Modulaaristen skriptien hallinnointityökalu

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestitekniikka, insinööri (AMK), Riihimäen kampus

Kevät, 2021

Tony Reittu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka, insinööri (AMK) Tiivistelmä

Riihimäen kampus

Tekijä Tony Reittu Vuosi 2021

Työn nimi Modulaaristen skriptien hallinnointityökalu

Ohjaajat Petri Kuittinen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa monialustainen työpöytäsovellus, jolla tiettyyn tarkoitukseen jalostettuja modulaarisia skriptejä voitaisiin hallita argumenteittain niin yksitellen kuin myös muodostaa näistä skripteistä käyttöliittymältä yhä tarkempiin automaatiotarpeisiin soveltuvia skriptiketjuja. Osana skriptien organisoitua hallinnointia ja työtehokkuuden kasvattamista oli myös implementoida pilvipalvelupohjainen sidosjärjestelmä, josta työkalun käyttäjien olisi mahdollista jakaa, etsiä ja ladata tarpeisiinsa soveltuvia skriptejä sekä skriptiketjuja.

Lopputuloksena syntyi hyvin käyttökelpoinen ohjelmisto, joka soveltuu erinomaisesti allekirjoittaneen automaatiotarpeisiin. Skriptiketjujen muodostaminen ja muokkaaminen onnistuu käyttöliittymällä suoraviivaisesti, ja lukuisat implementoidut kontrollirakenteet mahdollistavat monimutkaisempienkin kokonaisuuksien automatisoinnin. Toteutuksessa valmistui myös käyttökelpoinen pilvipalvelupohjainen skriptien ja skriptiketjujen jakelualusta, joka etenee toteutetun sovelluksen kanssa jatkokehitykseen. Projektin lähdekoodi sekä käyttöä helpottava oheismateriaali tulevat julkiseen jakoon.

Avainsanat Automaatio, AWS, Electron, JavaScript, React

Sivut 62 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor of Engineering, ICT **Abstract**

Riihimäki campus

Author	Tony Reittu	Year 2021
Subject	Management tool for modular scripts	
Supervisors	Petri Kuittinen	

ABSTRACT

The aim of this thesis was to design and implement a GUI-based desktop application capable of managing arguments of modular scripts both individually as well as to form script chains from these individual scripts for even more precise tasks. As part of improving organized management of scripts and work efficiency, it was also necessary to design and implement a simple binding system with which one could share, search and download scripts and script chains suitable for one’s needs.

As a result of this project came about a very practical software that met the set requirements. With the tool, it is straightforward to create and modify script chains with user interface, and the numerous implemented control structures enable automation of even more complex tasks. Along with the rest of the software, a usable cloud service based distribution platform for users to share their scripts and script chains on was configured and deployed. All the code will be open sourced together with all relevant reference material, and the product will be further improved according to the emerging needs.

Keywords	Automation, AWS, Electron, JavaScript, React
Pages	62 pages

Sisällys

1 Johdanto 1
2 Kehittämistyön tietoperusta 3
2.1 Skriptaus 3
2.1.1 AOT 3
2.1.2 Tulkitseminen 5
2.1.3 JIT 6
2.1.4 JavaScript 7
2.2 Arkkitehtuurisuunnittelu 9
2.2.1 C4-malli 9
2.3 Projektinhallinta 10
2.3.1 Scrum 11
2.3.2 Jira 13
2.4 Työpöytäkehityksen sovelluskehykset 14
2.4.1 Electron 15
2.5 Käyttöliittymäkehityksen sovelluskehykset 16
2.5.1 React 17
2.6 Pilvipalvelut 18
2.6.1 AWS 20
2.7 Muut työvälineet 21
2.7.1 Visual Studio Code 21
2.7.2 Git 22
2.7.3 Figma 23
3 Kehittämistyön tavoite ja tarkoitus 24
4 Tuotteen suunnittelu ja toteutus 26
4.1 Suunnitteluprosessi 26
4.2 Toteutus 30
4.2.1 Sprintti 1 31
4.2.2 Sprintti 2 36
4.2.3 Sprintti 3 40
4.2.4 Sprintti 4 43
4.2.5 Viimeistely 50
5 Johtopäätökset ja pohdinta 52
5.1 Lopputulos ja toteutuksen arviointi 52

5.2 Electron-sovelluskehys ... 54
5.3 Jatkosuunnitelmat ... 57
Lähteet ... 59

Kuvat, taulukot ja kaavat

Kuva 1. Yleiskuva työkalun perustoiminnallisuuksista. ... 2
Kuva 2. AOT-kääntäminen. ... 4
Kuva 3. Tulkitseminen. ... 5
Kuva 4. Esimerkki JIT-kääntämisestä. ... 7
Kuva 5. C4-mallin eri tasot. Yksittäistä järjestelmää voi tutkia tarkemmin säiliötasolla, säiliöitä komponenttitasolla ja komponenttia kooditasolla. ... 10
Kuva 6. Esimerkki Jiran sprinttinäkymästä. ... 13
Kuva 7. Google Trends -viivakaavio eri teknologioiden suosioon linkittyvistä hakumääristä 2015–2020. Purppura on jQuery, sininen React, punainen Angular, keltainen Vue.js ja vihreä Svelte.js. ... 17
Kuva 8. Järjestelmäkaavio toteutettavasta kokonaisuudesta. ... 26
Kuva 9. Järjestelmien yhdistetty säiliönäkymä. ... 27
Kuva 10. Työpöytäsovelluksen komponenttinäkymä. ... 28
Kuva 11. Jira-backlog suunnitteluvaiheen päätteeksi. ... 29
Kuva 12. Käyttöliittymän ulkoasun prototyyppi (Figma). ... 30
Kuva 13. Projektin hakemistorakenne ilman konfiguraatio- ja seurantatiedostoja. ... 32
Kuva 14. Sovelluksen oletusnäkymän asettelu toteutettiin CSS Grid -tekniikalla. ... 35
Kuva 15. Ruudukon yksittäisten osioiden sisäisten elementtien asettelu toteutettiin pääosin flexbox-tekniikalla ... 35
Kuva 16. Sovelluksen tilanne sprintin 2 päätteeksi: skriptiketjun suoritus ja lokitus käyttöliittymällä. ... 40
Kuva 17. Käyttäjäasetukset. ... 41
Kuva 18. Identiteettiryhmän määrittelyjä (Federated Identities). ... 45
Kuva 19. Rekisteröityneen käyttäjän roolille linkitetty S3-palvelun linjaus. ... 46
Kuva 20. Kirjautumisnäkymä uloskirjautumisen jälkeen. ... 48
Kuva 21. Vaalea teema ja skriptien uudelleenjärjestäminen. ... 49

Kuva 22. Gallerianäkymä. 50
Kuva 23. Sovelluksen testausta Ubuntu-käyttöjärjestelmässä ja objektimuotoisen tietorakenteen jäsennystä pistenotaatiolla. 51
Kuva 24. Tietokoneiden keskimääräisen keskusmuistin kehitys 2000–2020 (PC Matic, 2020). 55

Taulukko 1. Alkuperäisten tavoitteiden toteuma (MVP). 52
Taulukko 2. Alkuperäisten tavoitteiden toteuma (ei MVP)..53
Taulukko 3. Kehitysvaiheessa esiin nousseet tarpeelliset toiminnot.............................53

1 Johdanto

Skriptien kirjoittaminen ja suorittaminen on arkipäivää monille sovelluskehittäjille, DevOps-spesialisteille, järjestelmänvalvojille, tietokanta-asiantuntijoille, automaatiotestaajille kuin myös lukuisille muille IT-alan asiantuntijoille ja tehokäyttäjille. Kun automatisoitavat tehtävät ovat monivaiheisia ja vaativat kommunikointia useissa eri rajapinnoissa tai koskevat vähemmän suosittuja ohjelmia, on valmiiden ratkaisujen löytäminen todella harvinaista. Vaihtoehdoiksi jää tällöin usein kirjoittaa skripti itse joko täysin tyhjästä tai muokata jotakuinkin samankaltaiseen tarkoitukseen soveltuva skripti haluttuun tehtävään sopivaksi.

Vaikka skriptejä hyödynnetään automatisoinnissa laajalti, eivät ne toki ole ainut tapa automatisoida työtehtäviä. Maksuttomien ja maksullisten työkalujen, kuten n8n ja Zapier, avulla myös vähemmän teknisemmät käyttäjät voivat automatisoida erinäisiä työnkulkuja suoraviivaisesti käyttöliittymällä. Kaikilla näillä palveluilla on kuitenkin omat rajoitteensa, ja skriptit ovat käytännössä ainut tapa automatisoida kaikista erityislaatuisimmat toiminnot, joita nämä työkalut eivät tue. Sopivien skriptien etsiminen, muokkaaminen ja ohjattu suorittaminen vievät kukin kuitenkin huomattavasti aikaa. Mikäli käytettävissä olisi näitä tehtäviä varten optimoitu sovellus, säästyisi aikaa erinäisten komentojen ja skriptien integroinnilta sekä monimutkaisimpien skriptikokonaisuuksien argumenttien syöttämiseltä.

Looginen seuraava askel olikin alkaa kehittämään skriptien organisointiin, ajamiseen, ketjutukseen ja hakemiseen soveltuvaa ohjelmistoa. Tässä opinnäytetyössä keskitytään tehostamaan työtehtävien automatisointia suunnittelemalla ja toteuttamalla sovellus, jolla yksittäisten skriptien suorituksen argumentteja voidaan käyttöliittymätasolla hallita ja tallentaa näistä määrityksiä, jotka nopeuttavat skriptien myöhempää uudelleen suorittamista halutuilla argumenteilla. Skriptien ketjutus eli suorittaminen yksitellen määritellyssä järjestyksessä ja halutuilla argumenteilla on myös osa tämän prosessin tehostamista, minkä toteutettavan sovelluksen pitää ehdottomasti mahdollistaa. Mitä enemmän modulaarisuutta skripteillä on, sitä vähemmän täytyy kustomoituja ja monivaiheisia skriptejä luoda, jos käytettävissä on ketjuttamiseen soveltuva työkalu. Ketjuttamisen täytyy myös tukea syötteen ja tulosteen monipuolista hallintaa, jotta toisen skriptin tulosta voidaan käyttää jonkin sitä seuraavan skriptin argumenttina. Kuvassa 1

havainnollistetaan näitä työkalun perustoiminnallisuuksia. Ketjutettavat skriptit voivat myös olla monella eri kielellä kirjoitettuja, ja ketjun yksittäisen skriptin tulosteita voidaan käyttää jonkin sitä seuraavan skriptin argumentteina.

Tiivistettynä työkalun tulee vastata sopivien skriptien löytämiseen, kirjoittamiseen ja suorittamiseen liittyviin ajankäytön haasteisiin ja olla omalta osaltaan edistämässä avoimen lähdekoodin modulaarisia ratkaisuja, jotka säästävät aikaa ja toimivat referenssimateriaalina vastaavanlaisten työtehokkuutta parantavien välineiden kehityksessä. Järjestelmän tarkempiin määrittelyihin palataan vielä kehittämistyön tavoitetta käsittelevässä luvussa 3.

Kuva 1. Yleiskuva työkalun perustoiminnallisuuksista.

2 Kehittämistyön tietoperusta

Työssä käsitellään sovelluskehitystä ja tähän liittyy useita tärkeitä valintoja niin työmenetelmien kuin työvälineiden osalta. Tässä luvussa käydään läpi työn oleellisimmat teknologiat ja työkalut, perustellaan niiden valinnat toteutuksessa sekä avataan työhön liittyvää teoriaa varsinkin skriptauksen osalta.

2.1 Skriptaus

Skripti määritellään yleensä pienehköksi ohjelmaksi, joka suoritetaan tavanomaisesti asianmukaista kieltä tukevaa tulkkia (interpreter) hyödyntämällä. Siinä missä ohjelmat perinteisesti vaativat perusteellisen kääntämisen ihmisluettavasta lähdekoodista prosessorin ymmärtämäksi konekieleksi ennen kuin suorittaminen on mahdollista, skriptin suorittaminen voidaan aloittaa välittömästi, kun tulkki on muuntanut lähdekoodin ensimmäisen rivin konekieleksi. Skriptauskielten tulkeilla on usein myös niin sanottu REPL-tila (read-eval-print-loop), jossa käskyjä voidaan syöttää rivi riviltä, jolloin tulkki evaluoi komennon, palauttaa vastauksen ja odottaa seuraavaa käyttäjän syötettä. (1 & 1 IONOS, 2020)

Ohjelmointikieliä voidaan jaotella skriptauskieleksi sen mukaan, ajetaanko niitä tyypillisesti lähdekoodista suoraan tulkin avulla vaiko kääntämällä lähdekoodi ensin kokonaisuudessaan konekieleksi. On kuitenkin huomionarvoista, että tämä jaottelu on yleistystä eikä kieli itsessään ole käännetty tai tulkittu, vaan sen implementaatio. Usean ohjelmointikielen lähdekoodi voidaan sekä tulkita että kääntää hyödyntämällä asianmukaisia työkaluja, mutta yleistyksissä viitataan tavanomaisiin suoritusmenetelmiin (BGZDevTips, 2016). Jos kieli siis soveltuu paremmin tulkituksi kuin käännetyksi, sitä pidetään tulkittuna kielenä. Raja on näissä tilanteissa hyvin häilyvä ja tärkeämpää on ymmärtää, miten kääntäminen ja tulkitseminen eroavat toisistaan.

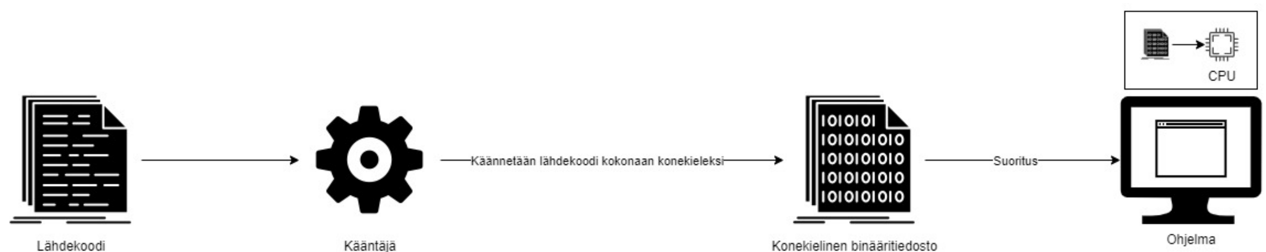
2.1.1 AOT

AOT-kääntämisellä (ahead-of-time compilation) viitataan käytäntöön, jossa lähdekoodi käännetään kokonaan konekieleksi ennen suorittamista. Kääntämisprosessissa koodi muun muassa tarkastetaan virheiden osalta ja sille voidaan tehdä kääntäjästä riippuen optimointia,

jotta suorituskyky olisi kohdejärjestelmässä ohjelmaa suorittaessa mahdollisimman hyvä. Kääntämisen tuotoksena on binääritiedosto, joka voidaan suorittaa sellaisenaan ilman erillistä tulkkia, kunhan ohjelmaa suorittavan järjestelmän käyttöjärjestelmä vastaa kääntäjän kohdekäyttöjärjestelmää. (Wikipedia, 2020)

Kuvassa 2 näkyy abstraktoituna lähdekoodin AOT-kääntäminen ja suoritus: lähdekoodi käännetään kokonaisuudessaan konekieliseksi ennen suoritusta, jolloin ohjelmaa suorittaessa prosessori ymmärtää kaikki alkuperäisessä lähdekoodissa annetut käskyt eikä kääntäjää tarvita enää binääritiedoston muodostamisen jälkeen.

Kuva 2. AOT-kääntäminen.



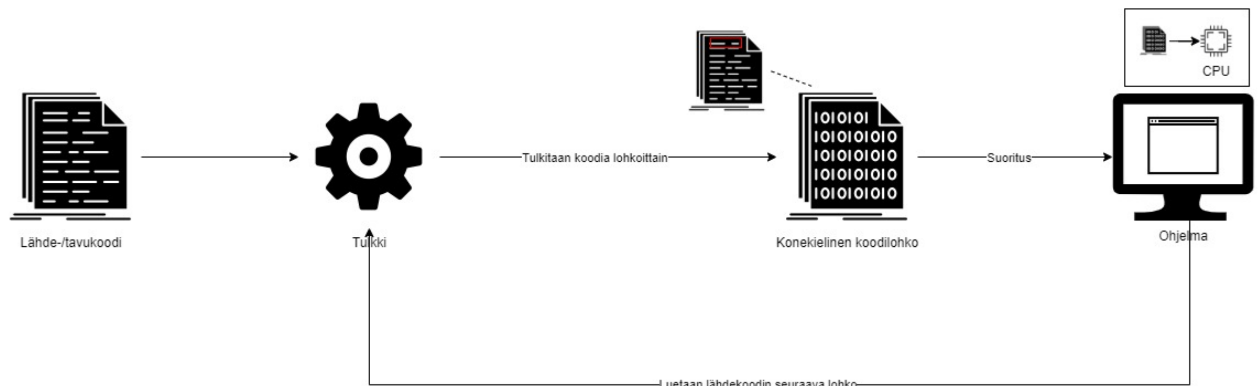
Sovelluksen ladannut käyttäjä ei voi kääntää konekielistä binääritiedostoa kokonaan takaisin alkuperäiseksi lähdekoodiksi. Tähän tarkoitukseen on kuitenkin olemassa työkaluja, joilla konekielinen tiedosto voidaan kääntää osittain käytetyn ohjelmointikielen mukaiseksi lähdekoodiksi tai vaihtoehtoisesti kokonaan symboliseksi konekieleksi (Assembly). Näistä käänteismallinnuksen menetelmistä käytetään termejä decompilation ja disassembly. (Krügel ym., 2004, s. 1)

Vaikka sovelluksen kääntäminen kokonaan ennen suoritusta mahdollistaa kattavan optimoinnin vuoksi erittäin hyvän suorituskyvyn, AOT-kääntäminen ei sovellu kuitenkaan hyvin kaikille kielille. Jos kieli on dynaaminen ja heikosti tyypitetty, kääntäjällä saattaa olla vaikeuksia päätellä muuttujien pituuksia ja tyyppejä (Seymour, 2020). Tällöin turvaudutaan yleensä joko JIT-kääntämiseen tai tulkitsemiseen.

2.1.2 Tulkitseminen

Tulkitulla ohjelmointikielellä tarkoitetaan perinteisesti kieltä, jossa ohjelmakoodi voidaan suorittaa muuntamalla se konekieleksi suorituksen aikana rivi riviltä hyödyntämällä asianmukaista tulkkia (Kuva 3). Tulkitulla kielellä kirjoitettu ohjelma voidaan siirtää ihmisluettavassa muodossa tietokoneelta toiselle ja suorittaa sellaisenaan, jos kohdejärjestelmässä on kieltä tukeva tulkki. Vaikka tulkitun lähdekoodin suorittaminen alkaa lähes välittömästi, tämä ei takaa sitä, että skripti suoriutuisi loppuun nopeammin kuin vastaavan lähdekoodin kääntäminen ja suorittaminen. Tämä riippuu pitkälti siitä, kuinka paljon optimoitavaa koodissa on. (Wikipedia, 2021g) Jos lähdekoodissa suoritetaan sama toiminto useita kertoja esimerkiksi silmukan sisällä, täytyy tulkin muuntaa sama lohko konekieleksi useita kertoja, kun taas kääntäjän puolestaan vain kerran.

Kuva 3. Tulkitseminen.



Nykyään ei ole olemassa montaa puhtaasti tulkittua kieltä, vaan useat tulkituiksi kieliksi miellettyjen kielten lähdekoodi käännetään joko ennen suoritusta tai suorituksen aikana konekieltä lähellä olevaksi tavukoodiksi (bytecode), joka on luokiteltavissa välikieleksi (indermediate language – IL). Tavukieli on usein alustariippumatonta, jolloin se on lähdekoodin tapaan hyvin siirrettävissä. Tavukieli on myös lähempänä konekieltä kuin lähdekoodi, jonka vuoksi se on muunnettavissa konekieleksi nopeammin. Esimerkiksi modernissa Javassa koodi käännetään ennen suoritusta tavukoodiksi ja suorituksen aikana virtuaalikoneessa edelleen ohjelmaa suorittavan järjestelmän käyttöjärjestelmän ja prosessorin mukaisesti konekieleksi. (Wikipedia, 2021e) JavaScriptin suosituin moottori – V8 – puolestaan kääntää lähdekoodin suorituksen aikana ensin optimoimattomaksi tavukieleksi

ja myöhemmin optimoiduksi konekieleksi (Hinkelmann, 2017). Näissä molemmissa tilanteissa hyödynnetään JIT-kääntämistä.

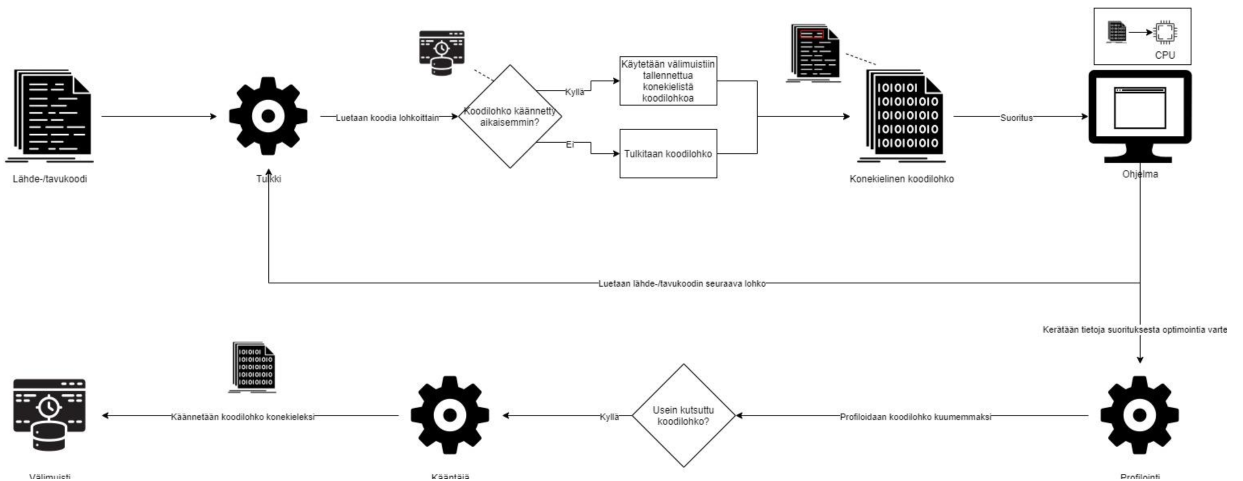
2.1.3 JIT

JIT-kääntäminen (just-in-time compilation) on AOT-kääntämisen ja tulkitsemisen välimuoto, jossa osa lähdekoodista tai tavukoodista käännetään suorituksen aikana konekieleksi. Tämä mahdollistaa tulkitsemista huomattavasti paremman suorituskyvyn ilman, että komentojen tai ohjelman käynnistyksessä olisi yhtä pitkää viivettä kuin AOT-kääntämisessä. Perinteisessä JIT-kääntämisessä kaikki kääntäminen lähdekoodista konekieleksi toteutuu suorituksen aikana. Tämän lisäksi on olemassa muita välimuotoja ja implementaatioita, joissa lähdekoodi esimerkiksi ennen suoritusta käännetään kokonaisuudessaan alustariippumattomaksi tavukoodiksi, joka puolestaan suorituksen aikana käännetään virtuaalikoneessa käytössä olevan prosessorin ymmärtämäksi konekieleksi. Muun muassa Javan käännösprosessissa noudatetaan tätä mallia. (Wikipedia, 2021b)

Kun suoritettavan ohjelmakoodin osia suoritetaan useammin kuin kerran, on järkevämpää suorituskyvyn näkökulmasta tallentaa lähdekoodista konekieleksi käännetyt toistettavat osat välimuistiin ja käyttää niitä seuraavilla iteraatioilla, jotta vältytään turhalta uudelleen kääntämiseltä. Tyypillisesti moderneissa JIT-kääntämisen implementaatioissa kaikki suoritettava koodi profiloidaan aluksi kylmäksi ja mitä useammin tiettyä koodia suoritetaan, sitä kuumemmaksi se merkitään. Koodia optimoidaan suorittamisen yhteydessä sitä enemmän, mitä kuumempaa se on. Näin toimii niin V8-moottori, moderni Java kuin Pythonin vaihtoehtoinen PyPy-implementaatio (Doglio, 2020; Evans, 2014; PyPy, n.d.).

