Kasperi Mutku

Vedenkulutuksen seurantasovellus hyödyntäen Vanilla JavaScriptiä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

31.10.2021

Tiivistelmä

Tekijä:	Kasperi Mutku
Otsikko:	Vedenkulutuksen seurantasovellus hyödyntäen Vanilla JavaScriptiä
Sivumäärä:	48 sivua
Aika:	31.10.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mediatekniikka
Ohjaaja:	Lehtori Toni Spännäri

Insinöörityön tarkoitus oli toteuttaa automaatio- ja tietotekniikka-alalla toimivalle yritykselle sovellus, jonka avulla loppukäyttäjä pystyy tarkastelemaan asunnon vedenkulutusta.

Tavoitteena oli, että sovelluksen ominaisuudet ja lähdekoodi olisi mahdollisimman hyvin automatisoitu tekemään asiat, joita kehittäjät joutuisivat käsin suorittamaan, kun sovellusta ylläpidetään ja hallinnoidaan. Toinen tavoite oli myös, että sovelluksen kehittämisessä käytettäisiin mahdollisimman vähän kolmannen osapuolen koodia minimoimalla kirjastojen ja ohjelmistokehyksien käyttöä.

Työn ohjelmointitoteutuksessa käytettiin selainpuolella Vanilla JavaScriptiä. Palvelinpuolella ohjelmointikielinä käytettiin Node.js sekä Javaa. Työssä hyödynnettiin myös yrityksen valmista rajapintaa, johon osa lähdekoodin toiminnallisuudesta perustui.

Toteutustapana sovellusta kehitettiin yhdessä yrityksen kanssa. Tämä sisälsi palavereja, joissa yhdessä mietittiin, mitä ominaisuuksia lisätään tai otetaan pois senhetkisestä versiosta. Työnkulku eteni näissä palavereissa, ja lopulta projektista saatiin valmis lopputulos, johon oltiin tyytyväisiä. Työn tuloksissa saatiin toteutettua lähes kaikki tavoitteet, jotka oli tarkoitus saada tehtyä, kun projektia suunniteltiin.

Avainsanat: JavaScript, Node.js, Vanilla JavaScript, moderni web-sovelluskehitys

Abstract

Author:	Kasperi Mutku
Title:	Water metering application made by Vanilla JavaScript
Number of Pages:	48 pages
Date:	31 October 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Media Technology
Supervisor:	Toni Spännäri, Senior Lecturer

The purpose of this thesis was to create an application about water metering, where the client can examine the metering results. The application was created for a company that operates within automation and information technology industry.

The goal was to achieve as much automation as possible for this application. The purpose of this automation was that the developer does not need to manage this application, as the automation aspect of the source code will do the steps that a human should otherwise do manually. Another goal was to minimize the third-party source code. This was done by minimizing the use of the libraries and frameworks.

The programming languages of the application were decided during the planning phase of the project. The frontend was created with a Vanilla JavaScript. The backend on the other hand was done with Node.js and Java. There was also an API created by the company, that was used in this project.

The implementation of this project was done by cooperating with the other workers of the company. There were meetings where the decisions of the current version of the application were done. The decisions included which features were good and which not, and the conclusions of pros and cons kept the developing working going on.

The outcome of the project was very good, as it filled almost all the standards that were set for this application.

Keywords: JavaScript, Node.js, Vanilla JavaScript, Modern Web-development

Sisällys

Lyhenteet

Runtime: Järjestelmä, joka antaa ohjelman suorittaa komentoja ohjelmaa suoritettaessa.

Skriptikieli: *Scripting Language*. Ohjelmointi- ja runtime-kieli, jossa koodi suoritetaan samaan aikaan, kun ohjelmaa suoritetaan, sen sijaan että koodi käännettäisiin ensin ohjelmaksi.

HTTP: Hypertext Transfer Protocol. Verkossa toimiva protokolla, jolla voidaan siirtää selaimeen tai palvelimelle dataa.

REST API: Representational state transfer (REST). Arkkitehtuurimalli ohjelmointirajapintojen kehittämiseen. API (Application programming interface), REST API antaa kehittäjälle mahdollisuuden ohjelman kommunikoida ja hakea dataa HTTP-protokollan kautta.

Node.js: Javascript-pohjainen ajoympäristö ohjelman suorittamiseen palvelimella.

Open Source: Avoin lähdekoodi. Kehitysmenetelmä, jolla ohjelmoijat saavat tutustua ohjelman lähdekoodiin ja muokata sitä tarpeidensa mukaan.

HTML: Hypertext Markup Language. Verkossa standardoitu kuvauskieli, jolla voidaan kuvata hyperlinkkejä sisältävää kuvatekstiä.

JSON: JavaScript Object Notation. Avoimen standardin tiedostomuoto tiedonvälitykseen. Ei ole JavaScriptista riippuvainen.

DOM: Document Object Model. Puurakenne esimerkiksi HTML-tiedostoon. Sen avulla dokumentissa olevat elementit kommunikoivat toistensa kanssa.

Endpoint: Palvelimen tai rajapinnan päätepiste, jonka kautta järjestelmät pystyvät käyttämään toimintojensa suorittamiseen tarvittavia resursseja.

Merkkijono: Useissa ohjelmointikielissä käytetty tietotyyppi, johon on järjestetty jono peräkkäisiä merkkejä, yleensä kirjaimia.

Deployment: Prosessi, jossa sovellus tuodaan kehittäjiltä loppukäyttäjille.

SEO: Search Engine Optimization, eli hakukoneoptimointi. Prosessi, jolla pyritään saamaan sivulle näkyvyyttä hakukoneilla, kuten Googlella.

Crawler: Verkossa suoritettava, yleensä automatisoitu ohjelma, joka kerää erilaista tietoa verkkosivusta.

NPM: Paketinhallintaohjelma, jolla voidaan ladata Javascript-kirjastoja suoraan komentoriviltä.

Single Page App: Web-sovellus tai -sivu, joka uudelleenkirjoittaa saman sivun HTML-sisällön käyttäjän interaktiossa. Single Page Appilla ei ladata sisältöä siirtymällä uudelle sivulle.

Branch: Haarautuminen. Versionhallinnassa käytetty termi, jossa objektista/koodista luodaan kopio.

SSR: Server-Side Rendering. Prosessi, jossa palvelin tuottaa HTML-tiedostot selaimeen.

Commit: Sitoutuminen. Versionhallinnassa käytetty termi, jossa lähdekoodin muutokset arkistoidaan pääversioon.

1 Johdanto

Insinöörityön tarkoitus oli toteuttaa COBA International Oy:lle vedenkulutuksen seuraamiseen hyödynnettävä sovellus, jonka avulla loppukäyttäjä voi tutkia kerättyä dataa monipuolisesti. Tavoitteena oli kehittää sovellukseen mahdollisimman hyvin automatisoitu logiikka, joka helpottaisi sen ylläpitämistä kehittäjien näkökulmasta. Toinen tavoite liittyi myös ylläpidon helpottamiseen siten, että käytettäisiin mahdollisimman vähän kolmannen osapuolen koodia, ettei kehittäjien tarvitse päivittää erilaisia valmiita kirjastoja, jotka mahdollisesti vaikuttavat sovelluksen toiminnallisuuteen.

Moderni ohjelmistokehitys keskittyy yhä enemmän web-ohjelmointiin nykyisen Internetin jatkuvasti laajentuessa ja mukautuessa erilaisiin laitteisiin. Tämä laajentuminen antaa web-ohjelmoijille mahdollisuuden tehdä koko ajan omaperäisempiä ratkaisuja, kun laitteet pystyvät yhdistämään Internetiin ja kommunikoimaan verkossa. Pelkästään jo laitteisiin, joissa on jonkinlainen näyttö, pystytään modernilla web-ohjelmoinnilla toteuttamaan ulkoasun skaalautuminen sekä sommittelu näyttämään hyvältä, toisin kuin vielä 2000-luvun alussa, jolloin ulkoasu ja käyttöliittymä oli samanlainen laitteen näytön resoluutiosta riippumatta. [1.]

Web-ohjelmoinnin suosion nousu on saanut ohjelmoijat kehittämään erilaisia vapaan lähdekoodin ohjelmistokehyksiä ja kirjastoja, joita muut ohjelmoijat voivat käyttää projekteissaan.

Kaiken ytimenä ja tärkeimmässä roolissa modernissa web-kehityksessä on kuitenkin JavaScript. Se on skriptikieli, jolla voidaan lisätä web-sivulle dynaamista toiminnallisuutta. JavaScriptiä käyttää arviolta 94,5 % verkossa olevista sivuista. [2.] JavaScriptin ympärille onkin kehitetty juuri paljon suosittuja ohjelmistokehyksiä. Nämä jatkuvassa kehityksessä olevat ohjelmistokehykset (engl. framework), on rakennettu Vanilla JavaScriptin pohjalta, jota tässä insinöörityöprojektissa käytetään pääasiassa.

Työssä vertaillaan myös, mitkä ovat hyviä ja huonoja puolia, kun web-ohjelmoinnin asiakaspään kieleksi valitaan Vanilla JavaScript ja pyritään minimoimaan mahdollisimman paljon kolmannen osapuolen koodia.

Selainpuolen lisäksi tarkastellaan myös palvelinpuolen JavaScript-ohjelmointia, jossa pyritään mahdollisimman hyvin automatisoimaan sovelluksen hallintaan liittyvät tehtävät, joita ihminen joutuisi mahdollisesti tekemään käsin. Tämä automatisointi käydään vaiheittain läpi tässä työssä.

Työhön liittyy myös oma JavaScript-pohjainen kirjasto, jonka toiminnallisuus selostetaan näyttäen siihen liittyvät kuvat.

2 Moderni web-ohjelmointi

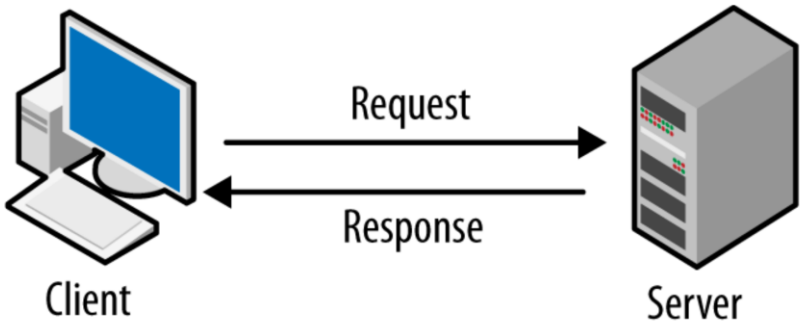
2.1 Selainpuoli ja palvelinpuoli

Web-sovellus tai -sivu perustuu asiakas-palvelinarkkitehtuuriin (kuva 1), jossa ohjelman rakenne on sidonnainen kahteen osapuoleen:

1. Selainpuoli on asiakkaalle näkyvä, interaktiivinen osa sovelluksesta. Siitä voidaan käyttää myös nimeä asiakaspää tai selain.

2. Palvelinpuoli on palvelimella suoritettava osuus sovelluksesta, ja se ei ole avoin loppukäyttäjälle. Palvelinpuolelta voidaan myös ottaa yhteys tietokantaan, johon voidaan tallentaa dataa ja hakea tämä data, joka voidaan lähettää takaisin loppukäyttäjälle.

Nämä kaksi päätä kommunikoivat keskenään koko ajan. Kun loppukäyttäjä esimerkiksi kirjautuu sisään sovellukseen, hän kirjoittaa sovellukseen käyttäjänimen ja salasanan, jotka ovat ruudulla HTML-elementtilomakkeen muodossa yleisimmin. Kun hän painaa ”Kirjaudu sisään”, lähetetään nämä lomakkeen tiedot palvelimelle käsiteltäväksi. Jos tiedot täsmäävät palvelinpään autentikointijärjestelmän kanssa, palauttaa palvelinpuoli takaisin asiakaspäähän tiedon, että kirjautuminen onnistui ja käyttäjä pääsee eteenpäin sovelluksessa.



Kuva 1. Asiakas-palvelinarkkitehtuurin mallinnus [3].

Yleensä selainpuoli koostuu yhdestä tai useammasta HTML-tiedostosta, joihin on lisätty JavaScriptiä mukaan. HTML-tiedoston elementit yhdistää DOM (Document Object Model), jonka avulla nämä elementit pystyvät kommunikoimaan toistensa kanssa.

DOM on ohjelmointirajapinta, joka mahdollistaa verkossa olevien dokumenttien, kuten XML- tai HTML-tiedostojen, puurakenteen muodostavien objektien tarkastelun ja muokkaamisen. DOM-manipulaatio taas tarkoittaa sitä, että verkkotiedostojen rakennetta, tyyliä ja sisältöä voidaan ohjelmallisesti muokata. JavaScript on ohjelmointikieli, jolla voidaan toteuttaa selaimessa DOM-manipulointia. HTML-tiedoston asiakaspuolen DOM-muutoksien ja web-sovelluksen ulkoasun elementtien lisäksi JavaScriptin avulla voidaan kommunikoida myös palvelimen kanssa. Tämä kommunikaatio mahdollistaa datan lähettämisen palvelimelle, sen käsittelyn siellä sekä tämän käsitellyn datan uuden arvon palauttamisen takaisin asiakaspäähän. Ilman JavaScriptiä ei sovellus pysty olemaan yhteydessä palvelimeen.

Ohjelmointikielien valitseminen

Kun web-ohjelmointiprosessi aloitetaan, valitaan työkalut molempiin päihin: sekä selainpuoleen että palvelinpuoleen. Nykyisin web-sovelluksen selainpuolen työkalut ovat juuri JavaScript tai sen ympärille rakennettuja ohjelmistokehyksiä.

Palvelinpuoli koodista voi puolestaan sisältää erilaisia ohjelmointikieliä, kuten Java, C#, Python tai PHP. Nykyään myös palvelinpuolella voidaan käyttää palvelinpään kielenä JavaScriptiä sen palvelinohjelmointiin tarkoitetulla Node.js:llä, joka kehitettiin vuonna 2009. Node.js mahdollistaa JavaScriptin suorittamisen palvelimella, ja se on avointa lähdekoodia. [4.]

Myös palvelinpuolelle on kehitetty ohjelmistokehyksiä. Näistä yleisimpiä työkaluja ovat esimerkiksi Express Node.js:lle, Spring Javalle tai .NET C#:lle. [5.]

2.2 Ohjelmistotuotanto

Ohjelmistotuotantoprojektin kehityskaari alkaa suunnittelusta ja päättyy aina valmiiseen lopputuotteeseen asiakkaalle. Projektin alussa ohjelmointiaspektin lisäksi on monta muuta tekijää, jotka tulee ottaa huomioon, jotta saavutetaan paras ja haluttu lopputulos. Ohjelmointi itsessään ei ole ainoa osuus sovelluskehitystä, vaan tämä prosessi, sovelluskehitys, on paljon laajempi. Kokonaiskuva kehityskaaresta sekä projektin kulku on prosessi, joka yhdistää prosessiin ohjelmoinnin lisäksi

- ulkoasun ja grafiikan suunnittelua
- versionhallintaa
- testausta ja virheiden etsimistä
- prototyyppien tekemistä
- ylläpitoa ja dokumentointia.

Ohjelmointivaiheen valmistuttua siihen pisteeseen, että sovellus on julkaistu loppukäyttäjille, sitä ei voida jättää vain pyörimään, vaan sovellusta täytyy jatkuvasti ylläpitää. Valmiista tuotteesta löytyy ennemmin tai myöhemmin virheitä, joita pitää korjata.

Toinen mahdollinen skenaario on myös, että asiakas toivoo uutta ominaisuutta sovellukseen. Tässä kohtaa versiohallinnan ja dokumentoinnin tärkeys tulee kriittiseksi. Jos sovellusta ei ole dokumentoitu kunnolla tai lähdekoodi on tehty vaikeasti ymmärrettäväksi kehittäjälle, jatkokehitys vaikeutuu huomattavasti.

Suurissa projekteissa, joiden kehitys kestää vuosia, saattavat sovelluksen ohjelmoijat vaihtua useaan kertaan. Vaihtoehtoisesti myös voi tulla tilanne, että alkuperäinen kehittäjä ei koske sovelluksen lähdekoodiin pitkään aikaan, jolloin hän saattaa unohtaa tärkeitä asioita ohjelman lähdekoodin ominaisuuksista. Tässä kohtaa huonosti dokumentoidun sovelluksen kehitys tulee hankalaksi.

