Full stack -verkkosovelluskehitys modernilla ja harkitulla arkkitehtuurilla

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK)

Kevät 2023

Patrik Kumpula

Tieto- ja viestintätekniikka Tiivistelmä
Tekijä Patrik Kumpula Vuosi 2022
Työn nimi Full stack -verkkosovelluskehitys modernilla ja harkitulla arkkitehtuurilla
Ohjaaja Petri Kuittinen

Työssä esitellään IT-alan verkkosovellusten kehittämistä sekä teorian että käytännön osalta. Teoriaosassa myös käydään läpi IT-ammattilaisten työympäristöjä tutkimalla dataa, joka kertoo tekijöiden suosimista sekä oikeasti käyttämistä teknologioista ja teknologiapaketeista (engl. Tech stack). Tämän opinnäytetyön taustatarina oli projekti, avulla esiteltiin mahdollista modernisointitapaa 2000-luvun alussa alkujaan kehitetylle todelliselle tuotannossa laajasti käytetylle sovellukselle, jonka vanhat juuret aiheuttavat hitautta ja haasteita sovelluksen jatkokehitykselle. Projektissa on tarkoitus tuoda esille erilaisia metodeja, jonka avulla voidaan rakentaa moderni arkkitehtuuri, ja sen hyötyjä perustellaan kvantitatiivisesti kokeella, jossa verrataan molempia järjestelmiä mittaamalla ajassa, että kuinka kauan kummassakin kestäisi konfiguroida 'prosesseja', järjestelmälle keskeisiä moduuleja.

Teoreettisen sekä sovelluksen kehittämistä esittävän osan on tarkoitus kuvata lukijalle millä tekniikoilla modernin verkkosovelluksen voisi rakentaa. Esittelyssä ei tuoda pelkästään ilmi projektissa käytetyt kehykset ja teknologiat, mutta sen lisäksi tuodaan esille muita vakiintuneita vaihtoehtoja, mutta myös potentiaalisesti tulevaisuudessa enemmän käytettyjä ratkaisuja. Teoria esitellään kuvatulla tavalla sen vuoksi, että määrällisiä merkittäviä tekijöitä voi olla vaikea perustella, jonka vuoksi tilastoihin tukeudutaan harvoin.

Lopputuloksena on työn ohella tehty projekti, jota on esitelty vanhan tuotteen seuraajana muutamassa eri demotilaisuudessa. Demoprojekti on ollut tuloksiltaan vaikuttava, ja sitä harkitaan pohjana pienemmälle projektille, jossa rakennetta pystytään ottamaan tuotantoon.

Avainsanat verkkosovellukset, ohjelmistoarkkitehtuuri, verkko-ohjelmointi
Sivut 87 sivua ja liitteitä 1 sivu

Information and Communication Technology Abstract
Author Patrik Kumpula Year 2022
Subject Full stack web application development using modern and well deliberated architecture
Supervisor Petri Kuittinen

This work presents web application development partially using both theoretical and pragmatic mindset, while also presenting data of modern IT-professionals' work environments by examining their most preferred and used technologies. This thesis is based on a project, which presents a possible way to modernize an application originally developed in the beginning the 21st century. Its old techniques and technologies are among the reasons why the system is comparatively more difficult to develop further. The project is meant to exhibit ways to build modernized web application architecture. Its benefits are justified using a quantitative experiment, where time is measured in both systems for manually configuring 'processes', the central modules of this system.

The theoretical and application development part of this thesis exhibits to the user how an application could be developed with modern methods. The frameworks and technologies in the project are not the only methods presented in this work, but also other established modern technologies are given as examples. The examples even include frameworks that might be potentially more generally used in the future are touched upon. It is difficult to simply quantify meaningful differences, which is why the theoretical part does not rely on statistics.

The result is a real-world business centered project that has been demonstrated as a proof of concept in a few team meetings. The given approach has been reviewed as a possible future and it is being, at the time of writing, considered as a base for a project where the technology stack could be tested in the field using a small production environment.

Keywords Web applications, software architecture, web-development
Pages 87 pages and appendices 1 page

Sisällys

1 Johdanto 1
2 Sovellusarkkitehtuurin merkitys 3
3 Verkkokehityksen termit 5
4 Verkkokehityksen konseptit 8
4.1 Front-end 12
4.2 Front-end esimerkkinä 15
4.3 Front-end kehityksen teknologiavalinnat 17
4.3.1 React 19
4.3.2 Angular 23
4.3.3 Vue 24
4.4 Ohjelmistokehysten hyöty kehittäjän näkökulmasta 24
4.5 Back-end ja API 26
4.6 Back-endin ohjelmointikieli ja alusta 28
4.7 Back-end rakenne pintaliitona 30
5 Projekti 33
5.1 Vanha järjestelmä 33
5.2 Uusi back-end konsepti 35
5.3 Uusi front-end konsepti 40
5.4 Uuden front-endin kehitys 40
5.5 Näkymän komponentit 44
5.6 Front-end -sovelluksen toiminta käyttäjän näkökulmasta 51
5.7 Uuden back-endin rakenteen yleiskatsaus 54
5.7.1 API-kirjasto 59
5.7.2 Core-kirjasto 61
5.7.3 Infrastructure-kirjasto 65
6 Tuotekohtaisia parannuksia esiteltynä 66
7 Vertailukoe 72
8 Yhteenveto 74
Lähteet 78

Liitteet

Liite 1 Selitteet

1 Johdanto

Tässä opinnäytetyössä tutkitaan ja havainnollistetaan verkkosovellusten kehittämistä esittelemällä alan termejä, työkaluja ja hyväksi havaittuja työskentelytapoja ja sovelluskehitykseen liittyviä paradigmoja. Tämän avuksi osa tästä opinnäytetyöstä on omistettu esiteltäviin aiheisiin liittyvällä verkkosovelluskehityksen työelämäprojektilla. Projektin avulla esitellään kehityksen työvaiheita järjestelmän osia tehdessä (front-end ja back-end -kehitys) ja perusteellaan projektin sovellusarkkitehtuurin toimivuutta. Teoriassa teknologiapaketteja on hyvin vaikea perustella kvantitatiivisesti toista paremmaksi, etenkin kun esimerkiksi nopeuserot ovat jopa virhemarginaalin kokoisia. Tämän vuoksi teoriaosio ei keskity niinkään tilastolliseen perusteluun, mutta pyrkii esittämään laadulliset sekä mielipiteeseen perustuvat tekijät tärkeämpinä.

Todelliset merkittävät tekijät sovelluksen kehityksessä perustuvat enemmän laadullisiin tekijöihin, joita ovat muun muassa seuraavat:

- sovelluskehitystiimin kokoonpano
- kehityksen sujuvuus valituilla työkaluilla valittuun tarpeeseen
- yksittäisten kehittäjien mieltymykset ja kokemus
- sovelluksen ulkoiset riippuvuudet
- iteroituvaan kehitykseen vaikuttavat tekijät
- logistiset tekijät, kuten lisenssimaksut, aikataulu ja koulutus.

Tämän vuoksi pragmaattisuuteen perustuva ajattelu käyttäen perustellusti järkeiltyä arkkitehtuuripohjaa on merkittävän tärkeä lähestymistapa. Reaalimaailmassa sovelluksen projektisuunnitelmassa on paljon olosuhteista johtuvia tekijöitä, jonka vuoksi ratkaisevaa vastausta jokaisen projektin haasteisiin on vaikea vastata yksinkertaisella tavalla.

Projektin sovellusarkkitehtuurille on asetettu tässä opinnäytetyössä hyvin suuri rooli. Sen tarkoituksena on havainnollistaa tarkasti harkittujen sovellusarkkitehtuuriin liittyvien päätösten positiivisia vaikutuksia. Tarkoituksena on perustella miksi suunnitelmallisuus kannattaa, lainaten perusteltuja ja luotettavia tiedonlähteitä, kuten esimerkiksi IT-alan

kokeneita ammattilaisia. Projektissa esitelty arkkitehtuurillinen malli tulee ”Uncle Bob” Robert Martininin kirjallisesta tuotannosta. **Clean Architecture** on peräisin kyseisen kirjailijan kirjasta “Clean Architecture: A Craftsman’s Guide to Software Structure and Design” (2017). Tavoitteena on kuitenkin pitää aina tasapainotella kirjan mukaista dogmaattisuutta sekä reaalimaailman pragmaattisuutta. Sovellusarkkitehtuuria suunnitellessa ei kannata aina seurata tietolähteiden tai yleisen diskurssin mielipiteitä poikkeamatta sen esittelemästä normista. On tärkeämpää kehittää omaa käyttötarkoitusta palveleva sovellusarkkitehtuuri. Sen lisäksi ohjelmointiin liittyvät kirjat ovat nopeasti vanhentuvaa materiaalia, jonka vuoksi painettua kirjaa ei voi pitää yhtä luotettavana materiaalina kuin verkkolähteet.

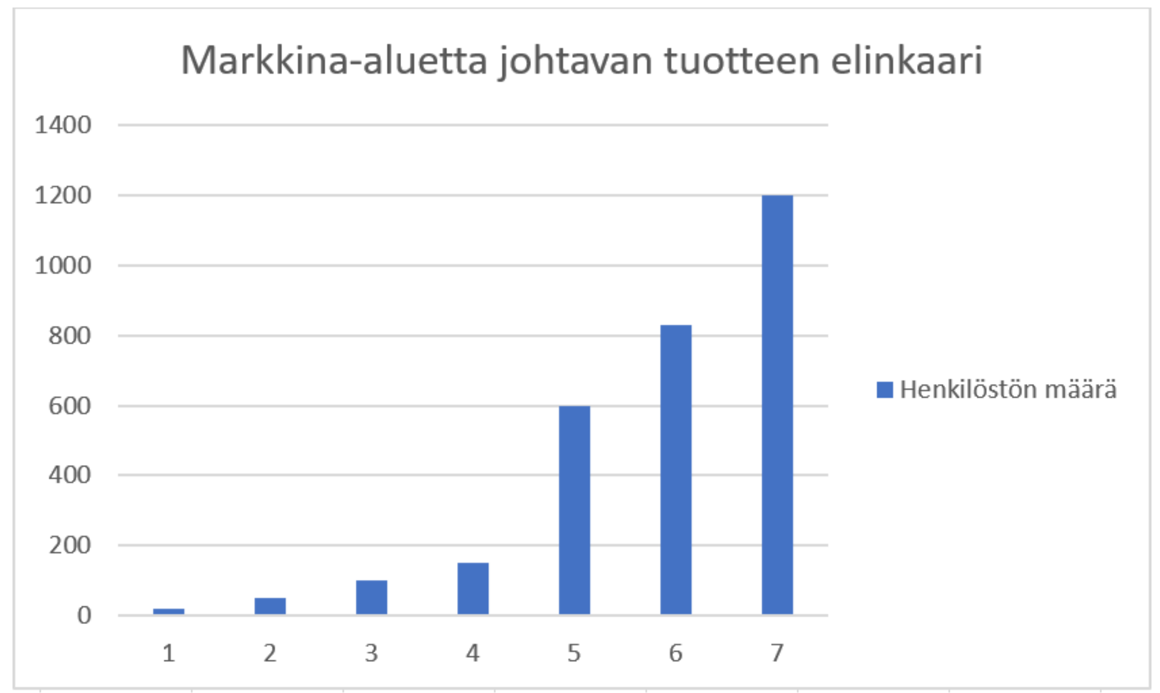
Lähteet ovat tässä apuvälineitä mutta niihin täytyy suhtautua kriittisesti oman projektin kannalta. Lähteiden esittämiä ideoita ei ole tehty juuri jotakin tiettyä projektia varten, mutta niistä voi poimia omaa projektia hyödyttäviä tekniikoita. Tekniikkaa on vaikea vakiinnuttaa, sillä sen trendit saattavat ajoittain muuttua jopa joka vuosi. Tämä ei tarkoita sitä, että sovellusarkkitehtuuriin konseptina kannattaa suhtautua kriittisesti. Hyvä arkkitehtuuri on välttämätöntä hyvälle sovellusprojektille. Sovellusarkkitehtuurin korkean tason yksityiskohdat tukevat sovellusten alemman tason yksityiskohtia, eli implementaation logiikkaa. Se arkkitehtuuri määrittelee yhteisen järjestelmän. Ilman arkkitehtuuria toiminnan logiikka ei voi toimia ja toisinpäin. Sovellusarkkitehtuuri ja varsinainen ohjelmakoodi on yhteinen looginen ketju, jota ei pysty toisistaan irrottaa.

Tavoitteita on seuraavanlaisia: Lukija ymmärtää millaista ajattelua vaaditaan isolta sovellusprojektin teolta niin, että ohjelmakoodia olisi jatkokehityksen tarpeessa helppo tuottaa uusia ominaisuuksia sekä ylläpitää vanhaa. Lukija oivaltaa eron ja funktion eri verkkosovelluksen osilla. Sovellusprojektin osalta on tarkoituksena esitellä viimeisteltyä projektia, sekä arvioida miten työelämästä projektia seuranneet näkevät projektin ja sen lopputuloksen. Tämä opinnäytetyö on suunnattu sellaisille lukijoille, jotka ovat kohtuullisen päteviä opinnäytetyössä mainituissa ohjelmointikielissä, ja ovat kiinnostuneet sovellusprojektien kehittämisestä teknisessä ja arkkitehtuurillisessa mielessä. Liittyvä alan puhekieli ja työkalut esitellään lyhyesti niitä käsittelevissä kappaleissa

2 Sovellusarkkitehtuurin merkitys

Kuvassa 1 on esimerkki pilalle menneestä nimettömästä, hyvin suuresta sovellusprojektista. (Martin, 2017) Siinä esitellään nimiä mainitsematta erään sovellusyrityksen insinöörityöryhmän kasvu tuotteen elinkaaren aikana. Yksittäinen pylväs vastaa isoa julkaisuversioita. Pinnallisesti tilanne voi vaikuttaa hyvältä, jopa sellaiselta että kyseisen ryhmän projekti oli hyvin kannattava, mutta peilaako todellisuus tilastoja?

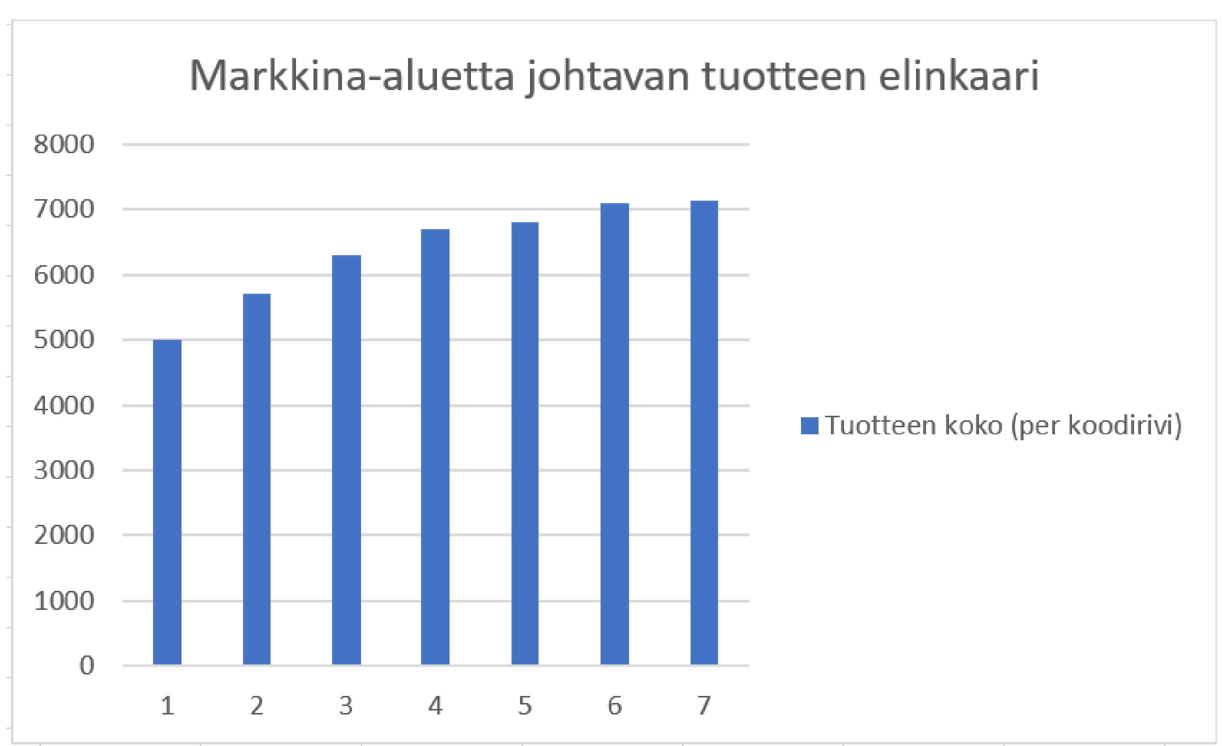
Kuva 1 Henkilöstön määrä projektissa per iso julkaisunumero. Tuotettu silmämääräisenä kopiona. (Martin, 2017, s. 5)



Katsotaan samaa tilanne kuvan 2 taulukon avulla. Tuottavuuden kannalta projekti ei taas vaikuta hyvältä. Miksi henkilöstön määrän moninkertaistuessa projektin kasvu hidastui? Tämä ei voi näyttää hyvältä yrityksen tuottaman katteen kannalta. Miltä vaikuttaisi tilasto, jossa tuotteen hintaa per yksi uusi koodirivi esitellään, verraten ensimmäisiin versioihin? Syy

tälle tapahtumien kululle on yksinkertaisesti siinä, että lopullisesta koodikannasta muodostui suuri sotku. Jos sovellusprojektissa ei pohdita merkityksellisessä mielessä sovelluksen arkkitehtuuria, se väistämättömästi törmää samankaltaiseen tilanteeseen.

Kuva 2 Tuotteen koko koodirivien perusteella. Tuotettu silmämääräisenä kopiona. (Martin, 2017, ss. 6-7)



Koodin siisteys on tärkeää, jonka vuoksi koodin suunnittelu on myös tärkeää. Vaikka siististi tehty koodi aluksi vaikuttaa hitaalta tekniikalta, niin se lopulta ohittaa tuottavuudessa huonosti suunnitellun koodikannan. Todellisuudessa ”nopeasti kirjoitettu” sovellus pystyy ainoastaan hidastaa kehittäjien työskentelyä, mutta nopeasti tavoitteeseen pääsy on kehittäjissä inhimillinen piirre; markkinat eivät odota kehittäjää, ja ne liikkuvat jatkuvasti. Sen vuoksi moni tekijä pyrkii kiirehtimään, jonka lopputuloksena on ryvettyvää koodia.

Hyvän kehittäjän voi tunnistaa siitä, että häneltä löytyy tarpeeksi itsekunnioitusta välttää sellaiset päätökset, jossa lähdekoodin laatua pyritään vähentämään esimerkiksi kiireestä johtuvista tekijöistä. Periaate on se, että ainoa tapa kehittää nopeasti on kehittää harkiten.

(Martin, 2017) Hyvä kehittäjä välttää ryvettyvän koodiin niin, että hän ottaa koodin laadun tosissaan. Koodikannan rakenteen suunnittelu on varsinaisen tärkeää. Hyvin suunnitellun järjestelmän etu on siinä, että se minimoi vaaditun tekemisen (Voidaan vähentää kuluja ja käytettyä aikaa) ja maksimoi tehokkuuden (Voidaan parantaa sovelluksen laatua ja helpottaa sen ylläpitoa).

Hyvä arkkitehti tunnistaa millä keinoilla hyvä sovellusarkkitehtuuri saavutetaan. Hyvä arkkitehti tunnistaa esitellyistä keinoista sen, mistä ei ole omassa tapauksessa mitään apua. Sovellusarkkitehtuuriin piirretyt rajat voivat olla tehokkaita vain silloin, kun niitä käytetään niille sopivissa tilanteissa. Todellisuudessahan sovellusarkkitehtuurien esittämät ideat ovat monesti tyhjästä keksittyjä pedanttisia sääntöjä. Hyvä ohjelmakoodi ei vaadi sitä, että koodikanta on jonkun tietyn sovellusarkkitehtuurimallin mukainen. Näin ollen en pyri tällä opinnäytetyöllä antamaan vastausta jokaisen tarpeisiin, mutta esitän keinoja ja ajatuksia hyvän arkkitehtuurin saavuttamiseen.

3 Verkkokehityksen termit

Verkkokehitys on sisäisesti hyvin laaja konsepti tietotekniikan alalla, ja eri työtehtäviä kuvaa moni eri termiä, mutta tunnetuimpien kirjoon kuuluu englanniksi termejä kuten ”Full stack Developer” ja ”Front-end Developer”. ”Full stack Developer” terminä ei ole aina ollut tuli

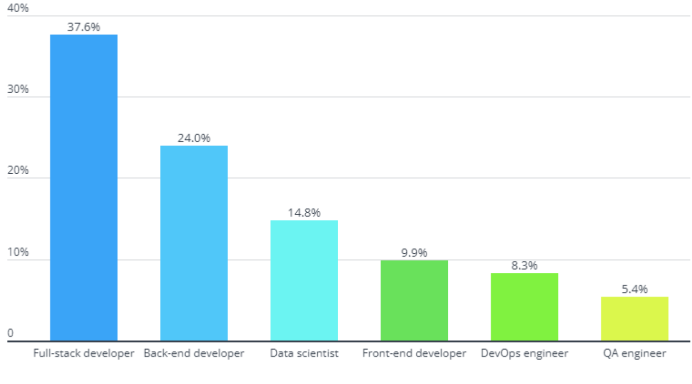
esille terminä vasta vuonna 2008. Voidaan havainnollistaa termin kehitystä käyttämällä Kuvan 3 tehtyä Google Trends tutkimusta.

Kuva 3 Vertailuna termit "Software Developer" ja "Full-stack developer"

Termiä ”Full stack developer” voidaan kuvata turhana semantiikkana, mutta se on varmaa, että ”Full stack developer” esiintyy terminä helposti teknologialan työpaikkailmoituksissa ja koulutuksissa sekä yleisessä diskurssissa, sillä nämä termit helpottavat keskustelua alan monimutkaisista konsepteista. Todetaan nämä termit vakiintuneiksi ja näitä termejä käytetään tässä työssä viittaamaan verkkosovelluskehitykseen ja kehittäjiin. Kuvassa 4 on

tilastossa käytetty ”Full-stack developer” termiä, ja siitä samoin näkee kyseisen konseptin suosion.

Kuva 4 HackerRank 2020 Raportista otettu kuvankaappaus (Frederick, n.d.)



Ohjelmistoteknologian edetessä 2000- ja 2010 luvuille ohjelmointikielien ja ohjelmointikehysten abstraktio ja virtaviivaistaminen on sallinut nykypäivän teknologiarakenteet, jossa kehittäjän on olennaista tietää millä tekniikoilla voidaan luoda ”end-to-end” verkkoperusteisia ohjelmia. Niissä ohjelmiston front-end, eli verkossa näkyvä osa näytetään asiakkaalle, jonka kautta asiakas pystyy vuorovaikuttamaan ohjelmiston back-endissä, eli palvelimen takana tapahtuvaan toimintaan käyttämällä asiakkaalle esitetyn käyttöliittymän toimintoja.

Tämän tyylin on mahdollistanut nykyajan tehokkaat ja pitkälle kehitetyt verkkoselaimet ja huomattavasti 90-lukua ja 2000-luvun alkua nopeammat verkkoyhteydet, ja kehittäjän työtä tehostavat sovelluskehykset. Nykypäivän verkkosivut ovat kuitenkin kookkaampia, joten sivun esitykseen vaaditut kirjastot ja staattiset tiedostot tarvitaan kuitenkin ladata varsinaisen verkkosivun lisäksi. Nopeampi keskimääräinen latausnopeus sallii sivun osien monipuolistamisen kirjastoilla ilman että siitä aiheutuu käyttäjälle ylimääräistä haittaa. Tarpeen tullen monimutkaisiakin front-end -sovelluksia voidaan tarjota pätkittäin käyttäjälle käyttämällä ”lazy-loading”, tekniikkaa, eli jopa yhteen sivuun perustuvilla sivuilla voidaan

ladata tietoa näin vasta sitä käyttäjän pyytäessä. Silloin hitaat verkkoliittymät pystyvät lataamaan sivuja kohtuullisella vauhdilla.

RAIL-suorituskykymallin mukaan käyttäjän toimista aiheutuva viive on maksimissaan 100ms. (Google, 2023) Tämän saavuttaminen voi olla vaikeampaa lazy-loadia käyttäessä, jos esimerkiksi uusien näkymäkomponenttien lataamisessa kestää, kun esimerkiksi valikosta painetaan jotakin nappia, mutta tämän saa ratkaistua ”prefetch”-tekniikalla, eli ladataan selaimeen sellaiset komponentit, jota käyttäjä todennäköisesti tulee lataamaan seuraavassa näkymäsiirrossa. Lataus tehdään silloin, kun muuta lataamista ei tapahdu käyttäjän selaimella, jolloin se ei vaikuta muutoin sivun lataamiseen.

Teknologiankehitys on selkeästi vaikuttanut uusien ja jatkokehitettyjen sovellusten arkkitehtuuriin. Verkkosovelluksilla on ollut samanlainen vaikutus työpöytäsovelluksiin, kun on ollut työpöytäsovelluksilla perinteisiin terminaaliohjelmiin. Työpöytäsovellusten käyttö- ja kehitysaste on trendinä vähentynyt huomattavasti, ja sen tilalle on tullut verkkosovellukset. Tämän vuoksi sillä on ollut työllistävä vaikutus vakiintuneisiin työpöytäsovelluksiin, sillä niistä on kehitelty verkkosovellustrendin noustessa myös verkossa toimivia versioita. Tästä esimerkkinä moni Microsoftin Office 365 perheen sovellus on käytettävissä verkkosovelluksena työpöytäliittymän sijasta, kuten esimerkiksi Microsoft Word ja Excel ovat myös verkkokäyttöisiä. Moni VOIP (Voice over IP) palvelua on mahdollista käyttää verkkoliittymänä, kuten Microsoftin Skype ja Discord Inc:in Discord. Jopa moni teollisuuden toiminnanohjausjärjestelmä toimii monimutkaisuudestaan verkossa, kuten CGI:n kehittämä IT4Cargo logistiikkasovellus (CGI, 2023).

