ARCADA

Utveckling av en skrivbordsapplikation med Electron-ramverket

Joni Andersson

Examensarbete
Informationsteknik
2021

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	
Författare:	Joni Andersson
Arbetets namn:	Utveckling av en skrivbordsapplikation med Electron-ramverket
Handledare (Arcada):	Jonny Karlsson
Uppdragsgivare:	

Sammandrag:
Skrivbordsapplikationer utvecklades under en lång tid skilt för varje plattform med hjälp av nativa utvecklingsmetoder. Detta ledde till att en stor del applikationer endast var tillgängliga för Windows operativsystemet, eftersom det är plattformen med flest användare. År 2015 utvecklades dock Electron-ramverket av GitHub som öppen källkod. Syftet med Electron-ramverket var att möjliggöra utvecklandet av plattformsoberoende skrivbordsapplikationer. Idag har Electron redan använts för att utveckla många populära plattformsoberoende skrivbordsapplikationer som till exempel Slack, Skype och Discord. Detta arbete ger läsaren en förståelse för hur Electron-ramverket är uppbyggt och hur det fungerar, ramverkets för-och nackdelar och vad som möjliggör att Electron-applikationer är plattformsoberoende. Arbetet innehåller även ett kapitel som fokuserar på att demonstrera hur utvecklingsprocessen ser ut med ramverket och på så sätt fungerar arbetet även som en guide för andra som vill skapa skrivbordsapplikationer med Electron. Till slut analyseras arbetet och slutsatser presenteras om de olika alternativ som finns för utveckling av applikationer.

Nyckelord:	Electron, skrivbordsapplikation, javascript, nativ, programmering, node.js, ramverk, jämförelse
Sidantal:	33
Språk:	Svenska
Datum för godkännande:	

INNEHÅLL

1 Inledning **5**

1.1 Bakgrund 5
1.2 Syfte och mål 6
1.3 Metoder 6

2 Electron Ramverket **6**

2.1 Electron 6
2.1.1 HTML, CSS och Javascript *7*
2.1.2 Node.js och NPM *7*
2.2 Blink, V8 och Chromium 8
2.3 Arkitekturen av Electron 8
2.3.1 Main process *9*
2.3.2 Renderer process *10*
2.3.3 IPC *10*
2.4 Alternativ till Electron 11
2.4.1 Nativ utveckling *11*
2.4.2 För- och nackdelar med Nativ utveckling *11*
2.4.3 Webbapplikationer *12*
2.4.4 För- och nackdelar med Webbapplikationer *12*
2.5 För- och nackdelar med Electron 13

3 Utveckling med electron **15**

3.1 Ändamål 15
3.2 Verktyg 16
3.3 Installation av Electron-ramverket 16
3.3.1 Electron-applikationens struktur *18*
3.3.2 Implementering av Main process och BrowserWindow *19*
3.3.3 Implementering av en nativ meny *23*
3.3.4 Implementering av Tray *25*
3.3.5 Implementering av Globala tangentbordsgenvägar *27*
3.3.6 Distribution av Electron applikationen *28*

4 Slutledning **30**

Källor **32**

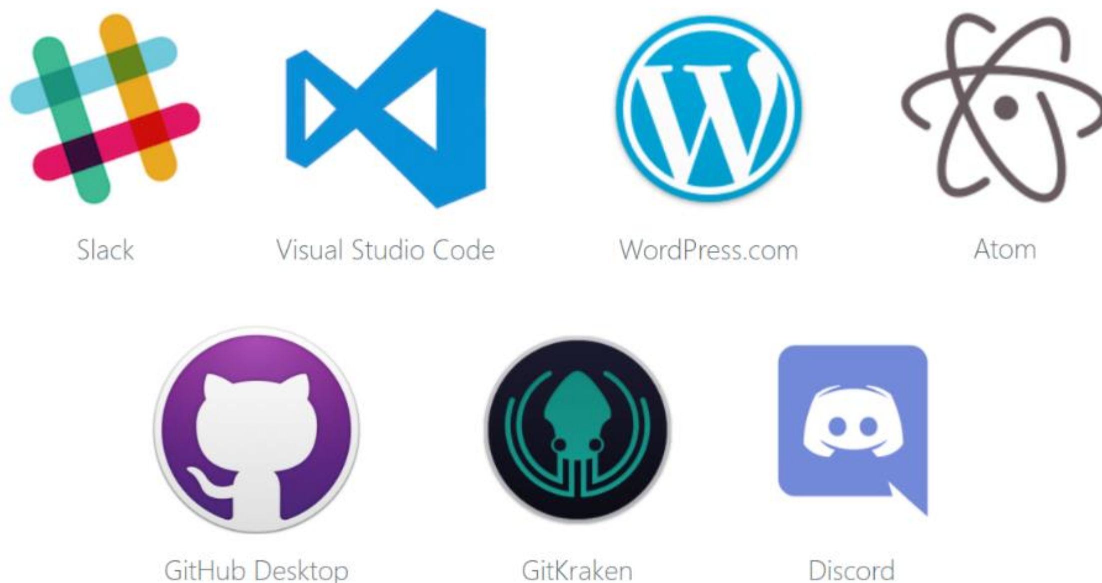
Figurer

Figur 1. Applikationer som utvecklats med hjälp av Electron (Mondal 2020) 5
Figur 2. Bild som demonstrerar hur HTML CSS och JavaScript fungerar ihop. (HTML-CSS-JS The Client-Side Of The Web 2021) 7
Figur 3. En ny BrowserWindow klass skapas inom Main process (electronjs.org 2021) 9
Figur 4. app modulen används för att stänga applikationen om inga fönster längre är öppnade.. 9
Figur 5. Exempel på simpel uppbyggnad av Electron-applikation (Perkaz 2021)......... 10
Figur 6. Användning av ipcMain och ipcRenderer för kommunikation mellan processer. (electronjs.org 2021)... 11
Figur 7. Statistik över marknadsandel för olika operativsystem 2009–2021 (Desktop Operating System Market Share Worldwide 2021) 13
Figur 8. Fråga: Which programming, scripting, and markup languages have you done extensive development work in over the past year, and which do you want to work in over the next year? (2021 Developer Survey 2021) 15
Figur 9. npm init kommandot 17
Figur 10. Skapandet av package.json 17
Figur 11. package.json filens struktur 17
Figur 12. Installering av Electron 18
Figur 13. package.json scripts 18
Figur 14. Strukturen på ett simpelt Electron projekt 19
Figur 15. Kod för grunden av en Electron-applikation 20
Figur 16. Kod för HTML grunden 21
Figur 17. Installation av node-os-utils 21
Figur 18. Kod för att hämta systeminformation med hjälp av modulen node-os-utils... 22
Figur 19. npm start kommandot. 22
Figur 20. Den öppnade Electron applikationen. 23
Figur 21. Menu-klassen hämtas från Electron-modulen. 24
Figur 22. En ny meny skapas enligt egengjord mall. 24
Figur 23. Electron-applikation med nativ meny. 25
Figur 24. Importering av Tray-klassen. 26
Figur 25. Kod för att skapa en System Tray app-ikon. 26
Figur 26. Electron-applikation i System Tray. 26
Figur 27. Kod för att gömma applikationen när användaren minimerar eller stänger. .. 27
Figur 28. Importering av globalShortcut-modulen. 27
Figur 29. Kod för att registrera globala tangentbordsgenvägar med hjälp av globalShortcut.. 28
Figur 30. Kod för att avregistrera globala tangentbordsgenvägar. 28
Figur 31. Installation av Electron Forge. 28
Figur 32. Importering av Electron Forge byggställning. 29
Figur 33. Kommandot för att skapa en installationsfil med Electron Forge. 29
Figur 34. Filer skapade av Electron Forge. 29