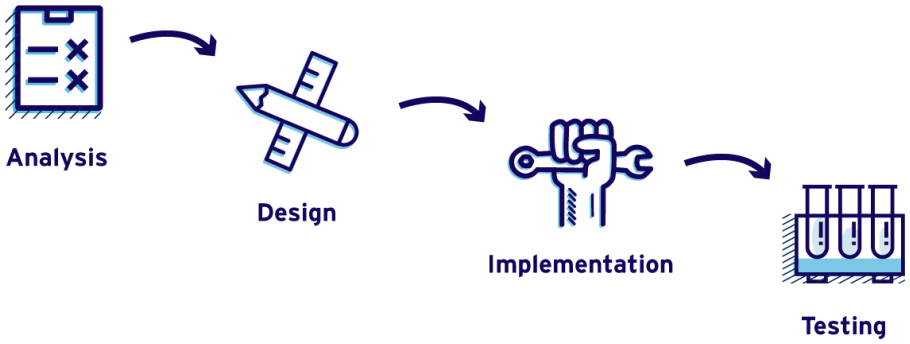
Kun sovelluksen ohjelmointiin osallistuu enemmän kuin yksi henkilö, on tärkeää, että kaikki tekevät kehitystyötä samalla lähestymistavalla. Jos toinen tiimiläinen käyttää eri tapaa kuin toinen, tulee sovelluksen ylläpidosta vaikeaa. Esimerkiksi erilaisilla formaateilla ohjelmoitu lähdekoodi aiheuttaa konflikteja kehityksessä. Silloin kun ohjelmointikehitys on organisoitu hyvin ja tiimiläiset ohjelmoivat samoilla standardeilla ja formaatilla, saadaan lopputulos, jota on helppo ylläpitää ja jatkokehittää.

Ohjelmistotuotannon menetelmät

Sovelluskehityksessä voidaan käyttää erilaisia menetelmiä, joilla pyritään saavuttamaan paras mahdollinen lopputulos organisoimalla tiimiä tietynlaisilla käyttäytymismalleilla. Tällaisia menetelmiä ovat esimerkiksi

- ketterä ohjelmistokehitys
- prototyyppimenetelmä
- vesiputousmalli.

Vesiputousmallissa (kuva 2) ohjelmistokehitys etenee vaihe vaiheelta alaspäin, kuten vesiputous. Tavoitteena on tehdä mahdollisimman tarkka analyysi ja tutkimus prosessin edetessä, jotta riskit voidaan minimoida. [6.] Vesiputousmalli on joustamaton ja raskas, mikä johtuu siitä, että jokainen vaihe tehdään aina kokonaan valmiiksi ennen seuraavaan siirtymistä.



Kuva 2. Vesiputousmallin vaiheet [6].

Toinen yleisin ohjelmistotuotannon menetelmä on ketterä ohjelmistokehitys. Tässä lähestymistavassa on tarkoitus, että sovellusta ei rakenneta kerralla alusta loppuun, vaan kehitystyö tehdään palasina pienissä intervalleissa lisäten pikkuhiljaa uusia ominaisuuksia projektiin. Yleisesti nämä intervallit ovat 1–4 viikon pituisia. Niiden tarkoituksena on saada toimiva ohjelmisto minimoimalla riskit, kun projektia tehdään pienissä osissa, jolloin mm. virheet on helpompi korjata. [7.]

2.3 Suunnittelusta valmiiseen sovellukseen

Ohjelmistotuotanto prosessina alkaa aina ideasta, joka tulee joko suoraan asiakkaalta tai se voi olla ohjelmoijan itse keksimä idea. Kun kehittäjällä on valmis idea, alkaa suunnittelutyö, jossa mietitään isompaa kokonaisuutta sovelluksesta. Tässä vaiheessa tulee jo ottaa huomioon mahdollisen asiakkaan toiveet. Hyvä tapa on tehdä lista pienemmistä ominaisuuksista, joita sovellus tulee pitämään sisällään. Näistä pienistä osista saadaan rakennettua lopulta isompi kokonaisuus.

Suunnittelu

Suunnitteluprosessi itsessään ei vaadi ohjelmointia, mutta voi sisältää pseudokoodia, mikä tarkoittaa sitä, että kirjoitetaan muistiin vain tietyn ominaisuuden

perusrakenteen algoritmi ilman ohjelmointikielien syntaksia. Tässä vaiheessa myös kartoitetaan, mitkä työkalut ja ohjelmointikielet sopisivat mahdollisesti parhaiten haluttuun lopputulokseen. Myös sovelluksen ulkoasua voidaan hahmotella ja piirtää luonnoksia ulkoasusta. Isommissa organisaatioissa tiimissä on usein erillinen graafikko tai käyttöliittymäsuunnittelija, joka suunnittelee sovelluksen ulkoasun.

Kun suunnitteluvaihe on valmis, alkaa tekninen osuus, jossa projekti alustetaan ja ohjelmointityö voidaan aloittaa. Riippuen organisaatiosta ja projektista, tässä vaiheessa yleensä otetaan käyttöön ohjelmistotuotannon menetelmiä projektin etenemisen edistämiseksi. Tämä ei kuitenkaan ole välttämätöntä, sillä erilaisia menetelmiä voidaan kokeilla myös kesken projektin kehitysvaiheen.

2.4 Versionhallinta

Ohjelmointiosuus itsessään ei sisällä pelkästään koodin kirjoittamista, vaan myös versionhallintaa. Oli projekti toteutettu sitten tiimissä tai yksin, on sovelluksen ja lähdekoodin ylläpidon kannalta erittäin kriittistä pitää yllä hyvää versionhallintaa.

Tämä tarkoittaa käytännössä sitä, että lähdekoodia voidaan ajatella kuin kasvavana puuna. Ohjelmoija kehittää lähdekoodia ja puu kasvaa samalla. Sitten ohjelmoija haluaa saada valmiiksi tietyn ominaisuuden tai ottaa varmuuskopion nykyisestä koodista. Hän ikään kuin tallentaa kohdan, jossa hän on tällä hetkellä. Puu haarautuu ja siihen kasvaa oksa. Ohjelmoija saattaa kehittää ominaisuutta, jolloin oksa kasvaa. Hän ei välttämättä ole tyytyväinen lopputulokseen ja palaa kohtaan, jossa oksa haarautui, ja kasvattaa uuden oksan. Jos ohjelmoija on tyytyväinen lopputulokseen, hän voi tallentaa sen hetken lähdekoodin, jolloin oksa yhdistyy puun runkoon. Tähän kohtaa runkoa voidaan kirjoittaa muistiin mitä tässä kohdassa on tapahtunut. Ohjelmoija voi aina palata haaraumiin tai oksiin, mutta viimeisin tuotos sovelluksesta on kasvavassa päärungossa.

Lyhyesti sanottuna, kehittäjän ei tarvitse pelätä, että hän tekee jotain peruuttamatonta, koska versionhallinta hoitaa varmuuskopioinnin. Tämä antaa myös ohjelmoijalle varmuutta kokeilla erilaisia lähestymistapoja kehitykseen, kun voi palata takaisin versioissa. [8, s. 26.]

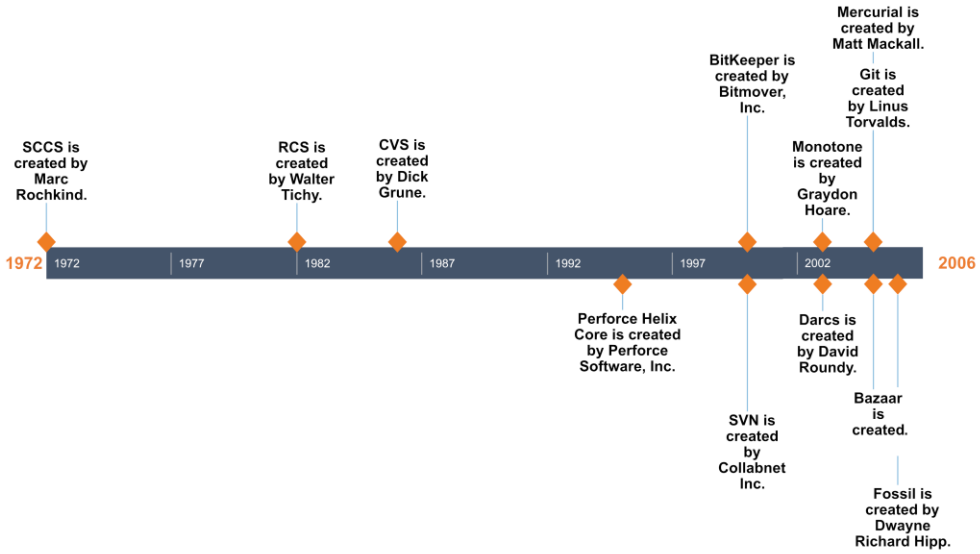
Versionhallinta on erittäin tärkeä osa ohjelmointiprosessia, koska sillä voidaan tehdä lähdekoodista helposti jatkokehitettävää, palata takaisin aiempiin versioihin ja tiimityöskentelyssä se helpottaa huomattavasti eri koodiversioiden yhdistämistä isommaksi kokonaisuudeksi.

Versionhallintajärjestelmät

Sovelluksen lähdekoodin versionhallintaan löytyy useita eri vaihtoehtoja. Ne ovat yleensä versionhallintajärjestelmiä, mutta versionhallintaa voi toteuttaa myös ottamalla varmuuskopion senhetkisestä lähdekoodista ja tallentaa se uudeksi tiedostoksi. Versionhallintajärjestelmiä ovat esimerkiksi

- Git
- CVS
- SVN
- BitKeeper
- Mercurial.

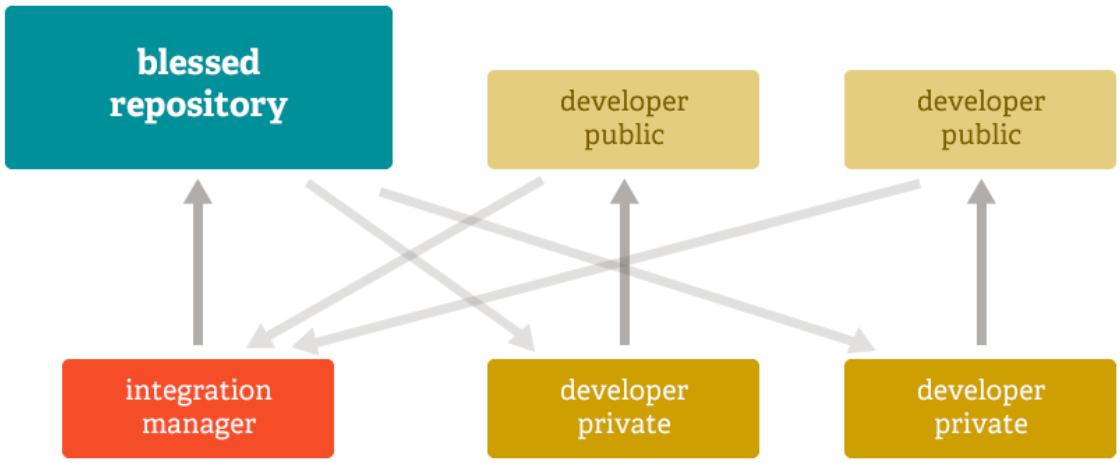
Versionhallintajärjestelmät jakautuvat kolmeen polveen. Ensimmäinen versionhallintajärjestelmä on ensimmäisen polven Source Code Control System (SCCS), jonka kehitti Marc J. Rochkind vuonna 1972 [9.] Tästä eteenpäin on versionhallintajärjestelmien historiassa kehittynyt uudempia järjestelmiä, kuten kuvasta 3 voi nähdä.



Kuva 3. Erilaisten versionhallintajärjestelmien historian aikajana [10].

Git ja sen suosio

Git on modernin ohjelmistokehityksen suosituin versionhallintajärjestelmä. Se kuuluu kolmannen sukupolven versionhallintajärjestelmiin, ja se on vapaata lähdekoodia. Gitin loi Linux Torvalds vuonna 2005 pääasiassa C-kielellä ja Shell-skripteillä [9]. Yksi yleinen Gitin avulla käytetty työnkulku on ”Integration Manager Workflow” (kuva 4), jossa yksi kehittäjä tallettaa muutokset pääversioon, josta toiset voivat ladata version. [10.]



Kuva 4. Integration Manager Workflow [10].

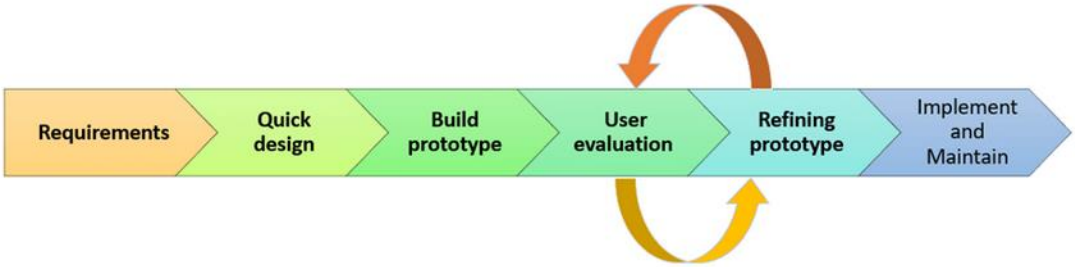
Gitin nopeus, joustavuus ja ominaisuudet tekevät siitä erinomaisen versionhallintajärjestelmän. Sillä voi työskennellä lokaalisti offline-tilassa, ja muutokset koodiin on mahdollista ladata myöhemmin verkkoyhteydellä sovelluksen pääversioon. Muista versionhallintajärjestelmistä Git poikkeaa sen ”branching”-mallinnusominaisuuden osalta. Se rohkaisee ohjelmoijaa tekemään useita brancheja, joilla voi esimerkiksi kokeilla jotain ideaa ja palata nopeasti takaisin, jos ei ollut tyytyväinen. Tämä mallinnus antaa kehittäjälle mahdollisuuden tehdä brancheja, jotka menevät tuotantoon tai pelkästään testiin itse kehittäjälle. [10.]

GitHub ei ole sama asia kuin Git, vaan se on verkkosivu, joka mahdollistaa lähdekoodin varastoimisen ja versionhallinnan käyttämällä Gitiä.

2.5 Prototyypit

Ensimmäinen valmis versio sovelluksesta ei ole välttämättä viimeinen versio loppukäyttäjälle tuotannossa. Kun sovellus on saatu vietyä pidemmälle vaiheeseen, jossa se vastaa haluttua lopputulosta, saadaan ensimmäinen prototyyppi.

Tässä vaiheessa prototyyppi voidaan antaa asiakkaalle kokeiluun, jolloin asiakas testaa sovellusta ja antaa palautetta sen puutteista ja hyvistä puolista. Palautteen perusteella kehittäjät lisäävät tai poistavat ominaisuuksia, joita loppukäyttäjä haluaisi sovellukseen tai siitä pois. Tämä sykli on ”Prototyping Model” (kuva 5), jonka hyöty on paras silloin, kun asiakas ei ole aivan varma, mitä haluaa lopullisesta versiosta. [11.]



Kuva 5. Prototyping Model -syklin vaiheet [11].

Prototyyppejä voi tehdä niin kauan, kunnes päästään asiakkaan kanssa lopputulokseen, jossa hän on tyytyväinen.

Testaus

Testausta tehdään koko ohjelmistotuotantoprosessin ajan. Sen tarkoituksena on löytää mahdolliset bugit eli lähdekoodissa olevat virheet, jotka voivat rikkoa ohjelman toiminnallisuuden, jotta sovellus toimisi moitteettomasti.

Testausta voi tehdä joko manuaalisesti käyttämällä sovelluksen prototyyppiä niin, että tarkoituksena on ”rikkoa” sovellus, jolloin voidaan kirjoittaa muistiin, mikä asia aiheutti epätoivotun lopputuleman. Tiimissä voi olla erikseen henkilö, joka on testaaja, tai sitten ohjelmoija voi itse tehdä tämän toimenpiteen. Muistiin kirjattujen virheiden syy pyritään testauksen jälkeen löytämään lähdekoodista.

