Abstract

Cross-platform development employing Web technologies is gaining traction, with the Tauri Framework emerging as a competitive alternative to ElectronJS. This study delves into the disparities between these two frameworks, scrutinizing key parameters such as performance, installer size, and usability across diverse applications. Notably, ElectronJS often exhibits larger installer sizes in blank applications, although its performance is comparable to Tauri's. However, ElectronJS may lag when loaded with numerous installable packages. In contrast, Tauri optimizes performance by offering a streamlined set of packages in its Rust code repository. The crux of this research reveals a trade-off between developer experience and performance when deciding between the two frameworks, with the potency of user code proving influential. It posits that the choice of tool should hinge on the specific demands of the application, considering factors such as performance, ease of use, and the developer's preference. This study furnishes valuable insights into the differential attributes and performance of ElectronJS and the Tauri Framework under various scenarios. Consequently, it aids developers in making informed decisions on the most apt cross-platform development tool for their specific needs.

Zusammenfassung

Die plattformübergreifende Entwicklung unter Verwendung von Webtechnologien gewinnt zunehmend an Bedeutung, wobei sich das Tauri Framework als konkurrenzfähige Alternative zu ElectronJS abzeichnet. Diese Studie befasst sich mit den Unterschieden zwischen diesen beiden Frameworks und untersucht Schlüsselparameter wie Leistung, Größe des Installationsprogramms und Benutzerfreundlichkeit in verschiedenen Anwendungen. Dabei wird deutlich, dass ElectronJS bei leeren Anwendungen oft größere Installationsprogramme aufweist, obwohl seine Leistung mit der von Tauri vergleichbar ist. Allerdings kann ElectronJS zurückbleiben, wenn es mit zahlreichen installierbaren Paketen geladen wird. Im Gegensatz dazu optimiert Tauri die Leistung, indem es in seinem Rust-Code-Repository eine optimierte Anzahl von Paketen anbietet. Der wesentliche Punkt dieser Untersuchung ist, dass bei der Entscheidung zwischen den beiden Frameworks ein Kompromiss zwischen der Erfahrung der Entwickler und der Leistung besteht, wobei sich die Effizienz des Anwendercodes als einflussreich erweist. Es wird postuliert, dass die Wahl des Tools von den spezifischen Anforderungen der Anwendung abhängen sollte, wobei Faktoren wie Leistung, Benutzerfreundlichkeit und die Vorlieben des Entwicklers berücksichtigt werden. Diese Arbeit liefert wertvolle Einblicke in die unterschiedlichen Eigenschaften und Leistungen von ElectronJS und dem Tauri Framework in verschiedenen Szenarien. Folglich hilft sie Entwicklern, fundierte Entscheidungen über das am besten geeignete plattformübergreifende Entwicklungstool für ihre spezifischen Anforderungen zu treffen.

Table of Contents

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltext-Auszugverzeichnis

Kapitel 1

1 Hintergrundinformationen

Die Entwicklung von Desktop-Anwendungen mit Hilfe von Web-Technologien hat in den letzten Jahren stark zugenommen, angetrieben durch den Bedarf an rationalisierten Entwicklungsprozessen und plattformübergreifender Kompatibilität. Dieser Trend hat zur Entstehung verschiedener Frameworks wie Tauri und ElectronJS geführt, die es Entwicklern ermöglichen, native Desktop-Anwendungen mit vertrauten Web-Entwicklungssprachen wie HTML, CSS und JavaScript zu erstellen. Die Relevanz dieses Themas liegt im Verständnis der Unterschiede zwischen diesen beliebten Frameworks in Bezug auf Leistung, Ressourcenverbrauch und Entwicklererfahrung. Da immer mehr Unternehmen und Organisationen versuchen, benutzerfreundliche Desktop-Anwendungen mit minimalem Ressourcenaufwand und effizienter Bereitstellung auf mehreren Plattformen zu erstellen, wird ein umfassender Vergleich zwischen Tauri und ElectronJS immer wertvoller. Diese Analyse wird Entwicklern nicht nur dabei helfen, fundierte Entscheidungen bei der Auswahl des geeigneten Frameworks für ihre Projekte zu treffen, sondern auch zu einem breiteren Verständnis der aktuellen Landschaft und der zukünftigen Trends bei der Entwicklung von Desktop-Anwendungen mit Webtechnologien beitragen.

1.1 Tauri

Tauri wurde als Reaktion auf den Bedarf an einer leichtgewichtigen und sicheren Alternative zu ElectronJS entwickelt. Das Projekt wurde 2019 von Daniel Thompson-Yvetot und der Tauri-Apps Organisation initiiert. Das Framework zielte darauf ab, die Einschränkungen von ElectronJS, wie z.B. die große Anwendungsgröße und den hohen Ressourcenverbrauch, durch den Einsatz systemeigener Komponenten und einen sicherheitsorientierten Ansatz zu überwinden. (Tauri Contributors, 2023)

Das Open-Source-Framework, basiert auf der für Sicherheit, Geschwindigkeit und Parallelismus bekannten Programmiersprache Rust. Durch die Kombination von Rust und WebView-Komponenten hat es eine einzigartige Position in der Landschaft der plattformübergreifenden Desktop-Anwendungsentwicklung eingenommen. Tauri ermöglicht Entwicklern, Anwendungen mit geringem Speicherbedarf und minimalem Ressourcenverbrauch zu erstellen und dabei bekannte Webtechnologien wie HTML, CSS und JavaScript zu verwenden. Das Ziel des Frameworks ist es, eine sichere und effiziente Umgebung für die Anwendungsentwicklung zu schaffen, wobei eine einfache und optimierte Entwicklererfahrung im Vordergrund steht. Obwohl Tauri ein junges Framework ist, hat es schnell an Beliebtheit gewonnen, insbesondere bei denen, die eine kompakte Alternative zu herkömmlichen Desktop-App-Entwicklungslösungen suchen. (Tauri Contributors, 2023)

Das Framework hat mehrere Iterationen durchlaufen, wobei das Kernteam und die Mitwirkenden der Community ständig an der Verfeinerung und Verbesserung seiner Fähigkeiten arbeiten. Die zunehmende Popularität von Tauri hat es zu einer attraktiven Wahl für Entwickler gemacht, die mit Hilfe von Webtechnologien leistungsfähige und sichere Desktop-Anwendungen erstellen möchten. (Greif, 2023)

Tauri-Anwendungen laufen in einer Webview, einer nativen Komponente, die Webinhalte in einem Desktop-Fenster wiedergibt. Durch die Nutzung der in das Betriebssystem integrierten WebView-Komponente, kann Tauri den Overhead reduzieren, der normalerweise mit der Paketierung und Auslieferung einer vollständigen Browser-Engine verbunden ist, wie es bei ElectronJS der Fall ist. Das Ergebnis sind Anwendungen, die weniger Speicher und CPU-Ressourcen verbrauchen, was zu einer verbesserten Leistung und einem besseren Benutzererlebnis führt.

Das Framework priorisiert Sicherheit, indem es strenge Richtlinien befolgt und einen minimalen Satz von APIs implementiert und in der Standardkonfiguration ausliefert, was die Angriffsfläche für potenzielle Angriffe begrenzt. Darüber hinaus verwenden Tauri-Anwendungen native Kommunikationsbrücken, um Aufgaben auf Systemebene auszuführen, was die Sicherheit der damit erstellten Anwendungen weiter erhöht. (Tauri Contributors, 2023)

1.2 ElectronJS

ElectronJS, oft auch als Electron bezeichnet, ist ein beliebtes Open-Source-Framework, mit dem Entwickler plattformübergreifende Desktop-Anwendungen unter Verwendung von Webtechnologien wie HTML, CSS und JavaScript erstellen können. Electron wurde 2013 von GitHub entwickelt und war ursprünglich für den Atom-Texteditor gedacht, hat sich aber inzwischen zu einer weit verbreiteten Lösung für die Erstellung von Desktop-Anwendungen entwickelt. (OpenJS Foundation, Electron contributors, 2023)

