



Työpöytäsovellus Electronilla

Sovelluksen tekeminen Electronilla ja Angularilla

Elina Vääntinen

Opinnäytetyö, AMK

Kesäkuu 2024

Tietojenkäsittely

Tradenomi

Vänttinen, Elina

Työpöytäsovellus Electronilla. Sovelluksen tekeminen Electronilla ja Angularilla.

Jyväskylä: Jyväskylän ammattikorkeakoulu. Kesäkuu 2024, 31 sivua.

Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Työpöytäsovellusten kehittäminen natiiveilla kielillä voi vaatia yrityksiltä paljon aikaa, rahaa ja juuri oikean kehityskielen osaavia työntekijöitä. Electron-sovelluskehys on kehitetty tehokkaaksi vaihtoehdoksi niille, jotka haluavat tehdä kehitystä web-kielillä ja aikaansaada sekä monialustaisen että natiivinkaltaisen sovelluksen, joka voidaan asentaa useille eri käyttöjärjestelmille.

Tutkimuksen taustalla oli toimeksiantaja Skillwell Oy, joka oli kiinnostunut ensinnäkin Electron-sovelluskehystä ja sen tuomista hyödyistä hybridisovelluksessa, joka on kehitetty yhdessä frontend-sovelluskehys Angularin kanssa. Skillwellillä oli entuudestaan laajalti kokemusta asiakkaiden tarpeisiin tehdyistä PWA-sovelluksista, ja Electronilla toivottiin paikattavan tiettyjä PWA-sovelluksissa todettuja puutteita, kuten tiedostojärjestelmän hallintaa. Näin ollen tutkimuksen tavoitteeksi tuli tutkia, mikä Electron on ja lisäksi millaisia hyötyjä siinä on verrattuna PWA-sovellukseen.

Tutkimuksen toteutustavaksi valikoitui tutkimuksellinen kehitystyö. Aluksi otettiin selvää Electronista, sen taustasta ja rakenteista teoriassa pääasiassa verkkolähteitä hyödyntäen. Vertailevan otteen vuoksi mukaan valikoitui myös verkkolähteitä, joissa verrattiin Electronia ja PWA-sovellusta keskenään. Teoriaosuuden lisäksi kehitettiin proof of concept-tyylinen sovellus, jossa hyödynnettiin Electronin rinnalla Angularia. Sovelluksen kehityskulkua raportoitiin ja lopullisessa sovelluksessa esiteltiin Electronin hyötyjä.

Tulosten pohjalta johtopäätöksinä todettiin, että Electronilla ja Angularilla voi saada aikaan natiivilta työpöytäsovellukselta näyttävän sovelluksen, jossa on natiivin sovelluksen kaltainen menu ja esimerkiksi pääsy tietokoneen tiedostojärjestelmään. Sovelluksen saattoi pakata eri käyttöjärjestelmille sopiviin tiedostomuotoihin, kuten esimerkiksi Windowsille .exe-tiedostoksi, josta käyttäjä pystyi helposti asentamaan sen tietokoneelleen. PWA-sovelluksiin verrattuna Electronilla tehty sovellus oli kooltaan suuri, mikä todettiin Electron-sovelluksen suurimmaksi haittapuoleksi. Electronin todettiin kuitenkin olevan oiva vaihtoehto sellaisiin käyttötarkoituksiin, joissa suuresta koosta ei ole haittaa, halutaan kehittää alustariippumattomasti ja sovellukselta vaaditaan työpöytäsovellusten natiivinkaltaisia ominaisuuksia.

Avainsanat (asiasanat)

sovelluskehukset, ohjelmistokehukset, Electron

Vänttinen, Elina

Desktop Application with Electron. Making an Application with Electron and Angular.

Jyväskylä: JAMK University of Applied Sciences, June 2024, 31 pages.

Degree Programme in Business Information Technology. Bachelor's Thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

Building desktop applications may require a lot of time, money and just the right employees who know how to develop with the right platform specific languages. Electron framework was built as an alternative for those who wish to use web languages and accomplish a multi-platform application that still has the look and feel of a native desktop app.

The commissioner for this project was Skillwell Oy, a company with interest for Electron framework and the benefits it could provide when used together with frontend framework Angular in order to build a hybrid application. Skillwell had vast experience of building PWA applications for customers' needs, and they hoped to see Electron excel in some areas that PWA was found lacking in. Thus, the objectives of this research were to find out what is Electron framework and how do Electron apps compare to PWA apps.

The method of research was research-based development work which has the focus on working life development. For the theoretical basis mainly online sources were used to find out about Electron framework, its background and underlying technologies. Due to the comparative approach between Electron and PWA, some blog texts and articles comparing the two were used as well. In addition to writing theory, a small proof of concept application was developed using Electron and Angular. The flow of development was reported, and some benefits of Electron were shown in the final POC application.

In the results of the research, it was concluded that with Electron and Angular a native-like app can be achieved with a native-like menu and with access to computer's underlying file system. The same application could be packaged for various operating systems, for example for Windows in .exe format, so that a user could easily install the application to their computer. When compared to PWA apps, Electron apps are much bigger in size which was considered as the biggest drawback of Electron. However, it was concluded that when app size is not a problem and one wants to achieve a multi-platform desktop app with native-like feel, Electron is a beneficial framework to use.

Keywords/tags (subjects)

application frameworks, desktop applications, Electron

Sisältö

1	Johdanto	6
2	Tutkimusasetelma	7
2.1	Tausta ja tavoitteet	7
2.2	Tutkimusmenetelmät	8
2.3	Aineistonkeruu ja -analyysi	8
3	Toimintaympäristö ja teknologiat	9
3.1	Web-teknologiat.....	9
3.1.1	HTML ja CSS	9
3.1.2	Webin kehityskieli JavaScript.....	10
3.2	Web-sovelluskehys Angular	10
3.3	Web-, työpöytä- ja hybridisovellukset	11
3.4	Progressiivinen verkkosovellus PWA	12
3.5	Sovelluskehys Electron	12
3.5.1	Taustalla teknologiat Node.js ja Chromium	13
3.5.2	Kahden prosessin rakenne.....	14
3.5.3	Tärkeimmät moduulit	15
3.5.4	Tietoturva ja muut haasteet	16
4	Kehitysprosessi.....	17
4.1	Electronin käyttöönotto yleisesti	17
4.2	Kehitys Angularin kanssa.....	18
4.3	Prosessien välinen kommunikaatio käytännössä	20
4.4	Sovelluksen paketointi	25
5	Tulokset.....	25
6	Pohdinta.....	27
6.1	Luotettavuus ja eettisyys	27
6.2	Ideoita jatkekehitykseen	28
6.3	Johtopäätökset.....	28
Lähteet		30

Kuviot

Kuvio 1	Electron-API:t ja niiden käytettävyys prosesseissa	15
Kuvio 2	Esimerkki sovellusikkunan luonnista Electronilla	16

Kuvio 3 Angular-projektin avaaminen Electronin sovellusikkunaan main.ts-tiedostossa	20
Kuvio 4: Menu ja datan lähetys pääprosessista renderöintiprosessille	22
Kuvio 5: file.service.ts-tiedosto kokonaisuudessaan	23
Kuvio 6: Templatin koodi tiedostossa editor.component.html	23
Kuvio 7: Editor-komponentin koodi tiedostossa editor.component.ts	24
Kuvio 8: Viestin vastaanottaminen main.ts-tiedostossa	24
Kuvio 9: Electron forgen skriptit package.json-tiedostossa	25
Kuvio 10: POC-sovelluksen menu avataan	26

1 Johdanto

Alustariippumaton kehitys ja progressiivinen verkkosovellus eli PWA (progressive web application) ovat nykyajan sovelluskehitystrendejä. PWA:n hyötyihin kuuluu alustariippumattomuuden lisäksi natiivin sovelluksen kaltainen käyttökokemus, kehittäminen tutuilla verkkoteknologioilla, offline-käyttömahdollisuus välimuistin avulla ja asennettavuus sekä mobiililaitteille että työpöytäkoneille. On kuitenkin olemassa tärkeitä toimintoja, joihin pelkkä PWA ei pysty, kuten ohjelman alla olevaan käyttöjärjestelmään käsiksi pääsy. Saavuttaaksemme nämä toiminnot kehittämättä natiivia työpöytäsovellusta, sovelluksen voi muuttaa niin sanotuksi hybridisovellukseksi Electronilla. Hybridisovellus yhdistää web-sovelluksen ja natiivin sovelluksen elementit niin, että toteutus on tehty web-teknologioilla, mutta lopputulos vaikuttaa natiivilta sovellukselta (Sanderson 2020.)

Tässä opinnäytetyössä tarkastellaan Electron-sovelluskehystä, jolla voidaan muuntaa web-tekniikoilla kehitetty verkkosovellus natiivin työpöytäsovelluksen kaltaiseksi, tietokoneelle asennettavaksi hybridisovellukseksi. Työssä perehdytään erityisesti Electronilla kehitetyn sovelluksen hyötyihin PWA:han verraten. Aluksi tutustutaan aiheeseen teorialähteiden avulla. Tutustutaan lyhyesti sen toiminnan mahdollistaviin tekniikoihin, Nodeen ja Chromiumiin. Lisäksi käydään läpi, mitä web-tekniikat ja PWA ovat ja millaisia ratkaisuja niillä voidaan saada aikaan. Sen jälkeen paneudutaan tarkemmin Electronin prosessimalliin ja tärkeimpiin ominaisuuksiin.

