivien määrä löytyvät kohdasta 5.9.1 taulukosta 5.11.

Sovelluksen tietokantamalli löytyy tiedostosta wp-admin/includes/schema.php, joten määritellään se ensimmäiseksi moduuliksi. Valmiiksi olemassa olevat taulut ovat *wp_commentmeta*, *wp_comments*, *wp_links*, *wp_options*, *wp_postmeta*, *wp_posts*, *wp_termmeta*, *wp_terms*, *wp_term_relationships*, *wp_term_taxonomy*, *wp_usermeta* ja *wp_users*. Tietomallille tehdään kohdassa 5.3.1 esitellyt muutokset. Samassa kohdassa määriteltyjen kriteerien mukaan muokattavaksi valittava taulu on *posts*. Kyseinen taulu näkyy kuvan 5.12 tiedostossa wp-admin/includes/schema.php.

Seuraava kerros on tietokantaa hakeva rajapinta. Tämä sisältää wp_query -funktiot ja manuaaliset tietomallin elementtien määrittelyt luokkina. Rajapinnan käyttämä luokka näkyy kuvan 5.13 tiedostossa wp-includes/rest-api/endpoints/class-wp-rest-posts-controller.php.

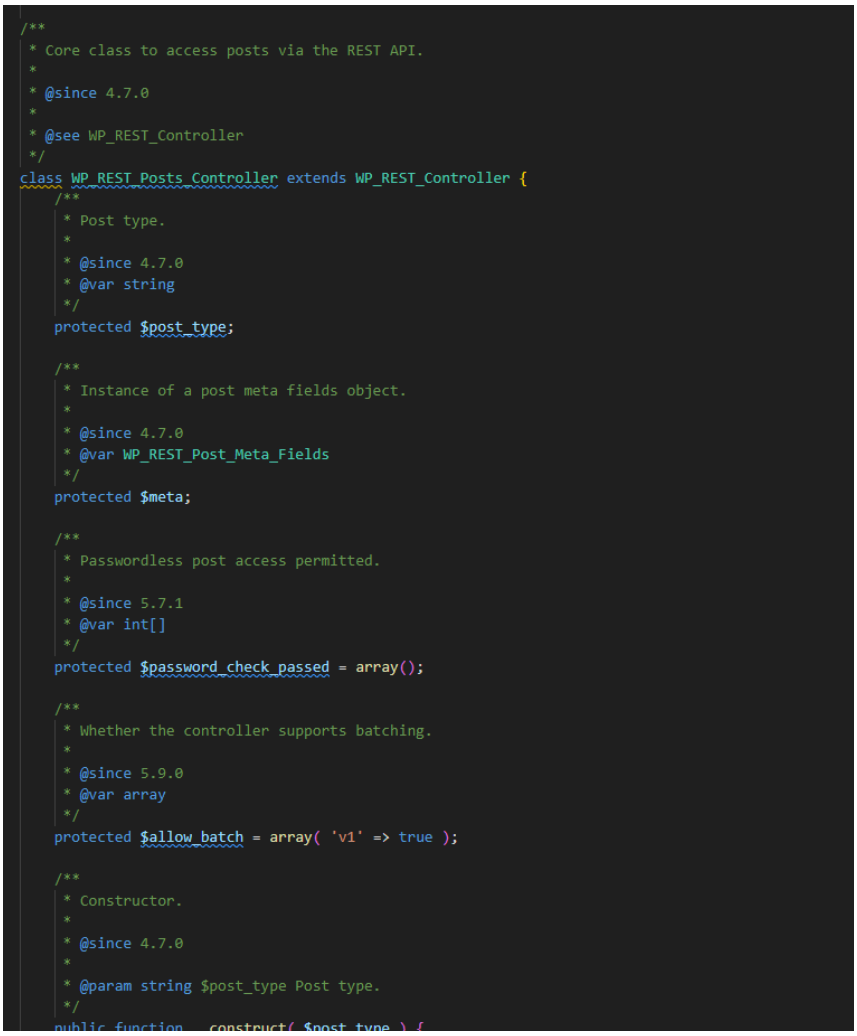
Rajapinnan luokkien lisäksi käyttöliittymän koodissa käytettävät oliot vaativat myös oman määrittelynsä. Esimerkki tällaisesta luokan määrittelystä näkyy kuvan 5.14 tiedostossa wp-includes/class-wp-post.php.