Vaihtoehtoisesti testauksen voi automatisoida ”Unit Testillä”, joka eroaa manuaalisesta testauksesta siten, että tietokone tarkastaa lähdekoodia sen sijaan, että ihminen manuaalisesti käyttää interaktiivista sovellusta testaukseen. Unit Test suoritetaan, kun prototyyppi on koottu, eli lähdekoodista luodaan suoritettava sovellus, ja Unit Test tarkastaa lähdekoodista, toimiiko koodin funktionaalisuus niin kuin pitäisi. Mikäli se löytää virheen, ei testi päästä tätä prototyypin koontia läpi, jolloin ohjelmoijan täytyy muuttaa lähdekoodia niin, ettei virhettä löydy. [12.]

2.6 Valmis lopputuote, ylläpito ja dokumentointi

Kun prototyyppi menee tuotantoon ja loppukäyttäjälle, jatkuu ohjelmistokehitys sovelluksen ylläpidolla. Tässä vaiheessa voidaan tehdä päivityksiä ja korjata mahdollisia löytyneitä virheitä, joita loppukäyttäjät voivat ilmoittaa kehittäjälle.

Ymmärrettävän koodin tärkeys

Lähdekoodin siistiminen on tässä vaiheessa viimeistään parasta tehdä. Tämä siksi, että voi tulla vastaan tilanne, jossa sovelluksen kehitys laitetaan tauolle ja

lähdekoodiin ei kosketa pitkään aikaan. Pitkän tauon jälkeen sovelluksen muuttaminen myöhemmin hankaloituu huomattavasti, jos koodi on vaikeasti luettavaa kehittäjälle itselleen, tai vielä vaikeampaa, jos vastuu kehitystyöstä siirtyy toiselle ohjelmoijalle.

Lähdekoodia voi siistiä esimerkiksi nimeämällä muuttujat, luokat ja metodit niin, että niiden nimet vastaavat niiden funktionaalisuutta. JavaScriptissä lähdekoodin syntaksi toimii niin, että ohjelma suoritetaan rivi riviltä alaspäin lukien lähdekoodia (kuva 6).

```
1   let a = ”Hello world!”;
2   let helloWorld = ”Hello world”;
3
4
5   const printText = (text) => {
6       console.log(text);
7   }
8
9   //Tämä funktio tulostaa ’Hello world’
10  printText(helloWorld);
```

Kuva 6. Koodiesimerkki JavaScriptin syntaksista.

Kuvan 6 esimerkkikoodissa ”a”-merkkijono tulostaa ”Hello world”, mutta muuttujan nimi ei ole mitenkään yhdessä sen tulostuksen kanssa. Muuttuja ”helloWorld” taas antaa saman tien jonkinlaisen käsityksen siitä, mihin tämä muuttuja liittyy.

Lähdekoodin kommentointi

Koodin nimeämisen lisäksi kommentointi on myös hyvä tapa antaa ohjelmoijalle tietoa siitä, mitä jokin tietty kohta koodissa tekee. Kommentti lähdekoodissa tarkoittaa selkokielellä luettavaa tekstiä, joka jätetään pois, kun sovellusta suoritetaan. Esimerkiksi rivi // on kommentti, josta ohjelmoija voi lukea nopeasti, mitä tapahtuu, kun alla oleva funktio suoritetaan.

Projektin dokumentoiminen

Lähdekoodin siistimisen lisäksi olisi hyvä tehdä kehittäjille laaja dokumentaatio sovelluksesta. Dokumentointi eroaa versionhallinnasta ja lähdekoodin siistimisestä siten, että siinä ei kosketa suoraan koodiin, vaan kirjoitetaan esimerkiksi Word-tiedostoon muistiinpanoja sovelluksesta. Tämä dokumentaatio voi kattaa seuraavat asiat:

- tieto siitä, miten sovellus alustetaan ja kootaan asiakkaalle
- lista päätepisteistä ja mikä niiden funktio on sovelluksessa
- kuvakaappauksia sovelluksesta
- ”askel askeleelta” -tyyppinen ohjeistus, mitä tapahtuu missäkin kohtaa koodia suorittaessa
- kuvaus lähdekoodin luokista ja siitä, mitkä ovat tärkeimpiä luokkia.

Kuten lähdekoodin versionhallinta, myös dokumentaatiota tulisi pitää ajan tasalla, kun sovelluksen koodia muutetaan tai siihen lisätään uusia ominaisuuksia.

3 Vanilla JavaScript ja sen käyttäminen ohjelmistotuotannossa

3.1 Vanilla JavaScript

Koska monet JavaScript-ohjelmistokehykset pitävät nimessään sanan JavaScript tai JS, kuten esimerkiksi ReactJs, voidaan helposti luulla, että Vanilla JavaScript on Reactin kaltainen kirjasto tai ohjelmistokehys. Tämä ei pidä paikkaansa, sillä Vanilla JavaScript on puhdasta JavaScriptiä ilman ylimääräisiä kirjastoja. Sana ”Vanilla” on tietojenkäsittelyssä termi, jolla viitataan tietokoneohjelmiston alkuperäiseen muotoon. [13.]

Historiaa

JavaScriptin kehitti Netscapen Brendan Eich vuonna 1996. Moni sekoittaa JavaScriptin Javaan, mutta nämä kaksi ohjelmointikieltä ovat aivan erilaisia. Alun

perin JavaScript nimi oli LiveScript, mutta markkinointisyistä nimi muutettiin JavaScriptiksi, kun Netscape liittoutui Java-kielen kehittäneen Sun Microsystemsin kanssa. [15.]

Ajan kuluessa JavaScript on kielenä kehittynyt ja siitä on julkaistu yhdeksän uutta versiota. Näiden versioiden nimien etuliitteessä on ES ja perässä numero. Ensimmäinen versio on ES1, ja virallinen nimi tälle on ECMAScript 1. Viimeisin versio JavaScriptistä on kehitetty vuonna 2018, ja sen virallinen nimi on ECMAScript 2018. [16.]

Versioiden päivittyessä on kieleen lisätty erilaisia uusia ominaisuuksia, kuten luokat, nuolifunktiot, asynkroninen kommunikaatio ”promiseilla” ja asynkronisten funktioiden ”await”. Myös alkuperäinen muuttujan alustaminen ”var”, on muutettu uusissa JavaScript-versioissa muotoon ”let” ja ”const”. Let tulee englannin kielen sanasta ”let”, joka verbinä ”letting” tarkoittaa ”allow”, eli JavaScript-ohjelmoinnissa se alustaa muuttujan tyypiksi, jota voidaan muuttaa jälkikäteen. Const tulee sanasta ”constant”, ja sillä määritellään muuttuja, joka ei muutu vaan pysyy samana.

Näitä versioita kehittävät JavaScript-kehittäjät, ohjelmistokehyksien luojat, ohjelmointikielten tutkijat ja vaikutusvaltaiset JavaScript-ohjelmoijat. He pitävät säännöllisesti kuusi palaveria vuodessa kolmen päivän ajan, ja niissä puhutaan JavaScriptin kehityksestä. [13.]

Yleistä JavaScript-ohjelmoinnista

JavaScript on skriptikieli, ja sen lähdekoodi tallennetaan ’.js’-muodossa olevaan tiedostoon. JavaScript-koodia ei käännetä erikseen omaksi konekieliseksi ohjelmaksi ennen suorittamista. Sen sijaan se on tulkattava ohjelmointikieli ja vaatii isäntäympäristön kuten selaimen, joka suorittaa JavaScript-koodin rivi kerrallaan.

Selain on JavaScriptin yleisin isäntäympäristö, mutta isäntäympäristönä voi toimia moni muukin, esimerkiksi

- Adobe Acrobat
- Adobe Photoshop
- SVG-kuvat
- Node.js-palvelinpuolelta.

JavaScriptin syntaksi perustuu Javaan ja C-kieleen. Se on aakkoskoon tunnistava (case-sensitive), mikä tarkoittaa sitä, että isot ja pienet kirjaimet ovat eri asia. [15.]

Selaimen toimiessa isäntäympäristönä JavaScript-koodia voidaan suorittaa HTML-tiedoston sisällä antamalla head- tai body-tagiin <script>-aloitustagi ja </script>-lopetustagi. Näiden <script></script>-tagien sisälle kirjoitetaan suorittaessa JavaScript-koodi. [17.]

JavaScript, ohjelmistokehykset ja kirjastot

Kun vuoden 2006 jälkeen kehitetty jQuery-JavaScript-kirjasto sai alkunsa, on sen jälkeen JavaScriptin pohjalta luotu useita erilaisia JavaScript-ohjelmistokehyksiä ja kirjastoja.

JavaScript-kirjastot

Kirjastot ovat valmiiksi kirjoitettua JavaScript-koodia, jonka ohjelmoija voi yhdistää omaan lähdekoodiinsa. Ne voivat sisältää objekteja, luokkia tai funktioita. Kirjastot voivat olla esimerkiksi pieniä apuohjelmia, kuten erilaiset kalenterit tai kaaviot, joita käytetään verkkosivulla. Ne voivat olla myös suurempia kokonaisuuksia, joilla sovelluksen JavaScript-lähdekoodi muuttuu radikaalisti verrattuna Vanilla JavaScriptiin. Yleisimpiä JavaScript-kirjastoja ovat jQuery ja React.js. [19.]

ReactJs sekoitetaan usein ohjelmistokehyksiin, mutta se on kirjasto. Suurin ero Reactissa verrattuna ohjelmistokehyksiin, kuten Angulariin, on se, että React huolehtii vain käyttöliittymän renderöinnistä. [20.]

Toiminnallisuudeltaan näissä kirjastoissa pätevät samat säännöt kuin Vanilla JavaScriptissä, vaikka ne saattavat näyttää syntaksiltaan täysin erilaiselta. Jotkut näistä kirjastoista, kuten jQuery, pitää ladata erikseen omana .js-tiedostonaan tai linkittää MDN:n kautta HTML-tiedoston head-tagiin. Head-tagi HTML-tiedostossa käsittelee metadataa, joka ei ole näkyvissä loppukäyttäjälle.

Toiset kirjastot taas, kuten ReactJs, voidaan alustaa npm:llä omana projektinaan, joka sisältää index.html-tiedoston, mutta siihen on linkitetty React-projektin pääluokka, jonka alle .js-tiedostot luodaan. React voidaan myös linkittää ilman npm:n käyttöä head-tagiin kuten jQuery. [12, s. 510.]

JavaScript-ohjelmistokehykset

Ohjelmistokehykset sen sijaan ovat JavaScriptin päälle rakennettuja apuvälineitä. Niiden ero kirjastoihin verrattuna on se, että kirjastoja käytettäessä ohjelmoija on vastuussa sovelluksen suorituksesta, kun taas ohjelmistokehykset toimivat Inversion of Control (IoC) -periaatteella. Tämä tarkoittaa sitä, että ohjelmistokehys ottaa vastuun ohjelman suorituksesta. Hyvänä puolena ohjelmistokehyksissä on se, että ohjelmoijan ei tarvitse kiinnittää huomiota kaikkiin pieniin yksityiskohtiin. Huonona puolena taas ohjelmistokehys sisältää paljon lähdekoodia, mikä vaikuttaa sovelluksen suorituskykyyn. Ohjelmistokehykset myös vaativat ohjelmoijaa ymmärtämään, miten tämä sovelluskehys toimii. [20.]

3.2 Yleisimmät modernit JavaScript-ohjelmistokehykset

JavaScript kielenä kehittyy koko ajan nopeasti, ja uusia ohjelmistokehyksiä ja kirjastoja tulee koko ajan lisää. Ohjelmointikielenä se on saavuttanut suurta suosiota myös sen takia, että sillä pystyy tekemään palvelinpuolen koodia Node.js:n kautta. [20.] Yleisimpiä ja suosituimpia moderneja JavaScript-ohjelmistokehyksiä ja kirjastoja ovat

- ReactJs (kirjasto)
- AngularJs (ohjelmistokehys)

- VueJs (ohjelmistokehys).

Jokaisella näistä on oma uniikki syntaksinsa, mutta pohjimmiltaan niiden toiminnallisuus periytyy JavaScriptistä.

ReactJs

ReactJs on Facebookin vuonna 2011 kehittämä avoimen lähdekoodin JavaScript-kirjasto, joka on saanut suurta suosiota, koska sen avulla voi kehittää web-sovelluksia nopeasti vähäisellä koodimäärällä.

ReactJs antaa kehittäjälle mahdollisuuden pilkkoa ulkoasun elementit pienempiin komponentteihin, joita on mahdollisesti helpompi ohjata. [21.]

VueJs

VueJs on viime vuosina kasvattanut paljon suosiotaan JavaScript-ohjelmistokehyksien joukossa. VueJs on vapaan lähdekoodin JavaScript-ohjelmistokehys, jonka kehitti Evan You, ja se julkaistiin vuonna 2014. Sillä on Reactin kanssa samantyyppinen lähestymistapa komponenttien käyttöön ja renderöintiin. [22.]

AngularJs

Angular on Googlen kehittämä JavaScript-ohjelmistokehys, joka on tehty TypeScriptin pohjalta. Sen vahvuuksia on TypeScript-tuki, mutta heikkoutena on raskaampi suorituskyky kuin Reactissa ja Vuessa. [22.]

Kaikkia näitä kolmea yhdistää niiden jatkuva kehitys ja aktiiviset yhteisöt. Jokaisella näistä on omat hyvät ja huonot puolensa, ja ne sopivat erilaisiin projekteihin. Esimerkiksi AngularJs ei ole hyvä vaihtoehto pieneen web-sovellukseen sen raskauden takia, kun taas VueJs sopii hyvin pieneen Single Page Appiin sen kevyen suorituskyvyn takia.

Bootsrap

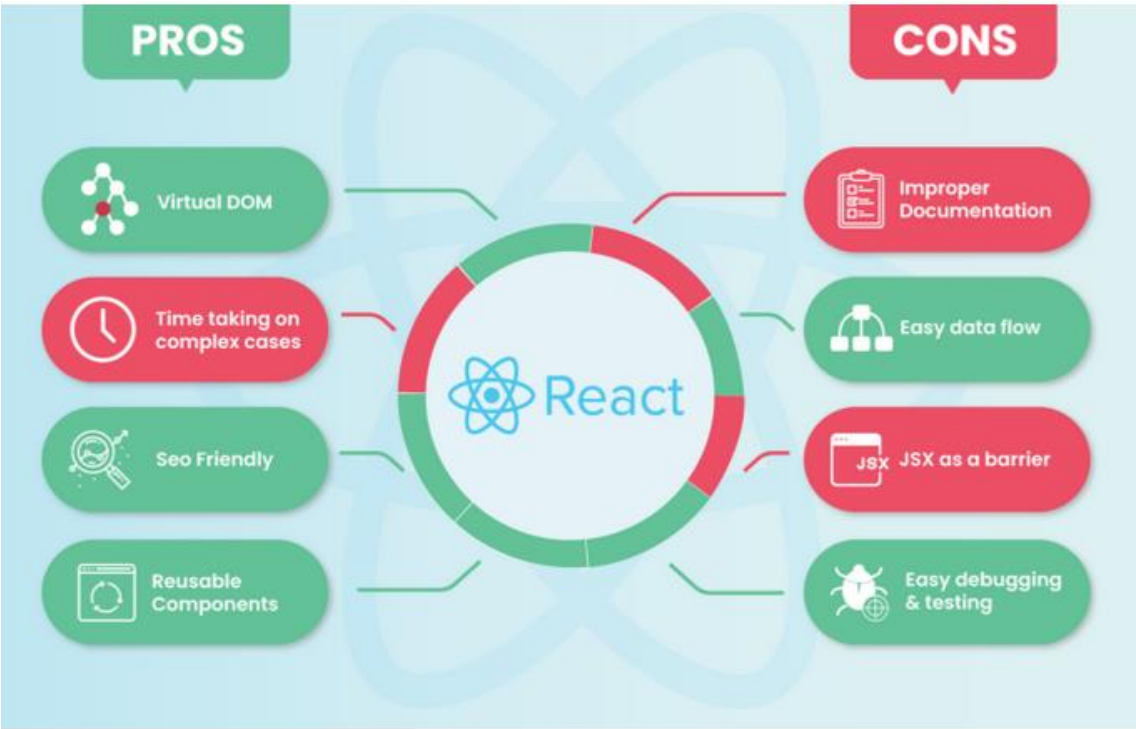
Bootstrap on kirjasto, joka sisältää valmiita CSS-muotoiluja HTML-elementeille. Se ei ole varsinaisesti JavaScriptiin liittyvä kirjasto, vaan se keskittyy enimmäkseen tyylitiedostojen hiomiseen. Tämän kirjaston hyvänä puolena on käyttöliittymä kehityksen nopeuttaminen, koska sillä pystyy laittamaan selainpuoleen valmiiksi tehtyjä paloja, jotka ovat kolmannen osapuolen työtä.