Electron-Anwendungen kombinieren Node.js und Chromium, um eine umfangreiche und effiziente Entwicklungsumgebung zu schaffen. Durch die Einbettung einer Chromium-Browser-Engine in die Anwendung haben die Entwickler Zugriff auf die neuesten Webstandards und -funktionen und können so problemlos komplexe und funktionsreiche Anwendungen erstellen. Mit dieser Kombination kann Electron eine konsistente Entwicklungserfahrung über verschiedene Plattformen hinweg bieten. (OpenJS Foundation, Electron contributors, 2023)

Die weit verbreitete Nutzung und Beliebtheit von Electron beruhten auf der Flexibilität, dem umfangreichen Funktionsumfang und der starken Unterstützung der Community. Das Electron-Ökosystem hat sich erheblich vergrößert und bietet zahlreiche Bibliotheken, Tools und Ressourcen, die Entwicklern bei der Erstellung funktionsreicher und zuverlässiger Anwendungen helfen. Einige der beliebtesten Anwendungen wie Visual Studio Code, Slack und Discord wurden mit Electron entwickelt. Die Vielzahl der Anwendungen unterstreicht die Fähigkeiten und die Vielseitigkeit des Frameworks. (OpenJS Foundation, Electron contributors, 2023)

Im Laufe der Jahre hat sich Electron ständig weiterentwickelt und es werden regelmäßig Updates und Verbesserungen an dem Framework vorgenommen. Das Electron-Team hat zusammen mit der breiteren Open-Source-Gemeinschaft daran gearbeitet, Probleme zu lösen, neue Funktionen hinzuzufügen und die Erfahrung der Entwickler zu verbessern. Obwohl Electron aufgrund seiner Vielseitigkeit und starken Community-Unterstützung weit verbreitet ist, gibt es auch Kritik hinsichtlich des Ressourcenverbrauchs und der Größe der Anwendungen. Diese Kritikpunkte haben zur Entwicklung alternativer Lösungen wie Tauri beigetragen.

1.3 Ziel der Arbeit und Forschungsfragen

Ziel dieses Vergleichs ist es, ein umfassendes Verständnis der Gemeinsamkeiten und Unterschiede zwischen Tauri und ElectronJS als Frameworks für die Entwicklung plattformübergreifender Desktop-Anwendungen unter Verwendung von Webtechnologien zu vermitteln. Durch die Untersuchung der jeweiligen Architekturen, Funktionen, Leistung, Sicherheit und anderer Aspekte soll dieser Vergleich Entwicklern und Entscheidungsträgern helfen, bei der Auswahl des für ihre spezifischen Bedürfnisse am besten geeigneten Frameworks eine fundierte Entscheidung zu treffen. Darüber hinaus bietet diese Bachelorarbeit Einblicke in die Zukunftsaussichten beider Frameworks und zeigt mögliche Trends und Auswirkungen auf die Entwicklungslandschaft für Webanwendungen auf.

Kapitel 2

2 Technische Architektur

2.1 Die Architektur von Tauri

Die Architektur von Tauri konzentriert sich auf Minimalismus und Leistung, wobei systemeigene Komponenten verwendet werden, um den Ressourcen-Overhead zu reduzieren. Tauri-Anwendungen basieren auf Rust und einer WebView, die den Webinhalt in einem Desktop-Fenster wiedergibt. Diese Kombination ermöglicht es Entwicklern, Anwendungen zu erstellen, die Web-Technologien nutzen und gleichzeitig von den Sicherheits-, Geschwindigkeits- und Gleichzeitigkeitsfunktionen von Rust profitieren. (The Tauri Programme within The Commons Conservancy, 2023)

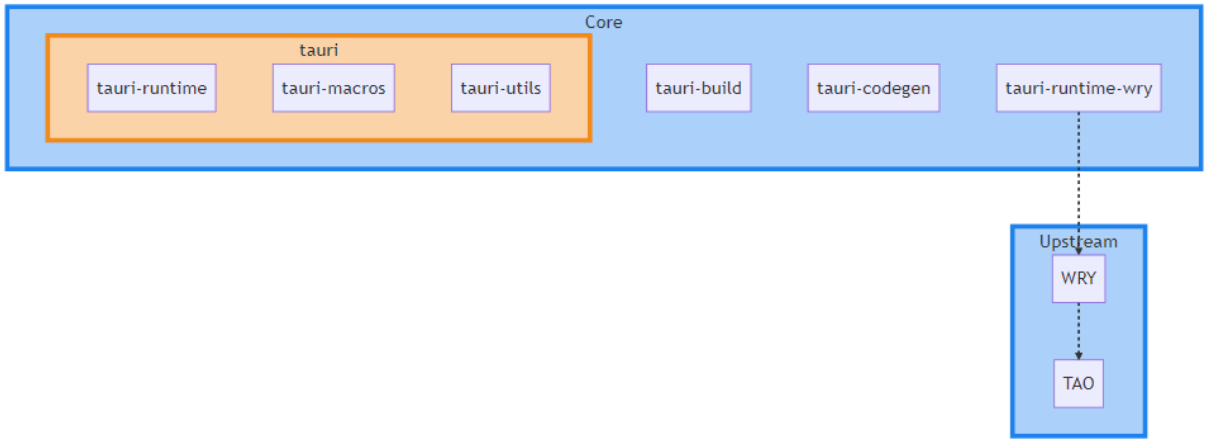

Abbildung 1 Übersicht über die Architektur von Tauri (Quelle: https://tauri.app/v1/references/architecture/)

Die Architektur von Tauri besteht aus einer Reihe von Komponenten, die jeweils eine bestimmte Rolle bei dem Build und Ausführung der Anwendung spielen. Diese Komponenten sind:

- Runtime: Die Tauri Runtime ist die Kernkomponente, die das Anwendungsfenster, die Ereignisse und die Kommunikation zwischen dem Web-Front-End und dem Rust-Back-End einrichtet und verwaltet.
- Macros: Tauri nutzt das Makrosystem von Rust, um bestimmte Aufgaben zu automatisieren, z.B. die Erzeugung von Bindungen zwischen dem Rust-Backend und dem JavaScript-Frontend.
- Utils: Diese Komponente stellt eine Reihe von Utility-Funktionen und -Strukturen bereit, die in der gesamten Tauri-Codebasis verwendet werden. Dazu gehören Funktionen für den Umgang mit Pfaden, URLs und anderen allgemeinen Aufgaben.
- Build: Die Komponente Build von Tauri kümmert sich um den Prozess der Kompilierung und Bündelung der Anwendung. Sie kümmert sich um die Bündelung der Web-Assets, die Kompilierung des Rust-Codes und die Verpackung in eine einzige ausführbare Datei.
- Codegen: Die Komponente Codegen ist für die Generierung des erforderlichen Rust- und JavaScript-Codes zuständig, um die Kommunikation zwischen den beiden Enden zu erleichtern. Sie generiert automatisch die Rust-Funktionsbindungen, die vom JavaScript-Frontend aus aufgerufen werden können, wodurch sich der manuelle Arbeitsaufwand des Entwicklers verringert.
- Runtime-WRY: Die Runtime-WRY ist eine spezielle Laufzeitumgebung, die auf der WebView Rendering-Bibliothek (WRY) aufbaut. Sie bietet die Schnittstelle für die Erstellung und Verwaltung von Anwendungsfenstern, die Handhabung von Systemereignissen und die Erleichterung der Kommunikation zwischen dem Frontend und dem Backend.

Der Tauri-Build-Prozess kompiliert die Webanwendung und das Rust-Backend in eine einzige Binärdatei, was die Bereitstellung und Verteilung vereinfacht. Entwickler können ihre Anwendungen problemlos für verschiedene Plattformen wie Windows, MacOS und Linux verpacken, ohne dass zusätzliche Abhängigkeiten oder Laufzeitumgebungen erforderlich sind.

2.1.1 WRY

WRY (WebView Rendering Library) ist eine plattformübergreifende WebView-Rendering-Bibliothek, die in Rust geschrieben wurde. Sie wurde entwickelt, um alle wichtigen Desktop-Plattformen zu unterstützen, darunter Windows, MacOS und Linux. WRY ermöglicht es Entwicklern, Webinhalte innerhalb nativer Desktop-Anwendungen zu rendern, indem sie die WebView-Komponente des Systems verwenden, d.h. WebKit unter MacOS, WebView2 unter Windows und WebKitGTK unter Linux. (Tauri Programme within The Commons Conservancy, 2021)

Im Zusammenhang mit Tauri spielt WRY eine entscheidende Rolle für die Benutzeroberfläche der Anwendung. Es bietet eine einheitliche Schnittstelle zum WebView des Systems und ermöglicht es Tauri-Anwendungen, HTML- und JavaScript-Inhalte zu rendern. Darüber hinaus ist WRY mit der TAO-Bibliothek für die Erstellung von Fenstern und die Verwaltung von Ereignisschleifen integriert, so dass Tauri-Anwendungen auf verschiedenen Plattformen ein natives Erscheinungsbild erhalten.