Kuvassa 4 on yksinkertaistettu kaavio tavanomaisesta JIT-kääntämistä. Käytännössä taustalla on usein kuitenkin esitettyä syvällisempää optimointilogiikkaa, jolloin esimerkiksi kerran konekieleksi käännetty kuuma koodiosio voidaan kääntää vielä myöhemmin uudelleen enemmän optimoidummaksi versioksi.

Kuva 4. Esimerkki JIT-kääntämisestä.



Tyypillisesti AOT-kääntämisessä optimointia voidaan tehdä enemmän, koska tavoitteena on toteuttaa mahdollisimman suorituskykyinen ohjelma, eikä kääntämiseen käytettävää aikaa olla rajoitettu. Koodia on mahdollista kuitenkin optimoida JIT-kääntämisessä suorituksen aikana tietyissä tilanteissa jopa enemmän, koska suoritettua koodia voidaan profiloida ja analysoida sekä tehdä kerätyn datan mukaan suorituskykyä tehostavia olettamuksia ja rajauksia. Tämä on oleellista varsinkin pitkän suoritusajan ohjelmissa. Optimointimenetelmät vaihtelevat laajalti ja menevät hyvin tekniselle tasolle, mutta esimerkiksi V8-moottorissa on erillinen tulkki, Ignition, sekä ajon aikainen kääntäjä, TurboFan, joiden integroidessa ajetaan saman suorituksen aikana sekä optimoimatonta että optimoitua koodia (Hinkelmann, 2017).

2.1.4 JavaScript

JavaScript on dynaamisesti tyypitetty ohjelmointikieli, jota käytännössä lähes kaikki verkkosivut ja -sovellukset hyödyntävät jossain määrin. Kieli on Stack Exchangen (2020) teettämän kyselyn mukaan kehittäjien keskuudessa jo kahdeksatta vuotta peräkkäin kaikista käytetyin. Käytännön tasolla JavaScriptiä suoritetaan selaimen JavaScript-moottorissa, jossa syötetty lähdekoodi käännetään ja suoritetaan, jolloin koodista riippuen esimerkiksi jonkin verkkosivulla näkyvän elementin rakenne tai arvo päivittyy. Asiakaspuolen eli käyttäjän selaimen lisäksi JavaScriptiä voidaan nykyään käyttää niin palvelinpuolella kuin jopa sulautetuissa järjestelmissä muun muassa Node.js tai Deno -ajoympäristöjä hyödyntämällä. (Wikipedia, 2021d)

Node.js on tässä työssä käytettävä oleellinen teknologia, joka perustuu JavaScriptiin. Node.js-ajoympäristön avulla voidaan hallita muun muassa sitä ajavan järjestelmän tiedostojärjestelmää ja näin ollen esimerkiksi lukea ja kirjoittaa konfiguraatiotiedostoja sekä suorittaa järjestelmään asennettuja sovelluksia (Zammetti, 2020, s. 2). Node.js soveltuu työssä siis erinomaisesti skriptien hallintaan ja suorittamiseen, eikä ohjelmistojärjestelmän muita komponentteja tarvitse näin ollen integroida toista ohjelmointikieltä – esimerkiksi Pythonia tai C#:a – käyttävän ohjauskomponentin kanssa, vaan kaikki kehitys voidaan toteuttaa samalla kielellä.

JavaScriptiin pohjautuu myös työssä hyödynnettävä TypeScript, joka ekstensiokielenä tuo dynaamisesti tyypitettyyn JavaScriptiin staattisen tyypityksen ja siten sen hyödyt niin dokumentaation kuin bugien ehkäisyn osalta. Mitä suuremmaksi sovellus kehittyy, sitä haastavampaa on ylläpitää kaikkia samaa tietorakennetta käyttäviä koodiosioita, jolloin yksikin pieni muutos yhdessä osiossa saattaa rikkoa toiminnan jossakin toisessa osiossa. Tyypitys ehkäisee suuren osan näistä bugeista, mutta vaatii osaltaan kehitysaikaa. Säästetty aika on kuitenkin suurissa projekteissa huomattava, joten tyypitys nähdään usein investointina tulevaisuuteen. (Microsoft, n.d.) TypeScript lisää tyypityksen lisäksi myös muitakin ominaisuuksia, kuten monista muista ohjelmointikielistä tutut enum ja namespace -avainsanat, joille ei perinteisessä JavaScriptissä ole kirjoitushetkellä tukea (Flanagan, 2020, s. 650). Edellä mainittujen avainsanojen lisäksi TypeScript mahdollistaa myös dekoraattorien (decorator) käyttämisen, joiden avulla luokkia ja niihin liittyviä olioita voidaan laajentaa hyvin kompaktilla syntaksilla (Microsoft, 2021c).

Tässä työssä päädyttiin käyttämään pääasiassa JavaScriptiä, koska olen toteuttanut sillä suoraviivaisesti niin verkkosivuja, web-sovelluksia kuin palvelinpuolen rajapintojakin, ja halusin tutkia käytännön tasolla, kuinka sujuvasti työpöytäsovellusten kehitys onnistuu JavaScriptiin pohjautuvassa Electron-sovelluskehyksessä, joka on jalostettu nimenomaan alustariippumattomaan työpöytäsovelluskehitykseen. Tavoitteena on tämän ohella myös tarkastella, kuinka ajantasaista kyseistä sovelluskehystä vuosien saatossa vastaan esitetty kritiikki on tänä päivänä.

2.2 Arkkitehtuurisuunnittelu

Uutta ohjelmistojärjestelmää rakentaessa on tärkeää toteuttaa suunnitelma, jossa huomioidaan toteutettavan tuotteen vaatimukset. Suunnitelman tulisi olla riittävän kattava, jotta kaikki määrittelyt voidaan täyttää ja vältytään yhteensopivuusongelmilta mahdollisten sidosjärjestelmien kanssa. Huomioitavia asioita ovat muun muassa käytettävät teknologiat, käytössä oleva osaaminen sekä aikataulun rajoitteet. Varsinaista suunnitteluprosessia ja käytettäviä teknologioita käsitellään tarkemmin luvussa 4.

Osana suunnittelua ja dokumentointia on arkkitehtuurin mallinnus. Tämän työn mallinnukseen valikoitui puhtaasta mielenkiinnosta lupaavan oloinen C4-malli, joka suhteellisen uutena tulokkaana haastaa hieman perinteisempiä UML (Unified Modeling Language), 4+1 sekä ERD (Entity Relation Diagrams) -tekniikoita, joihin se myös osittain perustuukin (Wikipedia, 2021a).

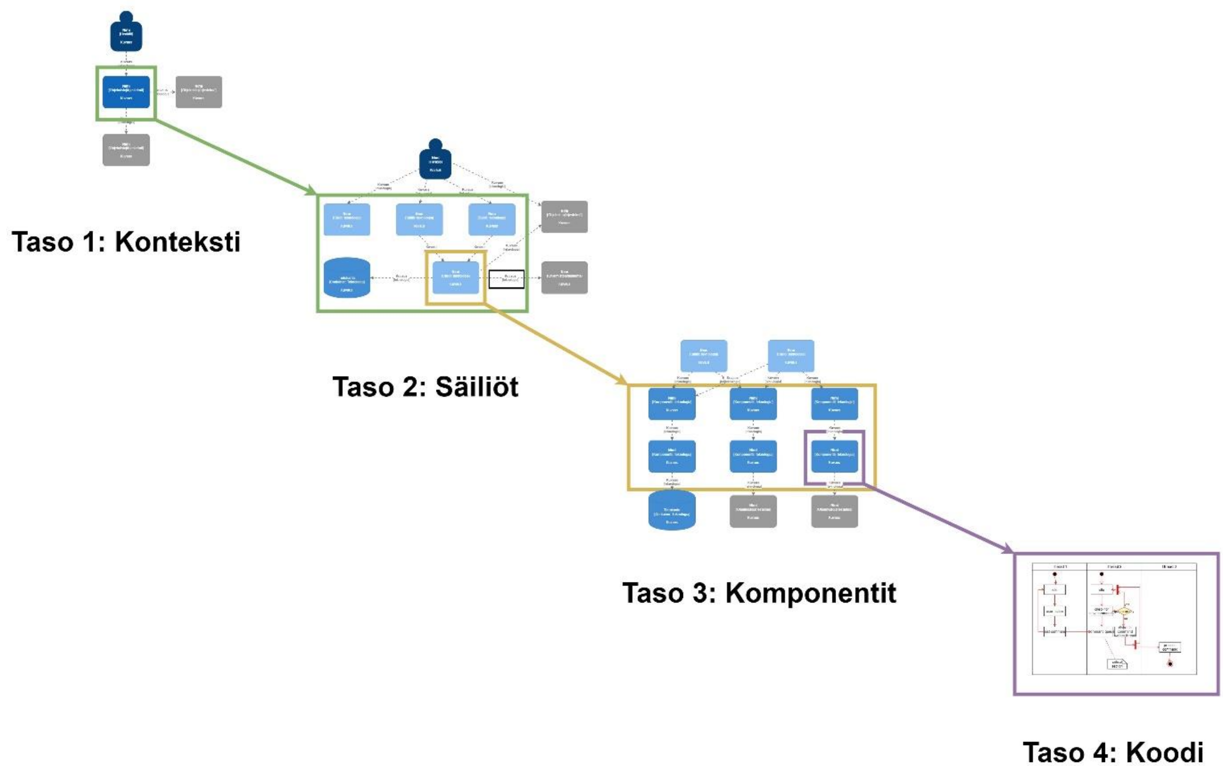
2.2.1 C4-malli

C4 on Simon Brownin (n.d.) kehittämä arkkitehtuurimalli, jossa asteittainen abstraktio on hyvin olennaisessa roolissa. Ohjelmistojärjestelmiä ja niiden komponentteja kuvataan tässä mallissa hierarkkisesti neljässä tasossa:

1. Konteksti (Context)
2. Säiliöt (Containers)
3. Komponentit (Components)
4. Koodi (Code)

Näitä tasoja käyttäen voidaan kohdeyleisön ja tilanteen mukaan mallintaa joko yleisluonteisia järjestelmäkaavioita, joissa järjestelmän toimintaperiaate ja mahdolliset sidosjärjestelmät näkyvät selkeästi vähemmän teknisellä tasolla, tai tarkemmin säiliö-, komponentti- ja kooditasolla, joissa tekninen implementaatio ja suhteet voidaan kommunikoida ymmärrettävästi (Kuva 5).

Kuva 5. C4-mallin eri tasot. Yksittäistä järjestelmää voi tutkia tarkemmin säiliötasolla, säiliöitä komponenttitasolla ja komponenttia kooditasolla.



C4-malli soveltuu hyvin useimpien ohjelmistojärjestelmien mallinnukseen, mutta Brownin (n.d.) mukaan se ei kuitenkaan ole erityisen hyvä valinta muun muassa sulautettujen järjestelmien osalta, vaikka näidenkin kohdalla mallista saattaa olla hyötyä korkeimman tason kaavioissa.

2.3 Projektinhallinta

Projektinhallinta on tärkeä osa onnistunutta ohjelmistokehitysprojektia. Hyvin suunniteltu ja toteutettu projektinhallinta mahdollistaa, että tavoitteet ja työn eteneminen ovat läpinäkyviä koko tiimille ja jokainen ymmärtää vastuunsa sekä toimii niiden puitteissa. Projektinhallinta on parhaimmillaan yhteinen kieli, jonka avulla tehtävät delegoituvat aina oikeille henkilöille ja kriittinen tieto kulkee esteittä kaikille, joita se koskee. Selkeät tavoitteet, tehtävien priorisointi ja toteuman jatkuva seuraaminen puolestaan minivoivat mahdollisia tavoitteiden saavuttamiseen liittyviä riskejä. (Asana, n.d.) Myös

projektinhallinnan ohjelmistot ovat erittäin tärkeitä tuottavuuden näkökulmasta, sillä ne tarjoavat tiimille tehokkaita välineitä dokumentointiin ja viestintään (Kashyapp, 2019).

Ohjelmistontuotannossa käytettävät projektinhallinnan menetelmät lajitellaan usein perinteisiin menetelmiin ja inkrementaalisiin ketteriin menetelmiin. Perinteisiin menetelmiin lukeutuvat muun muassa vesiputousmalli, prototyyppimalli ja spiraalimalli, kun taas ketteriin menetelmiin kuuluvat puolestaan Scrum, Lean, Extreme Programming (XP) sekä Kanban. (Huosianmaa, 2020, ss. 9–10; Tapio, 2010, s. 37) Scrum on ketteristä menetelmistä suosituin huomattavalla marginaalilla (Digital.ai, 2020, s. 10).

2.3.1 Scrum

Scrumin erottaa muista ketteristä menetelmistä muun muassa kielteinen suhtautuminen muutoksiin toteutusiteraatioiden aikana sekä ennalta määritetty roolitus. Kehittäjien lisäksi tiimiin kuuluu asiakasrajapinnasta ja kehitystyön priorisoinnista vastaava tuoteomistaja sekä työn organisoinnista vastaava Scrum Master. (Eby, 2017)

Viitekehyksen muutamia oleellisimpia käsitteitä ovat käyttäjätarina (user story), kehitysjono (backlog), EPIC, daily, MVP (minimum viable product), DoD (definition of done) sekä DoR (definition of ready).

- Käyttäjätarinalla tarkoitetaan kehityksessä olevan tuotteen jonkin toiminnon kuvaamista asiakkaan näkökulmasta. Käyttäjätarinassa on oleellista, että ilmi tulee käyttäjän tarve kyseiseen funktioon ja käyttäjän rooli tätä toimintoa käyttäessään. Käyttäjätarinoilla määritellään tuotteen vaatimukset.
- Kehitysjonolla eli backlogilla viitataan prioriteettilistaan, jolta poimintaan jaksoittain toteutettavaa.
- EPIC on toteutettavasta tuotteesta jaoteltu suuri kokonaisuus, jonka alle hierarkkisesti yksittäiset asiaan kuuluvat käyttäjätarinat lajitellaan.
- Daily on päivittäinen lyhyt kokous, jonka aikana käydään läpi työn alla olevia asioita.
- MVP tarkoittaa minimitavoitteen mukaista tuotetta eli tuotetta, jossa kaikki ehdottoman pakolliset käyttäjätarinat ovat valmiita kehityksen näkökulmasta.
- DoR on valmiin määritelmä määrittelyjen eli käyttäjätarinoiden kontekstissa.

- DoD on valmiin määritelmä kehityksen kontekstissa.

Lisäksi EPICeistä ja käyttäjätarinoista muodostuvaan hierarkiaan lisätään usein ominaisuuksia (feature) sekä tehtäviä (task). Ominaisuudet sijoittuvat käyttäjätarinoiden yläpuolelle ja tehtävät puolestaan pienimmäksi yksiköksi käyttäjätarinoiden alapuolelle. Viidentenä irrallisena osana ovat bugit, jotka voidaan linkittää käytetyssä projektinhallinnan työkalussa käyttäjätarinoihin. Kehitystyötä voidaan puolestaan jaksottaa Scrumissa 1–4 viikon pituisiin iteraatioihin, joita kutsutaan sprinteiksi. (Contribyte, n.d.)

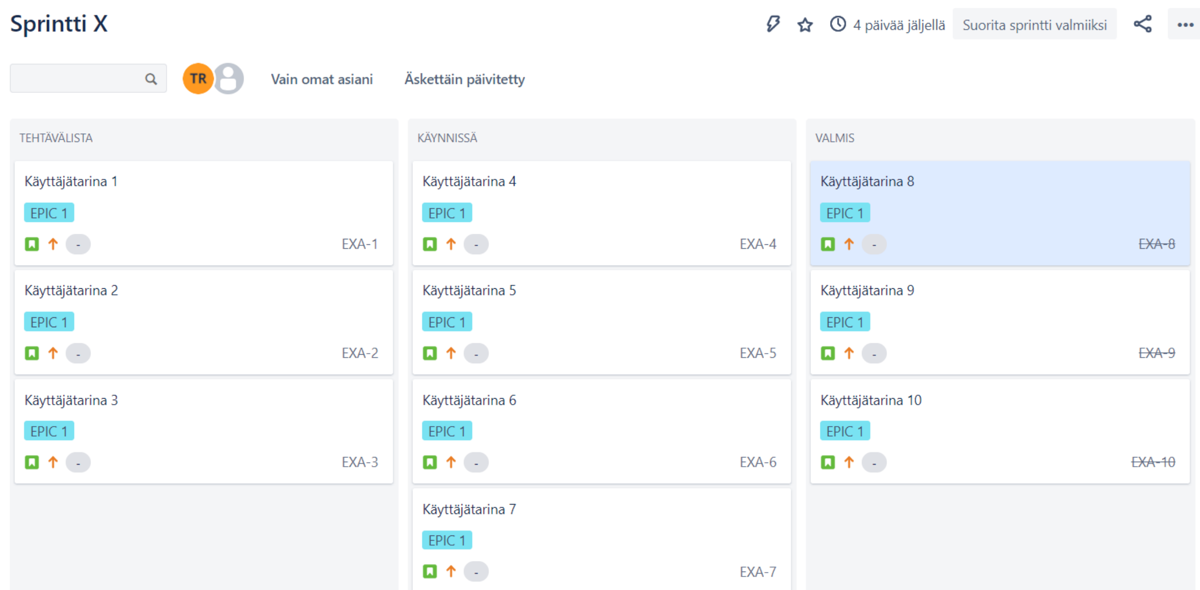
Menetelmää on kritisoitu varsinkin jäykkien roolirakenteiden vuoksi olevan kuitenkin jotain aivan muuta kuin ketterä. Kritiikkiin tyypillisesti vastataan, että suurin osa projektien epäonnistumisista johtuu viitekehyksen ala-arvoisesta soveltamisesta, eli ihmisistä, eikä itse Scrumista (Jeffries, 2018). Riippumatta siitä onko Scrum objektiivisesti ketterä vai ei, mitään projektia ei missään nimessä kannata sitoa täyttämään viitekehyksen tarpeita, vaan toimintamalleista pitää päinvastoin olla konkreettista hyötyä. Tehokkain tapa työskennellä on hyvin projekti- ja tiimikohtainen, joten valittuja menetelmiä kannattaa aina soveltaa kehittäjien ja asiakkaan tarpeet etusijalla. Hyvin itseohjautuvassa ja kokeneessa tiimissä voidaan esimerkiksi aloittaa Scrumilla ja projektin edetessä jalostaa käytäntöjä vaikkapa XP:n suuntaan. Mikään yksittäinen projektinhallinnan menetelmä ei ole valmiina pakettina ideaali kaikkiin projekteihin ja kokoonpanoihin. Mukautuvuus on erittäin tärkeä osa viitekehyksiä ja tämän vuoksi on olemassa myös useita hybridimalleja.

Työssä päädyttiin soveltamaan Scrumin käytäntöjä, koska viitekehys oli tullut työn ja opintojen kautta melko tutuksi ja opinnäytetyön toteutuksen jaksottaminen viikon mittaisiksi sprinteiksi vaikutti toteutuksen aikatauluun suhteutettuna järkevältä. Vaikka projekti toteutettiin yksin, jolloin suuri osa Scrumin oleellisimmista tekniikoista, kuten päivittäispalaverit ja katselmointi, eivät olleet tässä yhteydessä käyttökelpoisia, koin, että organisoidun työskentelyn lisäksi Scrumin käytäntöjen noudattaminen muodostaisi läpinäkyvää dokumentaatiota kehitysprosessista käyttäjätarinoiden ja niiden toteuttamisen myötä ja auttaisi siten myös itseäni hahmottamaan paremmin kehityksen tilannetta eri vaiheissa, kun käytössä on myös soveltuva projektinhallinnan ohjelmisto.

2.3.2 Jira

Jira on hyvin suosittu ketterän kehityksen eri viitekehyksiin soveltuva projektinhallinnan ohjelmisto. Se tarjoaa perustoiminnallisuuksina backlogin sekä sprinttien hallinnan ja seurannan, joihin lukeutuu kaikkien ominaisuuksien, tehtävien, bugien ja käyttäjätarinoiden elinkaari. Kuvassa 6 on esimerkki Jiran sprinttinäkymästä, jossa sprintille on poimittu muutamia käyttäjätarinoita toteutettavaksi. Esimerkin käyttäjätarinoilla on kaikilla työmäärä arvioimatta sekä oletusprioriteetti.

Kuva 6. Esimerkki Jiran sprinttinäkymästä.



Jira mahdollistaa myös projektin tilannetta kuvaavien raporttien luomisen sekä integraation erilliseen Confluence-alustaan, jossa voidaan lisätä vapaamuotoisia sivuja tiedon jakamiseen projektitiimin sisällä. Confluencen sivuilla voidaan Jira-integraation avulla näyttää esimerkiksi tiettyihin kokonaisuuksiin liittyvien käyttäjätarinoiden tila tai projektin bugitilanne muun materiaalin ohella.

Ohjelmisto on ollut jo useita vuosia suosituin projektinhallinnan työkalu ja Digital.ai:n (2020, ss. 16–17) vuosittaisen State of Agile -raportin mukaan Jira on edelleen suosituin ohjelmisto ketteriä menetelmiä hyödyntävissä projekteissa: kyselyyn vastanneista jopa 67 % vastasi, että he ovat käyttäneet Jiraa ja heistä 78 % suosittelisi sitä muille. Vertailukohtana

esimerkiksi Microsoft Project -ohjelmistoa oli käyttänyt vain 9 % vastanneista ja sitä suosittelisi muille vain 30 % ohjelmistoa käyttäneistä.

