Tampere University

Petri Lehtinen

# EVALUATION OF ELECTRON FRAMEWORK IN DATA GATHERING APPLICATION

Faculty of Engineering and
Natural Sciences
Master of Science Thesis
April 2022

# ABSTRACT

Petri Lehtinen: Evaluation of Electron framework in data gathering application
Master of Science Thesis
Tampere University
Information technology in Automation
March 2022

JavaScript has been developing rapidly since its beginning in 1995. JavaScript as a scripting language was firstly designed to be used in web-browsers but due to the popularity of the language the nature of the language has been expanded to the server-side. This is mainly happened because of popularity of Node.js

Server-side JavaScript programming has enabled development of desktop applications with JavaScript. This concept was the beginning of Electron framework. Electron framework is a complete tool to build cross-platform desktop applications with web technologies (JavaScript, HTML and CSS), Google's Chromium renderer on the front-end and Node.js on back-end.

In this thesis a proof of concept was developed with Electron framework. This proof of concept artifact is evaluated in the terms of software development process, security, and resource footprint. The proof of concept is also meant to be capable to read NFC (Near Field Communication) tags. The results of this thesis are conducted through design science research process (DSRP), which provides a model for research of IT artefacts.

The results of this thesis showed that it is possible to develop adequate application in terms of resource footprint and security with Electron framework. Low skillset in programming was needed to create a working application. It was possible to create NFC reading functionality in Electron application.

Keywords: Electron, JavaScript, Node.js, Near Field Communication, Design Science, data gathering

# TIIVISTELMÄ

Petri Lehtinen: Evaluation of Electron framework in data gathering application
Diplomityö
Tampereen yliopisto
Automaation tietotekniikka
Maaliskuu 2022

---

JavaScript on kehittynyt hurjasti siitä lähtien kun se julkaistiin vuonna 1995. JavaScript aluksi suunniteltiin käyttöön selaimissa skriptauskielenä, mutta nopeasti suosion kasvaessa kielen luonne saatiin siirrettyä palvelimille. Tämä on pääosin Node.js:n suosion ansiota.

Node.js on mahdollistanut työpöytäsovellusten kehityksen JavaScriptillä. Tämä konsepti oli Electron ohjelmistokehyksen alkuajatus. Electron ohjelmistokehys on kokonaisvaltainen työkalu työpöytäsovellusten ohjelmoimiseen web teknologioiden avulla (JavaScript, HTML ja CSS). Electronissa sivujen renderöinti tapahtuu Googlen Chromium kirjaston avulla ja back end on toteutettu Node.js:n avulla.

Tässä diplomityössä kehitettiin sovellus Electron ohjelmistokehyksen avulla. Tätä sovellusta arvioidaan sovelluskehityksen, suorituskyvyn ja tietoturvan osalta. Soveluksen on myös suoriuduttava NFC (Near Field Communication) tagien lukemisesta. Tulokset tässä diplomityössä on johdettu suunnittelutieteen metodein.

Diplomityön tuloksena todettiin, että on mahdollista kehittää toimiva Electron sovellus tietoturvaa ja resurssien käyttöä ajatellen. Työssä kehitetyn sovelluksen toteuttamiseksi ei tarvittu korkea taitotasoa ohjelmoimisessa. Electron sovelluksella onnistuttiin myös toteuttamaan NFC lukutoiminto.

Avainsanat: Electron, JavaScript, Node.js, Near Field Communication, suunnittelutiede, tiedon keruu

# PREFACE

This document has been written in partnership with Valmet Automation Oy. The purpose of this document is to study the readiness of web technologies in desktop applications. In cross-platform desktop applications to be more precise.

I want to thank Valmet Automation and Mika Karaila for offering such great learning experience. I'm certain that the process of writing this document grants me skills that I'm pleased to have in future. I'm also hopeful that this document will grant my employer insight to future problems that which would have been troublesome without this study. Also, I want to thank David Hästbacka and Hannu Koivisto for supervising this thesis and giving excellent guidance.

Finally, I want to thank my girlfriend Annariina and my family for the support during this thesis.

Tampere, 7 April 2022

# CONTENTS

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| ABI | Application Binary Interface |
| API | Application Programming Interface |
| AWS | Amazon Web Service |
| CSP | Content Security Policy |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| DS | Design Science |
| DSRP | Design Science Research Process |
| ECMA | European Computer Manufacturer's Association |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IDE | Integrated Development Environment |
| IS | Information Science |
| NFC | Near Field Communication |
| npm | Node Package Manager |
| OS | Operating System |
| RCE | Remote Code Execution |
| UI | User Interface |
| UX | User Experience |

.

# 1. INTRODUCTION

Ever since the discovery of server-sided JavaScript implementations, there has been an idea of developing desktop applications with JavaScript. This idea is old as is JavaScript which was released in 1995 [12]. Since the release of JavaScript there was attempts to run JavaScript on server side, but the technology was not ready.

node-webkit (later NW.js) was started as an innovation program by Roger Wang in 2011 who was working at Intel at the time. It started as a Node.js module but was later transformed as its own framework. NW.js main goal was to insert Node.js into WebKit browsers. As NW.js became more popular Roger Wang decided to hire an intern to maintain the GitHub pull requests and issues. Zhao Cheng was recruited in 2012 and during his 6 months internship he took over the NW.js project. After his internship Zhao was contacted by GitHub who also were creating a framework where Node.js and Chromium would be integrated. Zhao was hired by GitHub and the development of Atom Shell (later Electron) started in 2013. Since then, Electron has been one of the most used frameworks for cross-platform desktop applications.

In this thesis the main goal is to study the readiness of the Electron framework. This is done by creating a literature review considering the Electron framework topics and creating digitalized data gathering application with Electron. This thesis has been done in the co-operation with Valmet Automation Oy and Valmet Technologies Oy.

## 1.1 Research Questions

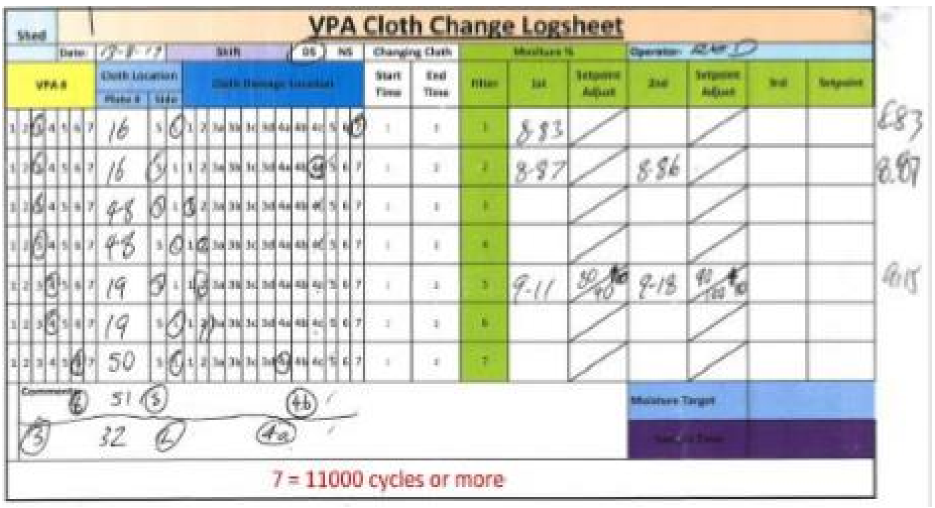
This thesis has following research questions,

- How Electron application manages in terms of:
  - software development process
  - resource footprint
  - security
- Is Electron framework suitable for creating an Near Field Communication (NFC) reading application?

First research questions consist of three attributes that are measured and evaluated during this thesis. Software development process is evaluated and reflected based on what problems were encountered during the development process of the Electron application created in this thesis. The resource footprint of the finished Electron application is evaluated. Resource footprint means how much memory application is using. Security is wide term what comes to software development. In this thesis we are researching the common Electron security issues and reflect how these are affected in the application of this thesis. Second research question is considering how Electron framework is capable of creating NFC functionality.

## 1.2 Use case for evaluation

In the customer's mining filtering processes, the filter cloth is essential part of the process. The filter cloth is a spare part that needs to be in good condition in order that the filtering process is desirable. Once the filter cloth reaches the end of its lifecycle or physically tears due to mechanical flaw, the filter cloth needs to be changed. This changing operation is done manually, and the operation is recorded to a table in a piece of paper.



VPA Cloth Change Logsheet

Shed | Date: 19-8-19 | Shift [✓ DS] NS | Changing Cloth | Moisture % | Operator: [illegible]

| VPA # | Cloth Location Plate # | Side | Cloth Damage Location | Start Time | End Time | Filter | 1st | Setpoint Adjust | 2nd | Setpoint Adjust | 3rd | Setpoint | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 2 [✓ 3] 4 5 6 7 | 16 | 5 [✓ 6] | 1 2 3a 3b 3c 3d 4a 4b 4c 5 6 [✓ 7] | : | : | 1 | 8.83 | | | | | | 8.83 |
| 1 2 [✓ 3] 4 5 6 7 | 16 | [✓ 5] 6 | 1 2 3a 3b 3c 3d 4a 4b [✓ 4c] 5 6 7 | : | : | 2 | 8.97 | | 8.56 | | | | 8.97 |
| 1 2 [✓ 3] 4 5 6 7 | 48 | [✓ 5] 6 | 1 [✓ 2] 3a 3b 3c 3d 4a 4b 4c 5 6 7 | : | : | 3 | | | | | | | |
| 1 2 [✓ 3] 4 5 6 7 | 48 | 5 [✓ 6] | 1 [✓ 2] 3a 3b 3c 3d 4a 4b 4c 5 6 7 | : | : | 4 | | | | | | | |
| 1 2 3 [✓ 4] 5 6 7 | 19 | [✓ 5] 6 | 1 [✓ 2] 3a 3b 3c 3d 4a 4b 4c 5 6 7 | : | : | 5 | 9.11 | 80/90 | 9.18 | 90/100 | | | 9.15 |
| 1 2 3 [✓ 4] 5 6 7 | 19 | 5 [✓ 6] | 1 2 [✓ 3a] 3b 3c 3d 4a 4b 4c 5 6 7 | : | : | 6 | | | | | | | |
| 1 2 3 4 5 [✓ 6] 7 | 50 | 5 [✓ 6] | 1 2 3a 3b 3c [✓ 3d] 4a 4b 4c 5 6 7 | : | : | 7 | | | | | | | |

Comments: 6 51 5 4b
5 32 6 4a

Moisture Target

[illegible]

7 = 11000 cycles or more

***Figure 1.*** *Example of cloth change logsheet*

The records of these operations are vital information for the administration of the mining company. From these records it is possible to calculate pre-emptive maintenance schedules in order to maximize productivity.

The proof of concept of this thesis provides an application that is capable of digitalizing and modernizing the data gathering of this operation. The application is a substitute for the pen-and-paper method and is only meant to provide a solution to record the data from the cloth change operation. Electron framework was chosen as a platform to build the proof of concept.

## 1.3 Structure of thesis

In this thesis we will study the readiness and the current state Electron development with JavaScript. Also, the NFC reading capability with Electron applications is studied. In background part of this thesis desktop applications are presented and technologies behind Electron framework are covered. After the background chapter the research model that is used in this thesis is presented. In fourth chapter the proof of concept application is introduced and the use environment is also covered briefly. Fifth chapter is the evaluation chapter where the results of the proof of concept application are reflected considering the research questions mentioned in chapter 1.1. Last chapter is conclusion chapter of this thesis.

# 2. BACKGROUND

In the following sub-chapters all the main components are previewed for the full understanding of this thesis. Firstly, desktop applications are introduced. After that, the technologies used in this thesis are covered. In final sub-chapter of this chapter the basics of Electron framework is introduced.

## 2.1 Desktop Applications

In this chapter few of the main qualities of desktop applications are presented. The evaluated technology of this thesis is Electron, which is a cross-platform desktop application framework.

The typical definition of portability goes something like this:

*Portability is a measure of how easily software can be made to execute successfully on more than one operating system or platform.*

This definition, however, is only a starting point. To truly define the term *portability*, one must consider more than the mere fact that the software can be made to execute on another operating system. [19]

Back in the mid-1990s, the beginning of JavaScript, web browsers had varying implementations of JavaScript built in, making applications run in specific browser. In the mid-2000s libraries such as jQuery and MooTools grew in popularity. These libraries provided an application programming interface (API) to solve the different implementations of JavaScript in browsers. In the early 2010s JavaScript was in such state that frameworks like Angular, Backbone and Ember were popular among developers in web application world. These frameworks provided boilerplates and provided structured way to handle templating, data, and user interactions. [7] As described above, JavaScript started from state which there was lot of work to be done to achieve cross-platform capabilities among browsers and progressed towards achieving it.

At the same time in desktop applications developers were forced to program with system-specific programming languages and idea of cross-platform desktop applications was just a beautiful idea. Situation was similar in mobile applications. Web applications were popular and powerful technologies were emerging every day among web applications. Although, by building a web-only application you are excluded from the mobile platform application shops, which are major factor in mobile application installation. [7]

Even though web applications were the only truly platform to create cross-platform user-interfaces and carried much power, users still must rely on desktop applications.

Desktop applications are software programs that can be run on standalone computers. They are not designed to run on a mobile device and not on the browser. Usually, desktop applications are designed to have polished user experience and good qualities in performance and security. Some of the most used desktop applications are text editors such as Microsoft Word and gaming applications. Web browsers are also desktop applications.

Below there is a group of positive features of desktop applications listed: [8]

1. No internet access required — Although internet is available in most use cases where software is used, it is sometimes beneficial to be able to run application without it. Desktop applications are installed on the device hard drive rather than relying on downloading content from server.