1 INLEDNING

1.1 Bakgrund

Under en lång tid har normen för att utveckla skrivbordsapplikationer varit att utveckla applikationer skilt för varje operativsystem. Detta har lett till att skrivbordsapplikationer ofta endast är tillgängliga för Windows som en nativ applikation eftersom det är operativsystemet med flest användare. För mjukvaruföretag är det ofta inte heller från en finansiell synvinkel värt att utveckla sin produkt skilt för mindre använda operativsystem som till exempel MacOS eller Linux, eftersom det kräver ett stort arbete för endast en liten ökning i mängden användare. År 2013 började dock GitHub jobba på ett ramverk med namnet Atom Shell. Syftet med Atom Shell var att bygga ett ramverk som skulle möjliggöra utvecklandet av en plattformsoberoende text-editor. GitHub lyckades med projektet och text-editorn lanserades år 2014 för alla operativsystem under namnet Atom. År 2015 fick Atom Shell sitt nya och nuvarande namn Electron, och ramverket släpptes som öppen källkod. Idag har Electron redan använts för att utveckla många populära plattformsoberoende skrivbordsapplikationer som till exempel Slack, Skype och Discord. (Dryka & Pluszczewska 2020)



Figur 1. Applikationer som utvecklats med hjälp av Electron (Mondal 2020)

1.2 Syfte och mål

Arbetet inleds med en teoretisk del där läsaren får en förståelse för hur Electron- ramverket är uppbyggt och hur det fungerar, ramverkets för- och nackdelar och vad exakt det är som möjliggör att skrivbordsapplikationer utvecklade med Electron automatiskt är plattformsoberoende. Den praktiska delen av arbetet går ut på att demonstrera hur utvecklingsprocessen ser ut med Electron-ramverket. Målet är även att arbetet kunde fungera som en guide för andra som vill skapa skrivbordsapplikationer med Electron.

1.3 Metoder

Electron-ramverkets för- och nackdelar, samt optimala användningsfall kommer utforskas genom att göra jämförelser mellan Electron och alternativa utvecklingsmetoder. Utvecklingsprocessen med Electron-ramverket kommer demonstreras genom att utveckla en simpel nativliknande applikation med hjälp av en del av ramverkets populäraste egenskaper, som bland annat nativa menyer och globala tangentbordsgenvägar, för att så tydligt som möjligt hämta fram fördelarna med Electron-ramverket.

2 ELECTRON RAMVERKET

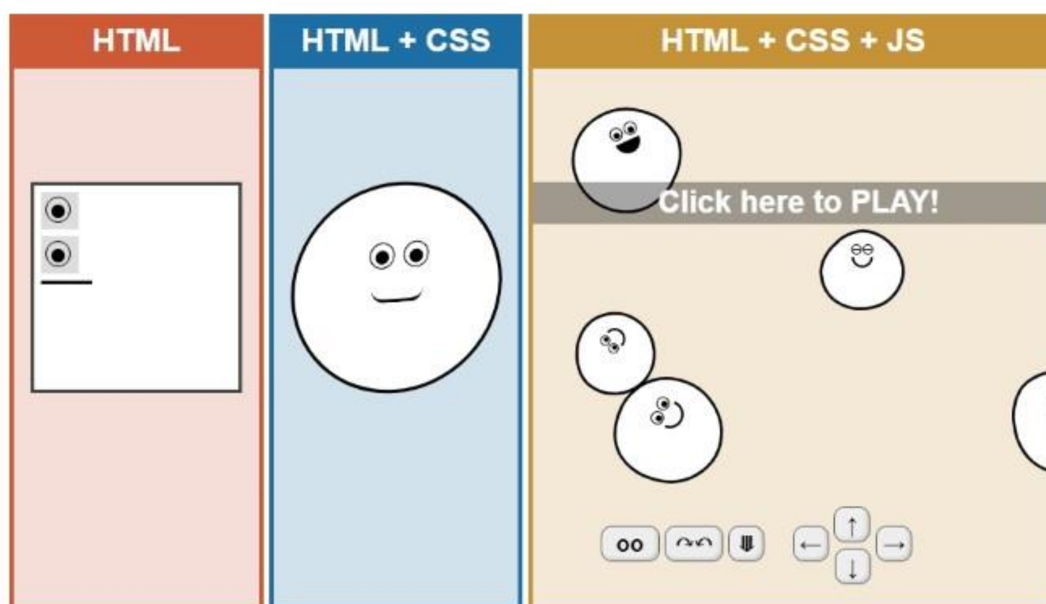
Detta kapitel går igenom hur Electron-ramverket är uppbyggt, hur ramverket fungerar samt vilka alternativ existerar för utveckling av applikationer.

2.1 Electron

Electron (tidigare känd som Atom Shell) är ett open-source ramverk utvecklat av Github. Electron-ramverket möjliggör utvecklandet av plattformsoberoende skrivbordsapplikationer med hjälp av HTML, CSS och Javascript, teknologier vilka förut endast har använts för webbutveckling. Electron-ramverkets back-end består av Node.js och front-end av Chromium. Från en utvecklares synvinkel är alltså Electron-ramverket väldigt lockande, eftersom samma kodbas som använts för webbapplikationer kan återanvändas för att göra en skrivbordsversion av applikationen med endast små ändringar. (Benno 2018)

2.1.1 HTML, CSS och Javascript

HTML, CSS och Javascript är grunden till alla webbsidor och webbapplikationer. HTML står för HyperText Markup Language. Det är alltså ett märkspråk. Med hjälp av HTML byggs strukturen för alla webbsidor genom HTML ”taggar” som berättar för webbläsaren hur innehållet på sidan skall indelas. CSS står för Cascading Style Sheets och är ett stilmallsspråk. Med hjälp av CSS skapas stilregler vilka berättar för webbläsaren bland annat hurdan bakgrund webbsidan skall ha, vilken färg och font texten ska bestå av och så vidare. JavaScript är ett skript- och programmeringsspråk som används för att skapa interaktivitet och funktioner till webbsidor, som till exempel validering av data eller för att visa mera information när användaren för muspekaren över en ett visst HTML-element. (Niekerk 2017) Figur 2 demonstrerar på ett tydligt sätt hur HTML, CSS och Javascript fungerar ihop.



Figur 2. Bild som demonstrerar hur HTML CSS och JavaScript fungerar ihop. (HTML-CSS-JS The Client-Side Of The Web 2021)

2.1.2 Node.js och NPM

Node.js är en open-source, plattformsoberoende exekveringsmiljö för utveckling av applikationer på servernivå och nätverksapplikationer. Node.js applikationer utvecklas med JavaScript-kod och V8 JavaScript-motorn används för att kompilera JavaScript till

maskinkod. Eftersom Node.js körs på servernivå möjliggör det utvecklandet av bland annat inloggningssystem och applikationer som anpassar data enligt användaren.

NPM (Node Package Manager) är default pakethanteraren för Node.js och har världens största mjukvaruregister. (npmjs.com 2021) NPM används under utvecklingen av JavaScript baserade applikationer för att hämta färdiga kodpaket som användare har frivilligt delat med sig. Detta underlättar utvecklingsprocessen betydligt eftersom utvecklaren inte behöver lösa problem som någon annan redan tidigare har löst och delat med sig av lösningen.

2.2 Blink, V8 och Chromium

Blink är en renderingsmotor utvecklad av Google. Uppgiften för en renderingsmotor är att ta in data i form av till exempel HTML och CSS, för att sedan visa den renderade datan i rätt format på skärmen. Alla webbläsare använder en renderingsmotor för att kunna presentera hemsidor. Som exempel använder Mozilla renderingsmotorn Gecko, Apple använder WebKit och Google använder givetvis sin egen produkt Blink.

V8 är namnet på den så kallade ”Javascript-motorn” som används inuti Blink för att kompilera Javascript till maskinkod.

Chromium är i sin tur en kodbas för webbläsare som är skapad huvudsakligen av Google från öppen källkod. Förutom Electron baserar även flera stora applikationer sig på Chromium, bland annat Google Chrome och Microsoft Edge.