4 Verkkokehityksen konseptit

Verkkokehityksellä viitataan kaikkien internetistä noudettavien verkkosivujen, sovellusten sekä verkkorajapintojen kehittämiseen. (GeeksForGeeks, 2021) Verkkosovellusten koko vaihtelee rajusti projekteja vertaillessa, mutta huolella suunniteltu verkkosovellus taipuu esimerkiksi laajasti käytettyjen ERP (Enterprise Resource Planning) järjestelmien käyttöön. Tämänkaltaisia ERP järjestelmiä voidaan käyttää esimerkiksi jonkin työtehtävään liittyvään prosessin tai prosessiketjun seurantaan, tekoon ja kirjaamiseen.

Pintapuolisesti verkkosovelluksilla on kaksi osaa: Palvelimella tapahtuva toiminta, sekä käyttäjän selaimella tapahtuva toiminta. Selaintoiminta tapahtuu tietokonepiirin näkökulmasta käyttäjän koneella. Näitä osia kutsutaan nimillä back-end, eli palvelimen osa ja front-end, eli selaimen osa (W3Schools, 2023-b). Front-end-osa järjestelmästä voi olla todellisuudessa varsin toimeton reaaliaikaisten vaikutusten näkökulmasta. Sen tarkoituksena silloin on vain esittää sivun käyttäjälle palvelimelta noukittuja verkkodokumentteja, jotka tyypillisesti lähetetään HTML (HyperText Markup Language) tiedostojen muodossa. Perinteiset verkkosivut eivät sisällä JavaScript ohjelmakoodikirjastoja, niiden ulkoasua ohjaa täysin kuvauskielinen pohja, sitä voidaan silloin kutsua deklaratiiviseksi ohjelmoinniksi. Sellaisesta sivusta ei löydy toimintaa ohjaavaa logiikkaa. Sen sijaan kaikki sivun elementit vaan kuvataan sellaisina kuin ne ovat.

Tätä toimintaa voi kuitenkin laajentaa antamalla eri sivun elementeille eri näköisiä arvoja ja muuttujia, jota voidaan yhteen koottuna pitää sivun ”tilana” millä tahansa annetulla ajanhetkellä sivulla tapahtuvassa laskennassa. Arvojen antaminen ei onnistu tavanomaisilla kuvauskielillä, joten sitä varten tarvitaan JavaScriptin käyttöä selaimen ohjelmointikielenä. Näin ollen sivulla voidaan esittää muuttuvaa tietoa, kuten palvelimelta haettuja taulukoita, jonka alkioita voidaan muuttaa suoraan tai vuorovaikutuksesta. Sivulle määritelty ”tila” sallii sen, että sivun raamien päälle voidaan rakentaa HTML elementtejä lennosta käyttäen pelkästään dataa, sekä front-end-kirjastojen tarjoamia työkaluja. Näin ollen sivun staattisia elementtejä ei tarvitse erikseen muuttaa, jotta esimerkiksi sivulla saadaan esiteltyä tuotekatalogi, ilman että jokaiselle tuotteelle rakennettaisiin sivulle eksplisiittisesti oma elementti sivun lähdekoodissa.

Nykyään koko sivu saadaan rakennettua kohtuullisen pienellä vaivalla erilaisten loogisten prosessien kautta käyttäen eri front-end-verkkosovelluskehyksiä. Tämänkaltainen rakentaminen voi tapahtua selaimen näkökulmasta staattisemmin käyttämällä SSR (Server Side Rendering) tekniikoita, jossa käyttäjän pyynnöstä tehdään jostakin sivusta jo valmiiksi palvelimella esikäsitelty versio, tai sitten dynaaminen generointi tehdään käyttäen selainpuolen ohjelmointia JavaScriptillä. Sivu voidaan generoida JavaScriptillä, jossa dynaamisten elementtien data perustuu puhtaasti dataperusteiseen sisältöön palvelimelta. Tällöin HTML-dokumentin rooli ei ole pelkästään deklaratiivisesti sisältöä kuvaava tiedosto,

mutta sen sisällä olevia HTML-elementtejä voidaan ohjata sivun dataa hyödyntäen. Tuotantoympäristössä toimintaan liittyvä koodi on todennäköisesti paketoitu JavaScript-lähdekoodiin viittaavaan linkkiin, joka on liitetty osaksi HTML-dokumenttia.

Back-endin tehtävänä on perinteisesti verkkosivun verkkodokumenttien lähetys käyttäjän selaimelle, mutta sillä voidaan ajaa monimutkaisempiakin prosesseja jonkin sovelluksen tietokannan ja käyttöliittymän välillä. Tämän kaltaisista tapahtumista esimerkkeinä seuraavat, käyttäen kontekstina logistiikka-alan kuljetustilausten suunnittelujärjestelmää:

- autentikointi (kirjautuminen)
- kuljetustilauksen kirjaus (tietokannan taulujen täyttö datalla, saadun datan validointi sekä prosessin luonnin sivuvaikutukset, kuten esimerkiksi liikennöitsijälle lähetetty ilmoitus)
- kateraportin muodostaminen (laskennallinen toiminta esimerkiksi datan yhdistämiselle ja dokumenttien muodostus tai niiden datan rakentaminen sekä lähetys käyttäjälle).

Back-endin tehtävänä on suorittaa sellainen laskennallinen toiminta, jota ei suoriteta käyttöliittymässä. Back-end muodostuu verkkosovelluksen bisneslogiikasta ja sen validoinnista, mutta sen lisäksi siihen kuuluu ”viemäröintiä”, jossa palvelimen ohjelman eri vastuualueiden ohjelmamoduulit liitetään yhdeksi kokonaisuudeksi käyttämällä moduuleja yhdistäviä ohjelmakoodia tai konfiguraatiotiedostoja.

Kommunikointi front-endin ja back-endin välillä voi tapahtua useammassa eri muodossa, mutta niistä yleisimmät ovat SOAP (Service Oriented Architecture Protocol) ja REST (Representational state transfer). SOAP on näistä kahdesta perinteisempi. SOAP eroa REST rajapinnasta sillä, että kommunikaatio SOA protokollalla tapahtuu eksklusiivisesti XML formaatissa. REST Rajapinnat (Application Programming Interface) taas keskustelevat useammassa formaatissa, joista suosituin on JSON (JavaScript Object Notation) helppolukuisuuden ja monikielisyyden vuoksi.

JSON formaatin nimi voi olla harhaanjohtava, sillä todellisuudessa JSON:illa ja ”puhdasverisillä” JavaScript objekteilla yhteistä on vain keskenään vaihtokelpoinen syntaksi. Front-endiltä kommunikaatio Back-endille tapahtuu usein käyttöliittymän näkökulmasta JavaScript kirjaston kautta, jonka vuoksi JSON on luonteva valinta palvelimelle lähetettävälle viestin sisällölle ja palvelimen vastauksen dataan.

SOAP liittymässä ei myöskään ole pakko käyttää pelkästään XML-formaattia, sillä viestin varsinainen sisältö voi olla silti jotain muuta formaattia, vaikka datan ”kirjekuori” on XML-muotoista. XML-muotoisessa datassa on kuitenkin JSON-formaattia enemmän turhaa dataa, jonka avulla rakennetaan viestin datan rakenne. JSON on kevyemmän syntaksin vuoksi teoreettisesti nopeampaa lähettää ja vastaanottaa, vaikka ero saattaa olla moderneilla lähetys- ja latausnopeuksilla virhemarginaalin alapuolella. Kuvassa 5 havainnollistetaan XML-formaattia. Kaikki ylimääräiset erikoismerkit sekä metatiedot tuovat XML-formaattiin ylimääräistä ladattavaa.

Kuva 5 XML-formaatin esimerkki. Dataa on lyhennetty

```
1   <?xml version="1.0" encoding="UTF-8">
2   <data>
3       <bookCategories>
4           <bookCategory>Fantasy</bookCategory
5           <booksInCategory>
6               <book>
7                   <name>Little Fantasy Book</name>
8                   <ISBN>...</ISBN>
9                   <author>Jane Doe</author>
10              </book>
11              ...
12          </booksInCategory>
13      </bookCategories>
14  </data>
15
```

Kuvassa 6 on taas JSON dataa. Siitä huomataan, että viestin datasta saa riisuttua huomattavan paljon sisältöä käyttämällä JSON-formaattia. Tämän lisäksi kuvan 6 esimerkkiä on helpompi käsitellä selaimella, sillä JSON dataa on helpompi lukea JavaScriptillä. Sen lisäksi formaattia on ihmisenkin helpompi lukea, sillä eri datatyypit kohdellaan eri tavalla JSON-formaatissa. Ominaisuudet kuvataan yksinkertaisesti mallilla **avain : arvo**. Jonot kuvataan **avain : []** merkinnällä, jossa hakasulkeet osoittavat jono-ominaisuuden. Objektit taas kuvataan **avain : {}** merkinnällä, jossa aaltosulkeet kuvaavat objektin sisällön, joka voi sisältää jonoja tai ominaisuuksia.

Kuva 6 kuvan 3 data JSON-formaatissa. Dataa on lyhennetty

```
1  {
2      "bookCategories" :[
3          {
4              "bookCategory" : "Fantasy",
5              "booksInCategory" : [
6                  {
7                      "name" : "Little Fantasy Book",
8                      "ISBN" : "...",
9                      "author" : "Jane Doe"
10                 }
11             ]
12         }
13     ]
14 }
```

4.1 Front-end

Front-end on selaimen rajapinta verkkosivulle ja sen tehtävänä on palvella esityskerros palvelimen toiminnoista käyttäjälle. (GeeksForGeeks, 2021) Teknisellä tasolla front-end on yksinkertaisimmillaan staattisia verkkosivuja, jossa asiakkaalle palvellaan HTML ja CSS

tiedostoja. Monet verkkosivut vaativat sivulle dynaamisuutta. Esimerkiksi partureiden ajanvarauspalvelut ja verkkokaupat tarvitsevat toimiakseen muuttuvaa tai funktionaalista sisältöä, jota ei ole tehty pelkästään tarkasteltavaksi, mutta hiirellä tai sormella painettavaksi, lisättäväksi tai poistettavaksi. Käytännössä käyttöliittymiä on kahta eri tyyppiä: Imperatiivinen, jossa käyttöliittymän toimintaa kuvataan askel askeleelta. Näitä on tyypillisesti vanhemmat järjestelmät. Imperatiivisen käyttöliittymäohjauksen huono puoli on se, että lähdekoodia on haastavampi ylläpitää ja ei ole kovin muutossietoinen. Deklaratiiviset käyttöliittymät taas kuvaavat toiminnan rakenteen ilman kuvailematta sen askeleita toiminnankulkuna, ja näitä ovat tyypillisesti modernit front-end -kirjastot.

Näissä järjestelmissä muutoksia on helpompi toteuttaa, sillä koodin implisiittisyys vähentää muutettavan koodin määrää. Deklaratiivisessa verkkokäyttöliittymässä pyritään parhaan mukaan säilyttää HTML-dokumentin kuvauskielen muutossietoisuus. Siinä siirretään ehdollistava logiikka esimerkiksi front-end -sovelluskehyksen pääteltäväksi sisäisesti. Kriittinen ero deklaratiivisen sekä imperatiivisen käyttöliittymän välillä on se, että deklaratiivisessa järjestelmässä data on irrallinen käyttöliittymästä, ja sen tila ei ole automaattisesti synkronoitavissa.

Tämän kaltaisilla sivuilla voidaan tuottaa sisältöä, jonka vaihteluväli on tiuhaa. Tämä vaatisi perinteisessä staattisessa sivussa paljon ylläpidollista työtä sivun kehitysryhmältä tai kehittäjältä, sillä tiedot täytyisi silloin kirjata kaikki käsin sivun muodostaviin HTML-dokumentteihin. Jossakin määrin sellaiset ratkaisut saattavat olla logistisesti mahdottomia pelkästään sivun kuvauskielien avulla. Kuvauskielille ei voida asettaa sivulle ”tilaa”. Dataa ei pystytä manipuloimaan perinteisen ohjelmointikielen tapaan. Ongelma voidaan kiertää yksinkertaisesti sillä, että palvelin rakentaa käyttäjälle näkymän valmiiksi itse, ja lähettää sen esimerkiksi XML formaattina eteenpäin komentona. Palvelimen HTTP-vastaus sitten käskyttää käyttäjän selainta rakentamaan HTML-elementtejä sivulle vastauksen perusteella. Tällöin sivusto on todellisuudessa melkein täysin naiivi sivun loogisen ohjaamisen kannalta, mutta se ei välttämättä näy käyttäjän selaimessa merkittävänä erona. Palvelimella rakennettu HTML on todennäköisesti tehty niin, että sen on täytynyt seurata jonkinlaista loogista prosessia, joka tarkoittaa myös sitä, että sen sisältöä on vaikeampi jälkeenpäin

muuttaa. Jokainen pienikin looginen lauseke vaikeuttaa muutosten tekoa, sillä ehdollistava koodi vaatii koodilta risteymäkohtia.

Dynaamisen verkkosivulla voidaan näyttää muuttuvaa tietoa vaivattomasti. Dynaamisen verkkosivuston ylläpito ja päivitys on huomattavasti valtavia staattisia sivuja suoraviivaisempaa, sillä suuren siivun sivuston sisällön tuottamisesta tekee verkkosovelluskehys tai muut verkkosivulla käytetyt JavaScript kirjastot. Nämä kirjastot voivat olla yksinkertaisimmillaan muodostamassa AJAX pyyntöjä back-endille sivun sisällön hakemista varten (W3Schools, 2023-a). Palvelimen tuottama staattinen HTML-sisältö komentojen kautta aiheuttaa haasteita HTML-sisällön ja JavaScript koodin muuttamiselle jälkeenpäin, etenkin jos kyseinen koodi nidotaan muun palvelinkoodin sekaan raakana merkkijonotyyppisenä muuttujana. Silloin esimerkiksi palvelinkoodia varten käytetty IDE (Integrated Development Environment) tai tekstieditori ei ymmärrä kuvauskielen syntaksia muuksi kuin raa'aksi tekstiksi, jolloin tekstieditorista ei saa käytettyä sen kaikkia hyötyjä.

JavaScriptillä saadaan siis tuotettua toiminnallisuutta sivulle, mutta dynaamisuuden laajentuessa sivun kannattaa hyödyntää sivun ”tilan” ylläpitoa helpottavaa front-end kirjastoja. Tyypillinen verkkosovelluskehysprojektin käyttöliittymä on rakennettu SPA:na (Single Page Application). SPA:n toiminta perustuu siihen, että koko verkkosivu palvellaan yksittäisen indeksisivun, eli etusivun kautta. SPA:ta voi toki palvella osana jotain muulla tavalla toteutettua sivua tai sivuhakemistoa. Indeksisivulle saapuessa selaimelle välitetään verkkodokumentissa viite verkkosovellukseen, joka ajaa verkkosovelluksen front-endin kirjaston käyttäjän selaimella, käyttäen HTML-dokumenttiin nidottua JavaScript tiedostoviitettä. Jos SPA:n sisällä sivun reitti muuttuu (mennään alasivulle yms.) Koko sivua ylä- ja alatunnisteineen ei ladata uudestaan, vaan pelkästään sivulla muuttuneet osat ladataan uudestaan. Esimerkkinä Verkkokauppa.com on SPA. Tämä sivu on selkeästi SPA, sillä koko sivu ei virkisty, kun siirrytään eri näkymään. Kontrastina ”Jimms” tietotekniikan verkkokauppa lataa koko verkkosivun uudelleen, jos vaihdetaan sivun tuotekategorioita. Ainoa ladattu HTML-komponentti Verkkokauppa.comin sivulla oli ensimmäinen sivu, jolle saavuttiin selaimesta. Jimmsin verkkosivu sen sijaan palveli uuden HTML-sivun jokaisen sivusiirron yhteydessä.

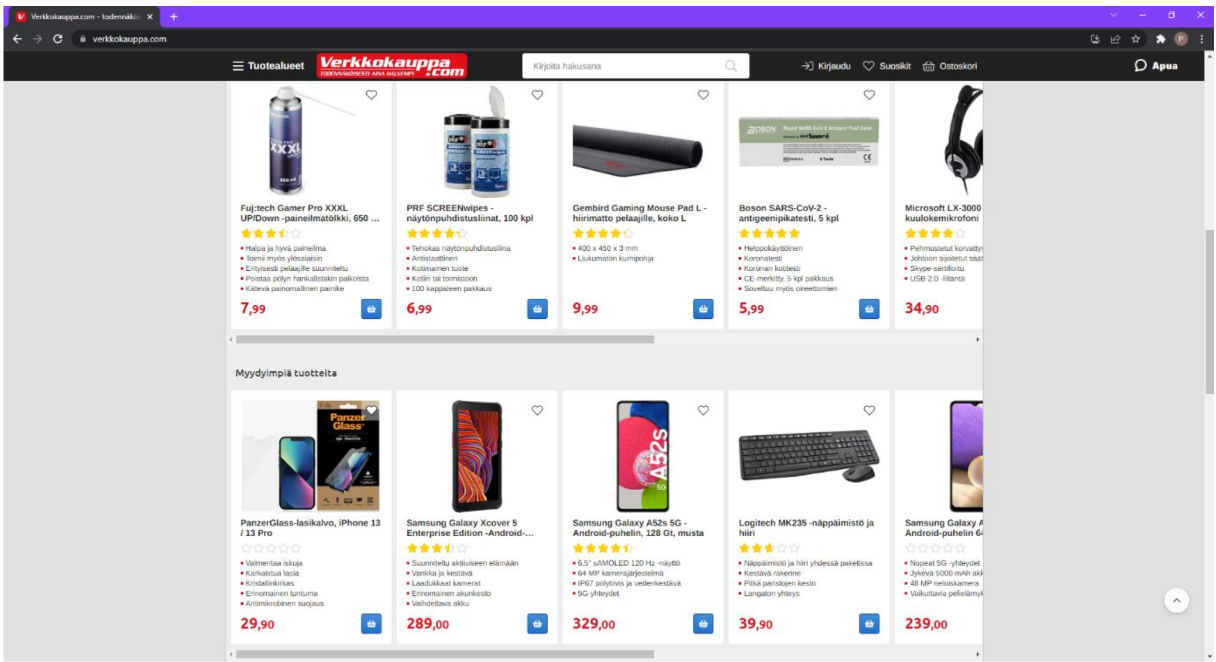
Käytännössä koko sovellus voi perustuu indeksiin, johon ladataan sivun navigaatioelementit, ja varsinaista sivun sisältöä sivun sisällä olevasta reititysnäkymästä, jotka voivat olla jopa sisäkkäisiä. Front-end -kirjastoilla tehtyjä sovelluksia voidaan myös tarjota osana jotakin yksittäistä reittiä verkkosivulla, samalla kun loput sivusta on luotu CMS (Content Management System) järjestelmän avulla, tai suoraan raakoina HTML sivuina. SPA ei kuitenkaan jokaiselle verkkosivulle hyvä ratkaisu. Se on kehno vaihtoehto esimerkiksi sellaisille sivuille, jonka suosio perustuu hakumoottorien tuottamiin hakutuloksiin. SPA-sivuja on ohjelmalogiikan vuoksi vaikea parsia hakumoottoreiden algoritmeilla, jonka vuoksi niiden SEO (Search Engine Optimization) on huonolaatuinen, jonka vuoksi ne ei välttämättä löydy hakumoottoreiden hakutuloksista kovin helposti.

Hyvä SEO ei ole mahdotonta SPA sivulla, mutta se on huomattavasti haastavampaa ja se saattaa vaatia sen takia erityistä tarkkuutta kehittäjältä. Moni front end -kehys toki tukee SSR tekniikoita, jonka avulla hakukoneet pystyvät parsimaan kyseisiä sivuja paremmin. Kaikki verkkoapplikaatiot eivät välttämättä kärsi samasta SEO-ongelman huonoista vaikutuksista, sillä esimerkiksi B2B (Business To Business) ohjelmat ovat yleensä käytöltään sopimusperusteisia, joten niitä ei välttämättä samalla tavalla etsitä hakumoottoreista kuin esimerkiksi blogeja tai kirjastojen verkkosivuja. Korkeintaan niitä mainostavilla verkkosivuilla pitäisi olla hyvä SEO, jotta niitä ostavat asiakkaat löytäisivät SPA:ta hyödyntävät sovellukset tuotteina.

4.2 Front-end esimerkkinä

Valtava osa nykyajan palvelusivustoista vaativat jonkin front-endiä varten tai front-endiin soveltuvan verkkosovelluskehyksen toimiakseen sujuvasti. Voidaan esimerkkinä käyttää Verkkokauppa.com sivua, joka näkyy kuvassa 7.

Kuva 7. Verkkokauppa.com sivun etusivu



Jotta tämä sivu toimisi oikein, front-endin tarvitsee muodostaa Verkkokauppa.com sivulle tiedossa oleville tuotteille erilaisia näkymiä. Front-end kyseisessä sivussa kykenee muodostamaan sivun back-endille tietynnäköisiä kyselyjä, kuten tietoja myydyimmistä tuotteista. Kyselyillä pystytään muun muassa lajittelemaan tuotteita hinnan mukaan laskevasti tai nousevasti. Tuotteita myös pystyy lajittelemaan tuotealueen perusteella sekä sivulta pystyy hakemaan tuotteita hakupalkin avulla. Lajittelun ja haun tekee siis palvelin, mutta selaimella tehdään alkuperäinen pyyntö, sekä selaimella data saadaan esiteltävään muotoon. Vastaavia toimintoja on kyseisellä verkkosivulla monia muitakin, mutta konkreettisin huomio asiassa on se, että jokaista erillistä ”sivua” ei välttämättä ole rakennettu käsin, vaan sivu on suurin osin tuotettu ohjelmallisesti käyttäen valmiita dokumenttisapluunoita sekä tietolähteen rakenteen muodostavia malleja. Niistä on manipuloitu sivun mallin avulla yksilölliset sivut, kuten esimerkiksi tuotetietosivut.

Pohja tuotelistaukselle voi olla hyvin yksinkertainen **<div>** HTML elementti (jonka sisällä on esimerkiksi kuva-elementti sekä linkki tuotesivulle), jota on iteroitu tuotteiden määrän verran sivulle, kunnes selaimen sivu täyttyy maksimäärästä tuotteita. Sen jälkeen lisää tuotteita saa näkyviin seuraavalta sivulta.

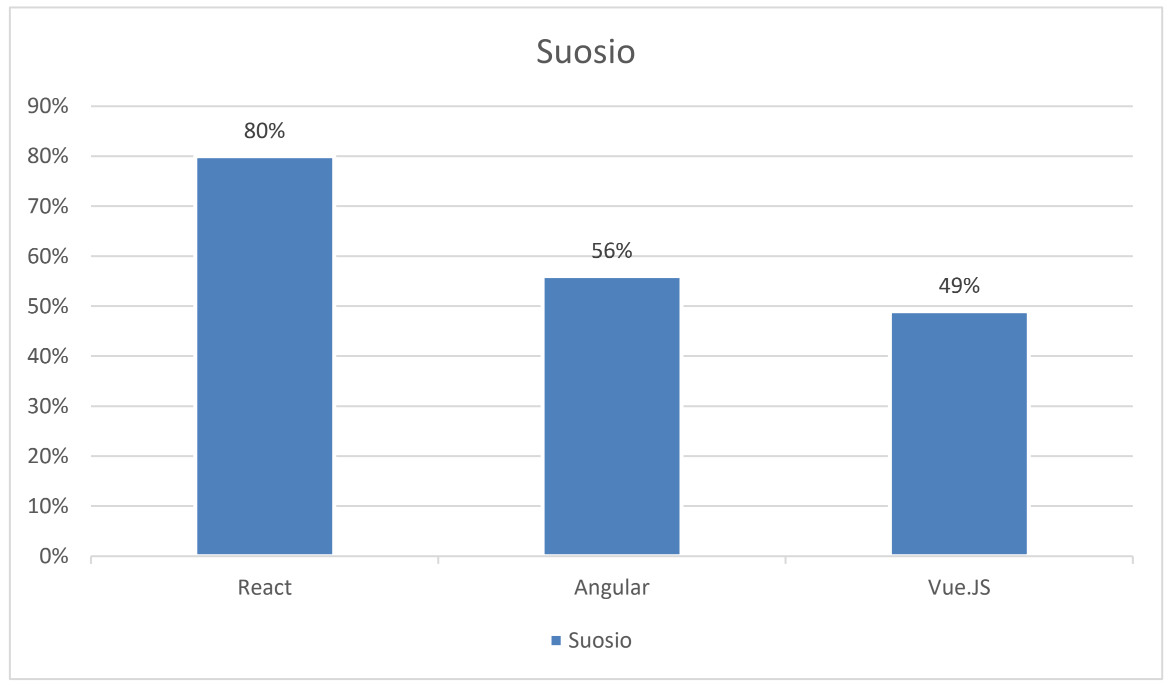
Modernilla JavaScriptillä on myös mahdollista kehittää tuotteiden loppuun saakka vieritettävä tuotekatalogi yhdellä sivulla lisäämällä jo ladattujen tuotteiden loppupäähän sellaisen tapahtumaa tunnistelevan ohjelmakoodin, joka lataa lisää tuotteita, silloin kun käyttäjä vierittää näkymässä (tai elementissä) tietyn pisteen ohi. Tämä on eräs tapa, jolla verkkosivun käyttäjäkokemuksen laatua voidaan yleisesti parantaa. Suosion sille on helppo ymmärtää modernina ratkaisuna, sillä esimerkiksi sormikäyttöisillä älypuhelimella on helpompi selata sivua vaan alas, sen sijaan että älypuhelimella joutuisi painamaan pienikokoisia sivunvaihtonäppäimiä.

Verkkokauppa.comin rakenteesta voidaan myös huomata, että vähintään katalogi ja etusivu toimivat SPA:n avulla. Verkkosivun reitin muutos ei päivitä koko sivua, mutta sen sijaan muuttuneet elementit virkistyvät. Tämä voidaan todistaa esimerkiksi selaimen kehittäjän työkalujen verkkoliikenteen paneelista. REST Rajapinnan käytön hyöty on myös sivun verkkoliikenteessä ilmeinen: tuotekatalogin rivit ladataan JSON formaattina, jossa tuotteet esitellään ”products” taulukkona. Datan esitys ei ole näin palvelimen suorana tehtävänä, vaan datan esitykseen omistettu käyttöliittymä on siitä vastuussa. Sen vuoksi palvelin ei ole riippuvainen front-endin rakenteista. Lyhyesti kuvattuna hyöty on konkreettisesti siinä, että helposti muuttuvat elementit (käyttöliittymä) ei vaikeuta harvemmin muuttuvien elementtien (palvelin back-end moduulit) ylläpito- ja kehitystoimintaa.