2. Security — The fact that almost everything is tracked on the web, it is more secure to use application on desktop.

3. Performance — Due to the fact that desktop application is installed on the device, you can use the device's resources more efficient. Applications like games and editing software which use resources heavily can truly benefit from being installed on the hard drive. Usually, desktop applications can also offer more wide variety of features in the application.

Below there is a group negative features of desktop applications described: [8]

1. Portability — User is not able to use the application wherever he chooses. Desktop applications are fixed to run only on the device it is installed on. Significant disadvantage compared to web application.

2. Hard drive space — Desktop applications are installed on the device. This fact requires hard drive space from the device, and usually the files it produces and uses are also locally maintained.

3. Updates — In web applications the contents are downloaded from the server when using. This means the user is always getting the newest version whenever using the web application. In desktop world you must do it manually. It is possible to notify user that updates are available but still it is not nearly as pleasant user experience than in web.

Desktop applications traditionally are developed with native languages in each operating system. Server-sided JavaScript and many modern frameworks like Electron and NW.js have made it possible to create applications that can be shipped to all platforms with one codebase. Even frameworks (React, Flutter) that were initially designed to run on all mobile platforms, are supporting desktop development. Electron is using server-side JavaScript to achieve cross-platform capability.

## 2.2 Web technologies

Web technologies are used as programming languages in Electron development. Web technologies that we are using in this thesis are JavaScript, The Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS).

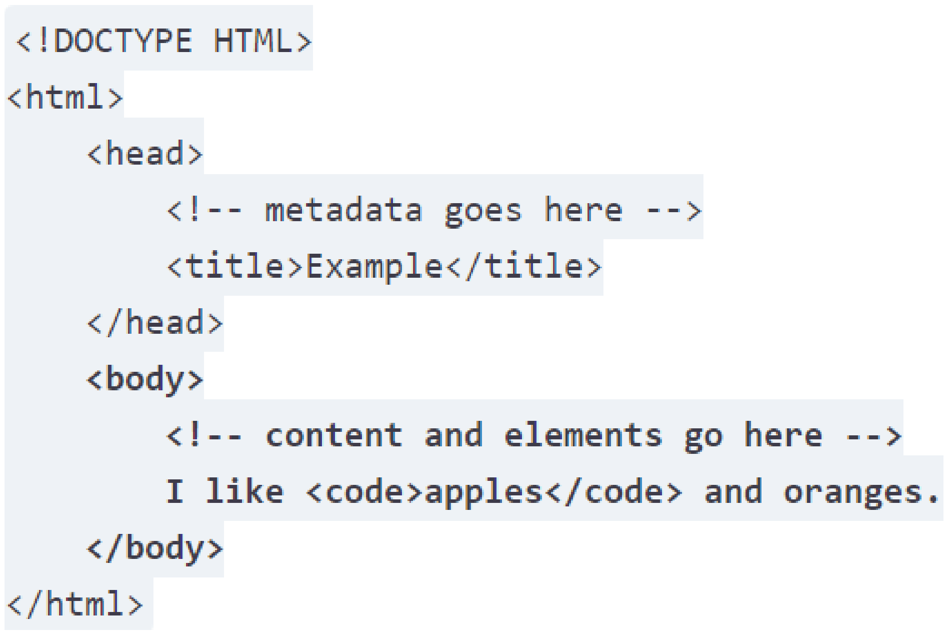
JavaScript is dynamic programming language, initially intended for client-side scripting, but currently also widely used in server-side in runtimes such as Node.js. JavaScript is well-suited for object-oriented and functional programming styles. JavaScript is not meant to be confused with Java, although its syntax is loosely based on Java. JavaScript was developed by Netscape to be part of their Netscape Navigator browser. First version of JavaScript was introduced in 1995 in Netscape Navigator 2. Before JavaScript browsers were only capable of displaying hypertext documents. JavaScript was created to add more responsiveness and functionality to webpages. [17]

JavaScript was submitted to ECMA ( European Computer Manufacturer’s Association ) for standardization by Netscape. This was done successfully and now JavaScript follow’s ECMAScript standardization. Most common ECMAScript (abbreviation ES) standards are ES5 that was used the most 2010s and ES6 released in 2015 with added features such as module and class syntax. Since ES6 release, a new ECMAScript specification is released every year (ES2016, ES2017, ES2018 etc.). [12]

Although JavaScript is thriving in web applications and in browsers, lately it has made its presence known in desktop applications through tools like Node.js and Electron that are also used in this thesis via Electron framework. This makes it possible for developers to use same skillset to build polished desktop applications in full stack (both front- and back-end solutions).

JavaScript is currently the most popular programming language. When looking the results of the Stack Overflow's developer survey for year 2019, JavaScript is at the number one position for the most popular programming, scripting, and markup languages. This has been the case for seven years in a row now. [30]

The Hypertext Markup Language (HTML) has been around since the beginning of the World Wide Web in late 80's. HTML is the very basic markup language that describes the structure of a web content. In current web development there is other languages that support the behavior of HTML in web documents. In addition, there is also JavaScript used for functionality and CSS for appearance [16].These technologies will also be covered in this thesis.



```
<!DOCTYPE HTML>
<html>
    <head>
        <!-- metadata goes here -->
        <title>Example</title>
    </head>
    <body>
        <!-- content and elements go here -->
        I like <code>apples</code> and oranges.
    </body>
</html>
```

***Figure 2.*** *Basic HTML code [13]*

HTML has various tags that are used to display text, images, and other content in a web browser. Basic HTML code structure usually goes something like this: HTML element is the top-level element (root element) of a HTML document. All other elements must be descendants of this element. To add content to a web page, a body element is needed. There can be only one body element in html document. Body element contains all the contents of a HTML document, such as images, headings, paragraphs, lists, tables, etc. In addition of HTML and body element also other elements are used outside of body element such as head element, which describes metadata about the HTML document such as titles, scripts, and stylesheets. [16] These elements can be found as an example in figure 2.

HTML is an old technology that has preserved in the rapid changes of software development. By doing so, it has had to create and renew functionalities. HTML has many versions ranging from the initial 1.0 to the current HTML5. In this thesis when talking about HTML we are referring to HTML5.

Cascading Style Sheets provide a way to alter the appearance and format of HTML elements. A single CSS declaration consists of one property and one value separated by a colon. It is common for an HTML element to have multiple declarations. These declarations can be separated by a semi-colon as shown in figure 3.

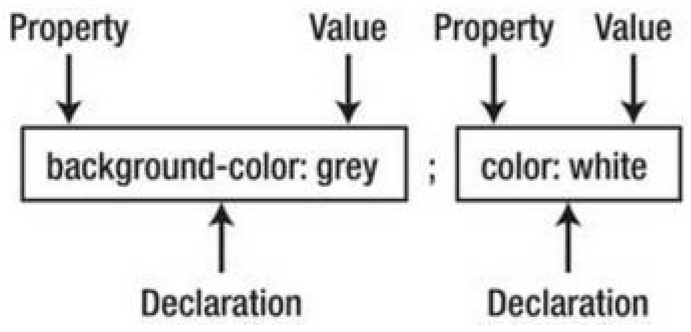


***Figure 3****. Cascading Style Sheets syntax [13]*

HTML among CSS and JavaScript is used to create the front-end documents of an application created in this thesis

## 2.3 Node.js

Traditionally JavaScript run on web browsers which are clients for the web servers. Node.js is a platform for writing JavaScript applications outside web browsers. Node.js is designed to be extremely scalable and is great for networked applications. This is achieved by being able to utilize server-side JavaScript, asynchronous I/O and asynchronous programming. Node.js is written in C++ language. It is built around JavaScript anonymous functions and a single execution thread event-driven architecture. [4] By using JavaScript on the server side it also makes the required skillset to build application to be smaller when you can write both back- and frontend with the same language.

The Node.js architecture is different compared to many other JavaScript runtimes. Node.js doesn't use blocking multi-threaded architecture due to threads' inherent complexity. By using single-threaded event-driven architecture, Node.js achieves lower memory footprint, higher throughput, better latency profile under load and simpler programming model. [4]

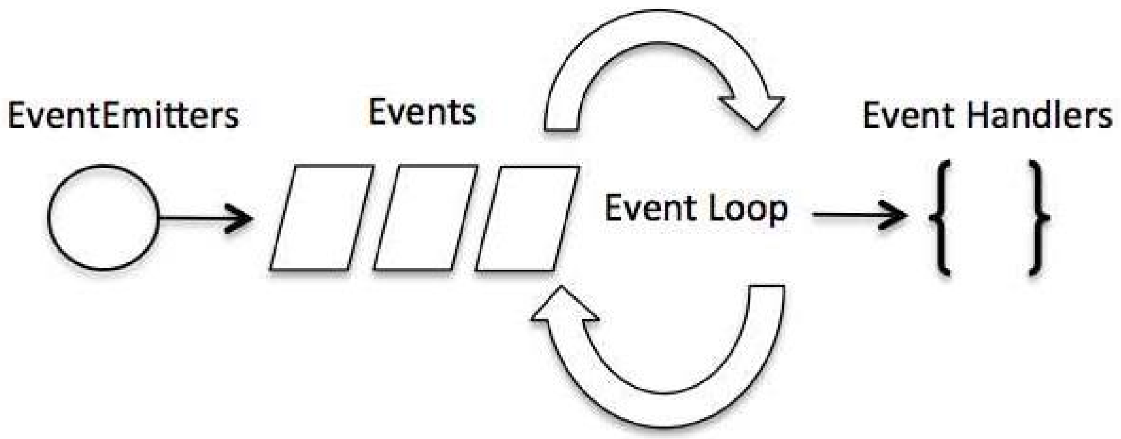


***Figure 4***. *Single-threaded event-driven architecture of Node.js [4]*

Node.js has single execution thread, where EventEmitters create events to be handled by the event loop. Event loop dispatches the events to the correct event handlers. The main idea behind this architecture is that any operation that would block or take time to complete must use the asynchronous model. These functions are to be given an anonymous function to act as a handler callback or to return a Promise (since ES2015) [4]. This execution thread is illustrated in figure 4 above. Node.js is built on non-blocking I/O event loop and a layer of file and network libraries, which are in turn built on V8 JavaScript engine (used in Google Chrome web browser). [4] V8 is becoming the de facto JavaScript engine and is the most used JavaScript engine with no rivals' insight.

Node.JS is in a major role in application that is made in this thesis as it is the backend runtime used in Electron. Electron uses Chrome's rendering machine wrapped in Node.js libraries.

## 2.4 Electron

Electron (previously known as Atom Shell) is an open source library developed by GitHub for building cross-platform desktop applications with HTML, CSS, and JavaScript. The fact that Electron applications are developed with web technologies makes them flexible among multiple platforms (Microsoft Windows, Linux, macOS). Notable open-source projects have been done with Electron such as Visual Studio Code, Slack and Atom [9]. Electron is currently the most popular framework and runtime for creating cross-platform desktop applications with web-technologies [28].

First version of Electron was introduced in August 12, 2013. The version was called atom-shell v0.3.1. Most recent version of Electron when writing this thesis is 18.0.1. In September 2021, development team of Electron decided to release stable major version every 8 weeks. This matches Chromium's plans to release a new milestone every 4 weeks. Microsoft Store also requires Chromium-based apps to be no older than 2 major versions.

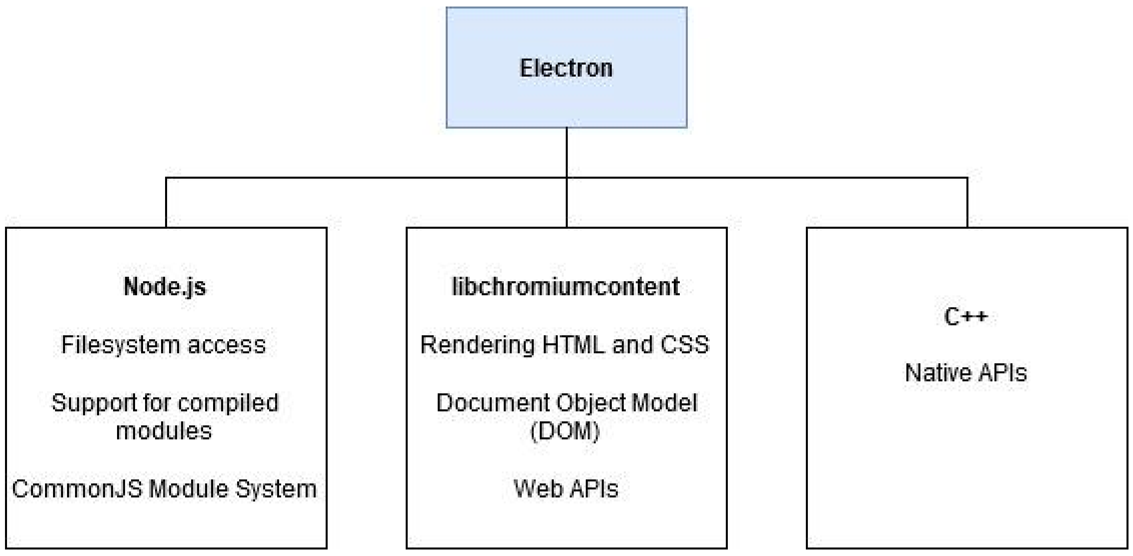
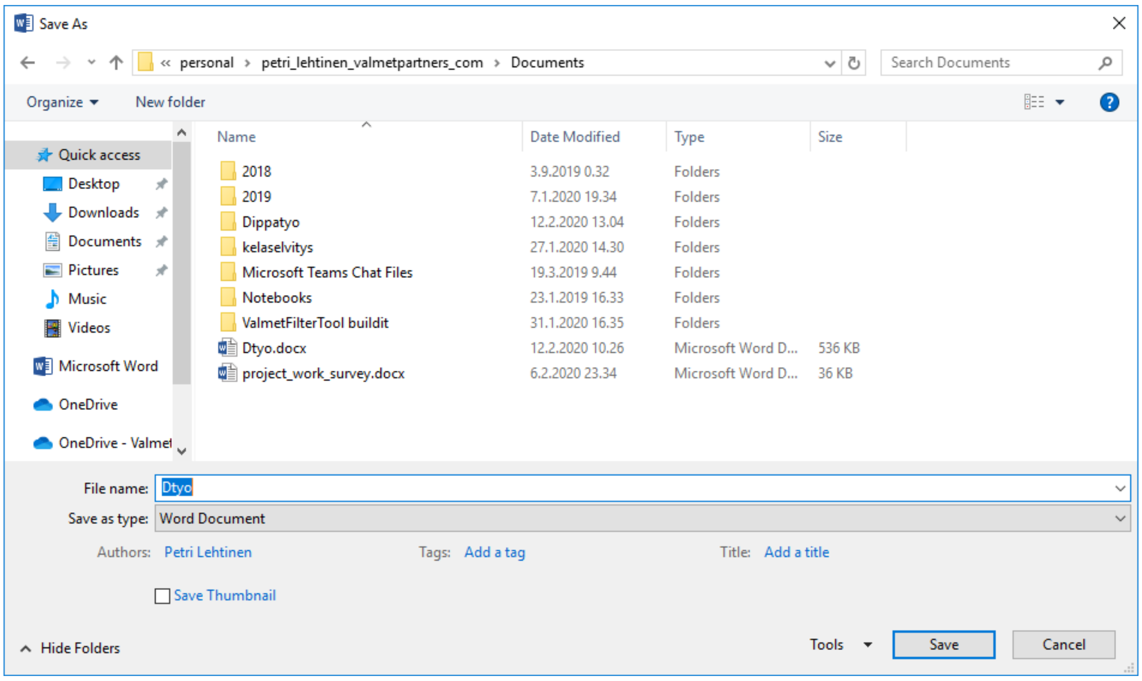