3.3 Vanilla JavaScript verrattuna valmiisiin kirjastoihin ja ohjelmistokehyksiin

Kun sovellusta suunnitellaan, tulee ajankohtaiseksi miettiä, mikä JavaScript-kirjasto tai ohjelmistokehys sopisi parhaiten sovellukseen vai toteutetaanko sovellus Vanilla JavaScriptillä ja minimaalisella kirjastojen käytöllä.

Projektin laajuus

Jos tiedetään, että toteutettava sovellus tulee olemaan erittäin laaja, kuten esimerkiksi Twitter tai Facebook, joissa käsitellään suurta määrää dataa, HTML-elementit vaihtuvat useasti ja sovellus vaatii paljon kommunikaatiota käyttäjältä, on ohjelmistokehyksen tai kirjaston kuten ReactJs:n valitseminen parempi vaihtoehto sovelluksen suorituskyvyn kannalta. Tämä sen takia, että esimerkiksi ReactJs:llä on virtuaalinen DOM-ominaisuus, mikä tarkoittaa sitä, että DOM-manipulointi tehdään virtuaalisesti sen sijaan, että se tehtäisiin suoraan selaimessa. Tämä parantaa suorituskykyä projekteissa, jotka sisältävät paljon datan käsittelyä ja käyttöliittymän elementtien vaihtelua. [24.] Kuvassa 7 esitellään ReactJs:n hyviä ja huonoja puolia.



Kuva 7. ReacJs:n hyviä ja huonoja puolia [21].

Vanilla JavaScript on hyvä valinta pienempiin projekteihin, joissa saattaa olla paljon sisältöä mutta vähemmän DOM-manipulaatiota. Tämä siksi, että Vanillassa DOM-manipulointi kirjoitetaan manuaalisesti. Jos projekti on erittäin laaja ja vaatii paljon DOM-manipulointia, tekee tämä lähdekoodin ylläpitämisen vaikeaksi Vanilla JavaScriptillä.

Projektin toteutuksen nopeus

Kun sovelluksen kehityksessä on otettu apuun JavaScript-kirjastoja, voidaan toteutusta nopeuttaa, kun käytetään valmiiksi tuotettua koodia. Esimerkiksi kun ohjelmoija tarvitsee kalenteria sovellukseen ja lataa valmiin JavaScript-kalenteri kirjaston, hän säästää aikaa sen sijaan, että hän alkaisi kehittää alusta omaa kalenterilisäosaa sovellukseensa. Tämä voi tosin kostautua myöhemmin, jos esimerkiksi kirjaston päivitys rikkoo sovelluksen toiminnallisuuden ja aikaa kuluu tämän korjaamiseen.

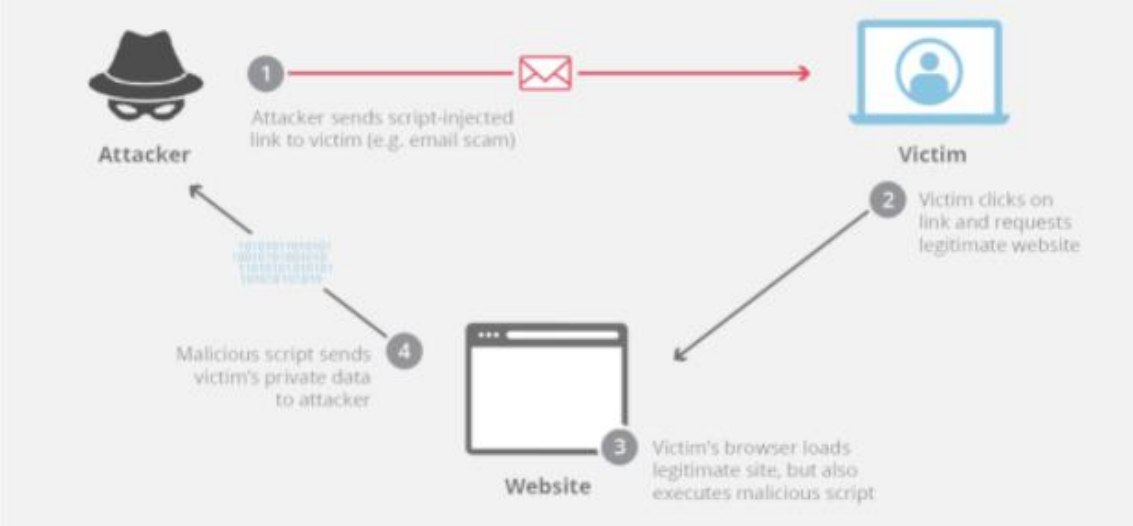
Oman koodin hallinta verrattuna kirjastoon

Kun käytetään valmista kirjastoa, kuten esimerkiksi kalenterilisäosaa, voidaan ajatella, että tämä on järkevä lähestymistapa, kun ”pyörää ei keksitä uudestaan” ja säästetään aikaa. Kirjastot ovat kuitenkin kolmannen osapuolen koodia ja rajoittuvat siihen, mitä niiden kehittäjät ovat ohjelmoineet niiden sisältämiin toiminallisuuksiin. Tämä tarkoittaa sitä, että ladatusta kirjastosta saattaa puuttua ominaisuuksia, joita projektin kehittäjä tarvitsisi. Jos kehittäjä ohjelmoi oman kirjastonsa kalenterille, hän hallitsee koko kirjaston lähdekoodin ja pystyy helposti lisäämään haluttuja ominaisuuksia tähän lisäosaan.

3.4 Tietoturva

Kolmannen osapuolen lähdekoodi ei ole ohjelmoijan omaa koodia. Vaikka kirjaston lähdekoodi on avointa lähdekoodia, ei moni kehittäjä tarkasta sitä. Tämä kolmannen osapuolen koodi saattaa olla huonosti kirjoitettua koodia ja sisältää mahdollisia tietoturva-aukkoja. Toinen mahdollinen skenaario on, että kirjaston kehittäjä on hylännyt projektin ja tällaisissa hylätyissä projekteissa on usein paljon haavoittuvuuksia, joita ei ole korjattu. Kirjaston kehittäjä voi myös vaihtua, mikä tarkoittaa sitä, että uusi kehittäjä voi tehdä lähdekoodiin suuria muutoksia ja jopa lisätä haitallista koodia. [25.]

Yksi yleisimmistä selainpuolen tietoturvahyökkäyksiä on Cross-site scripting (XSS) (kuva 8). Se perustuu siihen, että hyökkääjällä on tiedossa tunnettuja haavoittuvuuksia web-sovelluksista, ohjelmistokehyksistä ja JavaScript-kirjastoista. [26.]



Kuva 8. Cross-site scripting-hyökkäyksen toimintaperiaate [27].

Kirjastojen haavoittuvuus

Koska kirjastojen kehitys on kolmannen osapuolen vastuulla, ei ole takuita, että niistä ei löydy tietoturvaukkoja. Web-ohjelmista noin 37 % käyttää vähintään yhtä kirjastoa, josta löytyy haavoittuvuus. [28.] NPM:n virallisesta dokumentaatiosta löytyy tosin tapa tehdä turvatarkastus JavaScript-kirjastoille, ja sen avulla voidaan löytää potentiaalisia riskialttiita kirjastoja [29].

3.5 Sovelluksen toiminallisuus

Sovelluksen lähdekoodi on sovelluksen ydin. Kirjoitetun koodin laadulla ja ammatti- tai epäammattimaisuudella on erittäin suuri vaikutus moneen tekijään sovelluksessa. Se vaikuttaa

- sovelluksen yhteensopivuuteen erilaisten laitteiden kanssa
- tietoturvaan
- virheiden esiintymiseen
- ylläpitoon ja jatkokehitykseen
- suorituskykyyn.

Hyvälaatuinen koodi tekee sovelluksesta paljon ammattimaisemman ja helposti ylläpidettävän projektin. [30.]

Kirjastoja käytettäessä on otettava huomioon, että ne voivat olla huonosti kirjoitettua koodia, mikä saattaa heikentää sovelluksen toiminnallisuutta.

Kirjastojen päivitykset

Kirjastojen kehittäjät voivat päivittää lähdekoodiansa, mikä taas saattaa vaikuttaa projektiin, jossa tätä kirjastoa käytetään. Pahimmassa tapauksessa se voi rikkoa projektin toiminnallisuuden. Northeasterin yliopiston tutkimus löysi Mediaani-verkkosivulla käytettävän niinkin vanhoja kirjastoja, jotka olivat 1177 päivää jäljessä uusimmasta versiopäivityksestä [28].

Suorituskyky ja latausnopeus

Jokainen kolmannen osapuolen kirjasto kuormittaa sovelluksen suorituskykyä. Kun kirjaston JavaScript-koodi ladataan sivulle, tämä kuormittaa muistia ja hidastaa latausnopeutta, vaikka ohjelmoija käyttäisi vain pientä osaa laajasta lähdekoodista. Kun kirjastoja kertyy enemmän, hidastuu sovelluksen latausnopeus sitä kuin kirjastoja käytetään projektissa. [31.]

Ohjelmistokehykset ja kirjastot muuttuvat

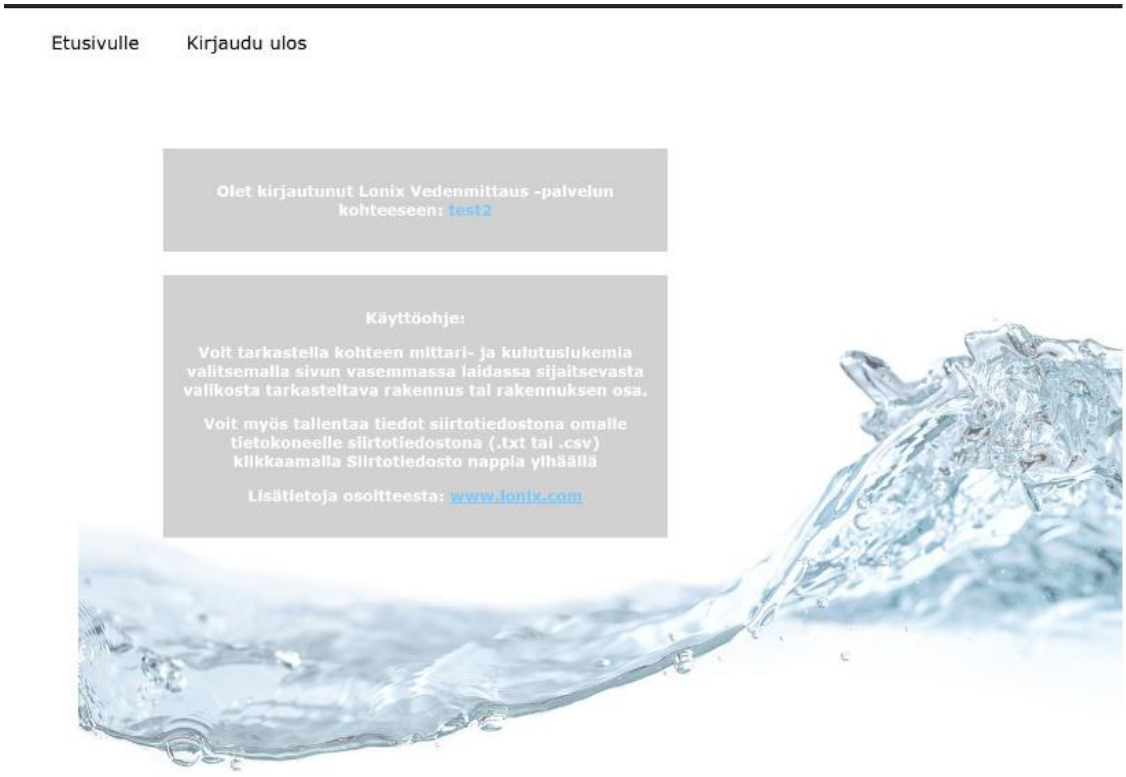
Vanilla JavaScript pysyy samanlaisena, kunnes sitä päivitetään uudempiin versioihin. Koska JavaScript on pääasiassa verkossa ainoa ”runtime”-kieli, on sen oltava yhteensopiva ja suoritettavissa selainten kanssa. Toisin on ohjelmistokehyksillä ja kirjastoilla, jotka muuttuvat eikä niillä ole takuita, että ne ovat yhteensopivia kuten Vanilla JavaScript. Niiden lisenssit voivat myös muuttua mikä voi johtaa muihin ongelmiin. [32.]

4 Insinöörityöprojekti ja Vanilla JavaScript -ohjelmointiprosessi

4.1 Projektin tavoite

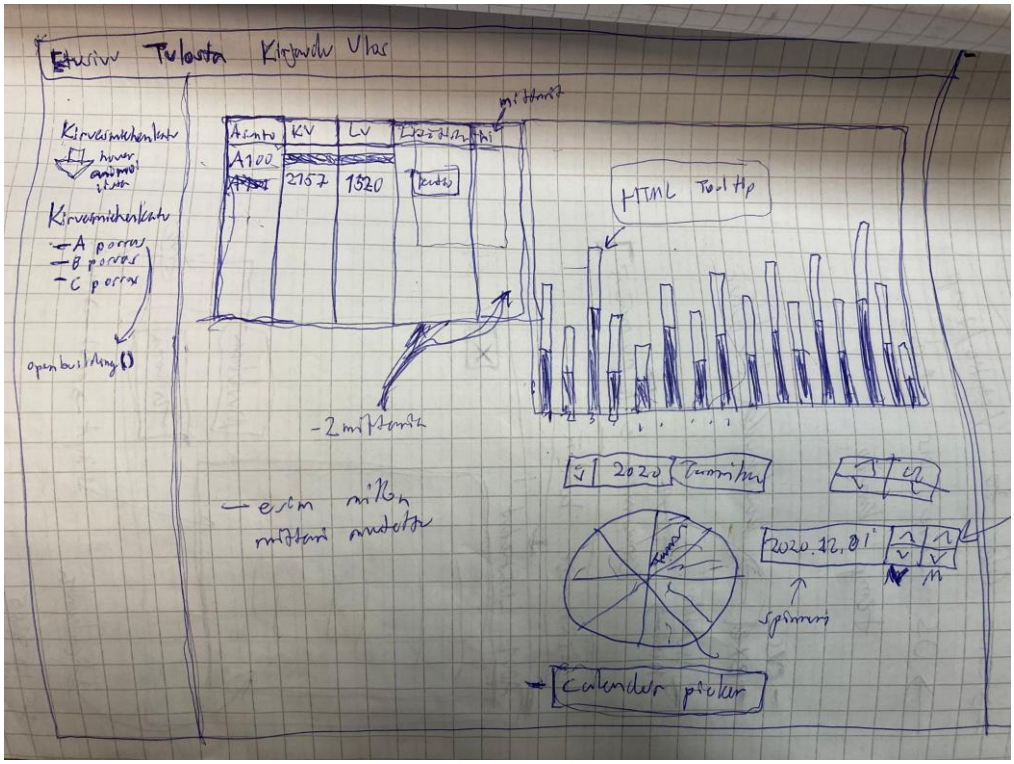
Insinöörityöprojektin tavoitteena oli kehittää asiakkaana olevien taloyhtiöiden isännöitsijöille sovellus, jonka kautta he pystyvät tarkastelemaan taloyhtiön asuntojen vedenkulutusta vuosi-, viikko- ja päivittäisellä näkymällä.