Tässä työssä hyödynnetään Jiraa pääasiassa käyttäjätarinoiden organisointiin ja toiminnallisuuksien toteuttamisen jaksottamiseen sekä seurantaan sprinteittäin. Asiaan palataan vielä tarkemmin luvussa 4.

2.4 Työpöytäkehityksen sovelluskehykset

Alustariippumattomien työpöytäsovellusten kehitykseen on olemassa lukuisia eri sovelluskehyksiä. Javalla tähän soveltuu muun muassa JavaFX, C#:lla Xamarin, Pythonilla Kivy ja C++:lla Qt (Wilcurt, n.d.). Koska työssä ensisijaiseksi kehityskieleksi valikoitui JavaScript, päädyttiin vertailemaan toteutusta varten ohjelmointikielen kahta suosituinta työpöytäsovellusten sovelluskehystä: Electronia sekä NW.js:ää.

Dokumentaatio on erityisen tärkeä osa sovelluskehyksen laatua ja käytettävyyttä. Mitä kattavampi ja paremmin ryhmitelty dokumentaatio on, sitä vähemmän aikaa kuluu tarvittujen toimintojen implementointiin. Ensisilmäyksellä molempien dokumentaatio oli loogisesti ryhmitelty ja toimiva, mutta Electronin dokumentaatiosta tuli kattavampi vaikutelma: toiminnallisuuksia ja selventäviä ohjeita oli yksinkertaisesti enemmän. Muun muassa sovelluspäivitysten osalta Electronissa on sisäänrakennettu ratkaisu, kun taas NW.js ei tällaista oletuksena tue. Electronin dokumentaatiossa oli myös paljon koodiesimerkkejä sekä videoita ja kuvia materiaalin tukena toisin kuin NW.js:n dokumentaatiossa.

Tarkemmassa tarkastelussa eroavaisuuksia löytyy NW.js:n eduksi esimerkiksi Legacy-järjestelmien tukemisen osalta, jos syystä tai toisesta toteutettavan sovelluksen tulee olla toimintakykyinen vaikkapa vanhassa Windows XP -käyttöjärjestelmässä. Kehitystä helpottamaan Electronille on puolestaan rakennettu yhteisön kehittämiä hyvin suosittuja eri teknologiapinoille soveltuvia projektipohjia, kuten Electron React Boilerplate, joiden avulla projektien alkuun saattaminen onnistuu hyvin ketterästi. Vastaavanlaisia suosittuja ja ajantasaisia eri teknologiapinojen projektipohjia ei NW.js:lle kirjoitushetkellä löydy.

Nopeaan suosio- ja aktiivisuusvertailuun on käytettävissä pintapuoleista metriikkaa, kuten projektin GitHub-tähtien lukumäärä ja sekä koodimuutosten (commit) lukumäärä. Electronin GitHub-sivulla on kirjoitushetkellä noin 90 000 tähteä, kun taas NW.js:llä on noin 40 000 tähteä. Koodimuutoksia Electron-projektilla on puolestaan noin 24 000 ja NW.js:llä 4000. Suositumpi ei todellakaan aina tarkoita parempaa, mutta mitä suurempi yhteisö, sitä todennäköisemmin omiin kysymyksiin löytää vastauksen jo toisen kysymänä. Tiukalla aikataululla on tärkeää minimoida riskit, ja suositummalla vaihtoehdolla riskit ovat tyypillisesti pienemmät. Tämän ja dokumentaatiovertailussa esiin nousseiden asioiden vuoksi työn toteutukseen valikoitui Electron, vaikka myös NW.js vaikutti varsin käyttökelpoiselta vaihtoehdolta.

2.4.1 Electron

Electron-sovelluskehys on monien hyvin suosittujen työpöytäsovellusten, kuten Atom, Discord, Microsoft Teams, Skype, Visual Studio Code, WhatsApp ja Yammer, takana (OpenJS Foundation, n.d.-2). Tästä listasta voi hyvin päätellä, että Electronilla on mahdollista toteuttaa hyvin käyttökelpoisia sovelluksia, mutta sovelluskehystä kohtaan on myös esitetty laajalti kritiikkiä. Electronia on kritisoitu muun muassa suorituskyvyn sekä muistin käytön osalta ja halusinkin tässä työssä tarkastella, vaaditaanko kohtalaiseen suorituskykyyn tänä päivänä erityistä aikaa vievää optimointia, vai onnistuuko kehitys suoraviivaisesti myös pienemmän skaalan projekteissa noudattamalla suositeltuja käytäntöjä.

Electron perustuu Node.js-ajoympäristöön sekä Chromiumiin, joka on suositun Chrome-selaimen ydin. Electronissa renderöinti tapahtuu Chromiumin sisällä, jolloin käyttöliittymätasolla voidaan hyödyntää samoja käytäntöjä ja koodirakennetta kuin web-sovelluksissa tai -sivuilla. Siinä missä perinteiset työpöytäsovellukset hyödyntävät käyttöjärjestelmälle jo oletuksena asennettuja tai muiden sovellusten mukana riippuvuutena tulleita ajoympäristöjä, tarvitaan hyvin pieniinkin Electron-sovelluksiin sisällyttää Chromium sekä Node.js, jolloin ohjelmakoodin ja käytettyjen kirjastojen osuus tiedostokoosta on melko marginaalinen. (Wikipedia, 2021f) Hyvin yksinkertainen sovellus saattaa tämän vuoksi olla optimoimattomana kooltaan jopa yli 100 megatavua, joka voi olla kymmeniä kertoja suurempi kuin perinteisillä teknologioilla toteutettu vastaavanlainen sovellus.

Electronissa on kahden tyyppisiä prosesseja: pääprosessi ja renderöintiprosessi. Pääprosessissa hallinnoidaan jokaista renderöintiprosessia ja pääprosessi on ainut prosessi, jolla on oletuksena pääsy Node.js-ajoympäristöön ja sen ohjelmointirajapintaan. Yksittäisessä renderöintiprosessissa puolestaan hallinnoidaan aktiivista verkkosivua. Renderöintiprosessi voi kommunikoida pääprosessin kanssa hyödyntämällä esimerkiksi niin kutsuttua IPC-rajapintaa. Tämä mahdollistaa muun muassa, että verkkosivulla olevan painikkeen avulla voidaan välittää pääprosessille pyyntö kutsua Node.js:n ohjelmointirajapinnan funktiota vaikkapa paikallisen ohjelman tai komennon suorittamiseen, mikä ei verkkosivuilla tai -sovelluksissa ole tyypillisesti mahdollista tietoturvallisuuden vuoksi. (OpenJS Foundation, n.d.-1)

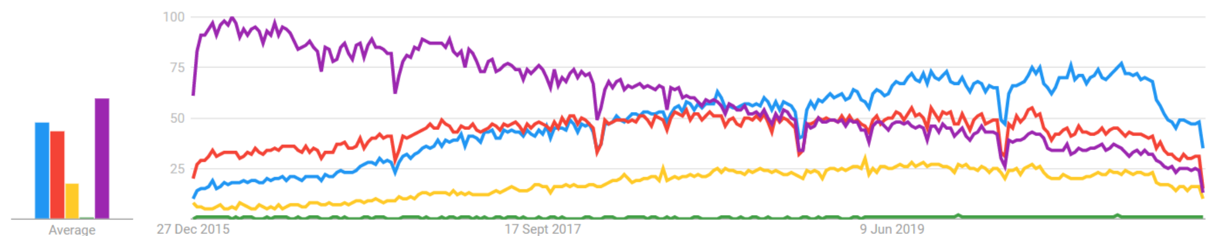
Koska Electron-sovellukset ovat pohjimmiltaan web-sovelluksia, liittyy näihin samoja tietoturvahaavoittuvuuksia kuin web-sovelluksiin. Jos sovelluksessa ladataan ulkoisia skriptejä joko suoraan ulkoisilta verkkosivuilta tai epäsuoraan käytettävien kirjastojen kautta, on potentiaalisen hyökkääjän mahdollista suorittaa komentoja renderöintiprosessissa. Tämän vuoksi renderöintiprosessin oikeudet on hyvä rajoittaa minimiin ja toteuttaa tarvittavat toiminnot ehdollisesti IPC-rajapinnan kautta. Noudattamalla tätä käytäntöä minimoidaan hyökkäysalaa ja mahdollista vahinkoa, jonka hyökkääjä voi potentiaalisesti aiheuttaa. Electronin nykyisessä versiossa on käytössä oletuksena tietoturva-asetuksia, jotka kannattaa ehdottomasti pyrkiä pitämään päällä, mikäli mahdollista. Myös Context Isolation -asetus on hyvä jättää päälle, jos sovelluksessa on tarve ladata ulkoista sisältöä, jotta mahdollinen pahansuopa skripti ei voi sovelluksessa muokata mahdollisesti käytettäviä globaaleja muuttujia sekä funktiota ja saa tällä tapaa lisää hyökkäysalaa. (OpenJS Foundation, n.d.-3)

2.5 Käyttöliittymäkehityksen sovelluskehykset

Suurempien verkkosovellusten ja dynaamisten verkkosivujen kehityksessä hyödynnetään lähes poikkeuksetta jotakin UI-sovelluskehystä tai JavaScript-kirjastoa. Suosituimmat tähän tarkoitukseen käytetyt teknologiat olivat vuonna 2020 React, Angular, Vue.js sekä Svelte.js (The Software House, 2020). Myös jQuery-kirjastoa on käytetty hyvin pitkään käyttöliittymäkehityksessä, mutta viime vuosina suorituskyvyltään parempien ja sovelluskehyksiksi sopivien kirjastojen kehitwritten on jQuery menettänyt suosiotaan (Kuva 7).

Tämän lisäksi ECMAScriptin – jota JavaScript myötäilee – päivitysten myötä useat jQueryn apumetodit eivät abstraktoi enää niin paljon logiikkaa taakseen kuin ennen, jolloin kirjastosta saatava hyöty ei enää konkretisoidu monille kehittäjille. Nykyään React on suosituin JavaScript-käyttöliittymäkehityksen sovelluskehys, jonka allekirjoittanut on todennut käytännössä erittäin toimivaksi.

Kuva 7. Google Trends -viivakaavio eri teknologioiden suosioon linkittyvistä hakumääristä 2015–2020. Purppura on jQuery, sininen React, punainen Angular, keltainen Vue.js ja vihreä Svelte.js.



2.5.1 React

React on Facebookin kehittämä ja ylläpitämä JavaScript-kirjasto. Se perustuu rakenteeseen, jossa dokumentin dynaamiset elementit kapseloidaan erillisiksi komponenteiksi. Komponentit voidaan tulkita esimerkiksi funktioina, joita korkeammalla tasolla oleva isäntäelementti kutsuu syöttämällä argumenteiksi komponentin senhetkiseen tilaan liittyviä ominaisuuksia (props). Näiden argumenttien lisäksi komponentilla voi olla omia tilamuuttujia niin kutsutussa state-objektissa. Kun joko jokin parametrina oleva ominaisuus tai komponentin sisäisen tilamuuttujan arvo muuttuu, renderöidään kyseinen komponentti uudestaan, jolloin tilamuutos konkretisoituu käyttöliittymällä. Komponentit noudattavat hierarkiarakennetta siten, että tavanomaisesti komponentit voivat vaikuttaa vain omaan ja suorassa yhteydessä olevien alemman tason komponenttien tilaan. Monipuolisempaan tilanhallintaan on kuitenkin olemassa sisäänrakennetun Context-ohjelmointirajapinnan lisäksi erillisiä kirjastoja, kuten Redux, joita hyödyntämällä voidaan implementoida sovellukseen globaali tilasäiliö. Tällä toteutuksella voidaan tarvittaessa hallitusti päivittää

minkä tahansa yksittäisen komponentin kautta globaalia tilaa ja siten myös minkä tahansa muun komponentin tilaa riippumatta komponenttien hierarkiarakenteesta. (Zammetti, 2020, ss. 55–59)

Reactissa on myös implementoitu suorituskyvyn vuoksi niin kutsuttu Virtual DOM (VDOM), jolla tarkoitetaan käytäntöä, jossa DOM:in (Document Object Model), eli dokumentin puurakenteen, nykyistä tilaa verrataan muutosten aiheuttamaan tavoitetilaan ja synkronoidaan sitten kaikki muutokset samalla kertaa todelliseen dokumenttiin. Ilman tämän implementointia todellisen DOM:in muuttuvat komponentit täytyisi renderöidä kokonaan uudelleen jokaisen yksittäisen muutoksen myötä, mikä on suorituskyvyllisesti kallis prosessi. (Zammetti, 2020, s. 47)

Muiden suosittujen käyttöliittymäkirjastojen tapaan Reactilla on useita käyttöliittymäkomponenttien sovelluskehyksiä, joiden avulla projekteihin saadaan ketterästi toteutettua yhtenäinen, toimiva ja dynaaminen käyttöliittymä. Nämä komponenttien sovelluskehykset noudattavat usein tietyn design-järjestelmän ohjenuoria, jotta kokonaisuudesta saadaan silmiä miellyttävä ja saavutettava. Yksi suosituimmista komponenttien sovelluskehyksistä on Material-UI, jota tässä työssä päätettiin käyttää toteutusvaiheessa. Material-UI perustuu Googlen kehittämään Material Design -järjestelmään (Material-UI, n.d.).

2.6 Pilvipalvelut

Pilvipalveluiksi luokitellaan muun muassa kaikki kolmannen osapuolen infrastruktuuri (IaaS), alustat (PaaS), ohjelmistot (SaaS) sekä pilvilaskentafunktiot (FaaS), joita voidaan hyödyntää internetin välityksellä. Pilvipalvelu voi olla siis esimerkiksi vapaasti konfiguroitava palvelin, tietovarasto, kehitystyökalu, BI-työkalu, web-sovellus tai yksittäiseen toimintoon jalostettu kutsuttava funktio. Palveluiden käyttöä laskutetaan ensisijaisesti käytön mukaan, oli käyttö sitten jatkuvaa tai epäsäännöllistä. Esimerkiksi virtuaalikoneita laskutetaan tyypillisesti tunneittain, kun taas pilvilaskentafunktioiden käyttöä laskutetaan suorittamiseen vaadittujen resurssien mukaan. Jos toteutettavalla palvelulla ei siis ole jatkuvaa käyttöä, tulee pilvilaskentafunktioihin perustuva ratkaisu tavanomaisesti huomattavasti edullisemmaksi kuin esimerkiksi virtuaalikoneen vuokraaminen. (Wikipedia, 2021c)

Suorituskyky on erittäin tärkeä osa-alue, kun halutaan rakentaa hyvin skaalautuvia järjestelmiä. Pilvipalveluita hyödyntämällä voidaan sovellustasolla implementoida esimerkiksi kuormantasaus ja välimuistin hyödyntäminen tietosäiliönä melko yksinkertaisesti sekä infrastruktuuritasolla ottaa tarvittaessa käyttöön enemmän palvelimia tai lisätä yksittäisen käytettävän palvelimen kapasiteettia. Käytettävien palvelimien lukumäärän nostamisesta käytetään näissä yhteyksissä usein termiä horisontaalinen skaalaus, kun taas yksittäisen palvelimen kapasiteetin kasvattamisesta vertikaalinen skaalaus. Pilvipalveluissa on usein myös ilman suurta lisäkonfiguraatiota mahdollisuus lisätä tilanteen, eli senhetkisen kuorman, mukaan käytössä olevaa kapasiteettia ja puolestaan tilanteen rauhoittuessa taas pudottaa kapasiteettia, jolloin asiakkaan ei tarvitse maksaa jatkuvasti käyttöpiikkien mukaisista resursseista. Dynaaminen skaalautuminen, josta puhutaan yleisemmin pilven elastisuutena, on lähes poikkeuksetta horisontaalista, koska yksittäisen palvelimen kapasiteetin kasvattaminen edellyttää palvelimen hetkellisen pysäyttämisen. (VMware, n.d.)

Varsinkin suosituimpien pilvipalveluita tarjoavien toimijoiden, kuten Amazon Web Services (AWS), Microsoft Azure ja Google Cloud Platform (GCP), Alibaba Cloud sekä IBM Cloud, kohdalla voi luottaa siihen, että hyödynnettävien palveluiden tietoturva on maailmanluokan tasolla. Nämä palveluntarjoajat noudattavat korkeimpien turvallisuusstandardien, kuten NIST 800-171 ja FIPS 140-2 käytäntöjä sekä varmentavat myös kolmannen osapuolen toimesta palveluidensa tietoturvallisuuden. Palveluntarjoajien on ehdottoman tärkeätä huolehtia tietoturvasta, koska yksikin hyväksikäytetty haavoittuvuus voisi viedä asiakkaiden luottamuksen pysyvästi ja näin ollen kaataa koko liiketoiminnan sekä ohjata asiakkaat kilpailijoille. Tämä ei tarkoita kuitenkaan, että käyttäjä voisi huolettomasti luottaa pilvipalveluiden avulla implementoitujen järjestelmien tai mikropalvelujen olevan hyökkäyksiltä suojattuna, vaan tietoturva on silti huomioitava kaikissa asiakkaan konfiguroitavissa olevissa osa-alueissa. Useimmat palveluntarjoajat tarjoavat pääsylista-, todennus-, auktorisointi-, kryptaus- ja lokituskerroksen lisäksi muita suojausominaisuuksia tavanomaisista verkkosovelluspalomuureista (WAF) DDoS-suojaukseen ja uhantunnistukseen. Asiakkaan vastuulle jää usein näiden käyttöönotto ja asianmukainen konfigurointi, kun taas palveluntarjoaja on vastuussa pilven sisäisestä infrastruktuurista sekä käytettyjen palveluiden virheettömästä toiminnasta. (Amazon Web Services, 2021; Google, 2021; Huang, 2019; Microsoft, 2020)

Pilvilaskentapalveluissa puolestaan on usein mahdollisuus integroida ja toteuttaa palveluiden käsittelysäännöt monilla eri ohjelmointikielillä. Kaikki suosituimmat pilvipalvelutarjoajat tukevat oleellisilta osin työssä käytettävää Node.js-teknologiaa, mutta tarvittaessa pilvilaskentafunktioita voitaisiin toteuttaa myös esimerkiksi Javalla, C#:lla, Pythonilla tai Golangilla. Tässä työssä pilvilaskentaa ei tarvita minimituotteen toteutukseen, mutta tulevaisuudessa, jos esimerkiksi erillinen tietokanta osoittautuisi tarpeelliseksi, voitaisiin toteuttaa kustomoituja HTTP-funktioita, jotka esimerkiksi päivittäisivät tietokantaa hallitusti sovelluksesta kutsuttaessa tai jonkin toisen konfiguroidun trigger-funktion suoriutuessa.

Pilvipalveluiden käyttäminen säästää valtavasti kehitysaikaa ja takaa, että järjestelmä on helposti skaalattavissa. Olen henkilökohtaisesti käyttänyt entuudestaan sekä AWS että Microsoft Azure -alustoja, ja päätin tämän ohjelmistojärjestelmän sidosjärjestelmän toteutukseen valita AWS:n, koska olin toisessa aiemmassa projektissa implementoinut ja integroinut sovelluksen käyttäjänhallinnan AWS:n Cognito-palvelulla ja tiesin S3-palvelun soveltuvan erinomaisesti toiseen tarvittavaan toiminnallisuuteen. Kehittämisaikaa kuluisi tämän vuoksi toimintojen toteuttamiseen todennäköisesti vähemmän kuin muiden alustojen vastaavilla palveluilla.

2.6.1 AWS

AWS on suosituimmista pilvipalvelualustoista pitkäikäisin ja tällä hetkellä myös käytetyin. AWS tarjoaa yli 175 palvelua, joiden käyttötarkoitukset vaihtelevat analytiikasta tietokantoihin, koneoppimiseen, laskentaan, sisällön jakeluun, infrastruktuuriin, tiedon säilytykseen ja lukuisiin muihin tarkoituksiin (Amazon Web Services, 2021b). Tämän työn kannalta oleellisimpia palveluita ovat S3 ja Cognito.

S3 (Simple Storage Service) on AWS:n tiedon säilytykseen jalostettu palvelu, joka mahdollistaa tiedostojen turvallisen ja luotettavan jakamisen. Palvelu tukee tavanomaisempien toimintojen, kuten varmuuskopioinnin ja toimintokohtaisten pääsylistojen, lisäksi muun muassa normaalia edullisempaa pitkäaikaista säilytystä datalle, johon ei ole tarvetta päästä käsiksi usein. (Amazon Web Services, 2021b)

Cognito-palvelu on puolestaan kehitetty vastaamaan sovellusten identiteetinhallinnan tarpeisiin. Cognito on integroitavissa lukuisten muiden AWS-palveluiden kanssa, jolloin autentikointi sekä käyttäjä- ja käyttäjäryhmäkohtaisten pääsyvaatimusten konfigurointi on toteutettavissa hyvin suoraviivaisesti. Cognitossa on huomioitu tietoturva perusteellisesti ja se läpäisee muun muassa ISO/IEC 27001 -tietoturvastandardin vaatimukset. Lisäksi palvelulla on muita käyttäjien tietoturvaa parantavia vapaavalintaisia toimintoja, kuten vuodettujen salasanojen seuranta, jonka avulla turvatonta vuodettua salasanaa käyttävä käyttäjä ohjataan vaihtamaan salasananansa. (Amazon Web Services, 2021a)

2.7 Muut työvälineet

Kehitys- ja suunnittelutyötä varten työssä oli tarpeellista käyttää muutamia muita työvälineitä. Kehitystyön osalta koodieditori tai integroitu kehitysympäristö (IDE) on käytännössä välttämättömyys, kun taas versionhallinta tekee uusien toimintojen implementoinnista hallitumpaa sekä dokumentoi kehitystä. Sovellusten käyttöliittymiä toteuttaessa on puolestaan hyvä myös ensin suunnitella ulkoasu suunnittelutyökalulla, jossa prototyyppien toteutus ja muokkaaminen onnistuu huomattavasti nopeammin kuin kehitysympäristössä. Aiemman hyvän kokemuksen perusteella työssä käytettäväksi koodieditoriksi valikoitui Visual Studio Code, versionhallintajärjestelmäksi Git ja käyttöliittymäsuunnittelun ohjelmistoksi Figma.

2.7.1 Visual Studio Code

Visual Studio Code on Microsoftin kehittämä avoimen lähdekoodin koodieditori, joka on tullut yhdeksi suosituimmista työkaluista kehittäjien keskuudessa. Tällä työkalulla on muiden editorien tapaan hyvin paljon aikaa ja vaivaa säästäviä korostus-, haku- ja korvaustoimintoja. Syntaksikorostus parantaa huomattavasti koodin luettavuutta ja ehkäisee alkeellisilta virheiltä, kun taas haku- ja korvaustoiminnot mahdollistavat ketterän navigoinnin ja debuggauksen sekä tehostavat massamuutosten tekemistä. Tuottavuuden näkökulmasta on ehdottoman tärkeää, että työkalu tukee myös säännöllisiä lausekkeita ja multikursoritilaa, jotta vältytään tarpeettomalta muokkaus- ja kirjoitustyöltä. Säännöllisten lausekkeiden avulla voidaan esimerkiksi rakentaa koodissa pohjaa jollekin säännöllisesti toistuvalle rakenteelle tai vaihtoehtoisesti muokata jotakin rakennetta toiseen muotoon nopeasti.