***Figure 5.*** Electron combines the core web browsing component of Chromium with the low-level system access of Node. [18]

In Electron three main components are embedded. These components are combined to single runtime called Electron. The frontend of Electron's rendering is done by Chromium's rendering library (known as *libchromiumcontent)*, an open source foundation for Google's Chrome browser. It has wide support from open-source community and continuous updates for latest web standards, efficient JavaScript engine (V8), and great developer tools for debugging. [28]

The second component is Node.js. Node.js is very popular open-source cross-platform JavaScript runtime built on top of V8 which is the same as in the *libchromiumcontent.* Third component of Electron is a layer of C++. This layer holds many native API implementations for most common operating systems (Windows, macOS and Linux) operations. These operations for each operating system are for example native notifications and dialogs (see figure 6). In addition of these native APIs there is access to Node.js APIs and Node.js native modules. [28]



| Name | Date Modified | Type | Size |
|---|---|---|---|
| 2018 | 3.9.2019 0.32 | Folders | |
| 2019 | 7.1.2020 19.34 | Folders | |
| Dippatyo | 12.2.2020 13.04 | Folders | |
| kelaselvitys | 27.1.2020 14.30 | Folders | |
| Microsoft Teams Chat Files | 19.3.2019 9.44 | Folders | |
| Notebooks | 23.1.2019 16.33 | Folders | |
| ValmetFilterTool buildit | 31.1.2020 16.35 | Folders | |
| Dtyo.docx | 12.2.2020 10.26 | Microsoft Word D... | 536 KB |
| project_work_survey.docx | 6.2.2020 23.34 | Microsoft Word D... | 36 KB |

File name: Dtyo

Save as type: Word Document

***Figure 6.*** Windows 10 native open file dialog

In a nutshell, developers can use Electron to build applications that contain Chrome's rendering engine while having access to all Node.js - including every module available on Node package manager (npm). [24] One of the main reasons that Electron is used in this thesis instead of creating a browser-based application, is that we need to have access to Node.js API.

### 2.4.1 Main and Renderer Processes

Since you can build both front- and backend implementations with Electron, it is wise design decision to distinct these two major components. This is done in Electron by dividing the back-end solution to so called ‘main process’ and front-end to ‘renderer process'. Electron application can have only one main process and for each window Electron application has it has its own renderer process. This multi-renderer behavior is inherited from Chromium as Electron uses it to render graphical interfaces [9].

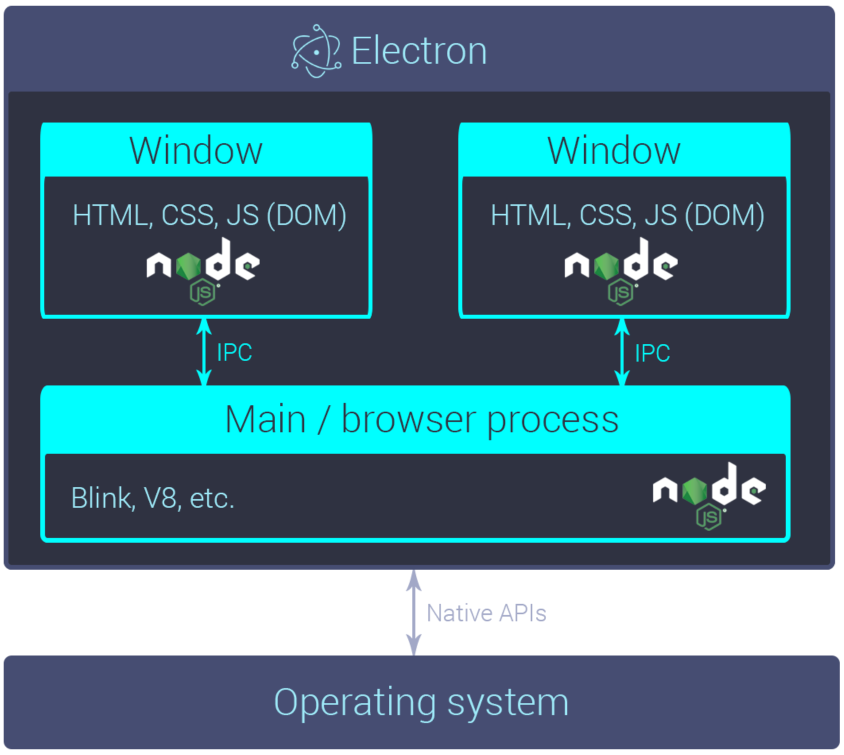


**Figure 7**. *Electron application architecture*

Main process has no access to Document object model (DOM) APIs and behaves a lot like a Node.js process. Main process has access to modules that provide functionalities that are typical to desktop applications such as Auto Updating, Crash Reports and system dialogs. Main process contains the events for Electron application lifecycle, such as events for application ready state (finished loading) and application termination. [20]

The renderer process is a graphical component of the Electron application. Renderer processes are invoked from main process by creating BrowserWindow instances as shown in the code snippet below.

Browser window instances have over 50 options which can be found on the Electron API documents. But the most important option is the node-integration. node-integration allows Browser window instances to access Node.js APIs in browser window in addition to browser provided APIs. Electron v5.0.0 node-integration option default has been set to false to improve security.

```
// In the main process.
const { BrowserWindow } = require('electron')

let win = new BrowserWindow({ width: 800, height: 600 })
win.on('closed', () => {
  win = null
})

// Load a remote URL
win.loadURL('index.html')
```

Renderer process script is loaded whenever BrowserWindow loads an HTML file. The script is included in the HTML file in the `<script>` tag as shown below.

```
<body>
  <h1>Hello World!</h1>
  We are using Node.js <span id="node-version"></span>,
  Chromium <span id="chrome-version"></span>,
  and Electron <span id="electron-version"></span>.

  <script src="./renderer.js"></script>
</body>
```

Electron application looks like a regular Node.js application on a directory level. It has package.json which lists all native module dependencies needed for application and developer environment. Package.json also defines the ‘main’ (main.js/index.js) script that launches Electron’s main process. In this file you can also create npm scripts for example packaging and distributing your application

### 2.4.2 Security in Electron

When developing Electron application, it must be taken into account that even though Electron applications are built on web technologies, it is not a browser-based application. Therefore, it can’t be trusted that our application runs in golden-sandbox provided by the browser that has been developed by large team of engineers who are able to respond quickly to newly discovered security threats. In Electron applications JavaScript code wield much greater power than in regular web applications because it runs on the backend via Node.js. It can access filesystem and shell and therefore it is more fragile to exploitations. Electron's document website states also the following: *"the inherent security risks scale with the additional powers granted to your code"* [10].

In Electron website there is a security checklist that covers most of the major security threats and ways to counter those [29]. This checklist is also part of this thesis. (see Appendix A)

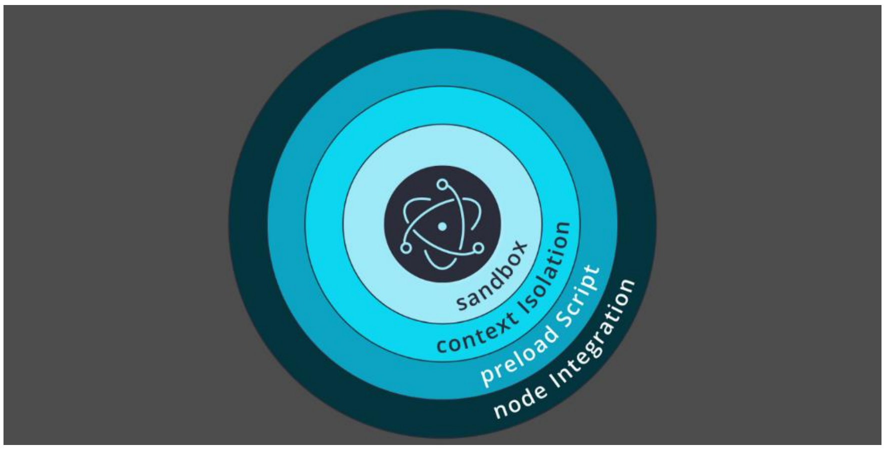
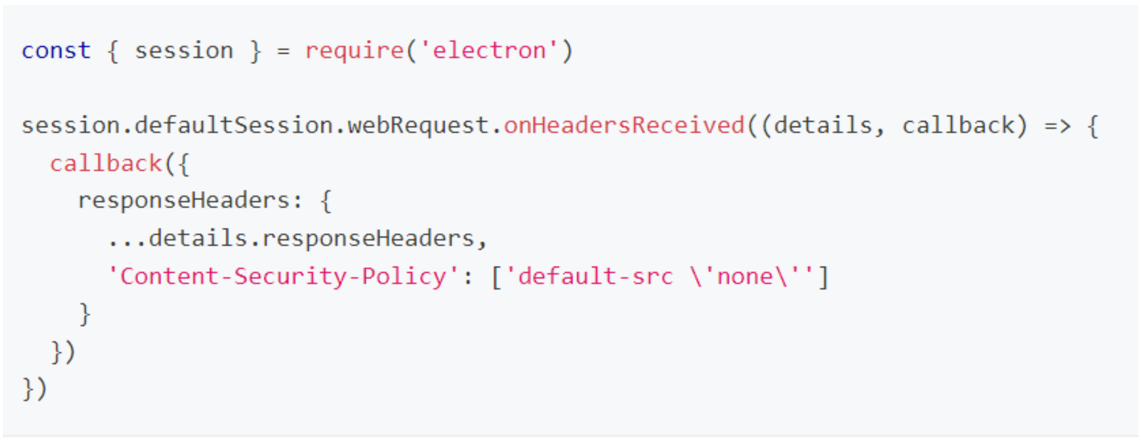


***Figure 8.*** *Layers of security in Electron's renderer processes. [15]*

By interpreting the security checklist (see Appendix A), we can conclude that high volume of the risks is connected to downloading and executing untrusted code. In order to tackle most of the risks above, it is recommended to use Electron's embedded Chromium (renderer process) to display locally hosted pages, instead of downloading from remote origins. However, this is not always possible in some scenarios. If Electron renderer process is needed to display pages from hosted destinations, properly configured CSP (Content Security Policy) is needed on these pages. [15] CSP provides an additional layer of protection against cross-site-scripting attacks and data injections. Electron is supporting the Content Security Policy Hypertext Transfer Protocol (HTTP) header. This header can be set using the webRequest.onHeadersReceived handler (see figure 8). CSP allows server to restrict and control the content and resources the Electron application can use. The example shown in figure 9 is only accepting resources from the current website. [11]



```
const { session } = require('electron')

session.defaultSession.webRequest.onHeadersReceived((details, callback) => {
  callback({
    responseHeaders: {
      ...details.responseHeaders,
      'Content-Security-Policy': ['default-src \'none\'']
    }
  })
})
```

***Figure 9.** Content Security Policy set by using Electron’s webRequest.onHeaders-Received handler. [11]*

This is valid for the Electron framework itself, but it doesn’t consider that Node.js and Electron ecosystem relies heavily on the modules created by other authors. When the number of external includes increase so does the probability of having vulnerable code transferred to the main code. Nikiforakis et al. states the following in their research paper concerning remote inclusions: “whenever developers of a web application decide to include a library from a third-party provider, they allow the latter to execute code with the same level of privilege as their own code” [22]. When talking about Node.js environment and external modules that is when npm is without exception always mentioned.

One of the major backdoors in Electron applications is that the framework exposes electron.asar file. This means that every Electron application has its own electron.asar file within its installation folder. Asar is a simple extensive archive format that works like tar-archive that concatenates all files together without compression, while having random access support. This file is not encrypted, obfuscated, or protected in anyway. It is possible to write these files without modifying the signature of the actual application. The fact that the code of the Electron framework is exposed means, that if you can create an RCE (Remote Code Execution) by exploiting the contents of the asar file, the exploitation is repeatable in other Electron based applications. However, this exploitation is also preventable by properly configuring security features in Electron application.