Chromium är alltså en webbläsare som sluter samman Blink och V8 genom att använda V8 JavaScript-motorn för att kompilera JavaScript medan renderingsmotorn Blink ansvarar för hur sidan presenteras till användaren. (Biro 2019)

2.3 Arkitekturen av Electron

Detta kapitel går igenom hur Electron-ramverket är uppbyggt och hur ramverket fungerar.

2.3.1 Main process

Alla Electron-applikationer har en Main process, alltså ”huvudprocess” vilken fungerar som ingångspunkt för applikationen. Main process körs i en Node.js miljö, vilket möjliggör användning av alla Node.js API:n.

Huvudrollen för Main process är att hantera applikationens olika fönster med hjälp av *BrowserWindow*-modulen. Varje instans av *BrowserWindow*-klassen skapar ett nytt fönster i en separat Render Process. I Figur 3 nedan är grundläggande koden för att skapa ett nytt fönster med hjälp av *BrowserWindow*-klassen.

```
main.js

const { BrowserWindow } = require('electron')

const win = new BrowserWindow({ width: 800, height: 1500 })
win.loadURL('https://github.com')

const contents = win.webContents
console.log(contents)
```

Figur 3. En ny BrowserWindow klass skapas inom Main process (electronjs.org 2021)

Main process kontrollerar även applikationens livscykel genom *app*-modulen som finns inuti Electron. Koden i Figur 4 använder *app*-modulen för att avsluta applikationen när alla fönster är stängda.

```
// Stäng applikationen om inga fönster är öppna (förutom på macOS plattformen)
app.on("window-all-closed", () => {
  if (process.platform !== "darwin") {
    app.quit();
  }
});
```

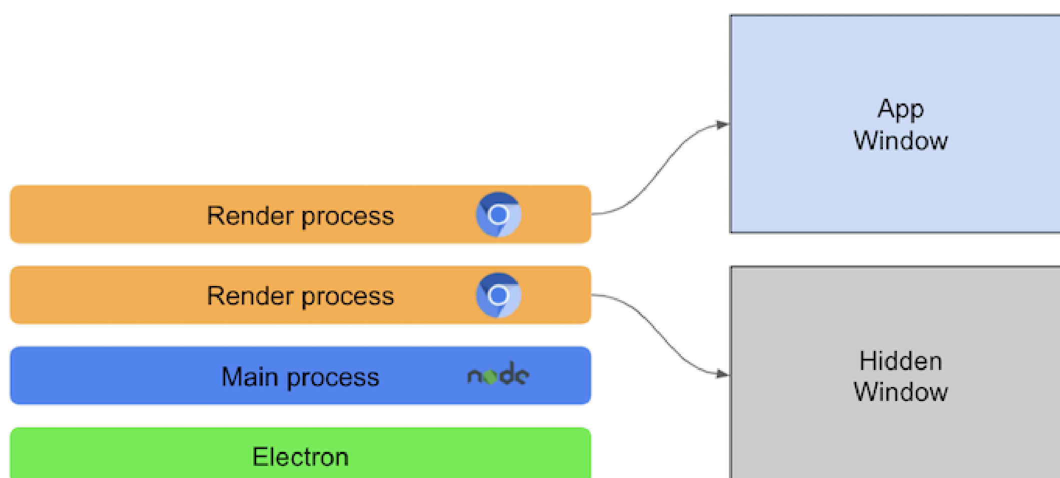
Figur 4. app modulen används för att stänga applikationen om inga fönster längre är öppnade.

Main process innehåller dessutom egna API:n som gör det möjligt att interagera bland annat med användarens operativsystem och hårdvara. (electronjs.org 2021)

2.3.2 Renderer process

Electron skapar en separat Renderer process, alltså ”renderingsprocess” för varje BrowserWindow (fönster) i applikationen. Som namnet antyder är Render process ansvarig för att rendera webb-innehåll som HTML och CSS med hjälp av renderingsmotorn Blink för att sedan presentera innehållet på skärmen. (electronjs.org 2021)

Till skillnad från Main process kan en applikation alltså ha flera Render processer. Varje Render process i applikationen körs isolerat i sina egna fönster, men alla Render process fönster måste inte vara synliga användargränssnitt. (Perkaz 2021) Figur 5 innehåller ett exempel på strukturen av en simpel Electron-applikation.



Figur 5. Exempel på simpel uppbyggnad av Electron-applikation (Perkaz 2021)

2.3.3 IPC

Render processer i Electron kan ses som olika flikar i en webbläsare. För att skilda processer i applikationen ska kunna kommunicera sinsemellan använder Electron Inter-Process Communication (IPC). Electron kommer färdigt med två IPC moduler för att hjälpa till med kommunikationen mellan processerna, ipcMain och ipcRenderer. ipcMain används för att kommunicera från Main process till Renderer processen, medan ipcRenderer används för kommunikation från Renderer processen till Main process. (khanna 2021) Figur 6 innehåller grundläggande koden för kommunikation mellan processer med hjälp av ipcMain och ipcRenderer.

```
// In main process.
const { ipcMain } = require('electron')
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.reply('asynchronous-reply', 'pong')
})

ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.returnValue = 'pong'
})
```

```
// In renderer process (web page).
// NB. Electron APIs are only accessible from preload, unless contextIsolation is disabled.
// See https://www.electronjs.org/docs/tutorial/process-model#preload-scripts for more details.
const { ipcRenderer } = require('electron')
console.log(ipcRenderer.sendSync('synchronous-message', 'ping')) // prints "pong"

ipcRenderer.on('asynchronous-reply', (event, arg) => {
  console.log(arg) // prints "pong"
})
ipcRenderer.send('asynchronous-message', 'ping')
```

Figur 6. Användning av ipcMain och ipcRenderer för kommunikation mellan processer. (electronjs.org 2021)

2.4 Alternativ till Electron

Detta kapitel går igenom de mest populära alternativen till Electron för utvecklandet av applikationer.

2.4.1 Nativ utveckling

Nativ utveckling, alltså utvecklandet av lokala applikationer för en specifik plattform betyder att utvecklaren använder programspråk och verktyg som endast fungerar på en viss plattform. Som exempel kan man välja att göra en applikation för Windows med hjälp av .net ramverket, men applikationen kommer inte fungera på bland annat macOS och Linux, så för att nå användare som inte använder Windows skulle utvecklaren vara tvungen att skapa nya versioner för dessa plattformar.

2.4.2 För- och nackdelar med Nativ utveckling

Fördelar:

- Bättre säkerhet. Eftersom Nativa applikationer lagrar användarens data lokalt på användarens egen dator är det svårare att få obehörig åtkomst till datan.

- Prestanda. Eftersom Nativa applikationer har åtkomst till Nativa API:n vilka är optimerade för specifika plattformar. Nativa applikationer blir heller inte påverkade av dålig internetanslutning.
- Fungerar offline. Eftersom Nativa applikationer inte hämtas från internet kan de användas även utan internetanslutning.

Nackdelar:

- Fungerar endast på en specifik plattform. För att nå användare på andra plattformar krävs en helt ny kodbas.
- Kräver installation. Installation av applikationen kräver både tid och utrymme på användarens dator.
- Svårare underhåll. Nativa applikationer kräver att användaren laddar ner uppdateringar för applikationen. (Desktop App vs Web App: Comparative Analysis 2020)

2.4.3 Webbapplikationer

Webbapplikationer är till skillnad från nativa applikationer lagrade på en server och presenterade via en webbläsare för användning. Eftersom webbapplikationer fungerar i webbläsaren är de naturligtvis plattformsoberoende. Webbapplikationer kräver då alltså inte skapandet av nya versioner för olika plattformar.

2.4.4 För- och nackdelar med Webbapplikationer

Fördelar:

- Plattformsoberoende. Eftersom Webbapplikationer fungerar i webbläsaren kan de användas oberoende av plattform, ofta även via mobiltelefoner och tabletter.
- Underhåll. Webbapplikationer kräver inte användaren att uppdatera applikationen eftersom uppdateringar är gjorda på servern så alla användare använder alltid den nyaste version av applikationen.
- Förmånligare. Eftersom Webbapplikationer är plattformsoberoende och samma kodbas till och med kan fungera på alla de populäraste enheterna sparar utvecklaren stort på resurser.