Osana työtä kehitettiin niin sanottu proof of concept -sovellus (lyhyesti POC), eli konseptitodistus Electronin toiminnasta. Kehitys toteutettiin hyödyntäen web-sovelluskehys Angularia yhdessä Electronin kanssa ja kehityksen vaiheet dokumentoitiin opinnäytetyön kehitysosioon. Osion tarkoituksena on tutustuttaa lukija hybridisovelluksen kehitysprosessiin, oppia itse käyttämään Electronia ja tuoda sen hyötyjä esiin käytännössä.

Toimeksiantajana tälle opinnäytetyölle toimi yritys Skillwell. Skillwell kehittää muun muassa PWA-sovelluksia asiakasyritysten tarpeisiin. Nykyisissä ratkaisuissa oli havaittu joitakin haasteita, joihin Electronin toivottiin tuovan helpotusta. Electron-integraation toivottiin lisäävän mahdollisuuksia kehitettävien sovellusten tehokäyttöön, kuten esimerkiksi tiedostojen ja kokonaisten hakemistorekenteiden massaoperaatioihin. Toimeksiantaja oli siis kiinnostunut alkamaan käyttää Electronia osana sovelluskehitystyötä, mutta halusi ensin saavuttaa paremman ymmärryksen tämän sovelluskehityksen toiminnasta.

2 Tutkimusasetelma

Tässä luvussa käydään läpi miksi ja miten tämä opinnäytetyö toteutetaan. Aloitetaan toimeksiantajasta ja tutkimusongelman taustasta, tutkimuksen tavoitteista ja niistä tutkimusmenetelmistä, joilla tämä opinnäytetyö toteutetaan. Lisäksi kerrotaan opinnäytetyötä varten kerätystä aineistosta, eli millaisista lähteistä aineistoa on kerätty ja miksi. Lopuksi kerrotaan vielä aineistohallinnasta ja analyysistä.

2.1 Tausta ja tavoitteet

Toimeksiantajana on jyväskylälainen vuonna 2018 perustettu IT-alan yritys Skillwell Oy, joka tarjoaa asiakkailleen digitaalisten palveluiden kehittämistä ja toimii Amazon Web Services-pilvipalvelukumppanina. Skillwellin tilanne on se, että osana liiketoimintaansa he kehittävät Angularilla PWA-sovelluksia asiakasfirmojen tarpeisiin. PWA-ratkaisuissa on kuitenkin havaittu rajoitteita, joita toivotaan ylitettävän mahdollisella Electron-integraatiolla. Havaituista puutteista olennaisimmat liittyvät työpöytäsovellusten tehokäyttöön, ja niistä tärkeimpinä toimeksiantaja mainitsee erilaiset tiedostojen ja kokonaisten hakemistorakenteiden massaoperaatiot. Muita mahdollisia tarpeita sanotaan olevan sovellusten väliseen yhteistoimintaan liittyvät asiat, kuten esimerkiksi parempi leikepöydän hallinta tai sovellusten väliset linkit, joista esimerkkinä mainitaan mahdollisuus avata sähköpostista jokin materiaali suoraan kehitetyssä sovelluksessa.

Tutkimuskysymyksiä ovat ”Mikä on Electron?” ja ”Mitä hyötyä Electron-sovelluksesta on verrattuna PWA-sovellukseen?”. Opinnäytetyössä pyritään tutustuttamaan lukija Electroniin ja selvittämään, miten sitä käytetään ja mitä hyötyjä sen käyttöönotosta on. Hyötyjen selvittämisen vastapainoksi etsitään vastausta myös siihen, voiko Electronin käytöstä olla jotakin haittaa, ja missä tilanteessa kannattaisi rakentaa PWA tai muu ratkaisu Electron-sovelluksen sijaan.

Osana opinnäytetyötä tehdään proof of concept-tyylinen kehitystyö. Proof of concept tarkoittaa ”konseptitodistusta”, jonka tarkoituksena on osoittaa jonkin tietyn menetelmän toteuttamiskelpoisuus. Tässä opinnäytetyössä siis kehitetään pieni esimerkkisovellus Angularilla ja Electronilla keskittyen erityisesti Electronin toimintaan. Työn rajaamiseksi Angular-sovellus pidetään yksinkertaisena ja pienenä.

2.2 Tutkimusmenetelmät

Tämän opinnäytetyön tutkimusmenetelmä on tutkimuksellinen kehittämistyö, jossa käytetään kvalitatiivista menetelmää. Kvalitatiivinen tutkimusmenetelmä tarkoittaa laadullista tutkimusta, jonka tavoitteena on kokonaisvaltaisesti ymmärtää valittua kohdetta, sen olemassaolon syitä ja seurauksia. Tällaisen tutkimusmenetelmän tyypillisiä tiedonhankinnan keinoja ovat haastattelut, osallistuva havainnointi ja erilaisten tekstien ja dokumenttien analyysit. Tässä opinnäytetyössä käytetään jälkimmäistä keinoa, eli tekstien ja dokumenttien analyysiä. Tällä tavoin pyritään aluksi muodostamaan kokonaisvaltainen käsitys Electron-sovelluskehiksestä ja siitä ympäristöstä, missä se toimii. (Hirsjärvi, Remes & Sajavaara 2013, 160–164.)

Toinen tiedonkeruutapa tässä opinnäytetyössä on kehittämistyö. Tutkimuksellisen kehittämistyön lähtökohtana on työelämästä noussut kysymys tai ongelma. Tutkimuksellisessa kehittämisessä yhdistyvät konkreettinen kehittäminen, tutkimuksellisten menetelmien soveltaminen ja aineiston analyysi (Jyväskylän ammattikorkeakoulu 2024). Kuten viime luvussa 2.2 jo todettiin, osana opinnäytetyötä kehitetään pieni konseptitodistus Electron-sovelluksesta. Tätä demon kaltaista sovellusta tehdessä opitaan, miten Electron otetaan käyttöön ja miten sitä hyödynnetään yhdessä Angularin kanssa. Kehitystyön pohjalta toivotaan löytyvän dataa, jota hyödyntää lopullisessa pohdinnassa muun aineiston rinnalla.

2.3 Aineistonkeruu ja -analyysi

Aineistoa kerätään pääasiassa avoimista verkkolähteistä, sillä ne ovat tässä aiheessa kaikkein eniten ajan tasalla. Sovelluskehiksestä tulee jatkuvasti uusia versioita ja PWA-teknologioihin tulee uusia ratkaisuja, joten on tärkeää tehdä tutkimus- ja kehitystyö ajantasaisen tiedon ja työkalujen avulla. Erityisesti kehitysvaiheessa on kiinnitettävä huomiota apuna käytettävien tutoriaalien julkaisuajankohtaan. Electronin omilta nettisivuilta löytyvä dokumentaatio on tässä hyvä lähde.

Opinnäytetyötä varten on etsitty myös vertaisarvioituja tutkimuksia ja artikkeleita, mutta etenkin jälkimmäisiä ei ole juuri löytynyt aiheeseen liittyen. Validia tutkimustietoa on haettu Statistasta, joka on yksi maailman kattavimmista tilastotiedon ja markkinadatan sivustoista. Teoriatietoa varten käytetään avointen verkkolähteiden rinnalla kirjoja, jotka on molemmat julkaistu vuonna 2017.

Tästä syystä tiedonhankinta näiden kirjojen osalta keskittyy vain teoriaan ja Electronin synnyn historiaan, sillä kehitystä varten lähteet ovat jo mahdollisesti vanhentuneet.

Aineistoa analysoidessa tyyli on vertaileva, sillä tarkoitus on löytää Electronin hyödyt pelkkään PWA-sovellukseen verraten. Koska yksi tapa perehtyä minkä tahansa järjestelmän toimivuuteen on omakohtainen kokeileminen, tullaan tässä opinnäytetyössä kokeilemaan itse Electron-sovelluksen kehitystä. Kehitystyön tuloksia voidaan sitten verrata PWA-sovelluksen toimintaan ja näistä tuloksista tuottaa esimerkiksi taulukkomuotoinen katsaus PWA:n ja Electron-sovelluksen hyviin ja huonoihin puoliin.

3 Toimintaympäristö ja teknologiat

Tässä luvussa avataan opinnäytetyön aiheeseen liittyviä termejä, teknologioita ja toimintaympäristöä. Jotta saisi täyden ymmärryksen Electron-sovelluskehiksestä, on ensin ymmärrettävä, mitä ovat web-teknologiat ja miksi niillä halutaan kehittää muutakin, kuin verkkosivuja ja -sovelluksia. Luvussa esitellään lyhyesti myös verkkosovelluskehys Angular, sillä sitä tullaan käyttämään opinnäytetyön myöhemmässä vaiheessa konseptitodistusta kehitettäessä.

3.1 Web-teknologiat

Web-teknologioita on muitakin, mutta tässä luvussa käsittelemme HTML:ää, CSS:ää ja JavaScriptiä, sillä ne kuuluvat web-kehittäjän ydinosamiseen ja ovat kaikista koodaus-, skriptaus- ja merkintäkielistä suosituimmat (2021 Developer Survey 2021). Niitä tullaan myös käyttämään opinnäytetyön kehittämisosiossa. Web-teknologioiden suurin hyöty on se, että niillä rakennettua sovellusta voidaan käyttää monilla erilaisilla alustoilla (Schmitz 2019).