Yllämainittujen luokkien lisäksi on myös määritelty erikseen niin sanotut *core*



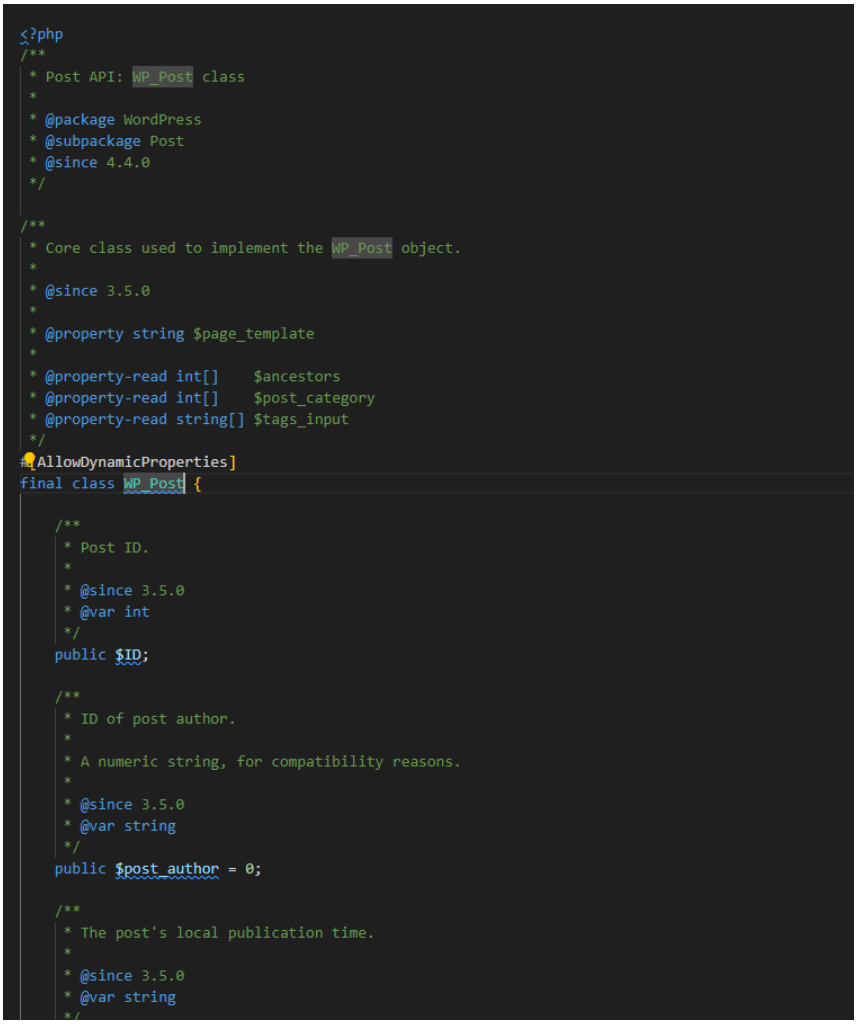
```
    KEY meta_key (meta_key($max_index_length))
) $charset_collate;
CREATE TABLE $wpdb->posts (
    ID bigint(20) unsigned NOT NULL auto_increment,
    post_author bigint(20) unsigned NOT NULL default '0',
    post_date datetime NOT NULL default '0000-00-00 00:00:00',
    post_date_gmt datetime NOT NULL default '0000-00-00 00:00:00',
    post_content longtext NOT NULL,
    post_title text NOT NULL,
    post_excerpt text NOT NULL,
    post_status varchar(20) NOT NULL default 'publish',
    comment_status varchar(20) NOT NULL default 'open',
    ping_status varchar(20) NOT NULL default 'open',
    post_password varchar(255) NOT NULL default '',
    post_name varchar(200) NOT NULL default '',
    to_ping text NOT NULL,
    pinged text NOT NULL,
    post_modified datetime NOT NULL default '0000-00-00 00:00:00',
    post_modified_gmt datetime NOT NULL default '0000-00-00 00:00:00',
    post_content_filtered longtext NOT NULL,
    post_parent bigint(20) unsigned NOT NULL default '0',
    guid varchar(255) NOT NULL default '',
    menu_order int(11) NOT NULL default '0',
    post_type varchar(20) NOT NULL default 'post',
    post_mime_type varchar(100) NOT NULL default '',
    comment_count bigint(20) NOT NULL default '0',
    PRIMARY KEY  (ID),
    KEY post_name (post_name($max_index_length)),
    KEY type_status_date (post_type,post_status,post_date,ID),
    KEY post_parent (post_parent),
    KEY post_author (post_author)
) $charset_collate;\n";

    // Single site users table. The multisite flavor of the users table is handled below.
    $users_single_table = "CREATE TABLE $wpdb->users (
    ID bigint(20) unsigned NOT NULL auto_increment,
    user_login varchar(60) NOT NULL default '',
    user_pass varchar(255) NOT NULL default '',
    user_nicename varchar(50) NOT NULL default '',
    user_email varchar(100) NOT NULL default '',
    user_url varchar(100) NOT NULL default '',
    user_registered datetime NOT NULL default '0000-00-00 00:00:00',
    user_activation_key varchar(255) NOT NULL default '',
    user_status int(11) NOT NULL default '0',
    display_name varchar(250) NOT NULL default '',
    PRIMARY KEY  (ID),
    KEY user_login_key (user_login),
    KEY user_nicename (user_nicename),
    KEY user_email (user_email)
```