4.3 Front-end kehityksen teknologiavalinnat

Front-end -kehyksiä on nykypäivänä monta, mutta niistä kolme suosituinta esitellään kevyesti tulevissa kappaleissa. Alla on esitelty kaavio, jonka pystyakselilta nähdään käyttävätkö kyselyyn vastanneet kyseistä verkkosovelluskehystä.

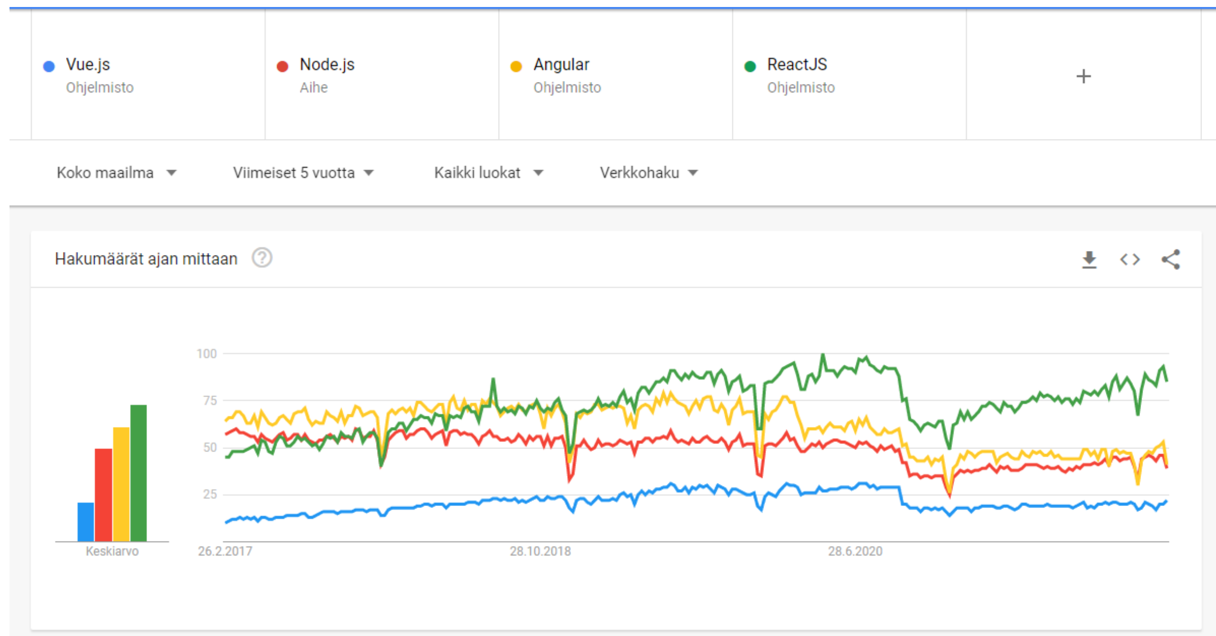
Kuva 8 Suosituimmat front-end -kehykset. Taulukko koostettu lähteen datasta. (Greif & Benitte, 2020)



Jopa 80 % kyselyyn vastanneista käyttävät vähintään jossakin määrin Reactia. Samaa suosiota ei saanut Angular tai Vue, mutta verrattuna kaikkiin muihin kehyksiin esitelty kolmikko selvästi dominoi front-end -kehysten suosiotilastoja. Ero käyttöasteessa neljänneksi suosituimmalle kehykselle on verrannollinen Reactin ja Angularin väliseen kuiluun.

Kuvassa 9 on tehty hakusanatutkimus Google Trendsin avulla. Siinä selkeäksi hakusanatilaston voittajaksi on noussut React. Samankaltainen trendin nousu on nähtävissä StateofJs-tutkimuksessa, jossa vuonna 2016 Reactin käyttöaste oli 53 prosenttia, joka on nykyistä tutkimuksen antamaa arvoa puolet pienempi. (Greif & Benitte, 2020). Suosio syy ja käyttökohteet ovat tarpeen esitellä seuraavissa kappaleissa.

Kuva 9 Google Trends hakuvertailu määrän perusteella termeille front-end kehyksille Vue, Node.js, Angular sekä React.



4.3.1 React

React on avoimeen lähdekoodiin perustuva työkalu käyttöliittymien rakentamiseen. React on tilastollisesti myös käytetyin JavaScript-kieleen perustuva verkkosovelluskehys. (Greif & Benitte, 2020) Reactin isoimmat hyödyt ovat laaja soveltuvuuden kirjo, valtava yhteisö sekä helposti opittavat rakenteet. (Sakovich, 2023) Reactin julkaisuajalla keskeistä oli myös Reactin esittämä paradigma ”tilaperusteisiin” käyttöliittymää.

React tulee sanasta reaktiivisuus, jolla taas sellaiseen käyttöjärjestelmään, joka pystyy muovautumaan sivun datan muuttuessa automaattisesti (Chedeau, 2016). Reactia myös ajateltiin nopeaksi järjestelmäksi, mutta väitteessä on huomionarvoisia seikkoja: React on Angulariin ja Vueen verrattuna edelläkävijä, ja oli julkaisuajallaan todellisesti nopeampi kuin muut ratkaisut. Reactin ja muiden samanlaisten verkkosovelluskehysten nopeutta nykyään haastaa uudemmat verkkosovelluskehykset, kuten Svelte. Tyypillisesti muut tunnetuimmat verkkosovelluskehykset kuitenkin ovat rakenteellisesti samankaltaisia kuin React.

React on komponenttiperusteinen ja käyttää JSX syntaksinlaajennusta JavaScriptille. JSX-syntaksin ajatus on luoda komponentteja Reactin käyttöliittymälle niin, että kuvauskieltä ja JavaScriptiä voidaan yhdistellä saman lausekkeen sisälle. Komponentit vastaavat JavaScript-funktioiden kaltaisia rakenteita, joita voidaan uudesti käyttää yksittäisessä verkkosovellusprojektissa useammassa paikassa kutsumalla kyseisen komponentin lähdettä, ja sitten niittaamalla uusiokäytetyn komponentin eri paikkoihin. Komponentin avulla jokaista elementtiä voi tarkastella ja kehittää eristyksessä muusta käyttöliittymän näkymästä, ja jokaiseen komponenttiin on mahdollista sisällyttää dataa. Kuvassa 10 on esimerkki yksinkertaisesta JSX-komponentista. Muuttujan tyyppi ja nimi ovat JavaScriptin syntaksin mukaista, niin on myös yhtäsuuruusmerkki, mutta muuttujan arvo on kuitenkin HTML-formaatin muotoista.

Kuva 10 Esimerkki JSX-syntaksin käytöstä.

```
4 | const reactElement = <p>Hello, this is an element in React!</p>
```

Esimerkin vakiota **reactElementiä** voidaan käyttää asiakkaalle näkyvässä verkkosivussa renderöimällä, eli piirtämällä kyseinen vakio käyttämällä Reactin renderöintimetodia. Kuvassa 11 on esimerkki siitä, että miten voidaan renderöidä kuvassa 10 esiintyvä komponentti Reactissa.

Kuva 11 Elementti piirretään ruudulle .render() metodin avulla

```
const reactElement = <p>Hello, this is an element in React!</p>
ReactDOM.render(
  reactElement,
  document.getElementById('root')
);
```

Komponenttien ajatus on hyödyntää uudesti käytettäviä komponentteja. Yhden React näkymän voi koostaa useammasta React komponentista. Komponentit voivat toimia mallipohjina elementeille, joten niille voi antaa tietoa ”props”-tietona. Props on lyhenne properties-sanalle ja viittaa ajonaikaisesti muuttumattomaan tietoon (read-only). Siihen saa

sijoitettua mitä tahansa ohjelmakoodiksi kelpaavaa dataa, kuten esimerkiksi etu- ja sukunimiä, tai tuotekuvauksia.

Käytännössä ”props” tiedoilla dokumenttihierarkiassa omistavat komponentit voivat siirtää erilaisia muuttuvia tietoja niiden omistamille komponenteille, muodostaen dynaamisesti renderöitäviä sivun osia. Merkittävä tekijä tässä on se, että se kyseinen data voi olla jopa hierarkkinen, mutta käyttöliittymä ja data voi silti säilyä oikeellisessa tilassa toisiinsa verraten. Syvä datan hierarkia voi nimittäin aiheuttaa perinteisen verkkosivuapplikaation kehittämisessä haasteita, kun käyttöliittymän oikeellista tilaa pitää synkronoida koko datan hierarkiaa myöten.

Kuvassa 12 on demonstroitu komponentin käyttöä **Hello**-funktiolla. Funktio vastaa JavaScript funktiota. Kyseinen funktio on Reactin kontekstissa komponentti. Ideana on se, että pääsivun rakentava JavaScript tiedosto (index.js) renderöi Reactin avulla **Hello**-funktion palautusarvon näytettävän sivun HTML-dokumentin sisällä. Tästä funktiota vastaavasta komponentista voi poimia argumentteja, jonka vuoksi ”props” muuttujaan voi tallentaa erilaisia tietoja.

Kuva 12 Hello nimistä komponenttia käytetään React-sovelluksen indeksisivulla

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

ReactDOM.render(
  <React.StrictMode>
    <Hello user="John" />
  </React.StrictMode>,
  document.getElementById('root')
);
```