3.1.1 HTML ja CSS

HTML eli HyperText Markup Language on avoimesti standardisoitu merkintäkieli, jolla näytetään hyperlinkkejä sisältävää tekstiä eli hypertekstiä nettiselaimessa. HTML on siis se kieli, jolla nettisivut on kirjoitettu. HTML5 on viides ja viimeisin versio HTML:stä, ja se on W3C-konsortion suosittelema. HTML5:n määrittely on dokumenttioliomallin (DOM) mukainen, eli dokumentti kuvataan

puurakenteisena. HTML-dokumentin ulkoasua voidaan muokata CSS:llä ja toiminnallisuuksia kirjoittaa JavaScriptillä. (HTML Living Standard 2022.)

CSS eli Cascading Style Sheets on verkkosivuja varten kehitetty tyyliohje. CSS-tiedostoihin määritellään merkkikielellä toteutetun verkkosivun tai -sovelluksen elementtien, kuten vaikkapa kuvien, linkkien tai navigaation ulkoasua koskevia sääntöjä. Sääntö koostuu valitsimesta ("selector"), jonka jälkeen aaltosulkeisiin merkitään ominaisuus ja sille annettu arvo. Sääntö voi yksinkertaisesti koskea vaikkapa otsikon kokoa ja sijaintia, mutta nykyään CSS:n avulla voidaan tehdä paljon monimutkaisempia asioita, kuten elementtien animointia.

3.1.2 Webin kehityskieli JavaScript

JavaScript tunnetaan ennen kaikkea webin kehityskielenä. Se on skriptikieli ja Stack Overflow:n teettämän tutkimuksen mukaan maailman eniten koodaamiseen käytetty kieli vuonna 2021 (2021 Developer Survey 2021). Sen avulla voidaan luoda dynaamisesti päivittyvää ja interaktiivista sisältöä nettisivuille, kuten klikattavia nappeja, animaatioita jne. JavaScript ajetaan selaimessa JavaScript-moottorin avulla. Kun käyttäjä lataa selaimellaan nettisivun, kyseisen sivun HTML, CSS ja JavaScript ladataan ja ajetaan selaimen välilehdessä ja käyttäjälle näytetään sivun sisältö. Jokainen selaimen välilehti muodostaa oman ajoympäristönsä. Vaikka JavaScript on alun perin luotu nimenomaan webin skriptikieleksi, sitä voidaan käyttää myös palvelinpuolen kehityksessä Node.js-ajoympäristön avulla. (What is JavaScript, 2021.)

3.2 Web-sovelluskehys Angular

Web-sovelluskehys Angular valikoitui mukaan tähän opinnäytetyöhön, sillä toimeksiantaja käyttää sitä projekteissaan. Stack Overflow:n teettämän tutkimuksen mukaan se oli vuonna 2021 maailmanlaajuisesti neljänneksi suosituin kehittäjien käyttämä web-sovelluskehys (Stack Overflow 2021). JavaScriptin sijaan Angular on TypeScript-pohjainen. TypeScript on JavaScriptin pohjalta Microsoftin kehittämä ja ylläpitämä, staattisesti tyyppitetty koodauskieli, joka kääntyy build-vaiheessa JavaScriptiksi. Se kehitettiin korjaamaan JavaScriptin puutteita suuren mittakaavan projekteihin. Esimerkiksi bugien löytäminen kehityksen aikaisessa vaiheessa on tyyppityksen ansiosta helpompaa TypeScriptillä kuin JavaScriptillä. (White, 2020.)

Angular-sovellus koostuu komponenteista. Kullakin komponentilla on luokkaosio, johon kirjoitetaan toiminnallisuudesta vastaava koodi ja data. Luokkaosion lisäksi komponentilla on oma HTML-templaatti, johon kirjoitetaan käyttöliittymän koodi. Käyttöliittymän ulkoasua määritellään tarkemmin komponenttikohtaisella tyyli tiedostolla, joka voidaan kirjoittaa esimerkiksi CSS:llä. Komponenttikohtaisten tyyli tiedostojen lisäksi sovelluksessa voi olla kaikkiin komponentteihin yleisesti päteviä tyyli tiedostoja. Angularilla tehtävää käyttöliittymän kehitystä helpottamaan on luotu kirjasto Angular Material, joka sisältää lukuisia valmiita ja helposti muokattavia komponentteja nettisivujen tai -sovelluksen rakentamiseen. (Getting started with Angular 2021.)

3.3 Web-, työpöytä- ja hybridisovellukset

Työpöytäsovellukset ovat käytännössä tietokoneelle asennettavia ohjelmia. Työpöytäsovellus voi olla koneelle esiasennettu tai sellaisen voi ladata koneen omasta sovelluskaupasta tai esimerkiksi sovelluksen tarjoajan nettisivuilta. Perinteisempään tapaan työpöytäsovelluksen voi asentaa myös CD- tai DVD-levyltä. Tietokone, jolla sovellusta käytetään, voi olla kannettava tai pöytätietokone. Työpöytäsovellukset ovat eri sovellustyypeistä vanhimpia ja sellaisen kehittämiseen tänä päivänä vaikuttaa usein tarve käyttää joko paljon prosessointitehoa tai sellaisia tietokoneen komponentteja, joihin ei päästä web-tekniikoilla käsiksi. (Boterhoven 2021) Kehityskielen valintaan vaikuttaa sovellusta käyttävän laitteen käyttöjärjestelmä (käytännössä Windows, macOS tai Linux) (Nehra 2021).

Web-sovellukset ovat web-teknologioilla rakennettuja sovelluksia, jotka nimensä mukaisesti sijaitsevat netissä. Web-sovellusta voi käyttää millä laitteella tahansa käyttöjärjestelmästä riippumatta, sillä se toimii selaimessa ja siihen pääsee käsiksi syöttämällä verkko-osoitteen selaimen osoitekenttään. Eri selaimissa on kuitenkin eroavaisuuksia, jotka tulee ottaa huomioon sovellusta kehitettäessä. Koska web-sovellusta ei asenneta laitteelle, käyttäjän ei itse tarvitse huolehtia sovelluksen päivittämisestä. Huonona puolena web-sovelluksissa on se, että ne yleensä vaativat internet-yhteyden toimiakseen. (Sanderson 2020)

Hybridisovellus on kahden sovellustyyppin välimuoto - siinä yhdistyvät web-sovelluksen ja natiivin sovelluksen elementit. Kehitys tapahtuu web-teknologioilla, mutta lopputulos muistuttaa natiivia sovellusta. Hybridisovellus voi toimia mobiilissa tai tietokoneella, mutta tässä opinnäytetyössä kes-

kitytään juuri tietokoneille tarkoitettuihin, työpöytäsovellusta imitoiviin hybridisovelluksiin. Sanderson, 2020, kuvailee tällaisia web-teknologioilla käytettäviä työpöytäsovelluskehyskehyksiä nimellä ”desktop container”, sillä yksinkertaistettuna voisi sanoa, että niiden avulla rakennetut sovellukset ovat tavallaan web-sovelluksia natiivissa kuoressa. (Sanderson 2020)

3.4 Progressiivinen verkkosovellus PWA

PWA eli progressiivinen verkkosovellus (engl. progressive web app) on verkkosovellus, joka muistuttaa käyttökokemukseltaan natiivia sovellusta siinä mielessä, että perinteisestä verkkosovelluksesta poiketen sen voi asentaa mobiililaitteen aloitusnäytölle tai tietokoneen työpöydälle. Asennuksen jälkeen näytölle ilmestyy sovelluksen oma ikoni, josta sovellus aukeaa ilman näkyvää selainikkunaa kuten natiivi sovellus. Välimuistin ja PWA:lle ominaisen ”service workerin” avulla PWA:han saa lisättyä myös offline-toiminnallisuutta. Koska kyseessä on kuitenkin verkkosovellus, käyttökokemus ilman toimivaa internet-yhteyttä jää helposti vaillinaiseksi, eikä PWA:lla juurikaan päästä käsiksi itse laitteen käyttöjärjestelmän ominaisuuksiin, mikä on yksi sen suurimmista puutteista. (Sanderson 2020.)

PWA on trendinä melko uusi. Termi on saanut alkunsa v. 2015, kun sitä käytettiin kuvaamaan sovelluksia, jotka hyödynsivät modernien nettiselainten uusimpia ominaisuuksia, kuten service workereita ja web-manifesteja. Google alkoi ensimmäisenä edistää PWA-kehitystä Android-laitteille, minkä jälkeen Firefox, Microsoft ja lopulta myös Apple seurasivat perässä. Tietokoneen työpöydälle PWA:n on saanut asennettua vuodesta 2018 lähtien, ja nykyisin kaikki suurimmat nettiselaimet Firefoxia lukuun ottamatta tukevat PWA:ta, joskin tuetuissa ominaisuuksissa on eroja. PWA-sovellusten nousu on arvioitu yhdeksi vuoden 2022 suurimmista web-kehitystrendeistä. PWA siis kehittyy jatkuvasti ja näyttäisi olevan vain ajan kysymys, milloin PWA-sovelluksella pääsee paremmin käsiksi laitteen käyttöjärjestelmän ominaisuuksiin. (Rykov 2021.)