Vedenkulutuksen tarkastelun lisäksi tarvittiin ominaisuus, jolla loppukäyttäjän on mahdollista tallentaa tämä data erilaisiin formaatteihin, kuten Excel- tai tekstitiedosto. Datan kerääminen kulutuksesta ei kuulunut projektiin, vaan se saatiin haettua Coban omasta REST-rajapinnasta. Sovelluksen etusivusta pyrittiin saamaan mahdollisimman selkeä ja visuaalisesti hyvännäköinen (kuva 9).



Kuva 9. Insinöörityönä tehdyn sovelluksen valmiin prototyyppiversion etusivun näkymä.

Vaatimuksina oli, että kolmannen osapuolen koodia käytetään niin vähän kuin mahdollista. Tämä sen takia, että tarkoituksena oli auttaa sovelluksen ylläpitoa niin, ettei kirjastoja tarvitse päivittää säännöllisesti. Kolmannen osapuolen koodin minimoiminen myös antoi enemmän mahdollisuuksia ja valtaa kehittää sovellusta haluttuun suuntaan ilman kirjastojen rajoituksia. Projektin luonteen takia myös Vanilla JavaScript sopi parhaiten selainpuoleen, eikä ohjelmistokehyksille ollut tarvetta. Ulkoasua ennen varsinaista selainpuolen luomista suunniteltiin paperille (kuva 10).



Kuva 10. Paperiluonnos suunnitteluvaiheessa tulevasta sovelluksen ulkoasusta.

Tavoitteena oli myös saada järjestelmävalvojan näkökulmasta helposti ylläpidettävä sivu, joka olisi mahdollisimman pitkälle automatisoitu vähentämällä ohjelmoijien työtä. Automatisoinnin periaate oli tarkoitus toteuttaa tekemällä järjestelmävalvojalle oma HTML-instanssisivu, josta voi lisätä, poistaa tai muokata käyttäjien tietoja. Nämä tiedot kommunikoivat tietokannan kanssa.

Tähän tarkoitukseen valittiin PM2-sovelluksen prosessinhallintatyökalu.

4.2 Projektin alustus

Kun sovelluksen suunnitteluvaihe oli saatu valmiiksi, aloitettiin projektin teknin-
nen osuus.

Järjestelmävalvoja ja sovelluksen hallinta

PM2-työkalu antaa järjestelmävalvojalle listan kaikista sovelluksen käyttäjistä ja niiden senhetkisestä statuksesta. Tarkoituksena on, että jokainen käyttäjä saa oman portin, joka antaa pääsyn sovellukseen.

Status-listassa näkyy käyttäjän nimi, portin numero sekä nykyinen tila, joka voi olla STOPPED/ONLINE/ERRORED.

Mikäli tälle käyttäjälle pitää tehdä jotain muutoksia sovellukseen, kuten lisätä tai poistaa tietoja, onnistuu tämä PM2-listasta laittamalla tilaksi ”Stopped”, jolloin käyttäjä saa ”Huoltokatko”-ilmoituksen, jos hän yrittää kirjautua sisään. PM2 suoritetaan palvelimen terminaalista. Tiedot käyttäjistä se saa ecosystem.js-nimisestä tiedostosta, jossa lukee portin numero ja käyttäjänimi (kuva 10).

Kuva 10. Esimerkki yhden käyttäjän PM2-ecosystem-tiedoston konfiguroin-
tinäkymästä.

Tietokanta ja REST-rajapinta

Sovelluksen keskeisin asia loppukäyttäjän näkökulmasta on analysoida dataa vedenkulutuksesta. Koska data on jo valmiiksi tallennettu erilliseen palveluun Coban REST -rajapinnassa, tietokantaa ei tässä sovelluksessa tarvita vedenkulutuksen tallentamiseen. Tämä datamäärä on suuri ja vaatisi sovellukselta paljon tietokannan kanssa kommunikointia. Rajapinta itsessään on laaja, eikä se liity suoranaisesti projektiin, joten sen läpi käyminen ei ole järkevää. Lyhyesti selitettynä rajapinnasta saa haettua tuntikohtaista dataa vedenkulutuksesta jokaisesta asunnosta, joka on liitetty siihen. Kulutus lasketaan vesimittareista, ja se lähetetään toiselle palvelimelle, jossa rajapinta sijaitsee.

Koska rajapinnassa on valmiiksi hoidettu tietokantakyselyt, ei sovellukseen tarvittu montaa tietokantaan kommunikoivaa funktiota. Funktiot, jotka vaativat yhteyttä tietokantaan, koostuivat

- loppukäyttäjään liittyvistä JSON-tiedostoista ja niiden linkittämisestä
- autentikoinnista
- järjestelmävalvojan toiminnoista, joilla lisätä käyttäjiä tai muokata niiden tietoja
- kohteiden muistiinpanotiedoista.

Vähäisen tietokantakommunikoinnin vuoksi palvelinpuolen lähdekoodin kehittäminen oli helpompaa ilman monimutkaisia relaatioita, joita tällaisen datamäärän kanssa olisi ollut. Rajapinta myös lisää tietoturvaa, sillä tärkeään dataan ei voi päästä kiinni sovelluksen kautta.

Sovelluksen oma tietokanta ei myöskään sisällä mitään arkaluontoista tietoa käyttäjistä, kuten sähköpostiosoitetta tai puhelinnumeroa. Se tarvitsee käyttäjänimen ja salasanan lisäksi vain sovelluksen funktionaalisuuteen tarvittavia tietoja, kuten listan JSON-tiedostoista, joilla käyttöliittymän kulutusnäkymä luodaan.

Huonona puolena tosin rajapinnan kanssa kommunikoinnissa on se, että sovellus on riippuvainen rajapinnasta. Jos rajapintaa suorittava sovellus sammuu tai

itse palvelin menee kiinni, se tekee tästä sovelluksesta turhan, sillä käyttäjä ei pysty näkemään mitään dataa.

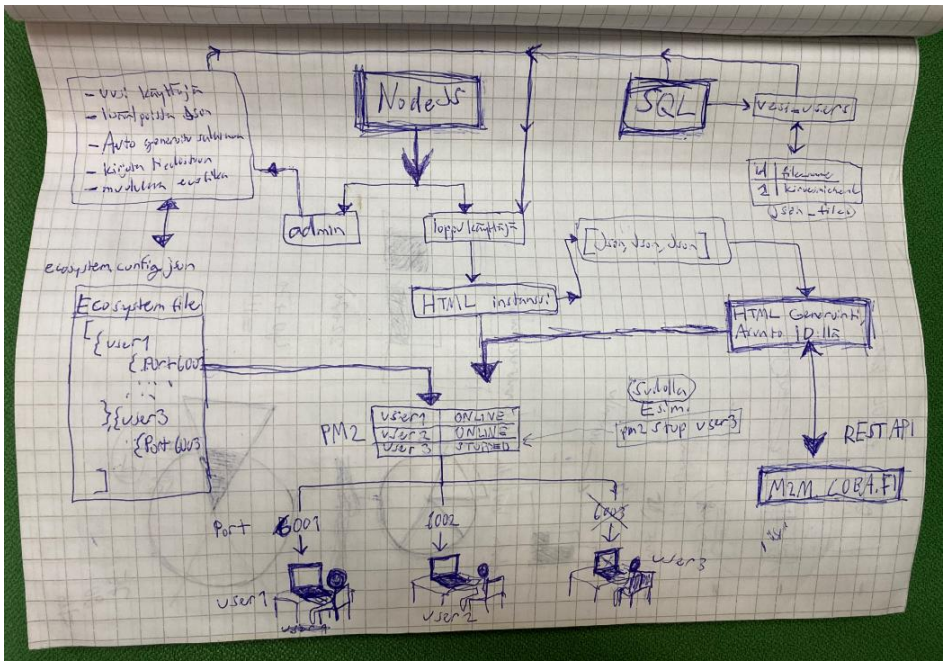
4.3 Toteutus

Sovelluksen loppukäyttäjän puoli toteutettiin Vanilla JavaScriptillä ja palvelinpuoli taas Node.js:llä. Loppukäyttäjän puoli itsessään ei ollut julkista, vaan vasta käyttäjän kirjauduttua sisään palvelinpuolella generoitiin HTML-sivun instanssi, joka sisälsi kaikki HTML-elementit, tyylitiedostot ja JavaScript-koodin, joita käytetään selainpuolella. Tämä lähestymistapa ei ole kovin yleinen projektin luonteen ja tavoitteiden vuoksi, jotka olivat Vanilla JavaScriptin käyttö ja kolmannen osapuolen koodin vähentäminen.

Asiakaspäässä pyrittiin minimoimaan kirjastojen käyttöä sivun suorituskyvyn nopeuttamiseksi ja antamalla ohjelmoijalle mahdollisimman paljon valtaa vaikuttaa omaan lähdekoodiin. Aluksi pylväsdiagrammia kokeiltiin kirjastolla, mutta tämä kirjasto ei sisältänyt tärkeitä ominaisuuksia, joita sovellukseen tarvittiin, joten päädyttiin ratkaisuun tehdä oma. Esimerkiksi datan tarkasteluun käytettävä pylväsdiagrammi on sovelluksen ohjelmoijan itse kehittämä. Kehittäjä hyötyy tästä diagrammista siten että sen avulla voi lisätä tarvittavia ominaisuuksia, joita ei löydy samantyylisestä kolmannen osa puolen kirjastosta. Tämän diagrammin ylläpitäminen on myös helpompaa, koska ohjelmoijalla on täysi kontrolli omaan lähdekoodiin, mikä tekee mahdollisten muutosten tekemisen ja jatkokehityksen joustavaksi. Kehittäjän ei tarvitse myöskään huolehtia kirjastojen päivittämisestä.

Automatisaatio

Sovelluksen automatisaatio suunniteltiin niin, että useampi kohta, jonka ihminen joutuisi tekemään käsin, on saatu ohjelman algoritmilla suorittamaan tietokone. Tätä automaation vaihetta suunniteltiin myös käsin paperille piirtämällä eri vaiheita prosessista (kuva 11).

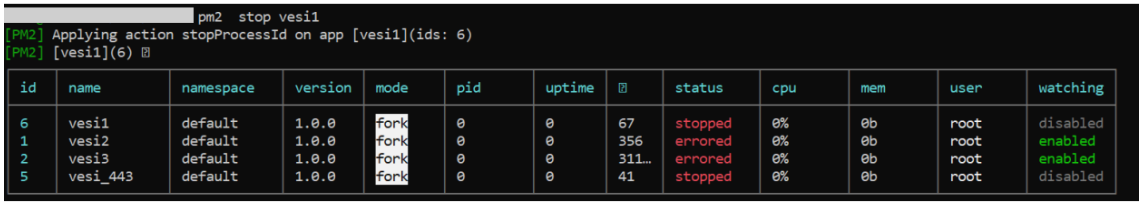


Kuva 11. Projektin automaation suunnittelua.

Pitkällä aikavälillä tämä automatisointi säästää runsaasti aikaa, kun tarkastellaan tilannetta, jossa käyttäjiä alkaa olla kymmeniä ja enemmän. Jos jokaiselle käyttäjälle pitää tehdä tietyt sovelluksen vaatimat askeleet käsin, vie tämä paljon aikaa järjestelmävalvojalta.

Projektin alustus ja PM2

Koko projekti perustui mahdollisimman hyvään automatisaatioon. Tekninen idea oli se, että käyttäjien data luodaan erillisessä sovelluksessa LXX-muotoon, joka tallennetaan palvelimelle. Koska LXX-tiedoston dataa ei voi suoraan saada sovelluksen käyttöön, on palvelinpuolella automatisoitu tämä siten, että sovellus muuntaa datan JSON-muotoon, jota sovellus osaa lukea, ja tallentaa siitä JSON-kopion palvelimelle. Samalla tästä tehdään tietokantaan uusi sarake tauluun, jonka sarakkeet sisältävät tiedoston nimen ja ID:n. Itse JSON-tiedoston sisältämän olion dataa ei tallenneta tietokantaan, vaan sovellus osaa lukea sen tiedostosta ID:n perusteella. PM2:n hallintapaneelista (kuva 12) pystyi luotujen sivujen käyttöä ohjaamaan.

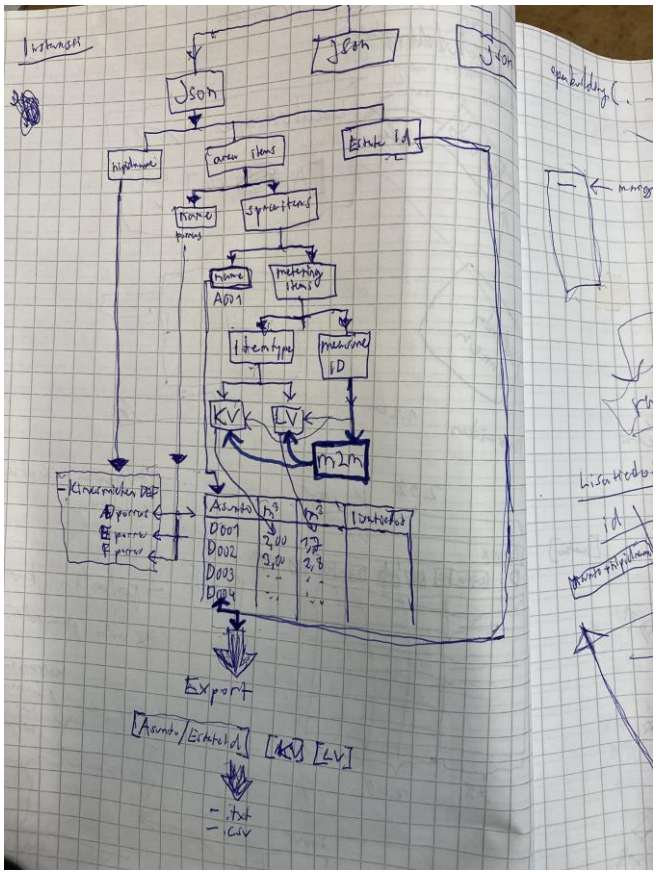


Kuva 12. PM2:n hallintapaneeli.

Tämä vaihe automatisoi sen, ettei järjestelmävalvojan tarvitse käsin muokkailla LXX:n dataa JSON-muotoon eikä erikseen tallentaa siitä uutta tiedostoa. Yksittäisissä tapauksissa tämä prosessi ei vie paljoa aikaa, mutta kun käyttäjiä tulee paljon, nopeuttaa tämän vaiheen automatisointi huomattavasti sovelluksen hallintaa.

Elementtien luominen instanssiin

Koska projektin loppukäyttäjien tarvitsema data saatiin haettua JSON-oliosta, oli tietoturvasyistä paras lähestymistapa käsitellä tätä dataa palvelinpuolella sen sijaan, että se olisi julkisesti selainpuolella. Tässä lähestymistavassa JSON-olion tiedoista luodaan palvelimella laaja merkkijono, joka vastaa suurta HTML-tiedostoa, ja sisältää kaikki HTML-elementit, jotka kyseiselle käyttäjälle tulisi näyttää käyttöliittymässä. Tälle prosessille on myös nimi Server-Side Rendering (SSR), mikä tarkoittaa, että palvelin pystyy renderöimään HTML-tiedostot palvelimelta selaimeen. Projektin JSON-objektit vaihtelevat käyttäjän mukaan, ja ne voivat olla todella pitkiä ja monikerroksisia. Nämä oliot saadaan tietokantakutsulla, jossa vertaillaan käyttäjän ID:tä taulukkoon, jossa on jokaiselle JSON-tiedostolle oma uniikki ID. Näin usealla käyttäjällä voi olla käytössä samat tiedostot, tai niitä voi olla useita yhtä käyttäjää kohden. Tämä monimutkainen automaatiologiikka suunniteltiin paperille piirtämällä (kuva 13), koska se sisälsi niin monta eri vaihetta ja JSON-objektia.



Kuva 13. Paperille suunniteltu automaatiologiikka sovelluksen taulukon renderöimiseen.

Sovellukseen tarvittavan HTML-merkkijonon luominen sisältää monta erilaista vaihetta, joissa käytetään useaa for-silmukkaa JSON-olion syvyyden vuoksi. Tämä oli yksi syy, miksi Vanilla JavaScript valittiin työkaluksi. SSR voi käyttää myös muilla työkaluilla, kuten ReactJs:llä, mutta tässä olisi tullut haasteeksi DOM-manipulaatio. Sovelluksen luonteen takia ei käyttöliittymässä ollut tarkoitus olla koko ajan muuttuvia HTML-elementtejä. Myös pyrkimys vähentää mahdollisimman paljon kirjastoja oli tärkeä tekijä juuri projektin automaatiomaisen lähestymistavan vuoksi.