Nackdelar:

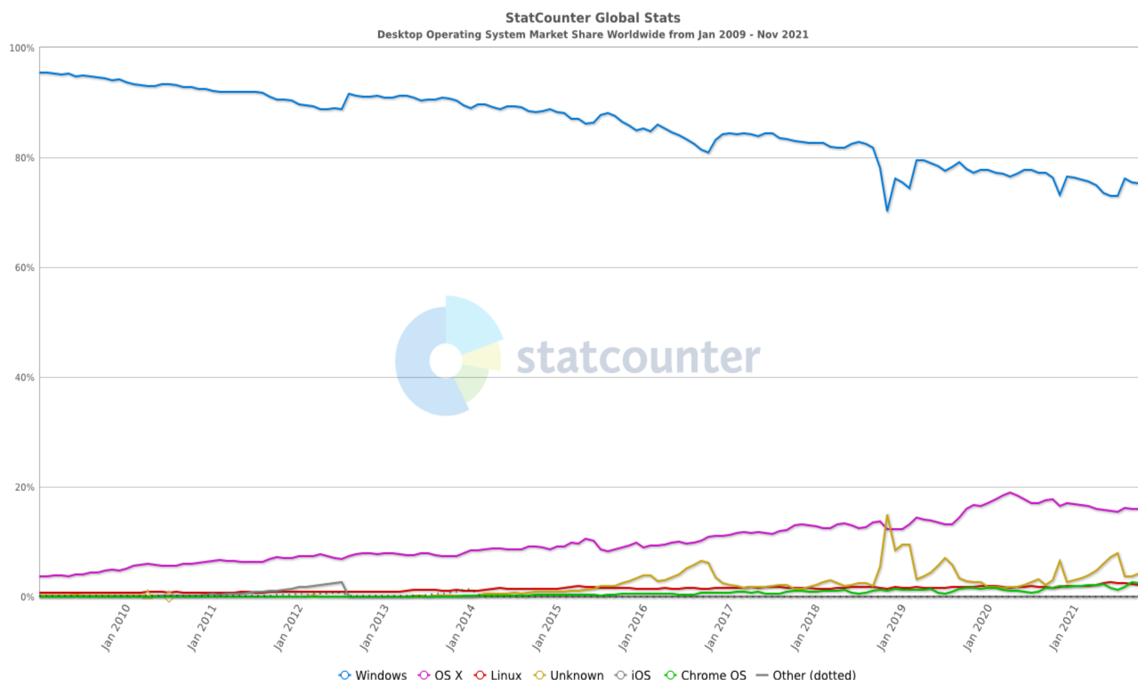
- Prestanda. Webbapplikationens prestanda beror bland annat på kvalitén av internetanslutningen.
- Tillgänglighet. Webbapplikationer kräver internetanslutning samt att servern som håller i gång applikationen är tillgänglig. (Roor 2021)

2.5 För- och nackdelar med Electron

Eftersom Electron-applikationer är skrivbordsapplikationer som baserar sig på webbteknologier kan man se Electron som alternativet mitt emellan Nativ utveckling och Webbapplikationer. Detta delkapitel innehåller jämförelser mellan Nativ utveckling, Webbapplikationer och utveckling med Electron. Fokus kommer vara på vilka för- och nackdelar Electron har gemensamt med dessa alternativ.

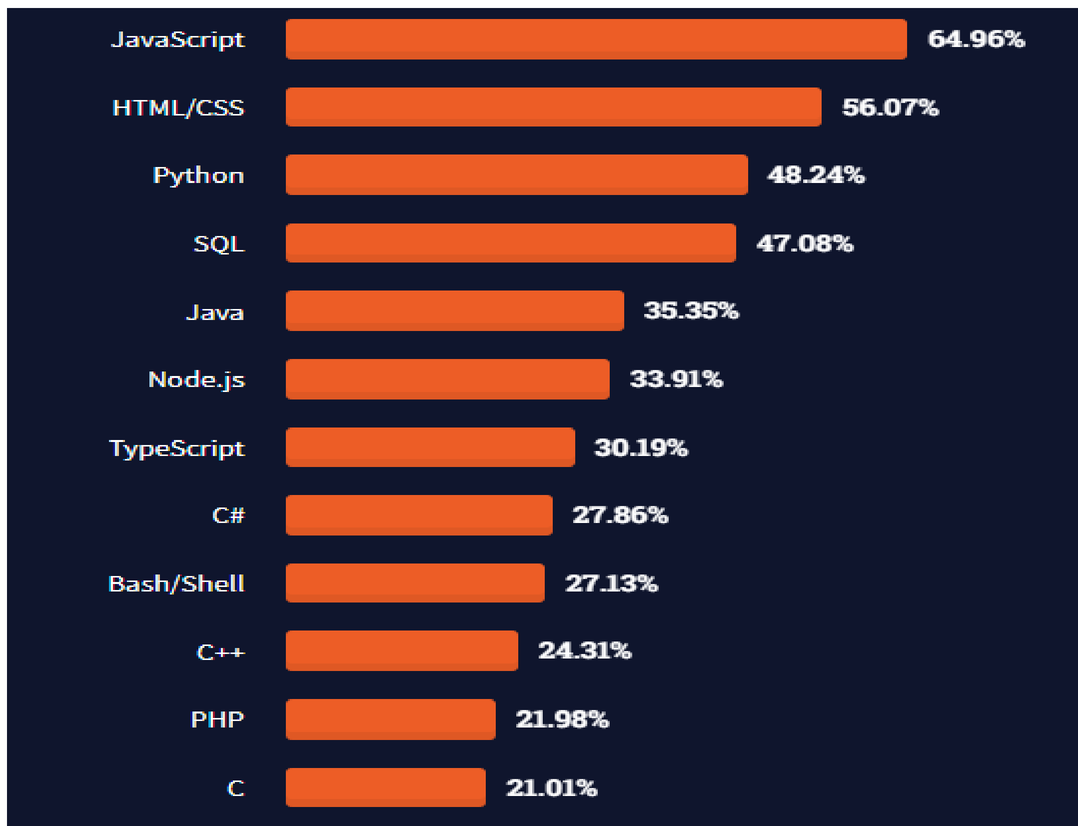
Fördelar:

- Plattformsoberoende. Eftersom Electron-applikationer baserar sig på Chromium kan de lika som Webbapplikationer användas oberoende av plattform. Värdet av att kunna erbjuda plattformsoberoende applikationer blir hela tiden högre, eftersom operativsystemet Windows som länge varit på toppen långsamt men säkert har mistat marknadsandel till andra operativsystem som macOS och Linux. Från Figur 7 nedan kan man tydligt se trenden över Windows sjunkande marknadsandel.



Figur 7. Statistik över marknadsandel för olika operativsystem 2009–2021 (Desktop Operating System Market Share Worldwide 2021)

- Förmånlighet. Samma kodbas för alla plattformar. Eftersom Electron-applikationer baserar sig på samma teknologier som Webbapplikationer kräver det även betydligt mindre resurser att erbjuda applikationen både som skrivbordsapplikation och Webbapplikation.
- Tillgång till nativa API:n. Det som mest skiljer åt Electron från Webbapplikationer är tillgången till plattformens nativa API:n. Med Electron kan utvecklare interagera med bland annat hårdvara, nätverksportar, kamera samt användarens filsystem. (Chandra 2018)
- Fungerar offline. Lika som Nativa applikationer kan även Electron-applikationer användas utan internetanslutning, eftersom applikationen körs lokalt.
- Låg inlärningskurva. Med tanke på att bland annat användare på en av de största hemsidorna för programmeringshjälp Stackoverflow.com år 2021 använde mest Javascript av alla programspråk, kan man anta, att det finns fler potentiella färdigt kunniga utvecklare för Electron-applikationer jämfört med Nativa applikationer vilka kräver plattform-specifik programkod. (stackoverflow.com 2021)
  Från Figur 8 kan man se att majoriteten av utvecklare just nu är intresserade av att utveckla med webbteknologier.