Kuva 5.12: WordPress tietokantamalli



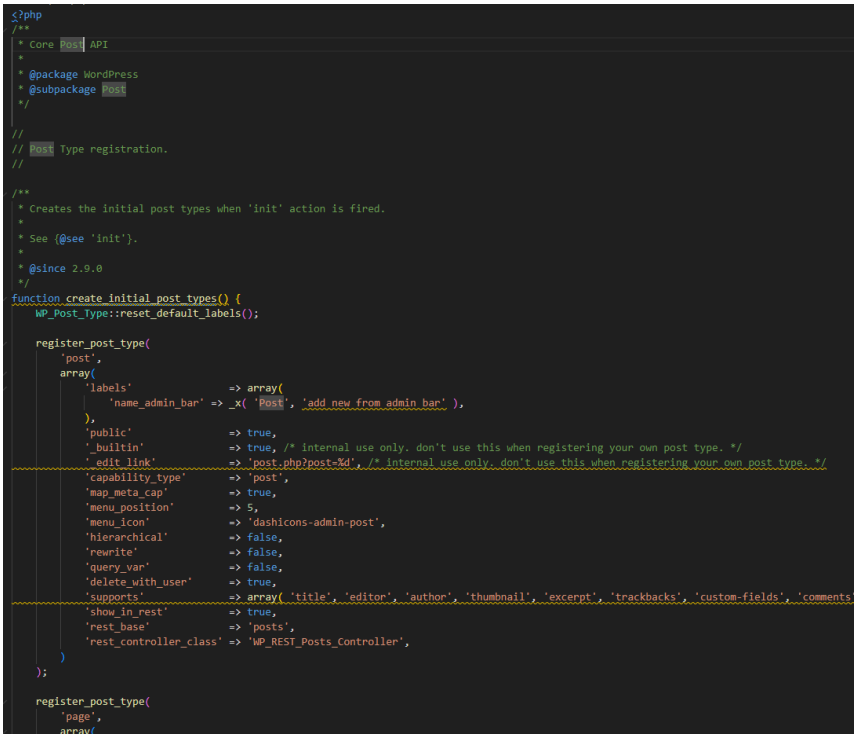
```php
/**
 * Core class to access posts via the REST API.
 *
 * @since 4.7.0
 *
 * @see WP_REST_Controller
 */
class WP_REST_Posts_Controller extends WP_REST_Controller {
    /**
     * Post type.
     *
     * @since 4.7.0
     * @var string
     */
    protected $post_type;

    /**
     * Instance of a post meta fields object.
     *
     * @since 4.7.0
     * @var WP_REST_Post_Meta_Fields
     */
    protected $meta;

    /**
     * Passwordless post access permitted.
     *
     * @since 5.7.1
     * @var int[]
     */
    protected $password_check_passed = array();

    /**
     * Whether the controller supports batching.
     *
     * @since 5.9.0
     * @var array
     */
    protected $allow_batch = array( 'v1' => true );

    /**
     * Constructor.
     *
     * @since 4.7.0
     *
     * @param string $post_type Post type.
     */
    public function __construct( $post_type ) {
```

Kuva 5.13: WordPress rajapinta



```php
<?php
/**
 * Post API: WP_Post class
 *
 * @package WordPress
 * @subpackage Post
 * @since 4.4.0
 */

/**
 * Core class used to implement the WP_Post object.
 *
 * @since 3.5.0
 *
 * @property string $page_template
 *
 * @property-read int[]    $ancestors
 * @property-read int[]    $post_category
 * @property-read string[] $tags_input
 */
#[AllowDynamicProperties]
final class WP_Post {

    /**
     * Post ID.
     *
     * @since 3.5.0
     * @var int
     */
    public $ID;

    /**
     * ID of post author.
     *
     * A numeric string, for compatibility reasons.
     *
     * @since 3.5.0
     * @var string
     */
    public $post_author = 0;

    /**
     * The post's local publication time.
     *
     * @since 3.5.0
     * @var string
     */
```

Kuva 5.14: WordPress luokat



```php
<?php
/**
 * Core Post API
 *
 * @package WordPress
 * @subpackage Post
 */

//
// Post Type registration.
//

/**
 * Creates the initial post types when 'init' action is fired.
 *
 * See {@see 'init'}.
 *
 * @since 2.9.0
 */
function create_initial_post_types() {
    WP_Post_Type::reset_default_labels();

    register_post_type(
        'post',
        array(
            'labels'               => array(
                'name_admin_bar' => _x( 'Post', 'add new from admin bar' ),
            ),
            'public'               => true,
            '_builtin'             => true, /* internal use only. don't use this when registering your own post type. */
            '_edit_link'           => 'post.php?post=%d', /* internal use only. don't use this when registering your own post type. */
            'capability_type'      => 'post',
            'map_meta_cap'         => true,
            'menu_position'        => 5,
            'menu_icon'            => 'dashicons-admin-post',
            'hierarchical'         => false,
            'rewrite'              => false,
            'query_var'            => false,
            'delete_with_user'     => true,
            'supports'             => array( 'title', 'editor', 'author', 'thumbnail', 'excerpt', 'trackbacks', 'custom-fields', 'comments'
            'show_in_rest'         => true,
            'rest_base'            => 'posts',
            'rest_controller_class' => 'WP_REST_Posts_Controller',
        )
    );

    register_post_type(
        'page',
        array(
```

Kuva 5.15: WordPress *core* rajapinta

-rajapinnat, jotka on ensisijaisesti tarkoittettu helpottamaan uusien ominaisuuksien lisäämistä modulaarisesti. Nämä sisältävät toiminnallisuutta esimerkiksi plugin lisäosien tukemista varten, mutta tämän tutktimuksen näkökulmasta nämä ovat duplikoitu tietomallin kuvaus. Tämän rajapinnan sisältämä määritelmä näkyy kuvan 5.15 tiedostossa wp-includes/post.php.