WRY ist leichtgewichtig und effizient und damit die ideale Wahl für die Erstellung performanter und ressourcenschonender Desktop-Anwendungen. Durch den Einsatz von WRY können Tauri-Anwendungen im Vergleich zu anderen WebView-Rendering-Bibliotheken oder Lösungen wie ElectronJS kleinere Binärgrößen, schnellere Startzeiten und einen geringeren Speicherverbrauch erzielen.

2.1.2 TAO

Tao ist eine plattformübergreifende Bibliothek zur Erstellung von Anwendungsfenstern, die in Rust geschrieben wurde und alle wichtigen Plattformen unterstützt. (Tauri Programme within The Commons Conservancy, 2021) Sie wurde für das Tauri-Projekt entwickelt und wird von diesem gepflegt. Tao erleichtert die Erstellung und Verwaltung von Anwendungsfenstern und handhabt Ereignisschleifen auf verschiedenen Betriebssystemen, was es zu einer unverzichtbaren Komponente bei der Entwicklung von plattformübergreifenden Anwendungen macht.

Tao ist ein Fork der Winit-Bibliothek, die die Linux-Portierung zu Gtk ersetzt hat. Diese Änderung war nicht nur wegen webkit2gtk notwendig, sondern auch wegen verschiedener Funktionen der Desktop-Umgebung wie Menüleisten, Systemtrays und globale Verknüpfungen. Für die Zukunft plant das Tauri-Team, diese Funktionen als separate Crates modularer zu gestalten, so dass das Projekt wieder zu Winit wechseln und die gesamte Rust-Entwicklergemeinschaft davon profitieren kann. (Tauri Programme within The Commons Conservancy, 2023)

Durch die Bereitstellung einer konsistenten und unkomplizierten Schnittstelle für die Erstellung von Fenstern und die Verwaltung von Ereignisschleifen spielt Tao eine wichtige Rolle in Tauri-Anwendungen, vereinfacht den Entwicklungsprozess und ermöglicht es den Entwicklern, sich auf die Kernfunktionalität ihrer Anwendung zu konzentrieren. (Tauri Contributors, 2023)

2.2 Die Architektur von ElectronJS

Electron ist ein Framework welches die Erstellung von Desktop-Anwendungen mit JavaScript, HTML und CSS ermöglicht. Durch die Einbettung von Chromium und Node.js in seine Binärdatei ermöglicht Electron mit der Verwaltung einer einzigen JavaScript-Codebasis die Erstellung von plattformübergreifenden Anwendungen, die unter Windows, MacOS und Linux funktionieren. Ohne

dass Erfahrung in der nativen Entwicklung beim Entwickler vorhanden sein muss. (OpenJS Foundation, Electron contributors, 2021)

Um eine leistungsstarke und flexible Entwicklungsumgebung für die Erstellung plattformübergreifender Desktop-Anwendungen zu schaffen. Electron-Anwendungen bestehen aus zwei Hauptkomponenten: dem Hauptprozess, auf dem eine Node.js-Instanz läuft und der die Applikation sowie die Fenster kontrolliert, und einem oder mehr Renderer-Prozessen, je nach Anzahl der Fenster, auf dem eine Chromium-Instanz zur Anzeige von Webinhalten läuft. (OpenJS Foundation, Electron contributors, 2021)

In der Architektur von Electron ist der Hauptprozess für die Verwaltung des Lebenszyklus der Anwendung, die Handhabung nativer Betriebssysteminteraktionen und die Kommunikation mit Renderer-Prozessen zuständig. Ein Renderer-Prozess hingegen ist für das Rendern der Webinhalte und die Ausführung von JavaScript zuständig. Die Kommunikation zwischen dem Haupt- und dem Renderer-Prozess erfolgt über IPC-Mechanismen (Inter-Process Communication), die einen effizienten und sicheren Datenaustausch ermöglichen. (OpenJS Foundation, Electron contributors, 2021)

Diese Architektur wurde von Chromium übernommen, hier läuft ebenfalls ein Renderer-Prozess für jeden Tab. Dies hat zum Beispiel den Vorteil das nicht die gesamte Applikation abstürzt, falls ein fataler Fehler in einem Tab auftritt. Durch diese Kapselung hat nur der Hauptprozess Zugriff auf privilegierte Electron-Funktionen, jedoch kann dieser nicht auf direkt auf die DOM-APIs des Fensters zugreifen. (Nokes, 2016)

2.2.1 Vergleich der zugrunde liegenden Technologien: WebView2 zu Chromium

Im Zusammenhang mit WebView2-Anwendungen dient die WebView2 Runtime als grundlegende Webplattform und funktioniert ähnlich wie die Visual C++ oder .NET Runtime für C++ bzw. .NET-Anwendungen. Diese Runtime ist weiterverteilbar und besteht aus angepassten Microsoft Edge-Binärdateien, die für die Kompatibilität mit WebView2-Anwendungen optimiert und gründlich getestet wurden. Besonders wichtig ist, dass die WebView2 Runtime, sobald sie auf einem Gerät installiert ist, nicht als Browseranwendung zu erkennen ist. Folglich werden die Benutzer keine Browser-Desktop-Verknüpfungen oder Einträge im Startmenü finden, die mit der Runtime verbunden sind. Für die Verteilung und Aktualisierung der WebView2 Runtime auf Client-Rechnern stehen zwei verschiedene Methoden zur Verfügung: der Evergreen-Verteilungsmodus und der Verteilungsmodus mit fester Version. (MSEdgeTeam, mikehoffms, nishitha-burman, vchapel, champnic,liminzhu,zoherghadyali,jasonstephen15,captainbrosset,jm-trd-ms,Reezaali,pagoe-msft,peiche-jessica,JoshuaWeber, 2023)

ElectronJS bündelt eine spezifische Version von Chromium direkt in seine Binärdatei. Dies bedeutet, dass jede Electron-Anwendung ihre eigene Chromium-Instanz enthält. Während dies den Vorteil hat, dass Entwickler genau wissen, welche Version von Chromium ihre Anwendung verwendet und daher eine konsistente Laufzeitumgebung haben, führt es auch zu Nachteilen. Erstens erhöht dies die Größe der Electron-Anwendung erheblich, da jede Anwendung ihre eigene vollständige Kopie von Chromium enthält. Zweitens bedeutet dies, dass Entwickler ihre Anwendung aktualisieren müssen, wenn eine neue Version von Chromium veröffentlicht wird, insbesondere wenn diese neue Version wichtige Sicherheitsupdates enthält.

2.3 Prozess Modelle

Die Prozess Modelle zwischen Electron und WebView2 unterscheiden sich dadurch, dass der bei der WebView2 einen weiter Hostprozess über dem Rederingprozess benötigt wird.