Figur 8. Fråga: Which programming, scripting, and markup languages have you done extensive development work in over the past year, and which do you want to work in over the next year? (2021 Developer Survey 2021)

Nackdelar:

- Storlek. Absolut största nackdelen för Electron är storleken på applikationen. Eftersom Electron använder Chromium kommer även den mest simpla Electron-applikationen med en storlek på minst ~120MB. (Nalegave 2018)
- Lika som Nativa applikationer kan inte heller Electron-applikationer installeras på bland annat mobiltelefoner och tabletter.

3 UTVECKLING MED ELECTRON

Detta kapitel av arbetet kommer fungera som en demonstration och guide över hur utvecklingsprocessen ser ut med Electron-ramverket.

3.1 Ändamål

Ändamålet med detta kapitel är att läsaren får en grundläggande bild över hur utvecklingsprocessen ser ut med Electron, samt att fungera som en guide så arbetet kunde an-

vändas för att läsaren själv kunde komma i gång med utveckling av skrivbordsapplikationer med Electron-ramverket.

Målet är att slutprodukten kommer vara en skrivbordsapplikation som möjliggör övervakning av systemets processor och primärminne oberoende av operativsystem. För att fokusera på Electron-ramverkets funktioner och fördelar valdes ett projekt som så tydligt som möjligt hämtar fram vad som skildrar Electron-applikationer från Nativa- och Webbapplikationer. Avsikten är att uppfylla detta genom att med teknologier som länge har setts som webbteknologier skapa en skrivbordsapplikation med egenskaper som länge endast varit möjliga att implementera via plattformens nativa programkod och verktyg. Av dessa egenskaper kommer projektet innehålla implementering av bland annat en nativ meny, globala tangentbordsgenvägar samt möjligheten att gömma applikationen i bakgrunden genom att ställa den inuti systemfacket/meddelandefältet på operativsystemets aktivitetsfält. Demonstrationen kommer gå igenom de viktigaste stegen ända från installation till distribution av applikationen.

3.2 Verktyg

För att göra demonstrationen av Electron-ramverket så tydligt som möjligt, kommer inte andra verktyg användas än de som hör till grunden av Electron-applikationer, alltså HTML, CSS, JavaScript/Node.js samt NPM för att hämta kodpaket. Som texteditor för projektet valdes Electron-baserade Visual Studio Code.

3.3 Installation av Electron-ramverket

För att kunna börja utveckla med Electron-ramverket måste man först installera Node.js och NPM. NPM kommer med i samma paket när man installerar Node.js. Efter det kan man öppna en tom mapp för projektet och öppna terminal/kommandotolken i mappen.

Första steget är att med hjälp av NPM skapa en package.json fil. Filen ger information om applikationen som bland annat vilket skript används som startpunkt för applikationen samt vilka NPM-moduler applikationen använder. package.json filen skapas med hjälp av kommandot i Figur 9.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm init
```

Figur 9. npm init kommandot

NPM ber sedan användaren fylla i grundläggande information om applikationen innan package.json filen skapas enligt Figur 10.

```
package name: (syscheck)
version: (1.0.0)
description: Ett verktyg för att övervaka användning av systemets CPU och minne.
entry point: (index.js) main.js
test command:
git repository:
keywords: Electron
author: Joni Andersson
license: (ISC)
About to write to C:\Users\v0id\Documents\Slutarbete\SysCheck\package.json:
```

Figur 10. Skapandet av package.json

I *Entry point* frågan fyller användaren i vilken fil fungerar som startpunkt för applikationen. För Electron-applikationer är *main.js* standardnamnet för huvudprocessen.

Efter att användaren har matat in informationen skapar NPM package.json filen, vilken vid detta skede ser ut som i Figur 11.

```
{} package.json ●
{} package.json > ...
1  {
2    "name": "syscheck",
3    "version": "1.0.0",
4    "description": "Ett verktyg för att övervaka användning av systemets CPU och minne.",
5    "main": "main.js",
     ▷ Debug
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [
10     "Electron"
11   ],
12   "author": "Joni Andersson",
13   "license": "ISC"
14 }
```

Figur 11. package.json filens struktur

Nu när vi har skapat en package.json fil kan vi börja hämta npm-moduler, alltså färdiga kodpaket. Första och viktigaste är då förstås själva Electron-ramverket. Detta görs med hjälp av kommandot i Figur 12.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm install electron

> electron@16.0.4 postinstall C:\Users\v0id\Documents\Slutarbete\SysCheck\node_modules\electron
> node install.js

npm WARN syscheck@1.0.0 No repository field.

+ electron@16.0.4
added 86 packages from 97 contributors and audited 86 packages in 9.098s
```

Figur 12. Installering av Electron

Från outputen kan vi se att Electron kräver 86 olika kodpaket från mjukvaruregistret som NPM uppehåller. Alla 86 paket inklusive Electron blev installerat på 9 sekunder.

Nu om package.json öppnas på nytt, ser man att det har genererats en ny lista med namnet ”dependencies”, vilken vid detta skede endast innehåller version 16.0.4 av Electron. Dependencies, alltså beroenden, betyder vilka moduler applikationen kräver att installeras innan applikationen kan fungera. Efter att moduler har installerats, skapas en *node_modules* mapp, där alla kodpaket förvaras.

För att starta en Electron applikation krävs ännu ett start-skript i package.json filens ”scripts” lista som kan ses i Figur 13.

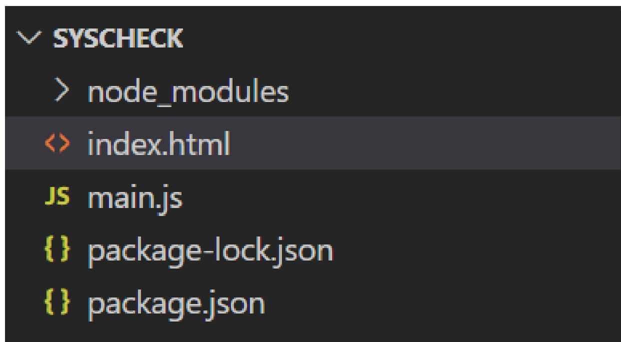
```
"scripts": {
  "start": "electron ."
},
```

Figur 13. package.json scripts

Kommandot berättar för Electron att starta applikationen i roten av projektmappen. Start-skripten kan nu startas med hjälp av *npm start* kommandot, men eftersom projektet inte ännu innehåller något att starta skulle det vid detta skede bara orsaka errormeddelanden.

3.3.1 Electron-applikationens struktur

Ett simpelt Electron-projektet innehåller filerna som kan ses i Figur 14.



Figur 14. Strukturen på ett simpelt Electron projekt

Filen *main.js* fungerar som startpunkt för applikationen och kommer bland annat innehålla koden för huvudprocessen samt koden som hanterar applikationens livscykel.

Filen *index.html* kommer innehålla strukturen för applikationens grafiska användargränssnitt.

Filen *package.json* innehåller information om applikationen som till exempel vilka NPM-moduler applikationen är beroende av.

Mappen *node_modules* innehåller de installerade NPM-modulerna.