Käyttöliittymäkoodissa käytössä olevien luokkien yhteydet tietokannan tietomalliin on erikseen määritelty. Käyttöliittymäkoodin ja palvelimen väliset interaktiot hoituvat *Ajax* -funktioilla, jotka mahdollistavat tässä projektissa uuden sisällön lataamisen verkkoympäristössä ilman verkkosivun päivittämistä. Funktiot käyttävät koodissaan sekä samoja luokkia kuin kuvassa 5.14, että myös omia funktioissa määriteltyjä käyttöliittymään perustuvia olioita. Esimerkki *Ajax* -funktiosta näkyy kuvan 5.16 tiedostossa wp-admin/includes/ajax-actions.php.

```php
/**
 * Ajax handler for Quick Edit saving a post from a list table.
 *
 * @since 3.1.0
 *
 * @global string $mode List table view mode.
 */
function wp_ajax_inline_save() {
    global $mode;

    check_ajax_referer( 'inlineeditnonce', '_inline_edit' );

    if ( ! isset( $_POST['post_ID'] ) || ! (int) $_POST['post_ID'] ) {
        wp_die();
    }

    $post_id = (int) $_POST['post_ID'];

    if ( 'page' === $_POST['post_type'] ) {
        if ( ! current_user_can( 'edit_page', $post_id ) ) {
            wp_die( __( 'Sorry, you are not allowed to edit this page.' ) );
        }
    } else {
        if ( ! current_user_can( 'edit_post', $post_id ) ) {
            wp_die( __( 'Sorry, you are not allowed to edit this post.' ) );
        }
    }

    $last = wp_check_post_lock( $post_id );
    if ( $last ) {
        $last_user      = get_userdata( $last );
        $last_user_name = $last_user ? $last_user->display_name : __( 'Someone' );

        /* translators: %s: User's display name. */
        $msg_template = __( 'Saving is disabled: %s is currently editing this post.' );

        if ( 'page' === $_POST['post_type'] ) {
            /* translators: %s: User's display name. */
            $msg_template = __( 'Saving is disabled: %s is currently editing this page.' );
        }

        printf( $msg_template, esc_html( $last_user_name ) );
        wp_die();
    }

    $data = &$_POST;
```

Kuva 5.16: WordPress käyttöliittymän rajapinta

Taulukko 5.9: WordPress moduulit tietomallia muuttaessa

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Koko malli määritelty yhdessä tiedostossa.	schema.php
API endpoint	Kentät hakutulosten lajittelua varten määriteltävä manuualisesti.	rest-api/endpoints/
API route	Erilliset rajapintatiedostot tietomallin kentille.	wp-includes/
luokka	Kaikki kentät määriteltävä manuualisesti.	wp-includes/
käyttöliittymä	Käyttöliittymän toimintojen funktiot, joissa määritellään omat oliot	ajax-actions.php

Taulukko 5.10: WordPress moduulit tietomalliin lisättäessä

moduuli	*kuvaus*	*sijainti*
tietokantamalli	MySQL. Manuaalinen migraatio.	schema.php
API endpoint	Kentät hakutulosten lajittelua varten määriteltävä manuualisesti.	rest-api/endpoints/
API route	Erilliset rajapintatiedostot tietomallin kentille.	wp-includes/
luokka	Kaikki kentät määriteltävä manuualisesti.	wp-includes/

5.9 Tulokset

5.9.1 Vaatimusten muutosten jalanjälki

Alla olevassta taulukosta 5.11 näkyy tiivistettynä tarkastelluista projekteista löytyneiden tietomallin kuvausten määrät. Sarakkeen *kuvaukset* arvo perustuu taulukkojen ja rivien määriin. Sarakkeen *tietomallin kuvaukset* data perustuu samojen edellä mainittujen taulukkojen rivien nimiin. *Taulu* sarakkeen kohteet nimellä *tutkimus* on luotu uusina tauluina, mikä on merkitty taulukossa asteriskeilla.