Kun HTML-sivun sisältämä merkkijono oli generoitu palvelimella, se lähetettiin se selaimeen, jossa siitä renderöitiin käyttöliittymä loppukäyttäjälle, kun tämä oli kirjautunut sisään.

Merkkijonon generoimisesta teki haastavaa se, että projekti pyrittiin automatisoimaan mahdollisimman paljon niin, ettei sovelluksen ohjelmoijille tule ylimääräisiä työvaiheita tai hallintaa. Tällaisia työvaiheita olisi voinut olla esimerkiksi PM2:n ecosystem.js-tiedoston muokkaaminen aina, kun uusi käyttäjä luodaan.

Tietoturvainstanssista

Tietoturvan kannalta SSR-lähestymistapa oli parempi, sillä ainoa julkinen sivu oli kirjautumissivu. Tämä estää crawlereita, verkossa suoritettavia robottimaisia ohjelmia, jotka pyrkivät keräämään esimerkiksi tietoa verkkosivusta, indeksoimasta muita sovelluksen sivuja, koska ne eivät ole olemassa vielä tässä vaiheessa. Yleisimpiä hyökkäyksen kohteita, joihin crawlerit pyrkivät pääsemään sisään, ovat juuri Wordpress tai muut yleisimmät sisällönhallintaohjelmat.

WordPress on yleinen käytetty sovellus, joka tarjoaa käyttäjälle mahdollisuuden luoda sisältöä verkkosivulle ilman ohjelmointia. Sitä käytetään enimmäkseen blogeissa, mutta nykyään myös monissa muissa verkkosivuissa, kuten verkkokaupoissa.

Crawelrien tai tietoa keräävien robottien hyökkäys yrityksiä pystyi monitoroimaan tuotantoon laitetusta testiversiosta ja niistä näki, että hyökkäykset on suunnattu yleisimpiin sisällönhallintaohjelmiin.

Indeksoiminen voi tosin haitata hakukoneoptimointia, mutta koska sovellus on tehty tietylle kohderyhmälle, ei tällä ole väliä. SEO (Search Engine Optimization, eli hakukoneoptimointi) tarkoittaa menetelmää, jolla pyritään tekemään verkkosivuun muutoksia niin, että se olisi mahdollisimman hyvin hakukoneiden, kuten Googlen löydettävissä.

Koska kaikki selainpuolen lähdekoodi on palvelimella turvassa, ei sitä pystytä kopioimaan verkkosivun kopio-ohjelmilla, kuten HTTRACK [33]. Vaikka selainpuoli koodissa ei saisi olla mitään kriittisiä haavoittuvuuksia, on silti aina tietoturvan kannalta parempi, mitä vähemmän lähdekoodia on näkyvillä. Selainpuolen

koodissa voi esimerkiksi olla avoimia päätepisteitä, joista hyökkääjä voi löytää mahdollisesti tietoturvahaavoittuvuuksia.

Koska selainpuolen koodissa ei käytetä kolmannen osapuolen kirjastoja, tämä parantaa myös tietoturvaa, kun kehittäjä tietää itse tarkalleen, mitä lähdekoodi pitää sisällään.

4.4 Omat elementit

Kun asiakaspäätä alettiin toteuttaa Vanilla JavaScriptillä, pyrittiin käyttämään mahdollisimman vähän kirjastoja, ei pelkästään JavaScript-kirjastoja, vaan myös CSS-kirjastoja, kuten Bootstrap. Mutta koska sovellus ei sisältänyt montaa erilaista nappia tai lomaketta, vaan elementtejä, joiden arvo sisältää ID:n rajapinnan kommunikoimisen takia ja tämän lisäksi sovelluksen ulkoasu sisältää elementtejä, jotka pitää itse tuottaa, ei Bootsrapin lisääminen olisi ollut järkevää.

Tähän vielä lisäksi se, että Bootstrap-kirjastoa pitäisi päivittää tasaisin väliajoin tietoturvasyistä ja SSR:n kanssa olisi voinut tulla vaikeuksia, koska kirjastojen minimoiminen oli yksi tärkeimmistä seikoista, käyttöliittymään luotiin omat elementit tyylitiedostoineen.

Vedenkulutuksen tarkasteluun otettiin seuraavat tavat:

- Excelin tapainen taulukko, jossa on rivit ja sarakkeet kokonaisen taloyhtiön jokaisen asunnon vuosi- tai kuukausikulutuksen kulutuksen tarkasteluun
- pylväsdiagrammi, jossa data näkyy asuntokohtaisesti palkkeina ja sisältää sekä kylmän että lämpimän veden kulutuksen tiettynä ajankohtana.

Molemmat kehitettiin insinöörityössä, ja diagrammista tehtiin oma JavaScript-kirjasto, johon kehittäjä voi antaa parametrejä, joiden perusteella selainpuoleen renderöityy automaattisesti diagrammi haluttuun kohtaan.

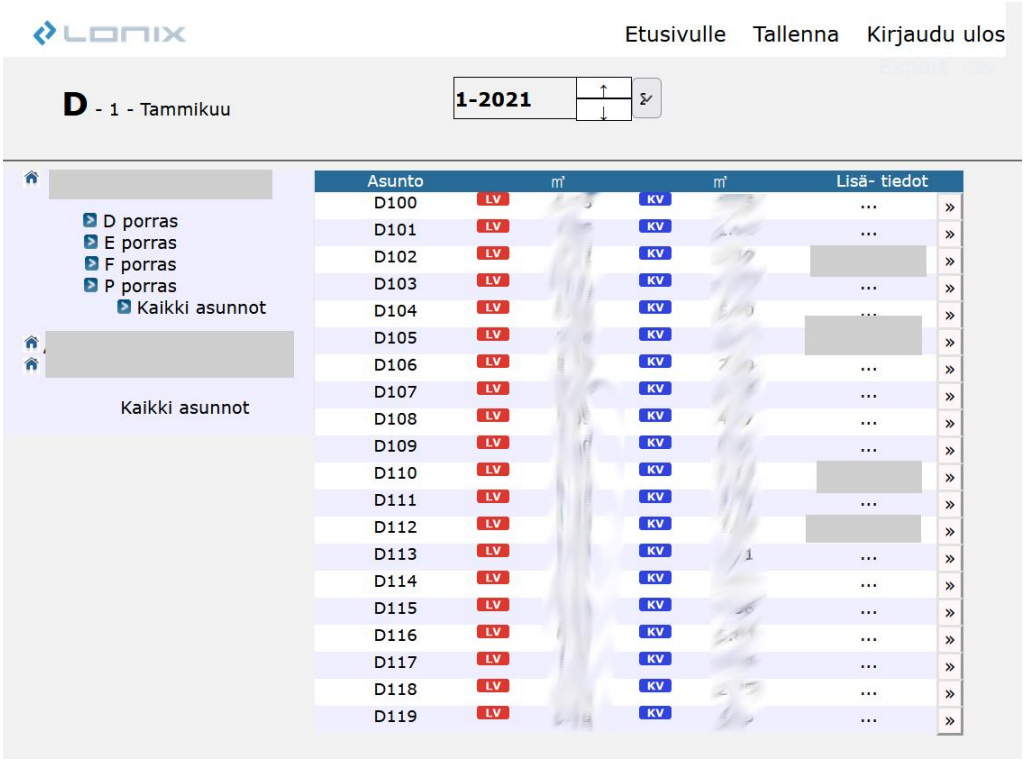
Taulukko datan tutkimiseen

Excel-tyyppinen kokonaiskuvan antava taulukko on ensimmäinen elementti, joka käyttäjälle avautuu, kun hän avaa taloyhtiön sivupalkista. Taulukossa on neljä saraketta, joissa näkyy

- asunnon numero
- kylmän veden kulutus
- lämpimän veden kulutus
- lisätietoja.

Taulukon vedenkulutustieto on senhetkisen kuukauden kokonaiskulutus. Navigaatiopalkissa sijaitsevat pudotusvalikot, joilla voi vaihtaa kuukautta tai hakua vuosikohtaiseksi.

Taulukon generoimiseen tarvitaan ID:t, joilla haetaan data rajapinnan päätepisteistä. Nämä ID:t otetaan JSON-olion kolmannesta haarasta, joka sisältää useita pisteitä, jotka vastaavat erilaisten ominaisuuksien tietoa, jota ID:llä haetaan. Tällainen tieto voi olla esimerkiksi lämpimän tai kylmän veden kulutus, mutta myös jokin muu tieto. Kuvassa 14 näkyy JSON-oliosta luotu taulukko vedenkulutuksesta.



Kuva 14. JSON-oliosta luotu oma HTML-elementtitaulukko annetuilla parametreillä.

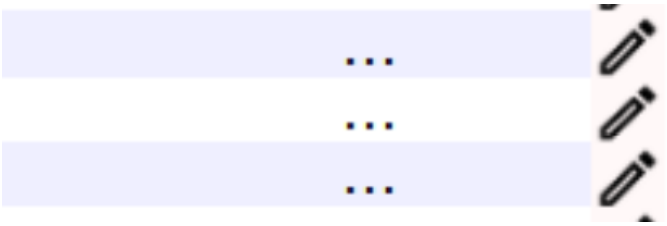
Taulukko generoidaan for-silmukalla, jossa ensin katsotaan kohteen (rapun) asuntojen kokonaismäärä. Vaihtoehtoisesti voi valita myös kaikki taloyhtiön asunnot. Tällä summalla määritellään silmukan ensimmäinen kierros, jossa määritellään taulukkoon tulevien rivien määrä.

Jokaisen rivin kohdalla kutsutaan uusi for-silmukka, jossa iteroidaan JSON-tiedoston läpi etsien ID:tä, joka täsmää asunnon ID:hen. Kun löytyy yhteensopiva ID-pari, kutsutaan funktio, johon annetaan parametreiksi ID:t. Samassa silmukassa luodaan -elementti, joka toimii rivinä, ja sille luodaan child-elementit sarakkeiksi. Ensimmäiseen sarakkeeseen tulee suoraan asunnon nimi. Taulukon pystyi generoimaan uusiksi sovellukseen tehdyllä hakemisominaisuudella, jossa pystyi antamaan tarkemmat parametrit, haetaanko dataa vuodelta, päivältä vai kuukaudelta (kuva 15).

Kuva 15. Hakuominaisuus eri tarkastelutyypeille.

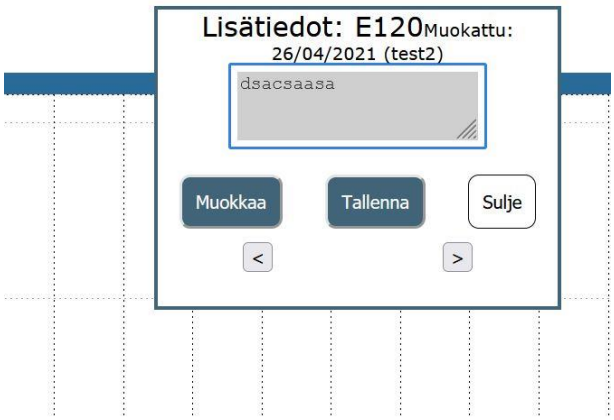
Toiseen ja kolmanteen sarakkeeseen tarvitaan funktio, jossa iteroidaan JSON-tiedostosta kylmän ja lämpimän veden ID:t. Ne lisätään sarakkeen innerHTML:ksi. Generoinnin yhteydessä lähetetään pyyntö rajapintaan kyseisillä ID:illä, mistä saadaan data visuaaliseen muotoon.

Neljäs sarake kutsuu suoraan generoinnin yhteydessä tietokannasta lisätietoa, mikäli sellainen on asetettu kohteelle. Muussa tapauksessa se on oletusarvoisesti vain kynäkuvake (kuva 16).



Kuva 16. Lisätietosarakkeen kynäkuvake.

Tämä kynäkuvake toimi nappina, jota painamalla avautuu lisätietolomakkeen modaali (kuva 17). Tähän modaaliin voidaan kirjoittaa esimerkiksi mittareiden tietoa.



Kuva 17. Lisätietolomakkeen modaali.

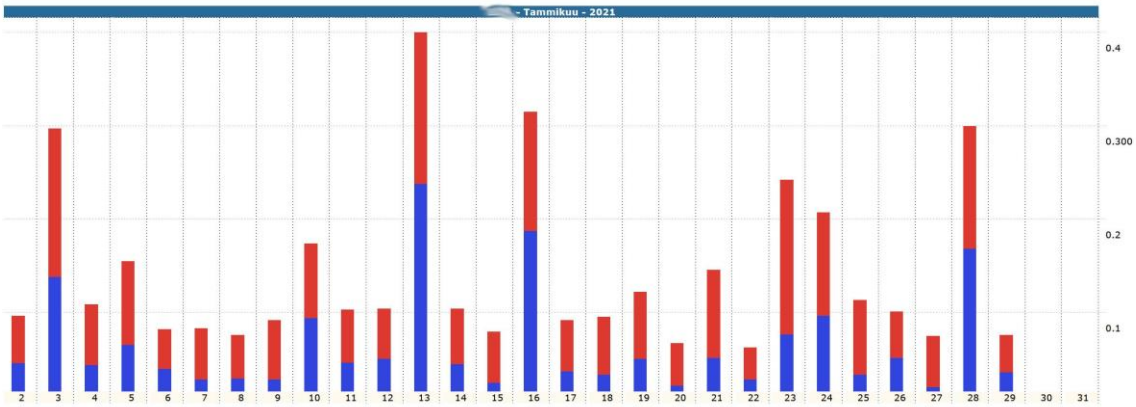
Taulukon generoiminen vie aikaa jonkin aikaa, sillä se joutuu kommunikoimaan rajapinnan ja tietokannan kanssa jokaisella rivillä. Latausaika on kohteen koosta riippuvainen, ja sovellukseen on tehty funktio, joka näyttää latausajan prosentteina. Tämä toimii siten, että aina kun taulukko generoidaan, luodaan kokonaisluku arvolla nolla. Joka kerta kun silmukassa rajapinta palauttaa dataa, lisätään tähän kokonaislukuun yksi numero lisää. Sovellukseen on vielä latausta varten tehty oma funktio, joka osaa suhteuttaa kohteiden summan ja sen perusteella tekee laskemisen, mikä palauttaa latausajan prosentteina.

Diagrammit

Kun käyttäjä haluaa tarkastella tietyn asunnon dataa, sovellukseen on kehitetty järjestelmä näyttämään tämä tieto pylväsdiagrammina. Pylväsdiagrammeissa on kolme eri vaihtoehtoa

- päivittäinen tuntikohtainen data, jossa käyttäjä voi nuolilla liikkua tunneissa eteen- tai taaksepäin; näitä pylväitä on 24
- kuukausinäkymä, jossa on 31/30/28 palkkia kuukauden mukaan; yksi palkki sisältää yhden päivän kokonaiskulutuksen
- vuosikulutus, joka generoi näkymän 12 palkista, joissa näkyy kuukauden kulutuksen summa.

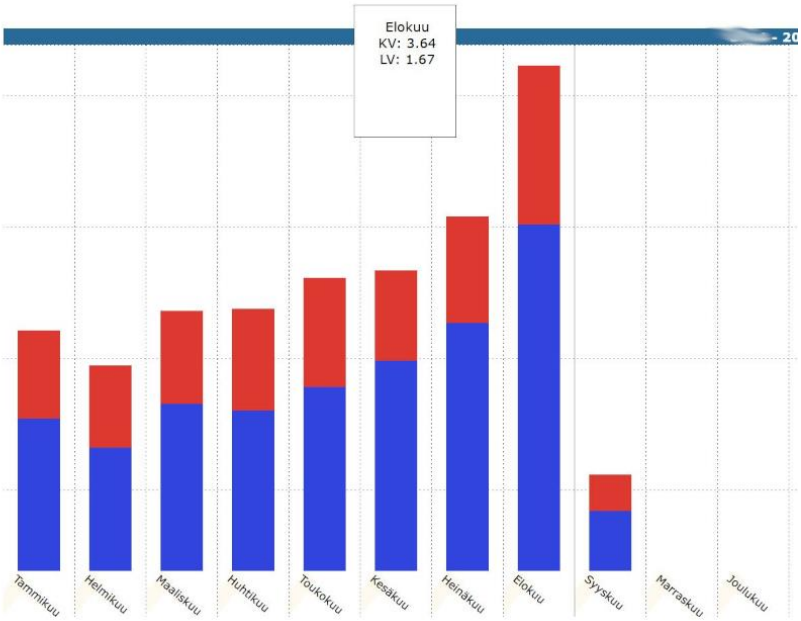
Näihin diagrammeihin on myös lisätty kuulijat, eli käyttäjä voi klikkaamalla diagrammin palkkia hiirellä siirtyä vuosinäkymästä tiettyyn kuukauteen ja siitä edelleen yhteen päivään.