Muina sisäänrakennettuina ominaisuuksina Visual Studio Code mahdollistaa suoraviivaisen navigoinnin projektin eri tiedostojen välillä ja hyvin toimivan tiedostolinkityksen, jonka avulla hakutoiminnoilla voidaan esimerkiksi löytää ja avata tarkasteluun jonkin toisesta tiedostosta tuodun apufunktion määritelmä. Lisäksi laajojen ulkoasu- ja toiminnallisuuskonfiguraatioiden ohella Visual Studio Code -ohjelmistossa on suurempien integroiduiksi kehitysympäristöiksi lukeutuvien työkalujen tapaan terminaali komentojen ajamista varten sekä suoritus- ja debuggaustilat. (Microsoft, 2021b)

Hyvin toteutettujen sisäänrakennettujen ominaisuuksien lisäksi editoriin on mahdollista asentaa lisäosia, joilla editorin ominaisuuksia voidaan laajentaa esimerkiksi tietyn ohjelmointikielen osalta. Lisäosilla sovellus voidaan saada toteuttamaan esimerkiksi automaattitäydennyksiä, koodivalidointia tai formatointia. Nämä lisätoiminnot voivat säästää huomattavasti kehitysaikaa niin varsinaisen ohjelmakoodin kirjoittamisen kuin myös bugien varhaisen löytymisen osalta.

Edellä mainittujen kehitystyötä helpottavien ja nopeuttavien toimintojen lisäksi Visual Studio Code -editorissa on sisäänrakennettuna myös Git-integraatio, joka mahdollistaa sujuvan versionhallinnan siirtyessä projektista toiseen.

2.7.2 Git

Git on alun perin Linus Torvaldsin suunnittelema ja jälkikäteen pääasiassa Junio Hamanon kehittämä hajautettu versionhallintajärjestelmä, joka on useita vuosia ollut sovelluskehittäjien päätyökalu versionhallintaan. Lukuisat verkon yli toimivat ohjelmavarastoja tarjoavat suositut palvelut, kuten GitHub, GitLab ja Bitbucket, pohjautuvat nimenomaan Gitiin. Järjestelmän kasvanutta suosiota selittänee näiden verkkopalveluiden, avoimen lähdekoodin ja järjestelmän hajautetun periaatteen lisäksi osaltaan Microsoftin virallinen siirtymä vuonna 2017 käyttämään Gitiä omaan versionhallintaansa sekä myöhemmin 2018 toteutunut yrityskauppa, jossa Microsoft osti GitHubin 7,5 miljardilla dollarilla. (Favell, 2020)

Vaikka Gitiä hyödynnetäänkin nykyään ylivoimaisesti eniten, on versionhallintaan olemassa muitakin vaihtoehtoja, kuten esimerkiksi SVN (Apache Subversion). Jos projektilla on

esimerkiksi tarvetta versioida koodin lisäksi hyvin suuria binääritiedostoja, saattaa keskitetty SVN olla joissakin tapauksissa soveltuvampi vaihtoehto. (Perforce Software, 2018)

2.7.3 Figma

Mitä tulee käyttöliittymäsuunnitteluun, Figma on noussut viime vuosina Adobe XD:n ja Sketchin ohi suosituimmaksi suunnitteluohjelmistoksi (UX Tools, 2020). Yksi oleellisimmista syistä työkalun valtavaan suosioon on ohjelmiston monialustaisuus, jolloin kollaboraatio onnistuu tiimin sisällä ja tiimien välillä saumattomasti riippumatta kenenkään projektiryhmäläisen käytössä olevasta käyttöjärjestelmästä (Kopf, 2018).

Kilpailevien työkalujen tapaan Figma tukee myös lisäosia ja jaettuja komponenttikirjastoja, joiden avulla prototyyppien suunnittelu onnistuu sujuvasti. Koska ohjelmistolla on suuri käyttäjäkunta, löytää vastauksen omiin käyttöä liittyviin kysymyksiin usein välittömästi. Figma soveltuu tavanomaisten suunnittelutarpeiden lisäksi erinomaisesti myös vektorigrafiikkaeditointiin, mutta tässä työssä ohjelmistoa hyödynnetään lähinnä sovelluksen ulkoasun prototyyppien toteutuksessa.

3 Kehittämistyön tavoite ja tarkoitus

Työn tavoitteena oli toteuttaa aiemmin kuvaillun kaltainen työpöytäsovellus (Kuva 1) sekä skriptien ja skriptiketjujen lataamiseen ja hakemiseen soveltuva sidosjärjestelmä. Päädyin rajaamaan MVP:n pelkästään paikallisesti toimivaan työpöytäsovellukseen ilman integraatiota sidosjärjestelmän kanssa. Mikäli saisin toteutettua paikallisiin toimintoihin vaadittavat komponentit nopeasti, toteuttaisin vielä sidosjärjestelmän konfiguraation, työpöytäsovelluksen integraatiokutsut sekä käyttöliittymälle vaadittavat näkymät.

Toteutettavan ohjelmistojärjestelmän määrittelyt:

- Alustariippumaton työpöytäsovellus (Linux, macOS ja Windows)
- Moderni käyttöliittymätoteutus
- Avatun skriptin parametrien jäsennys käyttöliittymälle: tuki MVP:ssä Python, PowerShell ja Node.js -skriptien yleisimmille parametrien syntakseille
- Paikallisten skriptien ja skriptiketjujen haku käyttöliittymällä
- Skriptiketjujen muodostaminen ja suorittaminen annetuilla argumenteilla
- Skriptien ohittaminen skriptiketjulla (tilapäinen valinta)
- Skriptien sisääntulon ja tulosteen hallinta: edellisten suoritettujen skriptien tulosteita voidaan käyttää seuraavien skriptien argumentteina
- Viimeisimmän skriptiketjun säilyttäminen istuntojen yli
- Kontrollirakenteet
    - Odottaminen ketjun skriptien suoritusten välillä
    - Suorittaminen vain tietyn ehdon täyttyessä
    - Saman skriptin suorittaminen silmukassa useampaan kertaan
- Asetukset
    - Pikanäppäimet tallennettujen skriptiketjujen suorittamiselle
    - Skriptikansio
- Ei MVP:ssä
    - Sidosjärjestelmä ja sen konfigurointi
        - Käyttäjähallinta
        - Skriptit ja skriptiketjut pilvessä
    - Sidosjärjestelmän integraatio

    - Rajapintakutsut
    - Lisäkonfigurointi esimerkiksi tietoturvan osalta
  - Käyttöliittymätoteutus sidosjärjestelmän osalta
    - Rekisteröityminen
    - Kirjautuminen
    - Sidosjärjestelmän skriptien ja skriptiketjujen haku- ja lähetysnäkymät
  - Tuki useammalle skriptauskielelle ja parametrien syntakseille parametrien jäsennyksen osalta
  - Kevyen Node.js-kirjaston toteuttaminen tallennettujen skriptiketjujen ajamiseen ilman käyttöliittymällistä työpöytäsovellusta
  - Tarkemmat käyttöohjeet ja muu oheismateriaali

Ohjelmisto suunniteltiin parantamaan kaikenlaisen toimistotyön tuottavuutta. Kohderyhmänä on tässä tapauksessa kaikki tehokäyttäjät, jotka haluavat melko usein suorittaa jotakin toistuvaa tehtävää. Ilman sidosjärjestelmätoteutusta kohderyhmään kuuluu lähinnä jo jonkin verran skriptauskokemusta omaavat henkilöt, kun taas sidosjärjestelmätoteutuksen kanssa kokemusta ei välttämättä tarvitse olla ollenkaan, kun tarvittavat skriptit voidaan etsiä ja ladata avainsanoilla sekä ketjuttaa suoraviivaisesti käyttöliittymällä.

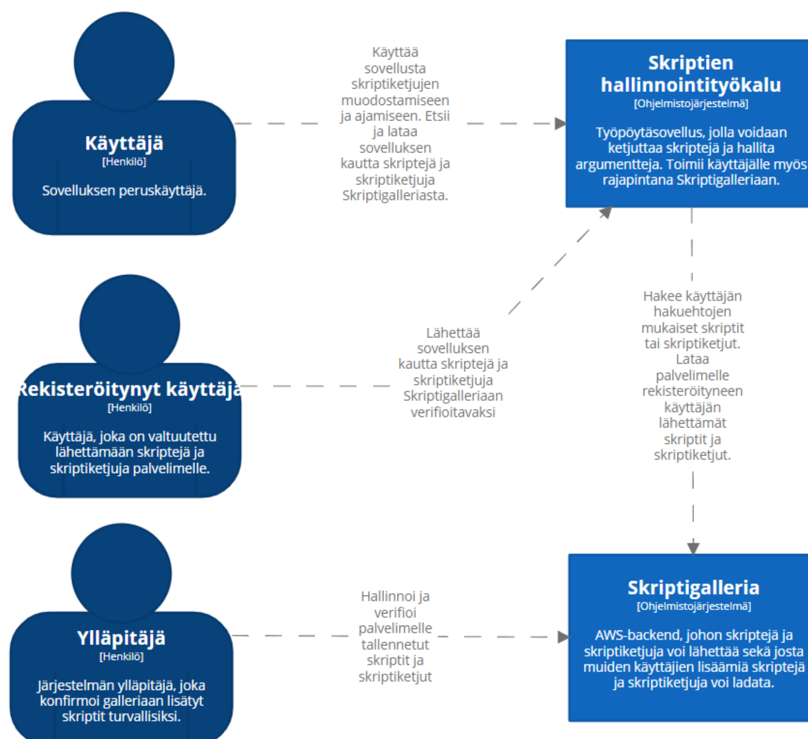
4 Tuotteen suunnittelu ja toteutus

Tuotteen suunnittelu ja toteutus jaoteltiin kahteen suunnilleen yhtä suureen aikaväliin. Ajatuksena oli, että perinpohjainen määrittely, suunnittelu ja perehtyminen käytettäviin teknologioihin säästäisi kehitysaikaa huomattavasti, kun kehitysvaiheessa voisi keskittyä lähes täysin kehitykseen.

4.1 Suunnitteluprosessi

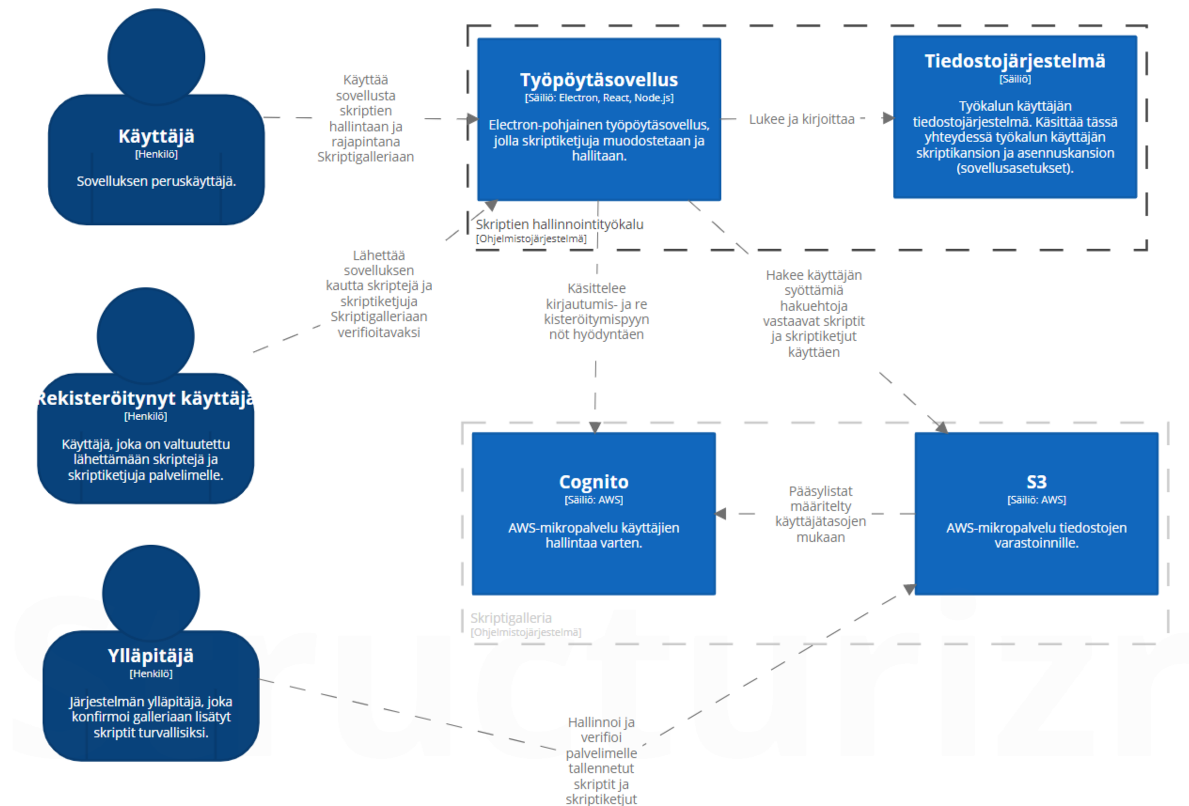
Aiemmin listattujen tuotteen määrittelyiden kirjoittamisen jälkeen projektin varsinainen suunnittelu aloitettiin jäsentämällä työn oleelliset komponentit ja mallintamalla muutamia havainnollistavia kaaviokuvia toteutettavasta kokonaisuudesta. Kuvassa 8 näkyy korkean abstraktion järjestelmäkaavio, jossa on kuvailtu käyttäjäsuhteet ja jaoteltu toteutettava kokonaisuus skriptien hallinnointityökaluun ja sidosjärjestelmään, jonka nimesin Skriptigalleriaksi.

Kuva 8. Järjestelmäkaavio toteutettavasta kokonaisuudesta.

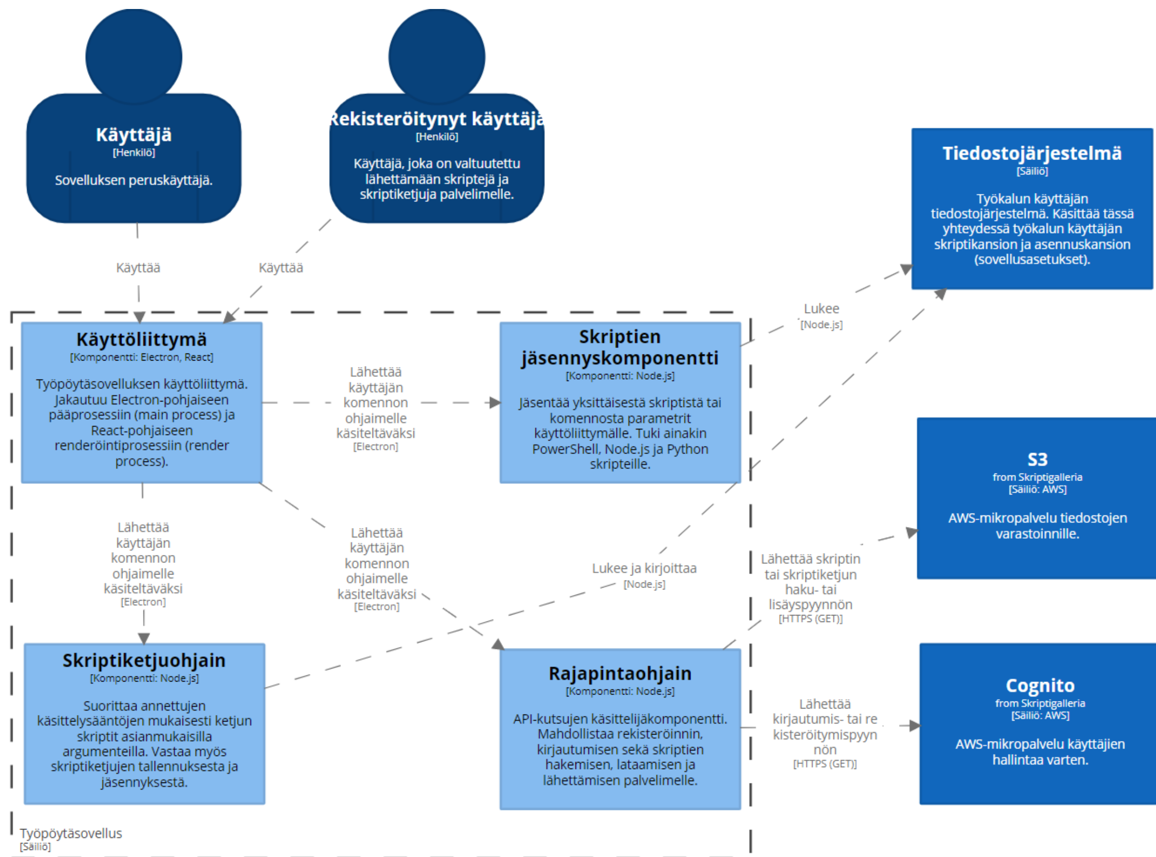


Toteutin seuraavaksi C4-mallin mukaisesti vielä kaksi hieman tarkemman tason näkymää, jotka havainnollistavat kumpaankin järjestelmään kuuluvia osa-alueita sekä komponentteja. Kuvassa 9 on toisen tason säiliönäkymä ja Kuvassa 10 kolmannen tason komponenttinäkymä työpöytäsovelluksen osalta.

Kuva 9. Järjestelmien yhdistetty säiliönäkymä.



Kuva 10. Työpöytäsovelluksen komponenttinäkymä.



Mallinnuksen jälkeen edettiin kirjoittamaan määrittelyitä Jiraan tarkemmiksi käyttäjätarinoiksi, jotta myöhemmin toteutusvaiheessa toimintojen toteuttaminen voitaisiin jaksottaa poimimalla niitä vastaavat käyttäjätarinat sprinteille toteutettavaksi. Tässä vaiheessa käyttäjätarinoita muodostui ennen tarkennuksia kolmisenkymmentä (Kuva 11). Projektien edetessä käyttäjätarinat usein tarkentuvat ja joitakin uusia joudutaan myös kirjoittamaan, joten kuvassa näkyvät käyttäjätarinat eivät ihan täysin vastaa viimeisen sprintin jälkeistä tilannetta.

Kuva 11. Jira-backlog suunnitteluvaiheen päätteeksi.

Backlog

Only My Issues   Recently Updated

EPICS   Create epic

All issues

Skriptien hallinnointityökalu
SMT-1 Skriptien hallinnointityökalu

Issues	30
Completed	1
Unestimated	29
Estimate	0

Create issue in epic
View linked pages

Skriptigalleria
SMT-2 Skriptigalleria

Issues	2
Completed	0
Unestimated	2
Estimate	0

Create issue in epic
View linked pages

Issues without epics

Backlog 31 issues

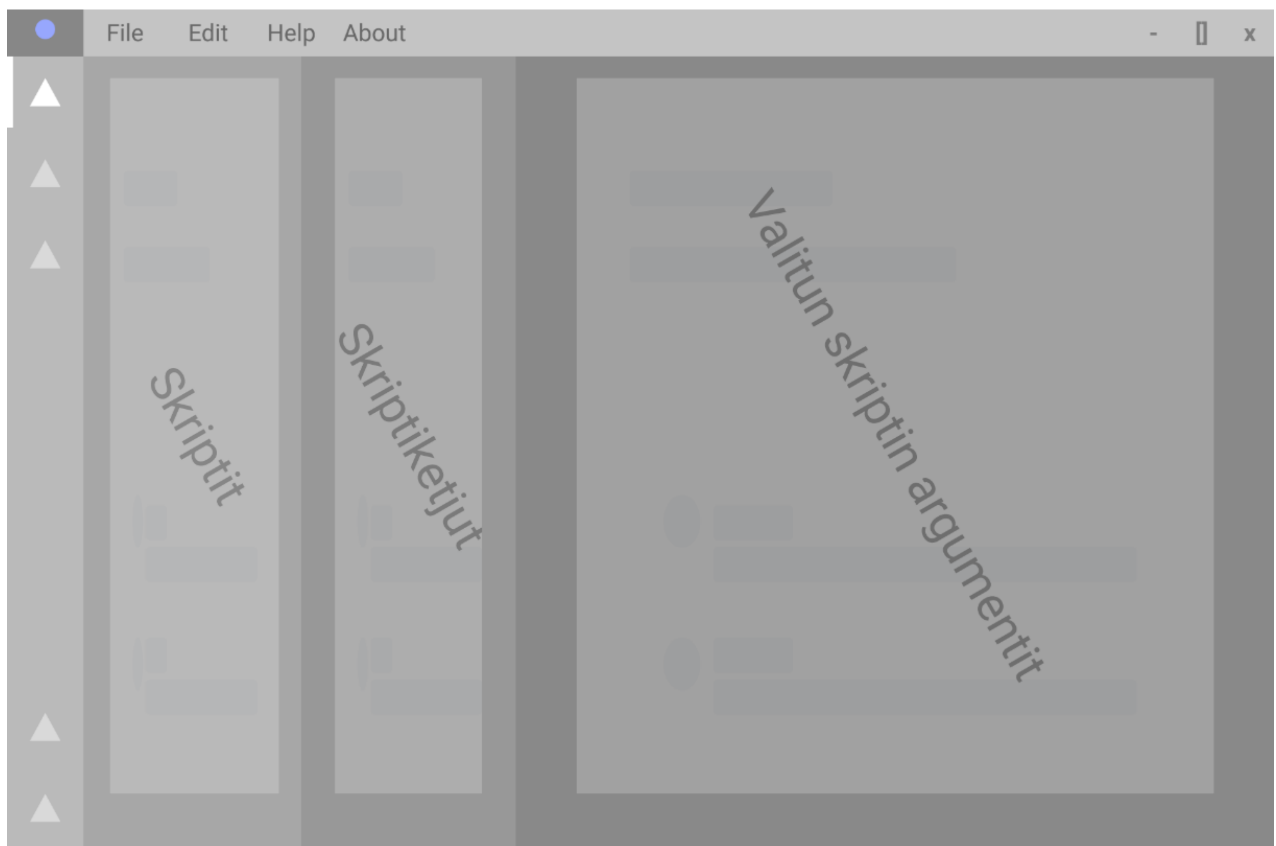
Issue	Epic	Key
Skriptikansion hallinta	Skriptien hallinnointity...	SMT-4
Käyttöliittymä: Valikkopalkki	Skriptien hallinnointity...	SMT-5
Käyttöliittymä: Sivupalkki	Skriptien hallinnointity...	SMT-6
Käyttöliittymä: Skriptinäkymä	Skriptien hallinnointity...	SMT-7
Käyttöliittymä: Gallerianäkymä	Skriptien hallinnointity...	SMT-8
Käyttöliittymä: Käyttäjänäkymä	Skriptien hallinnointity...	SMT-9
Käyttöliittymä: Asetusnäkymä	Skriptien hallinnointity...	SMT-10
Skriptin parametrien jäsentäminen	Skriptien hallinnointity...	SMT-11
Skriptin viimeisimpien argumenttien tallentaminen	Skriptien hallinnointity...	SMT-12
Skriptiketjun muodostaminen	Skriptien hallinnointity...	SMT-13
Skriptiketjun tallentaminen	Skriptien hallinnointity...	SMT-14
Skriptien hakeminen	Skriptien hallinnointity...	SMT-15
Skriptiketjujen hakeminen	Skriptien hallinnointity...	SMT-16
Skriptiketjun nimeäminen	Skriptien hallinnointity...	SMT-17
Skriptiketjun tyhjentäminen	Skriptien hallinnointity...	SMT-18
Skriptiketjun poistaminen	Skriptien hallinnointity...	SMT-19
Skriptikansion sisällön uudelleenlukeminen (päivittäminen)	Skriptien hallinnointity...	SMT-20
Skriptin lisääminen skriptiketjulle	Skriptien hallinnointity...	SMT-21
Skriptin poistaminen skriptiketjulta	Skriptien hallinnointity...	SMT-22
Skriptin asettaminen tilapäisesti pois käytöstä	Skriptien hallinnointity...	SMT-23
Skriptiketjun suorittaminen	Skriptien hallinnointity...	SMT-24
Skriptiketjun yksittäisen skriptin suorittaminen annetuilla argumenteilla	Skriptien hallinnointity...	SMT-25
Skriptien ulostulon hallinta	Skriptien hallinnointity...	SMT-26
Skriptien sisääntulon hallinta	Skriptien hallinnointity...	SMT-27
S3: Asennus ja konfigurointi	Skriptigalleria	SMT-28
Cognito: Asennus ja konfigurointi	Skriptigalleria	SMT-29
Skriptin tallentaminen S3-säiliöön rekisteröityneenä käyttäjänä	Skriptien hallinnointity...	SMT-30
Skriptiketjun tallentaminen S3-säiliöön rekisteröityneenä käyttäjänä	Skriptien hallinnointity...	SMT-31
Rekisteröityminen	Skriptien hallinnointity...	SMT-32
Kirjautuminen	Skriptien hallinnointity...	SMT-33
Käyttäjäasetukset	Skriptien hallinnointity...	SMT-34