Taulukko 5.11: Tulokset 1/2

Projekti	*Taulu*	*Kuvauksia*	*Tietomallin kuvaukset*
Mastodon	users	3	tietokantamalli, ORM, REST rajapinta
Mastodon	*tutkimus	2	tietokantamalli, ORM, REST rajapinta
PrestaShop	products	4	tietokantamalli, ORM, REST rajapinta, luokat
PrestaShop	*tutkimus	4	tietokantamalli, ORM, query rajapinta, luokat
Laravel.io	articles	4	tietokantamalli, ORM, REST rajapinta, käyttöliittymä
Laravel.io	*tutkimus	2	tietokantamalli, ORM
WordPress	posts	5	tietokantamalli, query rajapinta, REST rajapinta, luokat, käyttöliittymä
WordPress	*tutkimus	4	tietokantamalli, ORM, rajapinnan reitit, luokat

Pelkkä muutoksen vaikutuksen alaisten moduulien määrä ei kuitenkaan vielä

yksin täysin riitä mittaamaan jalanjälkeä lähdekoodissa. Myöskään muuttuneiden koodirivien määrä ei riitä, varsinkin, koska tähän tutkimukseen valittu kompleksisuudeltaan hyvin kevyt tietomallin muutos sopii paremmin muutoksen propagoitumisen jäljittämiseen moduulista toiseen. Koodirivien määrät ovat näillä muutoksilla niin pienet, etteivät ne riitä tilastollisesti luotettavaksi otannaksi. Tämän vuoksi alla olevassa taulukossa 5.12 projekteille on annettu lisäksi karkeat velkakertoimet, joiden tarkoitus on pintapuolisesti arvioida teknisen velan kertymistä tilanteissa, joissa samoja teknologioita käyttävissä projekteissa tapahtuu suurempi tietomallin vaatimusten muutos. Velkakertoimen tekijät käydään läpi seuraavassa kohdassa 5.9.2. Taulukon 5.12 koodirivien määrän yhteenlaskut noudattavat samaa järjestystä kuin taulukon 5.12 *Tietomallin kuvaukset* sarakkeen listat moduuleista.

Lisäksi taulukkoon 5.12 on myös jälkikäteen syntetisoitu teoreettinen optimitilanne vertailua varten, jossa tietomallin muutoksen oletetaan sisältävän ainostaan taulukon 5.2 esimerkistä johdettavissa olevat tiedot. Tarkemmin sanottuna taulukon 5.2 esimerkin muuttuneet tiedot esitettynä sellaisenaan ilman mitään oletuksia ohjelmointikielestä, syntaktista tai lähdekoodin rakenteesta. Käytännössä tämä tarkoittaa seitsemää (7) datapistettä. Nämä ovat yksi datapiste taulun *Tutkimus nimelle*, kolme datapistettä uusien kenttien nimille ja yksi datapiste kentän *tutkimus_id* relaatiolle. Tämän lisäksi myös kentän *artikkeli_id* määritys taulua yksiöiväksi kentäksi sekä ohje lisätä taulu osaksi tietokantamallia laskettiin datapisteiksi, mikä perustuu muissa tarkastelluissa kohteissa havaittuihin käytäntöihin. Tämän oletetaan olevan pienin mahdollinen jalanjälki koodissa, minkä tässä tutkimuksessa simuloitu tietomallin muutos voi aiheuttaa. Arvo on tarkoituksella kaukana tutkimuskohteissa havainnoidusta muutoksien jalanjäljistä, koska se ei ota kantaa siihen, mikä on pienin "realistinen"mahdollinen jalanjälki.

Taulukko 5.12: Tulokset 2/2 (pienempi velkakertoimen arvo = vähemmän teknistä velkaa)

Projekti	*Koodirivit*	*Arviotu "velkakerroin"*
Mastodon	8 + 4 + 14 = 26	2.101∼2.703
PrestaShop	8 + 6 + 13 = 27	2.624∼3.249
Laravel.io	9 + 5 + 3 + 1 = 18	2.416∼2.833
WordPress	10 + 8 + 22 + 2 + 60 = 102	4.859∼7.218
Teorettinen optimi	7	0.662∼0.824

5.9.2 Vaikutus tekniseen velkaan ja velkakerroin

Tässä tutkimuksessa kohteiden tietomalleille tehdyt muutokset olivat erittäin pieniä. Tästä johtuen vaikutus tekniseen velkaan on johdettava tuloksista tekemällä oletuksia tilanteesta, jossa tietomalliin tehtävät muutokset ovat laajempia. Tämän pohjalta voidaan syntetisoida ikään kuin velkakerroin, joka arvioi teknisen velan kertymistä projektin tietomallin koon kasvaessa sekä teknisen velan ilmentymisen määrää tietomallin muutoksen laajuuden kasvaessa.

Kertauksena aiemmin kappaleessa läpi käydystä teknisen velan teoriasta, vaikuttavia tekijöitä ovat teknisen velan syntymisessä ovat duplikoitunut lähdekoodi sekä tarpeettoman pitkä lähdekoodi. Näiden velkaa aiheuttavien tekijöiden suhde ongelman aiheuttajana on jokin tuntematon luku, mutta koska teknisen velan teoriassa duplikoitunut koodi mielletään suuremmaksi, oletetaan duplikoituneen koodin painoarvon teknisen velan kehittymisessä olevan jokin yhtä (1) suurempi arvo. Jos taas valitaan painoarvolle jokin tarpeeksi korkea yläraja, kuten kaksi (2), saadaan aikaan vaihteluväli, jolle oikea todellisuutta vastaava painoarvo jää. Yksinkertaistettuna tämä tarkoittaa, että kerrotaan kohteen duplikoituneen koodin määrä yhdellä (1), ja muokatun koodin määrä jollakin arvolla, joka on yhden (1) ja puolikkaan (0.5) välillä. Tämän vaihteluvälin oletetaan sisältävän näiden tekijöiden vaikutuksen painoarvon, vaikka sitä ei tiedetä tarkasti.