Kuva 18. Kuukausinäkymändiagrammi vedenkulutuksesta.

Graafisen datan näyttäminen kaavioilla, kuten esimerkiksi diagrammeilla, on visuaalisesti hienompi ja käyttäjäystävällisempi tapa antaa asiakkaan tarkastella haluttua dataa. JavaScript-pohjaisia kirjastoja, jotka näyttävät haettua dataa diagramminäkymässä, löytyy verkosta todella paljon. Osa niistä on ilmaisia ja toiset taas avointa lähdekoodia.

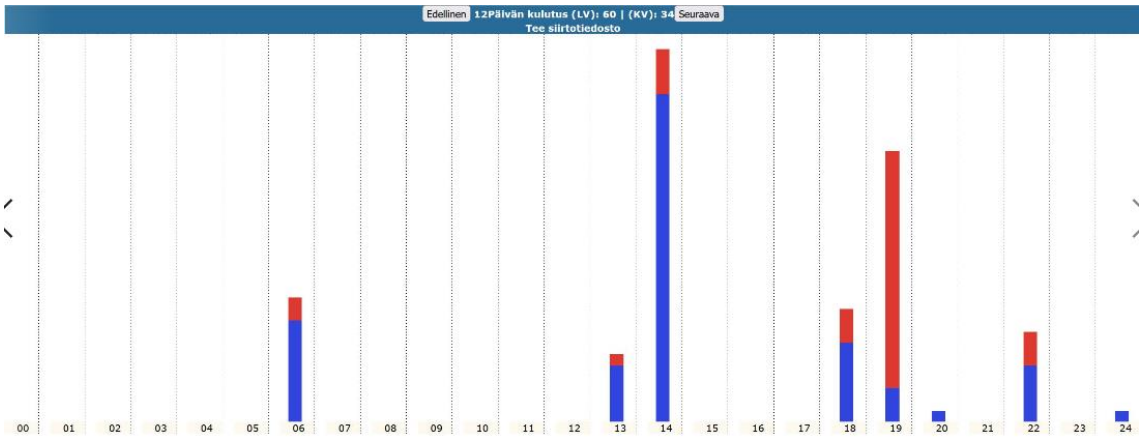
Tässä insinöörityössä toteutetun kaltaisiin sovelluksiin soveltuvat hyvin tällaiset kaaviokirjastot, jotka pystyvät näyttämään dataa visuaalisessa muodossa asiakkaalle sen sijaan, että data olisi pelkkänä tekstinä Excel-tyyppisessä taulukkomuodossa. Tämä diagrammiominaisuus ei ole pelkästään visuaalisesti hienompi ja ammattimaisemman näköinen lähestymistapa, vaan se myös säästää asiakkaan aikaa, kun hän pystyy näkemään kokonaiskuvan tietyltä aikajaksolta suoraan palkkeina ja huomamaan, milloin kulutusta oli vähän tai paljon (kuva 19). Tämän pohjalta hän pystyy taas jatkamaan nopeammin datan analysointia. Palkkeihin lisättiin ominaisuus, että asettamalla hiiren sen päälle avautuu laatikko, josta näkee kuukauden kuuman ja kylmän veden kulutuksen numeroina.



Kuva 19. Veden vuosikulutusnäkymä diagrammina ja laatikko, jossa data numeroina.

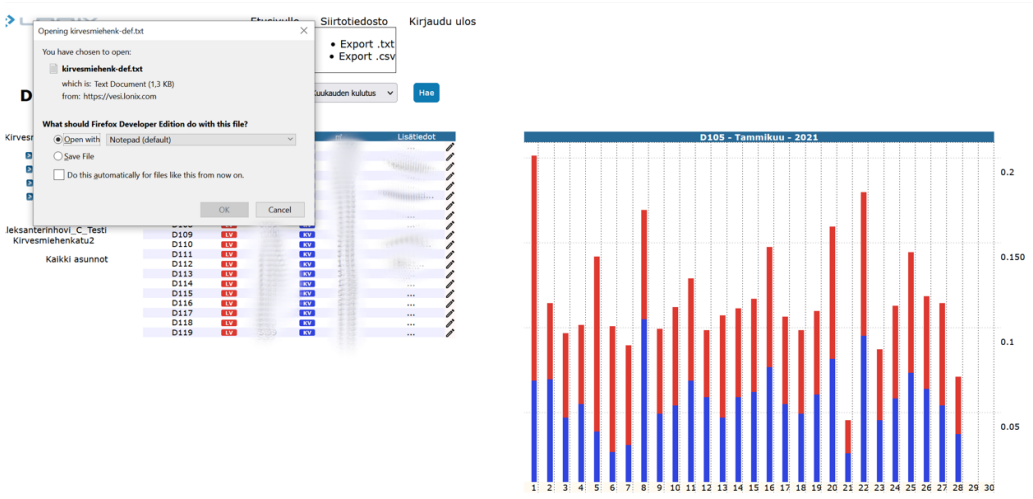
Alkuun pohdittiin, käytetäänkö sovelluksessa valmista JavaScript-kirjastoa diagrammiin, ja sitä kokeiltiinkin. Kirjaston implementoiminen SSR-lähdekoodiin tuli kuitenkin haastavaksi. Sen mukauttaminen käyttöliittymään sekä ulkoasuparametrien muuttaminen ei ollut helppoa johtuen siitä, koska kirjastossa oli rajoitettu määrä muuttujia, joihin kehittäjä pystyi vaikuttamaan ilman, että olisi alkanut muuttaa kirjaston omaa lähdekoodia.

Vuorokausinäkymään lisättiin vielä oma ominaisuus, jossa voidaan liikkua ajassa tuntikohtaisesti eteen- tai taaksepäin ylhäällä olevilla ”Edellinen”- ja ”Seuraava”-napeilla (kuva 20).



Kuva 20. Vuorokauden vedenkulutuksen näkymä.

Lisäksi palkkeihin laitettiin ominaisuus, että klikkaamalla sitä voidaan luoda kyseisen näkymän datasta Excel-tiedosto (kuva 21).



Kuva 21. Datan tallentaminen Excel-muotoon.

Kirjaston lähdekoodin muuttaminen ei ole hyvä lähestymistapa, ja se voi jopa olla laitonta riippuen kirjaston lisenssistä. Jotkin kirjastot voivat olla omisteisia ohjelmistoja, mikä tarkoittaa, että niillä on rajoitettu lisenssi. Näitä lisenssityyppejä voi olla useita, ja ne määrittävät, mitkä oikeudet kehittäjä saa ohjelmalle. [34.]

Vaikka kirjaston lisenssi olisi MIT-lisenssi, eli se sallisi ohjelmiston muokkaamisen jopa kaupalliseen tarkoitukseen [35], voi kirjaston lähdekoodin muuttaminen aiheuttaa ongelmia toiminnallisuuteen ja rikkoa kirjaston ominaisuuksia.

Koska sopivaa kirjastoa tähän sovellukseen ei löytynyt, päästiin toteuttamaan oma versio, jota olisi helppo ylläpitää, muokata ja antaa ohjelmoijalle työkalut muuntaa diagrammia juuri sopivaksi sovellusta varten. Funktionaalisuudeltaan tähän diagrammiin vaadittiin

- kyky näyttää palkkeja päällekkäin; esimerkiksi kylmällä vedellä oma palkki, joka on lämpimän veden palkin päällä vertikaalisesti
- responsiivisuus eli mukautuminen leveydeltään ja korkeudeltaan näytön koon ja resoluution mukaan
- kyky ottaa luku vastaan parametrinä, jolloin pystytään luomaan sama määrä palkkeja kuin annettu parametri on
- diagrammin yläpuolelle laatikko*, jossa näkyy aika ja kohteen nimi; tuntikohtaisessa näkymässä tähän laatikkoon renderöidään myös nuolet, joilla voidaan liikkua eteen- tai taaksepäin tunneissa
- ponnahdusikkuna, jossa dataa palkin ajankohdalta, kun hiiri liikutetaan sen päälle
- mahdollisuus klikata hiirellä palkkia, jolloin voidaan siirtyä pienempään aikayksikköön; esimerkiksi vuosinäkymän tammikuun palkista klikkaamalla renderöityy tammikuun kuukausinäkymä; toiseen suuntaan tämä ei toimi, eli suurempaan aikayksikköön siirtyminen
- mukauttaminen parametreillä, kuten värin vaihtaminen tai taustaruudukon viivojen määrän lisääminen tai vähentäminen.

Moni näistä toiminnoista olisi löytynyt jostain valmiista kirjastosta, mutta esimerkiksi ajassa liikkuminen hiiren klikkauksella olisi ollut vaikeaa toteuttaa kolmannen osapuolen kirjastolla. Myös SSR-puoli sovelluksessa tuo omat haasteensa kirjastojen kanssa.

4.5 Testaus ja virheet

Tuotantokäyttöön menevän sovelluksen yksi tärkeimpiä asioita on se, että siinä ei ole virheitä. Tämän vuoksi sovellusta testattiin virheiden varalta useaan kertaan koko kehitysprosessin ajan.

Aina kun jokin isompi toiminallisuus saatiin valmiiksi, käytettiin aikaa varmistamaan se, ettei tämä uusi funktio riko sovellusta.

Automatisoitua testausta, kuten Unit Test -menetelmää, ei tässä projektissa käytetty, vaan prototyyppiä kokeilivat tiimiläiset tarkoituksena löytää virheitä.

Projektin alussa tiimin kanssa tuli puheeksi, että palvelinpuolella on kriittinen virhe portteihin liittyen. Tämä kuitenkin osoittautui vääräksi oletukseksi. Kokonaisuudessaan mitään isompia virheitä ei löytynyt, vaan kaikki olivat lähinnä ulkoasuun vaikuttavia, kuten esimerkiksi sellainen, joka laittoi näytön tärisemään. Tämä johtui elementtien vääristä CSS-ominaisuuksista.

Oman diagrammin sovittaminen erikokoisiin resoluutioihin vaati paljon testaamista. Lopulta siitä saatiin kuitenkin yhteensopiva ja responsiivinen.

4.6 Projektin tulos

Projektin tavoitteena oli, että saadaan mahdollisimman paljon hyödynnettyä automatisaatiota hallinnan minimoimiseen sekä käyttää mahdollisimman vähän kolmannen osapuolen koodia tuotantoon menevään sovellukseen. Ohjelmistokehyksiä ei myöskään haluttu käyttää alusta alkaen, joten sovellus tuotettiin Vanilla JavaScriptillä kuten oli tarkoitus. Tässä luvussa tarkastellaan, millainen oli lopputulos ja miten tavoitteissa onnistuttiin. Tuotteen nimeksi annettiin ”Uusi vesipalvelu”.

Projektin vaatimuksista automatisaatio oli tärkein prioriteetti. Useampien prototyyppien ja testauksien jälkeen sovellus saatiin hiottua lopulta tuotteeksi, johon toimeksiantaja oli tyytyväinen. Sovelluksen automaatio saatiin samanlaiseen

tavoitteeseen kuin alkuperäisessä suunnitelmassa oli. Alun perin ei vielä oltu varmoja, käytetäänkö Server-side renderingiä vai ei. Kun projekti oli saatu aluilleen ja PM2 toimimaan, idea SSR:n käyttämisestä HTML-instanssien luomiseen tuli ajankohtaiseksi. Tämä toteutus onnistui, niin kuin sen oli tarkoituskin siitä asti, kun SSR tuli puheeksi.

Sovelluksen ulkoasu ja käyttöliittymä kuitenkin muuttuivat useaan kertaan alkuperäisestä prototyypin mallista. Projektin kehityskaaren tärkeimpänä prioriteettina oli saada toiminallisuus kuntoon, joten kehitysprosessissa ulkoasua muutettiin aina pikkuhiljaa sitä mukaa, kuin uusia funktioita saatiin valmiiksi.

Alkuperäisessä sovelluksessa oli tarkoitus olla enemmän valinta- ja pudotusvalikkotoimintoja, mutta osasta niistä luovuttiin projektin kehityksen mukana, esimerkiksi datan hakemisesta kahden annetun aikavälin ajalta. Lopulta käyttöliittymästä saatiin kompakti. Ulkoasusta tuli visuaalisesti helppokäyttöisen näköinen, kun sovelluksessa ei ollut liikaa erilaisia nappeja tai valintoja. Alkuperäiset kuvakkeet ja kuvat vaihdettiin. Tyylitiedostoa jouduttiin muuttamaan useaan kertaan, ja responsiivisuuden saavuttaminen vei aikaa. Responsiivinen käyttöliittymä tarkoittaa sovelluksen ulkoasun skaalautumista ja mukautumista erilaisiin näytönkokoihin ja resoluutioihin, kuten esimerkiksi tietokoneen näytön lisäksi tabletteihin ja puhelimiin.

Alun perin diagrammiksi oli mietitty toista kirjastoa, joka oli toisessa projektissa. Projektin SSR-lähestymistavan ja Vanilla JavaScriptin käytön takia tämä ei kuitenkaan toiminut. Graafiksi yritettiin toista kirjastoa, mutta sen toiminallisuuksien puutteiden takia päädyttiin tekemään oma kirjasto. Diagrammin palkkiin hiirellä klikkaaminen ja siitä seuraava toiminto ei kuulunut alkuperäiseen ideaan, mutta se keksittiin, kun päädyttiin kehittämään oma diagrammikirjasto.

Kommentointi ja lisätiedot -kenttään oli alun perin tarkoitus ottaa järjestelmävalvojan antamat parametrit. Myöhemmin kuitenkin idea muuttui niin, että loppukäyttäjät voivat kirjoittaa muistiinpanoja. Tässä vaiheessa piti tehdä tietokantaan uusia tauluja ja muuttaa taulukon generointifunktiota.

Järjestelmävalvojan oman paneelin ulkoasu jäi keskeneräiseksi. Visuaalisuudeltaan sen käyttöliittymä on alkeellisen näköinen, mutta sen toiminallisuudet kuitenkin saatiin toimimaan. Järjestelmävalvojan paneelin ulkoasun puute ei kuitenkaan ole kriittinen, sillä se ei ole näkyvissä kuin itse sovelluksen kehittäjille ja ylläpitäjille. Loppukäyttäjän käyttöliittymästä saatiin ammattimaisen näköinen.

Käyttäjätestausta lopullisille käyttäjille ei kunnolla saatu vielä kokeiltua, koska aikataulun kanssa oli kiireitä. Sovellus kuitenkin meni testikäyttöön organisaation sisällä sekä muutamalle oikealle kohteelle. Tarkoituksena on kuitenkin saada sovellus käyttöön paljon useammalle käyttäjälle.

Kokonaisuutena projektista on se, että se onnistui paremmin kuin alkuperäinen sovellusidea oli. Ulkoasu on hienompi kuin alkuperäisessä prototyypin mallissa. Lisäominaisuudet, kuten muistiinpanot, tekivät sovelluksesta käyttäjäystävällisemmän. Mutta tärkein, eli automatisaatio ja riippumattomuus kolmannesta osapuolesta, onnistui parhaiten. Tämä osuus valmistui melkein samanlaiseksi kuin alkuperäinen suunnitelma oli. Yhteenvetona projektin käyttöliittymä muuttui radikaalisti kehityksen aikana, mutta funktionaalisuus pysyi alkuperäisessä suunnitelmassa.