+ Create issue

Ennen toteutuksen aloittamista toteutettiin vielä hieman alustavaa käyttöliittymäsuunnittelua mallintamalla työpöytäsovelluksen käyttöliittymän ulkoasun prototyyppi (Kuva 12). Päätin alkuvaiheessa ottaa aika paljon vaikutteita Visual Studio Code -koodieditorin käyttöliittymästä ja palata myöhemmin mukauttamaan ulkoasua, mikäli muulta toteutukselta jäisi aikaa. Oleellisimpia asioita käyttöliittymän käytettävyyden näkökulmasta on värien kontrasti ja toimintojen ryhmittäminen. Mitä pidempään käyttäjän

katse viipyy tietyssä osiossa ja mitä pienempää kyseisen alueen fontti- tai elementtikoko on, sitä suurempi kontrasti elementin värillä ja sen taustavärillä pitää olla. Jos kontrasti on hyvin matala, saattaa joidenkin ihmisten olla mahdotonta lukea tekstiä tai erottaa elementtejä taustasta. Käyttöliittymän toteutuksessa on pyritty mahdollisimman tuttuun navigaatioon ja painikeasetteluun, jotta sovelluksen toiminnot löytyvät tavanomaisesti sieltä, mistä henkilö, joka ei ole sovellusta ennen käyttänyt, olettaakin niiden löytyvän. Toimintopainikkeiden ja näkymien jaottelun ansiosta käyttäjälle ei tule hukuttavaa ensivaikutelmaa, kun interaktiivisia elementtejä ei ole oletusnäkymässä liikaa.

Kuva 12. Käyttöliittymän ulkoasun prototyyppi (Figma).



4.2 Toteutus

Toteutus aloitettiin järjestelmällisesti poimimalla viikoittain backlogilta käyttäjätarinoita sprintille. Tässä luvussa toteutusprosessi on kuvattu kokonaisuudessaan kronologisessa järjestyksessä sprinteittäin, ja toimintojen toteuttamisen kannalta kriittisimmät tiedot ja menetelmät käsitellään samassa yhteydessä.

4.2.1 Sprintti 1

Ensimmäisellä sprintillä työhön oli käytettävissä työtunteja merkittävästi enemmän henkilökohtaisen tilanteen vuoksi, joten sprintille valikoitui toteutettavaksi ohjelmiston selkärangan muodostavat suurimmat toiminnot. Tällä menettelyllä saisin myös tarkennettua konkreettista arviota työn toteuttamiseen kuluvasta ajasta, jotta voisin tarvittaessa hyvissä ajoin valmistautua lisätunteihin tai projektin venymiseen. Sprintille valikoitui toteutettavaksi seuraavat käyttäjätarinat:

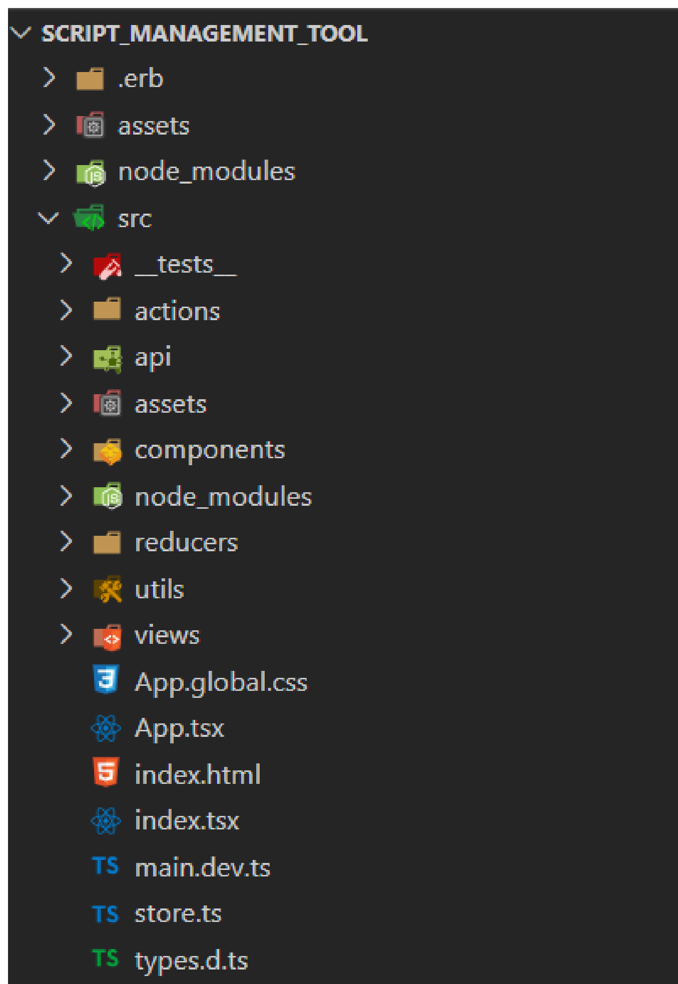
- Käyttöliittymä: sivupalkki
- Käyttöliittymä: skriptinäkymä
- Skriptikansion hallinta
- Skriptikansion sisällön jäsentäminen
- Skriptiketjun muodostaminen
- Skriptin lisääminen skriptiketjulle
- Skriptin poistaminen skriptiketjulta
- Skriptiketjun suorittaminen
- Skriptien sisääntulon hallinta
- Skriptien ulostulon hallinta

Päädyin projektipohjan luomiseen hyödyntämään Electron React Boilerplate -pohjaa. Pohjan mukana tulee hyvin laajalti web-sovelluskehityksessä käytetyt Babel ja Webpack, jotka ovat valmiiksi konfiguroitu soveltumaan TypeScript-kehitykseen React-projekteissa. Babel muun muassa mahdollistaa uusimpien ECMAScript-toimintojen käyttämisen koodissa kääntämällä kaikki uutta syntaksia käyttävä koodi matalamman tason JavaScriptiksi, jota vanhempien selainten tulkit ymmärtävät. Tällä on erityisesti merkitystä web-sovelluksilla, jossa käyttäjillä saattaa olla käytössä useita eri selaimia vuosien takaa, mutta myös Electron-kehityksessä käytettävä Chromium ei ole vakaiksi merkityissä versioissa tuorein, jolloin sillä ei ole myöskään tukea kaikille uusimmille toiminnoille. Webpack puolestaan pakkausvaiheessa pakkaa käytetyn koodin yksittäiseksi moduuliksi ja eliminoi samalla käyttämättömiä koodiosioita niin kirjoitetusta koodista kuin sovelluksessa hyödynnettävistä kirjastoista. Tällä ei ole työpöytäsovelluksissa yhtä paljon merkitystä kuin web-sovelluksissa, koska ohjelmakoodin ja kirjastojen osuus koko sovelluksesta on marginaalinen ja koodia ladataan

lokaalisti. Jos tarkoituksena on kuitenkin käyttää samaa koodikantaa yhden sovelluksen web- sekä työpöytävariaatioissa, konkretisoituu hyöty latausnopeuksissa. Pohjan mukana tulee myös käyttökelpoinen ESLint-konfiguraatio, jota käytetään kirjoitetun koodin tarkistuksessa.

Projektipohjaan asennettiin seuraavaksi tilanhallintaan varten Redux-kirjasto ja käyttöliittymätoteutukseen Material-UI-komponenttikirjasto. Jaottelua varten tilanhallinnalle ja käyttöliittymäkomponenteille lisättiin erilliset hakemistot, jolloin projektirakenne näytti konfiguraatio- ja seurantatiedostot pois lukien Kuvan 13 mukaiselta.

Kuva 13. Projektin hakemistorakenne ilman konfiguraatio- ja seurantatiedostoja.



- .erb-hakemisto käsittää käytetyn pohjan tarkistusskriptit ja konfiguraatiotiedostot kehitykseen sekä pakkaukseen.
- assets-hakemistoihin lajitellaan staattiset resurssit. Projektin juuressa sijaitsevaan assets-hakemistoon lisätään kääntämisen aikana tarvittavat tiedostot, kuten

sovelluskuvake, kun taas alemman tason hakemiston alle lajitellaan puolestaan suorituksen aikana tarvittavat resurssit, kuten kuvat, fontit ja konfiguraatiotiedostot.

- node_modules-hakemistoihin tallentuu asennetut kirjastot ja niiden riippuvuudet. Käytetyssä pohjassa kaikki paitsi natiivin riippuvuuden sisältävät kirjastot asennetaan juuressa.
- src-hakemisto sisältää sovelluksen lähdetiedostot.
- __tests__ on Jest-kirjaston testien oletushakemisto, johon lisätään komponenttikohtaisia testejä.
- actions-hakemisto sisältää Redux-kirjaston tilamuutoskutsut.
- api-hakemistossa on rajapintakutsufunktiot.
- components-hakemisto käsittää kustomoidut käyttöliittymäkomponentit.
- reducers-hakemistossa on Redux-kirjaston tilamuutoslogiikka kutsuittain.
- utils-hakemistoon on lajiteltu tiettyihin kokonaisuuksiin liittyvät apufunktiot.
- views-hakemisto käsittää sovelluksen uniikit näkymät. Yksittäisellä näkymällä voi olla dynaamisesti päivittyvää sisältöä, mutta tyypillisesti näkymällä on ainakin jokin staattinen elementti, kuten valikkopalkki.

Sovelluksen lähtöpisteenä on kehityksessä main.dev.ts-tiedosto, joka sisältää sovelluksen pääprosessin toimintalogiikan. Koodissa muun muassa konfiguroidaan sovellusikkuna sekä lisätään kaikki IPC-tapahtumankuuntelijat ja niitä vastaavat komennot, jotta renderöintiprosessista voidaan turvallisesti ja hallitusti lähettää pääprosessille komentoja suoritettavaksi. Pääprosessissa avattu selainnäkymäikkuna ohjataan puolestaan avaamaan index.html-tiedosto, joka sisältää referenssin mahdollisiin CSS-tyylitiedostoihin sekä koodimoduuliin, jonka puolestaan Webpack on koontivaiheessa muodostanut käymällä läpi sille konfiguraatioissa määritellyn lähtöpisteen (index.tsx) jokaisen riippuvuuden rekursiivisesti. Indeksissä kaikki sovelluksen näkymät sisältävä App-komponentti asetetaan Redux-säiliötä tarjoavan Provider-komponentin sisälle (Koodiesimerkki 1), jotta kaikki App-komponentin näkymien alikomponentit voivat tarvittaessa lukea ja päivittää jaettua tilaa.

Koodiesimerkki 1. Renderöintiprosessin lähtöpiste (index.tsx).

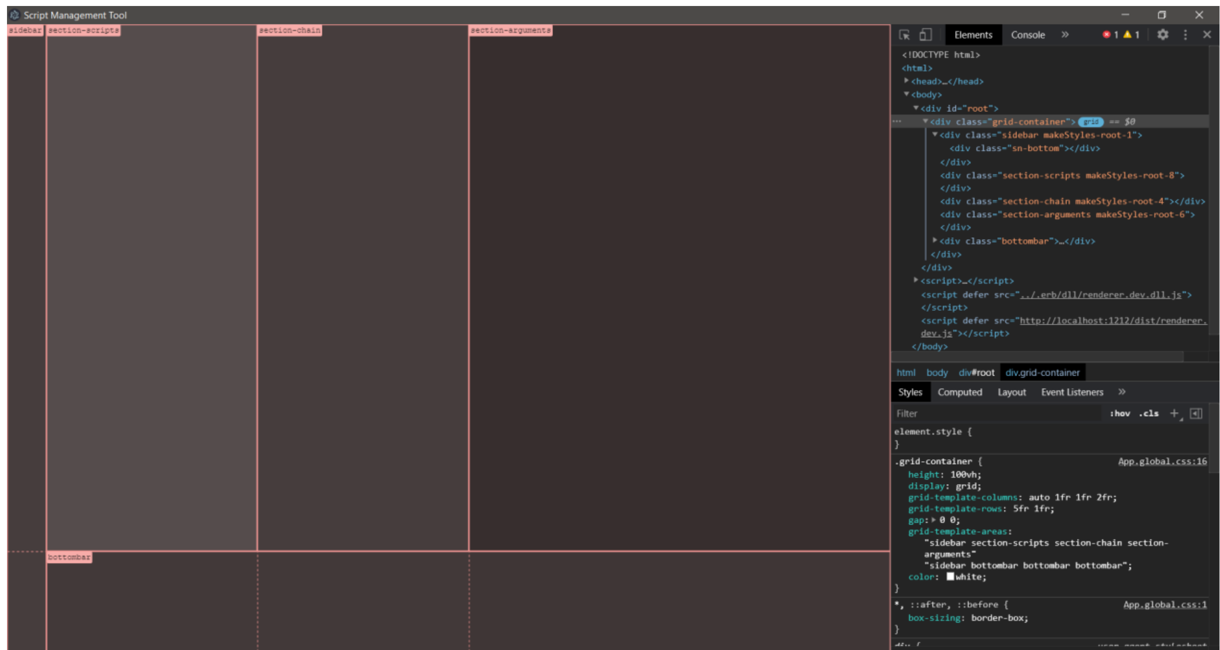
```
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './store';

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

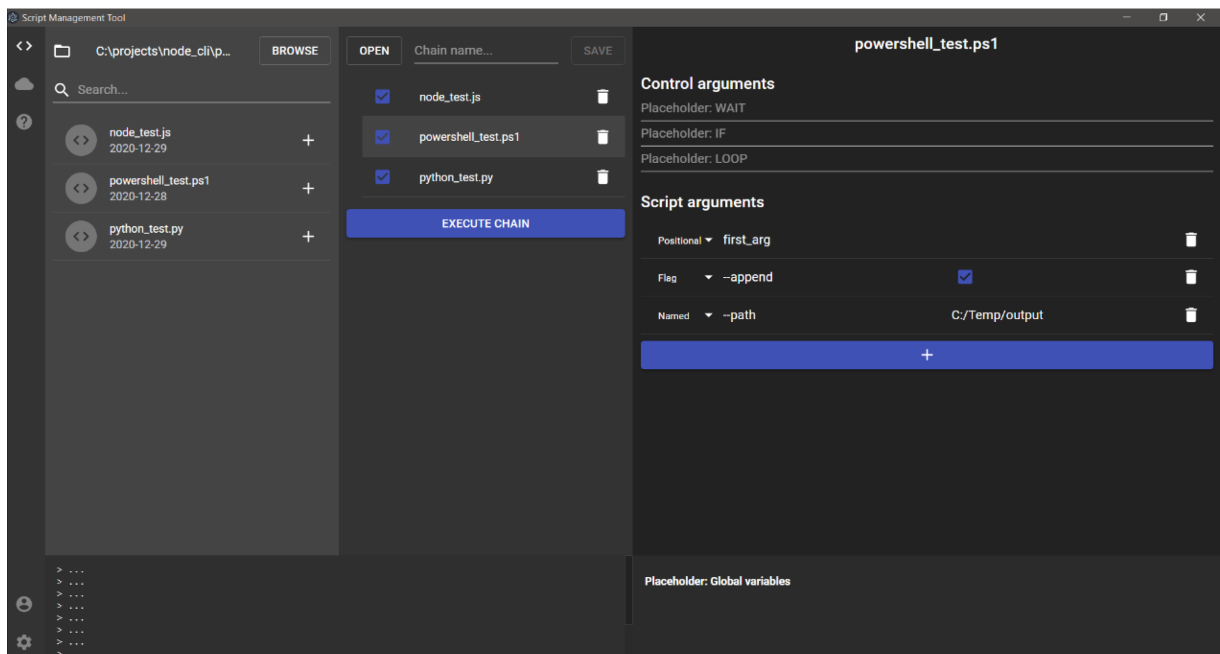
Sprintillä oli tarkoituksena toteuttaa myös päänäkymän käyttöliittymä. Tyypillisesti, kun web-kehityksessä tavoitteena on implementoida yksiulotteinen asettelu (layout), jolloin useita elementtejä on tietyssä osiossa joko vain yhdellä rivillä tai vaihtoehtoisesti yhdellä sarakkeella, hyödynnetään kustomointia vaativissa tilanteissa usein CSS Flexible Box Layout -mallia (flexbox). Moniulotteisessa asettelussa käytetään taas laajalti CSS Grid Layout -mallia. Toteutin prototyyppiä vastaavan asettelun CSS Grid -tekniikalla ja lisäsin vielä toteutukseen prototyypistä puuttuvan alapalkin, jossa olisi tarkoituksena näyttää tietoja skriptiketjun suorituksesta sekä mahdollisista virheistä (Kuva 14). Muokkasin myös värejä suunnitteluvaiheen prototyyppiin nähden hieman tummemmiksi.

Toteutettujen osioelementtien alle lisättiin tämän jälkeen yksitellen kaikki käyttöliittymäelementit yksiulotteisen yhden sarakkeen flexbox-säiliöelementtien sisään (Kuva 15). Tällä tavalla toteutettuna kaikki yhden flexbox-säiliön sisälle asetetut elementit voidaan sijoittaa niin, että mahdollinen tyhjä tila asetetaan tarvittaessa joko alkuun, loppuun, elementtien välille tai elementtien ympärille eri variaatioilla. Esimerkiksi sivupalkin tapauksessa tyhjä tila asetetaan kahden wrapper-elementin välille. Sivupalkin ensimmäinen wrapper-elementti sisältää kolme tilapainiketta, kun taas jälkimmäinen elementti käsittää kaksi erillisen dialogi-ikkunan avaavaa painiketta.

Kuva 14. Sovelluksen oletusnäkymän asettelu toteutettiin CSS Grid -tekniikalla.



Kuva 15. Ruudukon yksittäisten osioiden sisäisten elementtien asettelu toteutettiin pääosin flexbox-tekniikalla.



4.2.2 Sprintti 2

Toiselle sprintille päätin priorisoida seuraavat Jiran käyttäjätarinat:

- Skriptiketjun nimeäminen
- Skriptiketjun tallentaminen
- Skriptiketjun avaaminen
- Skriptien lokaali suodatus
- Viimeisimmän skriptiketjun säilyttäminen istuntojen yli
- Yksittäisen skriptin viimeisimpien argumenttien säilyttäminen istuntojen yli
- Yksittäisen skriptin asettaminen tilapäisesti pois käytöstä
- Skriptiketjun suorittamisen keskeyttäminen
- Skriptiketjun suorituksen vaiheiden lokitus
- Skriptiketjun suorituksen tilan esittäminen käyttöliittymällä

Koska kaikki sovelluksen skriptien polut ja argumentit tallennetaan jäsennetyssä objektirakenteessa Redux-säiliöön, skriptiketjun tallentaminen JSON-muotoon oli erittäin helppo ja looginen ratkaisu. Tiedostojärjestelmän prosessointiin käytettiin Node.js:n sisäänrakennettua fs-moduulia, kun taas tiedoston avaamiseen ja tallennukseen tarvittavan tiedostopolun valintaan soveltui Electronin dialog-moduuli, jossa on valmiina tarkistuksia muun muassa olemattomien polkujen osalta. Tallennusprosessi kokonaisuudessaan toimii seuraavasti:

1. Käyttäjä painaa käyttöliittymältä tallennuspainiketta.

2. Pääprosessille lähetetään komentona tallennuskutsu antamalla sille argumenteiksi Redux-säiliöstä haetut skriptit sekä käyttöliittymällä mahdollisesti syötetyn ketjun nimi oletusnimeksi.

3. Käsitellään pääprosessissa renderöintiprosessilta IPC-kanavaa pitkin tullut komento ja avataan Electronin ohjelmointirajapintaa hyödyntäen tallennusdialogi, jossa käyttäjä valitsee polun ja mahdollisesti muokkaa tiedostonimeä.

4. Jos käyttäjä valitsee polun, otetaan kutsusta skriptiketjun skriptien tiedot ja jäsennetään ne JSON-formaattiin sekä tallennetaan tiedosto valittuun polkuun. Muussa tapauksessa käsitellään peruuntunut valinta.

5. Palautetaan renderöintiprosessille tieto lopputuloksesta, jotta käyttöliittymälle voidaan päivittää esimerkiksi vaihtunut skriptiketjun nimi.

IPC-kommunikointi on esitetty renderöintiprosessin osalta Koodiesimerkissä 2 ja pääprosessin osalta Koodiesimerkissä 3.

Koodiesimerkki 2. Skriptiketjun tallennuksen käsittely renderöintiprosessissa.