```
function Hello(props)
{
    return (
        <div>
            <h1>Hello {props.user}, welcome to our React app!</h1>
        </div>
    );
}
export default Hello;
```

Hello John, welcome to our React app!

Tässä tapauksessa sille tallennetaan argumentti **user**. Tämän argumentin arvo määritellään esimerkissä olemaan ”John”, jonka vuoksi **Hello**-elementtiin voidaan lisätä muuttujana määritelty käyttäjänimi. Kaikki muu teksti on vakiota, mutta dynaamisuus on saavutettu

personifioimalla käyttäjän nimi sivulle muuttuvana tietona. Esimerkin vuoksi tässä tapauksessa muuttuva prop-tieto on annettu elementille vakioituna ”John” merkkijonona.

Reactin suurin innovaatio julkaisuajaltaan on Virtual DOM. DOM-manipulaatio viittaa Document Object Modelin muokkaamiseen. DOM tarkoittaa sen dokumentin rakennetta, joka muodostaa asiakkaan näkemän sivun. DOM-manipulaatiolla tarkoitetaan DOM:in rakenteellista muuttamista, kuten tekstisisällön muokkaamisen tai listojen (kuten tuotekatalogien) päivityksen kautta. (Mozilla, 2022)

DOM-manipulaatio on raskasta, etenkin jos on kyseessä valtava sivu, jonka DOM on iso. DOM-manipulaatio tehtiin aikaisemmin sellaisella tavalla, että koko DOM rakennetaan uusiksi, vaikka muutos koskisi vain yhtä sivun elementtiä. Reactin virtual-DOM:in avulla manipuloidaan vain DOM:iin perustuvia elementtejä, jotka vastaavat konseptitasolla DOM:in elementtejä. DOM:ia itsessään ei muokata kehittäjän näkökulmasta, mutta virtual-DOM:iin tapahtuvat muutokset peilataan varsinaiseen DOM:iin. Jos Reactin Virtual DOM:in ohjelma huomaa, että virtuaalista DOM objektia on päivitetty, React päivittää vain ja ainoastaan kyseisiä elementtejä DOM:issa. (Codeacademy, n.d.)

Reactin nopeuden väitteissä on kuitenkin erittäin tärkeää ottaa huomioon se, että väitteet on tehty silloin, kun monia nykypäivän tunnettuja verkkosovelluskehyksiä ei ollut vielä kehitetty, jonka vuoksi muita kehyksiä oli helppo vertailla hitaiksi. Eräs verrannollisesti nopea kehys on Svelte, jossa ei taas ole käytössä virtual-DOM:ia. (Harris, 2018) Mahdollisia verkkosovellusten käyttöliittymäteknologioita ei ollut yhtä paljon, ja useampi niistä ei pystynyt samaan nopeuteen. Virtual DOM:in epätäydellinen ratkaisu joka tapauksessa helpottaa varsinaisen front-endin rakentamista ilman että tarvitsee keskittyä tekijöihin, jotka saattaisivat tuottaa potentiaalisia bugeja. Hiukan hitaampi virtual-DOM:in suorituskyky on todennäköisesti silti riittävä enemmistölle projekteista.

Tässä tulee esiin se, että modernien teknologioiden välillä tehdyt vertailut ovat niin marginaalisia, että nopeus on harvemmin merkityksellinen tekijä. Sovelluskehitys ei ole kuitenkaan niin mustavalkoista, että jokaisella asialla on tietty kaava. Loogisempaa on ajatella asiaa omien tarpeiden mukaan. Avainasiana on nimenomaan riittävyys ja sopivuus.

Verrannollisesti React on suosituin verkkokehys front-end -kehityksessä (Kuva 2), jonka käyttäjäkunnasta löytyy muun muassa Facebook (Meta on kyseisen kehyksen kehittäjä ja ylläpitäjä), BBC sekä Netflix. Sen lisäksi Reactin suosiota tukee Google Trendsin avulla tehty hakusanatutkimus (Kuva 3).

Reactin käyttöä kannattaa etenkin harkita, jos haluaa käyttää kehystä, jolle löytyy paljon yhteisötukea. Reactin oppimiskäyrä ei myöskään ole kovin jyrkkä, eikä Reactin tiedostopaketit vaadi paljon valmiiden rakenteiden käyttöä. React ei sisällä kaksisuuntaista tietojen sidontaa, jonka vuoksi komponentin omistajan on vaikeampi kommunikoida suoraan omistetulle komponentille, joka saattaa aiheuttaa kehittämisessä ongelmia. Tätä varten on toki kehitetty liitännäisiä, jonka avulla kaksisuuntainen liikenne onnistuu. Tätä kuitenkin kannattaa tehdä vain harkiten, sillä se mahdollistaa lähdekoodin ryvettymisen.

4.3.2 Angular

Angular on tutkimuksen mukaan toiseksi käytetyin front-end -verkkosovelluskehys (Greif & Benitte, 2020) jonka suunnitteluideologiaan kuuluu ajatus äärimmäisestä joustavuudesta. Tämän vuoksi on melko luontevaa rakentaa monialustaisia sovelluksia Angularia käyttäen. Angular on myös tunnettu nopeista latausajoista sekä verkkokehitysjätti Googlen ylläpito- ja kehitystyöstä.

Angular on kuitenkin myös monimutkaisempi kuin React sekä moni muu vähemmän käytetympi kehys. Tämä myös peilataan siinä, että moni Reactin käyttäjä on tyytyväinen, mutta Angulariin käyttöön syystä tai toisesta on tyytyväisiä vain 42 % käyttäjistä. (Greif & Benitte, 2020) Monimutkaisuus ei kuitenkaan kaikesta huolimatta tee Angularista huonoa verkkosovelluskehystä, sillä Angularin monimutkaisesta rakenteesta hyötyvät raskaammat verkkosovellukset, jossa tarvitaan mahdollisimman monimutkaisia rakenteita sivun funktionaalisuuden saavuttamiseen. Angularin käyttö kehityksessä on tehotonta todella pienissä tai yksinkertaisissa sovelluksissa.

4.3.3 Vue

Vuen kannatus verrattuna Reactiin ja Angulariin on matalampi, mutta silti erittäin merkittävä, jos verrataan sen lisäksi muihin verkkosovelluskehyksiin, sillä käyttöä löytyi silti 49 % vastanneista, sekä tyytyväisyys on lähellä Reactin tasoa. Jopa vuoden 2018 tutkimus osoitti Vuen käyttäjät kaikista tyytyväisimmiksi. Silloin yksi kymmenestä ei ollut tyytyväinen Vuen käyttöön. (Greif & Benitte, 2020) Nykyään kyseistä tyytyväisyyden tilastoa johtaa vuorotellen React ja Svelte.

Vuen rakenne on Reactin ja Angularin kaltainen, mutta Vuella on laakeampi oppimiskäyrä kuin Reactilla tai Angularilla. Vuen komponentit rakennetaan HTML-pohjilla, jossa voidaan käyttää Reactista tuttua JSX syntaksia, mutta Vue suosittelee ns. template-tiedostojen käyttöä. Vuella on siis matalampi kynnys oppimiselle, jos kehittäjä ei osaa paljon JavaScriptiä, mutta siitä huolimatta potentiaali kehittää on käytännössä sama molemmissa teknologioissa.

Verkkosovelluskehyksenä Vue on erittäin lupaava huolimatta sen uutuudesta ja suhteellisen pienestä käyttäjäkunnasta sekä yhteisötuesta. Jos Vueta vertaa Reactiin tai Angulariin, huomataan että Vue on ainoa kolmesta, jolla ei ole isoa yritystä tukemassa kehitystä. Vuen alkuperäiskehittäjä on kuitenkin ollut töissä Googlella Angularin parissa, josta Vuen kehitysidea oli syntynyt. Silloin ajatuksena oli kehittää vastaavanlainen verkkokehys, joka on kuitenkin kevyempi ja Angularia intuitiivisempi. Vue ei ole myös yhtä teknisesti ylirakennettu kuin React, jonka vuoksi sitä on teoreettisesti helpompi oppia. Vuesta löytyy Reactista tuttuja etuja, kuten laakea oppimiskäyrä sekä Virtual DOM, eli käytännössä Vuen käyttökohteet peilaavat Reactia.

4.4 Ohjelmistokehysten hyöty kehittäjän näkökulmasta

Front-end -verkkosovellusten keskeisin käyttökohde on sovelluksen ”tilan” muistaminen, ylläpito sekä peilaaminen näkymään. Tämä on äärimmäisen hyödyllistä silloin, kun verkkosivulla halutaan esittää dynaamista dataa. Tämä on totta etenkin silloin, kun käyttäjällä on mahdollisuus vaikuttaa sen datan sisältöön. Esimerkkinä tästä voidaan ottaa

sivulla esitelty taulukko, joka sisältää rivejä jostakin taulusta palvelimelta: se sisältää ylätunnisteet, jonka kautta voidaan lajitella tietoa. Siinä on valikko, josta dataa voidaan poistaa tai lisätä, sekä ylläpitoikkuna, jossa dataa voidaan päivittää. Sellaisen näkymän rakentaminen verkkosivulle on suhteellisen työlästä ”käsin” sillä se tarkoittaisi sitä, että taulukon datan ”tilaa” joudutaan jatkuvasti ylläpitämään, silloin kun näkymällä tapahtuu joku käyttäjän selainkohtainen muutos. Jos data on muuttunut jonkun rivin sisällä, taulukon solujen täytyy myös päivittyä. Mitä rivinumeroinnille tapahtuu? Mitä jos sivulla on myös ylätunnisteita, pitääkö niiden päivittyä? Mitä jos taulukosta poistetaan rivi? Mitä jos taulukon rivejä halutaan lajitella eri järjestykseen? Kriittinen ongelma on se, että sivun tila ei perustu dataan vaan sivulle määriteltyihin lomakkeisiin ja HTML-elementteihin.

Tämän saa niinkin kierrettyä, että solut lähetetään back-endiltä suoraan HTML-dokumenttina, mutta se taas tarkoittaa sitä, että back-endissä on enemmän riippuvaisuuksia front-endiin. Data on silloin eksplisiittisesti määritelty HTML-muotoiseksi, mikä ei kelpaa komentoriville, työpöytäsovellukselle, tai mobiilisovellukselle. Silloin muiden käyttöliittymien liittymät palvelimen bisnestoimintaan pitää rakentaa erillisinä ohjelmina, joka lisää ylläpidettävien moduulien määrää. Palvelimella tehty käyttöliittymän rakennus on helpompi pitää synkronoituna (etenkin kun näkymä virkistetään aina uudelleen), mutta sen käyttöliittymää on vaikeaa muuttaa ja se on epäintuitiivinen vaadittujen sivuvirkistysten vuoksi.

Front-end -kirjastot ovat kehitetty reaktiivisiksi, eli selaimella esitettävä data muodostetaan näkyvään muotoon selaimelle verkkosovelluskehyksen työkalujen toimesta. Näin ollen kehittäjän ei itse tarvitse keskittyä kehittämään esimerkiksi taulukkoja ylläpitäviä ohjelmia. Tehtävä jää verkkosovelluskehyksen tehtäväksi, jolloin kehittäjä voi keskittyä sivun datan manipulointiin, ulkoasuun ja muiden korkeamman tason seikkojen kehitykseen.

Deklaratiivisuus nimenomaan tarkoittaa sitä, että ohjaavaa logiikkaa ei tarvitse kuvailla. Näin ollen muodostunut ohjelma on vähemmän arka bugeille, sillä funktionaalisuudesta naiivi käyttöliittymä on vikasietoisempi kuin hyvin syvällisesti kehitetty ja eksplisiittisesti ehdollistettu käyttöliittymäkirjasto. Vähemmän virheitä ja kehittämisen kohteita tarkoittaa myös sitä, että sovellus valmistuu nopeammin. Johdonmukaisesti kehitetty verkkosovellus

on myös luontevasti virhesietoisempi, sillä verkkosovelluskehyksen asettamat ”rajat” kehitykselle varmistaa, että tietyt usein bugeja sisältävät osat verkkosovelluksesta hoituvat verkkosovelluksen sisäisten rakenteiden kautta.

4.5 Back-end ja API

Back-endille löytyy paljon laajempi kielikirjo kuin front-endissä. Ohjelmakoodi ajetaan palvelimen puolella, jonka vuoksi palvelimen teknologiavalinta on käytännössä kieliriippumaton (riittää, että palvelimelle voidaan lähettää http-pyyntöjä!). Valintaan vaikuttaa se, että kuinka kohtuullista kyseisellä kielellä ja ohjelmointialustoilla on tehdä toimiva verkkosovellus. Kielivalinta ei näin oikeasti vaikuta verkkoselaimella tapahtuvaan toimintaan, sillä vastaukset näkyvät vaan kommunikointiprotokollan perusteella valitussa formaatissa, kuten JSON- tai XML muodossa. Back-endin vastuita on useita, mutta sovellustasolla merkittävin rooli on toimia datan validointina ja bisneslogiikan loogisena ohjaajana ja suorittajana.

Käyttäjä voi suorittaa sovelluksen bisneslogiikkaa muodostamalla kyselyjä johonkin back-endin päätepisteeseen. Selaintakaan tähän ei käytännössä tarvita, jos selaimen ja palvelimen toiminta on eristetty toisistaan, kuten esimerkiksi käyttämällä kommunikointivälineenä REST API-päätepisteitä. Näin ollen (tuotteesta ja tiimistä riippuen) on luontevaa, että front-endiä voi kehittää eri henkilöt kuin back-endiä, toki full stack -kehittäjän vastuulle kuuluu molemmat osat. Back-endin kehittäjän ei tarvitse tietää mitään käyttöliittymästä tai sen toiminnasta kunnolla eristetyssä järjestelmässä. Kehittäjälle riittää, että hän ymmärtää tuotteen toimialueen, tarkennettuna sovelluksen työhön liittyvän bisneslogiikan. Joku muu henkilö, joka on resursoitu tekemään front-endiä voi kehittää esityskerrosta. Ohjelmalle tehtävä looginen jako moduuleihin voidaan tehdä myös kohdistaen työtiimiin. Sen voi tässä tapauksessa helposti jopa ulkoistaa toiselle puolelle maailmaa tehtäväksi, mikä voi potentiaalisesti vähentää kehityksestä aiheutuvia kuluja.

REST-API on standardisoitu arkkitehtuurinen ohjelmointityyli, jonka avulla voidaan mahdollistaa verkkosivun back-end -palveluita käyttöliittymällä käytettäväksi. Termiä ”API” Googlen avulla etsiessä voidaan huomata merkittävä ilmiö: suurin osa hakutuloksista

käsittelevät melkein pelkästään REST API-rajapintoja. Tässä pitää huomata, että API terminä ei viittaa verkkopohjaiseen API:in, sillä API on huomattavasti sitä abstraktimpi termi. Se yksinkertaisesti viittaa kahden erillisen järjestelmän välissä kutsuttavaan rajapintaan, ja sen tietoliikenne ei tarvitse kulkea internetin välityksellä. Yksi esimerkki API:sta on Linux-käyttöjärjestelmän kernel, eli suomennettuna ydin, tai ”kerneli”. Linux kernelin avulla ohjelmoija voi suorittaa Linux-järjestelmän järjestelmäkutsuja, jotka huom. ei ole suoria ohjelmakutsuja. Samoin REST-API:lle tehtävät kutsut eivät ole suoria ohjelmakutsuja, vaan ne ensiksi lähetetään kutsun käsittelijälle ensin, jonka kautta kuin ”sivuvaikutuksena” tehdään toimintoja palvelimella.

Käyttöliittymän ja REST-API:n keskustelu tapahtuu HTTP-kyselyiden muodossa, eli ajatuksena on se, että back-endistä saadaan esimerkiksi tietokannan tauluihin perustuva projektio HTTP-viestin kehossa kuljetettua selainohjelmaan (esimerkkinä edeltävä Verkkokauppa.comin tuotteet-objekti). Selainohjelmassa voidaan käsitellä viestiä esimerkiksi JSON (JavaScript Object Notation) muodossa. Asiakasohjelma voi siten täydentää sivulle tietoja back-endin antamien tietojen perusteella. Tämän API-rajapinnan rakenne on täysin standardisoitu, ja käyttää http pyyntöjä ja http vastauksia kommunikointiin, jonka vuoksi API on agnostinen sen palveleman rakenteen suhteen. Ainoa todellinen merkitsevä tekijä on se, että pyynnön formaatti on oikeellinen kohteen bisnessääntöjä vasten, ja onko käyttäjä auktorisoitu käyttämään rajapinnassa olevaa resurssia pyynnön suoritukseen.

Jokainen hylätty pyyntö ei myöskään välttämättä ole ajonaikainen virhe, vaan voi myös johtua puutteellisesta auktorisoinnista tai bisneslogiikassa määritellystä bisnessäännöstä, jota ollaan mahdollisesti rikkomassa. Esimerkkinä lähetetään dataa, jossa on postipaketin vastaanottajan tiedoissa 7-merkkinen postinumero suomalaisella maakoodilla, vaikka kaikki suomalaiset postinumerot ovat 5-merkkisiä. Palvelin tulkitsee tämän virheeksi ja täten ilmoittaa virhetilasta ja hylkää pyynnön järjestelmän vastaamalla esimerkiksi HTTP-tilakoodeilla 400 ”Bad Request” tai 403 ”Forbidden”. Virheviestiin saa muiden HTTP-viestien tavoin lisättyä dataa, joten siihen voidaan esimerkiksi kirjoittaa standardisoitu viestiominaisuus, joka esitetään front endissä käyttäjälle virheikkunassa tai ilmoituksessa.

Back-end on huomattavasti laajempi konsepti käyttötarkoitukseltaan kuin front-end, jonka vuoksi yksittäisten kielien ja kehysten esittely lyhyesti on tarpeettoman lähellä pintaliitoa. Se ei riitä kuvaamaan eri järjestelmiä riittävän selkeästi. Tämän vuoksi back-end -osia ei esitellä samalla kaavalla kuin edeltäviä front-end kirjastoja, vaan niitä käydään läpi korkeammalla tasolla selittämällä erilaisten komponenttien käyttötarkoitukset. Esittelyyn yhdistetään teknologiaa opinnäytetyössä olevasta projektista. Osien esittelyn jälkeen voidaan olettaa riittävä ymmärrys lukijalta sille, että voidaan esitellä arkkitehtuurin suunnittelumallina.

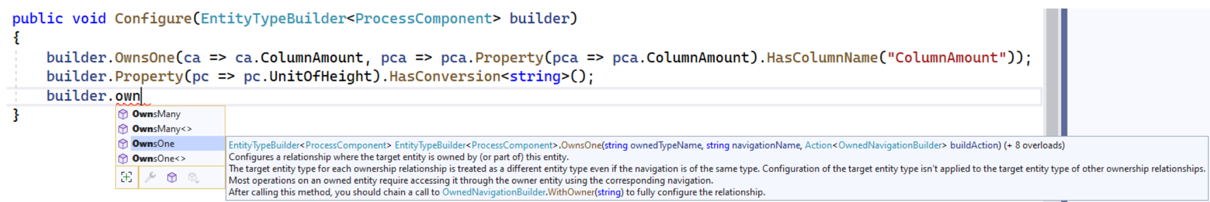
4.6 Back-endin ohjelmointikieli ja alusta

Projektin osalta back-end on tuotettu ASP.NET:illä, joka Microsoftin kehittämä verkkosovellusalusta, ja varsinainen ohjelmointikieli on tuotettu C#-ohjelmointikielellä. ASP.NET on perustellusti hyvä valinta verkkosovelluksen back-endin kehittämiseen monesta eri syystä: Sillä on hyvin suuri yhteisö, .NET tukee useampaa suosittua ohjelmointikieltä ja sille löytyy paljon hyödyllisiä liitännäisiä. C# on myös olio-ohjelmointikielenä helppo oppia, sekä kielen ominaisuudet ovat niin yleiskäyttöisiä, että C# taipuu helposti moneen eri käyttötarkoitukseen. .NET verkkosovelluksia on myös helppo suurentaa, joten isoille ”enterprise” sovelluksille C# on myös kiitettävä valinta. C# ja .NET yleisesti on hyvin työllistävä ympäristö, jonka vuoksi se on hyvä valinta työtä hakevalle kehittäjälle tai opiskelijalle. StackOverflown tutkimus osoittaa, että melkein kolmasosa (ammatikseen alalla työskentelevistä) tekee töitä C#-ohjelmointikielellä (StackOverflow, 2021). C# on myös hyvä ponnahdusalusta Javan, ja C++:n oppimiseen, joiden suosioluvut ovat myös korkealla listalla.

Alustavalinta on perustellusti myös hyvä, sen vuoksi, että C#:ia ja .NET:iä tukevat IDE:t (Integrated Development Environment) sisältävät paljon erilaisia ominaisuuksia, jotka automatisoivat kehittämiseen liittyviä toimintoja. Otetaan esimerkiksi Visual Studio, joka on myös Microsoftin alusta. Visual Studion hyödyllisiin ominaisuuksiin kuuluu monia eri toimintoja, joista esitellään muutama. Visual Studiossa voidaan hyödyntää IntelliSenseä. Sen avulla IDE pyrkii tarjoamaan kontekstiriippuvaisia ehdotuksia siitä, että mitä koodiriville seuraavaksi kirjoitetaan ennakoivana syöttönä. Tämä ennakoiva syöttö ei pelkästään tarjoa vaihtoehtoja, mutta myös pystyy kuvaamaan eri metodien ja ominaisuuksien käyttötarkoituksia. Tämän vuoksi tiedonhankinnan tarve hakumoottoreiden ja

dokumentaation avulla vähenee huomattavasti, sillä samaisen tiedon saa jo IDE:n avulla. Samaisen toiminnon saa myös monesta koodin kirjoitukseen tarkoitetusta tekstieditoristakin, mutta IDE:issä on usein enemmän tiettyyn käyttötarkoitukseen soveltuvia työkaluja jo valmiiksi, kuten tietynkaltaisten sovellusprojektien alustustyökaluja ja edistyneempiä testaus- ja virheenkorjaustyökaluja. Kuvassa 13 on esimerkki siitä, että millaista sisältöä automaattinen täyttö voi tarjota käyttäjälle. Ehdotukset ovat kontekstiriippuvaisia, joten se pystyy esimerkiksi tarjoamaan jonkin ohjelmaluokan sisäisiä arvoja tekstintäyttöön. Luokalle yksityiset ominaisuudet ei esimerkiksi näkyisi ehdotuksissa, jos luokkaa käsitellään luokkainstanssin kautta luokan ulkopuolella.

Kuva 13 Visual Studion IntelliSense tarjoaa valmiin metodin ja kuvailee sen käytön



Visual Studio pyrkii tarjoamaan työkaluja erinäköisiin refaktorointeihin, eli esimerkiksi jos refaktoroinnin ohessa on tarpeen tehdä moduuleja yksittäisestä funktiosta, niin Visual Studiolle riittää se, että oikeat rivit koodista maalataan hiirellä, ja valitaan alasvetovalikosta, nappi, jolla funktio voidaan refaktoroida uuteen funktioon. Kehittäjä säästää huomattavasti aikaa sellaisissa tehtävissä, jotka ei suoranaisesti edistä sovelluksen kehittämistä, vaan ohjelma sallii sellaisten toimien automatisoinnin.

Refaktorointia ja virheenkorjausta voidaan myös tehostaa tekoälyn avulla esimerkiksi käyttämällä Github Copilot -koneoppimistyökalua (Gershgorn, 2021). Samainen tekoäly voi myös kommentoida kehittäjän kirjoittaman koodin jopa esittelemällä käyttöesimerkkejä tietylle funktiolle. Github Copilot on siis koulutettu koneoppimisalgoritmina lukemaan eri kielin lähdekoodia, ja ymmärtää tehdä esim. kommentin perusteella ohjelmakoodia. (GitHub, 2023)

Visual Studiolla pystytään myös melko helposti tekemään moduuleja sovelluksen osa-alueista. Yksittäisen projektin alle voidaan lokeroida sovellus useampaan erilliseen

kirjastoon, jos ohjelma vaatii loogista jakoa. Useamman moduulin sovelluksessa jokainen moduuli voidaan myös toimittaa tai päivittää käyttökohteeseen erillisenä osana helposti, sillä jokainen moduuli kääntyy tämänlaisessa arkkitehtuurissa omaksi dynaamiseksi linkkikirjastoksi tai sovellukseksi. Jokainen moduuli voidaan näin versioida itsenäisesti, tai ajaa eri paikoissa, kuten erillisissä pilviklustereissa keinona tasata palvelun moduulien kuormittamisastetta. ”Microservices Architecture” on esimerkki hyvin aggressiivisesti moduuleiksi pilkotusta arkkitehtuurista, jossa moduuleja tehdään ohjelmasta aina, kun ohjelmakoodin tavoite toimialueena on eri.

Esimerkiksi Microservices Architecturen alle voidaan jossakin ohjelmaprojektissa tehdä erillinen autentikoinnin kirjasto, oma tilausten käsittelyn kirjasto ja oma varaston kirjasto, jotka kääntyvät sitten ajossa erikseen. Microservices projektissa on esimerkiksi helppoa osoittaa tiettyjen moduulien luonnin kehityksen yhdelle tiimille, ja sitten taas jonkin muun moduulin jollekin eri tiimille. Tämänlainen arkkitehtuuri on luonteensa vuoksi tyypillisempi todella isojen kehitysryhmien projekteissa.

4.7 Back-end rakenne pintaliitona

Hyvin suunniteltu back-end tyypillisesti koostuu useammasta moduulista, mutta yleisesti kuvattuna eri moduulien vastuina on seuraavanlaisia: Input/Output kanava, eli käytännössä mistä käyttäjä pystyy tekemään pyyntöjä sovellukselle. Sille yleinen moduulityyppi on API, siinä sovellukselle voidaan tehdä rajapinnan kautta pyyntöjä, jotka tyypillisesti reititetään ”Controller” luokkien kautta.

Controller luokkien päätarkoituksena on siirtää käyttäjän pyyntö syvemmälle järjestelmään. Käytännössä viesti välittyy selaimen saavuttamasta järjestelmän ulkoreunasta palvelimen sisässä oleviin moduuleihin, jonka jälkeen rajapinnasta välitetään viesti takaisin käyttäjälle vastauksena. Logistisesti paras tilanne on se, että ohjelman I/O kanava on mahdollisimman yksinkertainen ja ei tee muuta kuin siirrä dataa ulos ja sisään, sillä sellainen järjestelmä on muutossietoinen, ja sen välittämä bisneslogiikka on helposti yksikkötestattavissa.

Controllereita seuraava taso on sovellustaso, jonka tarkoituksena on ohjata ja suorittaa bisneslogiikan toimia ja validointia. Yksinkertaisimmillaan kyseiseen tasoon kuuluu ohjelman entiteetit (engl. Entity) ja palvelut (engl. Service). Entiteetit ovat projektioita tietokannassa olevista tauluista. Palvelut ovat entiteetteihin liittyviä funktioita. Käytännössä Service nimi ei ole merkitsevä, eikä mitenkään pakollinen, ja samankaltaiselle toiminnalle löytyy useampia eri nimiä, kuten ”Business Entity/Business Layer”, mutta merkitsevä tieto jokaisessa on se, että bisneslogiikka ja validointi mieluiten tapahtuu kyseisissä ohjelmissa, eristyksessä käyttöliittymästä ja tietokannasta.

”Service” nimi yleensä kielii siitä, että sen tarkoitus on orkestroida useamman taulun tai toiminnan yksikön organisoitua toimintaa. Malli eli ”Model” on toinen nimitys, jossa tauluprojektioon on sisällytetty myös datan validointia ja bisneslogiikkaa, mutta ohjelman sisällyttämää dataa voidaan esittää silkkana datana, jolloin mallia voidaan kutsua ”aneemiseksi malliksi”. Data ei silloin tiedä omasta validoinnistaan, vaan sen validointi on delegoitu jollekin toiselle ohjelmalle.

Jos datakommunikaatiota kantaan päin tai kannasta poispäin ei tehdä sovellustasolla, niin sen lisäksi projektissa voi olla usein erillinen infrastruktuuritaso, joka huolehtii tietokantayhteyksistä. Tämä yhteys usein tehdään ORM:in (Object Relational Mapper) avulla, jonka tarkoituksena on siis tehdä kannassa olevasta datasta helposti ohjelmointikoodissa käsiteltävää dataa, jolle voidaan datan lisäksi antaa toiminallisuuksia (engl. behaviour).

Otan englanninkielisen sanan käyttäytymiselle esiin sen vuoksi, että se on hyvä tapa kuvata sitä, että mitä se vaikuttaa, kun datalle luodaan metodien muodossa käyttäytymismalli. Ulkoinen funktio toki voi määritellä sääntöjä sen toiminnalle, mutta se ei varsinaisesti pysty koteloimaan sitä logiikkaa osaksi sitä dataa, eli data ei itse ymmärrä mikä on oikeellista datalle bisnestapauksena. Kontrastina jos datalla on metodeja, jotka määrittävät käyttäytymistä, niin silloin datan validoinnin varmuutta on helpompi turvata, sillä toiminnan säännöt ovat silloin kiinteästi osa dataa. Asiasta kiistellään toki paljon, että kumpi malli on parempi, ja tästä asiasta löytää lisää tietoa sekä vasta-argumentteja etsimällä hakuavaimella ”anemic domain model” tai ”rich domain model”. Olennaista siinä on kuitenkin se, että tyyli valitaan sen mukaan, mikä sopii millekin projektille paremmin.

Syy infrastruktuurin erittelylle ydinlogiikasta läpikäydään myöhemmässä kappaleessa. ORM ei ole kantayhteyden ajuri, mutta se pystyy käyttämään projektissa määriteltyä kantayhteystietoa yhteyden muodostamiseen kantaan. Kantakeskustelu tapahtuu kannan kyselykielellä, mutta ORM:in tuottama data näkyy kehittäjälle objekteina tai olioina, jotka ovat erillisiä kannasta, mutta sisältävät kannan taulujen rivien tiedot, näin ollen niitä voidaan kutsua projektioiksi.

Kaikki nämä osat ovat kuitenkin tarpeen liittää toisiinsa, jotta ne toimisivat. Suora ohjelmakutsu toimii, mutta sellainen ohjelmistorakenne aiheuttaa paljon riippuvaisuuksia ohjelmien välille. Vaihtoehtona suoralle riippuvuuden rakentamiselle koodissa on ”Dependency Injection”. Sen avulla voidaan ohjelman eristää varsinaiset implementaation yksityiskohdat omalle tasolleen tai omaan kirjastoon. Ideana DI:lle on se, että voidaan eliminoida eri ohjelmien välillä olevat riippuvuudet. Toisesta komponentista riippuvainen komponentti on vaarassa hajota, jos sen riippuvuudet muuttuvat, silloin riippuvaistakin komponenttia on tarpeen muuttaa. Näin voidaan myös välttää tilanne, jossa jotakin riippuvuutta on tarve kantaa useamman luokan yli, kunnes se saavuttaa todellisen käyttökohteen. Tarpeeton luokan kantaminen ohjelmasta toiseen luo kantavallekin ohjelmalle riippuvuuden kannettuun ohjelmaan, vaikka ohjelmassa ei tehtäisi mitään toimintoja kyseiselle luokalle.

Ajatuksena DI toimii niin, että injektion kohteena olevat ohjelmat lisätään erityiseen säilöön, josta sitten ne voidaan rekisteröidä esimerkiksi tietyn Interface-tyyppisen luokan implementaatioksi. Kutsuvalle ohjelmalle täten riittää se, että ohjelma tietää implementaation Interface luokasta. Se ei silti tunne sen ohjelman implementaatiota, jolloin kutsuva luokka ei ole enää riippuvainen implementaation yksityiskohdista. Luokan rakentaja (engl. constructor) pystyy silloin tuomaan implementaation käytettäväksi, ilman että pyydetylle ohjelmalle tarvitsee tehdä suoraa viitettä implementaation yksityiskohtiin. Injektion implementaatiota esitellään tarkemmin projektissa.

5 Projekti

Projektin ajatus syntyi minun aloitteestani rakentaa työelämässäni jatkokehittämälle jo hyvin vanhalle sovellukselle moderni rakenne ja toimintatavat. Lähdesovellusta ei voida kuvailla täysin tarkasti, mutta seuraavissa kappaleissa on tarkoitus esitellä yleisesti sellaisia kohtia vanhassa sovelluksessa, jotka modernista näkökulmasta katsottuna kaipaisivat parannusta. Näillä toimilla sovelluksen jatkokehitys olisi kustannustehokkaampaa ja ripeämpää.

Sovelluksen käyttötarkoituksena on tarjota erilaisille logistiikka-alan yrittäjille ratkaisuja, jonka kautta käyttäjät pystyvät dynaamisesti määritellyissä näkymissä ohjata erilaisia logistiikan prosesseja. Tästä esimerkkinä sovelluksessa pystytään hallinnoimaan kuljetustilauksina tehtyjä kuormia, johon voidaan kiinnittää useampi kuljetustilaus. Samalle kuormalle voidaan myös liittää resurssina esimerkiksi puoliperävaunu. Todellisessa lopputuotteessa voidaan lähettää liikennöitsijälle tieto kirjatusta keikasta. Ohjelman avulla kuljetussuunnittelijat voivat esimerkiksi tehdä kustannustehokkaita kuormia sekä vähentää tyhjien vaunujen kuljettamista kuormaamalla reittioptimoituja kuormia. Ilman tämänkaltaisia ohjelmistoja maailmanlaajuisia logistisia ketjuja olisi paljon vaikeampi suunnitella, sekä ne olisivat tehokkuudeltaan paljon heikompia.

5.1 Vanha järjestelmä

Vanhassa järjestelmässä ongelmat liittyvät pääosin järjestelmän vanhuuteen ja sille valittuun sovellusarkkitehtuuriin, jonka rakenne ei paljon oikeasti ohjannut kehittäjää tekemään sovellukseen osia arkkitehtuurin mukaisesti. Rakennetta voidaan kuvailla N-tier arkkitehtuuriksi, jossa N viittaa sovelluksen arkkitehtuurillisten tasojen määrään. Tämänkaltaisessa rakenteessa on tyypillisesti 3 tasoa, josta ylin on esitystaso, keskimmäinen on bisneslogiikkaa ohjaava taso ja alin on säilyvän tiedon taso (engl. persistence layer). Esitystaso viittaa käyttäjälle näytettävään esityskerrokseen, joka voi koostua HTML-sivuista tai työpöytäsovelluksen esityskerroksesta.

Tässä järjestelmässä tämä taso on käytännössä kehitetty SOAP-ohjelmana, joka pysyvästi ajettavien ohjelmien kautta palvelee staattisia HTML-sivuja käyttäjälle. Muut pyynnöt

voidaan rekisteröidä AJAX-pyynnöiksi (Asynchronous JavaScript and XML), johon vastataan palvelimelta tullessa XML-formaatissa suorina JavaScript komennoilla (XML on harhaanjohtava sana lyhenteessä, vastaanottavan pään laitteen ei tarvitse lukea pelkkää XML sisältöä, viestin sisällä oleva data voi olla muuta formaattia). Tämä tarkoittaa sitä, että järjestelmä on sidonnainen verkkotekniikkaan. Se ei sovellu komentorivi- tai mobiilisovellusperusteisiin käyttökohteisiin. AJAX-pyynnöt ovat JavaScriptillä tehtyjä pyyntöjä, jotka lähetetään XML-muotoisen viestin sisällä. Vastaukset palvelimelta esitetään aina komentoina datan sisässä XML alkioina.

Loogisesti suunnitellussa järjestelmässä datan käyttötavasta päättää käyttäjän selain, tai jokin muu esityspinta. Tämän järjestelmän esityskerrosta on haastavaa muuttaa sen käyttämien sovelluskirjastojen vuoksi: selain vaatii vähintään yhden funktion pyynnölle, ja yhden tai useamman funktion palvelimen komentojen tuottoon vastauksena. Nämä funktiot käskytetään ainoastaan palvelimen suunnasta. Sen lisäksi käyttäjäliittymää on ylipäätänsä vaikea muuttaa, sillä jos eri palvelimen funktiot luovat HTML-elementtejä käyttöliittymälle komentojen kautta, muutoksen tarpeessa saman HTML-elementin ilmentymät pitää muuttaa jokaisesta sijainnista lähdekoodissa, joka on monesti logistisesti mahdoton teko.

Bisneskerros tässä ohjelmassa on suhteellisen häilyvä konsepti. Sen toimintoja on helppo löytää myös esityskerroksesta palvelimen rajapinnan reunasta, sillä ohjelman arkkitehtuuri ei estä eikä ohjaa käyttäjää rakentamaan logiikkaa irrallisena moduulina, jolloin esityskerroksen ja bisneslogiikan kerrokset ovat tiukasti nidottuna yhteen. Tiedon säilyvyyden tasolla on samanlainen ongelma: bisneslogiikan kerros on täysin riippuvainen kantayhteyden implementaatiosta, jolloin kantakerroksen ohjelmointikehystä on haastava vaihtaa.

Koko ohjelma on myös tuotettu käyttäen proseduraalista ohjelmointia, jonka vuoksi ohjelma on näkyvästi datakeskeinen, operoidulle datalle on vaikeaa osoittaa ”omistettua logiikkaa”. Proseduraalinen ohjelmointi eroaa luokkapohjaisesta ohjelmoinnista niin, että proseduraalisessa ohjelmoinnissa ohjelmarakenne käytännössä jaetaan funktioihin tai proseduureihin, jotka sitten voivat käsitellä noudettua dataa, mutta olio-ohjelmoinnissa data sekä sen säännöt voidaan yhdistää yhdeksi toiminnallisuuden yksiköksi. Tämä sallii sen, että data voi validoida itse itsensä.

Proseduraalisessa ohjelmoinnissa validointi pitää suorittaa erillisen transaktioskriptin kautta, tällöin ohjelmakoodi helposti kopioituu useampaan eri paikkaan. Kopioitua lähdekoodia on vaikeampi ylläpitää, sillä koodissa tehdyt muutokset pitää silloin tehdä useampaan paikkaan. Skriptien tekemät toimintasäännöt voidaan jopa kiertää niin, että niitä skriptejä ei ajeta. Käytetty palvelimen lähdekoodin kieli on vanhentumassa (vanha ei tarkoita huonoa, vanhentunut on tärkeämpi huomio!) sekä kieli pienenee käyttäjäkunnaltaan vuosittain, jonka vuoksi tulevaisuuden rekrytointi kehittäjäryhmään vaikeutuu koko ajan. Kielelle on myös haastava löytää sellaisia edelleen kehityksessä olevia kirjastoja ja liitännäisiä, josta olisi ohjelmassa hyötyä, sillä pienen käyttäjäkunnan ohjelmointikielille ei tehdä tai ylläpidetä kovin paljon liitännäisiä.

5.2 Uusi back-end konsepti

Uuden järjestelmän konseptina on muuttaa koko järjestelmä perinpohjaisesti: Ohjelmointikieli vaihtuu vanhasta proseduraalisesta kielestä C#:iin, ja ohjelmarakenne vaihtuu olio-ohjelmoiduksi. Etuna on se, että monimutkaisia rakenteita on helpompi tuottaa: datan validoinnin säännöt voidaan nidota osaksi ohjelmassa käsiteltyjä olioita, jolloin data ja sekä sen toiminta on yhdessä paikassa Datan sääntöjä ei voida kiertää suoralla tietokentän manipuloinnilla.

Jos tässä ohjelmassa tarvitaan hyvin samankaltaisia entiteettejä eri käyttötarkoituksilla, niin ne voidaan luoda erillisinä luokkina. Tämä tarkoittaa toki sitä, että osa koodista monistuu, mutta siinä samalla vältetään esimerkiksi yleiset koodin perimisestä aiheutuvat ongelmat iteroituvasti kehitetyssä koodissa. Esimerkkinä kuljetustilausta voidaan tarvita kuljetuksen suunnitellussa, mutta myös esimerkiksi rahdituksessa sekä tullauksessa, suurin piirtein samoilla ominaisuuksilla, mutta hieman eroavin parametreillä. Silloin kannassa säilyvä taulu paisuu holtittomasti kenttämäärältään. Sellaisen taulun käyttötarkoitus leviää liian laajalle järjestelmään, jonka vuoksi siihen on vaikeaa tehdä johdonmukaisia kaikkialle sopivia validointia koskevia muutoksia. Sen lisäksi järjestelmässä dataa vasten ajetut bisnessäännöt kadottavat alkuperäisen aikeensa, sillä ne vaikuttavat nyt useampaan eri bisnestapahtumaan.

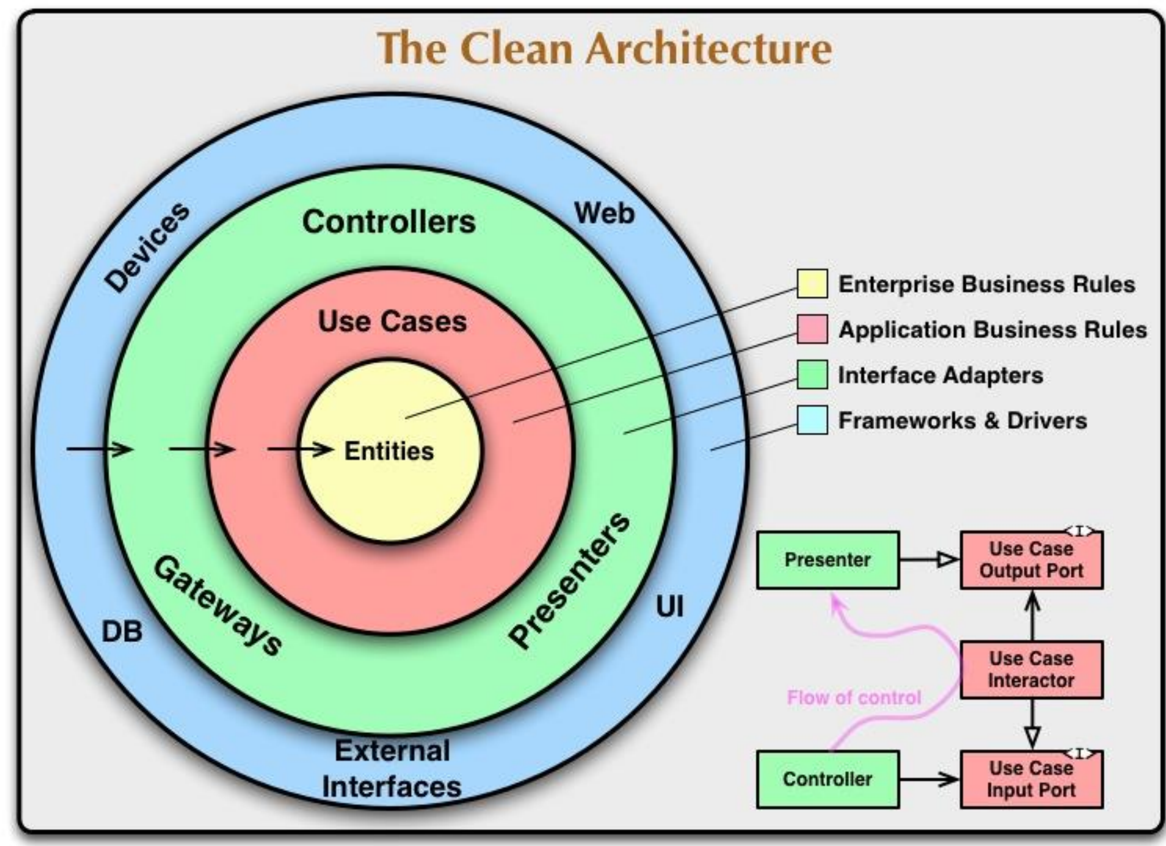
Monistuva koodi on pienempi paha, joten se sallitaan harkituissa tilanteissa. Tätä pienempää pahaa saa hillittyä ”perimällä” koodin esimerkiksi kompositiorakenteina (engl. Composition over Inheritance), missä yksi datarakenne voi koostua useasta kehittäjän määrittelemästä luokasta, ilman että kyseisiä luokkia pakosti peritään. Vastakkaisessa tapauksessa koodin eheyden heikentymistä koodin iteroituessa ei välttämättä voida estää samalla tavalla.

C#-ohjelmointikielessä sallitut abstraktion keinot, kuten Interface-luokat sallivat hyvin löyhästi yhteen nidotun koodin tuottamisen. C# on tyyppiturvallinen ohjelmointikieli, jonka vuoksi järjestelmän sisäisiä objekteja ei voida käsitellä yhtä dynaamisella tavalla, mutta tässä on hyväksi havaittuja seikkoja. Virheentunnistus on huomattavasti tehokkaampaa, ja koodi on optimoidumpaa, mutta koodin joustavuus heikkenee. Liiallinen joustavuus toki saattaa kannustaa ryvettyneen tai epäselkeän koodin tekemiseen. Taitavalle C# kehittäjälle tämä ei ole ongelma, sillä dynaamisen koodin ongelman saa ratkaistua jo mainittujen Interface luokkien avulla.

Sovellusarkkitehtuuri muutetaan mallintamaan ”Clean Architecturea” (Martin, 2017). Konseptiltaan koko arkkitehtuuripohja ei ole mitään uutta, sillä sitä vastaavia rakenteita on ollut olemassa jo kauan ennen koko termin olemassaoloa muun muassa nimillä ”Ports and Adapters”, ”Hexagonal” ja ”Onion Architecture”. ”Clean Architecture” on yksinkertaisesti aggregoitu käsite edellä mainittujen arkkitehtuurien ajatuksista, eikä itsessään tuo niin paljon mitään uutta samaiseen konseptiin, mutta on kuitenkin tämänkaltaisesta arkkitehtuurimallista tuorein iteraatio.

Clean Architecture -pohjaan kuuluu vaihteleva määrä kerroksia, mutta tyypilliseen malliin kuuluu neljä kerrosta. Näistä uloin kerros on ulkoiset yksityiskohdat (infrastruktuuri), kuten ohjelmistokehykset, ajurit sekä verkko ja käyttöliittymä. Tämä kerros on käytännössä kaikki järjestelmässä kaikista uloimpana oleva. Mikään muu kerros ei ole riippuvainen sen toiminnasta, jolloin sitä koskevat muutokset eivät vaikuta muihin kerroksiin. Sen käyttämät sovelluskehykset ja käyttöliittymät on helppo vaihtaa, sillä sen implementaatio ei vaikuta syvempiin kerroksiin. Tyypillinen tapa esitellä Clean Architecturen sovellusarkkitehtuurimallia on esitelty kuvassa 14.

Kuva 14 Clean Architecture arkkitehtuurin kaavio. (Martin, 2012)



Tässä projektissa ulkoisiin detaljeihin kuuluu verkkoselainsovellus (front-end, sekä sitä tarjoava verkkopalvelu, kuten Apache HTTP server tai Nginx, vaikkei se palveluna esiinny varsinaisesti koodissa, sitä käytetään alustana, ja käyttöön se vasta tulisi tuotantoympäristössä) sekä MySQL tietokantapalvelin, josta noudetaan dataa infrastruktuuritasolle Entity Framework ORM:in avulla. Tyypillisesti tämänlaiselle tasolle voidaan myös tehdä sähköpostille ja tulostimen ajurit ja lähetteet. Jos käyttäjä haluaa esimerkiksi jostakin tietoryhmästä Excel tiedoston, niin Excelin kanssa yhteensopivan .xls tiedoston voisi muodostaa infrastruktuurin ohjelmissa, mutta dataa koskeva käsittely olisi mahdollisesti järjestelmässä syvemmällä.

Seuraava kerros on karkeasti suomennettuna liittymien adapterit. Sen ajatuksena on muodostaa syötteestä pyyntöjä sopivassa formaatissa syvemmälle kerrokselle, sekä se myös pyrkii muodostamaan syvemmän kerroksen vastauksista sopivia vastauksia tulosteena. Tämä kerros ei ole siis tietoinen sovelluksen funktionaalisuudesta tai bisneslogiikasta, sen

tehtävänä on vain siirtää pyyntöjä ja vastauksia. Tästä kerroksesta esimerkkinä toimii sovelluksen API.

Seuraava taso käsittelee sovelluksen käyttötapauksia. Sen tarkoituksena on sisältää kaikki orkestraatio ohjelman prosesseille. Tämä kerros pystyy käyttämään sisimmän kerroksen luokkia muodostamaan loogisia toimintaketjuja, jossa käsitellään useampaa eri entiteettiä. Tästä esimerkkinä verkkokaupan käyttötapauskerros pystyisi tuotteen hinnan alentuessa orkestroimaan tapahtuman, jossa tuotteelta ensiksi vähennetään hintaa, ja jonka jälkeen voidaan lähettää hintaseurannan tilanneille sähköpostiviesti tarjoushinnasta. Tämä prosessi olisi tietoinen vain sähköpostijärjestelmän liittymistä, mutta ei implementaatiosta. Implementaatio määriteltäisiin infrastruktuuritasolla, joka on taas järjestelmän uloin taso. Tässä projektissa tasoon kuuluu erilaiset liiketoiminnan prosessit, kuten kuljetustilausten hallinnan toiminnallisuus.

Sisin taso sisältää sovelluksen datan entiteetit ja niitä keskeisesti koskevat validointi- ja bisnessäännöt. Ne voi olla tyypiltään pelkkiä dataa koteloivia luokkia, tai sitten olioita, joilla on dataa, mutta myös toiminnallisuutta metodien muodossa. Niiden ajatuksena on koteloida yleiset ohjelman bisnessäännöt, ja niillä ei ole riippuvaisuuksia uloimmilla tasoilla. Tämä tarkoittaa sitä, että ne ovat POCO:ja (Plain Old CLR Objects). POCO:n tunnistaa siitä, että se ei sisällä kolmannen osapuolen riippuvuuksia.

Ytimenä toimivaa lähdekoodia käyttää jossakin toisessakin ohjelmassa, ilman että lähdekoodiin tarvitsee koskea. Tämä johtuu siitä, että sillä ei ole ulkoisia riippuvuuksia. Bisneslogiikan yksikkötestaus on tällä tasolla helppoa, sillä yksikkötestauksessa ei tarvitse tehdä viitteitä muihin ohjelmiin. Entiteettien lisäksi ydintasolta löytyy Interface-luokkia, jonka avulla voidaan ulkoisien riippuvuuksien avulla hakea esimerkiksi dataa tietokannasta. Ydintaso ei kuitenkaan tunne Interface-luokkien todellista implementaatiota.

Hyötyjä rakenteelle on monia: Riippumaton rakenne tarkoittaa sitä, että järjestelmää ei sidota tiettyyn ohjelmistokehykseen. Ohjelmistokehystä tällöin käytetään ulkoisena työkaluna, eikä järjestelmää määrittelevänä moduulina. Tämä järjestelmä on myös tietokantariippumaton, sillä tietokantaa käsitellään tällöin ulkoisena yksityiskohtana.

Tietokantatyypin tai sen päällä olevan kehyksen saa vaihdettua vaivattomammin. Järjestelmä on käyttöliittymäriippumaton, käyttöliittymän saa vaihdettua, ilman että järjestelmään tarvitsee käydä syvästi käsiksi, eikä muutokset käyttöliittymän toiminnassa vaikuta sovellukseen sisäisesti. Yksikkötestaus järjestelmässä on yksinkertaisempaa, sillä moduuliriippumattomia luokkia on helpompi yksikkötestata.

Yksityiskohtien abstraktion tarvetta voidaan pohtia näin: Kehysten ja tietokantojen kehittäjät (eli kaikkien yksityiskohtina toimivien työkalujen kehittäjät) aina pystyvät vaikuttamaan sovellusprojektiin negatiivisesti. Ohjelmistokehys voi muuttua jotenkin radikaalisti, jolloin sitä käyttäviä ohjelmia täytyy muuttaa, tai sen ylläpito voi loppua. Ohjelmistokehysten kehittäjät ovat tämän vuoksi luontaisesti epäluotettavia toimijoita. Kehyksen käytöstä aiheutuvaa riskiä voidaan vähentää rajaamalla siihen tehtyjen viittauksen määrää tiettyihin moduuleihin. Jos siihen kehykseen tai työkaluun tehdyt viittaukset ovat minimoitu, sen vaihtaminen johonkin toiseen työkaluun on vaivattomampaa. Ohjelmistokehyksen muutokset vaikuttavat vain muutamaan ohjelmaan, joskus jopa vain yksittäiseen tiedostoon, vaikka kyseistä ohjelmaa käytettäisiin joka puolella projektia.

Todellisuudessa mallin kerrokset ovat vaan mekanismi selittämään rakenteesta kaikista tärkeimmän tekijän, joka käytännössä toimii koko arkkitehtuurin nyrkkisääntönä: ”engl. The Dependency Rule” (Martin, 2017). Dependency Rulen mukaan jokaisen kerroksen riippuvaisuudet saavat osoittaa vain ja ainoastaan järjestelmässä ”sisäänpäin”. Sisemmät kerrokset eivät saa siis tehdä viittauksia ulompiin kerroksiin. Uloimmilla kerroksilla tapahtuvat muutokset eivät näin vaikuta sisempiin kerroksiin mitenkään. Tämä saadaan saavutettua Dependency Injectionin ja Interface-luokkien yhdistelyllä: Sisemmät kerrokset pystyvät määrittelemään liittymiä datan tallennukselle, mutta sen implementaatio määritellään vasta infrastruktuuritasolla. Interface luokilla voidaan määrittää toiminnalle vaan sopimus, jonka joku muu ohjelma täyttää implementaatiolla. Implementaatioyksityiskohtia voidaan hyödyntää DI-säilöllä. DI-säilö voi yksinkertaisesti noukkia sen implementaation toisen ohjelman käyttöön, jos se on rekisteröity DI-säilössä kyseisen Interface-luokan implementaatioksi. Kutsuva ohjelma ei siis tiedä ennen ajoa, että mikä ohjelma on kyseessä, kun toimintoa suoritetaan, mutta implementaatio on silti täysin

kehittäjän määriteltävänä eksplisiittisesti. Ohjelma toimii vaikka suoraa viittausta ei muodostu ohjelmien välille.

Hyvin suunniteltu Clean Architecture -järjestelmä ei edellytä tarkkaa moduulien nimeämistä tai tiettyihin lohkoihin ohjelman viemistä. Hyvä arkkitehtuuri edellyttää sen, että ohjelmasta toiseen määritellyt viitteet kulkevat aikaisemmin esitellyssä loogisessa järjestyksessä. On tehokkaampaa miettiä sovelluksen käyttäytymistä bisnestoimintana ja jatkokehitettävänä projektina hyvin tarkasti nimetyn arkkitehtuurin sijaan. Käytännössä koko Clean Architecture kirjan 300 sivun sanoman voi kiteyttää seuraavaan lauseeseen: ”Irrota bisneslogiikka muista järjestelmän ulkoisista yksityiskohdista ja kolmannen osapuolen riippuvuuksista” (Martin, 2017)

5.3 Uusi front-end konsepti

Uusi front-end on rakennettu hyödyntämään API:n etuja. Uusi käyttöliittymä toimii samalla tavalla, kuin aikaisemmin esitetty Verkkokauppa.com sivun esimerkki. Sen sijaan että käyttöliittymä on suoraan käskytettävä selainpohjainen staattinen verkkosivu, se on Vue-sovellus, joka renderöidään yksittäisen verkkosivun indeksisivulle SPA:na. Tällä front-end-sovelluksella voidaan irrottaa palvelimella olevat riippuvuudet käyttöliittymästä.

Data siirretään palvelimelta käyttäjälle JSON-datana, jolloin ainoastaan esityskerros on vastuussa datan esittämisestä. AJAX-kutsuja ei reititä enää suorina funktiokutsuina palvelimelle, vaan ne suoritetaan suoraan front endin pyyntöohjelmassa API:n päätepisteiden kautta. Vuessa suoritetaan tämä toiminto käyttämällä hyödyksi Axios selainkirjastoa, jonka avulla voidaan suorittaa http-pyyntöjä palvelimelle.

5.4 Uuden front-endin kehitys

Front-endin kehityksen aloittaminen on melko yksinkertaista, asennus vaatii installaation Node.js:tä, jonka jälkeen voidaan seuraavalla komennolla suorittaa Vuen virallinen pohjustustyökalun kautta projektin aloitus. Kuvassa 15 suoritetaan kyseinen alustus komennolla npm init vue@latest. Npm tarkoittaa Node Package Manageria, jonka avulla

voidaan ajaa komentorivillä erilaisia projektiin liittyviä skriptejä, suorittaa yksikkötestejä ja ladata projektiin liittyviä riippuvuuksia. init komento alustaa uuden projektin, tässä tapauksessa sillä alustetaan Vueen perustuva projekti, mutta sillä voi alustaa useita muitakin projektityyppejä. vue@latest hakee tietosäilöstä uusimman julkaistun version halutusta npm-paketista.

Kuva 15 Visual Studio Coden komentorivikaappaus

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Github\Web\Cargo-Frontend> npm init vue@latest
```