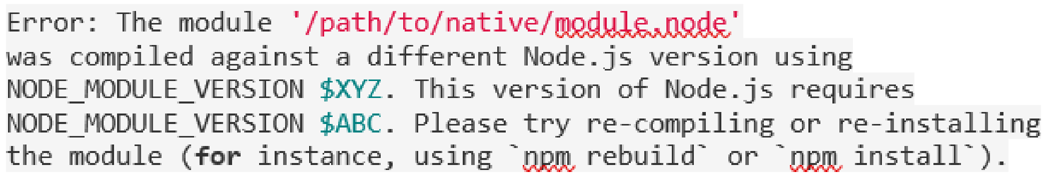
## 2.4.3 Native Node Modules in Electron

Native Node modules (or C++ addons) provide interface between JavaScript and C/C++ libraries. Native Node modules are dynamically linked shared objects written in C++.

With ‘require()’, it is possible to load Native Node modules as an ordinary Node.js modules. [31]

Electron applications roots are strongly in the JavaScript ecosystem and therefore mostly uses modules from npm. Sometimes this is not enough due to the desktop nature of the Electron applications. Native Node modules makes it possible to Electron applications utilize operating system (OS) APIs that are not implemented as a part of the Electron API. Native Node Modules can also be used if application is required to have lot of computing power since Native Node Modules are written C/C++.

In Electron Native Node modules are supported, but in most use cases Electron has a different application binary interface (V8 version) from given Node.js binary in your system. Due to this fact, the Native Node modules that are used, must be recompiled for current Electron version. In the case you don’t recompile the native modules that have different ABI (Application Binary Interface), you get following error in terminal when trying to execute your application. [31]



```
Error: The module '/path/to/native/module.node'
was compiled against a different Node.js version using
NODE_MODULE_VERSION $XYZ. This version of Node.js requires
NODE_MODULE_VERSION $ABC. Please try re-compiling or re-installing
the module (for instance, using `npm rebuild` or `npm install`).
```

***Figure 10.*** *Native Node module ABI error. [31]*

Fortunately, there are tools that deal with this issue effortlessly. ‘electron-rebuild’ is a module which automatically determines the version of Electron and executes the manual steps to fetch the correct headers for the Native Node modules for your application. ‘Electron Forge’ is an all-around tool that also does recompiling of Native Node modules automatically (uses electron-rebuild internally). Electron Forge is briefly covered in chapter 5.1.1. [31]

Native Node Modules are used to implement an NFC functionality in the proof of concept of this thesis.

## 2.5 Electron and Dissenting Opinions

Electron applications have reputation of consuming high amounts of memory compared to native applications [2]. This is also demonstrated briefly in the evaluation chapter of this thesis. The fact that Electron applications use memory heavily causes users that value performance and smoothness, hate Electron applications. Bayes states the following in his Medium article ‘Electron is Cancer’: ‘being slow on today’s super fast hardware is a bug.’ [2]. In his article, Bayes is tackling the argument ‘Electron improves productivity’ by stating the following: ‘Okay, sure having a plumber cut out a square wheel from a plank is also a lot easier to do than having a woodworker carve a perfectly round wooden wheel, but it is gonna be one hell of a bumpy ride, and square wheels are actually fine, right?’ [2]. This article is showing the very concerned side of the discussion when it comes to Electron. It is pointing out real issues of the Electron applications but ignores the features and power that Electron can provide.

So why do companies like Windows, Slack and Discord, keep shipping Electron applications [10]? In article ‘Electron isn’t Cancer but it is a Symptom of a Disease’, McCann is providing a set of common theories. In this chapter these theories are covered briefly.

1. Developers are Lazy — This theory states that most developers know how to use web technologies and genuinely think that Electron is good enough and Electron can replace native application frameworks. This lazy developer doesn’t see the point of investing time to learn how to develop efficient native application. [21]
2. Companies are Cheap — In this theory the responsibility is pointing out towards the employer, not the developer. Companies are not willing invest the extra resources and money to maintain native solutions for each platform when Electron applications provide the application that most end users are happy to use. [21]
3. Users Just Don’t Care Like they Used To — This theory is plausible if previous theory is assumed correct. Software companies can ship Electron applications that are not as efficient as native applications, if the end users don’t expect certain requirements what comes to performance. In this theory McCann also is discussing about the rise of popularity of Mac platform and what it has done for the expectations for initially well-built Mac applications. This theory illustrates that when smaller, more technically oriented userbase grows into more wide audience, it can loosen the requirements that the initial userbase expects. [21]

4. Cross Platform Apps are Better — The last theory is less common than the others. Previous theories have been blaming someone. This theory concentrating on the fact that Electron and cross platform applications are genuinely better than native applications. This statement is made from a view that prioritizes other things than performance and resource utilization. [21]

Pike talks about differences between native and cross platform applications and what attributes either one possesses in his blog post: ‘The Persistent Gravity of Cross Platform’. Pike is stating the following: ‘At the highest level, cross-platform user interface (UI) technologies prioritize coordinated featurefulness over polished user experience (UX).’ [25]

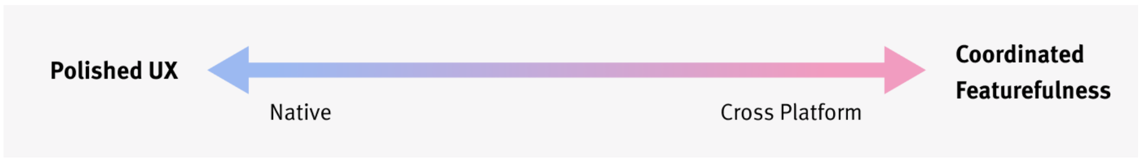


***Figure 11***. *Cross Platform is aiming for Coordinated Featurefulness. [25]*

Electron applications indeed provide a long list of OS related API’s and tools to create features easily. Once these features have been created for single platform, it is usually implemented in other platforms as well. Pike is mentioning in his blog post that large companies are more likely to create their product with cross platform frameworks because it is way easier to manage new features with single codebase than wait for all different platform teams to develop their own solutions [25].

# 3. RESEARCH METHODOLOGY

The answers for research questions presented in chapter 1.1 are conducted through the methods and perspectives of design science (DS). DS is a paradigm widely accepted in information science (IS) research. DS aims to solve problems scientifically through study of conceptual artefacts intended to solve identified organizational problems. [13]

More precisely the research part of this thesis is done by following the design science research process (DSRP) presented by Peffers et al. [24]. It provides the researcher a conceptual model for design science research in information science and mental model for its presentation [24]. The model is divided into six activities that are presented below.

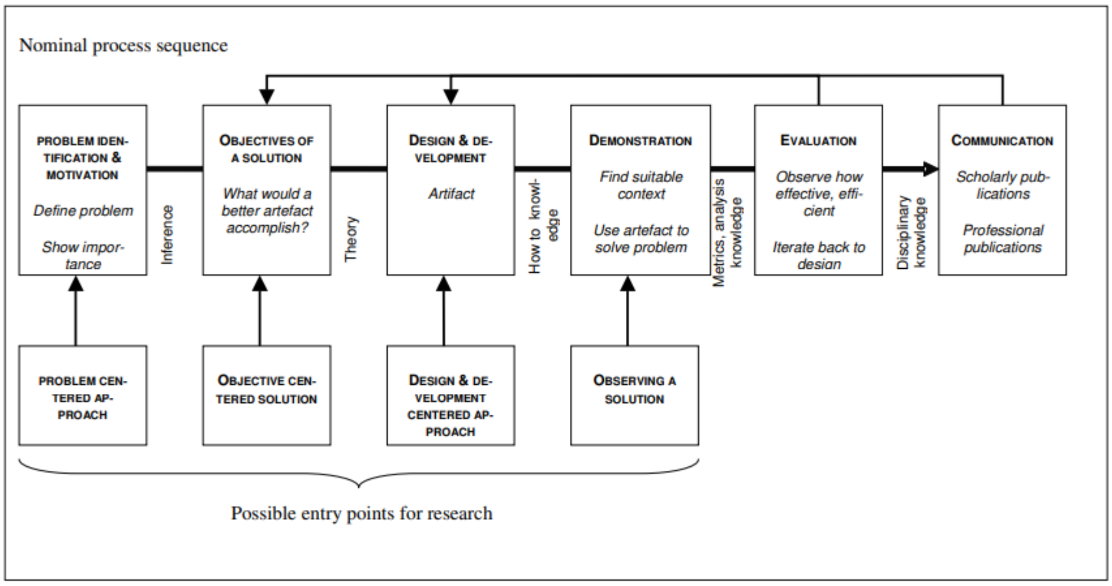

Nominal process sequence
PROBLEM IDENTIFICATION & MOTIVATION
Define problem
Show importance
Inference
OBJECTIVES OF A SOLUTION
What would a better artefact accomplish?
Theory
DESIGN & DEVELOPMENT
Artifact
How to knowledge
DEMONSTRATION
Find suitable context
Use artefact to solve problem
Metrics, analysis knowledge
EVALUATION
Observe how effective, efficient
Iterate back to design
Disciplinary knowledge
COMMUNICATION
Scholarly publications
Professional publications
PROBLEM CENTERED APPROACH
OBJECTIVE CENTERED SOLUTION
DESIGN & DEVELOPMENT CENTERED APPROACH
OBSERVING A SOLUTION
Possible entry points for research

***Figure 12**. DSRP Process Model [24]*

1. Problem identification and motivation — research problem is defined, and the value of a solution is justified. By justifying the value of the solution, the researcher and the audience of the research are more motivated to pursue the solution and accept the results. It also helps to understand the reasoning associated with the researcher’s knowledge and understanding of the problem. [24]
2. Objectives of a solution — objectives for the solution are concluded from the research problem definition. The objectives can be quantitative or qualitative.

Objectives may be presented as improvements to an existing solution or a set of goals to a problem that has not been solved yet. [24]

3. Design and development — This activity includes the designing and developing of the artifactual solution. The artifacts can be constructs, models, methods, or instantiations with each of the terms used broadly. [24]
4. Demonstration — The created artifact is to be demonstrated in this activity and its efficacy is observed. The demonstration can be done by using the artifact in experimentation, simulation, a case study, or other appropriate activity. [24]
5. Evaluation — In this activity the supportability of the solution is measured and observed. This is done by comparing the objectives of a solution to actual observed results from the use of the artifact in the demonstration activity. After successful evaluation, feedback is given to activities 2 and 3 and possible reiterations are done to those activities based on the feedback. This feedback can affect the objectives or the design of the artifact. [24]
6. Communication — Document the research and results and spread the results through proper channels. These channels could be professional and/or research publications for research papers. Reiterations to objectives and design can be done based on the feedback given by audience of the research. [24]

Although this process is presented in a nominally sequenced order, this model can be entered through multiple entry points as displayed in figure 12 [24]. In this thesis we are entering this model from activity ‘Problem identification and motivation’.

First and second activity are both presented in the research questions. Third activity is operated by developing the Electron application. Fourth activity is executed by using the Electron proof of concept application. Fifth activity is described in the Evaluation chapter of this thesis. Sixth activity is this document.

# 4. PROOF OF CONCEPT

In this chapter the proof of concept is described. The platform for the Electron application is covered and some tools are also presented.

## 4.1 Requirements and limitations

In this chapter we go through what sort of requirements we are looking Electron framework to fulfill. Also, limitations that are present in the proof of concept are listed in this chapter.

Proof of concept was done as a customer project to Valmet Fabrics and as a part of their Valmet Smart Cloth product. The main goal of the proof of concept was to build a desktop application that gathers information from the factory floor level and forwards this information to cloud (S3 Cloud Storage by Amazon Web Services (AWS)).

Valmet Smart Cloth is used to provide modern analytics for the chamber filter press. More precisely it is used to gather the lifecycle information of a filter cloth within the chamber filter press. Chamber filter presses are used to filter valuable minerals from the rock material and the condition of the filter cloths are vital aspect of the filtering process. In this thesis the filtering process is not covered but instead focus will be on the data gathering application itself. Valmet Smart Cloth also contains cloud analytics for the gathered data. However, it is not in the scope of this thesis.

The Valmet Smart Cloths are equipped with unique NFC tags that identify each cloth separately. These NFC tags are scanned with the help of the proof of concept application. NFC tags in these filters provide an interface for the proof of concept application.

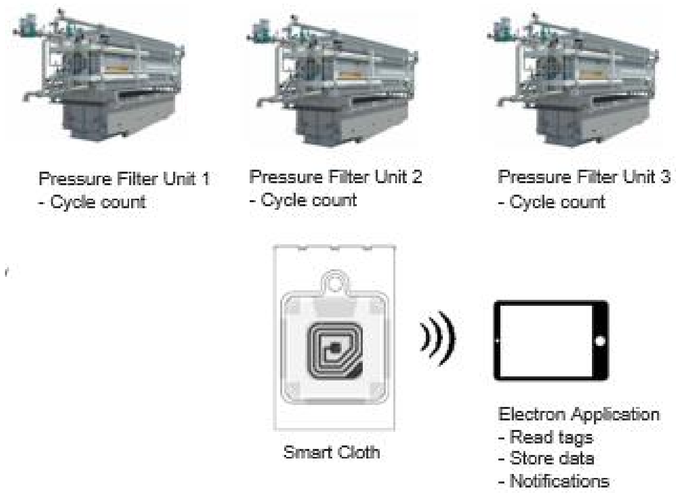


***Figure 13.*** *Proof of concept application environment*

Reasons that led this proof of concept to be developed with Electron are offline capability, NFC tag reading (can't be done in browser) and easy to implement (web-technologies). Electron also supports auto-updating out of the box which is also considered as a major feature in this application.

Main attributes of proof of concept are evaluated in chapter 5. These attributes are

1. software development process
2. resource footprint
3. security

These attributes are detailed more in chapter 1.1.