3.5 Sovelluskehys Electron

Electron on GitHubin kehittämä sovelluskehys, jonka avulla voi kehittää tietokoneen käyttöjärjestelmästä riippumattomia työpöytäsovelluksia web-teknologioilla. Tämä avoimen lähdekoodin sovelluskehys julkaistiin alun perin v. 2013 nimellä Atom Shell. Se oli tuolloin osa projektia, jonka

päämääränä oli kehittää uusi teksti- ja koodieditori Atom. Suosion kasvettua Atom Shell kuitenkin brändättiin pian uudelleen, ja nimeksi vaihtui Electron. (Griffith & Wells 2017, 2.)

Käytännössä saman web-teknologioilla rakennetun sovelluksen saa paketoitua toimimaan niin Mac-, Windows- kuin Linux-käyttöjärjestelmille. Sovelluskehittäjien ei siis tarvitse opetella natiiveja kieliä kääntääkseen sovelluksen jokaiselle käyttöjärjestelmälle erikseen, vaan Electron hoitaa asian paketoimalla sovelluksen halutulle käyttöjärjestelmälle sopivaksi. Tämä tietysti säästää kehittäjältä aikaa ja rahaa, sillä natiiveilla kielillä koodattujen sovellusten kustannukset voivat olla huomattavasti suuremmat niin sanottuun hybridisovellukseen verrattuna. (Kuprenko 2021.)

Hyviä esimerkkejä suosituista Electron-sovelluksista ovat Microsoft Teams, Visual Studio Code, Spotify, Discord, Skype ja Slack (Sanderson 2020; Quintana 2017). Näistä esimerkiksi Slack on ollut aluksi tavallinen verkkosovellus, johon haluttiin lisätä toiminnallisuuksia, joita verkkoratkaisu ei tukenut. Tällaisia toiminnallisuuksia olivat muun muassa näytön jakaminen, videopuhelut, natiivien ilmoitusten tarkka hallinta ja natiivit valikot (app menu, context menu). Siispä Slackin verkkoversiota päätettiin parannella Electronin avulla ja näin tehtäessä pystyttiin hyödyntämään jo olemassa olevaa koodipohjaa kirjoittamatta kaikkea uudelleen. (Quintana 2017)

3.5.1 Taustalla teknologiat Node.js ja Chromium

Electronissa yhdistyvät Node-ajoympäristön ja Chromium-selaimen teknologiat. Node.js tai lyhyemmin Node on JavaScriptin suorittamiseen tarkoitettu ajoympäristö, joka mahdollistaa JavaScriptin suorittamisen selaimen ulkopuolella. Se on kehitetty, jotta palvelinpuolen ohjelmointia voidaan suorittaa erillisen palvelinpuolen skriptikielen sijaan JavaScriptillä. Ensimmäinen versio Nodesta julkaistiin v. 2008 avoimen lähdekoodin projektina. (Griffith & Wells 2017, 2.) Node on alustariippumaton ja käyttää V8 JavaScript-moottoria. Noden asennuksen mukana tulee laaja kirjasto JavaScript-moduuleja, jotka helpottavat kehitysprosessia. Electron-sovelluksessa Node on vastuussa sovelluksen pääprosessista (englanniksi *main process*), jossa huolehditaan mm. sovelluksen elinkaaren toiminnoista, kuten käynnistyksestä ja sammuttamisesta. (Griffith & Wells 2017, 6.) Node myös mahdollistaa vuorovaikutuksen sovelluksen alla toimivan käyttöjärjestelmän kanssa. Esimerkiksi tietokoneelle tallennettuihin tiedostoihin pääsee käsiksi Noden file system-moduulin avulla (How Electron Works 2021).

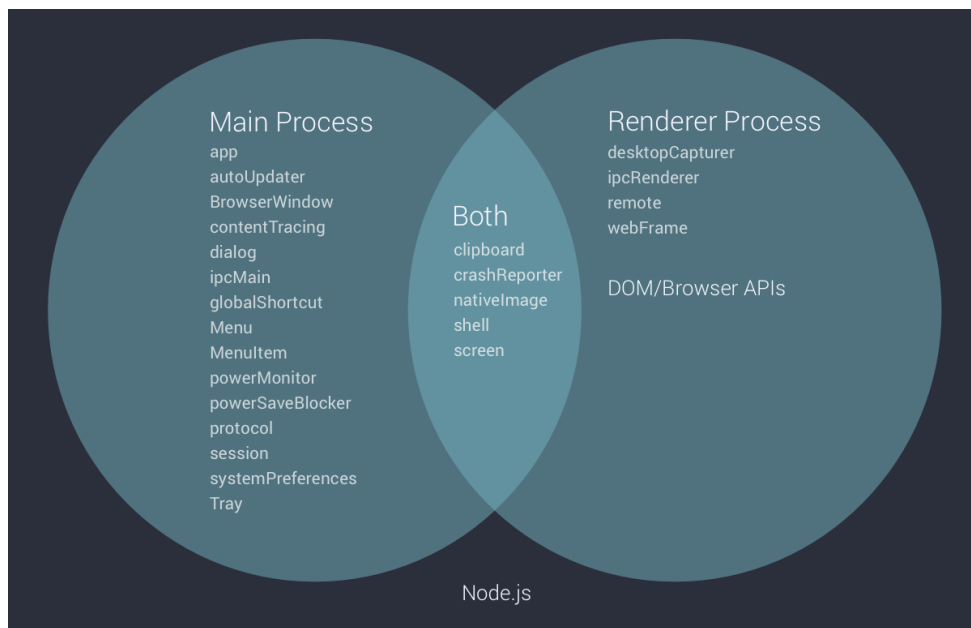
Chromium on pääasiassa Googlen kehittämä avoimen lähdekoodin verkkoselainprojekti. Griffith ja Wells kuvailivat v. 2017 Chromiumia Googlen Chrome-selaimen avoimen lähdekoodin versioksi. Chromiumista puuttuu Chromelle tyypillinen Google-brändäys, ja sen koodipohjaa hyödyntävät useat muut selaimet, kuten Opera, Microsoft Edge ja Samsung Internet. Chromiumin rooli Electron-sovelluksessa liittyy sovellusikkunoista ja käyttöliittymästä vastaavaan renderöijäprosessiin (englanniksi *renderer process*). Renderöijäprosesseja voi olla useita, ja kukin prosessi kommunikoi erikseen pääprosessin kanssa Electronin tarjoaman rajapinnan avulla. (Griffith & Wells 2017, 6.)

3.5.2 Kahden prosessin rakenne

Electron-sovelluksen toiminnan keskiössä on kaksi edellisessä luvussa mainittua prosessia: pääprosessi ja renderöintiprosessi. Pääprosesseja on aina vain yksi. Sen paikka on päätiedostossa, joka puolestaan on määritelty sovelluksen package.json-tiedostossa. Kun Electron-sovellus käynnistyy, ajetaan package.json-tiedoston main-kohdassa määritelty skripti, joka luo pääprosessin. Pääprosessin tehtäviin kuuluu huolehtia sovelluksen elinkaaresta ja vuorovaikutuksesta käyttöjärjestelmän kanssa. Lisäksi pääprosessista käsin luodaan sovelluksen ikkunat, mikä tapahtuu päätiedostossa BrowserWindow-moduulin avulla. Electronin verkkosivujen dokumentaation mukaan BrowserWindow-moduuli on koko Electron-sovelluksen perusta, sillä sen API:en avulla pystytään muokkaamaan sovellusikkunoiden ulkoasua ja toimintoja (Window Customization 2021). Näin ollen pääprosessi käytännössä hallitsee kaikkia sovellusikkunoita ja niiden renderöintiprosesseja. (How Electron Works 2021.)

Jokaisella pääprosessista luodulla sovellusikkunalla on oma renderöintiprosessi, joka selaimen väli-lehden tapaan ottaa HTML-tiedoston, hyödyntää CSS- ja JavaScript-tiedostoja sekä mahdollisia kuvia ym. tiedostoja ja renderöi tuloksen näkyviin sovellusikkunaan. Kun kyseisen sovellusikkunan BrowserWindow-instanssi tuhotaan, tuhoutuu myös sen renderöintiprosessi. (How Electron Works 2021.) Vaikka ilman renderöintiprosesseja sovelluksella ei ole yhtään sovellusikkunaa, muistuttavat Griffith ja Wells kirjassaan Electron: From Beginner to Pro: Learn to Build Cross Platform Desktop Applications using Github's Electron, 2017, että periaatteessa Electron-sovelluksessa ei ole pakko olla yhtään renderöintiprosessia, joskin se on hyvin epätodennäköistä.

Prosessien toiminnan lisäksi kehityksen kannalta on tärkeää tietää, mistä prosessista käsin Electronin API:t ovat käytettävissä. Kuten kuviosta 1 näkee, esimerkiksi leikepöytä on saavutettavissa sekä pää- että renderöintiprosessista käsin, kun taas sovelluksen elinkaaren hallintaan käytettävä app-moduuli on käytettävissä vain pääprosessissa. Node.js on kuvassa kokonaan ympyröiden ulkopuolella, sillä sen API:t ovat käytettävissä globaalisti. (Nokes 2016.)