Projektin automaatioluonnin jälkeen ollaan valmiita kirjoittamaan front-endin lähdekoodia, mutta ennen sitä, on tarpeellista asentaa moduuleita, jota projekti käyttää. Esimerkkinä SPA:ta varten tarvitaan vähintään vue-router-kirjasto, jonka tehtävänä on reitittää SPA:n reititysnäkymän eri reittejä. Muiden tarpeellisten riippuvuuksien asentamisen jälkeen lopullinen riippuvuuslistaus näyttää kuvan 16 mukaiselta tilanteelta.

Kuva 16 package.json-tiedoston sisältö ”dependencies”-kohdassa

```
"dependencies": {
  "@fortawesome/fontawesome-svg-core": "^6.1.2",
  "@fortawesome/free-brands-svg-icons": "^6.1.2",
  "@fortawesome/free-regular-svg-icons": "^6.1.2",
  "@fortawesome/free-solid-svg-icons": "^6.1.2",
  "@fortawesome/vue-fontawesome": "^3.0.1",
  "ag-grid-community": "^29.0.0",
  "ag-grid-enterprise": "^29.0.0",
  "ag-grid-vue3": "^29.0.0",
  "axios": "^0.27.2",
  "bootstrap": "^5.2.0",
  "bootstrap-vue-3": "^0.2.8",
  "jquery": "^3.6.2",
  "pinia": "^2.0.17",
  "pinia-plugin-persistedstate-2": "^1.0.1",
  "vee-validate": "^4.6.6",
  "vue": "^3.2.37",
  "vue-router": "^4.1.3",
  "vue-toastr": "^3.0.5"
},
```

Vue-sovelluksen autentikoinnin tilan seuraamista varten on parempi asentaa State-management kirjasto. Tämän avulla voidaan nostaa yleinen, ei-komponenttiriippuvainen datanhallinta objektille, jolle on pääsy kaikkialta sovelluksesta, näin voidaan tehdä käyttäjän kirjautumistilan tarkistuksesta yksinkertaista.

Ilman state management-kirjastoa kirjautumistilan tarkistus syvällä Vuen komponenttihierarkiassa olisi haastavaa, sillä tietoa pitäisi tuoda ”prop drilling” tekniikalla. Prop drilling tarkoittaa sitä, että prop-tietoa kuljetetaan komponenteissa, jossa tietoa ei käytetä, mutta prop-tiedon pitää silti saavuttaa jokin lapsikomponentti, joka sitä tietoa tarvitsee. ”Prop drilling” on mieluiten vältettävä antisuunnittelumalli front-end -sovelluskehyksellä tehtävässä kehityksessä.

Package.json tiedostosta myös nähdään muutama ajettava skripti. Näiden komentojen ajaminen suorittaa niiden määrittelemät komentosarjat komentorivillä. Kuvassa 17 skriptikomento **dev** on kehittämiselle olennaisin. Se käynnistää Vite kehityspalvelimen, jossa voidaan paikallisesti testata sivua, jopa niin, että sivun elementit päivittyvät ajossa, jos sivun HTML-sisältöä muutetaan koodista.

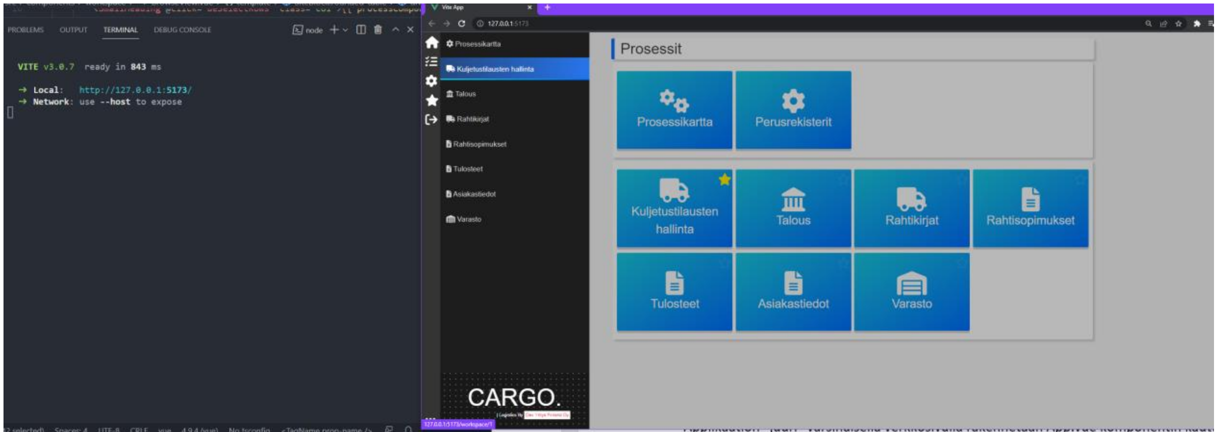
Kuva 17 ”scripts”-osio package.json-tiedostosta

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview --port 4173"
},
```

Kuvassa 18 on esitelty tilannetta, kun kehityspalvelin on käynnistetty. Vasemmalla on komentorivi, jossa **npm run dev** komennon ajo käynnistää palvelimen, joka on palveltu selaimelle oikealla puolella kuvankaappausta. Kehitystila on luontaisesti parempi paikka kehittää kuin tuotantotila, sillä tuotantotilassa Vue käännetään pakatuiksi tiedostoiksi, mutta kehitystilassa näkymää voi kirjoittaa koodia niin, että muuttuneet komponentit virkistyvät

selainnäkymässä reaaliaikaisesti. Kehitystilassa Vue ajaa itsensä palvelimena kehittäjän määrittelemästä paikallisen verkon portista palveluna.

Kuva 18 Esimerkkiajo sovelluksesta



5.5 Näkymän komponentit

Vuen näkymät ovat komponenttiperusteisia, seuraavaksi projektin rakennetta esitellään näkymän komponenttien avulla. Kuvassa 19 on esitelty pätkä ohjelman sisääntulopisteenä toimivasta main.js tiedostosta. Siinä konfiguroidaan Vuen lisäksi Axios, vue-router ja font-awesome -ikonipaketin Vue-moduuli ja Bootstrap -tyylikirjaston Vue-moduuli.

Kuva 19 Main.js tiedoston viimeiset rivit

```
//Configure axios for requests
const axiosInstance = axios.create({
    withCredentials: true
})
app.provide('Axios', axiosInstance);


app.use(BootstrapVue3);
app.use(VueToastr);
app.use(router);
app.component('font-awesome-icon', FontAwesomeIcon);
router.isReady().then(() => {
     app.mount('#app');
     window.$ = $;
})
```

Ohjelman tarkoituksena on ladata Vue-sovelluksen riippuvuudet osaksi muodostuvaa sovellusta, sekä konfiguroida konfiguroinnin tarpeessa olevat moduulit. Ohjelman viemäröinnin lopuksi sovellus ”asennetaan” (engl. mount) osaksi verkkosivua **<div>** elementtiin, jonka tunnisteena on ”app”. Kuvassa 20 nähdään millaista tulosta Vue tuottaa selaimen HTML-ulostuloon. Kuvan **<div>** elementti, jonka **id** on **app** on Vuen HTML-sisältö. Sen lisäksi sen alla oleva **<script>** sisältää Vuen varsinaisen ohjelmalogiikan, joka on käännetty kehityspalvelimelle projektin sisältämien moduulien perusteella. Tätä skriptiä ei siis kehittäjä itse kirjaa, vaan Vue muodostaa sen automaattisesti.

Kuva 20 Verkkosivun lähdekoodi selaimesta katsoen

```
<!DOCTYPE html>
<html lang="en">
  <link type="text/css" rel="stylesheet" id="dark-mode-custom-link">
  <link type="text/css" rel="stylesheet" id="dark-mode-general-link">
  <style lang="en" type="text/css" id="dark-mode-custom-style"></style>
  <style lang="en" type="text/css" id="dark-mode-native-style"></style>
  <style lang="en" type="text/css" id="dark-mode-native-sheet"></style>
 ▶<head>…</head>
 ▼<body cz-shortcut-listen="true">
   ▼<div id="app" class="app" style="min-height: 100vh;" data-v-app> flex == $0
     ▶<div class="app" data-v-7a7a37b1 style="min-height: 100vh;">…</div> flex
    </div>
    <script type="module" src="/src/main.js"></script>
   ▶<div data-v-app>…</div>
  </body>
</html>
```

Sovelluksen juurielementti verkkosivulla rakennetaan App.vue komponentin kautta. App.vue sisältää sivun navigaatioelementit, sekä muutamia yleiskäyttöisiä elementtejä, kuten latausikonin, joka ilmestyy ruudulle uutta sivua ladatessa. Latausikonin näkyvyys perustuu sivun tilaan state managementin kautta, jonka vuoksi muiden komponenttien ei tarvitse kuin vaikuttaa latausikonin käyttämään **isLoading** tietoon. Sivulla oleva <RouterView> komponentti on vue-router-kirjaston määrittelemä komponentti, jonka avulla voidaan esittää sille määrätyn elementin sisällä sivun kaikkia alareittejä.

Kuvan 21 lähdekoodista huomataan, että sivun varsinainen HTML on pinnallisesti yksinkertainen. Koko sivun lopulta määrittelevät komponentit itse, jotka tuodaan App.vuelle muista .vue tiedostoista. App.vueta ei tarvitse siis muuttaa, jos sivupalkkia pitää viilata, muutos onnistuu sivupalkin määrittelemän tiedoston sisällä. Vue komponentteja ei näe sellaisenaan sen jälkeen, kun sivu käännetään selaimella luettavaksi, sillä Vue-komponentit eivät ole selaimen ymmärtämää sisältöä.

Vuen syntaksia käyttävän kuvan 21 **App.vue** tiedoston sisältö on kuitenkin selaimelle kelpoa sisältöä. Pintapuolisesti koodi on siis varsin erilaista kuin mitä selaimella nähdään, sillä Vue koodi käännetään JavaScript paketiksi. Tämä ei toki paljon kehittäjää haittaa, sillä selaimella tehty testaus on Vueta käyttäessä todella intuitiivista, jos käytetään avuksi selaimille tehtyjä Vuen kehittäjätyökaluja. Varsinaista HTML-sisältöä tarvitsee hyvin harvoin tutkia, sillä kehittäjän rakentama ohjelmakoodi koskee pääosin sivun dataa, eikä sen rakentamista HTML-elementeillä, joten esimerkiksi Vuelle tehdyt selainliitännäiset riittävät hyvin virheiden korjaukseen kehityksen aikana.

Kuva 21 Pätkä App.vue ohjelmasta.