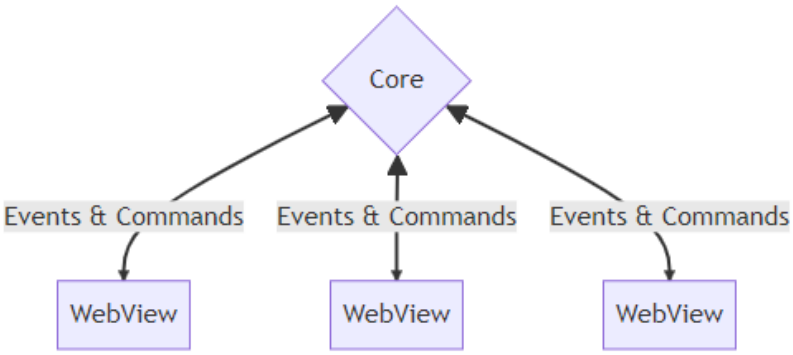

Abbildung 2 Vereinfachte Darstellung des Tauri-Prozessmodells. Ein einzelner Core-Prozess verwaltet einen oder mehrere WebView-Prozesse. (Quelle: https://tauri.app/v1/references/architecture/process-model)

Im Falle von Electron ist Node.js mitintegriert, so dass Electron-Anwendungen jede Node.js-API, jedes Modul oder jedes Node-Native-Addons sowohl Renderprozess als auch vom Hauptprozess diese nutzen können. Dagegen machen WebView2-Anwendungen keine Annahmen über die Sprache oder das Framework, die im Rest der Anwendung verwendet werden. Um auf Betriebssystemfunktionen zugreifen zu können, muss der JavaScript-Code in einer WebView2-Anwendung über den Anwendungs-Host-Prozess kommunizieren, der als Proxy. (OpenJS Foundation and Electron contributors, 2021)

2.3.1 Weitere Unterschiede

	Electron	**WebView2**
Build dependencies	Chromium	Edge
Offen Zugänglicher Quellcode	Ja	Nein
Teilt DLLs mit Edge/Chrome	Nein	ja (seit Edge 90)
Geteilte Laufzeitumgebung zwischen Anwendungen	Nein	Optional
Programm APIs	Ja	Nein
Node.js	Ja	Nein
Sandbox	Optional	Immer
Benötigt ein Programm Framework	Nein	Ja
Unterstütze Plattformen	Mac, Windows, Linux	Windows (Mac/Linux geplant)
Geteilte Prozesse zwischen Apps	Nie	Optional
Framework Aktualisierungen verwalten von	Applikation	WebView2

Tabelle 1 Vergleich ElectronJS zu WebView Anwendungen (Quelle: https://www.electronjs.org/blog/webview2#architecture-overview)

Kapitel 3

3 Merkmale und Fähigkeiten

3.1 Kernfunktionen von Tauri

Tauri bietet eine Reihe von Kernaspekte, die zu seiner Attraktivität als Framework für die Entwicklung von Desktop-Anwendungen beitragen. Einige dieser Funktionen sind:

- Integration bestehender Projekte. Durch den einfachen Aufbau kann jede auf HTML, CSS und JavaScript basierende Webseite oder Anwendung in Tauri integriert werden. So ist man Framework unabhängig und muss bestehende Anwendungen nicht migrieren. Es gibt bereits verschiede Anleitungen um die Integration je nach Framework weiter zu erleichtern, sodass Tooling wie gewohnt funktioniert. (Tauri Contributors, 2023)
- Geringer Ressourcenverbrauch: Durch die Nutzung der systemeigenen WebView-Komponente. (Tauri, 2022)
- Kleine Anwendungsgröße: Die Architektur von Tauri minimiert den Platzbedarf der Anwendung, indem sie unnötige Abhängigkeiten ausschließt und die Webanwendung und das Rust-Backend in einer einzigen Binärdatei bündelt. (Tauri, 2022)
- Plattformübergreifende Kompatibilität: Mit Tauri können Entwickler mühelos Anwendungen für verschiedene Plattformen erstellen, darunter Windows, MacOS und Linux, was die Bereitstellung und Verteilung vereinfachten. (Tauri, 2022)
- Erfahrung für Entwickler: Tauri bietet eine intuitive Befehlszeilenschnittstelle (CLI) und flexible Konfigurationsoptionen, die Entwicklern den Einstieg erleichtern und die Anpassung ihrer Anwendungen ermöglichen. (Tauri, 2022)

3.2 Roadmap und Zukunftspläne für Tauri

Das Entwicklungsteam von Tauri hat mehrere Pläne und Ziele für die Zukunft des Frameworks. Zu den wichtigsten Zielen gehören:

- Mobile Bundler: Bündelung auf alle gängigen Betriebssysteme für mobile Geräte
- Cross Compiler: Generieren von gebündelten Binärdateien aus ausgewählten Betriebssystemumgebungen
- Other Bindings: Go, Nim, Python, C++ und andere Anbindungen sind mit der stabilen API möglich
- One-Time Commandos: Einen Befehl ausführen, der nach der ersten Ausführung nicht mehr verfügbar ist
- Alternative Renderer: Vorstellung von alternativen Renderern
- Channel API: Nachrichten über einen Kanal senden

3.3 Kernfunktionen von ElectronJS

ElectronJS bietet eine breite Palette von Funktionen und Möglichkeiten, die es zu einer beliebten Wahl für die Erstellung von plattformübergreifenden Desktop-Anwendungen unter Verwendung von Webtechnologien gemacht haben. Einige dieser Funktionen sind:

- Reichhaltige Entwicklungsumgebung: Electron kombiniert Chromium und Node.js und bietet Entwicklern Zugang zu den neuesten Webstandards und -funktionen, so dass sie komplexe und funktionsreiche Anwendungen erstellen können. (OpenJS Foundation and Electron contributors, 2021)
- Umfangreiches Ökosystem: Electron verfügt über ein umfangreiches Ökosystem von Bibliotheken, Tools und Ressourcen, das es Entwicklern erleichtert, Lösungen für bestimmte Probleme zu finden und ihre Anwendungen zu verbessern.

- Starke Unterstützung durch die Gemeinschaft: Electron verfügt über eine große und aktive Community, die Entwicklern zahlreiche Ressourcen zur Verfügung stellt, darunter Dokumentationen, Tutorials und Foren, die sie bei der Entwicklung ihrer Anwendungen unterstützen. (Sorhus, 2023)
- Plattformübergreifende Kompatibilität: Wie Tauri unterstützt Electron die Entwicklung von Anwendungen für mehrere Plattformen, darunter Windows, MacOS und Linux, und vereinfacht so den Prozess der Entwicklung, Bereitstellung und Verteilung
- Interprozesskommunikation (IPC): Electron bietet effiziente und sichere IPC-Mechanismen für die Kommunikation zwischen dem Haupt- und dem Renderer-Prozess, die es Entwicklern ermöglichen, den Anwendungsstatus zu verwalten und Aufgaben auf Systemebene auszuführen. (OpenJS Foundation and Electron contributors, 2021)

3.4 Roadmap und Zukunftspläne für ElectronJS

ElectronJS verfügt über eine aktive Entwicklergemeinschaft und wird bereits seit 2013 entwickelt die letzten Releases konzentrierten sich auf Stabilität und Komfortfeatures. Zukünftig will man sich dem Chrome-Release Zyklus angleichen. (Zhao, 2023)

Verbesserte Leistung: ElectronJS zielt darauf ab, seine Leistung kontinuierlich zu optimieren, den Ressourcenverbrauch zu minimieren und es für eine breitere Palette von Anwendungen effizienter zu machen. (Anon., 2023)

Verbesserte Sicherheit: ElectronJS ist bestrebt, Entwicklern die Tools und Best Practices an die Hand zu geben, die für die Erstellung sicherer Anwendungen erforderlich sind, und arbeitet kontinuierlich an der Verbesserung der Sicherheitsfunktionen des Frameworks. (Xu, 2023)

Bessere Erfahrung für Entwickler: Das ElectronJS-Team arbeitet kontinuierlich daran, die Erfahrung der Entwickler zu verbessern, indem es bessere Werkzeuge, Dokumentationen und Community-Ressourcen bereitstellt, um die Erstellung und Wartung von Electron-Anwendungen zu erleichtern.

3.5 Direkter Vergleich der Funktionen

Funktion	Tauri	Electron
Integration bestehender Projekte	Gut	Gut
Plattformübergreifende Kompatibilität	Gut	Sehr gut
Erfahrung für Entwickler	Gut	Gut
Mobile Bundler	Geplant	Nein
Cross Compiler	Geplant	Teilweise
One-Time Commandos	Geplant	Möglich
Language Binding	Geplant	Eingeschränkt möglich
Alternative Renderer	Geplant	Nein
Channel API	Geplant	Ja
Ökosystem	Gut	Sehr gut
Community	Gut	Gut
Interprozesskommunikation	Gut	Sehr gut

Tabelle 2 Vergleich von Funktion zwischen Tauri und Electron

Beide Frameworks sind plattformübergreifend kompatibel und ermöglichen es Entwicklern, Anwendungen für Windows, MacOS und Linux zu erstellen. Allerdings bietet Tauri durch die Verwendung von nativen Komponenten und Rust eine schlankere Lösung, während Electron durch

die Integration von Chromium und Node.js eine reichhaltigere Entwicklungsumgebung bietet. In der der 2.0.0 Alphaversion von Tauri gibt es eine Unterstützung für Native Ios und Android Applikationen.