```
ipcRenderer.send('saveChain', {
  name: chainName,
  scripts,
});
```

Koodiesimerkki 3. Pääprosessin skriptiketjun tallennuksen IPC-tapahtumankuuntelijan määritykset.

```
ipcMain.on('saveChain', async (event, chain) => {
  const pathObject = await dialog.showSaveDialog(mainWindow as BrowserWindow,
      {
    title: 'Save script chain',
    defaultPath: chain.name,
  });
  if (!pathObject.canceled && pathObject.filePath) {
    try {
      const res = await saveFile(
        pathObject.filePath,
        JSON.stringify({
          scripts: chain.scripts,
        })
      );
      event.reply('saveChain', {
        success: true,
        message: res,
        path: pathObject.filePath,
        name: path.parse(pathObject.filePath).name,
      });
    } catch (error) {
      event.reply('saveChain', {
        success: false,
        message: error.message,
```

```
        path: pathObject.filePath,
        name: path.parse(pathObject.filePath).name,
      });
    }
  }
});
```

Myös avausprosessi on hyvin samankaltainen sillä erotuksella, että pääprosessille ei tarvitse antaa argumentteina mitään ja mahdolliset suoritustilat resetoidaan käyttöliittymällä uuden skriptiketjun avaamisen yhteydessä. Renderöintiprosessissa määritellään omat IPC-tapahtumankuuntelijat, jotta käyttöliittymä voidaan päivittää vastaamaan suoritetun toiminnon tulosta (Koodiesimerkki 4).

Koodiesimerkki 4. Pelkistetty skriptiketjun avaamisen käsittely renderöintiprosessissa tässä vaiheessa toteutusta.

```
useEffect(() => {
  ipcRenderer.on('openChain', (_event, result) => {
    if (result.success) {
      localStorage.setItem('chainName', result.name);
      setChainName(result.name);
      setExecutionResults([]);
      dispatch(setScripts(result.message.scripts || []));
    }
  });
  return () => {
    ipcRenderer.removeAllListeners('openChain');
  };
}, [dispatch]);
```

Skriptien lokaaliin suodatukseen hakutuloksista päädyin käyttämään säännöllisiä lausekkeita (regex). Hakukenttään syötetään tällöin siis säännöllinen lauseke eikä suoraa osaa tiedostonimestä, kuten tavanomaisessa haussa. Tämä toimii monissa tilanteissa identtisesti normaalin haun kanssa, mutta erikoismerkit tulkitaan hieman poikkeavasti. Ratkaisuna tämä ei ole kokemattomille käyttäjille käyttäjäystävällisin, mutta tehokäyttäjille se tarjoaa normaalia hakua huomattavasti monipuolisemmat mahdollisuudet, koska hakuun voi sisällyttää suoraan ehdollisia termejä tai tiettyjä merkkijonoja poislukevia osioita. Säännöllisen lausekkeen käyttäminen oletushaussa vaatii kuitenkin virheenkäsittelyä, jotta epäkelvot säännölliset lausekkeet eivät rikkoisi sovellusta. Jos säännöllinen lauseke ei ole

kelvollinen, toteutetaan regex-haun sijaan normaali haku vertailemalla hakuehtoa tiedostojen nimiin sellaisenaan.

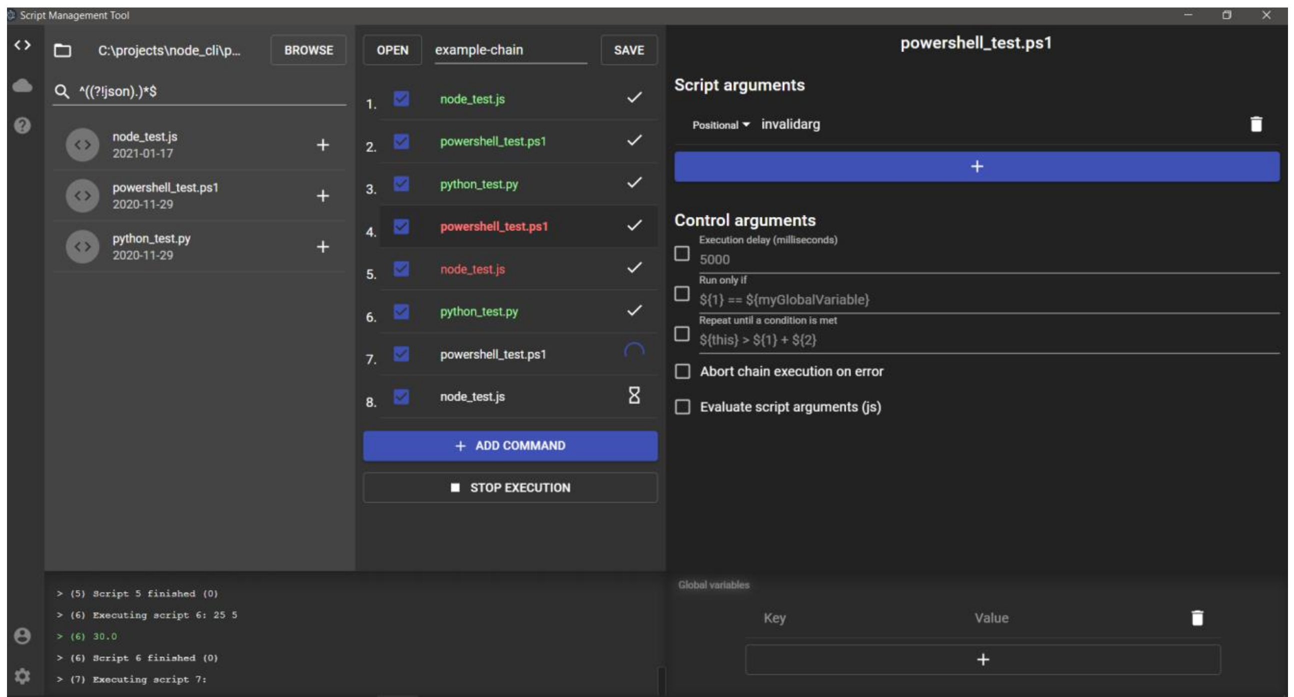
Tietojen säilyttämiseen istuntojen yli päädyin käyttämään selainten tukemaa Web Storage -ohjelmointirajapintaa. Oleelliset tilatiedot tallennetaan tilamuutosten yhteydessä localStorage-objektiin, josta ne voidaan hakea seuraavan käynnistyksen ensirenderöinnin yhteydessä takaisin käytettäväksi.

Viimeisten käyttäjätarinoiden osalta lisäsin skriptiobjektille erillisen boolean-tyyppisen muuttujan ja skriptiketjun suorittamisesta vastaavalle komponentille tarkistuksen tästä, jotta yksittäisen skriptin suorittaminen voidaan tarvittaessa ohittaa ja siirtyä ketjun seuraavan skriptiin. Skriptiketjun suorittamisen keskeyttäminen puolestaan oli helpoiten toteutettavissa tallentamalla referenssi nykyisestä skriptiprosessista suorituksen ulkopuolelle, jotta pääprosessista olisi prosessi mahdollista tarvittaessa keskeyttää. Keskeytykselle lisättiin myös skriptin lopetuksen tapahtumankäsittelijään käsittelysäännöt, jotta tieto keskeytyksestä saadaan välitettyä myös renderöintiprosessille.

Lokitus toteutettiin syöttämällä renderöintiprosessille lähetettäviin IPC-viesteihin jokaisen suoritetun skriptin konsolista tulevan datan lisäksi jäsennysominaisuuksina viestin tyypin ja linkityksen asiaankuuluvaan skriptiin. Tällöin renderöintiprosessissa voidaan toteuttaa jäsennystietoja hyödyntäen visuaalisia muotoiluja vastaanotetuille suoritustiedoille. Myös ketjun suoritustilat käyttöliittymällä toteutettiin pitkälti samalla jäsennysperiaatteella. Skriptien suorituksen elinkaaren tapahtumankäsittelijöihin lisättiin myös IPC-kutsut renderöintiprosessille, jotta käyttöliittymälle voitaisiin päivittää skriptiketjua suorittaessa jokaisen yksittäisen skriptin tila – oli se sitten valmis, keskeytynyt tai virheeseen ajautunut.

Kuvassa 16 näkyy sovelluksen tila sprintin 2 päättyessä. Toteutin sprintin alussa poimittujen käyttäjätarinoiden lisäksi myös hieman alustavaa käyttöliittymätoteutusta tuleville toiminnoille.

Kuva 16. Sovelluksen tilanne sprintin 2 päätteeksi: skriptiketjun suoritus ja lokitus käyttöliittymällä.



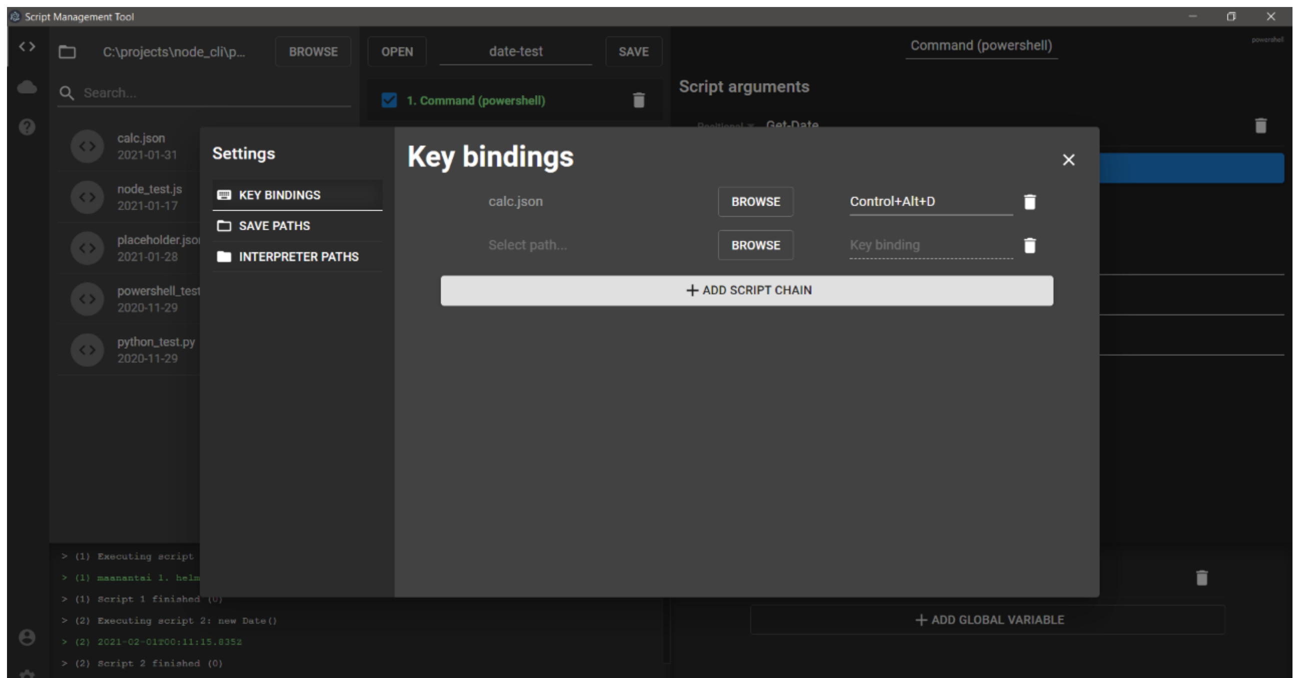
4.2.3 Sprintti 3

Kolmannelle sprintille toteutettavaksi otettiin käyttäjäasetusten, komentojen ja kontrolliargumenttien perustoimintojen käyttäjätarinat:

- Käyttäjäasetukset: käyttöliittymätoteutus
- Skriptiketjujen avaaminen ja suorittaminen pikanäppäimillä
- Kontrolliargumentit: odottaminen ennen suoritusta
- Kontrolliargumentit: suoritusehto
- Kontrolliargumentit: koko ketjun keskeyttäminen virheessä
- Globaalit muuttujat
- Komentojen lisääminen skriptiketjulle
- Komentojen suorittaminen

Käyttäjäasetuksien implementointiin soveltui hyvin Material-UI-kirjaston Dialog-komponentti. Aikataulun vuoksi en lähtenyt käyttämään tähän osioon enempää aikaa, joten lopputulos oli hyvin minimalistinen (Kuva 17).

Kuva 17. Käyttäjäasetukset.



Asetusten osalta toteutettiin tässä vaiheessa skriptiketjun suorittaminen pikanäppäimellä. Tämä mahdollistaa skriptiketjujen suorittamisen myös silloin, kun sovellus ei ole aktiivinen. Rekisteröinti ja suoritus tapahtuu seuraavasti:

1. Käyttäjä valitsee haluamansa skriptiketjun ja syöttää sille näppäinyhdistelmän erillisessä dialogissa avautuvaan syöttökenttään.

2. Renderöintiprosessista lähtee kutsu pääprosessille rekisteröidä ketjun suorittaminen annettuun näppäinyhdistelmään.

3. Pääprosessi rekisteröi haluttuun näppäinyhdistelmään takaisinkutsufunktion, jossa skriptiketjutiedosto avataan ja jäsennetty sisältö lähetetään renderöintiprosessille ylimääräisellä välittömän suorittamisen ohjausargumentilla.

4. Renderöintiprosessi päivittää tiedot vastaamaan uutta ladattua ketjua ja ohjausargumentin mukaisesti lähettää takaisin pääprosessille suorituspyynnön, jonka jälkeen suoritus etenee tavanomaisesti.

Implementaatiotasolla pikanäppäinten rekisteröintiin käytettiin Electronin globalShortcut-moduulia, jonka ohjelmointirajapintaa varten käyttäjän syöte prosessoidaan käyttöliittymältä kelvolliseen muotoon ennen pääprosessille lähettämistä.

Globaalit muuttujat implementoitiin käsittelemällä tilat käyttöliittymää varten Redux-kirjastolla ja tallentamalla ne myöhempää käyttöä varten skriptiketjutiedostoon sekä istuntojen yli Web Storage -säiliöön. Jotta ennen suoritusta annetun representaation, eli käyttäjän syöttämän ”${avain}”-muotoisen syötteen, sijasta käytettäisiin hajautustaulun arvoa, täytyi suoritukseen lisätä hieman vastaavaa logiikkaa kuin aiemmin suoritettujen skriptien tulosteen käyttämiselle argumenttina. Representaatio ajetaan käytännössä säännöllistä lauseketta vasten, jonka täsmätessä haetaan globaalien muuttujien hajautustaulusta avainta vastaava arvo, mikäli sellainen on luotu. Jos argumenttisyötteestä ei tunnisteta säännöllisellä lausekkeella näitä representaatioita, käsitellään käyttäjän syöte suoritusvaiheessa sellaisenaan.

Kontrolliargumentit päätin puolestaan sisällyttää muiden argumenttien tapaan skriptien tietorakenteeseen (Koodiesimerkki 5). Suoritusvaiheessa käynnistettävältä skriptiltä luetaan asianmukaisessa kohdassa näiden kontrolliargumenttien arvot ja toteutetaan tarpeelliset tarkistukset ja käsittelyt. Myös komentojen eli käskyjen, joita varten ei ole erillistä tiedostoa, lisäämisen ja suorittamisen käsittelyä varten muokkasin hieman tietorakennetta lisäämällä Script-objektille erillisen jaottelevan tyyppikentän.

Koodiesimerkki 5. TypeScript-määrittelyt päivitetylle tietorakenteelle.

```
export interface Script {
  name: string;
  path: string;
  type: 'file' | 'inline';
  enabled: string;
  uuid: string;
  args: Arg[];
  controlArgs: ControlArgs;
  notes: string;
}

export interface Arg {
  type: 'positional' | 'flag' | 'named';
  key: string;
  value: string;
```

```
  uuid: string;
}

export interface ControlArgs {
  delay: { enabled: boolean; value: string };
  executionCondition: { enabled: boolean; value: string };
  repeatCondition: { enabled: boolean; value: string };
  abortOnError: { enabled: boolean };
  evaluateArgs: { enabled: boolean };
  combineOutput: { enabled: boolean };
  parseJSON: { enabled: boolean };
}

export interface GlobalVariable {
  key: string;
  value: string;
  uuid: string;
}

export interface Log {
  message: string;
  type: 'info' | 'data' | 'error' | 'warning';
  scriptId: number | null;
  uuid: string;
}
```

Lisättyä kenttää hyödyntämällä komennot käsitellään varsinaisessa ohjelmakoodissa hieman eri tavalla kuin tiedostollisia skriptejä: komentoon muun muassa lisätään asianmukaisen tulkin vaatima komentovalinta. Esimerkiksi Node.js-tulkki vaatii komentojen suorittamiseen ”-e”- tai ”-p”-valinnan, kun taas Python ja PowerShell käyttävät ”-c”-valintaa. Puolestaan Bash tai Windowsin komentokehote (cmd) eivät luonnostaan näitä valintoja ymmärrä, joten valintojen lisääminen käyttäjän syöttämiin komentoihin täytyy käsitellä ohjelmakoodissa suoritusympäristöittäin.

4.2.4 Sprintti 4

Toiseksi viimeiselle sprintille jäi toteutettavaksi skriptigallerian palvelinpuolen konfigurointi, integrointi sekä selainpuolen käyttöliittymäosuus. Toteutukseen otettiin myös ilmoitukset sekä skriptiketjun skriptien uudelleenjärjestäminen. Sprintin käyttäjätarinat:

- S3: asennus ja konfigurointi
- Cognito: asennus ja konfigurointi

- Käyttöliittymä: gallerianäkymä
- Tiedostojen hakeminen galleriasta
- Tiedostojen lähettäminen galleriaan
- Omien tiedostojen poistaminen galleriasta
- Kirjautuminen
- Rekisteröityminen
- Salasanan nollaus
- Skriptiketjun skriptien uudelleenjärjestäminen
- Ilmoitukset
- Äänet
- Teemat: tumma ja vaalea

Gallerian toteutus aloitettiin luomalla S3-palveluun säiliö, joka konfiguroitiin estämään kaikki tunnistautumattomat yhteydet. Päädyin kansiorakenteen osalta ratkaisuun, jossa kaikki skriptit löytyvät public-kansion alta, jossa on erikseen vielä jaoteltu verifioidut skriptit omaan kansioon ja kaikki verifioimattomat skriptit tunnistautuneen käyttäjän henkilökohtaiseen kansioon, joilla vain asianomaisella käyttäjällä on kirjoitusoikeudet. Tällä tavalla toteutettuna tiedostojen laatujaottelu ja käyttöoikeuskonfiguraatio on toteutettavissa helposti, kun tiedoston omistaja ja tiedoston luonne ovat pääteltävissä suoraan tiedostopolusta.

Käyttöoikeuskonfiguroinnin osalta aluksi luotiin Cognito-palvelussa käyttäjäryhmä (User Pool), johon yhdistettiin App client -sovellustunniste. Tätä tunnistetta käytetään sovelluksessa autentikointiin liittyvissä rajapintakyselyissä, jotta tunnistautumaton käyttäjä voi sovelluksessa esimerkiksi rekisteröityä, kirjautua tai toteuttaa muita operaatioita, jotka eivät vaadi tunnistautumista. Samassa yhteydessä määriteltiin käyttäjän pakolliset tiedot, salasanamääritykset ja uuden käyttäjän verifioinnin käytännöt. Cognito tukee myös monivaiheista tunnistautumista sekä toimintokohtaisia trigger-funktioita, joiden avulla esimerkiksi rekisteröitymisen tai kirjautumisen yhteydessä voidaan suorittaa jokin haluttu toiminto, mutta tässä vaiheessa ennen tietokantaintegraatiota ei näille ollut tarvetta.

Jotta käyttöoikeuksia voitaisiin määritellä tarkemmin AWS-palveluiden välillä, luotiin myös identiteettiryhmä, jossa määritellään roolit käyttäjille ja linkitetään tunnistautuneen käyttäjän rooli aiemmin luotuun käyttäjäryhmään sovellustunnisteen ja

käyttäjäryhmätunnisteen avulla (Kuva 18). Kuvassa näkyvää identiteettiryhmätunnistetta (Identity pool ID) käytetään myöhemmin sovelluksen rajapintakutsuissa, jotta pyynnöt voidaan yhdistää luotuun identiteettiryhmään ja siten myös ryhmässä oleviin rooleihin ja niiden oikeuksiin.

Kuva 18. Identiteettiryhmän määrittelyjä (Federated Identities).

Identity pool name* smt identity pool

Identity pool ID eu-north-1:9356cf25-15b0-4f37-86bc-2d64d428828e (Show ARN)

Unauthenticated role Cognito_smtidentitypoolUnauth_Role Create new role

Authenticated role Cognito_smtidentitypoolAuth_Role Create new role

Unauthenticated identities

Authentication flow settings

Authentication providers

Amazon Cognito supports the following authentication methods with Amazon Cognito Sign-In or any public provider. If you allow your users to authenticate using any of t application ID that your identity pool is linked to will prevent existing users from authenticating using Amazon Cognito. Learn more about public identity providers.

Cognito Amazon Apple Facebook Google+ Twitter / Digits OpenID SAML Custom

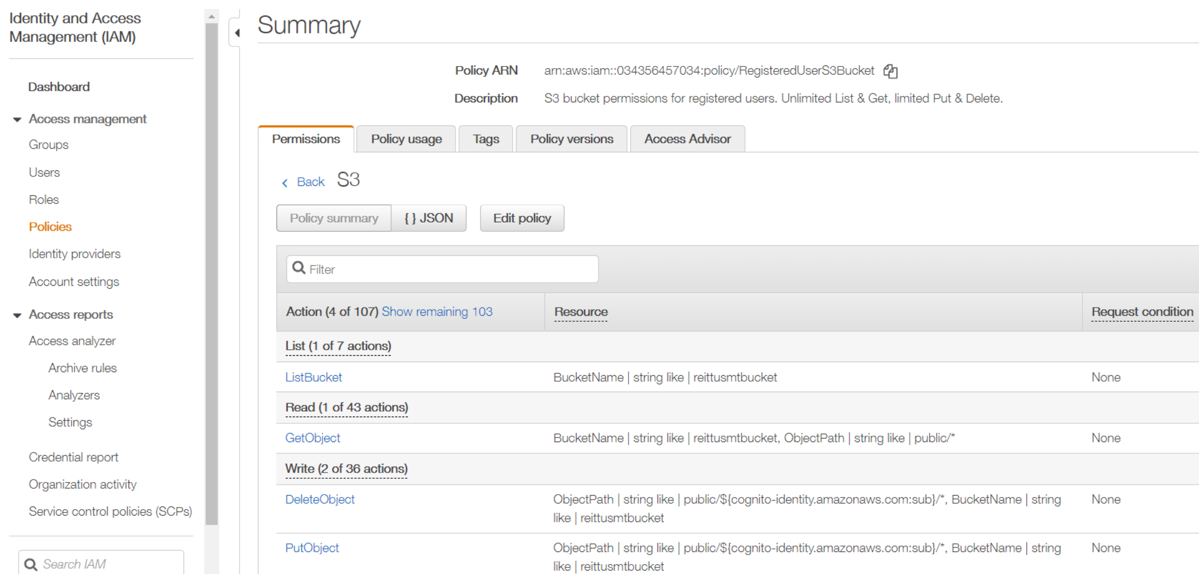
Configure your Cognito Identity Pool to accept users federated with your Cognito User Pool by supplying the User Pool ID and the App Client ID.

User Pool ID eu-north-1_ILXV8BIri Unlock

App client id 391qfndfoupek19vgpq7o5nit0 Unlock

Jotta identiteeteistä olisi hyötyä, täytyy myös määritellä niihin linkitettyjen roolien oikeudet tarvittaviin muihin AWS-palveluihin. Yksi suoraviivainen tapa toteuttaa oikeusmäärittelyt on hyödyntää rooleihin liitettäviä linjauksia (policies). Linjauksissa määritellään mitä toimintoja voidaan kutsua minkäkin resurssin yhteydessä. Kuvassa 19 näkyy toimintokohtaisesti mihin resursseihin rekisteröityneellä käyttäjällä on luku- ja kirjoitusoikeudet. S3-palvelun DeleteObject ja PutObject -toimintojen resurssimääritysten kohdilla näkyvällä "${cognito-identity.amazonaws.com:sub}" -muuttujalla viitataan käyttäjäkohtaiseen uniikkiin identiteettitunnisteeseen. Tällä tavalla rajataan rekisteröityneen käyttäjän kirjoitusoikeudet vain kyseisen käyttäjän oman kansionsa alta löytyviin tiedostoihin.

Kuva 19. Rekisteröityneen käyttäjän roolille linkitetty S3-palvelun linjaus.



Sovelluksen ja AWS-palveluiden integraatiossa olennaisessa osassa oli Amplify-kirjasto, joka abstraktoi suuren osan AWS-palveluiden rajapintakutsujen logiikasta. Asiakaspuolella tarvitsee määritellä vain käytettävien palveluiden tunnistetiedot (Koodiesimerkki 6) ja tämän jälkeen palveluiden käyttäminen onnistuu yksinkertaisten asynkronisten metodien avulla (Koodiesimerkki 7). Esimerkissä käytetään Amplify-kirjaston Storage-moduulin List-metodia, jolla tässä tapauksessa haetaan kaikki tiedostot S3-säiliön juuresta alkaen, sillä argumentiksi on annettu tyhjä merkkijono.

Koodiesimerkki 6. Amplify-määrittelyt (index.tsx).