Kuvio 1 Electron-API:t ja niiden käytettävyys prosesseissa (Nokes 2016)

3.5.3 Tärkeimmät moduulit

Electronin nettisivujen dokumentaatioista ilmenee, että Electron-kirjastoon kuuluu pitkä lista erilaisia kehittäjän käytettävissä olevia moduuleita. Tällaisia ovat muun muassa moduulit BrowserWindow, webContents, ipcMain ja ipcRenderer. Aiemmissa luvuissa mainittu BrowserWindow-moduuli ja erityisesti sen ominaisuus webContents tarjoavat kehittäjälle avaimet käyttöliittymän hallintaan. Electronin dokumentaatiosta voi päätellä, että BrowserWindow-objekti vastaa sovellussikkunasta kokonaisuutena ja erityisesti ikkunan kehyksistä, kun taas webContents vastaa siitä, mitä ikkunan sisällä tapahtuu. (Inter-Process Communication 2021.) Nokes (2016) puolestaan tarkentaa blogissaan, että jokaisen ikkunan sisältö on instanssi webContentsista, eikä renderöintiprosessia ole ilman tällaista instanssia (Nokes 2016). Seuraavassa koodiesimerkissä kuvio 2:ssä luodaan BrowserWindow-moduulia käyttäen ikkuna, määritellään sen koko ja ladataan siihen index.html-sivun sisältö.

```

let mainWindow

function createWindow() {
  mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  })

  mainWindow.loadURL(
    url.format({
      pathname: path.join(__dirname, `index.html`),
      protocol: "file:",
      slashes: true
    })
  );
}

```

Kuvio 1: Esimerkki sovellusikkunan luonnista Electronilla

Toinen keskeinen osa työpöytäsovellusten rakentamista Electronilla on pää- ja renderöintiprosessien välinen kommunikointi ipcMain- ja ipcRenderer-moduulien avulla. Moduulien nimissä näkyvä ipc-lyhenne tarkoittaa kirjaimellisesti prosessien välistä kommunikaatiota (englanniksi inter process communication). Näitä moduuleita tarvitaan esimerkiksi silloin, kun halutaan kutsua jotakin pääprosessin API:a käyttöliittymästä eli renderöintiprosessista käsin – vaikkapa natiivin dialogin aukeaminen käyttäjän painaessa painiketta. Prosessilta toiselle lähetetään viestejä kehittäjän itse nimeämän kommunikointikanavan (englanniksi IPC-channel) kautta. Esimerkiksi yksisuuntainen viesti renderöintiprosessista pääprosessille onnistuu ipcRenderer.send-API:n avulla, ja viestin vastaanottaminen pääprosessissa ipcMain.on-API:n avulla. Viestintä voi olla myös kaksisuuntaista, kun halutaan pääprosessilta vastaus renderöintiprosessista käsin lähetettyyn viestiin. (Inter-Process Communication 2021.)

3.5.4 Tietoturva ja muut haasteet

Electron-sovelluskehiksestä lukiessa ei voi välttyä kommenteilta liittyen sovellusten suureen kokoon. Suuri koko johtuu siitä, että Electron-sovellukset yhdistävät Node.js-ympäristön ja Chromium-selaimen, joka jo itsessään vie suhteellisen paljon tilaa tietokoneen kiintolevyiltä. Yksinkertainen Electron-sovellus vie helposti 120 Mt tilaa. (Is Electron the Best Desktop Framework to Use in 2021? 2021) Sovelluksen kokoon voi kuitenkin vaikuttaa, joskin keinoja voi olla vaikea löytää.

Ainakin sovelluksen koontiversion luontiin käytetyllä kirjastolla on merkitystä, sillä niiden asetusmahdollisuuksissa on koontiversion kokoon vaikuttavia eroja.

Toinen seikka, joka usein mainitaan Electronin huonoista puolista kerrottaessa, on sovellusten käyttämä keskusmuisti. Electron-sovellus voi käyttää satoja megabittejä keskusmuistia ja sen sanotaan vaativan paljon tietokoneen prosessorilta, mikä puolestaan kuluttaa esimerkiksi kannettavan tietokoneen akkua nopeammin. (Is Electron the Best Desktop Framework to Use in 2021? 2021)

Yleisin vasta-argumentti Electronin käytölle vaikuttaa olevan tietoturvariskit. Electronin dokumentaation mukaan Electron-sovellusten kyky päästä käsiksi käyttäjän tietokoneen tiedostojärjestelmään ja muihin ominaisuuksiin tuo mukanaan tietoturvariskejä, jotka kehittäjän on otettava huomioon. On tärkeää käyttää viimeisintä versiota Electronista, arvioida riippuvuuksien, kuten kolmannen osapuolen kirjastojen turvallisuutta ja käyttää turvallisiksi todettuja koodauskäytäntöjä. Erityistä huomiota on kiinnitettävä ulkopuolista materiaalia ladattaessa – missään tapauksessa ei saisi ladata ja ajaa ulkopuolista koodia niin, että Node-integraatio on päällä, vaan muistaa aina asettaa ”nodeIntegration”-asetus pois päältä ja ”contextIsolation”-asetus päälle. Kehittäjän kannattaa tutustua huolellisesti Electronin verkkosivujen dokumentaatiosta löytyvään tietoturvaosioon. (Security 2021)

4 Kehitysprosessi

Tässä osio keskittyy opinnäytetyön toiminnalliseen osuuteen eli POC-sovelluksen kehittämiseen. Alussa kerrotaan, miten Electron ja Angular otetaan käyttöön ja miten ne saadaan toimimaan yhdessä. Jo aiemmin luvussa 3.5.2 kerrottiin Electronin kahden prosessin rakenteesta teoriatasolla, ja tässä luvussa käsitellään pää- ja renderöintiprosessien toimintaa käytännössä Angularin kanssa. Kehitysprosessin lopuksi konseptisovellus pakataan käyttäjän tietokoneelle asennettavaan muotoon.

4.1 Electronin käyttöönotto yleisesti

Tässä luvussa käydään läpi, mitä sovelluskehittäjän tulee osata entuudestaan ennen Electronin käyttöä, ja mitä koneelle täytyy asentaa käyttöönottoa varten. Kehittäjällä täytyy olla osaamista HTML:stä, CSS:stä ja JavaScriptistä (tai TypeScriptistä, jos käytetään kehyksenä esim. Angularia)

sekä jonkin asteinen ymmärrys Nodesta. Lisäksi olisi hyvä osata käyttää versionhallintaa kuten Git:iä, mikä on toki kaikille koodareille olennainen taito. Natiiveja koodauskieliä ei tarvitse opetella. (Griffith & Wells 2017, 3.)

Jotta Electron toimisi kehittäjän tietokoneella, on ensin asennettava Node.js. Electronin verkkosivujen Quick Start-oppaassa suositellaan asentamaan viimeisin Node-versio käyttämällä Noden omilta verkkosivuilta nodejs.org löytyvää, omalle käyttöjärjestelmälle tarkoitettua asennusohjelmaa yhteensopivuusongelmien välttämiseksi. Oman tietokoneen Node-version voi tarkistaa kirjoittamalla komentokehoteeseen **node -v**. Jos projekti on uusi, on siihen luotava package.json-tiedosto komennolla **npm init**. Package.json-tiedoston "main"-kohtaan tulee Electron-sovelluksen pää tiedosto, esimerkiksi näin: **"main": "main.js"**. (Quick Start 2021.)

Noden asennuksen jälkeen voidaan asentaa itse Electron käyttämällä paketinhallintajärjestelmää kuten edellä mainittu npm eli node package manager. Asennus tehdään kehittäjäriippuvuutena komennolla **npm install --save-dev electron**. Jotta Electron-sovelluksen pystyy käynnistämään kehitystilassa, on projektin package.json-tiedoston scripts-osioon lisättävä käynnistyskomento **"start": "electron ."**. Nyt sovelluksen voi käynnistää kehitystilassa komennolla **npm start**. (Quick Start 2021.)

4.2 Kehitys Angularin kanssa

Selkeyden vuoksi kansiorakenne kannattaa pohtia niin, että Electronin koodi on omassa kansiossaan ja Angularin koodi omassaan. Koska Electron ei ymmärrä typescriptiä, jota käytetään Angularilla kehitettäessä, on Angular-sovellus buildattava, jotta sen voi näyttää Electronin natiivin näköisessä ikkunassa. Tässä projektissa buildaaminen ja sovelluksen käynnistys toteutettiin skriptillä **"ng build --base-href ./ && tsc --p electron && electron electron/dist/main.js"**. Jatkuva buildaaminen kuitenkin hidastaa kehitysprosessia tavanomaiseen Angular-kehitykseen verrattuna melkoisesti. Avuksi voi ladata esimerkiksi nodemoduulit wait-on ja concurrently (Darida 2019).

Tässä projektissa käytettiin yleisesti Angularilla kehitettäessä käytettävää komentokehotetyökalua Angular cli:tä, joka asennetaan komennolla **npm install -g @angular/cli**. Version voi sen jälkeen tarkistaa komennolla **ng -v**. Projektissa käytettiin Angularin versiota 12, koska projektin

kehitys alkoi maaliskuussa 2022. Cli:n asennuksen jälkeen voidaan luoda itse Angular-projekti komennolla **ng new SovelluksenNimi**. Juuri luotua aloitussivua pääsee tarkastelemaan selaimessa komennolla **ng serve**.