```
<template>
  <div class="app"
      :class="{ 'prevent-coolness': UserSettingsStore.preventCoolAnimations }"
      style="min-height:100vh">
    <Notifications></Notifications>
    <div v-if="loading" class="loading">
      <div class="loading-element" style="position:sticky; z-index:9999; top:40%;">
        <div class="loader"></div>
      </div>
    </div>
    <div style="margin-left:45px" >
      <RouterView :key="$route.path" />
    </div>
    <Confirm ref="popup"></Confirm>
    <Prompt ref="prompt"></Prompt>
    <GeneralModal ref="generalModal"></GeneralModal>
    <ContextMenu ref="contextMenu"></ContextMenu>
    <Footer></Footer>
    <Sidebar></Sidebar>
  </div>
</template>
<script>
import { inject, ref, provide } from 'vue';
```

```
import Sidebar from './components/main/Sidebar.vue';
import Footer from './components/main/Footer.vue';
import Notifications from './components/main/Notifications.vue';
import Confirm from './components/general/Confirm.vue';
import GeneralModal from './components/general/GeneralModal.vue';
import ContextMenu from './components/contextmenu/ContextMenu.vue';
import Prompt from './components/general/Prompt.vue';
export default {
  name: 'Cargo-App',
  el: '#app',
  computed: {
    loading() {
      return this.UserStore.isLoading;
    }
  },
  mounted: function () {
    document.body.addEventListener('error', (error) => this.onError(error));
    document.body.addEventListener('success', (success) => this.onSuccess(success));
  },
```

Kuvassa 22 prosessikarttamoduulin lähdekoodilla voidaan esitellä dynaamisten elementtien muodostamista. Näkymässä käytetään Vuen **v-for** syntaksia, jonka avulla voidaan rakentaa Vuen HTML syntaksin sisälle ohjelmoinnille tyypillisiä **for-**silmukoita. V-for attribuutin ensimmäinen symboli merkitsee ohjelmointikielien tyypillisen for silmukoiden iteroitua tietoa. Tässä esimerkissä iteroitava tieto on päävalikosta löytyvät prosessit. Vuen v-for rakenne on lähempänä for each silmukkaa, jossa iteroitava kohde ei ole indeksi tai numero, vaan silmukan kohteena olevat objektit, mutta siinä voidaan toki käyttää indeksiäkin hyödyksi pilkuttamalla ensimmäisen symbolin jälkeen toinen mielivaltaisesti nimetty symboli, esimerkiksi v-for=”(obj, index) in arr”.

Kuva 22 Prosessikarttanäkymän lähdekoodia

```
<SiteBlock class="row mx-1 px-2">
  <MenuBlock v-for="process in mainMenuProcesses" class="col-3">
    <ProcessMapProcessLink :process="process"
     :link="{ name: 'workspace', params: { id: process.id } }">
    </ProcessMapProcessLink>
  </MenuBlock>
</SiteBlock>
```

Näkymälle on määritelty Vuen **setup()** funktiossa muuttuja, joka sisältää kaikki päävalikossa näkyvät prosessit. Muuttuja määritellään tyhjänä taulukkona, joka sitten syötetään Vuen

reactive funktiolle. Reactive funktio pystyy luomaan mistä tahansa JavaScript objektista reaktiivisen kopion. Jos reaktiivisen kopion arvoja muutetaan sovelluksen ajon aikana, niin silloin näkymä päivittyy vastaamaan muuttuneita tietoja. Reactive() funktion tilalla voidaan myös käyttämään ref funktiota, jos esimerkiksi käsiteltävänä on primitiivinen tieto, kuten binäärinen totuusarvo tai numeerinen tieto, mutta reactive() funktiota ei voida käyttää primitiivisen muuttujan kanssa.

Komponentit HTML-koodissa pystyvät käyttämään setup() funktion palauttamia reaktiivisia muuttujia ja prop-tietoja näkymässä käyttämällä v-bind attribuutin liitettä. Sivulla näkyvään tekstiin saadaan muuttujan arvo käyttämällä niin sanottua viiksisyntaksia, eli tupla-aaltosulkuja prop- tai muuttujanimen ympärillä. Jos jollekin HTML-elementille halutaan lisätä HTML attribuutti, jonka arvo perustuu prop-tietoon, attribuutin määrittelyssä pitää käyttää ”v-bind:” etuliitettä, joka voidaan lyhentää myös pelkästään ”:” merkiksi. v-bindin sisältö ymmärtää heittomerkkien sisällön aina muuttujiksi, joten tekstin interpolaatio v-bindissä vaatii JavaScriptin standardin mukaisen ”Template literalin” hyödyntämisen gravis (`) erikoismerkeillä.

Kuvassa 23 esitellään ref funktio setup() funktion sisällä. Huomion arvoinen asia ref funktiossa on se, että sen arvoa täytyy käsitellä .value muuttujan kautta. HTML-koodissa ref funktion arvoa voidaan käsitellä suoraan viittaamalla muuttujan nimeen ilman .value ominaisuutta (mutta skriptin sisällä ei!). Reactive funktiolla saa tehtyä siis hieman siistimmän ja johdonmukaisen näköisen koodin, mutta ref() funktion kanssa voidaan käyttää myös primitiivisiä arvoja. Niitä voidaan käyttää samanaikaisesti, mutta kannattaa käyttää sitä tapaa, joka tuntuu omasta mielestä luontevalta. Molemmilla funktioilla voidaan nimittäin saavuttaa sama tavoite.

Kuva 23 Pätkä kirjautumisruudun setup() funktiosta. Siinä esimerkkejä ref() funktiolla alustetuista muuttujista.

```
export default {
  setup(props, ctx) {
    const userName = ref("");
    const password = ref("");
    const loggingIn = ref(false);
    const UserStore = inject("UserStore");
```

```
const Axios = inject("Axios");
const router = useRouter();
const errorMessage = ref(null);
async function login() {
  if (userName.value == '' || password.value == '') {
    errorMessage.value = "Syötä tarvittavat tiedot";
    return;
  }
```

API-yhteys ja sen funktionaalisuus front-endissä voidaan suorittaa Vue komponenteilla. Otetaan käyttöesimerkiksi sovellukseen kirjautuminen. API-yhteyttä varten injektoidaan Vuen provide funktion kautta määritelty instanssi Axios-kirjastosta, jonka avulla muodostetaan API-kyselyitä. Provide ja inject ovat funktioita, jonka kautta voidaan tuoda useammassa paikassa tarvittuja riippuvaisuuksia suoraan halutulle komponentille inject funktion säilöstä, ilman että niitä tarvitsisi prop-tietoina siirrellä pitkin komponentteja, jotka eivät itse tarvitse niiden tietoja (aikaisemmin mainittu ”prop drilling” on kyseessä). Axioksen injektio sallii Axios-kirjaston käytön komponentissa esimerkiksi kirjautumisessa kuvan 24 funktiolla.

Kuva 24 Login funktio

```
async function login() {
  if (userName.value == '' || password.value == '') {
    errorMessage.value = "Syötä tarvittavat tiedot";
    return;
  }
  errorMessage.value = null;
  loggingIn.value = true;
  UserStore.setLoading(true);
  this.Axios.post("api/UserSession/login", {
    userName: this.userName,
    password: this.password,
  }).then(function (response) {
    UserStore.setToken(response.data.token);
    UserStore.setUser(response.data.userName);
    router.push({ name: "home" });
  }).catch(function (response) {
    errorMessage.value = response.response.data.Message
  }).finally(() => UserStore.setLoading(false));
  loggingIn.value = false;

}
```

Tämä funktio käyttää Axiosta tekemään back-endille HTTP POST pyynnön, joka sitten reititetään back-endin REST API:n kautta autentikointipalveluun. Palvelu tarkistaa käyttäjän syöttämän tiedon vertaamalla tietokannassa olevaa tarkistetietoa niin, että vastaako käyttäjän syötteeseen. Jos tarkiste on oikeellinen, niin käyttäjä kirjataan sisään. Sisään kirjautuminen tapahtuu niin, että Pinialla määriteltyyn UserStore säilöön kirjataan tieto käyttäjän ”token”-tiedosta, sekä käyttäjänimestä, joka on yksilöivä. Samassa säilössä oleva loggedIn laskennallinen arvo lasketaan täten totena olevaksi arvoksi, jolloin voidaan olettaa käyttäjän kirjautuneeksi.

”Tokenia” käytetään sitten API:ssa niin, että käyttäjän Axios instanssi lähettää tokenin jokaisen pyynnön mukana, jolloin API:ssa voidaan arvioida käyttäjän autentikoinnin tila. Lopuksi ”$router” muuttujan avulla voidaan käskyttää vue-routeria siirtämään käyttäjän kirjautumisen jälkeen etusivulle. Virhetilassa sen sijaan annetaan uusi arvo errorMessage-muuttujalle. Jos errorMessage saa arvon, niin se näytetään hetkellisesti näytöllä, jonka jälkeen se poistuu ruudulta tietyn ajan kuluessa. Kuvassa 25 on esitelty esimerkkinä selaimen kehitystyökaluista onnistuneen kirjautumispyynnön palvelinvastauksen.

Kuva 25 Palvelimen vastaus onnistuneesta kirjautumisesta. Koko token ei mahdu järkevästi kuvankaappaukseen


```
▼{userName: "string",…}
   token: "eyJhbGciOiJodHRwOi8vd3d3LnczLm9yZy8yMDAxLzA0L3htbGRzaWctbW9yZSNobWFjLXNoYTUxMiIsInR5cCI6IkpXVCJ9.eyJzdHJpbmciOiJzdH
   userName: "string"
```


Selaimen verkkopaneelista voidaan tarkastella vastauksen muotoa. Käyttäjälle palautetaan token, jonka avulla voidaan auktorisoida kaikki muut API-pyynnöt tulevan kirjautuneelta

käyttäjältä. Käyttäjänimi myös palautetaan, jos sitä tarvitaan uniikkina tunnisteena jossakin tilanteessa, tai esimerkiksi näkyvänä tietona käyttöliittymässä.

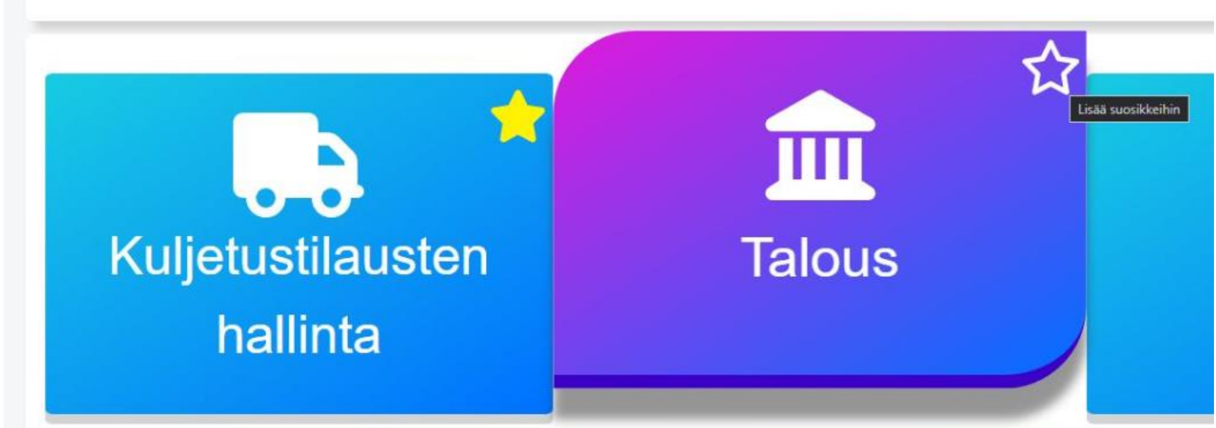
5.6 Front-end -sovelluksen toiminta käyttäjän näkökulmasta

Sovelluksen käyttöliittymää on vanhaan verraten huomattavasti helpompi käyttää. Vanha järjestelmä perustui usean eri iframe-elementin luontiin. Sivulle lisättiin useamman eri HTML-dokumentin sisältö päällekkäin. Ongelmana tässä on isoissa näkymissä välttämättömästi iso määrä erilaisia pyyntöjä palvelimelle, joka hidastaa sekä palvelimen että selaimen toimintaa. Sen lisäksi iframe elementtien välinen kommunikointi on hyvin haastavaa, jonka vuoksi eri sivujen yhteen liimaus kiteytyy hyvin ”palikkamaiseksi” rakentamiseksi.

Iframeilla on myös neliömäinen rakenne, jota ei voi sen sisällä olevalla elementillä ylittää, jonka vuoksi hyvin syvällä olevat elementit vaativat erilaisten teknisten poikkeusratkaisujen tekemistä, jotta esimerkiksi saadaan niille tehtyä koko sivun päällä oleva elementti. Navigointi on myös haastavaa, sillä verkkosivun todellinen osoite ei päivity, jos esimerkiksi ylätason alla olevan iframen osoite muuttuu, jolloin kyseisten alasivuja on todella hankala merkata esimerkiksi selaimessa suosikkisivuksi, tai edes kotisivuksi.

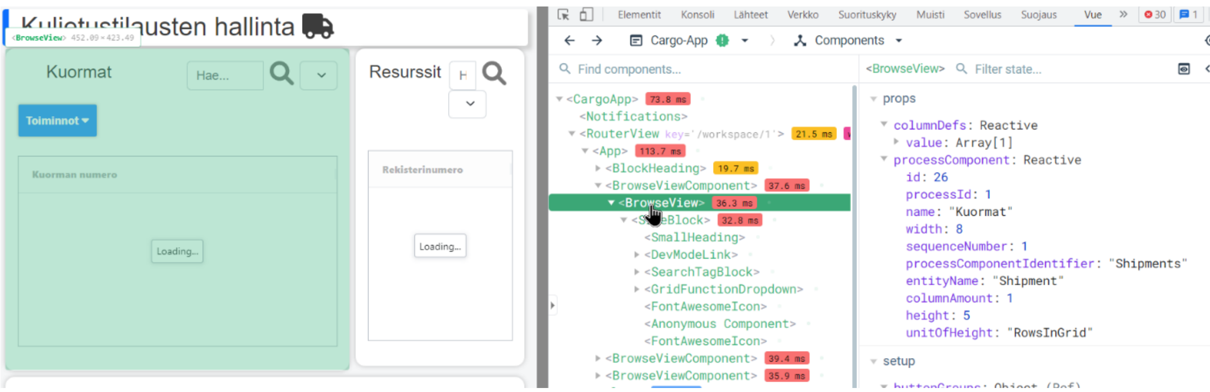
Uudessa järjestelmässä navigointi helpottuu huomattavasti. Jokaiseen sivuun pääsee linkin kautta, sillä navigaatio tapahtuu vue-routerin avulla, ei useasta iframesta. Nämä voi täten myös tallentaa selaimessa suosikiksi tai lisätä kotisivuksi. Jos tallennettu istunto ei ole enää aktiivinen, käyttäjä toki joutuu ensiksi uudestaan kirjautumaan, mutta pääsee kuitenkin toivotulle sivulle takaisin, eikä takaisin indeksisivulle. Kuvassa 26 on esimerkkejä etusivun navigaatioelementeistä.

Kuva 26 Uuden käyttöjärjestelmän prosesseja. Niitä voidaan tallentaa suosikeiksi



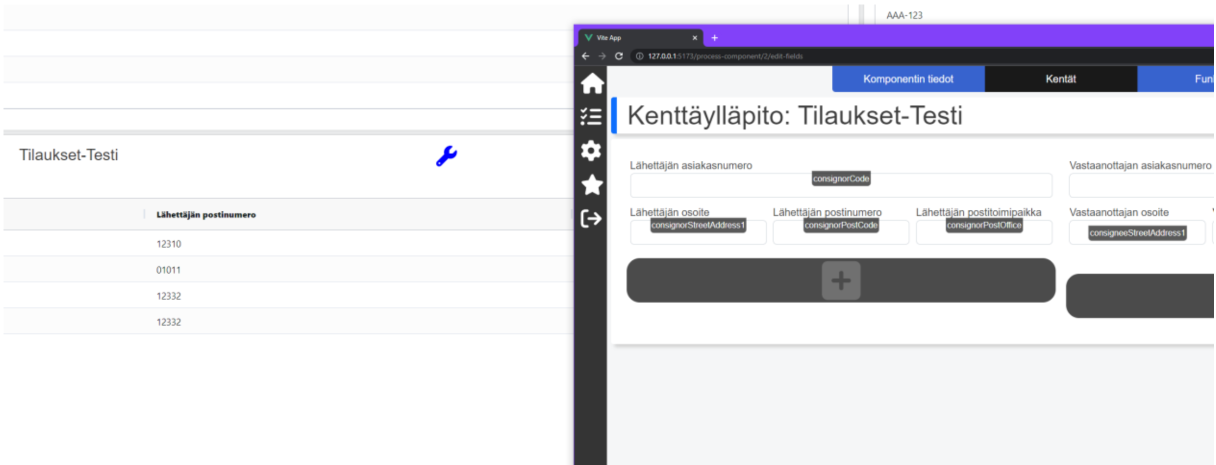
Kuvassa 27 esitellään uutta sivurakennetta moduuleina. Vihreänä näkyvä alue on yksittäinen Vue komponentti, jonka sisällä on useampi muu Vue komponentti (hakupalkki, sivun taustaelementti, taulukko, napit). Rakenteena tämä on parempi kuin iframe-rakenne, sillä iframet rajoittuvat tyypillisesti oman kontekstinsa sisälle, jossa kelpaa toiminnoiksi juuri ne, mitä kyseisen iframen lähde-URL:issa on määritelty. API:n vuoksi logiikka ei ole enää sidottu iframeihin, ja komponenteista pystytään lähettämään käyttäjän toimintaan perustuvia tapahtumia ylöspäin komponenttien hierarkiassa. Tämän vuoksi esimerkiksi kuvankaappauksessa näkyvän ”Kuormat” komponentin sisällä olevat toimintonapit voitaisiin siirtää esimerkiksi ”Resurssit” komponentin alle, ja ne silti toimisivat. Vanhassa järjestelmässä Resurssit-komponenttia vastaavalle verkkoresurssille pitäisi ensin käsin koodata rajapinta kuorman lähdekoodin funktioihin. Tässä järjestelyssä heterogeenistä dataa sisältävää prosessia on vaikeaa luoda ilman ryvettynyttä koodikantaa. Vanhassa järjestelmässä täytyy pakosti olla myös jossakin iframe, jossa on viite haluttuun verkkoresurssiin, vaikkei koko iframea muutoin käytettäisi sivun visuaalisessa asettelussa tai sijoittelussa. Toki kyseisiin resursseihin voisi lisätä useita eri kommunikaatiopisteitä, mutta se on ylipäätänsä haastavaa tehdä jäykän käyttöliittymän vuoksi. Se vähintään vaatisi uuden AXAX-kirjaston rakentamista.

Kuva 27 Uusi sivurakenne



Kehittäjän työ nopeutuu ja helpottuu tässä järjestelmässä paljon. Käyttöliittymä on suunniteltu enemmän niin, että sitä on helppo konfiguroida. Vanhassa järjestelmässä työtilan editoriin pääsy on seuraavanlainen: ensiksi täytyy avata ruudun yläreunasta editori, jonka jälkeen editorista täytyy hakuavaimilla etsiä vastaava työtila. Työtilan alta pitää löytää oikea komponentti, jonka jälkeen taulukkomaisesta listasta pitää löytää ne kentät, jota käyttäjä haluaa editoida. Kuvassa 28 näkyy uuden järjestelmän pikalinkitysjärjestelmä. Painamalla sinistä työkaluikonia kehittäjä pääse heti editoimaan avattua työtilaa. Avattu editori vastaa työtilan ylläpitotilaa visuaalisesti, jonka vuoksi siitä on helpompi löytää muokattavaa sisältöä.

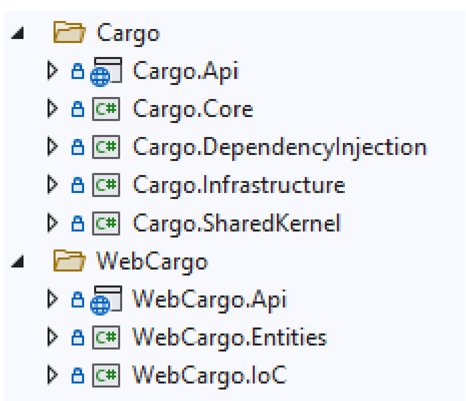
Kuva 28 Vas. osa työtilaa, oikealla työtilan konfiguraatiota



5.7 Uuden back-endin rakenteen yleiskatsaus

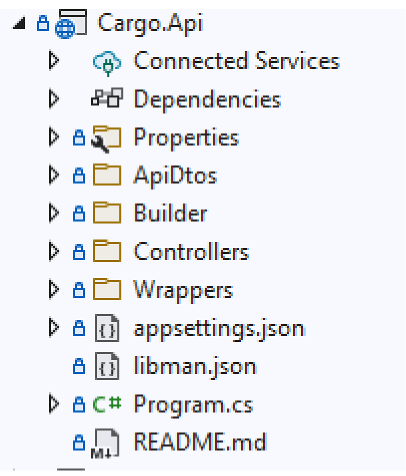
Back-end on rakennettu Clean Architecturen mukaisesti useammasta eri komponentista, joka on viety lohkoihin vastuun perusteella erillisiksi luokkakirjastoiksi. Hakemistorakenteen näkee kuvasta 29. Käyttäjän näkökulmasta sisääntulopiste on API-kirjasto, joka on normaalin luokkakirjaston lisäksi määritelty ASP.NET projektiksi. API sisältää kaikki päätepisteet, jotka sitten määritellään ASP-NET controller ohjelmina.

Kuva 29 Hakemistorakenne



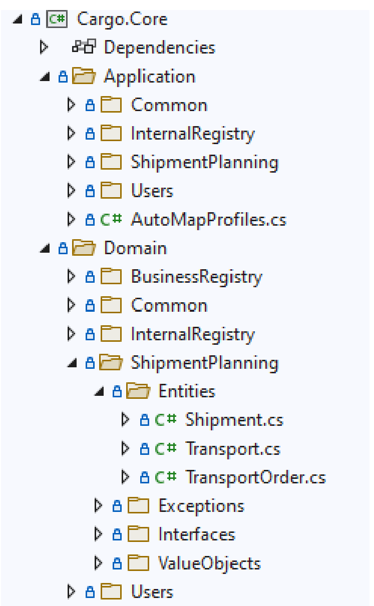
Back-end on rakennettu Clean Architecturen mukaisesti useammasta eri komponentista, joka on viety lohkoihin vastuun perusteella erillisiksi luokkakirjastoiksi. Käyttäjän näkökulmasta sisääntulopiste on API-kirjasto (kuvassa 30), joka on normaalin luokkakirjaston lisäksi määritelty ASP.NET projektiksi. API sisältää kaikki päätepisteet, jotka sitten määritellään ASP-NET controller ohjelmina.

Kuva 30 Rajapinnan sovellus



Core-kirjasto (kuva 31) sisältää sovellustason, jonka tarkoituksena on ajaa kaikki sovellukseen liittyvä bisneslogiikka. Se myös sisältää kaikki sovelluksen käyttämät entiteetit, jonka tiedot tallennetaan lopulta tietokantaan. Clean Architecturen rakennetta katsoen ”Core” kirjasto on sovelluksen tasoista kaikista sisin kirjasto, eli sillä ei ole toisin sanoen riippuvuuksia muihin projektissa määriteltyihin luokkakirjastoihin. Tähän kirjastoon toki sallitaan viittauksia sellaisiin apukirjastoihin, jonka avulla bisneslogiikka saadaan ajettua. Sisältäähän normaalit C#-luokat muutenkin viitteitä yleiskäyttöisiin C# kirjastoihin **using** lausekkeilla, joten miksei sallittaisi sellaiset liitännäiset tällä tasolla, jotka eivät ole täysin kriittisiä kaikista keskeisimmälle toiminnalle? Pienenkokoiset työkalut eivät ole välttämättömän isoja riskejä tuoda ydinkirjastoonkin.

Kuva 31 Core-sovellus. Domain on sisempi kerros ja Application on ”use-cases” kerros



Core-kirjastosta pitää tosin huomata se, että ”use-cases” ja ”entities” tasot ovat tuotu yksittäisen kirjaston sisälle, sillä niiden erottamista omiin projekteihin ei tuntunut riittävän merkitykselliseltä ratkaisulta projektin hierarkiaa miettiessä. Kansiohierarkiassa osat ovat kuitenkin jaettu omiin kansioihinsa. Tarpeen tullen kirjastojen eriyttämien ei ole iso projekti.

DependencyInjection kirjasto (kuva 32) on tehty ohjelmakirjastojen yhteen liittämistä varten. Sen pääasiallinen tarkoitus on tarjota funktioita API kirjastolle, jota kutsumalla voidaan rekisteröidä DI-säilön ohjelmat, sekä liittää ohjelman kantayhteyteen oikeantyyppinen kantayhteysajuri, sekä antamaan kantayhteydelle tietokantatunnukset sekä kannan IP osoitetiedot. DependencyInjection on irrallinen projektin API:sta sen vuoksi, että API kirjastossa ei tarvitsisi tehdä suoria viitteitä Infrastructure-kirjastoon sen yhteen liittämistä varten.

Kuva 32 DI-kerros. Yksinkertaisuus johtuu siitä, että kirjasto sisältää vaan laajennuksia eri luokille, jonka avulla sovelluksen osat yhdistetään.

- Cargo.DependencyInjection
  - Dependencies
  - Extensions
    - AddDbContext.cs
    - DependencyInjector.cs

Infrastructure kirjaston (kuva 33) tarkoituksena on liittää ulkoisia yksityiskohtia ydinkerrokseen, josta olennaisin on tietokannan manipulaatiota käsittelevät ohjelmat. Infrastructure on ainoa projekti, jossa on suoria viitteitä Entity Framework-kirjastoon, jota käytetään projektin ORM-kehyksenä, joten sen käyttö on abstraktoitu muilta sovelluksen osilta. Infrastucture projektia käytetään myös implementoimaan ”Core” ohjelman määrittelemät ”repository” luokat, jonka tarkoituksena on ladata dataa tietokannasta ohjelman entiteetteihin.

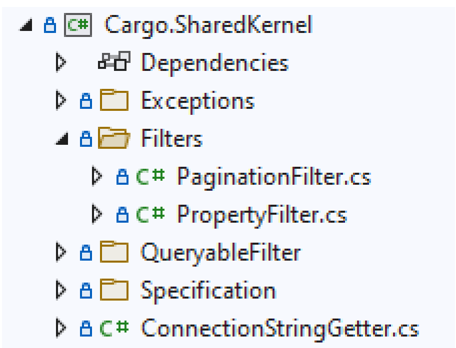
Kuva 33 Infrastructure kerros. ShipmentRepository implementoi kuvassa IShipmentRepositoryn, joka löytyy ydinohjelmasta.

- Cargo.Infrastructure
  - Dependencies
  - Common
  - Entities
    - InternalRegistry
    - ShipmentPlanning
      - Shipments
        - ShipmentRepository.cs
      - TransportOrders
      - Transports
    - Users
  - Migrations
  - NewFolder
  - Persistence

SharedKernel (kuva 34) ei ole varsinaisesti osa jotakin tiettyä sovellustasoa, mutta sen tarkoituksena on tarjota apuohjelmia, jota käytetään eri kirjastoissa. Ohjelmassa lähdekoodin kopioitumisen estämiseksi niiden yhteiset toiminnot on yksinkertaisesti

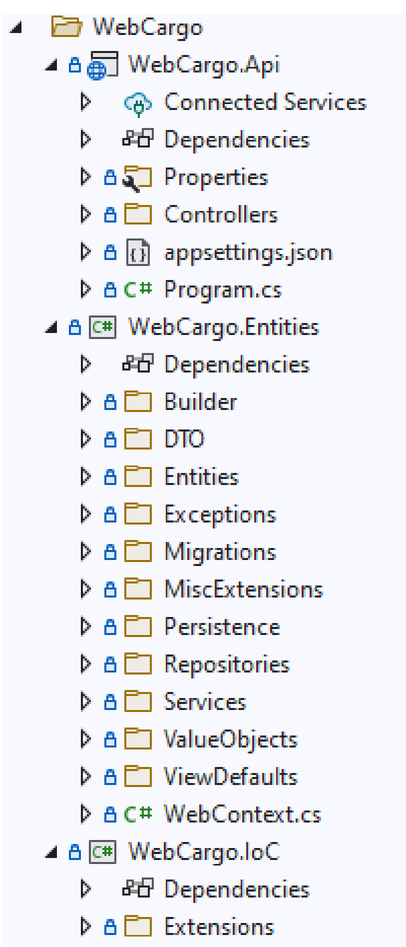
yhdistetty omaan kirjastoon. Kirjastosta löytyy luokkia elementtien suodatusta, virheenhallintaa ja tiedon siivutusta varten.

Kuva 34 SharedKernel kerros. Kuvassa näkyvää PaginationFilteriä käytetään kahdessa eri kirjastossa haettujen tietojen sivunumerointiin API:ssa



”Cargo” kirjastojen lisäksi projektista löytyy myös ”WebCargo” kansio (kuva 35). Se sisältää pelkistetympään hierarkiaan perustuvan ohjelman, jonka tarkoituksena on tarjota sovelluksen verkkosivuversiota varten objekteja, jonka avulla voidaan generoida verkkosivun työtiloihin dynaamisia näkymiä, jonka asettelua ja elementtejä voidaan muokata lennosta, ilman että näkymien asettelua kovakoodataan.

Kuva 35 WebCargo kokonaisuudessaan. Infrastruktuuri on yhdistetty tässä mallissa suoraan bisneslogiikkaan, mutta koodin siisteyden vuoksi DI on eroteltu omaksi kirjastoksi



WebCargon arkkitehtuurillinen rakenne on paljon yksinkertaisempi, sillä se ei ole yhtä keskeinen kuin pääsovelluksen toiminalle muutoin kuin käyttöliittymän generoinnissa. Sen tarkoituksena on ainoastaan tarjota näkymille tallennettuja komponentteja, eli ohjelma ei sinänsä tarjoa bisneslogiikkaa. Tämän vuoksi se ei ole järkevää jaotella tätä osaa projektista samanlaisen arkkitehtuurin perusteella kuin pääprojektissa. Sovellusarkkitehtuurin päätökset täytyvät kuitenkin perustua aina omiin tarpeisiin tarvitulle sovellukselle.

5.7.1 API-kirjasto

Cargo.Api on projektin sisääntulopiste. Sen merkitys on hyvin kapea tässä projektissa, ja sen toimia ohjataan controller luokista, jonka toimintaa esitellään kuvassa 36 lyhyellä ohjelmapätkällä kuljetustilauksen rajapinnasta.

Kuva 36 GET päätepiste kaikkien tilausten hakemiselle sekä uuden tilauksen luonnin POST päätepiste