Also, Web NFC API is briefly introduced in the evaluation chapter as an alternative approach to create NFC reading application.

## 4.2 Software

The application is built on Electron framework version 6.0.3. Electron versions 6.x contain Chromium version 76.0.3809.88, Node.js 12.4.0, and V8 7.6.303.22. Electron updates Node.js every major update and Chromium more frequently. By the time that this thesis written, the newest version of Electron is 18.0.1. It is fair to say that the proof of concept is significantly out of date in versions. This fact is also taken in account in the evaluation chapter later in this thesis. The fundamentals of Electron framework are covered in chapter 2.4.

Visual Studio Code (not to be confused with Visual Studio) was used as an IDE (Integrated Development Environment) in this application development process. Visual Studio Code is a freeware source-code editor created by Microsoft. Visual Studio Code was chosen for its popularity in the industry and for its convenience in the web- and node development. Also, the fact that Visual Studio Code is built on Electron is beautiful idea [1]. Visual Studio Code provides support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

For version controlling Bitbucket was used. Bitbucket is a Git-based source code repository hosting service owned by Atlassian.
The source code of the application is built on a boilerplate called ‘electron-quick-start’ (https://github.com/electron/electron-quick-start). This boilerplate provides all the essential files to start Electron application development.

- package.json – Points to the app’s main file and lists dependencies and scripts of the environment
- main.js – This file starts the app and creates browser window to render HTML. This is the app’s main process that is covered in chapter 2.4.1.
- index.js – A web page to render. This is the app’s renderer process, also covered in chapter 2.4.1.

The application is bundled via command line tool called ‘electron-packager’. electron-packager can bundle OS-specific bundles (.app, .exe, etc.) from Electron-based source code. To create distributable like installer or Linux package some other tool is required.

## 4.3 Hardware

Electron is a cross-platform desktop framework. This means that Electron applications can be run on Windows, Linux, and Mac environments. However, in this proof of concept use case, Windows was decided to be the platform for developing and testing. Cross-platform capability was considered to prove value in the later stages of the application development. The data gathering application can be run on 64-bit version of Microsoft Windows. Since this is a factory floor level data gathering tool, the PC must be mobile. In addition, it should be IP-graded to provide the protection against the hazards of a factory environment. In this proof of concept, a 10-inch Panasonic Toughpad FZ-G1 with Windows 10 is used to test the application.

To make it more native feeling experience for the user of the application, a Windows single-app kiosk mode is turned on from the device. This means that user of the application is not able to access the underlying features of the Windows 10 and restricting it to only use the application.

Since it is mandatory to able to read NFC tags in this application, also an NFC reader is needed. The Panasonic Toughpad FZ-G1 that is used in this proof of concept is using an integrated NFC-reader.

***Figure 14.*** *Panasonic Toughpad FZ-G1*

## 4.4 Installation/Removal workflow

In this chapter the workflow of the filter cloth change with the created proof of concept application is covered in order to understand the process and the proof of concept application use environment.

When the filter cloth has reached the end of its lifecycle or has torn due mechanical flaw, the filter operator needs to change the filter cloth. The filter cloth is removed from the filter and placed on a working station, where the operator then proceeds to scan the NFC tag in the filter cloth that was taken from the filter with the data gathering application.

Once successfully scanned the NFC tag, the application will ask the removal reason of that cloth. If user chooses a pre-specified reason in example: ‘Breaking’, the application then proceeds to ask information about the hole locations. After filling the information about removal reason and possible hole locations, the application will ask the current cycle count of the filter that the filter cloth was taken from. Once this information is provided to the application a confirmation is displayed for the user and the removal process is completed. Removal process has one exception behaviour. If the scanned cloth is not installed according to the database, the application offers to make simulated install event for the scanned cloth in order to fix the database. Once this simulated install event in completed, the application will proceed with the removal process normally. This is the only way to proceed when there is no install event for the cloth that is to be removed.

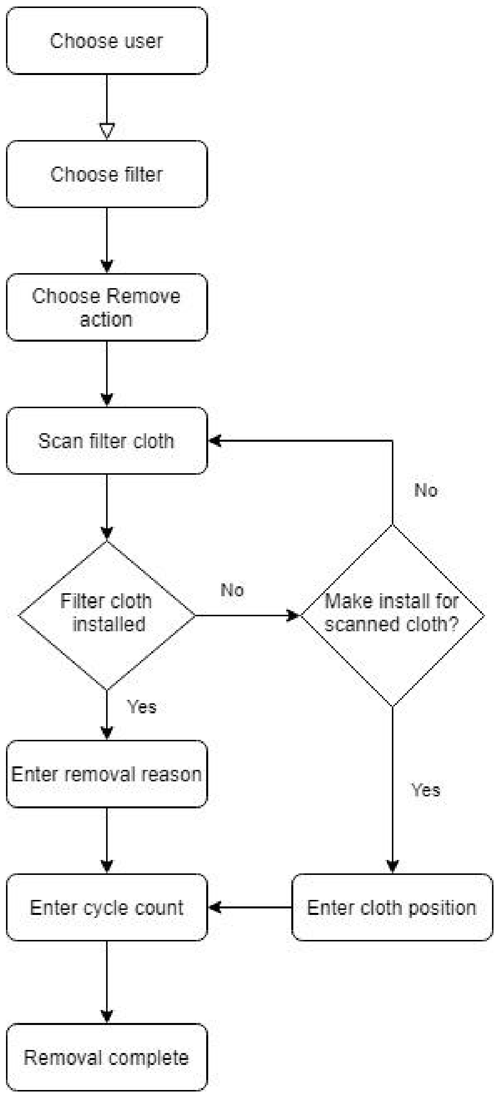


***Figure 15.*** *Removal process*

Once user has successfully removed the filter cloth from the filter with the help of the application, an installation operation must be done for the filter cloth that is to be replaced. This operation is very similar to the removal operation that was described above. Operator scans the NFC tag of the new filter cloth and provides the required information in the application. The required information that needs to be provided to the application in the installation process after successful NFC tag scan are filter cloth plate position, side of the plate that filter cloth is installed and the current filter cycle count number. Once the required information is provided the installation process is complete. Install process has two exception behaviours. First exception is handling the case, where the scanned filter cloth is already installed. In this case the application will ask user to scan again. Second exception is handling the case where the position that the cloth is meant to be installed, is taken according to the database. In order to proceed, the application will offer to make simfigulated removal event for the cloth that is in the desired position. Once this simulated remove process is completed user can proceed with the initial install process.

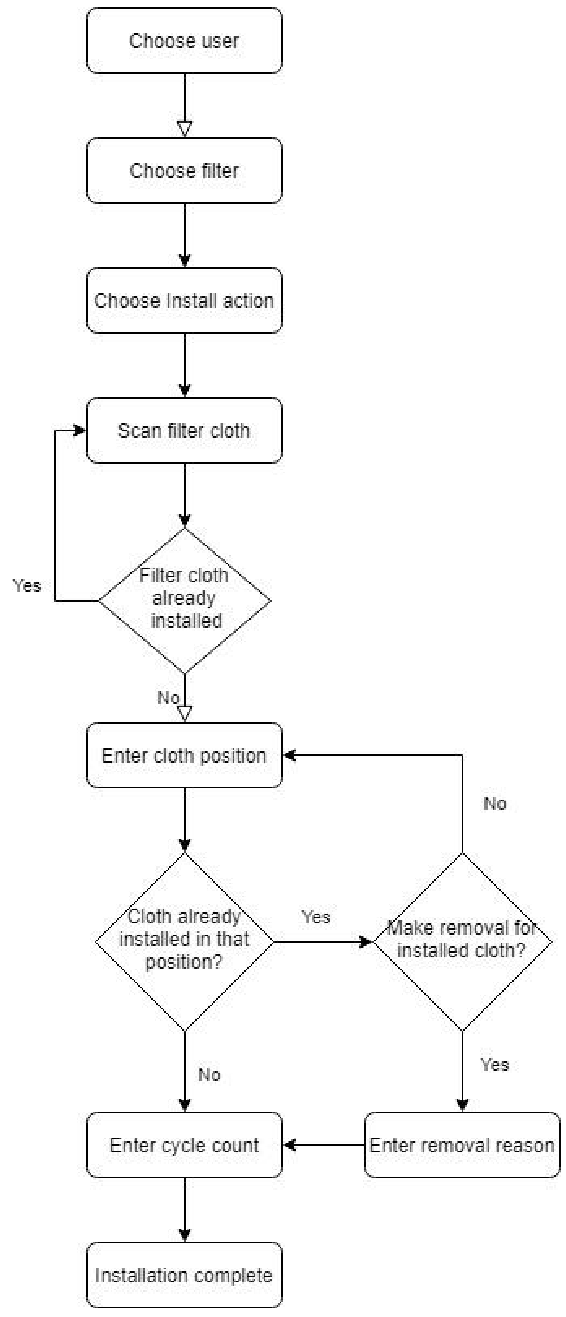


***Figure 16.*** *Install process*

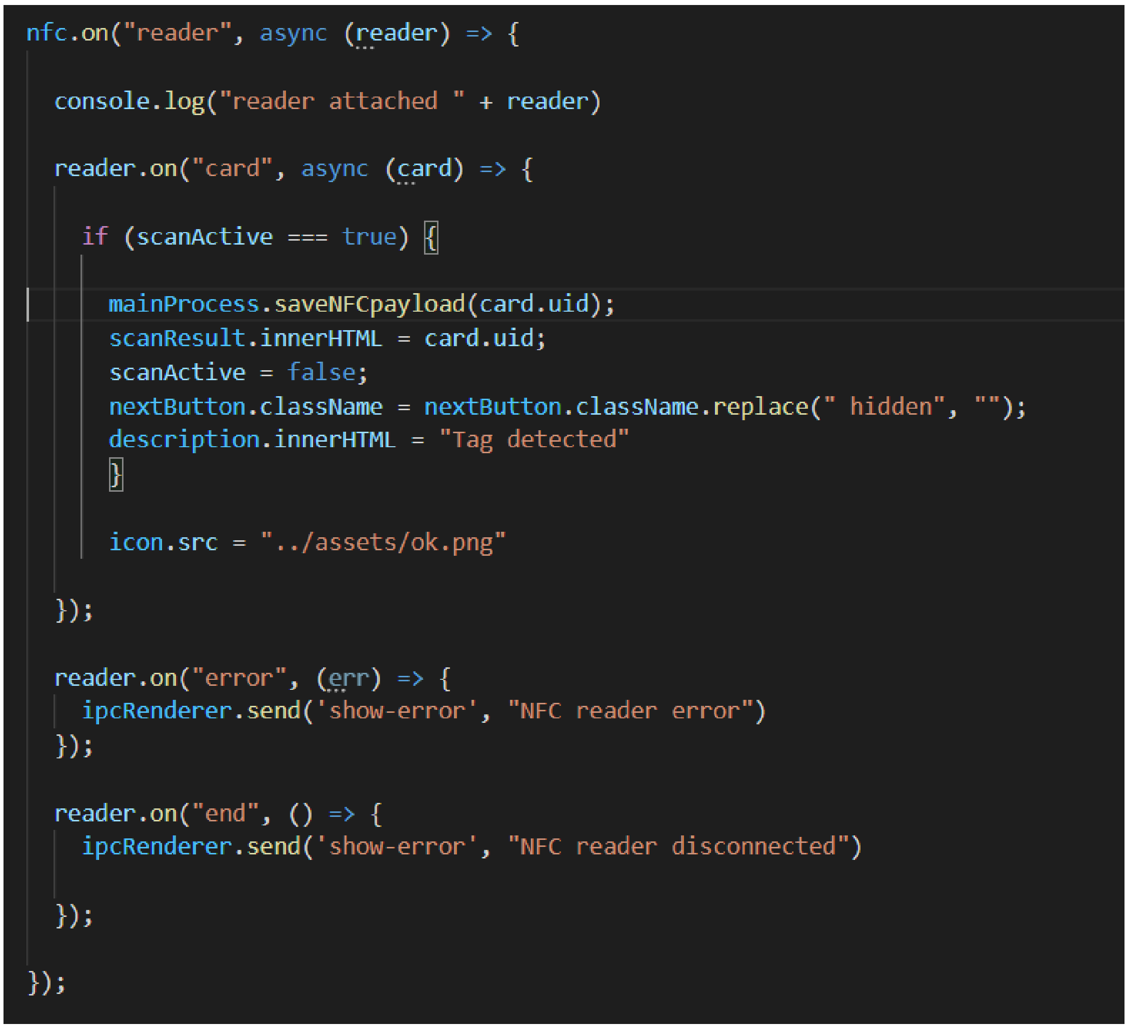
Both processes are independent and can be successfully completed without the other. After both processes a data file containing the provided information about the event is sent to cloud storage, where it is held for later computing.

## 4.5 NFC Module

The ability to read and write NFC tags is one of the major function requirements. At the time of this thesis is written it hasn't been possible to use NFC in browser or via Electron API. This means that in Electron applications to achieve NFC functionality, you have to use native Node module. In this chapter the NFC solution for the proof of concept is covered. The usage of native Node modules in Electron applications is covered in chapter 2.4.3.

The module to implement the NFC functionality is called ‘nfc-pcsc’ (https://github.com/pokusew/nfc-pcsc). It is a small library with weekly downloads ranging from 100 to 350. It uses PCSC lite project via node bindings to achieve access to smart card API in Node environments. This makes nfc-pcsc possible to use in most desktop environments. PCSC lite is a middleware library supported in GNU/Linux, macOS, Solaris, FreeBSD, NetBSD, and OpenBSD operating systems. Since this is a Native Node Module and used in Electron application, it affected the development process significantly.