Kuvion 3 koodissa näkyy, miten Angular-projekti saadaan näkyviin Electron-ikkunassa Electron-kansion main.ts-tiedostossa. Koodissa luodaan funktio, jossa win-muuttujaan asetetaan uusi BrowserWindow-instanssi, jolle annetaan parametreina ikkunan korkeus ja leveys pikseleinä, sekä webPreferences-olioon asetukset nodeIntegration true ja contextIsolation false. Tietoturvan kannalta nämä kaksi boolean-muuttujaa kannattaa myöhemmin kääntää toisin päin, mutta kehitysversiossa ne voivat aluksi olla näin. Kohdassa win.loadUrl ikkunalle annetaan polku Angular-sovelluksen index-tiedostoon, joka on tässä tapauksessa dist-kansiossa. Jotta Electron voi käynnistyessään löytää ja ladata onnistuneesti tarvittavat tiedostot, on kyseisen index.html-tiedoston <base href="/"> -polku päivitettävä muotoon <base href=".">. Kehittämisen avuksi koodissa avataan selaimen kehittäjätyökalut webContents-moduulin openDevTools-funktiolla.

```
import { app, BrowserWindow, ipcMain, Menu, dialog, webContents } from 'electron';
import * as path from 'path';
import * as url from 'url';

function createWindow() {
  win = new BrowserWindow({
    width: 1000,
    height: 800,
    webPreferences: {
      nodeIntegration: true,
      contextIsolation: false,
    },
  });

  win.loadURL(
    url.format({
      pathname: path.join(__dirname, `../../dist/AngTextEditor/index.html`),
      protocol: 'file:',
      slashes: true,
    })
  );

  win.webContents.openDevTools();

  win.on('closed', () => {
    win = null;
  });
}
```

```

}

app.on('ready', createWindow);

app.on('activate', () => {
  if (win === null) {
    createWindow();
  }
});

// Quit when all windows are closed.
app.on('window-all-closed', () => {
  // On macOS it is common for applications and their menu bar
  // to stay active until the user quits explicitly with Cmd + Q
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

```

Kuvio 3: Angular-projektin avaaminen Electronin sovellusikkunaan main.ts-tiedostossa

4.3 Prosessien välinen kommunikaatio käytännössä

Osana tätä POC-sovellusta toteutettiin yksinkertainen tekstieditori, jolla käyttäjä voi luoda tekstitiedoston tai avata tietokoneelta valmiiksi olemassa olevan tekstitiedoston, muokata sen sisältöä ja tallentaa sen. Toiminnallisuus toteutettiin Electron-ikkunan vasemmasta yläkulmasta löytyvää menua käyttäen. Kuviossa 4 luodaan menu tiedostonavaus- ja tallennustoimintoihin edellä mainittuun main.ts-tiedostoon. Koodissa fs viittaa tietokoneen tiedostojärjestelmään (file system). Kun käyttäjä on klikannut jotakin menun vaihtoehtoista, lähetetään asynkroninen viesti renderöintiprosessille webContents.send-API:a hyödyntäen.

```

import * as fs from 'fs';

let chosenFilepath: string;

const template = [
  {
    label: 'File',
    submenu: [
      {
        label: 'Open File',
        accelerator: 'Ctrl+O',
        click: async () => {
          const { filePaths } = await dialog.showOpenDialog({

```

```

        properties: ['openFile'],
        filters: [{ name: 'Text', extensions: ['.txt'] }],
    });

    const file = filePaths[0];
    const contents = fs.readFileSync(file, 'utf-8');
    const openedObject = {
        contents: contents,
        filePath: file,
    };
    win.webContents.send('fileOpened', openedObject);
    // enabling Save file option below
    const saveFileItem = menu.getMenuItemById('save-file');
    saveFileItem.enabled = true;
},
{
    id: 'save-file',
    enabled: false,
    accelerator: 'Ctrl+S',
    label: 'Save',
    click: async () => {
        win.webContents.send('saveFileClicked', 'Save file clicked!');
    },
},
{
    id: 'save-file-as',
    enabled: true,
    accelerator: 'Ctrl+Shift+S',
    label: 'Save As...',
    click: async () => {
        await dialog
            .showSaveDialog({
                properties: ['createDirectory'], // only on Mac
                filters: [{ name: 'Text', extensions: ['.txt'] }],
            })
            .then((val) => {
                chosenFilePath = val.filePath;
                win.webContents.send('saveAsClicked', val.filePath);
                //enable save after naming new file
                const saveFileItem = menu.getMenuItemById('save-file');
                saveFileItem.enabled = true;
            });
    },
},
],
},
];

const menu = Menu.buildFromTemplate(template);

```

```
Menu.setApplicationMenu(menu);
```

Kuvio 4: Menu ja datan lähetys pääprosessista renderöintiprosessille

Kun menu toimintoinen on luotu koodin Electron-puolella, voitiin Angularin puolella reagoida webContentsin kautta lähetettyihin viesteihin. Tätä varten luotiin file-service Angular cli:n komennolla **ng generate service file**. Servicessä kuunnellaan viestejä ipcRenderer.on-API:n avulla. Pääprosessista viestin mukana lähetetty tiedosto-olio asetetaan servicen konstruktorissa rxjs-kirjaston tarjoamaan BehaviorSubject-muuttujaan sen next-metodia kutsumalla. Tiedoston sisältöä ja tiedostopolkua voidaan tarkastella myöhemmin Angularin komponenteissa file-serviceä kutsuamalla. Servicen kautta myös lähetetään dataa renderöintiprosessista pääprosessille ipcRenderer.send-API:n avulla kuvio 5:n loppupuolella sendNewContents-funktiossa. Funktiota kutsutaan silloin, kun halutaan tallentaa käyttäjän lisäämä syöte käyttöliittymästä tekstitiedostoon.

```
import { Injectable, NgZone } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
import { TextFile } from './models/textfile.model';
const electron = (<any>window).require('electron');

@Injectable({
  providedIn: 'root',
})
export class FileService {
  fileObject = new BehaviorSubject<TextFile>({ contents: '', filePath: '' });

  constructor(private ngZone: NgZone) {
    electron.ipcRenderer.on(
      'fileOpened',
      (event: any, openObject: TextFile) => {
        this.ngZone.run(() => {
          this.fileObject.next(openObject);
        });
      }
    );

    electron.ipcRenderer.on('saveFileClicked', (event: any, text: string) => {
      this.sendNewContents();
    });

    electron.ipcRenderer.on(
      'saveAsClicked',
      (event: any, chosenFilepath: string) => {
        this.ngZone.run(() => {
          this.newFilepath(chosenFilepath);
        });
      }
    );
  }
}
```

```

    });

    this.sendNewContents();
  }
});

newFilepath(chosenFilepath: string) {
  this.fileObject.next({
    ...this.fileObject.value,
    filePath: chosenFilepath,
  });
}

sendNewContents() {
  electron.ipcRenderer.send('newContents', this.fileObject.value);
}
}

```

Kuvio 5: file.service.ts-tiedosto kokonaisuudessaan

Sovelluksen käyttöliittymää varten luotiin editor-komponentti Angular cli:llä komennolla **ng generate component editor**. Komponentin templaattiin lisättiin yksinkertainen tekstinsyöttökenttä, johon käyttäjä voi kirjoittaa uuden tiedoston sisällön tai muokata menun kautta avaamansa tiedoston sisältöä. Kuten kuviossa 6 näkyy, file-servicen tallessa pitämä tiedosto-olio fileObject tuodaan templaattiin Angularin async-putken avulla, ja textarea-kenttä saa sisältönsä kyseisen olion contents-ominaisuudesta. Käyttäjän kirjoittama sisältö tallentuu contentsiin [(ngModel)]-direktiivillä.

```

<div *ngIf="this.fileService.fileObject | async as fileobject">
  <h2>Text File Editor</h2>
  <h3>{{fileobject.filePath}}</h3>
  <textarea id="code" [(ngModel)]="fileobject.contents" (keyup)="storeContents(fileobject)">{{fileobject.contents}}</textarea>
</div>

```

Kuvio 6: Templaatin koodi tiedostossa editor.component.html

Editor-komponentin typescript-puoli on myös yksinkertainen ja näkyvissä kuviossa 7. Komponentissa määritellään storeContents-funktio, joka nimensä mukaisesti tallettaa tekstitiedoston sisällön. Tämä tehdään jälleen file-servicessä olevan behaviorSubjectin next-metodin avulla. Muutosten havaitsemisen (change detection) toiminnan varmistamiseksi service-kutsu laitetaan Angularin core-moduulista tuodun NgZone-servicen run-metodin sisään.

```

import { Component, NgZone } from '@angular/core';
import { FileService } from '../file.service';
import { TextFile } from '../models/textfile.model';
import { NgModel } from '@angular/forms';

@Component({
  selector: 'app-editor',
  templateUrl: './editor.component.html',
  styleUrls: ['./editor.component.css'],
})
export class EditorComponent {
  constructor(
    public fileService: FileService,
    private ngZone: NgZone
  ) {}

  storeContents (contents: TextFile) {
    if (contents) {
      this.ngZone.run(() => {
        this.fileService.fileObject.next(contents);
      })
    }
  }
}

```

Kuvio 7: Editor-komponentin koodi tiedostossa editor.component.ts

Jotta tiedoston tallentaminen toimisi, muokataan vielä hieman main.ts-tiedostoa kuvion 8 tapaan. Luodaan funktio saveFile, joka ottaa parametreikseen tiedostopolun ja käyttäjän lisäämän tekstisisällön. Aikaisemmin tiedostossa käytettiin tiedostojärjestelmän funktiota fs.readFileSync, joka avasi tiedoston, kun taas nyt kutsutaan fs.writeFileSync-funktiota, joka tallentaa tiedoston tietokoneelle. SaveFile-funktiota on kutsuttava pääprosessista kuuntelemalla kanavaa newContents, johon lähetettiin dataa aiemmin file-servicellä ipcRenderer.send-API:lla. Nyt samaa kanavaa siis kuunnellaan pääprosessissa ipcMain.on-API:lla.

```

function saveFile(path: string, contents: string) {
  try {
    fs.writeFileSync(path, contents, 'utf-8');
  } catch (err) {
    console.error(err);
  }
}

ipcMain.on('newContents', (event: any, newFile: any) => {
  saveFile(newFile.filePath, newFile.contents);
});