Entwickler sollten die spezifischen Anforderungen und Prioritäten ihrer Projekte berücksichtigen, wenn sie die Funktionen und Möglichkeiten von Tauri und ElectronJS vergleichen. Für Projekte, bei denen Leistung, Ressourceneffizienz und Sicherheit im Vordergrund stehen, ist Tauri möglicherweise die bessere Wahl. Für Anwendungen, die eine reichhaltige Entwicklungsumgebung, einen umfangreichen Funktionsumfang und ein großes Ökosystem von Bibliotheken und Ressourcen benötigen, könnte ElectronJS dagegen die bessere Wahl sein.

Es ist auch erwähnenswert, dass die Entwicklungserfahrung und die Unterstützung der Community eine entscheidende Rolle bei der Auswahl spielen. Während ElectronJS über eine etabliertere Community und eine Fülle von Ressourcen verfügt, wächst die Community von Tauri.

Für Tauri und Electron können Fragen im Offiziellen Discord Server und auf Stack Overflow gestellt werden.. Die Fragen werden dort sehr schnell beantwortet. Vor allem am Discord ist die Community aktiv. Bei Electron sind es Stand 27.04.2023 14500 Mitglieder und bei Tauri 10000.

Kapitel 4

4 Sicherheit

4.1 Tauri

Vor dem 1.0 Release hat sich das Team hinter Tauri entschlossen einen Feature Freeze zu machen. Und die Codebase einem externen Audit unterzogen. Dabei sind Zahlreiche Fehler und Schwachstellen bekannt geworden. Diese wurden vor dem offiziellen Release 1.0 behoben. Unter den Fehlern waren 0 als extrem und 8 als hoch nach dem PTES (Penetration Testing Execution Standard) eingestuft. (Thompson-Yvetot, 2022)

Aus dem Abschlussbericht geht hervor, dass das Tauri-Projekt eine vielversprechende Alternative zu beliebten Frameworks wie Electron oder React Native darstellt, und zwar aufgrund seines Ansatzes bei der Entwicklung, der Konfiguration, der Benutzerfreundlichkeit und dem Fokus auf einen geringen Ressourcenverbrauch. Im Gegensatz zu diesen bestehenden Projekten vermeidet Tauri grundlegende Sicherheitsprobleme, indem es sich auf die Webansicht des Systems verlässt, anstatt die Rendering Engine einzubetten. Diese Unterscheidung ermöglicht eine schnellere Übernahme von Upstream-Sicherheitskorrekturen und verringert das Risiko von schwerwiegenden Angriffen. (Daniel Attevelt, 2022)

Tauri Anwendungen benötigt keinen Server um lokale Ressourcen zu Laden, deshalb sind Angriff Szenarien wie Networksniffing nicht möglich. (Tauri Contributors, 2023)

Dadurch das der Core Layer von Tauri in Rust geschrieben ist, können Entwickler verschiedene Arten von Angriffen abwehren, darunter Angriffe auf den Speicher (z. B. Pufferüberläufe, Use-after-free), Angriffe auf Typverwechslungen, Data Races und Angriffe, die undefiniertes Verhalten oder Schwachstellen im Zusammenhang mit Parallelität ausnutzen. Die Sicherheitsfunktionen und bewährten Praktiken, die von der Sprache Rust durchgesetzt werden, tragen dazu bei, sicherere und zuverlässigere Software zu erstellen und die Wahrscheinlichkeit von Schwachstellen zu verringern, die von Angreifern ausgenutzt werden können. (Fernandez, 2019)

Tauri legt den Schwerpunkt auf Sicherheit mit seinem minimalen Satz an APIs und nativen Kommunikationsbrücken, was die Angriffsfläche reduziert und die Sicherheit der mit dem Framework erstellten Anwendungen insgesamt erhöht. (Tauri, 2022)

4.2 ElectronJS

Electron, ein Tool, das die Entwicklung von Desktop-Anwendungen auf der Grundlage von Webtechnologien ermöglicht, unterscheidet sich in seinen Sicherheitsüberlegungen von herkömmlichen Webbrowsern. Seine weitreichenden Fähigkeiten, einschließlich des Zugriffs auf das Dateisystem und die Benutzer-Shell, erhöhen potenziell die Sicherheitsrisiken proportional zum Umfang der Befugnisse des Codes. Electron ist nicht für den sicheren Umgang mit beliebigen Inhalten aus nicht vertrauenswürdigen Quellen konzipiert ist. Anwendungen wie Atom, Slack und Visual Studio Code, die auf Electron aufbauen, zeigen überwiegend lokale Inhalte oder vertrauenswürdige, entfernte Inhalte an, die nicht in Node integriert sind. Die Verantwortung für die Sicherheit von Code, der aus dem Internet stammt und in einer Anwendung ausgeführt wird, liegt beim Entwickler. (OpenJS Foundation and Electron contributors, 2023)

Die Sicherheit einer Electron-Anwendung ist mehrdimensional und ruht auf den Schultern der Framework-Grundlage (Chromium, Node.js), Electron selbst, allen NPM-Abhängigkeiten und dem Code des Entwicklers. Daher ist es ratsam, die Anwendungen mit der neuesten Version des Electron-

Frameworks auf dem neuesten Stand zu halten, vertrauenswürdige Bibliotheken von Drittanbietern sorgfältig auszuwählen und sichere Codepatterns anzuwenden. Das Empfangen und lokale Ausführen von Code aus einer nicht vertrauenswürdigen Quelle stellt ein Sicherheitsrisiko dar. Daher wird vom Laden und Ausführen von Remote-Code bei aktivierter Node.js-Integration dringend abgeraten. Stattdessen sollten lokale Dateien, die mit der Anwendung verpackt sind, die Quelle für die Ausführung von Node.js-Code sein. Für die Anzeige von Remote-Inhalten sollte das <webview>-Tag oder BrowserView verwendet werden, wobei die nodeIntegration deaktiviert und die contextIsolation aktiviert ist. Electron erweitert die Unterstützung durch Sicherheitswarnungen und Empfehlungen, die auf der Entwicklerkonsole angezeigt werden und je nach Bedarf manuell aktiviert oder deaktiviert werden können. In den allgemeinen Sicherheitsempfehlungen gibt es eine Checkliste, die häufige Sicherheit relevante Fehlkonfigurationen abdeckt. (OpenJS Foundation and Electron contributors, 2023)

Electron hat seit dem Release der Version mit der Nummer 12 die Kontextisolation für die Renderprozesse und das Preloadscript in der Standardkonfiguration aktiviert. Diese Einstellung soll verhindern das Webseiten auf die interne Electronapi Zugriff bekommen. Natürlich ist es möglich Funktionen aus der Internenapi für den Renderer Prozess bereitzustellen, dies sollte aber nur für einzelne Funktionen passieren. Das und die Bereinigung der Aufrufe von Unerlaubten Befehlen liegt aber in der Verantwortung des Entwicklers. (OpenJS Foundation and Electron contributors., 2023)

4.3 Vergleich

Beide Frameworks informieren die Entwickler über den richtigen und vernünftigen Umgang ihrer Funktionen. Da aber nicht jeder Entwickler die Dokumentation zu den Tools vollständig liest haben sich die Framework Entwickler entschieden weitere Maßnahmen zu treffen. Bei Elektron werden in der Entwickerkonsole sicherheitsrelevante Warnungen und Fehler angezeigt. Bei Tauri ist die bereitgestellte API restriktiver, hier müssen zum Beispiel Zugriffe auf das Dateisystem die Außerhalb des Anwendungskontextes liegen selbst in Rust implementiert werden und können nicht über die bereitgestellte JavaScript Schnittstelle abgesetzt werden.