3.3.2 Implementering av Main process och BrowserWindow

Koden i Figur 15 skapar grunden för en Electron-applikation.

```
JS main.js ×
JS main.js > ...
1   const { app, BrowserWindow } = require('electron')
2
3   // Funktion för att skapa en BrowserWindow (Ett fönster)
4   function createWindow() {
5       const mainWindow = new BrowserWindow({
6           title: 'sysCheck',
7           width: 355,
8           height: 500,
9           resizable:false,
10          webPreferences: {
11              nodeIntegration: true,
12          },
13      })
14      mainWindow.loadFile('./index.html')
15  }
16
17  /* app-modulen används för att granska när applikationen har laddat klart
18  för att sedan kalla på createWindow funktionen */
19  app.on('ready', () => {
20      createWindow()
21  })
22
23  /* app-modulen används för att granska när alla fönster är stängda
24  för att sedan sluta köra applikationen (förutom på macOS) */
25  app.on("window-all-closed", () => {
26      if (process.platform !== "darwin") {
27        app.quit();
28      }
29  });
```

Figur 15. Kod för grunden av en Electron-applikation

Från Electron-modulen hämtas alltså *app* och *BrowserWindow* klasserna. *BrowserWindow* används för att skapa ett fönster och ladda in HTML-filen till fönstret, medan *app* används för att hantera applikationens livscykel. Orsaken att app inte avslutar applikationen på macOS även om alla fönster är stängda, är för att följa macOS nativa egenskap att inte avsluta applikationer innan användaren uttryckligen väljer att stänga genom att trycka CMD + Q.

Figur 16 nedan visar HTML-filen som laddas in.

```html
index.html ●
index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <meta
7        http-equiv="Content-Security-Policy"
8        content="script-src 'self' 'unsafe-inline'"
9      />
10     <link rel="stylesheet" href="css/all.min.css" />
11     <link rel="stylesheet" href="css/style.css" />
12     <title>sysCheck</title>
13   </head>
14   <body>
15     <main>
16       <div class="content show">
17         <h1>CPU</h1>
18         <ul style="padding:10px;width: auto;">
19           <li style="padding:10px;"><strong>CPU Användning: </strong><span style="color: white;" id="cpu-usage"></span></li>
20           <li style="padding:10px;"><strong>CPU Fristående: </strong><span style="color: white;" id="cpu-free"></span></li>
21           <li style="padding:10px;"><div style="color: white;" id="cpu-model"></div></span></li>
22         </ul>
23         <h1>Primärminne</h1>
24         <ul style="padding:10px;width: auto;position: relative;bottom:3%;">
25         <li style="padding:10px;"><strong>Minne totalt: </strong><span style="color: white;" id="mem-total"></span></li>
26         <li style="padding:10px;"><strong>Använt minne: </strong><span style="color: white;" id="mem-used"></span></li>
27       </ul>
28       </div>
29     </main>
30   </body>
31 </html>
```

Figur 16. Kod för HTML grunden

För att läsa in information om systemets processor och minne installeras *node-os-utils* NPM-modulen i Figur 17.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm install node-os-utils
```

Figur 17. Installation av node-os-utils

För att visa den önskade informationen i applikationsfönstret används modulen enligt Figur 18.

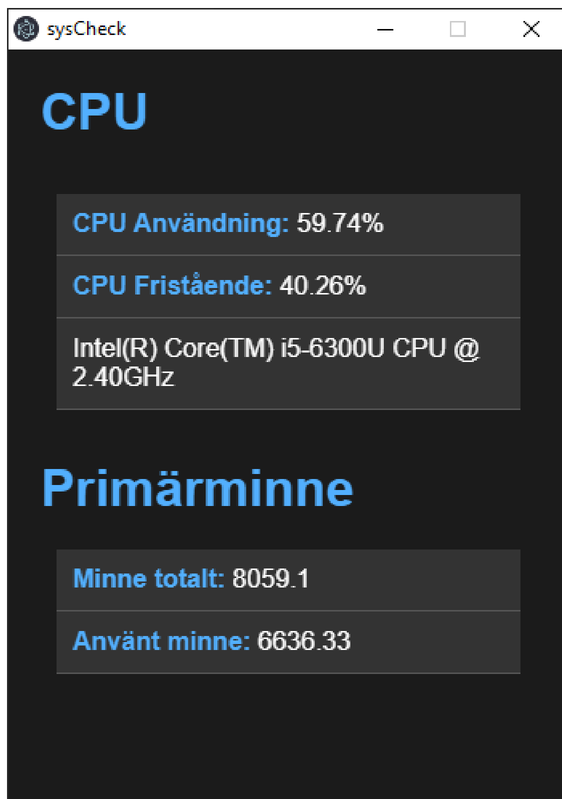
```
JS sysUtils.js ●
JS sysUtils.js > ...
1   const sysUtils = require('node-os-utils')
2   const cpu = sysUtils.cpu
3   const minne = sysUtils.mem
4
5   // Uppdatera varannan sekund
6   setInterval(() => {
7       // CPU Användning
8       cpu.usage().then((info) => {
9         document.getElementById('cpu-usage').innerText = info + '%'
10      })
11
12      // CPU Fristående
13      cpu.free().then((info) => {
14        document.getElementById('cpu-free').innerText = info + '%'
15      })
16
17      // Minne totalt
18      minne.info().then((info) => {
19      document.getElementById('mem-total').innerText = info.totalMemMb
20      })
21
22      // Använt minne
23      minne.info().then((info) => {
24      document.getElementById('mem-used').innerText = info.usedMemMb
25      })
26
27  }, 2000)
28
29  // CPU modell
30  document.getElementById('cpu-model').innerText = cpu.model()
```

Figur 18. Kod för att hämta systeminformation med hjälp av modulen node-os-utils

Nu kan applikationen startas med hjälp av kommandot i Figur 19 för att kolla hur applikationen ser ut vid detta skede (Figur 20) .

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm start
```

Figur 19. npm start kommandot.



Figur 20. Den öppnade Electron applikationen.

3.3.3 Implementering av en nativ meny

Electron-ramverket kommer färdigt men en *Menu*-klass, vilken gör det enkelt att implementera nativa menyer till applikationer. Att menyn är nativ betyder då alltså att menyn automatiskt kommer se annorlunda ut på olika operativsystem, till exempel på macOS placeras menyn i vänstra hörnet av övre balken i stället för inuti fönstret, lika som med andra nativa applikationer på plattformen.

I detta delkapitel implementeras en nativ meny i applikationen med hjälp av Menu-*klassen*.

Lika som med *app* och *BrowserWindow* klasserna, måste *Menu* först hämtas från Electron-modulen enligt Figur 21.

```
JS main.js ×
JS main.js > ...
1   const { app, BrowserWindow, Menu } = require('electron')
2
3   const isMac = process.platform === 'darwin'
```

Figur 21. Menu-klassen hämtas från Electron-modulen.

Eftersom standard-menyn på macOS skiljer sig från Windows och Linux, tilläggs även en variabel i koden som kommer fungera som check om applikationen körs på macOS, för att i det fallet skapa en meny som fungerar enligt macOS-standarder.

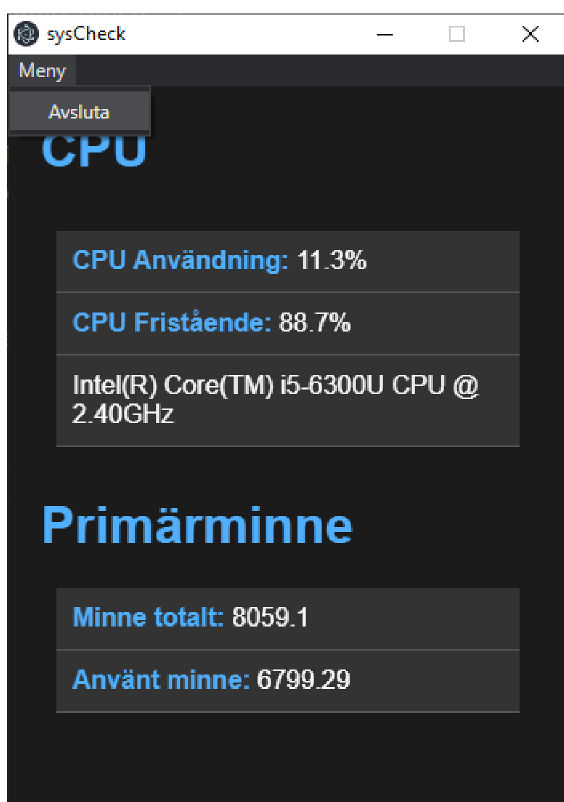
För att skapa en egen meny används *Menu*-klassens *buildFromTemplate*-metod för att bygga menyn enligt en egen mall som vi skapar med koden i Figur 22.

```
app.on('ready', () => {
    createWindow()

    const huvudMeny = Menu.buildFromTemplate(meny)
    Menu.setApplicationMenu(huvudMeny)
})

const meny = [
    ...(isMac ? [{ // Om applikationen öppnas på macOS skapas en klassisk macOS meny
        label: app.name,
        submenu: [
          { role: 'about' },
          { type: 'separator' },
          { role: 'services' },
          { type: 'separator' },
          { role: 'hide' },
          { role: 'hideOthers' },
          { role: 'unhide' },
          { type: 'separator' },
          { role: 'quit' }
        ]
      }] : []),
      {
        label: 'Meny',
        submenu: [
          isMac ? {label:"Avsluta", role: 'close' } : // För macOS
            { // Om inte macOS så innehåller menyn endast följande
              label: 'Avsluta',
              role: 'quit'
            }
        ]
      }
]
```

Figur 22. En ny meny skapas enligt egengjord mall.