```

Kuvio 8: Viestin vastaanottaminen main.ts-tiedostossa

4.4 Sovelluksen paketointi

Electronin dokumentaatiossa suositellaan käyttämään sovelluksen paketoimiseen Electronin omaa työkalua Electron Forge. Electron Forgen asennus tapahtuu komennolla **npm install --save-dev @electron-forge/cli**. Asennuksen jälkeen ajetaan skripti **npm exec --package=@electron-forge/cli -c "electron-forge import"**, joka luo konfiguraatiotiedoston `forge.config.js` ja valmistelee Electron Forgen sovelluksen paketointia varten. Edellä mainittujen komentojen ajamisen jälkeen `package.json`-tiedoston on `scripts`-kohtaan on ilmestynyt kuvio 9:ssä näkyvät skriptit.

```
"start": "electron-forge start",  
"package": "electron-forge package",  
"make": "electron-forge make",  
"publish": "electron-forge publish",
```

Kuvio 9: Electron forgen skriptit `package.json`-tiedostossa

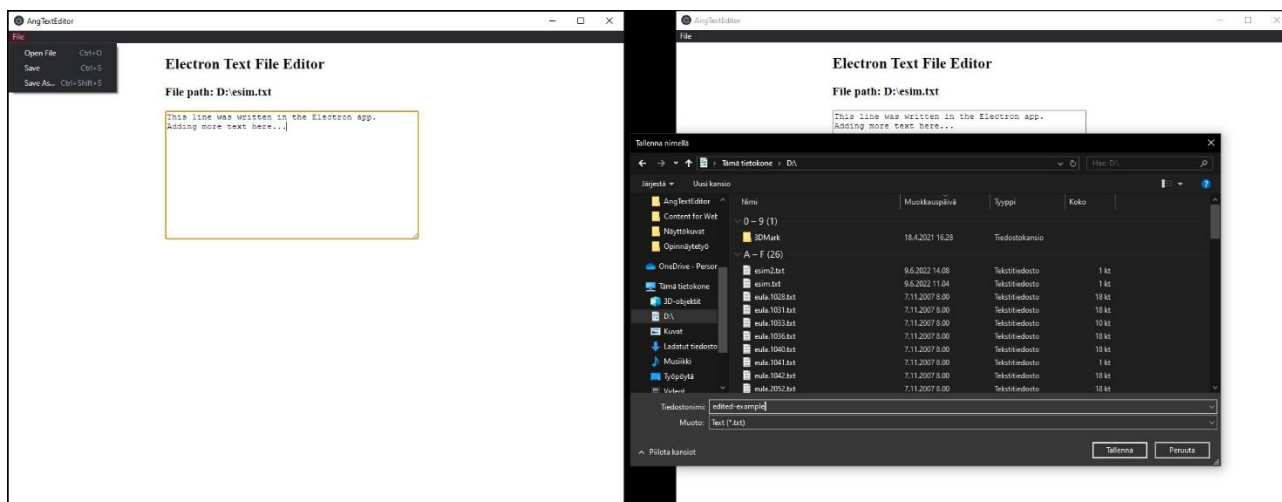
Jos skriptin ajaminen ei onnistu, voi tarvittavien pakettien asennuksen ja konfiguraatiot tehdä manuaalisesti. Tällöin `forge.config.js`-tiedosto sisältöineen täytyy luoda itse ja `Main.ts`-tiedostoon tulee myös lisätä rivi `"if (require('electron-squirrel-startup')) app.quit();"` tiedoston alkupäähän ennen `"app.on('ready' ... "` -lausetta. Kun Electron Forge on konfiguroitu onnistuneesti joko skriptillä tai manuaalisesti, tulisi paketoitun version valmistaminen onnistua kuvio 9:ssä näkyvällä skriptillä, jonka voi ajaa komennolla **npm run make**.

Electron Forgen ja Electronin `autoUpdater`-moduulin avulla voi myös julkaista sovelluksen ja suorittaa sille automaattisia päivityksiä. Julkaisut määritetään `forge.config.js`-tiedoston `publishers`-osassa. Electron Forgen dokumentaation mukaan automaattisten päivitysten järjestämiseen on kolme vaihtoehtoa: ensinnäkin Githubissa julkaistuille avoimen lähdekoodin sovelluksille käytettävissä oleva maksuton palvelu `update.electronjs.org`, toiseksi päivitykset staattisen tallennustilan tarjoajille kuten Amazon S3-bucketeille, tai kolmanneksi oman päivitysserverin hostaaminen (Auto Update 2024).

5 Tulokset

Opinnäytetyön pääasiallisena tavoitteena oli selvittää, mikä on Electron ja lisäkysymyksenä pohtia miten se eroaa PWA:sta. POC-sovelluksen kehittämisen jälkeen voidaan todeta, että Electronia voi käyttää yhdessä Angularin kanssa ja web-teknologioita hyödyntäen aikaan voi saada natiivilta

näyttävän sovelluksen, jolla on sovellusmenu (kuten kuviossa 10) ja pääsy esimerkiksi tietokoneen tiedostojärjestelmään. Opinnäytetyössä kuvatus sovelluksen lisäksi kehitysvaiheessa kokeiltiin rakentaa muitakin pieniä esimerkkisovelluksia, joilla pystyi mm. monitoroimaan tietokoneen prosessoria. Kuten Kuprenko (2021) mainitsi blogissaan, web-kielillä aikaansaadun sovelluksen pystyi lähinnä yhden työkalun ja konfiguraatiotiedoston avulla paketoimaan useille eri käyttöjärjestelmille toimivaksi (tämä testattiin käytännössä Windows- ja Mac-tietokoneilla).



Kuvio 10: POC-sovelluksen menu avataan

Mitä tulee Electronin hyviin ja huonoihin puoliin, parhaiksi puoliksi voisi todeta sen natiivin ulkoasun ja käyttökokemuksen, jotka voidaan saavuttaa web-kehityksestä tutuilla ohjelmointikielillä kuten JavaScript ja TypeScript. Alustariippumattomuuden vuoksi ohjelmoijan ei tarvinnut opetella laitekohtaisia natiiveja kieliä. Esimerkkisovellus ei myöskään vaatinut toimiakseen internet-yhteyttä. Electronin verkkosivujen tutoriaalissa Inter-Process Communication (2021) todetaan ipcMain- ja ipcRenderer-moduulien avulla hallittavien prosessien välisen kommunikaation olevan keskeinen osa Electron-sovelluksen kehitystä, mikä todistettiin POC-sovellusta kehitettäessä. Pääprosessin ja renderöintiprosessin välinen kommunikaatio oli nopea oppia ja suoraviivaista toteuttaa. Angularin servicet todettiin tähän tarkoitukseen sopiviksi.

Jackson (2019) mainitsi artikkelissaan asennettavan sovelluksen suuren koon yhdeksi Electronin haittapuolista, ja tämä todettiin myös tutkimuksen kehitystyön aikana. Hyvin yksinkertaisista ja pienistä sovelluksista tuli paketoituna kooltaan 180–500 Mb. Suuresta koosta ja tietokoneen kes-

kusmuistia syövästä raskaudesta löytyikin useita mainintoja opinnäytetyön teoriavaiheessa tutkittuista lähteistä. Oletettavasti nämä johtuvat Electronin renderöintipuolen taustalla hyödyntämästä Chromium-selaimesta, joka on tavallaan paketoituna jokaiseen lopulliseen Electron-sovellukseen.

PWA-sovelluksiin verrattaessa lopputuotteen kokoero on merkittävä. PWA:lla voi lisäksi kehittää samalla kertaa sovelluksen myös mobiililaitteille, kun taas Electron-sovellukset soveltuvat vain tietokoneille. PWA:n voisi sanoa soveltuvan paremmin pieniin ja yksinkertaisiin sovellustarkoituksiin, kuten tässä opinnäytetyössä rakennettu esimerkkisovellus. Alustariippumattomuus vähentää kehitysaikaa ja siten kuluja natiiveihin sovelluksiin verrattaessa, mutta tämä puoli on toki myös PWA-sovelluksissa. Electronilla on kuitenkin varmasti puolensa niissä työpöytäsovelluksissa, joissa toivotaan natiivia tuntua ja vaaditaan pääsyä sellaisiin käyttöjärjestelmän rajapintoihin, joita PWA ei tue. Riippuu siis täysin käyttötarkoituksesta, kumpi on parempi valinta.