Muun datan puutteessa oletetaan taulukon 5.12 koodirivien määrän olevan in-

dikaattori lähdekoodin määrän tarpeettomalle kasvulle sillä perustelulla, että ainakin yksi kohteista vaati tutkimuksessa pienemmän määrän muokattuja koodirivejä. Taulukon 5.11 tietomallin kuvausten määrät sen sijaan ilmaisevat duplikoitunutta koodia. Tietomallin kuvausten määrän keskiarvo oli neljä (4), joten jotta koodiriveille saadaan velkakertoimen ylärajan selvittämiseksi yhtä (1) vastaava painoarvo, on koodirivien keskiarvo jaettava luvulla, joka tuottaa saman keskiarvon kuin tietomallin kuvausten määrällä. Tässä tapauksessa tämä kerroin on 4 / 43.25 = ∼0.0925. Koodirivien painoarvoa voidaan sitten muuttaa jakamalla koodirivien määrät kahdella, jolloin saadaan aikaan velkakertoimen alaraja. Taulukon 5.12 velkakertoimet on laskettu tällä metodilla.

([koodirivit] × 0.0925 × (0.5∼1) + [tietomallien toteutusten määrä]) / 2

5.9.3 Vastaus tutkimuskysymykseen TK2

Tutkimuksen toinen tutkimuskysymys TK2 liittyi tietomallin kuvausten määriin tarkasteltaviksi valituissa projekteissa. Valittujen kohteiden altistaminen tietomallin muutokselle näytti, että tarkastellut projektit sisälsivät keskimäärin neljä (4) kattavaa tietomallin kuvausta, mikä laskee 3,5:een, kun otetaan huomioon tilanteet, joissa tietomalliin lisättiin vain uusia elementtejä jo olemassa olevien elementtien muuttamisen sijaan. Lähdekoodin alueet, tai moduulit, jossa tietomallin kuvauksia ilmeni, olivat kyseisestä sovelluksesta riippuen tietokantamalli, ORM (Object Relational Mapper), verkko-osoitteiden rajapinnat, REST -hakurajapinnat, plug-in rajapinta, luokka- ja oliorakenne tai käyttöliittymän koodi.

6 Yhteenveto

Tämä kappale kokoaa yhteen tutkimuksessa tuotetut vastaukset tutkimuskysymyksiin. Lisäksi myös tutkimuksen merkitystä pohditaan aiheen kontekstissa.

6.1 Tulokset

Lyhyenä kertauksena, tutkimuksen tavoitteena oli kartoittaa laajasti käytössä olevat verkkosovellusten teknologiat ja analysoida tämän jälkeen näitä teknologioita edustavien sovellusten lähdekoodien sisältämiä kattavia tietomallin kuvauksia. Toisin sanoen tarkoituksena oli kartoittaa, toistetaanko samaa tietomallia mahdollisesti turhaan useassa kohtaa lähdekoodia.

6.1.1 Suositut teknologiat ja tutkimuskysymys TK1

Ensimmäisen tutkimuskysymys liittyi laajasti käytössä oleviin verkkosovellusten teknologioihin. Tämän selvittämiseksi suoritettu tutkimus tuotti tulokset, jotka vastaavat odotuksia. Suosituiksi ohjelmointikieliksi osoittautuivat JavaScript, Java, TypeScript ja PHP. Rajapinta- ja ORM -ratkaisut osoittautuivat pitkälti yksinkertaisiksi REST rajapinnoiksi. Tulosten pohjalta pystyttiin myös syntetisoimaan alla olevat esimerkit teknologiapinoista jokaista tutkimuksessa johdettua sovelluskategoriaa varten.

- **Staattinen:** JavaScript - next.js - TypeScript

- **Dynaaminen:** JavaScript - Mongo DB - angular.js - JavaScript
- **Verkkokauppa:** JavaScript - MySQL - SpringBoot - Java
- **Portaali:** JavaScript - MySQL - Vue - PHP
- **Sisällönhallinta:** JavaScript - MySQL - WordPress - PHP

Muuta huomioitavaa on esimerkiksi SQL -tietokantojen suosio NoSQL -tietokantojen sijaan. Myös suositut verkkosovellusten viitekehykset, kuten esimerkiksi Spring, Django, Laravel sekä Ruby on Rails näkyivät tuloksissa. Yleisesti tulokset olivat painottuneita juuri tämän kaltaisiin aloituspisteiksi tarkoitettuihin viitekehyksiin, eikä valmiisiin verkkosovelluksiin. Tämä ei ole yllättävää, koska suosion kriteerinä käytetty GIT -haarautumisten määrä painottaa projekteja, joista on helppo uuden haaran haarukoinnin jälkeen luoda omat tarpeet täyttävä verkkosovellus.