In figure 17 is a code snippet using the nfc-pcsc module to obtain the identifier of the NFC tag.



```
nfc.on("reader", async (reader) => {

  console.log("reader attached " + reader)

  reader.on("card", async (card) => {

    if (scanActive === true) {

      mainProcess.saveNFCpayload(card.uid);
      scanResult.innerHTML = card.uid;
      scanActive = false;
      nextButton.className = nextButton.className.replace(" hidden", "");
      description.innerHTML = "Tag detected"
      }

      icon.src = "../assets/ok.png"

  });

  reader.on("error", (err) => {
    ipcRenderer.send('show-error', "NFC reader error")
  });

  reader.on("end", () => {
    ipcRenderer.send('show-error', "NFC reader disconnected")

  });

});
```

***Figure 17.*** *Utilizing nfc-pscs module in Electron application to obtain NFC tag identifier*

# 5. EVALUATION

In this chapter we go through the evaluation of the created artifact as described in design science methodology in chapter 3. The three main attributes that are evaluated in this thesis are:

- Software development process
- Resource footprint
- Security

Web NFC API is also introduced briefly as an alternative approach to achieve the same functionality as the proof of concept of this thesis.

## 5.1 Enhancements for Electron development process

Software development with Electron was found easy with low skillset in application development. Electron has gained a lot of interest in the developer community what comes to cross-platform desktop applications. There is lot of community created material regarding Electron and many upcoming problems have already been solved. What comes to Electron software development process, Electron is described as following on the electron.js site: ‘If you can build a website, you can build a desktop app. Electron is a framework for creating native applications with web technologies like JavaScript, HTML, and CSS. It takes care of the hard parts so you can focus on the core of your application’. This sentence was proved to be correct in this thesis. However, there are tools available to enhance the development process in many ways that are not included in the default development environment setup of Electron framework.

In this chapter some enhancements for Electron development are presented. This is being founded on development experience when working on the proof of concept that is the result of this thesis.

### 5.1.1 Electron Forge

Electron Forge is an all-around tool for creating, publishing, and installing Electron applications. Electron Forge was not used in the development of the proof of concept of this thesis. It’s mentioned in Electron’s official website that: ‘The simplest and the fastest way to distribute your newly created app is using Electron Forge.’

Electron Forge is a tool that helps with many development issues that requires lot of work when doing manually. It provides templates for:

- makers
- publishers
- auto updating
- debugging
- code signing
- typescript boilerplates

By starting the proof of concept development with Electron Forge, it would have saved lot of time. The fact that Electron Forge has TypeScript boilerplates, makers, and publishers built in, there is no need to add these items separately, as was done in the proof of concept application. Also, the fact that updating these items had to be done separately rather than updating the Electron Forge tool which maintain these items.

### 5.1.2 TypeScript

The proof of concept that was made as an end-product of this thesis is programmed with JavaScript. The proof of concept is experimental project and so the nature of the code is not very complex. In some cases, Electron projects tend to grow and thus the dynamic stating of JavaScript can cause bugs via faulty states of items. Typescript provides type system and module system that aid programmers to catch errors statically. Every TypeScript program is a JavaScript program. This fact makes it easy to adapt from JavaScript programs. Even though the proof of concept was written in JavaScript it wouldn’t be big task to transform it into TypeScript program

Electron has been supporting Typescript from June 2017 and installations from Electron 1.6.10 includes its own Typescript definition file.

TypeScript is not intended to be a new programming language in its own right, but to enrichen and support JavaScript development. TypeScript adds number of language constructs, such as classes, modules, and lambda expressions. One of the main key design goals of TypeScript is to support JavaScript styles and idioms and to be applicable to majority of existing JavaScript libraries. Bierman et al. [3] states that these goals of TypeScript lead to number of distinctive properties of the type system:

**Full erasure:** The types of a TypeScript program leave no trace in the JavaScript emitted by the TypeScript compiler. Since TypeScript is compiled to JavaScript and it runs in its own run-time, no traces of TypeScript types are shown in run-time. [3]

**Structural types:** TypeScript offers structural type system for JavaScript rather than nominal. This means that types in TypeScript are compared with the structure of the elements rather than the name (in nominal type systems). In JavaScript objects are often built from scratch and not from classes and used based on their expected structure. This fact makes structural type system more suitable for TypeScript. [3]

**Unified object types:** In JavaScript, objects, constructors, and arrays are not separate kind of values. Objects can simultaneously play several of these roles. In addition of describing members in TypeScript objects contain call, constructor and indexing signatures, describing the different ways the object can be used. [3]

**Type inference:** TypeScript relies on type inference to minimize the amount of excess syntax that would accumulate from explicitly provided type annotations from programmers. Often only small number of type annotations need to be given to allow the compiler to infer meaningful type signatures. [3]

**Gradual typing:** Gradual typing allows parts of the program to be dynamically typed and other parts statically typed. TypeScript is allowing this kind of behaviour with distinguished dynamic type *any.* As a result, typing errors not identified statically may remain undetected at run-time. [3]

Adding TypeScript to the development process of this particular proof of concept, would decrease the number of mistakes when it comes to variable types. However, in this proof of concept it was legitimate to implement the proof of concept with vanilla JavaScript as the proof of concept was not very complex. In cases where Electron application is growing out of conceptual into bigger and more complex application, TypeScript is strongly recommended by this thesis.

### 5.1.3 Frontend framework

The frontend of the Electron application that was developed during this thesis was achieved with vanilla JavaScript, HTML and CSS. To create more innovative and complex frontend user interfaces more easily a frontend framework should be considered.

Because the frontend of Electron applications is rendered by browser engine powered by standard web technologies, it is possible to apply any JavaScript library or web framework to enrichen the frontend development process of Electron projects. There are boilerplates created for all major frontend frameworks such as: React, Vue and Angular. These popular JavaScript frontend frameworks can be utilized to create complex interfaces with less work compared to vanilla web technologies. React, Vue and Angular all were included in the Stack Overflow's developer survey: Most Loved Web Frameworks of 2020 [30]. The results of the survey can be seen in the figure 18.

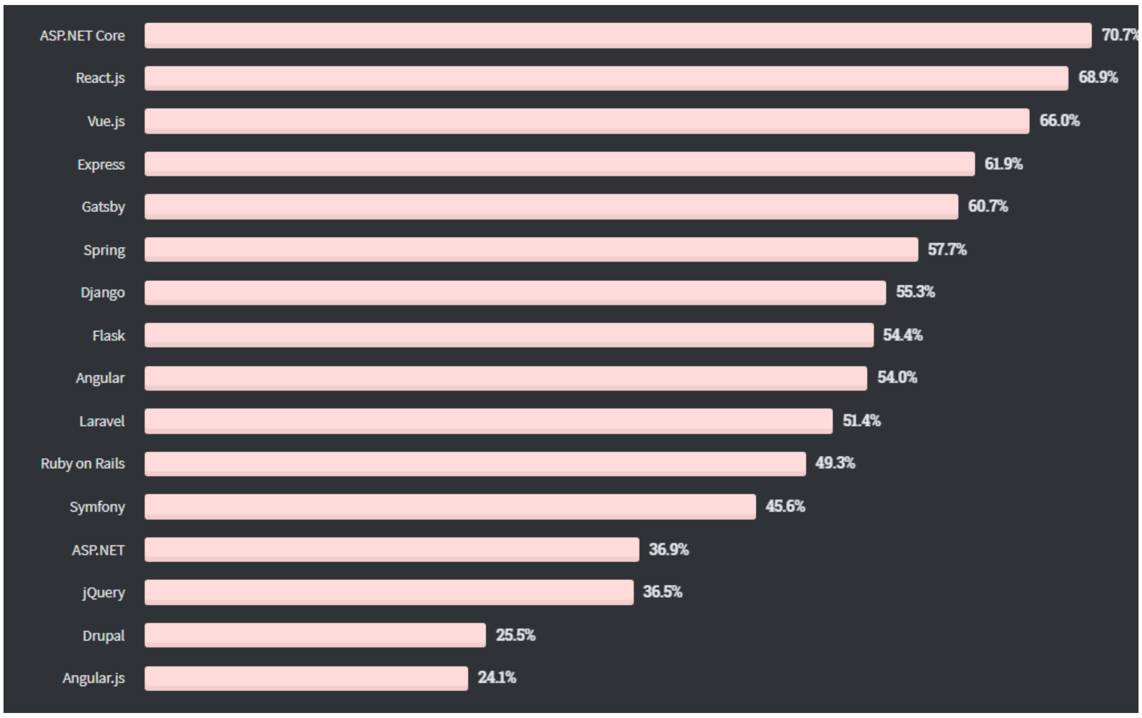


***Figure 18.*** *Stack Overflow Developer Survey 2020. Most Loved Web Frameworks [30]*

Usually, these kinds of frameworks are very lightweight and easy to use, so high amount of skill is not needed to include one in Electron projects. For example, with React you can use a library called React-Bootstrap that allows developer to use wide variety of frontend components such as buttons, alerts, and tooltips etc. Frontend frameworks work great with prototyping which would have fit perfectly with the nature of the proof of concept done in this thesis.

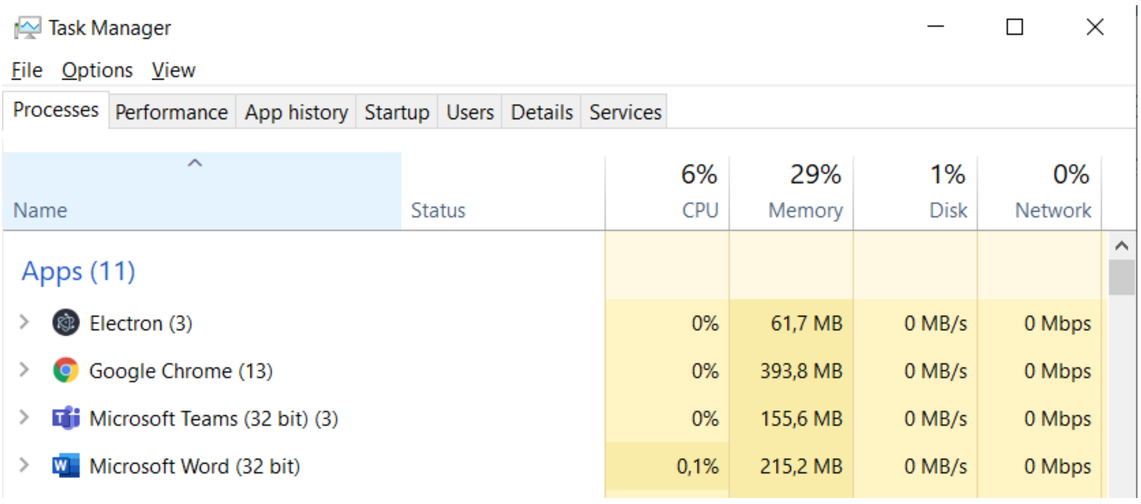
Front-end frameworks tend to give developers advantages in their developing processes JavaScript behind it. Also, frameworks can sometimes be too strict and the time that is gained with using a framework can sometimes be consumed on trying to find out a way to customize a ready component provided by the framework.

The proof of concept of this application didn't have high requirements for polished user experience. However, if the user experience of Electron application is required to be polished and easily achieved, a frontend framework should be considered.

## 5.2 Resource footprint

Electron has bad reputation in performance aspect of applications. Electron is bundled with two big entities: Node.js and Chromium. Every instance of BrowserWindow launches an instance of Chromium. In this proof of concept only one BrowserWindow instance is used but still the amount of memory used is significant.

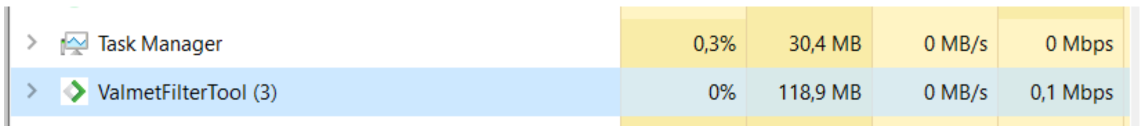
A fresh startup of electron-quick-start repository that was used as a template of the proof of concept, uses approximately 61,7 MB of memory (see figure 19). The build folder size of this repository 150 MB. This folder is containing the two big entities Node.js and Chromium since they are shipped with every with every distribution.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services

| Name | Status | 6% CPU | 29% Memory | 1% Disk | 0% Network |
|---|---|---|---|---|---|
| Apps (11) | | | | | |
| Electron (3) | | 0% | 61,7 MB | 0 MB/s | 0 Mbps |
| Google Chrome (13) | | 0% | 393,8 MB | 0 MB/s | 0 Mbps |
| Microsoft Teams (32 bit) (3) | | 0% | 155,6 MB | 0 MB/s | 0 Mbps |
| Microsoft Word (32 bit) | | 0,1% | 215,2 MB | 0 MB/s | 0 Mbps |

***Figure 19.*** *Fresh instance of electron-quick-start repository memory consumption*

The proof of concept application 'Valmet Filter Tool' is using 118,9 MB of memory in idle state. (See figure 20). The build folder size of this application is 178 MB.