6 Pohdinta

6.1 Luotettavuus ja eettisyys

Opinnäytetyön luotettavuuteen vaikuttaa muun muassa teoriaosuuteen käytetyt lähteet ja kehittämistyön laatu. Lähdekirjallisuutta oli saatavilla vain yhden teoksen verran, joten suurin osa lähteistä on nettialustoilla julkaistuja artikkeleita ja blogitekstejä, sekä sovelluskehysten omia dokumentaatioita. Koska aiheesta julkaistu kirja oli jo useita vuosia vanha, sitä hyödynnettiin lähinnä sovelluskehysten historiasta ja teoriasta kertoviin osuuksiin. Kehityksen tukena hyödynnettiin erityisesti dokumentaatioita, jotka ovat lähteistä eniten ajantasaisia. Hybridisovelluksista kertovista artikkeleista ja blogiteksteistä sai käsitystä siitä, mitä yleisesti pidetään Electron- ja PWA-tekniikoiden hyötyinä ja haittoina, ja näihin seikkoihin pyrittiin kiinnittämään huomiota kehitysosion tuloksissa. Kaikista käytetyistä lähteistä dokumentaatiot ja kirjallisuus lienevät luotettavimpia. Muihin lähteisiin voi suhtautua kriittisemmin.

Kehitysosiossa dokumentoitiin yhden POC-sovelluksen kehitys, ja sovellus jäi kovin yksinkertaiseksi, jolloin kehitysosion anti on varsin niukka. Kehitetyn sovelluksen natiivinkaltaisia ominaisuuksia olivat sovellusmenu ja pääsy tiedostojärjestelmään. Tuloksesta saisi luotettavamman ja hyödyllisemmän, kun kehittäisi monipuolisemman sovelluksen. Lisäominaisuudet vaatisivat kuitenkin lisää kehitystä ja tarkempaa perehtymistä aiheeseen. Yksi suurimpia haasteita kehityksessä oli

nimittäin sopivien lähteiden ja syväluotaavampien esimerkkien löytäminen Angularin ja Electronin yhteen sovittamisen kanssa. Opinnäytetyön kirjoittajalla ei ollut aiempaa kokemusta Electronista.

6.2 Ideoita jatkokehitykseen

POC-sovellusta voisi kehittää eteenpäin lisäämällä siihen toimintoja ja tekemällä siitä siten monipuolisemman. Sovellukseen voisi ottaa mukaan esimerkiksi käyttöjärjestelmän omat ilmoitukset (system notifications) tai näytönjakamisominaisuuden, joilla voisi demonstroida enemmän Electronin pääsyä käsiksi tietokoneen natiiveihin rajapintoihin. Electron-sovelluksen kehityksessä on myös tietoturvaan liittyviä huomioonotettavia asioita, joihin voisi paneutua erikseen vaikka omana aiheenaan.

Sovelluksen kokoa ja tietokoneen resurssien kuluttamista olisi voitu tarkkailla paremmin kehittämällä sovellukseen useampi näkymä, sillä nämä vaatisivat useita renderöintiprosesseja, joiden on sanottu olevan Chromium-selainalustan takia raskaita. Vertailua PWA:han parantaisi se, että kehittäisi mahdollisimman vastaavanlaisen sovelluksen PWA:na Angularia käyttäen. Toki kaikkea, mihin Electron pystyy, ei voi toteuttaa PWA:lla, mutta esimerkiksi tietokoneen tiedostojärjestelmään pääsee nykyisin käsiksi myös PWA-sovelluksella.

6.3 Johtopäätökset

Tehtäessä päätöstä Electronin ja PWA:n välillä on oltava hyvin selvillä sovelluksen käyttötarkoitukset ja siihen halutut ominaisuudet. Jos sovellukseen tulevista ominaisuuksista ei ole vielä varma, voi PWA olla parempi valinta. On tiedettävä myös, millä alustoilla sovellusta halutaan käyttää. Monissa tapauksissa PWA on sopivampi valinta, sillä se taipuu niin moneen käyttökohteeseen mobiililaitteista tietokoneille, kun taas Electronilla on selkeämpi paikka natiivinkaltaisissa työpöytäsovelluksissa. Toisaalta jos on ensin kehittänyt sovelluksen pelkillä web-teknologioilla ja tahtoisii siihen myöhemmin Electronin tuomia ominaisuuksia, voi sen siirtäminen Electron-kehitykseen olla haastavaa.

Electron-sovellus voi sisällöstään riippuen olla raskas pyörittää ja asennettavan sovellusten koko kasvaa helposti suureksi. Sovelluksen tietoturvaan on myös syytä kiinnittää erityistä huomiota. Jos

edellä mainitut asiat eivät ole ongelma ja kehitettävän sovelluksen käyttötarkoitus ja siihen tulevat ominaisuudet ovat selkeitä ja Electronin tarjoamia ominaisuuksia kaivataan, voi Electron olla oikea valinta. Angular-sovelluskehys tuo sovelluksen frontend-puoleen rakennetta ja on mahdollista saada toimimaan hyvin yhdessä Electron-kehysten kanssa.

Lähteet

Auto Update. Electron Forge-työkalun verkkosivujen dokumentaatio. Viitattu 19.4.2024. <https://www.electronforge.io/advanced/auto-update>.

Boterhoven, D. 2021. Web, Mobile or Desktop App – Which Is Right for Your Project? Blogi. Viitattu 24.5.2022. <https://denimdev.com.au/blog/web-mobile-or-desktop/>.

Bouchefra, A. 2019. Build a Desktop Application with Electron and Angular. Viitattu 8.2.2022. <https://www.sitepoint.com/build-a-desktop-application-with-electron-and-angular/>.

Darida, R. 2019. Electron and Angular Live Reload. Artikkele Medium-verkkosivuilla. Viitattu 15.4.2024. <https://rdarida.medium.com/electron-angular-live-reload-13ebc9808bb5>.

Getting started with Angular. 2021. Angular.io verkkosivut. Viitattu 21.2.2022. <https://angular.io/start>.

Griffith, C. & Wells, J. 2017. Electron: From Beginner to Pro: Learn to Build Cross Platform Desktop Applications using Github's Electron. Berkeley, CA: Apress.

How Electron Works. 2021. Tutorialspoint verkkosivut. Viitattu 17.2.2022. https://www.tutorialspoint.com/electron/how_electron_works.htm.

HTML Living Standard. 2022. WHATWG. Viitattu 16.2.2022. <https://html.spec.whatwg.org/multipage/>.

Inter-Process Communication. 2021. Electronin verkkosivut. Viitattu 12.4.2022. <https://www.electronjs.org/docs/latest/tutorial/ipc>.

Jackson, J. 2021. Is Electron the Best Desktop Framework to Use in 2021? Artikkele Medium-verkkosivuilla. Viitattu 25.5.2022. <https://javascript.plainenglish.io/is-electron-the-best-desktop-framework-to-use-in-2021-e525638b9b6a>.

Jyväskylän ammattikorkeakoulu. 2024. Tutkimuksellinen kehittämistyö (AMK ja YAMK). Viitattu 15.4.2024. [https://help.jamk.fi/opinnaytettyo/fi/toteutustavat-ja-rakenne/tutkimuksellinen-kehittamistyo/](https://help.jamk.fi/opinnaytettyo/fi/toteutustavat-ja-rakenne/tutkimuksellinen-kehittamisty/).

Kuprenko, V. 2021. Electron.js: Great Tool to Design Powerful Multi-Platform Desktop Apps. Web-DataRocks verkkosivujen blogi. Viitattu 21.2.2022. <https://www.webdatarocks.com/blog/electron-js-great-tool-to-design-powerful-multi-platform-desktop-apps/>.

Nehra, M. 2021. Top 10 Programming Languages for Desktop Apps In 2021. Blogi. Viitattu 24.5.2022. <https://www.decipherzone.com/blog-detail/top-programming-languages-for-desktop-apps-in-2021>.

Nokes, C. 2016. Deep dive into Electron's main and renderer processes. Blogi. Viitattu 12.4.2022. <https://cameronnokes.com/blog/deep-dive-into-electron%27s-main-and-renderer-processes/>.

Quick Start. 2021. Electronin verkkosivujen dokumentaatio. Viitattu 16.3.2022. <https://www.electronjs.org/docs/latest/tutorial/quick-start>.

Rykov, S. 2021. Progressive Web App Development: How to Cook PWA in 2022. Dev.to verkkosivut. Viitattu 22.2.2022. <https://dev.to/sergeyrykov/progressive-web-app-development-how-to-cook-pwa-in-2022-15g2>.

Sanderson, C. 2020. The best of both worlds: Progressive web apps and desktop containers. UX Collectiven verkkosivut. Viitattu 15.2.2022. <https://uxdesign.cc/the-best-of-both-worlds-progressive-web-apps-and-desktop-containers-45157e8ee7f0>.

Schmitz, H. 2019. Build a Desktop Application with Angular and Electron. Viitattu 10.2.2022. <https://developer.okta.com/blog/2019/03/20/build-desktop-app-with-angular-electron>.

Security. 2021. Electronin verkkosivujen dokumentaatio. Viitattu 26.5.2022. <https://www.electronjs.org/docs/latest/tutorial/security>.

What Is Electron.js? 2020. Electron Development: A Quick Guide. Viitattu 8.2.2022. <https://brainhub.eu/library/what-is-electron-js/>.

What Is JavaScript? 2021. MDN Web Docsin verkkosivut. Viitattu 18.2.2022. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

White, M. 2020. Why Static Typing & Why is TypeScript so popular? Section EngEd Community. Viitattu 21.2.2022. <https://www.section.io/engineering-education/typescript-static-typing/>.

Window Customization. 2021. Electronin verkkosivut. Viitattu 24.3.2022. <https://www.electronjs.org/docs/latest/tutorial/window-customization>.

2021 Developer Survey. 2021. Stack Overflow:n tutkimus. Viitattu 16.2.2022. <https://insights.stackoverflow.com/survey/20>