6.1.2 Tietomallin muutosten propagoituminen ja tutkimuskysymys TK2

Teoriaosiossa käytiin läpi verkkosovellusten kehityksessä hallittavia asioita, mutta erityisesti teknistä velkaa. Lähdekoodin määrän kasvamisen ja yksittäisten muutoksien vaikutuksen propagoituminen lähdekoodin muihin osiin todettiin kummatkin läpi käydyn materiaalin perusteella osasyiksi teknisen velan muodostumiselle. Tästä heräsi kysymys, että tuottavatko tietomallien muutokset moderneissa verkkosovelluksissa juuri näitä ilmiöitä, kun otetaan huomioon näiden sovellusten kerrostunut rakenne ja näiden kerrosten tarve vaikuttaa toistensa kautta saamaan tietomalliin. Toisin sanoen, onko duplikoitunut tietomallia kuvaava koodi rakenteellinen ongelma verkkosovelluksille? Kyseisen asian kartoittamista varten vastattiin tutkimuskysymykseen TK2.

Tutkimuksen toinen tutkimuskysymys TK2 liittyi tietomallin kuvausten määriin tarkasteltaviksi valituissa projekteissa. Projekteissa käyttötarkoituksella ei ollut

tutkimuksen kannalta suurta merkitystä, vaan analyysin kohteena olivat käytössä olevat teknologiat ja näiden asettamat rakenteelliset rajoitteet. Analysoitavaksi valitut kohteet olivat Mastodon, PrestaShop, Laravel.io ja WordPress. Valinta perustui tutkimuskysymykseen TK1 saatuun vastaukseen.

Tutkimuksessa havaittiin, että tarkastellut kohteet sisälsivät keskimäärin neljä (4) kattavaa tietomallin kuvausta, mikä laskee 3,5:een, kun otetaan huomioon tilanteet, joissa tietomalliin lisättiin vain uusia elementtejä jo olemassa olevien elementtien muuttamisen sijaan. Lähdekoodin alueet, tai moduulit, jossa tietomallin kuvauksia ilmeni, olivat kyseisestä sovelluksesta riippuen tietokantamalli, ORM (Object Relational Mapper), verkko-osoitteiden rajapinnat, REST -hakurajapinnat, plug-in rajapinta, luokka- ja oliorakenne tai käyttöliittymän koodi. Projekteille syntetisoitiin myös hyvin teoreettiset velkakertoimet, joiden on tarkoitus arvioida vauhtia, jolla tietomallin laajuus kerryttää teknistä velkaa ja vauhtia, jolla tietomallin muutoksen laajuus lisää teknisen velan ilmentymistä.

Yleisesti ainakin kyseisistä tarkasteltavista kohteista tuli yleinen vaikutelma, että tässä tutkimuksessa tarkastelun aiheena ollut tietomallien duplikoituminen on vähenevä ongelma. Uudemmat verkkosovellusteknologiat tarjoavat parempia työkaluja samojen kerran määriteltyjen luokkien käyttämiselle tietomallina sovelluksen kaikissa osissa. Esimerkiksi Ruby on Rails -viitekehyksen ”malli”-luokat riittävät rajapinnaksi tietokannan ja koko muun sovelluksen välillä, vähentäen tarvetta muille mahdollisesti ylimääräisille tietomallin kuvauksille. Tutkimuksen tuloksissa tämän näkyi siinä, että tietomallin muutokset tuottivat kyseisessä kohteessa vaikuttivat kaikkein pienimpään määrään moduuleja, koska palvelimen ja tietokannan tai palvelimen ja käyttöliittymän välille ei tarvittu erillistä rajapintaa, vaan interaktio tietokannan kanssa tapahtui suoraan Ruby -luokkien kautta. Tosin tämä riippuu, kuinka laajasti tällaiset käytännöt halutaan ottaa käyttöön.

6.2 Jatkossa

Seuraava mahdollinen toimenpide tutkimuksen aihealueella voisi hyvin olla jonkinlaisen ratkaisun etsiminen tietomallin kuvausten kopioitumiselle. Se mitä tarvittaisiin, olisi jonkinlainen tietomallien unioni, eli kaikkien relevanttien tietomallien kuvausten superset, josta voisi johtaa tietomallin ilmentymisestä seuraavat toteutukset eri tarpeisiin tai tilanteisiin, sisältäen kunkin toteutustason tietomallien määritteet sekä mahdolliset järjestelmän omat lisätarpeet. Geneerinen tietomalli on toki haastava tavoite.

Toinen mahdollinen tutkittava alue voisi olla jo käyttöönotettujen verkkosovellusten laajan määrän tietoa sisältävien tietokantojen migraatio uuteen tietomalliin. Laajasti käyttöönotetun ja suuret määrät siirrettävää dataa sisältävän tietokannan päivittäminen on aihealue, joka on jokseenkin rinnastettavissa tietomallin muutoksiin lähdekoodissa.

Lisäksi tämän tutkimuksen otanta oli melko suppea, joten pelkästään tutkimuksen metodien hiominen ja toistaminen suuremmalle määrälle tarkasteltavia projekteja on mahdollinen keino jatkaa tutkimustyötä tällä aihealueella. Tällainen laajempi tutkimus edellyttäisi myös parempaa tutkimusmetodien automatisointia.