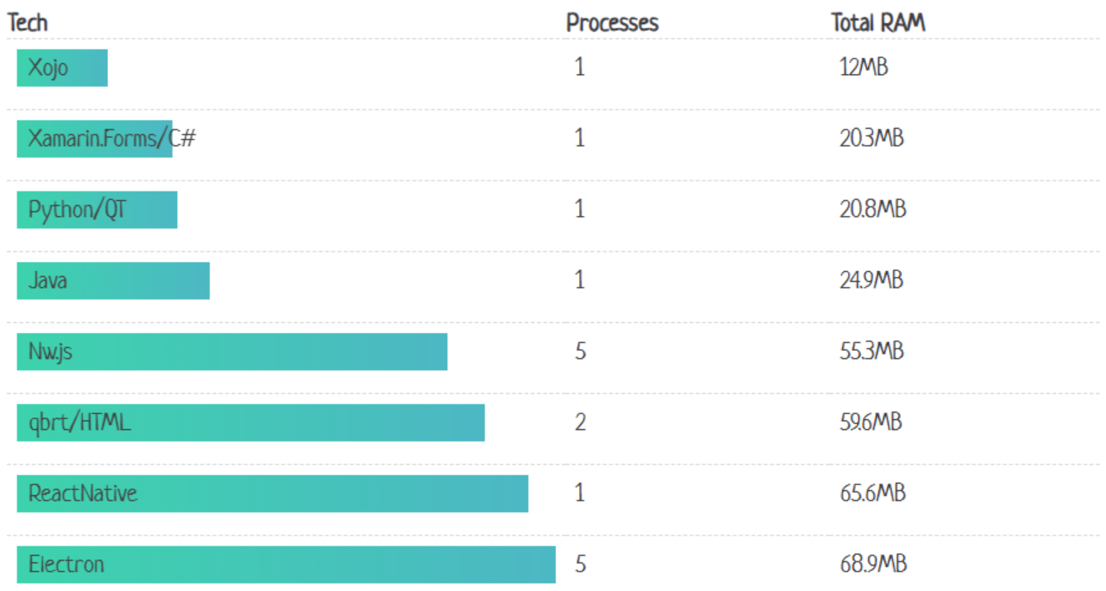


| | | | | |
|---|---|---|---|---|
| Task Manager | 0,3% | 30,4 MB | 0 MB/s | 0 Mbps |
| ValmetFilterTool (3) | 0% | 118,9 MB | 0 MB/s | 0,1 Mbps |

***Figure 20.*** *Valmet Filter Tool Electron application memory consumption*

Both instances are using Electron 6.0.3. and bundled with electron-packager 13.1.1. From this simple comparison it can be concluded that Electron applications tend to use memory heavily when growing in size.

Rory O'Reilly has made a simple Hello world test between cross-platform frameworks including Electron see (figure 21). The test shows that Electron uses the most of RAM memory between the test units. Test units contain non-javascript frameworks but in order to make this test more fair, all frameworks use an HTML String so any given framework must render HTML. [23]



| Tech | Processes | Total RAM |
|---|---|---|
| Xojo | 1 | 12MB |
| Xamarin.Forms/C# | 1 | 20.3MB |
| Python/QT | 1 | 20.8MB |
| Java | 1 | 24.9MB |
| Nw.js | 5 | 55.3MB |
| qbrt/HTML | 2 | 59.6MB |
| ReactNative | 1 | 65.6MB |
| Electron | 5 | 68.9MB |

***Figure 21.*** *Electron 'Hello World' test compared to other cross-platform frameworks. [23]*

These results show that even the leanest Electron application is using a lot of re-sources. The results that are showed in figures 19 and 20 are completed on computer with the following specification:

- Processor: AMD Ryzen 2600X 3,8 GHz
- Graphics Card: NVIDIA GeForce RTX 2060
- Memory: 16 GB DDR4 at 2600 MHz
- Windows 10

When running proof of concept Electron application on computer mentioned above, no issues is caused by high memory allocation. It must be considered that if Electron applications keep gaining popularity this could cause problems to environments where memory is not highly available.

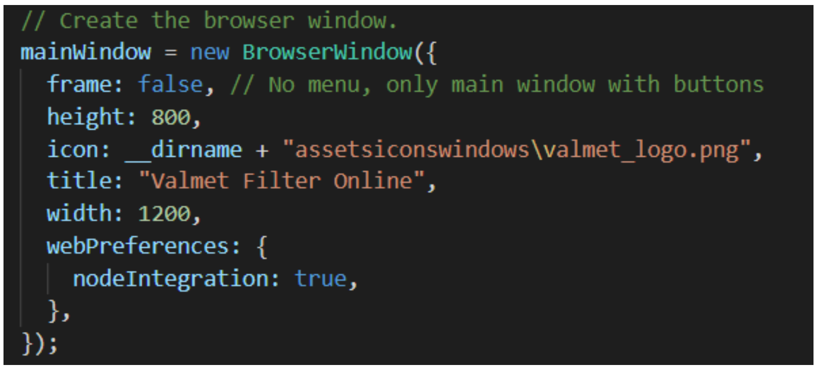
## 5.3 Security

In this chapter some improvements to the security methods for the created artifact are introduced and the overall security of completed artifact is reflected

### 5.3.1 Security and Networking

In this chapter security and network events of the proof of concept are covered. Security aspects are reflected in respect to chapter 2.5.2.

According to the chapter 2.5.2. most of the security risks of Electron applications are related to renderer processes. More precisely the privileges of remote content downloading of the renderer processes. The proof of concept is designed to use one renderer process to show locally stored pages and one minor renderer process to handle admin panel popup. The admin panel renderer process doesn't have access to Node and is sandboxed so it is not a security threat and therefore not discussed further in the matter of security. However, the main renderer process of the application has Node privileges by enabling the nodeIntegration flag in the webPreferences options, and therefore must be considered.



```
// Create the browser window.
mainWindow = new BrowserWindow({
  frame: false, // No menu, only main window with buttons
  height: 800,
  icon: __dirname + "assetsiconswindows\valmet_logo.png",
  title: "Valmet Filter Online",
  width: 1200,
  webPreferences: {
    nodeIntegration: true,
  },
});
```

***Figure 22.*** *Creation of renderer process element in Electron*

All the pages that are loaded into the mainWindow renderer process are stored locally and shipped with the application itself. This means that no recommendations considering downloading remote content is not violated.

However, the proof of concept is using remote files (not running remote scripts e.g., web pages) downloaded with SFTP (SSH File Transfer Protocol). Application runs a checkup cycle every 5 minutes. This means that application is downloading single json file for every filter is installed and one json file containing the configurations for every filter. Also, whenever a filter cloth is installed or uninstalled, compressed CSV (Comma-separated value) file is uploaded. This file contains information about the filter cloth event. Both actions are executed through the main process of Electron which makes these actions secure. Actions are executed through SFTP. SFTP provides user authentication and encryption to files that are transferred. It uses client-server architecture. Client in this case is the proof of concept application and server is Amazon S3 bucket. Authentication of connections is done with public-key cryptography. With all these things considered the data exchange of the application is evaluated secure.

### 5.3.2 Code Signing

Code signing is a process where code or script is digitally signed. By using code signing you are verifying code’s integrity. It binds the code to the authenticated publisher. Code signing can be used in various types of codes including Windows executables, Java JAR files, Android applications and Electron applications. Code signing uses public-key cryptography to compute hash of the code and this hash is signed using publisher’s private key. The public key of the code’s publisher is authenticated using a X.509 code signing certificate. This certificate is given to the publisher by CA (Certification Authority) after verifying the publishers identity. [5]

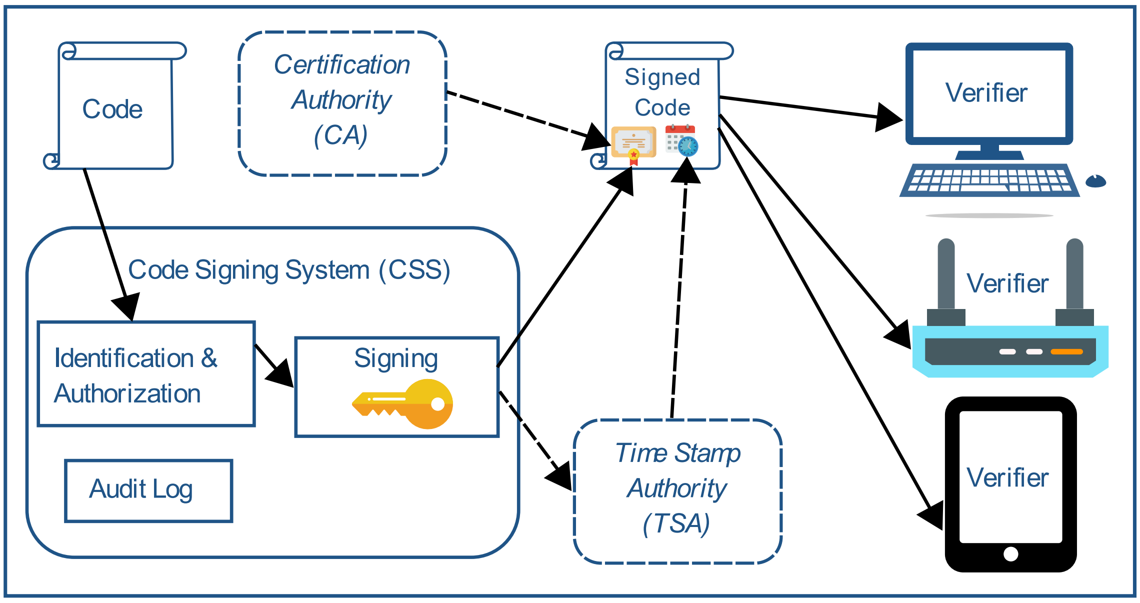

Code
Certification Authority (CA)
Signed Code
Verifier
Code Signing System (CSS)
Identification & Authorization
Signing
Audit Log
Time Stamp Authority (TSA)
Verifier
Verifier

***Figure 23.*** *Code Signing Process. [6]*

In Electron applications code signing process varies depending on OS platform and application build pipeline configuration. In macOS, in the addition of code signing, applications must be also uploaded to Apple for a process called notarization. Notarization is a set of automated systems that further verify that the code signed application is safe to distribute to users.

In order to implement code signing in this proof of concept application few things must be taken in account. Although Electron allows cross-platform option, the application is first deployed in Windows environment and is the only OS platform that is observed in this thesis. For signing Windows builds Windows Authenticode code signing certificate is needed (requires an annual fee). In addition, Visual Studio is needed to get the signing utility. The actual code signing certificate is possible to buy from various resellers. Popular resellers include digicert and Sectigo [6]. Building pipeline defines how to proceed in code signing process once all the required tools are in possession. As described in the software chapter of this thesis, electron-packager is used in this proof of concept application build pipeline. Electron-packager can be used to implement code signing, but it was not completed to this proof of concept application.

## 5.4 Web NFC API

By the time this thesis is written, it hasn't been possible to communicate with NFC devices via web browsers. This is a major feature that defines whether the proof of concept would have been built on browser. However, there is a specification draft for Web NFC

API issued by Web NFC Community Group. This draft is in experimental phase of the Chrome Android. The Web NFC Community group has been working on finding ways to give web sites permission to read and write nearby NFC devices securely since the foundation of the group in 2015. [32]

The Web NFC is limited to a lightweight binary message called NDEF (NFC Data Exchange Format) because security properties of reading and writing NDEF data are more easily quantifiable. NDEF provides standardized method for readers to communicate with NFC devices.

If NFC communication would be possible via browser, this application could have been implemented as web application or PWA (Progressive Web Application) application. PWAs utilizes modern web APIs along with traditional progressive enhancement strategy to create cross-platform web applications. Offline capability is a major design feature of industrial applications. Modern web apps and PWAs are capable to function when the network is unreliable, or even non-existent by using these modern web APIs such as: Service Workers to control page requests, the Cache API for storing responses to network requests offline, and client-side data storage technologies such as Web Storage and IndexedDB to store application data offline. [26]

By implementing an application of the nature of the proof of concept of this thesis as modern web application, the required development skillset would be rather equivalent to the Electron application since both implementations are based on web technologies (HTML, CSS, JavaScript). Although the advantages of an modern web application would be achieved in hardware. The fact that the application would run in web, it would be accessible through many different types of devices rather than a single desktop device that the application is installed on.

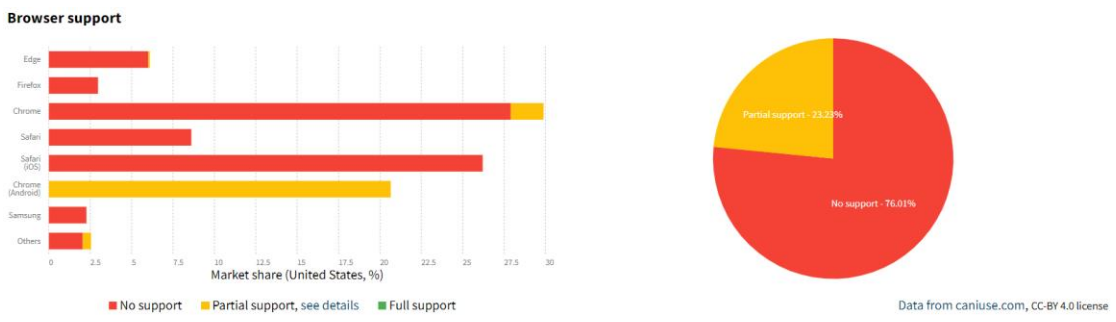


***Figure 24.*** *Browser support for Web NFC API. [32]*

## 5.5 Native Node Module maintainability

The NFC module (nfc-pcsc) used in the proof of concept in this thesis is Native Node module. The characteristics of Native Node module are described in chapter 2.5.7. The use of this module proved to be tedious in development process. As mentioned in the chapter 4.5, the NFC module had weekly downloads ranging from 100 to 350 with quite infrequent updates. This led to a scenario where the internals of the NFC module were not compatible with the corresponding Node.js C++ compiler when updating Electron version from 4.0 to 5.0. With this Electron patch, Electron included Node.js version 12.0.0.

With Node.js version 12.0.0 the data structure v8::Handle was deprecated and v8::Local was meant to replace it. In the NFC module that was used in the proof of concept the old data structure v8::Handle was used. This caused a situation where the NFC module was not compatible with the new Node.js version. The Electron version 5.0.0 with the Node.js version 12.0.0 was released April 24, 2019 and the patch for the updated NFC module was released January 24, 2020. This means that the NFC module was not compatible with the latest version of Electron for 9 months.