5 Leistungsvergleich:

Die Zeit, die für das Einrichten der Entwicklungsumgebung benötigt wird, ist ein entscheidender Faktor im Entwicklungsprozess. Eine schnelle und einfache Einrichtung kann wertvolle Zeit und Ressourcen sparen, die besser in die eigentliche Entwicklung investiert werden können. Eine langwierige und komplizierte Einrichtung kann für neue Entwickler entmutigend sein und zu einem langsameren Projektstart führen. Aus diesem Grund haben sich viele Entwicklungswerkzeuge und Frameworks, darunter Electron und Tauri, bemüht, den Einrichtungsprozess zu rationalisieren und den Entwicklern eine detaillierte Dokumentation zur Verfügung zu stellen. Ein einfaches und unkompliziertes Setup kann eine positive Erfahrung für Entwickler schaffen und zu einer höheren Produktivität und besseren Entwicklungsergebnissen führen. (Bulajic, 2013)

Aufgabe (Betriebssystem)	**Tauri**	**ElectronJS**
Einrichtung (Windows)	8sec +4sec	30sec
Erster Start von Entwicklung-Build (Windows)	37sec	4sec
Zweiter Start (Windows)	6sec	4sec
Initialisierung bis gestartetes Hello World (Windows)	~60sec	~40sec

Tabelle 3 Vergleich Einrichtung zwischen Tauri und Electron

5.1 Electron Einrichtung

Die Geschwindigkeit des Einrichtungsvorgangs wurde mit dem PowerShell Measure-Command gemessen, weil es genau und praktisch ist. Measure-Command ist ein integriertes PowerShell-Cmdlet, welches die Zeit für die Ausführung von Skriptblöcken oder Cmdlets genau misst und die Ausführungszeit in Millisekunden zurückgibt. Dies gewährleistet eine detaillierte und genaue Messung der Dauer des Einrichtungsprozesses, die für die Bewertung von Leistung und Effizienz entscheidend ist. Durch die Verwendung von Measure-Command erhalten wir zuverlässige und konsistente Messungen und können so genaue Vergleiche anstellen und fundierte Entscheidungen über die Geschwindigkeit des Setup-Prozesses treffen.

```
PS D:\Uni\BA_Tauri_ElectronJS> Measure-Command { npm init electron-app@latest file-access }

TotalMinutes      : 0,500137268333333
TotalSeconds      : 30,0082361
TotalMilliseconds : 30008,2361
```

Abbildung 3 Initialisierung eine Electron Projektes mit npm gemessen durch das Measure-Command Cmdlet

Für das Installieren von Electron sowie Tauri unter Windows wird npm benötigt. Für das Einrichten von Electron wurde Electron Forge verwendet. Dies hat gegenüber dem manuellem einrichten den Vorteil, dass der Prozess automatisch passiert und man ein funktionierendes “Hello World” bekommt.

```
PS D:\Uni\BA_Tauri_ElectronJS\file-access> Measure-Command { npm run start }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 4
Milliseconds      : 63
Ticks             : 40638173
TotalDays         : 4,70349224537037E-05
TotalHours        : 0,00112883813888889
TotalMinutes      : 0,0677302883333333
TotalSeconds      : 4,0638173
TotalMilliseconds : 4063,8173
```

Abbildung 4 Starten der Electron Anwendung gemessen durch das Measure-Command Cmdlet

Durch das Verwenden des Electron Forge Skriptes werden auch einige Skripts in das package.json File geschrieben. So ist es dann möglich diese komfortable mit „npm run“ aufzurufen. Das „start“ Skript startet dabei die Anwendung im Entwickler Modus. Diese Übernimmt Änderungen des Sourcecodes live.

5.2 Tauri Einrichtung

Tauri kann zwar auch mit anderen Tools wie Bash, PowerShell und Cargo installiert werden, um den Vergleich leichter zu gestalten wurde npm verwendet. Das Commando läuft zwar sehr schnell ab, dabei fehlt allerdings der Schritt der Installation der npm Pakete. Hier wäre es wünschenswert, wenn dieser Schritt mit ausgeführt werden würde, um die nötigen Nutzerinteraktionen zu minimieren.

```
PS D:\Uni\BA_Tauri_ElectronJS> Measure-Command { npm create tauri-app@latest my-tauri-app --template vanilla --manager npm --yes   }

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 7
Milliseconds      : 405
Ticks             : 74059022
TotalDays         : 8,57164606481481E-05
TotalHours        : 0,00205719505555556
TotalMinutes      : 0,123431703333333
TotalSeconds      : 7,4059022
TotalMilliseconds : 7405,9022
```

Abbildung 5 Initialisierung eines Tauri Projektes mit npm gemessen durch das Measure-Command Cmdlet

Es folgt eine Ausgabe, in der die nächsten Schritte aufgeführt werden, das ist für Entwickler sehr komfortable und kann beim Prozess Fehler vermieden.

```
Template created! To get started run:
  cd tauri-file-access
  npm install
  npm run tauri dev
```

Abbildung 6 Ausgabe bei der Erstellung einer Tauri Anwendung

Das Installieren der Abhängigkeiten war zwar mit 5 Sekunden schnell, aber beinhaltete einen extra Schritt im Vergleich zu der Electron Einrichtung. Diese ist mit 30 Sekunden viel langsamer bei der Ausführung, schlussendlich, aber schneller den Benutzer, da dieser den extra Befehl nicht absetzten muss.

```
PS D:\Uni\BA_Tauri_ElectronJS\tauri-file-access> Measure-Command { npm i }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 4
Milliseconds      : 470
Ticks             : 44701157
TotalDays         : 5,17374502314815E-05
TotalHours        : 0,00124169880555556
TotalMinutes      : 0,0745019283333333
TotalSeconds      : 4,4701157
TotalMilliseconds : 4470,1157
```

Abbildung 7 Installieren der Abhängigkeiten für Tauri gemessen durch das Measure-Command Cmdlet

Die Tauri-CLI punktet beim Thema Benutzerfreundlichkeit da der Prozess interaktiv gestaltet ist und hier mittels Auswahl in der Shell mein Projekt konfigurieren kann. Bei Electron müssen die Flags und die möglichen Inputs erst aus der Dokumentation gesucht werden. So könnte ein Aufruf mit einem UI-Template aussehen: npm init electron-app@latest my-new-app -- --template=vite. Objektiv betrachtet ist die Variante von Tauri besser.

```
PS D:\Uni\BA_Tauri_ElectronJS> npm create tauri-app@latest
Need to install the following packages:
  create-tauri-app@3.4.0
Ok to proceed? (y) y
✓ Project name · tauri-cpu
✓ Choose which language to use for your frontend · TypeScript / JavaScript - (pnpm, yarn, npm)
✓ Choose your package manager · npm
✓ Choose your UI template · Vanilla
✓ Choose your UI flavor · JavaScript

Template created! To get started run:
  cd tauri-cpu
  npm install
  npm run tauri dev
```

Abbildung 8 Interaktionen mit dem Tool von Tauri

Tauri verwendet für das Rustbackend natürlich auch Bibliotheken. Diese werden beim ersten Start des Developmentservers installiert dies dauerte ungefähr 2 Minuten. Leider werden diese Pakete nicht in einem Cache abgelegt so wie es bei npm der Fall ist. Sondern müssen für jedes Projekt runtergeladen und kompiliert werden. Das ist eine Eigenheit von Crates ist dennoch für den Vergleich relevant.

```
PS D:\Uni\BA_Tauri_ElectronJS\tauri-file-access> Measure-Command { npm run tauri dev }
        Info Watching D:\Uni\BA_Tauri_ElectronJS\tauri-file-access\src-tauri for changes...
    Updating crates.io index
  Downloaded serde_derive v1.0.162
  Downloaded serde v1.0.162
  Downloaded 2 crates (132.1 KB) in 1.96s
   Compiling proc-macro2 v1.0.56
   Compiling unicode-ident v1.0.8
```

Abbildung 9 Ausgabe beim ersten Start von Tauri gemessen durch das Measure-Command Cmdlet

Zu erwähnen ist das dieser Prozess zwar lange dauert dennoch immer verlässlich funktioniert hat.

```
Days              : 0
Hours             : 0
Minutes           : 1
Seconds           : 37
Milliseconds      : 673
Ticks             : 976737288
TotalDays         : 0,00113048297222222
TotalHours        : 0,0271315913333333
TotalMinutes      : 1,62789548
TotalSeconds      : 97,6737288
TotalMilliseconds : 97673,7288
```