Menu-metoden innehåller fördefinierade funktioner för att bland annat avsluta och gömma applikationen. I figuren ovan skapas en meny med ett val för att avsluta applikationen med hjälp av *quit*-rollen för operativsystem som inte är macOS. På macOS används valet enligt nativa macOS-beteendet endast för att gömma applikationen med hjälp av *close*-rollen.

Nu om applikationen startas igen med hjälp av kommandot *npm start* kommer applikationen på Windows se ut som i Figur 23. Genom att klicka på *Avsluta* kommer applikationen sluta köras och fönstret stängs.



Figur 23. Electron-applikation med nativ meny.

3.3.4 Implementering av Tray

Eftersom en applikation som visar systemprestanda troligen inte kommer vara i fokus största delen av tiden kan det vara behändigt för användaren att applikationen göms i *System Tray*, alltså notifikations-fältet, i stället för aktivitetsfältet. Electron-ramverket innehåller klassen *Tray* vilken möjliggör detta.

I det här delkapitlet kommer *Tray* implementeras i applikationen.

Implementeringen börjar som vanligt genom att hämta Tray från Electron-modulen enligt koden i Figur 24.

```
const { app, BrowserWindow, Menu, Tray } = require('electron')
```

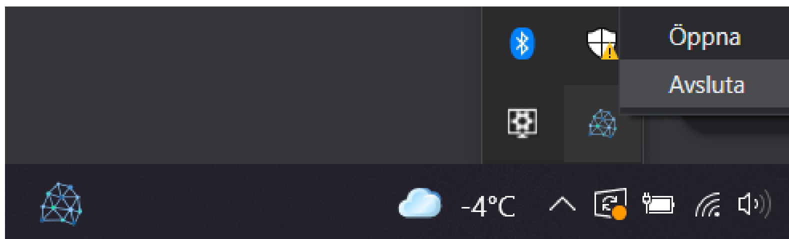
Figur 24. Importering av Tray-klassen.

När klassen har importerats kan Tray-ikonen skapas med en meny enligt koden i figur 25 nedan.

```
tray = new Tray('icon.png') // Skapa nytt Tray object
const contextMenu = Menu.buildFromTemplate([ // Skapa meny för ikonen i System Tray
    { label: 'Öppna',
    click: () => {
        mainWindow.show()
      },
    },
    { label: 'Avsluta',
    click: () => {
        app.isQuitting = true
        app.quit()
      },
    },
])

tray.setToolTip('sysCheck')
tray.setContextMenu(contextMenu)
```

Figur 25. Kod för att skapa en System Tray app-ikon.

Om applikationen startas vid detta skede, hittas applikationen i *System Tray* vilket kan ses i Figur 26 nedan.



Figur 26. Electron-applikation i System Tray.

Som kan ses i figuren ovan, hittas applikationen nu i System Tray, men ikonen syns även i aktivitetsfältet. För att applikationen endast skall synas i System Tray krävs det att man stoppar default-beteendet för minimering av fönster. Eftersom tanken är att applikationen kommer köras i bakgrunden hela tiden kommer även default-beteendet ändras för när användaren klickar på Stäng-knappen, det vill säga krysset i applikationens

övre högra hörn. Båda alternativen kommer med hjälp av koden i Figur 27 orsaka applikationen att gömma sig.

```
mainWindow.on('minimize',function(event){
    event.preventDefault(); // Default beteendet stoppas när användaren minimerar applikationen
    mainWindow.hide(); // Appen blir gömd i stället för att minimeras i aktivitetsfältet
});

mainWindow.on('close', function (event) {
    if(!app.isQuitting){ // Appen avslutas inte även om användaren klickar på krysset eller "Avsluta"-knappen i menyn
        event.preventDefault();
        mainWindow.hide(); // i stället blir den gömd och kör i bakgrunden
    }
    return false;
});
```

Figur 27. Kod för att gömma applikationen när användaren minimerar eller stänger.

Efter ändringen av default-beteende kommer applikationen endast synas i System Tray när den inte är i fokus. För att avsluta applikationen krävs det att användaren öppnar menyn för Tray-ikonen och väljer Avsluta.

3.3.5 Implementering av Globala tangentbordsgenvägar

För att göra det så smidigt som möjligt att granska systemprestandan, kommer detta delkapitel gå igenom hur man med Electron-ramverket kan skapa globala tangentbordsgenvägar för att öppna och gömma applikationen. Att en tangentbordsgenväg är global innebär att den kan användas även om applikationen inte är i fokus, på så sätt behöver användaren aldrig röra på muspekaren för att använda applikationen.

Electron-ramverket innehåller modulen *globalShortcut* vilken gör det väldigt enkelt att registrera nya globala tangentbordsgenvägar. Som vanligt importeras modulen först enligt koden i figur 28 nedan.

```
const { app, BrowserWindow, Menu, Tray, globalShortcut } = require('electron')
```

Figur 28. Importering av globalShortcut-modulen.

Efter att modulen är importerad kan globala tangentbordsgenvägar registreras med hjälp av *register*-metoden. Det enda man måste komma ihåg är att metoden kan kallas först efter att applikationen har laddat klart.

register-metoden från *globalShortcut*-modulen används enligt koden i Figur 29 för att registrera en ny global tangentbordsgenväg.

```
globalShortcut.register('CommandOrControl+T', () => {
    if (mainWindow.isVisible() === true) {
      mainWindow.hide()
    } else {
      mainWindow.show()
    }
  })
```

Figur 29. Kod för att registrera globala tangentbordsgenvägar med hjälp av globalShortcut.

Koden i figuren registrerar CMD+T på macOS och Ctrl+T på Windows och Linux. Nu när användaren trycker den registrerade kombinationen kommer applikationen gömmas om den vid det tillfället är synlig, och hämtas fram om den är gömd, även om användaren just då har en helt annan applikation i fokus.

Till sist när applikationen avslutas är det lönsamt att avregistrera de genvägar som applikationen har skapat. Detta lyckas med hjälp av metoden *unregisterAll* enligt koden nedan i Figur 30.

```
app.on("window-all-closed", () => {
    globalShortcut.unregisterAll()

    if (process.platform !== "darwin") {
      app.quit();
    }
});
```

Figur 30. Kod för att avregistrera globala tangentbordsgenvägar.

3.3.6 Distribution av Electron applikationen

När applikationen är klar för användning är det dags att packa ihop applikationen för distribution. Snabbaste sättet att distribuera Electron-applikationer är med hjälp av Electron Forge.