The compatibility issue could be solved in 3 different ways:

1. Downgrade to a older Electron version which would also contain older Node.js version that is compatible with v8::Handle.
2. Wait for the NFC module maintainer to patch the data structures. In this case this took 9 months.
3. Modify the NFC module to be compatible with the new Node.js version.

Third option would require the developer to move out of the required skillset that was set for Electron software because the NFC module is written in C++.

The actual solution in the proof of concept was done as a hybrid of options 1 and 3. When the new Electron version was firstly introduced, the NFC module was modified by the developer to be compatible with the new Electron version. Later when upgrading the proof of concept packages, it was noticed that patch was introduced to the NFC module maintainer and the NFC module was updated from the original author.

This scenario caused significant burden on the software development process attribute that was introduced in the chapter 4 and is one evaluated attribute also in terms of research question of this thesis. It made the developer to jump out of the original skillset that was required to develop Electron.

## 5.6 Summary of Evaluation

In this chapter the research questions are answered in more concentrated manner.

Software development process of Electron application is easy to learn. A developer with previous knowledge from web development could easily handle the whole development process of an Electron application. Only case where developer had to move out of the web development stack was when, developer had to modify the required Native Node module. However, it is not mandatory to use Native Node modules in Electron applications. In this proof of concept, it was the only way to achieve the NFC capability. Considering these matters, it is fair to say that Electron development process is easy to learn and not complex.

Security has been improved in Electron lately. This is one main concern when evaluating Electron's readiness as a complete framework. Electron development team has involved lots of default options to the framework that prohibit the use of remote content in Electron applications. Usage of remote content is the main security risk when using Electron applications as this could enable an RCE or XSS attack.

JavaScript used in Electron applications is not compiled language. This means that the source code of the application is available to everyone who gets their hands on the application. Very similar behaviour can be seen in every web application. This may sound dangerous, but with proper configuration and proper security policies this causes no harm in Electron applications.

Resource footprint in Electron applications is the main flaw when usually discussing about Electron. Electron's front-end rendering is based on Chromium. Chromium is notorious about using lot of memory on the devices that it is running on. This doesn't cause issues on medium to high end devices, but on low end devices Electron may cause serious lag.

All in all, Electron is a suitable framework for creating a complete NFC application consisting of front- and backend solutions. The required skillset to develop such application is not high. Electron may not be the best and lightest solution in future as there is promising NFC API for browsers on the making. Also, a different platform than desktop should be considered.

## 5.7 Threats to Validity

This thesis took three years to complete. It is fair to say that this is a long time when evaluating a piece of information technology. For example, during this time the technology that is evaluated in this thesis, the Electron framework, started from version 4.0.2 and is now running version 18.0.1. It must be mentioned that all the covered background research of this thesis may not be exactly as they are today. The results of this thesis could be also a little different with updated version Electron framework. Software development process has improved as more tools are created for Electron development. Electron development team keeps constantly shipping performance and security updates so these may be also better with updated Electron version. However, the structure of Electron is same as presented in this thesis. Electron is still based on same architecture and has the same basic problems that are presented in this thesis.

The research process of this thesis found out to be hard to keep consistent. The fact that this thesis took three years to complete made it hard to keep the studied information on date. The research process was successful to find the answers to the research questions. However, the long period of time that this thesis took made this thesis not ideal.

# 6. CONCLUSION

In this thesis a cross-platform desktop framework called Electron was evaluated. The evaluation was concentrating on three main attributes: software development process, resource footprint and security. To evaluate these attributes a proof of concept artifact was created with Electron framework. Second research question is: 'Is Electron framework suitable for creating a Near Field Communication (NFC) reading application'. This was also answered in this thesis with the help of the proof of concept application. This application was used and evaluated in real customer environment.

In order to systematically evaluate these attributes a design science research process was used. This research methodology provided methods to make this thesis systematic. Before the proof of concept development an literature research was concluded. Literature research quickly showed results dissenting opinions regarding Electron. However, the decision to use Electron was kept.

Since Electron is basically a combination of two major entities: Node.js in the backend and Chromium on the frontend, the capabilities and prowess of the framework are limited. There is lot of Electron's own API's but still the major capabilities are limited by these two components.

Electron has developed significantly in recent years and the developers of the Electron framework have been making sure that the project is here to stay. In December 2019 Electron was added to the OpenJS Foundation. This is one sign of maturity in an open-source project of this kind. By this time, Electron codified its governance structure and invested a lot in formalizing how decisions affecting the entire project are made. By getting added to the OpenJS Foundation Electron also moves into more neutral grounds from single corporate entity and hereby supporting the JavaScript ecosystem. [27]

Development enhancement ideas and other platform solutions were also introduced as a result of this thesis. One worth mentioning is the Web NFC API that is making NFC reading possible via browser.

Research questions were answered in this thesis. The results showed that although Electron is not the lightest solution in order to create NFC application, it was possible with low software development skillset.

Electron provided lot of features such as cross-platform capability, automatic updates and code signing and more that were not implemented in this thesis. Electron’s feature-fulness was not capitalized as well as it could have been in this thesis. Features of Electron framework showed great potential even for big and complex applications.

# 7. REFERENCES

[1] Ars Staff. Microsoft?s new Code editor is built on Google?s Chromium. 2015; Available at: https://arstechnica.com/information-technology/2015/04/microsofts-new-code-editor-is-built-on-googles-chromium/. Accessed Mar 12, 2021.

[2] Beyer C. Electron is Cancer, 2017, Available at: https://medium.com/commit-log/electron-is-cancer-b066108e6c32

[3] Bierman G, Abadi M, Torgersen M. Understanding TypeScript. ECOOP 2014 – Object-Oriented Programming Berlin, Heidelberg: Springer Berlin Heidelberg; 2014. p. 257-281.

[4] Bojinov B, Herron D, Resende D. Node.js Complete Reference Guide. Available at: https://learning.oreilly.com/library/view/nodejs-complete-reference/9781789952117/.

[5] Certified PUP | Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Available at: https://dl-acm-org.lib-proxy.tuni.fi/doi/abs/10.1145/2810103.2813665. Accessed Sep 27, 2021.

[6] Code Signing Architecture: How the Code Signing Process Works. Available at: https://codesigningstore.com/code-signing-architecture-how-it-works

[7] D.Scott A. JavaScript Everywhere. 2020; Available at: https://learning.oreilly.com/library/view/javascript-everywhere/9781492046974/. Accessed Jul 22, 2021.

[8] Desktop App vs Web App: Comparative Analysis. Available at: https://digitalskynet.com/blog/Desktop-App-vs-Web-App-Comparative-Analysis. Accessed Jul 23, 2021.

[9] Documentation | Electron. Available at: https://www.electronjs.org/docs. Accessed Nov 24, 2020.

[10] Electron Apps | Electron. Available at: https://www.electronjs.org/apps. Accessed Nov 25, 2020.

[11] Electron Tutorials, Security https://www.electronjs.org/docs/latest/tutorial/security

[12] Flanagan D. JavaScript : The Definitive Guide : Master the World's Most-Used Programming Language. Sebastopol, CA: O'Reilly Media; 2020.

[13] Freeman A. The Definitive Guide to HTML5. Available at: https://learning.oreilly.com/library/view/the-definitive-guide/9781430239604/. Accessed Nov 25, 2020.

[14] Hevner AR. Design Science in Information Systems Research. 2004; Available at: https://www-proquest-com.lib-proxy.tuni.fi/docview/218119584?OpenUrlRefId=info:xri/sid:primo&accountid=14242. Accessed Dec 28, 2020.

[15] Honton J. Electron Apps Are Getting Safer to Use. 2020; Available at: https://betterprogramming.pub/2020-005-electron-apps-are-getting-faster-and-safer-3c045c39f61f. Accessed Sep 21, 2021.

[16] HTML: HyperText Markup Language. Available at: https://developer.mozilla.org/en-US/docs/Web/HTML. Accessed Nov 25, 2020.

[17] Keith J, Sambells J. DOM Scripting: Web Design with JavaScript and the Document Object Model, Second Edition. 2010; Available at: https://learning.oreilly.com/library/view/dom-scripting-web/9781430233893/. Accessed Nov 25, 2020.

[18] Kinney S. Electron in Action. 2018; Available at: https://learning.oreilly.com/library/view/electron-in-action/9781617294143/. Accessed Nov 25, 2020.

[19] Logan S. Cross-Platform Development in C++. 2007; Available at: https://learning.oreilly.com/library/view/cross-platform-development-in/9780321246424/

[20] Lynch A, Gfeller M. Developing an Electron Edge. Place of publication not identified: Bleeding Edge Press; 2016.

[21] McCann F. Electron isn't Cancer but it is a Symptom of a Disease. Available: https://duckrowing.com/2021/09/04/electron-isnt-cancer-but-it-is-a-symptom-of-a-disease/

[22] Nikiforakis Nea. You are what you include | Proceedings of the 2012 ACM conference on Computer and communications security. Available at: https://dl-acm-org.libproxy.tuni.fi/doi/abs/10.1145/2382196.2382274. Accessed Jul 15, 2021.

[23] O'Kelly R, Electron memory usage compared to other cross-platform frameworks. Available at: http://roryok.com/blog/2017/08/electron-memory-usage-compared-to-other-cross-platform-frameworks/

[24] Peffers K, Tuunanen T, Gengler C, Rossi M, Hui W, Virtanen V, et al. The design science research process: A model for producing and presenting information systems research. Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST 2006 February 24,.

[25] Pike A. The Persistent Gravity of Cross Platform. 2021, Available at: The Persistent Gravity of Cross Platform

[26] Rieseberg Felix. Progressive Web Apps & Electron. 2019; Available at: https://felixrieseberg.com/progressive-web-apps-electron/. Accessed Nov 24, 2020.

[27] Rieseberg F. Electron joins the OpenJS Foundation | Electron Blog. 2019; Available at: https://www.electronjs.org/blog/electron-joins-openjsf. Accessed Apr 23, 2021.

[28] Rieseberg F. Introducing Electron. 2017; Available at: https://learning.oreilly.com/library/view/introducing-electron/9781491996041/. Accessed Nov 25, 2020.

[29] Security, Native Capabilities, and Your Responsibility | Electron. Available at: https://docs.w3cub.com/electron/tutorial/security. Accessed Jun 9, 2021.

[30] Stack Overflow Developer Survey 2019. 2019; Available at: https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019. Accessed Nov 24, 2020.

[31] Using Native Node Modules | Electron. Available at: /docs/tutorial/using-native-node-modules. Accessed Aug 10, 2021.

[32] W3C Community Group Draft Report. 2022. Available at: https://w3c.github.io/web-nfc/

[32] What Web Can Do, NFC. 2022. Available at: https://whatwebcando.today/nfc.html

# APPENDIX A: ELECTRON SECURITY CHECKLIST

1. Only Load Secure Content – Resources that are not included with the Electron application should be loaded using secure protocol like HTTPS.

2. Do not enable Node.js Integration for Remote Content – (default behavior in Electron since 5.0.0.) It is very important that Node.js integration is not enabled for renderer processes. By enabling Node.js in processes that may load remote content, cross-site scripting attacks are more severely more dangerous if the attacker can execute code on the user's computer

3. Enable Context Isolation for Remote Content – (default behavior in Electron since 12). Context Isolation makes sure that developers preload scripts and Electron's internal logic execute in separate context to the website you load in your application. By enabling Context Isolation, it is made sure that websites are not able to access Electron's internals or powerful APIs your preload script has access to.

4. Handle Session Permission Requests From Remote Content – By default Electron will approve all permission requests (camera, notifications, microphone etc.), unless the developer has manually configured a custom handler. This default behavior is often assumed other way around like in web browsers.

5. Do Not Disable WebSecurity – By disabling WebSecurity property you are allowing execution of insecure code from unknown domains.

6. Define a Content Security Policy – Content Security Policy (CSP) adds another layer of protections against cross-site scripting attacks and data injection attacks. By determining a good quality CSP, Electron's asar files are not open for remote data injections.

7. Do Not Set allowRunningInsecureContent to true – By setting allowRunningInsecureContent to true you are enabling websites loaded over HTTPS to execute scripts, CSS, or plugins from insecure sources (HTTP).

8. Do Not Enable Experimental Features – By enabling Experimental Features you are unlocking Chromiums Experimental Features that are not fully supported. Legitimate use cases exist for enabling these features, but unless you are well known what you are doing, it is not recommended to enable Experimental Features.

9. Do Not Use enableBlinkFeatures – This attribute is like previous one. Instead of accessing features of Chromium, you are accessing the rendering engine behind Chromium. It is not recommended to enable this.

10. Do Not Use allowpopups – allowpopups attribute enables pages and scripts loaded in <webview> components to create new BrowserWindows. This is not suggested behaviour by default unless you know a website you are loading needs this feature.

11. Verify WebView Options Before Creation – a <webview> component which live in Document Object Model (DOM) always has its own webPreferences attribute. It is a good design model to observe and make sure that it is not possible that WebView components don't turn off safety features with webPreferences

12. Disable or limit navigation – If your application has no need to navigate unknown pages it is good idea to disable it. It is also possible to limit navigation to known scope of pages if needed with example Node.js parsers.

13. Disable or limit creation of new windows – Very similar principles here than parts 10-12. If you have known set of window elements it is good idea to limit the creation of windows in general as attackers tend to persuade victims to open new windows with more privileges.

14. Do not use openExternal with untrusted content – Electron's Shell API has openExternal method that is very similar to macOS *open* terminal command. It opens given URI with desktop's native utilities. When openExternal is used with untrusted content, it can be used to execute malicious scripts or commands.

15. Use a current version of Electron – This is self-explanatory. When publishing new versions of frameworks, it is always aiming for better security and performance qualities. Since Electron is a combination of two major entities: Chromium and Node.js, Electron updates always update these two as well. Older versions of these tend to contain security issues and exploits that are found throughout