Abbildung 10 Ergebnisse der Messung des ersten Starts von Tauri

Wie bereits erwähnt ist der Prozess bei den weiteren Starts des Entwicklungsservers schneller. Bei einem Produktiv Build müssen die Pakete dann gesondert installiert werden, hier ist ein ähnliches Verhalten zu beobachten.

```
PS D:\Uni\BA_Tauri_ElectronJS\tauri-file-access> Measure-Command { npm run tauri dev }
        Info Watching D:\Uni\BA_Tauri_ElectronJS\tauri-file-access\src-tauri for changes...
   Compiling tauri-file-access v0.0.0 (D:\Uni\BA_Tauri_ElectronJS\tauri-file-access\src-tauri)
    Finished dev [unoptimized + debuginfo] target(s) in 2.62s


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 6
Milliseconds      : 89
Ticks             : 60896008
TotalDays         : 7,04814907407407E-05
TotalHours        : 0,00169155577777778
TotalMinutes      : 0,101493346666667
TotalSeconds      : 6,0896008
TotalMilliseconds : 6089,6008
```

Abbildung 11 Ergebnisse des zweiten Starts von Tauri

5.3 Inhalt der Projekte

Der Benchmarking-Prozess besteht aus drei Tests: CPU-Auslastung, Dateizugriff und Inter-Prozess-Kommunikation (IPC). Der Code ist auf Github[1] verfügbar. Die Electron Anwendungen sind in den Ordnern „cpu“, „file-access“ und „ipc“ zu finden. Die Tauri Anwendungen haben zusätzlich das Präfix „tauri-„.

Der Prozessorleistungsvergleich misst die Zeit, die benötigt wird, um alle Primzahlen unter einer bestimmten Grenze zu berechnen, ohne die Benutzeroberfläche zu blockieren, und liefert gleichzeitig die Anzahl der bis dahin gefundenen Primzahlen.

[1] https://github.com/tariva/BA_Tauri_ElectronJS

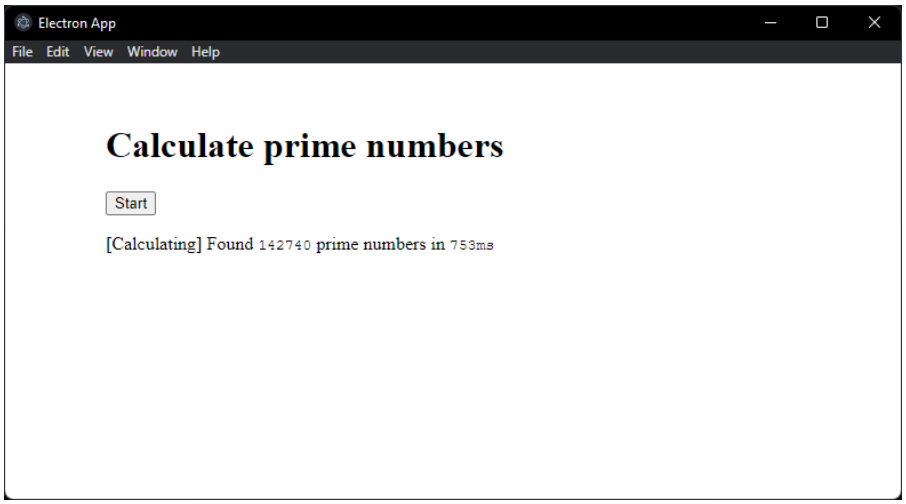

Abbildung 12 Electron Anwendung zum Erfassen der Leistungskennzahlen der Prozessorleistung

Die Dateizugriffstests messen die Geschwindigkeit beim Auflisten des Inhalts eines Ordners mit 10.000 Dateien und beim Lesen des Inhalts dieser Dateien sowohl sequenziell als auch gleichzeitig (mit maximal 100 gleichzeitig geöffneten Dateien). Dieser Test zielt darauf ab, die Benutzeroberfläche nicht zu blockieren, daher wird async/await sowohl in Electron als auch in Rust verwendet. Die Tests werden für Dateien von 4KB und 1MB durchgeführt.

Die IPC-Tests messen die Geschwindigkeit von Renderer -> Main -> Renderer für Electron mit ipcMain, ipcRenderer und WebView2 -> Rust -> WebView2 für Tauri unter Verwendung von Events. Die Tests senden eine bestimmte Anzahl von Nachrichten ohne jegliche Optimierung, um Standardbedingungen zu testen.

Dieses Benchmarking-Setup basiert auf dem Code, aus dem Crossplatform-dev-Repository auf Github[2], indem Electron zu anderen WPF verglichen wird.

5.4 Code Übersicht / Benchmark Übersicht

Die folgenden Benchmarks wurden unter Windows 11 (16 Kerne 32GB Arbeitsspeicher) und MX-Linux (4 Kerne 4GB Arbeitsspeicher durchgeführt. Bei beiden Frameworks wurde der im Standard konfigurierte Buildprozess verwendet.

5.4.1 CPU-Benchmark

Es wurde versucht die Struktur so zu gestalten, dass diese zwischen den beiden Frameworks möglichst übereinstimmt. Am besten war dies beim Prozessorleistungsvergleich möglich. Hier ist der Frontendcode identisch des Webworkers identisch und das Skript für die Oberfläche unterscheidet sich nur in den Pfaden, die benötigt werden.

```
// Create web worker
const THRESHOLD = 10000000;
const worker = new Worker("./worker.js");
const ITERATIONS = 15;
let resolver;

const onMessage = (message) => {
  // Update the UI
```

[2] https://github.com/crossplatform-dev/xplat-challenges

```
  // gekürzt..

  if (message.data.status === "done") {
    resolver(message.data.time);
  }
};


const benchmark = () => {
  return new Promise((resolve) => {
    const startTime = Date.now();
    resolver = resolve;
    worker.postMessage({ value: THRESHOLD, startTime });
  });
};

const calculate = async () => {
  let total = 0;

  for (let i = 0; i < ITERATIONS; i++) {
    const result = await benchmark();
    total += result;
  }

  const average = total / ITERATIONS;

  results.innerText = `Average time: ${average}ms`;
};
```

Quelltext-Auszug 1Programmablauf CPU Benchmark

Zunächst wird ein Web-Worker eingerichtet, der die Berechnungen durchführt, wobei sich das Skript des Workers unter "./worker.js" befindet. Die Funktion onMessage ist ein Ereignis-Listener für die Nachrichten des Workers. Sie aktualisiert die Benutzeroberfläche mit dem aktuellen Status des Workers und der Anzahl der bisher gefundenen Primzahlen. Benchmark ist eine Funktion, die eine Nachricht an den Worker sendet, damit dieser mit der Berechnung der Primzahlen beginnt, und ein Promise zurückgibt, das aufgelöst wird, wenn der Worker fertig ist, und die benötigte Zeit angibt. Calculate ist eine asynchrone Funktion, die die Benchmark-Operation mehrmals durchführt (bestimmt durch ITERATIONS), auf die Ergebnisse wartet und die durchschnittlich benötigte Zeit berechnet.

Die durch schnittliche Dauer der Berechnung für Primzahlen unter 10000000 bei 15 Iterationen

Platform	Tauri	Electron
Windows	~2212ms	~2235ms
Linux	~6191ms	~7563ms

Tabelle 4 Ergebnisse CPU-Benchmark

Bei der Analyze laufen beide Anwendungen gleich schnell, und es kein Overhead vom Framework erkennbar. Tauri benötigt bei der Ausführung 7 Prozesse, wobei nur beim WebView Prozess eine messbare Last entsteht. Electron benötigt 4 Prozesse, wobei ebenfalls nur bei einem eine messbare Last vorhanden ist.