För att kunna använda Electron Forge måste modulen först installeras via NPM genom kommandot i Figur 31.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm install --save-dev @electron-forge/cli
```

Figur 31. Installation av Electron Forge.

--save-dev parametern betyder att modulen installeras för användning under utvecklingsprocessen.

Efter att Electron Forge är installerat krävs ännu importering av modulens byggställningar med hjälp av NPX, vilket är ett NPM-verktyg för att hantera moduler installerade via NPM. Importering utgörs med hjälp av kommandot i Figur 32.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npx electron-forge import
✓ Checking your system
✓ Writing modified package.json file
✓ Installing dependencies
✓ Writing modified package.json file
✓ Fixing .gitignore


We have ATTEMPTED to convert your app to be in a format that electron-forge understands.
```

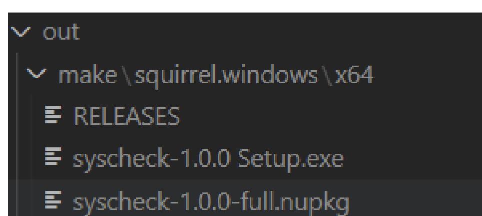
Figur 32. Importering av Electron Forge byggställning.

Vid detta skede är Electron Forge färdigt för användning. För att packa ihop applikationen används kommandot i Figur 33 nedan.

```
PS C:\Users\v0id\Documents\Slutarbete\SysCheck> npm run make

> syscheck@1.0.0 make C:\Users\v0id\Documents\Slutarbete\SysCheck

✓ Checking your system
✓ Resolving Forge Config
We need to package your application before we can make it
✓ Preparing native dependencies
✓ Packaging Application
Making for the following targets: squirrel
✓ Making for target: squirrel - On platform: win32 - For arch: x64
```

Figur 33. Kommandot för att skapa en installationsfil med Electron Forge.

Kommandot packar ihop applikationen och skapar en ny mapp med namnet ”out” som innehåller bland annat en *.exe* fil som nu fungerar för installation av applikationen på Windows-plattformen. Produkten kan ses i Figur 34.



Figur 34. Filer skapade av Electron Forge.

Kommandot för att skapa installationsfiler för macOS och Linux är samma, men med olika parametrar. För macOS tilläggs parametern ”*–platform darwin*” och för Debian-baserad Linux parametrarna ”—platform linux” samt ”—targets deb”.

4 SLUTLEDNING

Målet med detta arbete var att ge läsaren en förståelse för hur Electron-ramverket är uppbyggt och hur det fungerar, ramverkets för-och nackdelar och vad som möjliggör att Electron-applikationer är plattformsoberoende. Målet var även att demonstrera hur utvecklingsprocessen ser ut med ramverket och på så sätt även fungera som en guide för andra som vill skapa skrivbordsapplikationer med Electron.

I den första delen av arbetet gick jag igenom arkitekturen av Electron-ramverket samt jämförde för- och nackdelar med Electron med de mest populära alternativen för utveckling av applikationer. Med tanke på för- och nackdelarna av de olika alternativen, kom jag personligen till följande slutsats:

Eftersom Electron-applikationer utvecklas med hjälp av webbteknologier är det väldigt enkelt samt kostnadseffektivt att erbjuda sin Webbapplikation även som skrivbordsapplikation med hjälp av Electron-ramverket, och vice versa. Utveckling av skrivbordsapplikationer med hjälp av nativa verktyg och programkod kräver betydligt mera resurser eftersom det behövs både utvecklare med ett annat kunnande samt olika kod-baser för alla plattformar.

Nativ utveckling har ändå fortfarande sina fördelar över både Electron och Webbapplikationer. Min personliga åsikt efter detta arbete är att nativ utveckling kan vara lönsamt i bland annat följande situationer:

- Behovet att även erbjuda en webb-version finns inte.
- Applikationen är liten (Eftersom även den simplaste Electron-applikationen kräver minst 100MB av minne).
- Applikationen kräver hög datasäkerhet.
- Prestandan är väldigt viktig, som i bland annat datorspel samt andra grafiskt-krävande applikationer.

I den andra delen av arbetet demonstrerade jag utvecklingsprocessen med Electron-ramverket. Den utvecklade applikationen lyfter enligt mig på ett tydligt sätt fram att

man med hjälp av Electron kan utveckla väldigt nativ-liknande applikationer. Demonstrationen av utvecklingsprocessen kunde möjligen ha förbättrats genom att också under det kapitlet göra jämförelser till nativ utveckling, till exempel genom att vid sidan om visa hur processen hade sett ut med .NET-ramverket.

Sist och slutligen är nativ utveckling naturligtvis alternativet med flest möjligheter och fördelar när man ser på endast en plattform, men faktumet som även kom fram under arbetet är att plattformarna som konsumenterna använder blir hela tiden flera och marknadsandelen mera uppdelad, och i en sådan värld är det väldigt användbart att kunna utveckla plattformsoberoende applikationer.

KÄLLOR

Dryka, M & Pluszczewska, B., 2020, *5 Reasons Why You Should Build Electron Desktop App*. Tillgänglig: https://simovits.com/om-node-js-electron-och-den-nya-uppbyggnaden-av-vara-skrivbordsapplikationer Hämtad: 07.12.2021

van Niekerk, J., 2017, How Do HTML, CSS and JavaScript Work Together? Tillgänglig: https://www.itonlinelearning.com/blog/how-do-html-css-and-javascript-work-together/ Hämtad: 07.12.2021

Benno, C., 2018, *Om Node.js, Electron och den nya uppbyggnaden av våra skrivbordsapplikationer*. Tillgänglig: https://simovits.com/om-node-js-electron-och-den-nya-uppbyggnaden-av-vara-skrivbordsapplikationer Hämtad: 02.06.2020

Biro, J., 2019, Browser Engines… Chromium, V8, Blink? Gecko? WebKit? Tillgänglig: https://medium.com/@jonbiro/browser-engines-chromium-v8-blink-gecko-webkit-98d6b0490968 Hämtad: 03.12.2021

Process Model., 2021, electronjs.org. Tillgänglig: https://www.electronjs.org/docs/latest/tutorial/process-model Hämtad: 04.12.2021

Perkaz, A., 2021, Advanced Electron.js architecture. Tillgänglig: https://blog.logrocket.com/advanced-electron-js-architecture/ Hämtad: 04.12.2021

Khanna, R., 2021, Inter-Process Communication (IPC) in ElectronJS. Tillgänglig: https://www.geeksforgeeks.org/inter-process-communication-ipc-in-electronjs/ Hämtad: 04.12.2021

Desktop App vs Web App: Comparative Analysis, 2020, Digital Skynet. Tillgänglig: https://digitalskynet.com/blog/Desktop-App-vs-Web-App-Comparative-Analysis Hämtad: 07.12.2021

Mondal, A., 2020, Cross-platform Desktop Apps, using Electron. Tillgänglig: https://medium.com/@1806290/cross-platform-desktop-apps-using-electron-ec59d2c93bfb Hämtad: 04.12.2021

Roor, M., 2021, 5 Advantages and Disadvantages of Web Application | Drawbacks & Benefits of Web Application.
Tillgänglig:
https://www.hitechwhizz.com/2021/04/5-advantages-and-disadvantages-drawbacks-benefits-of-web-application.html
Hämtad: 07.12.2021

Chandra, S., 2020, What Is Electron and Why Should We Use it?
Tillgänglig:
https://dzone.com/articles/what-is-electron-amp-why-should-we-use-it
Hämtad: 07.12.2021

Nalegave, S., 2018, Electron | Pros And Cons.
Tillgänglig:
https://medium.com/@nalegaveshardul40/electron-pros-and-cons-8f58fd6313d5
Hämtad: 07.12.2021

2021 Developer Survey., 2021, stackoverflow.com.
Tillgänglig:
https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language
Hämtad: 05.12.2021

Desktop Operating System Market Share Worldwide., 2021, statcounter.com.
Tillgänglig:
https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-200901-202111
Hämtad: 05.12.2021

HTML-CSS-JS The Client-Side Of The Web., 2021, html-css-js.com.
Tillgänglig:
https://html-css-js.com/#about
Hämtad: 05.12.2021

Introduction., 2021, electronjs.org.
Tillgänglig:
https://www.electronjs.org/docs/latest
Hämtad: 09.12.2021