Lähdeluettelo

[1] K. Beck, M. Beedle, A. van Bennekum et al., *Manifesto for Agile Software Development*, 2001. url: http://www.agilemanifesto.org/.

[2] Z. Liu ja B. Gupta, "Study of Secured Full-Stack Web Development.", teoksessa *CATA*, 2019, s. 317–324.

[3] A. Leff ja J. T. Rayfield, "Web-application development using the model/-view/controller design pattern", teoksessa *Proceedings fifth ieee international enterprise distributed object computing conference*, IEEE, 2001, s. 118–127.

[4] C. Northwood, *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer*. Springer, 2018.

[5] N. S. Smith, "Spatial data models and data structures", *Computer-Aided Design*, vol. 22, nro 3, s. 184–190, 1990, ISSN: 0010-4485. DOI: https://doi.org/10.1016/0010-4485(90)90077-P. url: https://www.sciencedirect.com/science/article/pii/001044859090077P.

[6] J. Hickey, *Building a Scalable E-Commerce Data Model*, en, joulukuu 2020. url: https://fabric.inc/blog/ecommerce-data-model/ (viitattu 26. 04. 2023).

[7] T. Ruokolainen, "Arkkitehtuurinen reflektio",

[8] A. Watt ja N. Eng, *Database design*. BCcampus, 2014.

[9] M. Chapple, "Sql fundamentals", *Databases*, 2009.

[10] G. Brito ja M. Valente, ”REST vs GraphQL: A Controlled Experiment”, helmikuu 2020. DOI: 10.1109/ICSA47634.2020.00016.

[11] Ł. Kamiński, M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski ja R. Roszczyk, *Comparative review of selected Internet communication protocols*, 2022. DOI: 10.48550/ARXIV.2212.07475. url: https://arxiv.org/abs/2212.07475.

[12] A. Aljohani, *An empirical study on discovering a new self-admitted technical debt type-API-debt*. Rochester Institute of Technology, 2019.

[13] J. H. Tomi‘bgt’Suovuo, J. Smed ja V. Leppänen, ”Mining knowledge on technical debt propagation”, teoksessa *14th Symposium on Programming Languages and Software Tools*, vol. 1525, 2015, s. 281–295.

[14] A. Potdar ja E. Shihab, ”An exploratory study on self-admitted technical debt”, teoksessa *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, s. 91–100.

[15] D. Spinellis, *Code quality: the open source perspective*. Adobe Press, 2006.

[16] R. Wang, R. Huang, B. Qu et al., ”Network-based analysis of software change propagation”, *The Scientific World Journal*, vol. 2014, 2014.

[17] C. Wohlin ja A. Aurum, ”Towards a Decision-Making Structure for Selecting a Research Design in Empirical Software Engineering”, *Empirical Softw. Engg.*, vol. 20, nro 6, s. 1427–1455, joulukuu 2015, ISSN: 1382-3256. DOI: 10.1007/s10664-014-9319-7. url: https://doi.org/10.1007/s10664-014-9319-7.

[18] *What is Full Stack*, en-US. url: https://www.w3schools.com/whatis/whatis_fullstack.asp (viitattu 10. 05. 2023).

[19] E. K. Huizingh, ”The content and design of web sites: an empirical study”, *Information & management*, vol. 37, nro 3, s. 123–134, 2000.

[20] D. Barker, *Web content management: Systems, features, and best practices*. "O’Reilly Media, Inc.", 2016.

[21] *Clean Code Tools for Writing Clear, Readable & Understandable Secure Quality Code*, en. url: `https://www.sonarsource.com/` (viitattu 15. 06. 2023).

[22] *OpenTelemetry*, en. url: `https://opentelemetry.io/` (viitattu 19. 06. 2023).

Liite A Liitedokumentti

Liitteen ohjelmakoodi 1 kuvaa matemaattisen monadirakenteen pohjalta rakentuvan Haskellin tyyppiluokan. Tyyppiluokan voi nähdä eräänlaisena abstraktina ohjelmointirajapintana (API), joka muodostaa ohjelmoijalle abstraktin ohjelmointikielen käyttöliittymän (UI).

Ohjelmalistaus 1 Tyyppiluokka 'Monad'.

```
{haskell}

class Monad m where

    ( >>= )         :: m a -> (a -> m b) -> m b

    return          :: a                 -> m a


    fail            :: String            -> m a

    (>>)            :: m a -> m b        -> m b

    m >> k          =  m >>= \_ -> k       -- default


instance Monad IO where  ...               -- omitted
```

Ensimmäisen liitteen toinen sivu. Ohjelmalistaus 2 demonstroi vielä monadin käyttöä.

Ohjelmalistaus 2 Monadin käyttöä.

```
{haskell}

main =

return "Your name:" >>=

putStr >>=

\_ -> getLine >>=

\n -> putStrLn ("Hey " ++ n)
```

Liite B Liitedokumentti 2

Tässä esimerkki

toisesta kaksisivuisesta liitteestä.