```
import Amplify from 'aws-amplify';

Amplify.configure({
  Auth: {
    mandatorySignIn: false,
    identityPoolId: 'eu-north-1:9356cf25-15b0-4f37-86bc-2d64d428828e',
    region: 'eu-north-1',
    userPoolId: 'eu-north-1_ILXV8Blri',
    userPoolWebClientId: '391qfndfoupek19vgpq7o5nit0',
  },
  Storage: {
    AWSS3: {
      bucket: 'reittusmtbucket',
      region: 'eu-north-1',
    },
  },
```

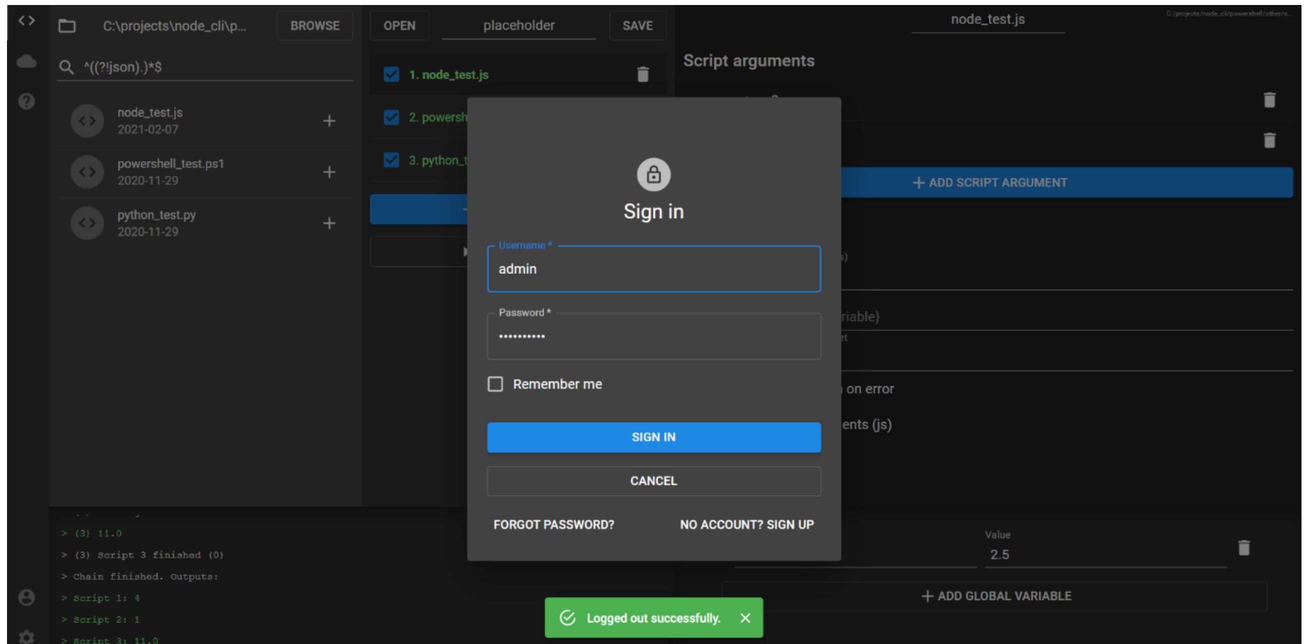
```
});
```

Koodiesimerkki 7. Logiikkaa säiliön tiedostojen listaukseen ja käsittelyyn, kun käyttäjä suorittaa haun.

```
const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  dispatch(showSpinner());
  Storage.list('')
    .then((results: S3File[]) => {
      dispatch(
        setGalleryFiles(
          results.map((file) => {
            const indexOfLastPathDelimiter = file.key.lastIndexOf('/');
            return {
              name: file.key.substring(indexOfLastPathDelimiter + 1),
              folder: file.key.substring(0, indexOfLastPathDelimiter),
              lastModified: file.lastModified.toUTCString(),
            };
          })
        )
      );
      return undefined;
    })
    .finally(() => dispatch(hideSpinner()))
    .catch((error) =>
      snackbarMessage(
        `Failed to fetch Gallery files. ${error}`,
        'error',
        dispatch
      )
    );
};
```

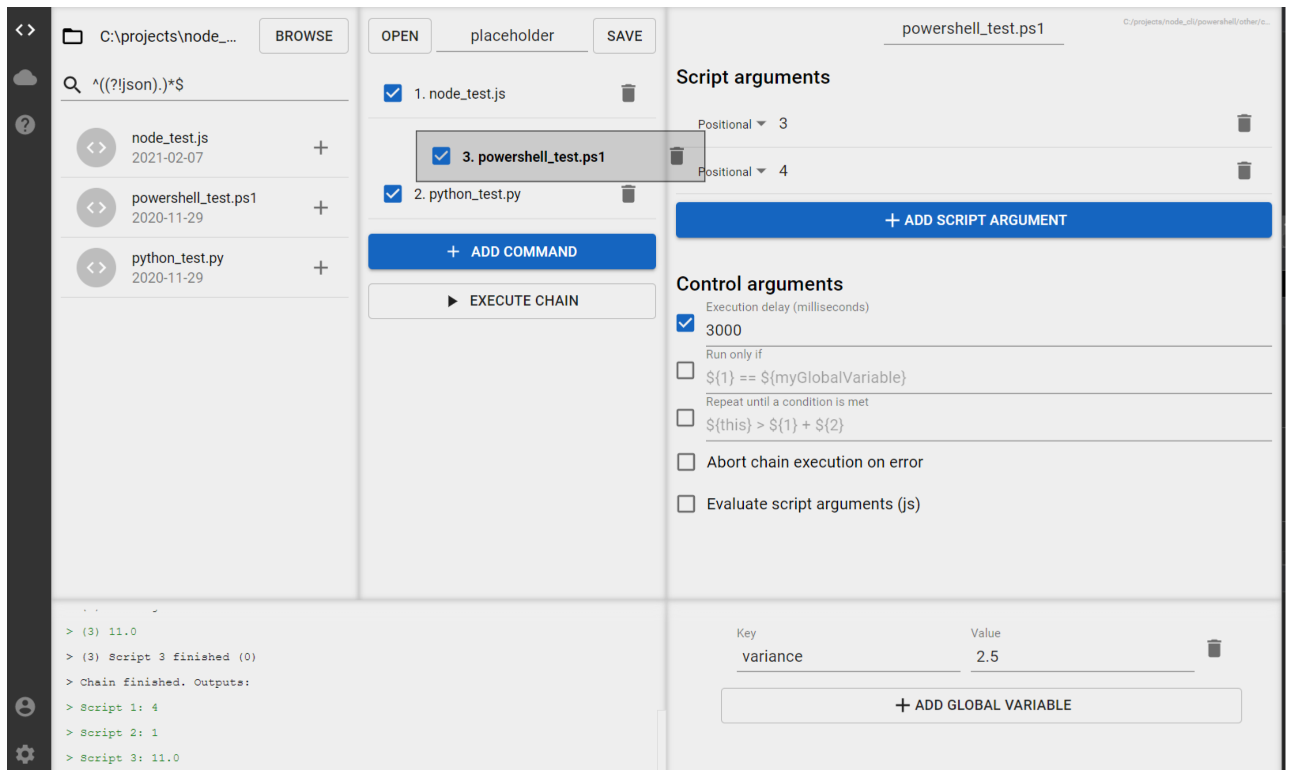
Käyttöliittymän osalta kaikki kirjautumis-, rekisteröitymis-, verifiointi- ja hallintanäkymät toteutettiin asetusten tapaan modaali-ikkunoilla (Kuva 20). Ilmoituksiin käytettiin Material UI:n Snackbar-komponenttia, jonka tilaa hallitaan Redux-säiliön kautta. Tällä tavalla toteutettuna ilmoitusten näyttäminen mistä tahansa komponentista on hyvin suoraviivaista. Täytyy kuitenkin huomioida, että kaikki ilmoitukset jakavat saman komponentin, jolloin usean yhtäaikaisen ilmoituksen näyttäminen ei ole mahdollista. Ilmoitusten pinoamiseen on olemassa Reactille esimerkiksi suosittu notistack-kirjasto, mutta en toistaiseksi nähnyt tälle tarvetta.

Kuva 20. Kirjautumisnäkymä uloskirjautumisen jälkeen.



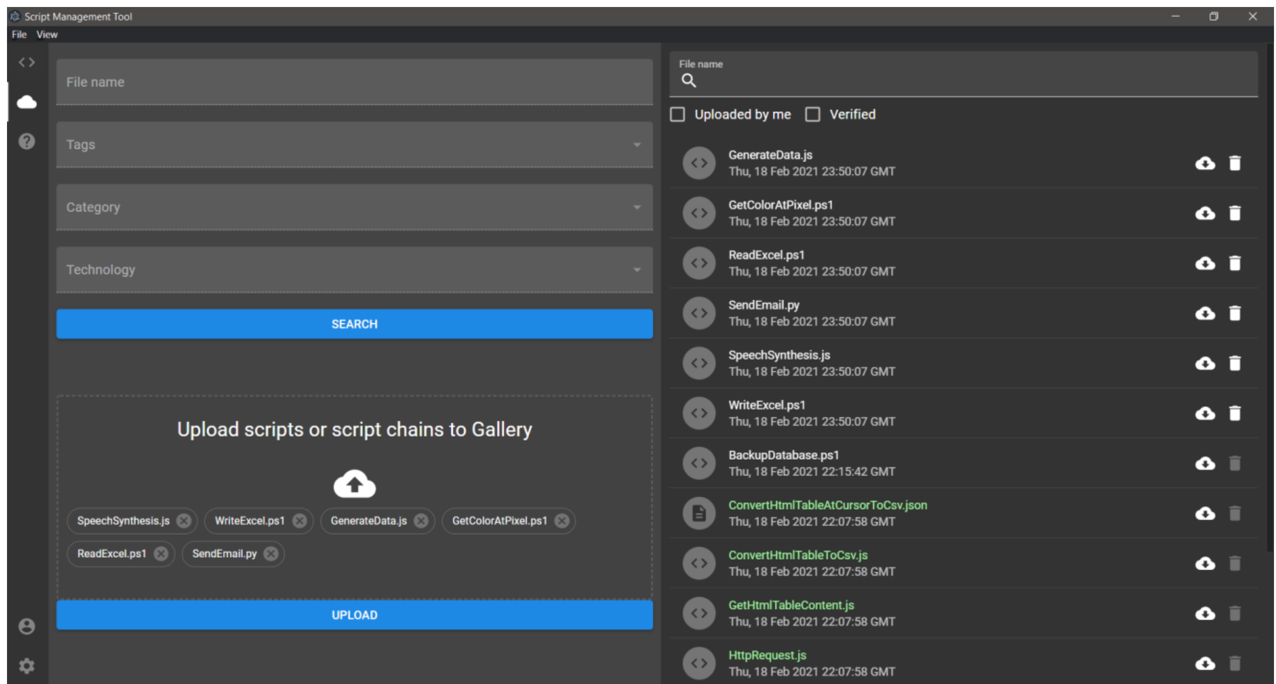
Käyttöliittymälle lisättiin vielä myös adaptiivinen teema, jotta sovelluksen värimaailma olisi yhdenmukainen käyttöjärjestelmän aktiivisen teeman kanssa. Tätä varten CSS-määrityksissä kaikki kustomoidut värit erotettiin prefers-color-scheme-ominaisuuden arvon mukaan, jota käyttämällä myös Material-UI-komponenttien värejä hallinnoidaan ylätasolla teeman kautta. Kuvassa 21 näkyy toteutetun vaalean teeman lisäksi myös Atlassian RBD-kirjastolla implementoitu skriptiketjun skriptien uudelleenjärjestäminen.

Kuva 21. Vaalea teema ja skriptien uudelleenjärjestäminen.



Käyttöliittymän gallerianäkymään lisättiin tarvittavat komponentit tiedostojen haku-, hakutulossuodatus-, lataus- ja lähetystoiminnoille (Kuva 22). MVP-toteutuksessa kaikki säiliön tiedostot haetaan yhdellä haulla ja tämän jälkeen tiedostoja voidaan suodattaa lokaalisti. Myöhemmässä toteutuksessa tiedostoja voidaan hakea niiden metatietojen, kuten tagien tai kategorian, perusteella, joten toteutukseen lisättiin paikkamerkit kyseisille hakukriteereille. Tulevaisuuden tiedostohaku metatietojen mukaan voidaan toteuttaa esimerkiksi lisäämällä tiedoston tallennuksen yhteyteen S3-säiliön konfiguraatiossa Lambda-pohjainen trigger-funktio, joka päivittää toteutukseen valittuun tietokantaan linkityksen tiedoston polun ja tämän metatietojen välille. Tällöin kaikki haut tehtäisiin tietokannasta eikä S3-rajapinnan kautta, joten toteutus vaatisi hieman enemmän työtä. Muutos on tarpeellinen, koska S3-palvelu ei itsessään tue tiedostojen hakemista käyttäjän määrittämien metatietojen mukaan.

Kuva 22. Gallerianäkymä.

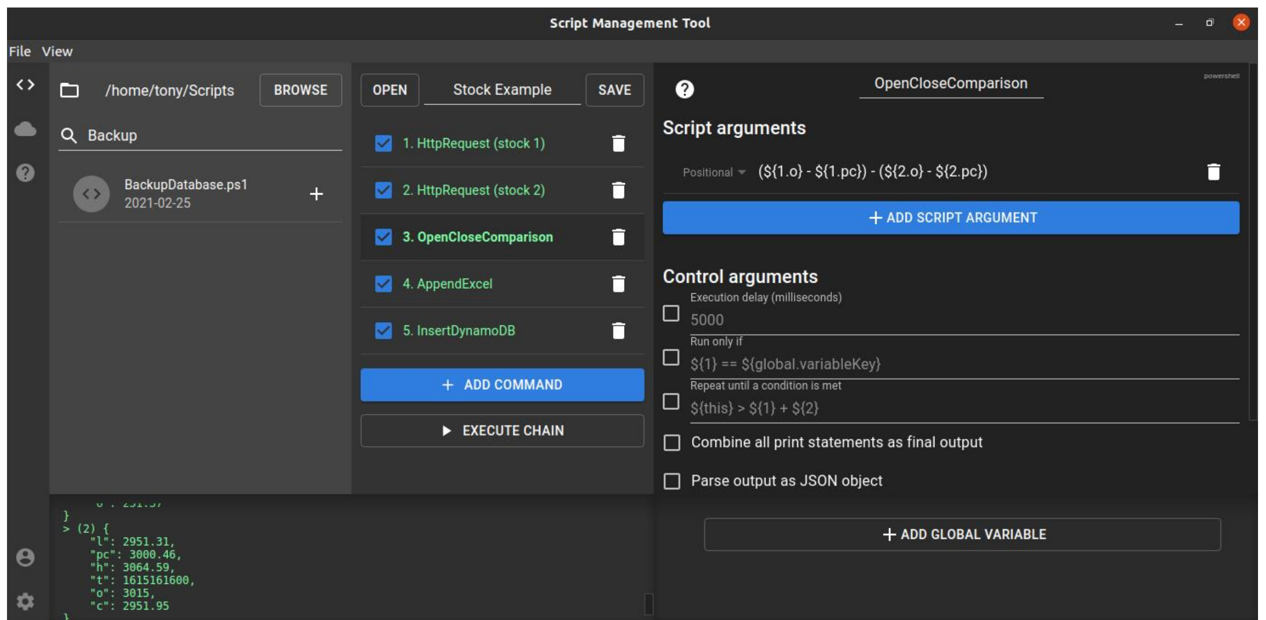


4.2.5 Viimeistely

Toteutusvaiheen päätteeksi toteutettiin vielä muutamia pienehköjä tarpeellisiksi osoittautuneita toimintoja sekä testattiin sovellusta Linux-pohjaisessa käyttöjärjestelmässä.

Yhtenä uutena toiminnallisuutena oli skripti- ja skriptiketjukohtaiset muistiinpanot, joiden avulla voidaan esimerkiksi avata tarkemmin skriptiketjulla olevan skriptin tai komennon argumentteja. Lisäksi skriptien tulosteiden hallinta vaati hieman täydennystä, joten sovellukseen lisättiin kaksi uutta kontrollirakennetta tähän liittyen. Ensimmäisenä uutena rakenteena oli skriptin kaikkien tulosteiden yhdistäminen lopputulokseksi, kun tavanomaisesti ainoastaan viimeinen tuloste tulkitaan tulokseksi. Toisena uutena rakenteena oli puolestaan skriptin tuloksen jäsentäminen JSON-objektiksi, minkä avulla skriptin tuloksesta voidaan pistenotaatiota käyttämällä jäsentää haluttuja osia. Kuvassa 23 esitetään näiden uusien kontrollirakenteiden hyödyntämistä konkreettisesti Linux-testauksen yhteydessä.

Kuva 23. Sovelluksen testausta Ubuntu-käyttöjärjestelmässä ja objektimuotoisen tietorakenteen jäsennystä pistenotaatiolla.



5 Johtopäätökset ja pohdinta

Työ eteni pitkälti suunnitelmien mukaan ja valmistui määritellyssä ajassa hyvin kelvolliseksi. Tässä luvussa verrataan vielä työn tavoitteita toteumaan sekä palataan työn alussa esitettyyn Electron-sovelluskehystä koskevaan kritiikkiin ja käyttökokemuskysymyksiin. Näiden jälkeen avataan aivan lopuksi projektin jatkosuunnitelmat niin työpöytäsovelluksen kuin jakelualustankin osalta.

5.1 Lopputulos ja toteutuksen arviointi

Toteutettu skriptienhallintasovellus vastasi erinomaisesti omiin automaatiotarpeisiini, vaikkakin toteutuksesta jäi puuttumaan yksittäisiä toiminnallisuuksia. Alkuperäisten tavoitteiden (Taulukko 1 ja Taulukko 2) lisäksi toteutukseen tuli suunnitteluvaiheessa huomaamattomaksi jääneitä toimintoja (Taulukko 3), jotka osoittautuivat tärkeämmiksi kuin osa suunnitelluista toiminnoista.

Taulukko 1. Alkuperäisten tavoitteiden toteuma (MVP).

Tavoite (MVP)	Toteuma
Alustariippumaton työpöytäsovellus	Toteutunut
Moderni käyttöliittymätoteutus	Toteutunut
Avatun skriptin parametrien jäsennys käyttöliittymälle	Osittain toteutunut
Paikallisten skriptien ja skriptiketjujen haku käyttöliittymällä	Toteutunut
Skriptiketjujen muodostaminen ja suorittaminen annetuilla argumenteilla	Toteutunut
Skriptien ohittaminen skriptiketjulla (tilapäinen valinta)	Toteutunut
Skriptien sisääntulon ja tulosteen hallinta	Toteutunut
Viimeisimmän skriptiketjun säilyttäminen istuntojen yli	Toteutunut
Kontrollirakenteet: odottaminen ketjun skriptien suoritusten välillä	Toteutunut
Kontrollirakenteet: suorittaminen vain tietyn ehdon täyttyessä	Toteutunut
Kontrollirakenteet: saman skriptin suorittaminen silmukassa useampaan kertaan	Toteutunut
Asetukset: pikanäppäimet tallennettujen skriptiketjujen suorittamiselle	Toteutunut
Asetukset: skriptikansio	Toteutunut

Taulukko 2. Alkuperäisten tavoitteiden toteuma (ei MVP).

Tavoite (ei MVP)	Toteuma
Sidosjärjestelmä ja sen konfigurointi: käyttäjähallinta	Toteutunut
Sidosjärjestelmä ja sen konfigurointi: skriptit ja skriptiketjut pilvessä	Toteutunut
Sidosjärjestelmän integraatio: rajapintakutsut	Toteutunut
Sidosjärjestelmän integraatio: lisäkonfigurointi esimerkiksi tietoturvan osalta	Osittain toteutunut
Käyttöliittymätoteutus: rekisteröityminen	Toteutunut
Käyttöliittymätoteutus: kirjautuminen	Toteutunut
Käyttöliittymätoteutus: sidosjärjestelmän skriptien ja skriptiketjujen haku- ja lähetysnäkymät	Toteutunut
Tuki useammalle skriptauskielelle ja parametrien syntakseille	Osittain toteutunut
Asetukset: pikanäppäimet tallennettujen skriptiketjujen suorittamiselle	Toteutunut
Asetukset: skriptikansio	Toteutunut
Kevyen Node.js-kirjaston toteuttaminen tallennettujen skriptiketjujen ajamiseen ilman käyttöliittymällistä työpöytäsovellusta	Ei toteutunut
Tarkemmat käyttöohjeet ja muu oheismateriaali	Osittain toteutunut

Taulukko 3. Kehitysvaiheessa esiin nousseet tarpeelliset toiminnot.

Toiminto	Toteuma
Skriptien uudelleenjärjestäminen	Toteutunut
Lokitus	Toteutunut
Skriptit ilman tiedostoja (komennot)	Toteutunut
Globaalit muuttujat	Toteutunut
Rikkoutuneiden tiedostopolkujen korjaus	Osittain toteutunut
Skriptikohtaiset muistiinpanot	Toteutunut
Ketjun suorituksen keskeyttäminen	Toteutunut
Kontrolliargumentit: skriptin tulosteiden yhdistäminen lopulliseksi tulokseksi	Toteutunut
Kontrolliargumentit: skriptin tulosteen jäsentäminen JSON-objektiksi	Toteutunut
Kontrolliargumentit: skriptien argumenttien ohjelmallinen käsittely ennen suoritusta (eval)	Toteutunut
Kontrolliargumentit: ketjun keskeyttäminen virheessä	Toteutunut
Äänimerkit (virhe ja suoritus)	Toteutunut
Käyttäjäasetukset: äänet	Osittain toteutunut
Käyttäjäasetukset: tulkkipolut	Osittain toteutunut
Adaptiiviset väriteemat	Toteutunut
Ilmoitukset	Toteutunut

Vaikka toiminnallisesti toteutus onnistui erinomaisesti, jäi se kuitenkin kauas täydellisestä. Muun muassa yksikkö- ja integraatiotestit jäivät täysin kirjoittamatta sekä sovelluksen testaus macOS-käyttöjärjestelmällä toteuttamatta. Myös ohjelmakoodi jäi paikoittain luettavuuden näkökulmasta puutteelliseksi, ja skriptigallerian hakutoiminnon osalta tyydyttiin hyvin vähään S3-palvelun rajoitteiden vuoksi. Tarkempien hakuehtojen implementointia varten tarvitaan tulevaisuudessa suurempia rakenteellisia muutoksia ja ottaa käyttöön tietokanta sekä toteuttaa tälle tietoturvallinen haku- ja päivitysrajapinta esimerkiksi Lambda-funktioilla. Aikataulurajoitteiden ja projektin laajuuden vuoksi nämä puutteet olivat kuitenkin odotettavissa ja hyväksyttäviä kokonaisuutta ajatellen.

5.2 Electron-sovelluskehys

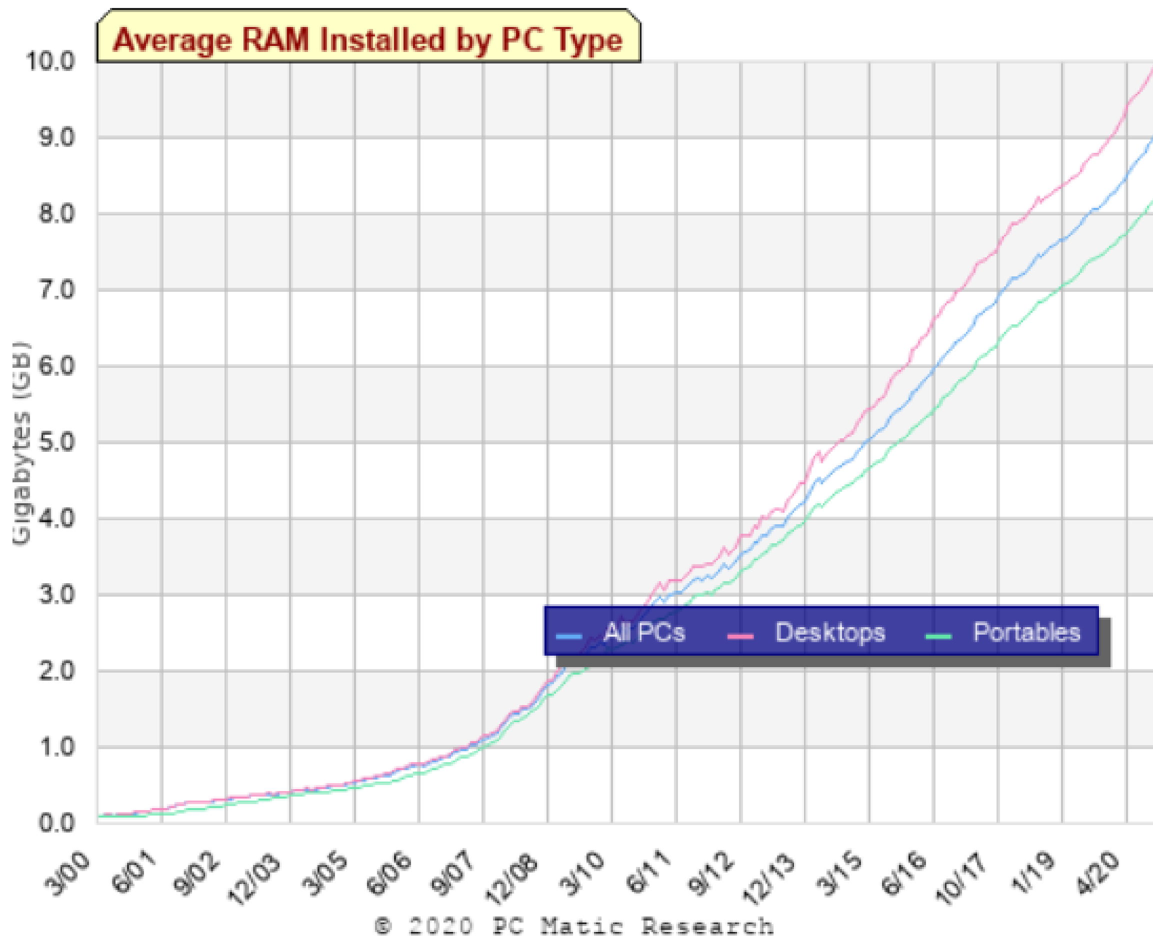
Yhtenä opinnäytetyön tavoitteena oli arvioida Electron-sovelluskehyksen käyttöä työpöytäsovellusten kehityksessä ja tarkastella, kuinka relevanttia sovelluskehystä kohtaan esitetty kritiikki on nykypäivänä.

Oman kokemukseni perusteella kehitystyö Electronilla on hyvin sujuvaa. Vastaan ei tullut kehityksen aikana kertaakaan ylitsepääsemättömiä ongelmia: suurimmat ajankäytölliset haasteet liittyivät viimeaikaisiin sovelluskehyksen tietoturvauudistuksiin, joiden vuoksi monet Electron-projektit ja -kirjastot olivat vanhentuneet eikä täten käytettävissä referenssimateriaalina sellaisenaan. Dokumentaatio ja sovelluskehyksen tarjoamat työkalut olivat kuitenkin erinomaiset omiin tarpeisiini, joten toimintojen implementointi onnistui ketterästi.

Muistinkäytön osalta Electron on odotetusti melko raskas. Sovellus käyttää Windows 10 -kannettavallani keskimäärin noin 150 MB muistia aktiivisessa käytössä ja suunnilleen 70 MB tämän oltua pidemmän aikaa passiivisena taustalla. Vaikka edellä mainitut luvut ovat suhteellisen suuria verrattuna useisiin vastaavan kokoluokan natiivisovelluksiin, ovat ne moderneissa kokoonpanoissa nykyisin melko kohtuullisia. Tietokoneissa oli tilastollisesti vuonna 2020 keskimäärin yli 8 GB muistia ja trendi on edelleen selkeästi kasvava (Kuva 24), joten käytetyn muistin optimointi tulee sovelluskehityksessä olemaan entistä vähemmän tärkeää. Sen sijaan, että sovelluksia optimoitaisiin käyttämään mahdollisimman vähän muistia, on tärkeämpää varmistaa, että muistivuotoja ei tapahdu. Skriptauskielissä tyypillisesti implementoitu automaattinen roskienkeräys (garbage collection) pitää tästä useimmissa tilanteissa huolen ilman erityistoimenpiteitä kehittäjiltä, kun taas manuaalista muistinhallintaa vaativissa kielissä muistin käyttöä voidaan optimoida enemmän, mutta prosessi altistaa järjestelmän kehitysvirheille ja näistä johtuville muistivuodoille. Electron-sovelluksissa mahdollisilta muistivuodoilta voidaan pitkälti välttyä varmistamalla muun muassa, että kaikki rekisteröidyt tapahtumankuuntelijat poistetaan asianmukaisesti ja muistissa pidettyjen dynaamisten listojen ei anneta kasvaa loputtomiin. Toteutetussa sovelluksessa IPC-tapahtumankuuntelijat rekisteröidään ja poistetaan komponentin ensimmäisen renderöinnin ja poistamisen yhteydessä Reactin useEffect-funktion avulla (Koodiesimerkki 8). Muulta osin JavaScriptissä kannattaa ehdottomasti suosia lokaalisti

määritettyjä muuttujia, koska automaattinen roskienkeräys ei puhdista käyttämättömiä globaaleja muuttujia. Jos globaaleja muuttujia käyttää, tulee ne käytön jälkeen puhdistaa manuaalisesti esimerkiksi asettamalla niille null-arvon.

Kuva 24. Tietokoneiden keskimääräisen keskusmuistin kehitys 2000–2020 (PC Matic, 2020).



Koodiesimerkki 8. IPC-tapahtumankuuntelijoiden rekisteröinti ja poistaminen Reactin useEffect-funktiolla.