```
[HttpPost]
public async Task<ActionResult<IEnumerable<TransportOrderDTO>>> GetAll([FromQuery] PaginationFilter
pagination, [FromBody(EmptyBodyBehavior = EmptyBodyBehavior.Allow)] List<PropertyFilter>?
propertyFilters) {
  var result = await standardSearchService.Search(new StandardSearch { Pagination = pagination,
PropertyFilters = propertyFilters });
  return OkOr404(result.Select(x => x.ProjectToDTO()));
}
[HttpPost("create")]
public async Task<IActionResult> Create(TransportOrderDTO dto)
{
  var order = await _mediator.Send(new CreateTransportOrder { RequestTransportOrder = dto });
  return CreatedAtAction(
    actionName: nameof(Get),
    routeValues: order,
    value: order);
}
```

Controller ohjelmassa tapahtuvien reittien määrittely on .ASP NET ympäristössä hyvinkin yksinkertaista: Annetaan metodi, jonka palautusarvona on IActionResult Interface-luokan implementoiva vastaus, joka tässä kontekstissa on rajapinnalle palautettava vastauskoodi vastaussisältöineen. Varsinaisen reitin määrittely onnistuu siten, että metodille annetaan [Http] alkuinen attribuutti, kuten [HttpPost], [HttpGet] tai [HttpDelete]. Sen lisäksi attribuutille voidaan määritellä argumenttina alareitti. Esimerkissä POST metodilla on argumentti ”create” joka tarkoittaa, että sen reitti tulee olemaan /rajapinta/resurssi/create.

Controller ohjelma itsessään on hyvin ”tyhmä” ohjelman logiikan suhteen. Sen tarkoituksena on tuottaa hyvin kevyitä controller-luokkia, jotka ainoastaan välittävät käyttäjän pyyntöjä ohjelman bisneslogiikalle, ja taas toisinpäin. Tätä voidaan toteuttaa niin, että käytetään ainoastaan sovellustasolla määriteltyjä Interface-luokkia bisneslogiikan suorittamiseen. Siinä rakenteessa näiden Interface-luokkien bisneslogiikan implementoi jokin muu ohjelma, olkoon se samalla tasolla kuin sovellus, tai tiedonhakutapauksissa implementaatio saattaa tulla puhtaasti infrastruktuuritasolta. Kuvan esimerkissä create tapahtuma on toteutettu tätä hieman monimutkaisemmalla ”mediator” suunnittelumallilla

Mediator-suunnittelumalli tarkoittaa sellaista suunnittelumallia, jossa toisistaan riippuvaisten ohjelmien väliin luodaan erityinen ohjelma, joka pystyy välittämään kutsuvalle ohjelmalle tehtävää varten ajettavan ohjelman, ja ajaa sen. Tämä tekniikka siis irrottaa ohjelmasta sen käyttämät riippuvuudet, jonka vuoksi niitä on helpompi ylläpitää. Tätä mediator-mallia ei käytetä tässä projektissa muuhun kuin controller-luokkien irrottamiseen ydinlogiikkaa ja dataa hakevista ohjelmista, eli varsinaisen sisäisen järjestelmän sisällä toisiin ohjelmiin tehdyt viittaukset ovat edelleen esimerkiksi DI-säilöstä haettuja tietoja.

5.7.2 Core-kirjasto

Core-kirjasto on sovelluksen ydin. Tässä voidaan esitellä kyseistä tasoa kahdella esimerkillä. Ensiksi näytetään, että miten tietyn bisnestoimintaan liittyvän tapahtuman orkestrointi tapahtuu tällä tasolla, jonka jälkeen esitellään myös ohjelman dataa määritteleviä luokkia.

Kuvan 37 ohjelmakoodi on se ohjelma, johon kuorman suunnitellun rajapinnasta tulevat pyynnöt kutsuvat, kun rajapinnasta pyydetään resurssia, joka liittää valitut kuljetustilaukset tietylle kuormalle. Ohjelma käyttää ensiksi repository Interface-luokkia hakemaan dataa, jonka perusteella voidaan suorittaa kannan projektioita vasten tapahtuman bisneslogiikkaa. Datan saannin jälkeen iteroidaan joukko kuljetustilauksia niin, että niistä jokainen yritetään liittää kuormalle. Kuorma-objektissa itsessään on tilauksen liittämismetodi, mutta sen orkestrointi tapahtuu tämän ohjelmakoodin kautta. Kuorma voi siis itse validoida tapahtuman. Mahdollinen virhetila muodostuu kuorman tilauksen kuormaamismetodissa, jos yritetään kuormata tilausta, joka on jo kyseisellä kuormalla.

Kuva 37 ShipmentPlanning (Kuorman suunnittelu) domainista esimerkki ohjelmasta, joka liittää tilauksia kuormalle. Unit.Value on tapa palauttaa ”void”-, eli tyhjätyyppinen arvo.

```
public async Task<Unit> Handle(AttachTransportOrderToShipment request, CancellationToken cancellationToken)
{
   Shipment shipment = await shipmentRepository.GetShipmentWithTransportsAndTransportOrders(request.ShipmentId);
  if (shipment == null) throw new EntityNotFoundException(request.ShipmentId, typeof(Shipment));
  List<TransportOrder> transportOrders = await transportOrderRepository.GetAsync(request.TransportOrderIds);
  foreach (TransportOrder transportOrder in transportOrders)
  {
    shipment.AttachTransportOrder(transportOrder);
  }
  await transportOrderRepository.UpdateAsync(transportOrders.ToArray());
  return Unit.Value;
}
```

Toinen piirre Core-kirjastosta on varsinaiset oliot, jolla varsinaista bisneslogiikkaa voidaan ajaa. Se sisältää kenttiä, jotka kertovat erilaisista kuorman tiedoista kuorman suunnittelussa. Olion rakentajasta voidaan nähdä, että ohjelmassa ei voida luoda kuormaa ilman, että siinä on vähintään yksi kuljetustilaus. Tämä on siis yksi esimerkki siitä, että kuinka datan validointia voidaan suorittaa entiteettitasollakin.

Kuljetustilauksen entiteetistä kuvassa 38 voidaan huomata, että entiteettien tietoja voidaan esitellä muinakin kuin primitiivisinä tietoina. Osoitetiedothan voisi esittää pelkkinä merkkijonomuuttujina, tästä monesti aiheutuu ilmiö, jota kutsutaan nimellä ”Primitive obsession” (Shvets, 2014), karkeasti käännettynä ”pakkomielle primitiivien käyttöön”.

Kuva 38 Shipment-luokan pätkä

```
public class Shipment : Entity
{
    public ulong ShipmentNumber { get; set; }
    public DateTime RealizedDeliveryTime { get; private set; }
    public virtual ICollection<TransportOrder> TransportOrders { get; private set; } = new
List<TransportOrder>();
    public Transport? TransportOfShipment { get; private set; }

    private Shipment() { }
```

```
    public Shipment(ulong shipmentNumber, ICollection<TransportOrder> orders, Transport? transport =
null)
    {
        ShipmentNumber = shipmentNumber;
        if (orders.Count == 0)
            throw new NoOrdersSelectedForNewShipmentException();
        TransportOrders = orders;
        TransportOfShipment = transport;

    }
```

”Primitive obsession” tarkoittaa sellaista tilannetta, jossa tosi iso määrä primitiivisiä kenttiä (teksti, numerot, loogiset kentät) edustavat ohjelman olioiden dataa. Ongelma pelkkien primitiivien käytössä on se, että ne ovat liian yleispäteviä, eikä ne voi suoraan sisältää mitään tiettyä ohjelmalogiikkaa tai validointia. Esimerkiksi merkkijonotyyppinen osoitekenttä ei pystyisi tarkistamaan, että onko syötetty postinumero oikeellinen. Sen vuoksi data koteloidaan objektiin, jonka aie on ilmiselvä nimestä, ja säilytetään sen yleispätevä bisneslogiikka sen sisällä. Tekniikalle kontrastina on proseduraalisessa ohjelmoinnissa tyypilliset transaktioskriptit, jossa toiminnanohjauslogiikka on osa erillistä proseduuria. Esimerkiksi kuvassa 39 Primitiivisiä riippuvuuksia on purettu rakentamalla osoitteet ”Address” luokkina.

Kuva 39 Kuljetustilausten entiteetti

```
public class TransportOrder : Entity
{
    public string? ConsignorCode { get; set; }
    public Address? ConsignorAddress { get; set; }
    public string? ConsigneeCode { get; set; }
    public Address? ConsigneeAddress { get; set; }
    public TransportOrder() { }
}
```

Primitiivisten objektien tilalla voidaan käyttää luokkia, jossa validoidaan dataa, jotka keskeisesti liittyvät toisiinsa. **Value object** luokat ovat eräs tapa toteuttaa sellaisia rakenteita. **Value object** on sellainen luokka, joka edustaa pientä entiteettiä, jonka samanvertaisuus toiseen samantyyppiseen entiteettiin ei perustu objektin identiteettitietoihin (esimerkiksi rivin ID tietokantataulussa), vaan sen yhteisarvoon.

Ymmärtämisen helpottamiseksi voit ajatella **Value object**-luokkaa moniulotteiseksi luokan ominaisuudeksi, jolla voi olla sisäistä validointia. Esimerkiksi osoite on identtinen toiseen osoitteeseen, jos niillä on sama katuosoite ja postinumero. Niitä ei silloin ei voi ajatella kahtena osoitteena, vaikka loogisesti instansseja on kaksi. **Value object**-luokilla ei siis täten ole ID kenttää, ja ovat yksinään arvottomia ohjelman kannalta. Ne pitää sitoa johonkin muuhun tietoon, jotta niillä olisi arvoa, kuten esimerkiksi kuljetustilauksen lähetysosoitteeksi. Tästä perspektiivistä huomataan, että osoitteella ei olisi näin yksinään mitään virkaa, ellei sen toiminta perustu jonkinlaiseen osoiterekisteriin, mutta silloin osoitteella olisi identiteetti tavallisena entiteettinä. Osoitteelle pitäisi silloin lisätä tunniste, kuten ID kenttä, joka on selkeästi yksilöivä tieto. Kuvassa 41 on kuvattu osoitteille oma luokka, joka on value-object. Ohjelmassa käytetään Address-luokan tapaisia rakenteita kuvaamaan selkeästi omaksi yksiköksi kuvattavia tietoja.

Kuva 40 Address value-objektin sisältöä

```
public Address(string? streetAddress1 = null,
               string? streetAddress2 = null,
               string? postCode = null,
               string? postOffice = null)
{
    if (string.IsNullOrEmpty(streetAddress1))
        throw new AddressStreetAddressEmptyException();
    //Naive post code validation.
    if (postCode.Length != 5)
        throw new AddressPostCodeInvalidException();

    StreetAddress1 = streetAddress1;
    StreetAddress2 = streetAddress2;
    PostCode = postCode;
    PostOffice = postOffice;
}
public override string ToString()
{
    string fullAddress = StreetAddress1;
    if (!string.IsNullOrEmpty(PostCode))
    {
        fullAddress = $"{fullAddress}, {PostCode} {PostOffice}";
    }
    return fullAddress;
}
```

5.7.3 Infrastructure-kirjasto

Projektin infrastruktuurissa kuvataan implementaatioita järjestelmän ulkoisista tekijöistä, jotka kommunikoivat esimerkiksi tulostimien, pdf-tiedostojen ja tietokantojen kanssa. PDF-tiedoston muodostaminen ei varsinaisesti kuulu osaksi ohjelman bisneslogiikkaa, jonka vuoksi se säilyy sisäisen järjestelmän ulkopuolella, samaan tapaan kuin tietokannan luokat. Tässä tapauksessa infrastruktuuriohjelman ohjelmat pääosin määrittelevät millaisia tauluja ohjelman tietokannasta löytyy, sekä määrittelee niille ohjelmalogiikassa käytettävät toiminnan yksiköt (engl. unit of work). Ohjelma pystyy käyttämään näitä toiminnan yksiköitä manipuloimaan tietokannasta löytyvää dataa (vaikka ydinohjelma ei tietäisikään tietokantaimplementaation sisällöstä mitään).

Tietokannan operointiin ohjelmassa käytetään MySQL kantayhteyden tekevää ohjelmaa, jonka palauttamaa dataa palvellaan ohjelmalle Entity Frameworkin muodostamina luokkaprojektioina. Käytännössä tämän avulla kannassa oleva data muuttaa muotoaan tietokoneen välimuistilla yhdeksi ohjelmassa määritellyksi olioksi, esimerkiksi resurssin tai kuljetustilauksen olion instanssiksi. Sitä on helpompi käsitellä ohjelmakoodissa, kuin suoraa kantariviä. Sille voidaan esimerkiksi määritellä olemassaolo ohjelman sisällä muistisijaintina, käyttämällä aikaisemmin mainittuja olioita muuttujina. Sille voidaan tehdä kaikki tavanomaiset manipulaatio-operaatiot, mitä voi tehdä mille tahansa normaalille C#-luokkainstanssille. Kun kaikki halutut toimet on tehty sille instanssille, se voidaan tallentaa uudestaan kantaan käyttämällä Entity Frameworkia.

Infrastruktuuri implementoi ydinohjelman Interface-luokkia, jossa määritellään, että miten data tallennetaan kantaan, ja miten se haetaan sieltä. Kuten aikaisemmin oli todettu, ydinohjelma ei tiedä implementaatiosta mitään. Nämä tallentamiseen ja lukemiseen liittyvät toimet tehdään niiden Interface luokkien avulla, jolloin implementaatio säilyy ydinohjelmiston ulkopuolella. Tämän vuoksi ydinohjelmasta ei löydy kirjastoviitteitä Entity Frameworkiin. Entity Framework on tässä tapauksessa helppo vaihtaa johonkin muuhun

ORM kehykseen, kuten Dapperiin, sillä tehtyjen kirjastoviittauksen määrä on rajoitettu tiettyyn moduuliin.

6 Tuotekohtaisia parannuksia esiteltynä

Teknisten parannusten lisäksi tässä projektissa on useampi sellainen rakenne, joka tekisi tuotteen kehittämisestä sujuvampaa ja järjestelmän rakenteista joustavampia. Niitä esitellään tässä kappaleessa pintapuolisesti. Vanhan järjestelmän ylläpitonäkymien rakentava syntaksi on äärimmäisen haastavan näköinen syntaksi, etenkin järjestelmästä tietämättömälle tekijälle. Ylläpitoikkunoihin muodostettavat lomakekentät eivät ole staattista tietoa, mutta niiden oletuksena toimiva konfiguraatio on erittäin huonon syntaksin takana. Niiden konfiguraation syntaksi luodaan suoraan ohjelmakoodiin muun logiikkaa ohjaavan koodin sekaan merkkijonoina, jotka erotellaan kaksoispisteillä, ja jokainen kenttä erotellaan pilkulla, esimerkiksi seuraavalla tavalla:

KentänNimi:KentänTyyppi:KentänKäännösTeksti:KirjoitusOikeus:AlasvetoValikonRekisteri:AlasvetoValikonRekisterinKentät:AlasvetoValikonPaluuArvo:…

Esimerkki ei ole tarkka oikealle versiolle, mutta se kuitenkin havainnollistaa vanhan järjestelmän ongelmaa: sitä on todella vaikea lukea ja on todella positiosidonnainen. Jos sijoitat kentissä olevat tiedot väärien kaksoispisteiden väliin, kentät eivät silloin toimi. Pahimmassa tilanteessa järjestelmä ei kerro, että syntaksi on virheellinen, ja ei ole välttämättä tietoa mistä virhetilanne johtuu. Tämä järjestelmä myös vaatii tarpeetonta toistoa; jos esimerkiksi haluat täyttää tiedot, jotka ovat kaksoispisteiden 30-35 välillä, mutta et tarvitse 20-30 puolipisteiden tietoja, joudut silti täyttämään ne kaksoispisteet, jotta syntaksi toimii. Tämä tarkoittaa sitä, että kehittäjällä kuluu joskus aikaa pelkkien puolipisteiden laskemiseen, joka kuulostaa enemmän tietokoneen tehtävältä, ei kehittäjän tehtävältä.

Syntaksia on myös todella haastava muuttaa jälkeenpäin ja uusien sivun konfiguraatioon kuuluvat kenttä tarvitaan ensiksi ehdollistaa järjestelmässä joksikin puolipistepositioista, jotta uusi tieto vaikuttaisi järjestelmässä. Pahimmassa tapauksessa olemassa olevat

konfiguraatiot aiheuttavat ajonaikaisia virheitä, koska niillä ei ole yhtä paljon kaksoispisteitä kuin muokatussa syntaksissa on. Tämän vuoksi kaksoispisteiden määrä pitää ehdollistaa, joka kerta kun uutta positiota halutaan käsitellä.

Uudessa järjestelmässä koko konfiguraatio on ensinnäkin purettu pois kaiken muun koodin seasta. Sille on omistettu oma luokka, joka sitten tallennetaan omaan tiedostoon. Sen lisäksi tässä on ajateltu konfiguraatiota enemmän tietokoneen näkökulmasta. Konfiguraatiota ajatellessa nousee esiin yksi hyvinkin tärkeä seikka: Ohjelmaprojekteissa on käytössä tietokoneen luontainen etu: automatiikka, joten miksei sitä voisi käyttää hyödyksi? Microsoft rakentaa esimerkiksi ohjelmassa käytetyt datan säilyvyyden konfiguraatiot ohjelmakoodilla, joka on suhteellisen lähellä kuvan 41 syntaksia. (Microsoft, 2022)

Kuva 41 Kuvassa kuljetustilauksen entiteetin konfiguraatio oletuksilla

```
public class TransportOrderViewConfig : IViewDefaultConfig
{
    ProcessComponentFieldBuilder<TransportOrderDTO> builder = new
ProcessComponentFieldBuilder<TransportOrderDTO>();
    public ViewConfig ConfigureViewDefault(IViewDefaultConfigurer viewDefaultConfigurer)
    {
        return viewDefaultConfigurer.Build().WithColumns(new ViewColumn[]
        {
            new ViewColumn(new ProcessComponentField[]
            {
                builder.NewField(t => t.ConsignorCode, "Lähettäjän asiakasnumero")
                    .AsTableColumn().Build(),
                builder.NewField(t => t.ConsignorStreetAddress1, "Lähettäjän osoite 1")
                    .Build()
            }),
            new ViewColumn(new ProcessComponentField[]
            {
                builder.NewField(t => t.ConsigneeCode, "Vastaanottajan asiakasnumero")
                    .AsTableColumn().Build(),
                builder.NewField(t => t.ConsigneeStreetAddress1, "Vastaanottajan osoite 1")
                    .Build(),
            })
        });
    }
}
```

Tässä järjestelmässä konfiguraation pääosin hoitaa ajonaikainen konfiguraatio-ohjelma, joka vähentää toistuvan koodin määrää. Kaikkea tietoa ei tarvitse tähän syöttää, sillä järjestelmä tietää mitä tietoa voidaan käyttää tyhjän tiedon oletustietona. Oletustiedon voi tehdä osana luokan rakentajaa, ja voi käyttäytyä eri tavalla eri konteksteissa. Tämä poistaa turhien puolipisteiden ongelman kokonaan, sillä uupuvat tiedot hoidetaan tietokoneen laskennalla, kuten automatisoidussa konfiguraatiossa kuuluisikin tehdä.

Konfiguraatio on myös hierarkkinen, jonka vuoksi sitä on huomattavasti helpompi lukea. Sen lisäksi konfiguraatiossa usein toistuvia tietoja voidaan myös purkaa apumetodeihin, kuten kuvassa näkyvä AsTableColum() metodi, joka alustaa kentän valmiiksi näkyväksi prosessissa selailunäkymän taulusarakkeena, ilman että sille tarvitsee eksplisiittisesti määritellä kaikkia siihen liittyviä tietoja. Tämä vähentää konfiguraation pituutta huomattavasti, ja mahdollistaa uuden tiedon lisäämistä retroaktiivisesti apumetodeilla, ilman että olemassa olevia konfiguraatiota tarvitsee muuttaa tai ehdollistaa.

Uudelle kentälle vaaditut tiedot ovat minimissään kaksi eri tietoa: Kenttä entiteetin ominaisuutena, joka kohdentaa sen käsittelyssä olevan ominaisuuden (entiteetti => entiteetti.ominaisuus argumentti), sekä staattinen otsikkoteksti. Kaikki muu hoituu silloin automaattisesti konfiguraattoriohjelman kautta. Jopa staattisen otsikkotekstin voisi purkaa pois jonkin tekstikannan automatiikaksi, jolloin konfiguraatiossa olisi vain yksi pakollinen tieto.

Nämä kaikki konfiguraatiot haetaan ohjelmankirjaston kokoonpanosta automaattisesti, kun ohjelma käynnistetään (kuva 42). Ne sen jälkeen ajetaan, ja jokainen niistä tallennetaan ohjelman käyttämään välimuistiin erityiseen konfiguraatiosäilöön, josta niitä voidaan hakea tarvittaessa. Tämä toki tarkoittaa sitä, että osa välimuistista varataan näille konfiguraatioille, mutta se on kuitenkin nykypäivänä aivan todella pieni määrä muistia. SaaS pilviäkin ajatellessa huomataan äkkiä, että ylimääräisen välimuistin varaaminen esimerkiksi Microsoft Azuresta tai Amazonin AWS pilvestä on helppoa ja välitöntä. Välimuistia saa käytännössä loputtomasti suhteutettuna tarpeeseen, niin miksei sitä voisi hyödyntää?

Kuva 42 Metodi, joka löytää kaikki konfiguraatiot ohjelmasta

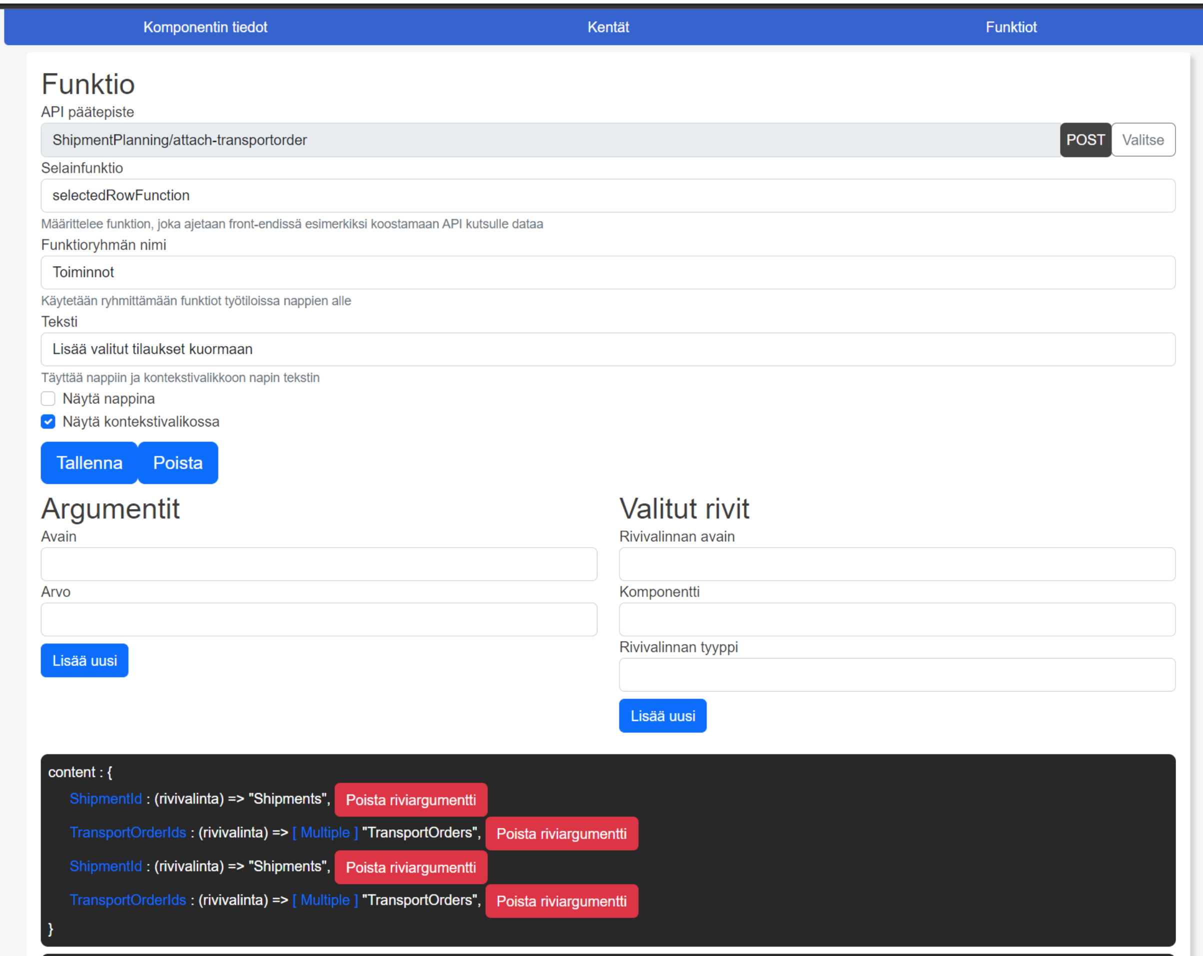
```
public void BuildViewDefaults()
{
    var ViewDefaultConfigInterface = typeof(IViewDefaultConfig);
    var Types = AppDomain.CurrentDomain.GetAssemblies()
            .SelectMany(x => x.GetTypes())
            .Where(x => ViewDefaultConfigInterface.IsAssignableFrom(x) && x.IsClass);
    foreach (var type in Types)
    {
        IViewDefaultConfig viewDefaultConfig = (IViewDefaultConfig)Activator.CreateInstance(type);
        ViewConfig result = viewDefaultConfig.ConfigureViewDefault(viewDefaultConfigurer);
        viewDefaultContainer.AddDefault(result);