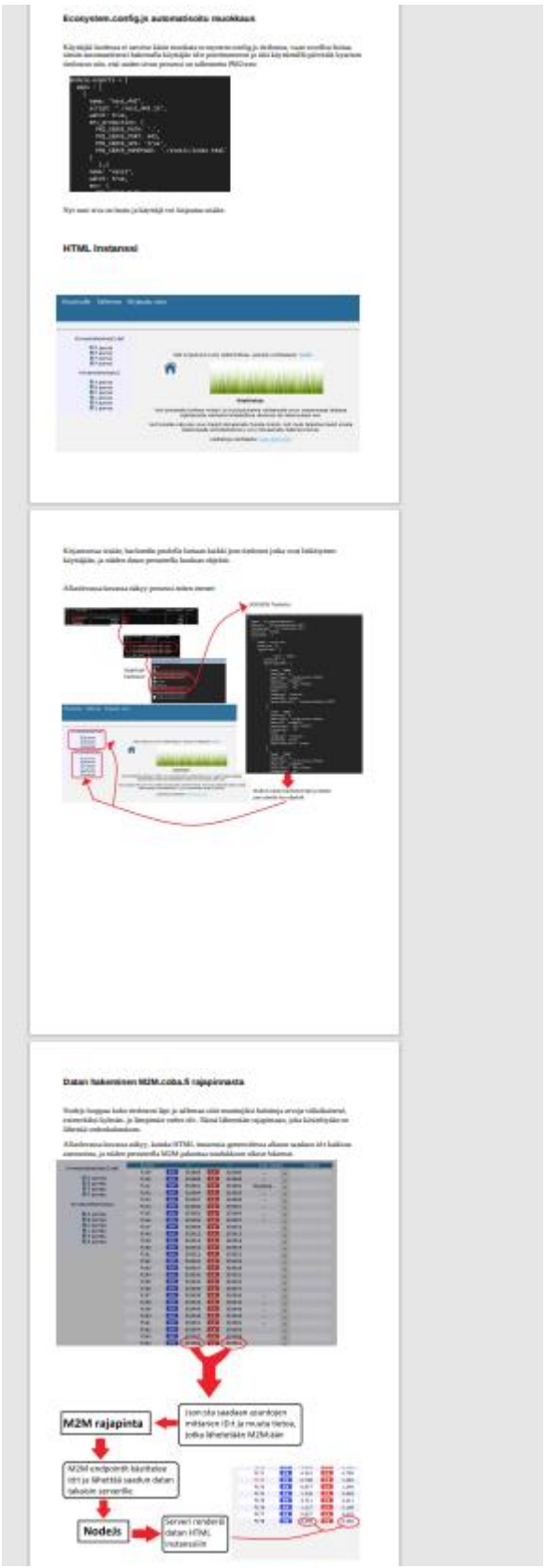
Sovelluksen ylläpito ja jatkokehitys

Kehittäjän ja järjestelmävalvojan näkökulmasta sovelluksen ylläpito näyttää valoisalta. Koska kirjastojen käyttö on minimoitu ja funktionaalisuus on automatisoitu mahdollisimman hyvin eikä kriittisiä virheitä löytynyt, on sovellusta helppo ylläpitää.

Jatkokehityksen kannalta ei ohjelmoijan tarvitse päivittää asiakaspuolen kirjastoja, vaan hänellä on valmiina helposti luotettavaa lähdekoodia projektin omista kirjastoista, joihin voi lisätä ominaisuuksia.

Sovelluksesta tehtiin myös kehittäjille noin 10-sivuinen dokumentaatio, jossa näytetään askel askeleelta, miten sovellus alustetaan (kuva 22). Tämä ohje alkaa PM2:n ohjauksesta ja projektin käynnistämisestä palvelimella. Sen jälkeen

se sisältää yksityiskohtaisesti kirjoitettua tietoa tärkeimmistä tiedostoista, funktioista ja luokista. Mukana on ruutukaappauksia sekä lähdekoodista että itse sovelluksesta. Näissä kuvissa selitetään asioita selkokielellä ilman, että lukijan tarvitsee lukea lähdekoodia ja analysoida sitä.



Kuva 22. Kehittäjälle tarkoitettu dokumentaatio, joka helpottaa jatkokehitystä.

Jatkokehityksessä voitaisiin keskittyä järjestelmävalvojan paneeliin seuraavaksi, koska sen ulkoasun muuttaminen paremmaksi voisi helpottaa paneelin käyttöä. Vaikka paneeli ei tällä hetkelläkään tarvitse paljoa interaktiota käyttäjän kanssa, selkeä ulkoasu ei ole pahitteeksi. Tällä hetkellä paneelista voi luoda uuden käyttäjän yhdellä hiiren klikkauksella.

5 Yhteenveto

Vaikka modernissa web-sovelluskehityksessä käytetään yhä useammin JavaScript-ohjelmistokehyksiä, on Vanilla JavaScript silti olennainen osa tietotekniikkateollisuutta. Kaikki ohjelmistokehykset ovat Vanilla JavaScriptin pohjalta rakennettuja kokonaisuuksia ja niiden osaaminen vaatii ymmärrystä Vanilla JavaScriptistä. Tässä insinöörityössä ei käytetty ohjelmistokehyksiä vaan puhdasta Vanilla JavaScriptiä. Tämä lähestymistapa antoi kehittäjille joustavuutta sovelluksen selainpuolelle vaadittuihin ominaisuuksiin, koska joissain JavaScript-ohjelmistokehyksissä tai -kirjastoissa olisi mahdollisesti tullut vastaan rajoitteita, jotka eivät olisi mahdollistaneet helposti näiden uniikkien ominaisuuksien kehitystä, mitä insinöörityö vaati. Myös palvelinpuolella Node.js:n avulla käytetty SSR-menetelmä helpotti huomattavasti selainpuolen Vanilla JavaScriptin käyttöä.

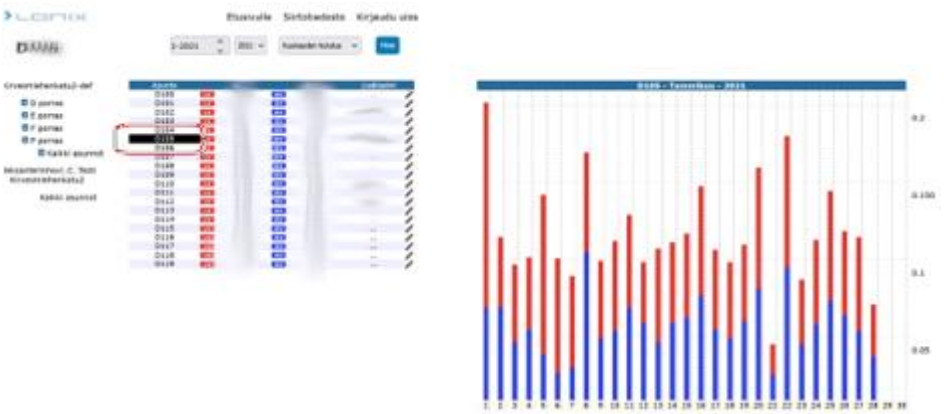
Insinöörityössä tehtiin sovellus, joka tarjoaa loppukäyttäjälle mahdollisuuden tutkia ja analysoida asunnon vedenkulutusta. Tähän analysointiin sovellus antaa laajat työkalut, joilla pystyy tutkimaan dataa vuosikulutuksesta lähtien aina tuntikohtaisen kulutuksen tarkkuuteen.

Projektin tarkoitus oli saada aikaan sovellus, joka olisi mahdollisimman helppokäyttöinen sekä loppukäyttäjälle että järjestelmävalvojalle. Loppukäyttäjälle, kuten isännöitsijä, oli tarkoitus saada helppokäyttöinen käyttöliittymä, jolla hän voi tutkia eri asuntojen vedenkulutusta. Järjestelmävalvojan näkökulmasta taas tarkoitus oli, että sovellus on automatisoitu niin hyvin, että käyttäjien ja sovelluksen hallinta onnistuu ilman ylimääräisiä tehtäviä, joita ihminen joutuisi manuaalisesti

tekemään. Siksi lähdekoodi hiottiin niin, että saatiin mahdollisimman moni asia tehtäväksi tietokoneella.

Toinen seikka projektissa oli, että projektin teknisen puolen kannalta pyrittiin minimoimaan kolmannen osapuolen koodi. Tämä johti siihen, että tarvittiin visuaalisesti näyttävä diagrammi, johon vaadittiin tietyt ominaisuudet sovelluksen kannalta, ja näin päädyttiin kehittämään oman JavaScript-diagrammikirjasto.

Hyviä puolia tässä oman pienen kirjaston kehittämisessä on, että kehittäjä pystyy itse hallitsemaan paljon paremmin kirjaston toiminnallisuutta sekä muokkaamaan sitä. Huonona puolena taas on se, että kirjaston kehittäminen vie paljon aikaa. Esimerkiksi tapauksissa, joissa aikataulu on kiireinen ja asiakas haluaa, että tuote valmistuu nopeasti, on valmiiden kirjastojen käyttö parempi vaihtoehto kehityksen nopeuden takia. Lopullisessa versiossa saatiin sommiteltua sekä taulukko että diagrammi hyvin ulkoasuun (kuva 23).



Kuva 23. Viimeisimmän version käyttöliittymän ulkoasu.

Pitkällä aikavälillä tämä voi kuitenkin olla huonompi päätös, kun tuotetta halutaan ylläpitää ja kolmannen osapuolen kirjaston toiminnallisuus on rajattua. Itse tehtyä sovellusta on paljon helpompi ylläpitää, ja sitä pystyy tarvittaessa jatkokehittämään. Kolmannen osapuolen kirjastot taas vaativat päivityksiä, ja niissä voi tulla vastaan lisenssiongelmia.

Sovelluksen automatisaation osuus ja oman kirjaston luominen onnistui hyvin, mutta ulkoasua jouduttiin vaihtamaan muutaman kerran, kunnes projektipäällikkö oli siihen tyytyväinen.

Sovellus saatiin valmiiksi, mutta ei vielä kunnolla lopulliseen käyttöön useammalle loppukäyttäjälle.

Lähteet

1 Northwood, Chris. 2018. The Fullstack Developer. Appress.

2 How important is JavaScript for Modern Web Developers? 2017. Verkkoaineisto. Mindfire Solutions. <https://medium.com/@mindfiresolutions.usa/how-important-is-javascript-for-modern-web-developers-2854309b9f52>. Luettu 15.8.2021.

3 Madooei, Ali. 2020. Client-Server Application. Verkkoaineisto. <https://madooei.github.io/cs421_sp20_homepage/client-server-app/>. Luettu 16.8.2021.

4 Node.js. Verkkoaineisto. <https://nodejs.org/en/>. Luettu 16.8.2021.

5 Goel, Aman. 2021. 10 Best Web Development Frameworks. Verkkoaineisto. <https://hackr.io/blog/web-development-frameworks>. 11.5.2021. Luettu 17.8.2021.

6 Product development: The Waterfall methodology (model) in software development. Verkkoaineisto. MaRS. <https://learn.marsdd.com/article/product-development-the-waterfall-methodology-model-in-software-development/>. Luettu 17.8.2021.

7 Agile Software Development. Verkkoaineisto. Agile Alliance. <https://www.agilealliance.org/agile101/>. Luettu 16.08.2021.

8 KemperIan, Chris. 2012. Foundation Version Control for Web Developers. Oxley.

9 Stopak, Jacob. 2019.The Evolution of Version Control System (VCS) Internals. Verkkoaineisto. <https://initialcommit.com/blog/Technical-Guide-VCS-Internals>. 30.11.2019. Luettu 18.8.2021.

10 Branching and Merging. Verkkoaineisto. Git. <https://git-scm.com/about>. Luettu 19.8.2021.

11 Matthew, Martin. 2021. Prototyping Model in Software Engineering: Methodology, Process, Approach. Verkkoaineisto. <https://www.guru99.com/software-engineering-prototyping-model.html>. 27.4.2021. Luettu 29.8.2021.

12 Definition of 'Unit Testing‘. Verkkoaineisto. The Economic Times. <https://economictimes.indiatimes.com/definition/unit-testing>. Luettu 24.8.2021.

13 Cloud, Nicholas; Sufyan, bin Uzayr & Ambler, Tim. 2019. JavaScript Frameworks for Modern Web Development. Appress.

14 Vanilla Software. Verkkoaineisto. WhatIs.com®. <https://whatis.techtarget.com/definition/vanilla> Luettu 16.8.2021.

15 Crowder, T. J. 2020. JavaScript: The New Toys. Wrox Press.

16 Kopecky, Christina. 2020. JavaScript Versions: How JavaScript has changed over the years. Verkkoaineisto. <https://www.educative.io/blog/javascript-versions-history>. 18.12.2020. Luettu 25.8.2021.

17 A re-introduction to JavaScript. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript>. Luettu 17.8.2021.

18 JavaScript. Verkkoaineisto. MDN Web Docs. < https://developer.mozilla.org/en-US/docs/Web/JavaScript >. Luettu 17.8.2021.

19 Morris, Scott. Tech 101: JavaScript Frameworks vs Libraries—What's the Difference? Verkkoaineisto. <https://skillcrush.com/blog/javascript-frameworks-vs-libraries/>. Luettu 28.8.2021.

20 Baer, Eric. 2018. What React Is and Why It Matters. O'Reilly Media.

21 Why ReactJS is gaining so much popularity these days. Verkkoaineisto. Thinkwik. <https://medium.com/@thinkwik/why-reactjs-is-gaining-so-much-popularity-these-days-c3aa686ec0b3>. 6.12.2017. Luettu 19.8.2021.

22 Dhaduk, Hiren. 2021. Angular vs Vue: Which Framework to Choose in 2021? Verkkoaineisto. <https://www.simform.com/blog/angular-vs-vue/>. 17.6.2021. Luettu 27.8.2021.

23 Sofiene, Ben Khemis. 2019. What is the Difference Between a Framework and Library? Verkkoaineisto. <https://sofienebk.medium.com/what-is-the-difference-between-a-framework-and-library-2b712a1a1c41>. 16.12.2019. Luettu 28.8.2021.

24 Momin, Afraz. 2020. React v/s Vanilla JS - When to use what? Verkkoaineisto. <https://dev.to/afrazchelsea/react-vs-vanilla-js-what-why-and-when-1jin>. 14.4.2020. Luettu 2.8.2021.

25 Rudenko, Artem. 2020. Security concerns with using third-party dependencies. Verkkoaineisto. <https://ottofeller.com/blog/security-concerns-with-using-third-party-dependencies>. 16.10.2020. Luettu 1.9.2021.

26 Liang, Y.E. 2014. JavaScript Security. Packt Publishing.

27 What is cross-site scripting? Verkkoaineisto. Cloudflare. <https://www.cloudflare.com/learning/security/threats/cross-site-scripting/>. Luettu 30.8.2021.

28 Sargent, Jenna. 2017. Security vulnerabilities in JavaScript libraries are hard to avoid. Verkkoartikkeli. <https://sdtimes.com/angular/security-vulnerabilities-javascript-libraries-hard-avoid/>. 23.10.2017. Luettu 3.9.2021.

29 Auditing package dependencies for security vulnerabilities. Verkkoaineisto. NPM. <https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities>. Luettu 3.9.2021.

30 Why Source Code Quality Is Crucial in Software Product Development.2020. Verkkoaineisto. RabIT software engineering. <https://www.rabitse.com/blog/why-source-code-quality-is-crucial/>. 20.3.2020. Luettu 25.8.2021.

31 Avoid using Javascript libraries (Jquery). Verkkoaineisto. GiftOfSpeed. <https://www.giftofspeed.com/dont-use-javascript-libraries/> Luettu 21.8.2021.

32 Aphinya, Dechalert. 2021. Why you should learn vanilla JavaScript in isolation if you want to be a better developer. Verkkoaineisto. <https://www.dottedsquirrel.com/learn-vanilla-javascript>. 26.3.2021. Luettu 2.9.2021.

33 Engebretson, Patrick. 2011. The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy. 2nd Edition. Syngress Media.

34 Proprietary software. Verkkoaineisto. Unix Operating System Organization. <https://www.gnu.org/philosophy/categories.html#ProprietarySoftware>. Luettu 3.9.2021.

35 MIT Licence. Verkkoaineisto. Debian. < https://www.debian.org/legal/licenses/ >. Luettu 4.9.2021.