```
useEffect(() => {
    ipcRenderer.on('chainLogs', (_event, log: Log) => {
      dispatch(addNewLog(log));
      if (log.type === 'error') soundError.play();
    });
    ipcRenderer.on('startedScript', (_event, scriptId: number) => {
      setExecutingScriptId(scriptId);
    });
    ipcRenderer.on('finishedScript', (_event, currentChainResults: []) => {
      setExecutionResults(currentChainResults);
```

```
    });
    ipcRenderer.on('finishedChain', () => {
      setChainIsExecuting(false);
      setExecutingScriptId(null);
      soundEndChain.play();
    });
    ipcRenderer.on('saveChain', (_event, result) => {
      if (result.success) {
        localStorage.setItem('chainName', result.name);
        setChainName(result.name);
      }
    });
    ipcRenderer.on('verifyUpdatedPaths', (_event, invalidPaths) => {
      setInvalidScriptPaths(invalidPaths);
    });
    return () => {
      ipcRenderer.removeAllListeners('chainLogs');
      ipcRenderer.removeAllListeners('startedScript');
      ipcRenderer.removeAllListeners('finishedScript');
      ipcRenderer.removeAllListeners('finishedChain');
      ipcRenderer.removeAllListeners('openChain');
      ipcRenderer.removeAllListeners('saveChain');
      ipcRenderer.removeAllListeners('verifyUpdatedPaths');
    };
  }, [dispatch]);
```

Suorituskyvyllisiä ongelmia en sovellusta normaaliolosuhteissa testattaessa havainnut. Koska toteutuksessa hyödynnettiin Reactia, renderöinti kohtuullisen suurillakin dynaamisilla listoilla toteutuu nopeasti. Renderöintiä on mahdollista tarvittaessa optimointia esimerkiksi useMemo ja useCallback -funktioilla, jos jokin toiminto on laskennallisesti raskas. Jos taas suorituskyvylliset ongelmat eivät liity renderöintiin tai käytössä olevat työkalut eivät riitä ratkaisemaan ongelmaa, voidaan Electron-sovelluksissakin tarvittaessa hyödyntää WebAssembly-teknologiaa. Optimointi ilman konkreettista syytä ei ole kuitenkaan järkevää, koska ajankäytön ja optimoinnista seuranneiden mahdollisten bugien lisäksi ohjelmakoodin luettavuus kärsii.

Yhteenvetona suosittelen Electronin käyttämistä työpöytäsovellusten kehittämiseen tilanteissa, joissa projektitiimillä on käytettävissä osaamista JavaScript-kehityksestä enemmän kuin natiivien työpöytäsovellusten kehittämisestä tai toteutettavasta sovelluksesta on jo valmiiksi olemassa web-implementaatio. Jälkimmäisessä tilanteessa Electronin käyttäminen on erittäin perusteltua, koska suuri osa alkuperäisen sovelluksen koodikannasta on sellaisenaan hyödynnettävissä työpöytäversiossa. Jos erinomaisen

suorituskyvyn ja pitkän kehitysajan sijaan kohtuullinen suorituskyky ja lyhyt kehitysaika kuulostaa paremmalta vaihtoehdolta, tarjoaa Electron tähän erinomaiset puitteet. Mikäli sovellusta on kuitenkin kriittistä suorittaa hyvin vanhoissa järjestelmissä, joissa on erityisen rajoitetut resurssit – varsinkin muistin osalta – on natiivisovellusten toteuttaminen perusteltua. Electron on kuitenkin vuosien saatossa kehittynyt valtavasti, joten on odotettavissa, että Electron-sovellusten erot natiivisovelluksiin kaventuvat nykyisestään.

5.3 Jatkosuunnitelmat

Sovellus ja sen lähdekoodi tulevat alkuperäisten suunnitelmien mukaan julkiseen jakoon, ja aion ylläpitää projektia sekä kehittää sitä esille tulevien tarpeiden mukaisesti. Julkisen jaon vuoksi oleellisimpia ensiaskeleita tulee kuitenkin olemaan ohjelmakoodin siistiminen sekä dokumentaation täydentäminen visuaalisin esimerkein ja mahdollisten vikatilanteiden ratkaisemisen osalta. Pitkäikäisillä projekteilla myös yksikkö- ja integraatiotestit ovat hyvin tärkeitä, joten tämänkin suhteen löytyy paljon tehtävää.

Varsinaisten toimintojen osalta toteutukseen tulee todennäköisesti ensimmäisten joukossa kokonaisten skriptiketjujen ketjuttaminen toisilla ketjuilla, jolloin täytyy myös implementoida ketjukohtaiset parametrit. Jatkokehityksessä käytettävyys tulee kuitenkin olemaan lisätoimintoja korkeammalla prioriteettilistalla. Tällä hetkellä esimerkiksi toisilta käyttäjiltä peräisin olevien skriptiketjujen suorittaminen on turhan hankalaa, koska ennen suorittamista täytyy manuaalisesti korjata käyttöliittymällä jokaisen ketjun tiedostollisen skriptin tiedostopolut vastaamaan oman tiedostojärjestelmän uusia polkuja. Tämä olisi osittain automatisoitavissa hakemalla esimerkiksi asetetusta skriptikansiosta rekursiivisesti kaikki samannimiset skriptit ja pyytämällä käyttäjää valitsemaan oikean polun. Vastaavanlaisia käytettävyyttä parantavia toimintoja tuli kuitenkin projektin loppupuolella mieleen useita, joten en näitä aikarajoituksen vuoksi MVP-versioon lähtenyt toteuttamaan.

Tiivistettynä jatkokehityksessä prioriteetteina tulee olemaan dokumentaatio, ohjelmakoodin luettavuus ja käytettävyystoiminnot. Näiden jälkeen tulee lisätoiminnot ja migraatio nykyisestä järjestelmäarkkitehtuurista Lambda-funktioihin ja tietokantaimplementaatioon, jotta voidaan kiertää S3-palvelun rajoitteet ja täten suorittaa tarkempia hakuja suuremmilla

tiedostojen lukumäärillä. Tämä transitio tulee mahdollistamaan myös käyttäjäkohtaisten pyyntöjen lukumäärän rajoittamisen API Gateway -palvelulla sekä tarkemmat palvelinpuolen tarkistukset esimerkiksi vastaanotettavien tiedostojen osalta Lambda-funktioilla.

Lähteet

1 & 1 IONOS. (2020). *Scripting languages: Explanation, Features and Examples.* https://www.ionos.com/digitalguide/websites/web-development/what-are-scripting-languages

Amazon Web Services. (2021a). *Amazon Cognito - Simple and Secure User Sign Up & Sign In | Amazon Web Services (AWS)*. https://aws.amazon.com/cognito

Amazon Web Services. (2021b). *Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)*. https://aws.amazon.com/s3

Amazon Web Services. (2021c). *Shared Responsibility Model - Amazon Web Services (AWS)*. https://aws.amazon.com/compliance/shared-responsibility-model

Asana. (n.d.). *What is Project Management & What are the Benefits?* Haettu 7.3.2021 osoitteesta https://asana.com/resources/benefits-project-management

BGZDevTips. (22.6.2016). *Compilation vs Interpretation [video].* YouTube. https://www.youtube.com/watch?v=JNMy969SjyU

Brown, S. (n.d.). *The C4 model for visualising software architecture.* Haettu 15.3.2021 osoitteesta https://c4model.com

Contribyte. (n.d.). *Agile-sanasto*. Haettu 15.3.2021 osoitteesta https://contribyte.fi/ajatuksia/agile-sanasto

Digital.ai. (2020). 14th Annual State of Agile Report. https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494

Doglio, F. (2020). *The JIT in JavaScript: Just In Time Compiler*. https://blog.bitsrc.io/the-jit-in-javascript-just-in-time-compiler-798b66e44143

Eby, K. (2017). *Full Comparison: Agile vs Scrum vs Waterfall vs Kanban*. https://www.smartsheet.com/agile-vs-scrum-vs-waterfall-vs-kanban

Evans, B. (2014). *Understanding Java JIT Compilation with JITWatch, Part 1*. https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html

Faurescu, P. (2017). *The C4 software architecture model.* https://qappdesign.com/code/the-c4-software-architecture-model

Favell, A. (2020). *The History of Git: The Road to Domination*. https://www.welcometothejungle.com/en/articles/btc-history-git

Flanagan, D. (2020). *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language 7th Edition.* O'Reilly Media.

Google. (2021). *Trusted Cloud Insfrastructure (IaaS) | Google Cloud*. Haettu 15.3.2021 osoitteesta https://cloud.google.com/security/infrastructure

Hinkelmann, F. (2017). *Understanding V8's Bytecode.* https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775

Huang, R. (2019). *Is Your Data Secure in the Cloud? An Overview of Alibaba Cloud's Data Security Architecture*. https://www.alibabacloud.com/blog/data-security-on-the-cloud_594862

Huosianmaa, T. (2020). *Projektinhallinnan menetelmät ohjelmistokehityksessä* [kanditaatin tutkielma, Tampereen yliopisto]. http://urn.fi/URN:NBN:fi:tuni-202012068540

Jeffries, R. (2018). *Scrum is not an Agile Software Development Framework*. https://ronjeffries.com/articles/018-01ff/scrum-not-asd-1

Kashyap, S. (2019). *The Importance of Project Management Software*. https://www.proofhub.com/articles/importance-of-project-management-software/

Kopf, B. (2018). *The Power of Figma as a Design Tool | Toptal*. https://www.toptal.com/designers/ui/figma-design-tool

Krügel, C.;Robertson, W. K.;Valeur, F.;& Vigna, G. (2004). Static Disassembly of Obfuscated Binaries. Teoksessa *Proceedings of the 13th USENIX Security Symposium, 13(18).* https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/kruegel/kruegel.pdf

Material-UI. (n.d.). *Our Vision - Material-UI*. Haettu 15.3.2021 osoitteesta https://material-ui.com/discover-more/vision

Microsoft. (2021a). *Security technical capabilities in Azure*. Haettu 15.3.2021 osoitteesta https://docs.microsoft.com/en-us/azure/security/fundamentals/technical-capabilities

Microsoft. (2021b). *Basic Editing in Visual Studio Code*. Haettu 15.3.2021 osoitteesta https://code.visualstudio.com/docs/editor/codebasics

Microsoft. (2021c). *TypeScript: Documentation - Decorators*. Haettu 15.3.2021 osoitteesta https://www.typescriptlang.org/docs/handbook/decorators.html

Microsoft. (n.d.). *TypeScript: Why does TypeScript exist?* Haettu 15.3.2021 osoitteesta https://www.typescriptlang.org/why-create-typescript

OpenJS Foundation. (n.d.-1). *Quick Start Guide | Electron*. Haettu 15.3.2021 osoitteesta https://www.electronjs.org/docs/tutorial/quick-start#main-and-renderer-processes/

OpenJS Foundation. (n.d.-2). *Electron Apps | Electron*. Haettu 15.3.2021 osoitteesta https://www.electronjs.org/apps

OpenJS Foundation. (n.d.-3). *Security, Native Capabilities, and Your Responsibility | Electron*. Haettu 15.3.2021 osoitteesta https://www.electronjs.org/docs/tutorial/security

PC Matic. (2020). *Memory Trends | PC Matic TechTalk [kuva]*. https://techtalk.pcmatic.com/research-charts-memory

Perforce Software. (2018). *Git vs. SVN | Git and SVN Features, Commands, & More*. https://www.perforce.com/blog/vcs/git-vs-svn-what-difference

PyPy. (n.d.). *Goals and Architecture Overview — PyPy documentation*. Haettu 15.3.2021 osoitteesta https://doc.pypy.org/en/latest/architecture.html

Seymour, B. (5.3.2020). *Understanding AOT Compilers, JIT Compilers, and Interpreters [video]. YouTube.* https://www.youtube.com/watch?v=5rn_MAspYFM

Solanki, J. (2021). *Cloud Pricing Comparison 2021: AWS vs Azure vs Google Cloud*. https://www.simform.com/compute-pricing-comparison-aws-azure-googlecloud

Stack Exchange. (2020). *Stack Overflow Developer Survey 2020.* https://insights.stackoverflow.com/survey/2020

Tapio, T. (2010). *Riskienhallinta perinteisessä ja ketterässä ohjelmistokehityksessä* [pro gradu -tutkielma, Tampereen yliopisto]. http://urn.fi/urn:nbn:fi:uta-1-20674

The Software House. (2020). *State of Frontend 2020 Report – frontend development trends*. Haettu 15.3.2021 osoitteesta https://tsh.io/state-of-frontend

UX Tools. (2020). *2020 Design Tools Survey*. https://uxtools.co/survey-2020

Wikipedia. (2020). *Ahead-of-time compilation*. Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Ahead-of-time_compilation

Wikipedia. (2021a). *C4 model.* Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/C4_model

Wikipedia. (2021b). *Just-in-time compilation*. Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Just-in-time_compilation

Wikipedia. (2021c). *Cloud computing*. Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Cloud_computing

Wikipedia. (2021d). *JavaScript.* Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/JavaScript

Wikipedia. (2021e). *Bytecode*. Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Bytecode

Wikipedia. (2021f). *Electron (software framework)*. Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Electron_(software_framework)

Wikipedia. (2021g). *Interpreter (Computing).* Haettu 15.3.2021 osoitteesta https://en.wikipedia.org/wiki/Interpreter_(computing)

Wilcurt, J. (n.d.). *Cross-Platform Desktop App Development*. Haettu 15.3.2021 osoitteesta https://xpda.net

VMware. (n.d.). *What is Cloud Scalability?* Haettu 15.3.2021 osoitteesta https://www.vmware.com/topics/glossary/content/cloud-scalability

Zammetti, F. (2020). *Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker.* Apress.