    }
}
```

Vanhassa järjestelmässä on eräs piirre, joka rajoittaa front endin muuttamista ankarasti: Kaikki näppäimistä tapahtuvat ja kentistä pois siirtymisen aiheuttamat JavaScript funktiot tallennetaan kantaan sellaisenaan. Sitten ne tuodaan selaimeen raakana tekstinä, joka sitten liitetään nappiin tai kenttään attribuuttina. Tämä aiheuttaa haasteita muun muassa sen vuoksi, että funktiot pitää kirjoittaa käsin tekstialueisiin, jossa ei mm. ole koodia tunnistavaa syntaksia tai tekstiä täydentäviä ehdotuksia, kuten normaalisti olisi IDE:ssä tai koodin tekstieditorissa. Sen lisäksi tekstikenttä on avuttoman pieni, ja sitä on vaikea lukea, jos on kyseessä pitkä koodinpätkä. Siihen kenttään kirjoitetaan usein myös hyvin vakioitua tekstiä, eli todella monessa funktiossa toistuu tarpeettomasti samat ehtolausekkeet. Tämä myös vaikeuttaa rakenteen muuttamista, sillä ne kentät usein sisältävät viittauksia, jotka ovat käyttöliittymän riippuvuuksia kirjastoista, jonka takia niiden muuttaminen jälkeenpäin on haastavaa. Pahimmassa tapauksessa muutokset pitäisi tehdä satoihin eri paikkoihin.

Uudessa järjestelmässä ei koskaan tallenneta kantaan JavaScriptiä, sinne tallennetaan pelkästään viitteitä, jonka avulla selaimessa päätellään mitä toimintoja käyttöliittymässä halutaan tehdä esimerkiksi napin painalluksella. Tässä järjestelmässä kannan tietoja voidaan käyttää, vaikka jossakin eri käyttöliittymässä, jossa ei ajeta JavaScriptiä, vaan jotain muuta. Sen lisäksi olemassa olevia käyttöliittymärakenteita on mahdollista muuttaa globaalisti, sillä koodiin tehtävät muutokset rajoittuvat muutamaan tiedostoon, joilla ei ole toimintaa rajoittavia tekijöitä sidottuna tietokantaan.

Uudessa funktioeditorissa (kuva 43) ei määritellä kentälle suoraa koodia, sille määritellään viite selainfunktiona. Sen lisäksi sille annetaan viite päätepisteestä, jonne käyttöliittymän pyyntö ohjataan. Kaikki päätepisteet löytyvät API:n metadataa palvelevasta päätepisteestä, jonka vuoksi päätepisteiden osoitteitakaan ei tarvitse käsin kirjoittaa tai edes etsiä koodista. Tämä tarkoittaa sitä, että funktioeditoria voi periaatteessa käyttää sellainen henkilö, joka ei osaa koodata.

Kuva 43 Uuden käyttöjärjestelmän funktioeditori



Kuvassa 44 on esitelty vastauksen sisältöä metadatan päätepisteistä. Sen datasta rakennetaan Vue-komponenteilla näkymä, jossa voidaan valita haluttu päätepiste palvelimen funktiopyynnölle.

Kuva 44 esimerkki metadatan päätepisteiden sisällöstä.

Code	Details
200	Response body


```
[
  {
    "methodType": "POST",
    "route": "ShipmentPlanning/attach-transport",
    "actionInfo": "ShipmentPlanning.AttachTransport",
    "controllerInfo": "ShipmentPlanningController:AttachTransport"
  },
  {
    "methodType": "POST",
    "route": "ShipmentPlanning/attach-transportorder",
    "actionInfo": "ShipmentPlanning.AttachTransportOrder",
    "controllerInfo": "ShipmentPlanningController:AttachTransportOrder"
  },
  {
    "methodType": "POST",
    "route": "ShipmentPlanning/create-shipment-with-orders",
    "actionInfo": "ShipmentPlanning.CreateNewShipmentWithOrders",
    "controllerInfo": "ShipmentPlanningController:CreateNewShipmentWithOrders"
  },
```

Päätepisteille tehdyt pyynnöt sisältävät usein argumentteja, jonka vuoksi päätepisteiden metadatasta voidaan pyytää kaikki tietyn päätepisteen pyynnön dataskeema (kuva 45), jonka vuoksi käyttöliittymässä voidaan rakentaa vakioitu editori argumenteille, jossa kaikki argumenttien nimetkin voidaan lisätä alasvetovalikkoon, joka helpottaa argumentin valitsemista tietylle pyynnölle.

Kuva 45 Yksittäisen päätepisteen argumenttimetadata

Curl

```
curl -X 'GET' \
  'http://localhost:5001/api/Info/endpoints/arguments/ShipmentPlanning.AttachTransport' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5001/api/Info/endpoints/arguments/ShipmentPlanning.AttachTransport
```

Server response

Code Details

200 Response body


```
[
  {
    "parameterSchema": {
      "$schema": "http://json-schema.org/draft-04/schema#",
      "title": "AttachTransportToShipment",
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "ShipmentId": {
          "type": "integer",
          "format": "int64"
        },
        "TransportToAttachId": {
          "type": "integer",
          "format": "int64"
        }
      }
    }
  }
]
```


7 Vertailukoe

Vertailuja näiden kahden järjestelmän välillä tehdään käyttämällä ajastettua tapahtumankulun vertausta, ja vertailun on pääosin tarkoitus havainnollistaa, että kuinka nopeaa kummassakin järjestelmässä on muodostaa yksinkertainen näkymä työtilaksi, jota voi asiakas käyttää. Nopeammin pystytettävä työtila tarkoittaa työmäärästä aiheutuvien kulujen vähentymistä, joka taas parantaa tuotteen kannattavuutta. Vanhan ja uuden järjestelmän muita määrällisiä eroja voi olla vaikea mitata, sillä järjestelmien kokoero on niin massiivinen, että esimerkiksi realistisia nopeuseroja selaimen ja palvelimen yhteistoiminnassa olisi haastava mitata.

Järjestelmiä kokeillaan kahdella erilaisella ajanmittauskokeella. Ensimmäinen on seuraavanlainen: On valmis ympäristö, johon luodaan seuraava näkymä: Luodaan kaksi prosessia, joista toinen on kuormien taulukko, ja toinen on kuljetustilausten taulukko. Kuljetustilauksen ylläpitoon lisätään tiedoiksi lähettäjän ja vastaanottajan tiedot ylläpidettäviksi, kuorman ylläpito voi pysyä oletustilassa, eli sille ei erikseen määritellä ylläpitonäkymään oletuksesta poikkeavia tietoja. Kuormalle voidaan näkymässä lisätä kuljetustilauksia osaksi kuormaa. Toinen skenaario on yksinkertaisempi: Siinä kuvitellaan, että ollaan kehittämässä jotakin työtilaa, mutta ylläpitoruudussa huomataan, että halutaan muuttaa jotakin tietoa näkymästä. Tästä tilanteesta mitataan, että kuinka kauan kestää hakea editorista näkymän konfiguraatio esille sekä kuinka kauan kestää löytää sieltä oikea komponenttikenttä editoitavaksi.

Aika tapahtumalle mitataan niin, että molempien rakentaminen ensiksi nauhoitetaan videolle, jonka jälkeen videoista voidaan mitata, että kuinka kauan kestää jokaisessa osassa tehtävää. Nämä tiedot sitten kirjataan Excel-taulukkoon, jossa niitä voi sitten verrata. Vertailu tehdään henkilöllä, joka tuntee molemmat järjestelmät, mutta uuden järjestelmän käyttöliittymän ja käyttäjäkokemuksen parannukset voisivat helposti lisätä kahden vertailun välistä aikaeroa. Todella monimutkaisissa näkymissä aikaero todennäköisesti kasvaa.

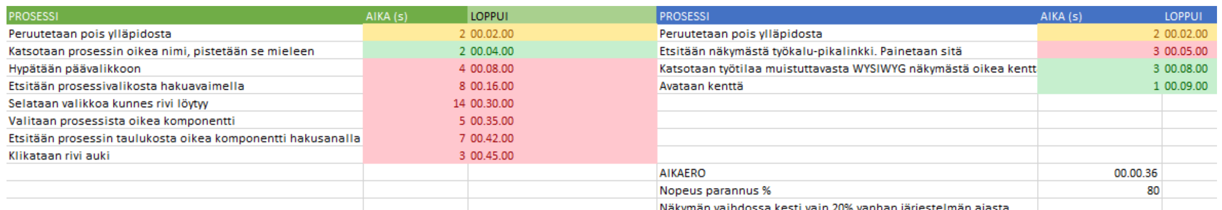
Tuloksessa punaisella on maalatut hitaammat kohdat verrannollisesti (kuva 46). Tulokset selkeästi osoittavat, että vanhaa järjestelmää on hitaampi käyttää konfiguraatiomielessä. Sen konfigurointi on merkittävästi hitaampaa taulukkomaisen konfigurointityökalun takia. Siinä joutuu konfiguroimaan paljon sellaista tietoa, minkä voisi automatiikalla täyttää. Sen lisäksi editorin näkymiä ei olla suunniteltu juuri nimenomaan editoria varten, vaan se käyttää ohjelmalle yleisiä taulukkokomponentteja tiedon esittämiseen. Sitä on sen vuoksi vaikea hahmottaa, eikä se keskity editorissa olennaisiin asioihin. Uusi järjestelmä on taas huomattavasti nopeampi konfiguroida, sillä sen näkymät ovat suunniteltu juuri editointia varten. Sen WSWIYG (What You See Is What You Get) konfigurointityökalut helpottavat editointia.

Kuva 46 Koe 1. Vihreä sarake on vanha järjestelmä, sininen sarake on uusi järjestelmä

PROSESSI	AIKA (s)	LOPPUI	PROSESSI	AIKA (s)	LOPPUI
Hypätään päävalikosta editoriin	1	00.00.01	Hypätään päävalikosta editoriin	1	00.00.01
Luodaan prosessi - sekä sen tiedot	23	00.00.24	Lisätään uusi prosessi, hypätään suoraan sen prosessin editoriin	12	00.00.13
Valitaan prosessi, lisätään sille työtila - kuormansuunnittelu	17	00.00.41	Lisätään kaksi komponenttia prosessiin: kuormat ja tilaukset	13	00.00.25
Luodaan tilaus- ja kuormaprosessit. Lisätään niille pakolliset t	59	00.01.40	Päivitetään kuormakomponentille oikea entiteetti	19	00.00.32
Lisätään tilausprosessille läh/vas tiedot	52	00.02.32	Päivitetään tilaukselle oikea entitetti + muutetaan sarakemäärää	23	00.00.42
Lisätään tilausprosesille kuormaan lisäys-nappi	77	00.03.49	Lisätään tilaukselle vas/läh tiedot kenttinä	54	00.01.36
			Lisätään tilausprosesille kuormaan lisäys-nappi	34	00.02.10
			AIKAERO	00.01.39	
			Suhde(%)	56,77	
			Uudessa prosessissa kului vain 42% ajasta prosessin tekoon		

Uudessa järjestelmässä navigointi editoriin on huomattavasti nopeampaa. Vanhassa järjestelmässä hidastavana tekijänä on se, että siinä joutuu ensiksi siirtymään prosessikartalle, jonka kautta oikean konfiguraation löytää vain etsimällä tietoa taulukoista. Prosessikartalta pitää löytää hakuavaimella oikea prosessi taulukosta, jonka jälkeen työtilasta pitää kaivaa esille oikea komponentti. Oikean komponentin löytämisen jälkeen komponenttikenttätaulukosta pitää etsiä oikea rivi editoitavaksi, mieluiten etsimällä kenttää hakupalkista. Uudessa järjestelmässä taas tilaan pääsee äärimmäisen nopeasti sen vuoksi, että järjestelmässä voi laittaa päälle ”kehittäjän tila”-asetuksen. Jos kehittäjän tila on päällä, niin silloin käyttöliittymään lisätään ominaisuuksia, jotka helpottavat kehittämistä tilassa kuin tilassa. Yksi näistä elementeistä on työkaluikoni, jonka avulla pääsee mm. prosessin työtilasta suoraan prosessieditoriin sekä komponenttityötilasta välittömästi komponentin editoriin

Kuva 47 Skenaario 2.



PROSESSI	AIKA (s)	LOPPUI	PROSESSI	AIKA (s)	LOPPUI
Peruutetaan pois ylläpidosta	2	00.02.00	Peruutetaan pois ylläpidosta	2	00.02.00
Katsotaan prosessin oikea nimi, pistetään se mieleen	2	00.04.00	Etsitään näkymästä työkalu-pikalinkki. Painetaan sitä	3	00.05.00
Hypätään päävalikkoon	4	00.08.00	Katsotaan työtilaa muistuttavasta WYSIWYG näkymästä oikea kentt	3	00.08.00
Etsitään prosessivalikosta hakuavaimella	8	00.16.00	Avataan kenttä	1	00.09.00
Selataan valikkoa kunnes rivi löytyy	14	00.30.00			
Valitaan prosessista oikea komponentti	5	00.35.00			
Etsitään prosessin taulukosta oikea komponentti hakusanalla	7	00.42.00			
Klikataan rivi auki	3	00.45.00			
			AIKAERO	00.00.36	
			Nopeus parannus %	80	
			Näkymän vaihdossa kesti vain 20% vanhan järjestelmän ajasta		

8 Yhteenveto

Lopullinen työ oli osoittautunut erittäin laajaksi oppimiskokemukseksi, jonka tärkeimmät piirteet vaihtuivat useamman kerran. Ensiksi työtä ajateltiin sellaisesta näkökulmasta, että teoreettinen osio projektin rakentamisesta olisi tärkeämpää, kuin mitä lopputulokseksi

muodostuu. Projektia rakentaessa huomasin sen tosiasian, että projektin ensisijainen piirre on seuraava: työ pitää suunnitella sen omien olosuhteiden mukaan. Jokaiselle ongelmalle ei löydy yhtä globaalia ongelmia helposti ratkaisevaa hopealuotia. Sen takia omia kehityksen haasteita ei kannatakaan ratkaista sillä tavalla.

Esittämäni ratkaisu projektin luontiin ei ole ainoa eikä oikea ratkaisu kaikille teknologiapaketin tai edes arkkitehtuurin osalta, sillä projekteja on silti vaikea suhteuttaa toisiinsa laadullisesti. Valitsin nämä teknologiat osittain sen vuoksi, että ne ovat minulle tuttuja ja osaan käyttää niitä tehokkaasti, sen vuoksi ne ovat minun tapauksessani hyviä valintoja. Toisinaan jollekin toiselle esimerkiksi MEAN-pino on parempi vaihtoehto (MongoDB, Express, Angular ja Node.js). Se on puhtaasti JavaScriptiin perustuva alusta, jonka vuoksi se on JavaScriptiä osaavalle hyvä vaihtoehto. Jotkut taas pitävät esimerkiksi Pythonista ohjelmointikielenä, jonka vuoksi he voisivat käyttää LAMP-pakettia (Linux, Apache, MySQL, Python). Tässä järjestelmässä esimerkiksi Django ja Flask ovat kelvollisia back-endin teknologioita. Sellainen teknologiapaketti ei vaadi paljon front-endin tietämystä oletusarvoisesti. Djangossa on esimerkiksi helppokäyttöinen käyttöliittymän mallinnuskieli, jonka vuoksi verkkosivun sisältöä voidaan ohjata kokonaan palvelimella SSR-tekniikoilla.

Asetettuihin tavoitteisiin pääsy oli odotettua helpompaa, vaikka kokonaisen sovellusarkkitehtuurin suunnittelu vei ajallisesti enemmän aikaa kuin olin alun perin olettanut. Todellinen lopullinen tuote oli myös huomattavasti pragmaattisempi ratkaisu kuin mitä oletin. Oletin ensiksi, että sovellusarkkitehtuuriin liittyvät pulmat on helpompi ratkaista käytännössä suurin osin ”valmiilla” pohjilla, näin se ei kuitenkaan ole. Todellisuudessa arkkitehtuurin muoto ei tuntunutkaan niin merkitsevältä, sillä kehittämistä enemmän helpottava tekijä oli koodikannan johdonmukaisuus. Hyvä sovellusarkkitehtuuri ehdottomasti helpottaa johdonmukaisuuden saavuttamisessa, mutta kirjan mukainen arkkitehtuuri ei ole johdonmukaisuuden vaatimus. Koodikannan helppokäyttöisyys ja monipuolisuus ohjelmointirajapintana merkitsi myös enemmän, kuin varsinaisen sovellusarkkitehtuurin muoto.

Jatkossa ottaisin hyvin tarkasti suunnitellut enemmän nyrkkisääntönä kuin ehtona, mutta en pidä oppimaani mallia huonona. Siinä on monia hyviä piirteitä, jota ehdottomasti käyttäisin,

kun huomaan niille hyvän tarkoituksen. Tietokannan ja muiden riippuvuuksien pito järjestelmän ulkoisena yksityiskohtana on mielestäni esimerkiksi todella hyvä idea. En koe, että haluaisin välttämättä pitää jotakin tietokantaa jonkin järjestelmän ulkopuolella vaan sen vuoksi, että koen että tietokantaa ei kuulu sotkea ohjelmakoodiin, vaan sen takia, että sitä vasten käytetyt ohjelmistokehykset ovat aina riski. Koen, että käytän tulevaisuudessa enemmän abstraktiota tukena, sillä abstraktiona tuotu riippuvuus on helpompi vaihtaa toiseen, kuin sellainen riippuvuus, joka on juotettu tiukasti järjestelmään.

Abstraktio myös monipuolistaa. Se voi sallia useamman eri ohjelmointikehyksen tai jopa ohjelmointikielen johdonmukaisen yhteistyön. Otetaan esimerkiksi .NET Frameworkin .NET Standard Library: Tämä kirjasto ei tee muuta kuin toimi rajapintana useammalle .NET ekosysteemin teknologialle, mutta siitä huolimatta kaikki .NET Standardia käyttävät ympäristöt pystyvät käyttämään samaa yhteistä infrastruktuuria hyödyksi, huolimatta siitä, että onko kyseessä .NET Framework WPF projekti, Xamarin mobiilisovellusprojekti, tai jokin muu.

Projekti onnistui mielestäni hyvin ja pystyin sen avulla havainnollistamaan vanhan järjestelmän epäkohtia. Projektin demonstraatio on herättänyt paljon keskustelua potentiaalisesti uuden järjestelmän kehittämisestä pienimuotoisena kokeiluna pienelle asiakkaalle, jossa testiympäristö ei olisi kovin suuri. Silloin uutta järjestelmää olisi suhteellisen turvallista kehittää. Mainitsemani ongelmat vanhassa järjestelmässä oli muiden kehittäjien mielestä realistisia, ja esittämäni parannukset ovat harkinnassa uusien toimintojen kehittämisessä. Eräs mahdollinen implementaatio toiminnoille on porras kerrallaan, jossa yksittäiset ominaisuuden implementoidaan yksi kerrallaan. Toinen vaihtoehto on rakentaa vierekkäinen uusi järjestelmä, joka voisi mahdollisesti kommunikoida vanhan järjestelmän kanssa, mutta tämä järjestelmä voisi hitaasti jyrätä vanhan järjestelmän ominaisuuksien päälle.

Käyttöliittymäymmärrykseni on projektin takia kasvanut järkyttävästi, ja olen ymmärtänyt sen vuoksi, että kuinka tärkeää on luoda toiminnasta erillinen käyttöliittymä. Olen pitkään ollut tietoinen siitä, että tiivisti logiikkaan kytketty, käyttöliittymä on rajoittava, sekä vaikeuttaa sen muuttamista jälkeenpäin, mutta sain todellisen kuvan sen haastavuudesta

vasta siinä vaiheessa, kun tein vertailuna uudemman järjestelmän. Silloin sain tietää, että kuinka paljon helpompaa yksittäisiä moduuleja on kehittää. Monet koko pinossa tapahtuvat muutokset ovat sellaisessa järjestelmässä olemattomia. Se voi olla kuitenkin kokeneelle tekijälle haastava askel. Tämä tyyli vaatii perspektiivin muuttamista kehittämisessä. Kompastelu on varmasti välttämätön osa tämänkaltaisen järjestelmän tekemisessä, jos tekniikka ei ole vielä tuttu.

Lähteet

CGI. (2023). *IT4Cargo.* https://www.cgi.com/fi/fi/tuoteratkaisut/it4cargo

Codeacademy. (n.d.). *React: The Virtual DOM* https://www.codecademy.com/article/react-virtual-dom

Mozilla. (2.2.2022). *Manipulating Documents*. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents

Frederick, D. (n.d.). HackerRank. *Top 5 Takeaways from the 2020 Developer Skills Report* https://blog.hackerrank.com/2020-developer-skills-report-top-takeaways/

GeeksForGeeks. (06.12.2021). *Web Development.* https://www.geeksforgeeks.org/web-development/

Gershgorn, D. (29.6.2021). *GitHub and OpenAI launch a new AI tool that generates its own code*. https://www.theverge.com/2021/6/29/22555777/github-openai-ai-tool-autocomplete-code

GitHub. (2023). *GitHub Copilot* https://github.com/features/copilot

Google. (2.2.2023). *Measure performance with the RAIL model*. https://web.dev/rail/

Greif, S.;& Benitte, R. (2020). *Front-end Frameworks*. https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

Harris, R. (27.12.2018). *Virtual DOM is pure overhead. Svelte Blog* https://svelte.dev/blog/virtual-dom-is-pure-overhead

Martin, R. C. (13.08.2012). *The Clean Architecture*. Clean Coder. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.

Chedeau, C (28.09.2016). *Our first 50,000 stars. React.js Blog* https://reactjs.org/blog/2016/09/28/our-first-50000-stars.html#fbolt-is-born

Microsoft. (14.12.2022). *Creating and Configuring a Model*. https://learn.microsoft.com/en-us/ef/core/modeling/

Sakovich, N. (10.2.2023). *Top Most Popular Frontend Frameworks 2023*. Sam Solutions https://www.sam-solutions.com/blog/best-frontend-framework/

Shvets, A. (2014). *Primitive Obsession*.
https://refactoring.guru/smells/primitive-obsession

StackOverflow. (1.5.2021). *2021 Developer Survey*.
https://insights.stackoverflow.com/survey/2021

W3Schools. (2023-a). *Ajax Introduction.*
https://www.w3schools.com/js/js_ajax_intro.asp

W3Schools. (2023-b). *What is a Front-End Developer?*
https://www.w3schools.com/whatis/whatis_frontenddev.asp

Liite 1: Selitteet

Avain	**Selite**
API	Sovellusrajapinta. Ympäripyöreä termi, mutta verkkosovellusten kontekstissa tämä usein tarkoittaa palvelimen ja selaimen välistä liittymäpintaa, jonka kautta suoritetaan toimintoja
Back-end	Palvelimella tapahtuva logiikka. Käynnistää toimintoja usein käyttäjän pyynnöstä, josta luodaan projektiona usein HTTP-vastauksen muotoinen tulos takaisin käyttäjälle.
C#	Monikäyttöinen olio-ohjelmointikieli. Voidaan käyttää useampaan tarkoitukseen, sillä sitä voidaan käyttää verkko-ohjelmoinnin lisäksi työpöytä- tai peliohjelmointiin.
Clean Architecture	Eräs sovellusarkkitehtuurimalli. On todellisuudessa aggregoitu monesta edeltävästä mallista, mutta se koostaa niiden edellisten mallien esittämät ideat uuden nimen alle. Pääpainona sen rakenteelle on kaikesta muusta logiikasta eroteltu bisneslogiikka, eli ohjelman ydin.
Deklaratiivinen ohjelmointi	Kuvaa mitä tehdään ohjelmakoodilla, mutta ei kuvaa miten se tehdään. **HTML** on deklaratiivista
Front-end	Loppukäyttäjän käyttöliittymä. Käyttäjän ruudulla tapahtuva ohjelmalogiikka
HTML	Verkkosivujen kuvauskieli. Selain tulkitsee HTML-sivun selaimen ruudulle selattavaksi verkkosivuksi.
Imperatiivinen ohjelmointi	Kuvaa miten ohjelmakoodilla määritelty toiminta tehdään, esimerkiksi käyttämällä ehtorakenteita. **JavaScript** ja **C#** on imperatiivista.

JavaScript	Käyttöliittymällä funktionaalista logiikkaa ohjaava ohjelmointikieli, jonka avulla voidaan ohjata toimintaketjuja selaimella.
MySQL	Yksi monesta relaatiotietokannoista. **SQL** on vakiintunut tapa tallentaa muuttuvaa ja toisiinsa relaatioilla kytkettyä dataa palvelimilla
ORM	Object Relational Mapper. Tätä käytetään heijastamaan kannasta dataa (sekä heijastamaan dataa takaisin kantaan), jotta sitä olisi helppo käsitellä lähdekoodissa.
Ohjelmistokehys/Sovelluskehys	Kehittäjän työtä helpottava rakenne, jonka avulla voidaan sukkelasti rakentaa ominaisuuksia, joista usein kaikista teknisintä osaa ohjaa sovelluskehys. Usein vaatii oman syntaksin.
State	Selaimella muuttuvien arvojen kokonaisuus, joka vaikuttaa **front-end** kehysten käyttöliittymäelementtien sisältöön