Dabei die Ressourcen Auslastungen mit Taskmanager unter Windows und htop unter Linux

Platform	Tauri	Electron
CPU Windows Task-Manager	12%	10%
CPU Linux Htop	26%	26%

Tabelle 5 Prozessorauslastung während dem CPU-Benchmark

Da die beiden Anwendungen eine sehr ähnlich Webanwendung ausführen ist es auch ein gutes Beispiel, um die Paket Größe einer kleinen Anwendung zu untersuchen

Platform (Metrik)	Tauri	Electron
Installer (Windows)	2,37MB	92,7MB
Größe auf Dateiträger (Windows)	5,14MB	316MB
Installer (.deb)	4MB	67.6MB
Größe auf Dateiträger (Linux)	11MB	203,6MB
Arbeitsspeicher gestartet (Windows)	60,4MB	62MB
Arbeitsspeicher während Berechnung (Windows)	90,2MB	94MB
Arbeitsspeicher gestartet (Linux)	147,8MB	200MB
Arbeitsspeicher während Berechnung (Linux)	225,1MB	256MB

Tabelle 6 Größenvergleich auf den Platformen

5.4.2 File-Access Benchmark

In diesem Benchmark ist es möglich, 1000 Dateien mit einer Größe von 4kb und 1000 Dateien mit 1MB entweder sequenziell oder parallel zu lesen und zu schreiben. Der derzeitige Dateizugriffstest misst die Geschwindigkeit in folgenden Szenarien:

- Schreiben von 1000 Dateien
- Auflistung der Inhalte eines Ordners mit 1000 Dateien
- Lesen des Inhalts dieser 1000 Dateien in sequenzieller Reihenfolge
- Gleichzeitiges Lesen des Inhalts dieser 1000 Dateien, wobei maximal 100 Dateien gleichzeitig geöffnet sein können

```
contextBridge.exposeInMainWorld("electron", {
  require: (moduleName) => {
    // Check the module name and return only the allowed modules
    if (["fs", "path", "del", "util"].includes(moduleName)) {
      return require(moduleName);
    }
    if (moduleName === "ROOT") {
      return process.cwd();
    }
    throw new Error(`Cannot require '${moduleName}' module in renderer.`);
```

```
  },
});
```

Quelltext-Auszug 2 Zugriff auf Bibliotheken gewähren in Electron

Für den Dateibenchmark wurden Node.js Bibliotheken verwendet. Diese wurden über die contextBridge für den Renderer Prozess verfügbar gemacht.

```
export function createTauriCommands() {
  const readAsync = async (path) => {
    return await invoke("read_file", { args: { path } });
  };

  const writeAsync = async (path, content) => {
    return await invoke("write_file", { args: { path, content } });
  };
  // ...
```

Quelltext-Auszug 3 Aufruf einer Tauri Funktion in JavaScript

```
/// Read the content of a file specified by the path in `ReadFileArgs`
#[tauri::command]
pub async fn read_file(args: ReadFileArgs) -> Result<String, String> {
    let mut file = File::open(args.path)
        .await
        .map_err(|e| e.to_string())?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .await
        .map_err(|e| e.to_string())?;
    Ok(contents)
}

/// Write content to a file specified by the path in `WriteFileArgs`
#[tauri::command]
pub fn write_file(args: WriteFileArgs) -> Result<(), String> {
    let mut file = fs::File::create(args.path).map_err(|e| e.to_string())?;
    file.write_all(args.content.as_bytes())
        .map_err(|e| e.to_string())
}
```

Quelltext-Auszug 4 Definition von Tauri Funktionen in Rust

In Tauri wurden die benötigten Funktionen durch das #[tauri::command] Attribut-Makro registriert und können somit über die Invoke-Methode aufgerufen werden. Die Join-Methode wurde direkt von der API, die Tauri zur Verfügung stellt, integriert. Dies ermöglichte eine Code-Gestaltung, die der Electron-Version sehr ähnlich ist. Die Umsetzung in Rust war notwendig, weil Tauri eine Dateisystem-API bereitstellt, die allerdings sehr einschränkend ist - sie erlaubt beispielsweise keine absoluten Pfadangaben. Um die Tests vergleichbar und einfacher zu gestalten, wurden sie eigenständig implementiert.

Aufgabe mit 1000 Dateien (Plattform, Dateigröße)	**Tauri**	**Electron**

Dateien schreiben (Windows, 1mb)	~92971ms	~1505ms
Verzeichnis lesen (Windows, 1mb)	~26ms	~2,4ms
Dateien lesen sequenziell (Windows, 1mb)	~1114ms	~101471ms
Dateien lesen parallel (Windows, 1mb)	~532.6ms	~64587ms
Dateien schreiben (Windows, 4kb)	~113ms	~1165ms
Verzeichnis lesen (Windows, 4kb)	~3ms	~21ms
Dateien lesen sequenziell (Windows, 4kb)	~198.4ms	~1276ms
Dateien lesen parallel (Windows, 4kb)	~23.2ms	~401ms
Dateien schreiben (Linux, 1mb)	~2707ms	~3335ms
Verzeichnis lesen (Linux, 1mb)	~6ms	~10ms
Dateien lesen sequenziell (Linux, 1mb)	~6418ms	~2329.8ms
Dateien lesen parallel (Linux, 1mb)	~1526ms	~1479ms
Dateien schreiben (Linux, 4kb)	~212ms	~141,6ms
Verzeichnis lesen (Linux, 4kb)	~13ms	~9,4ms
Dateien lesen sequenziell (Linux, 4kb)	~253ms	~351,8ms
Dateien lesen parallel (Linux, 4kb)	~82ms	~120,4ms

Tabelle 7 Ergebnisse File-Access Benchmark

Auf der Grundlage der Daten können wir feststellen, dass Electron im Allgemeinen in allen getesteten Szenarien auf beiden Plattformen signifikant schneller ist als Tauri. Wenn wir uns beispielsweise die Ergebnisse für das Schreiben von 1000 Dateien mit einer Größe von 1MB ansehen, sehen wir, dass Electron etwa 62-mal schneller ist als Tauri. Auch beim sequenziellen Lesen dieser Dateien zeigt Electron eine etwa 91-fache Geschwindigkeit gegenüber Tauri.

Auch bei kleineren Dateien, in diesem Fall 4kb, zeigt Electron eine hervorragende Leistung. Bei der Schreiboperation ist Electron etwa 10-mal schneller als Tauri und beim sequenziellen Lesen der Dateien ist es etwa 6-mal schneller. Interessanterweise ist die Diskrepanz beim parallelen Lesen der 4kb-Dateien noch größer, wo Electron etwa 17-mal schneller ist als Tauri.

Einige der Gründe für die überlegene Leistung von Electron könnten auf die reifere Codebasis und die Optimierung zurückzuführen sein, die im Laufe der Jahre entwickelt wurde.

5.4.3 IPC-Benchmark

In Electron kommunizieren Prozesse, indem sie Nachrichten durch vom Entwickler definierte "Kanäle" mit den Modulen ipcMain und ipcRenderer senden. Diese Kanäle sind beliebig (sie können

beliebig benannt werden) und bidirektional (man kann den gleichen Kanalnamen für beide Module verwenden).

Tauri hingegen verwendet eine spezielle Form der Interprozesskommunikation namens "Asynchronous Message Passing". Bei diesem Verfahren tauschen Prozesse Anfragen und Antworten aus, die mithilfe einer einfachen Datenrepräsentation serialisiert sind. Message Passing ist eine sicherere Technik als Shared Memory oder direkter Funktionszugriff, da der Empfänger in der Lage ist, Anfragen nach Belieben abzulehnen oder zu verwerfen.

Wir werden uns im Folgenden die zwei IPC-Grundbausteine von Tauri - Events und Commands - genauer ansehen. Bitte beachten Sie, dass wir in diesem Benchmark "Standardwerte" testen, daher kommen Mechanismen zur Optimierung von stark frequentierten IPCs nicht zum Einsatz.

```
const SEND_CHANNEL = "asynchronous-message";
const RECEIVE_CHANNEL = "asynchronous-reply";

contextBridge.exposeInMainWorld("comms", {
  sendMessage: (message) => ipcRenderer.send(SEND_CHANNEL, message),
  onMessage: (handler) => ipcRenderer.on(RECEIVE_CHANNEL, handler),
  clear: () => ipcRenderer.removeAllListeners(RECEIVE_CHANNEL),
});
```

Quelltext-Auszug 5 Definition von Kanälen in Electron

```
ipcMain.on("asynchronous-message", (event, ...args) => {
  event.reply("asynchronous-reply", ...args);
});

ipcMain.on("asynchronous-reply", (event, ...args) => {
  mainWindow.webContents.send("asynchronous-reply", ...args);
});

ipcMain.on("asynchronous-message-proxy", (event, ...args) => {
  backgroundWindow.webContents.send("asynchronous-message", ...args);
});
```

Quelltext-Auszug 6 Weiterleiten von Kanalinhalten in Electron

Hier wird eine sichere, asynchrone Kommunikation zwischen den Haupt- und Renderer-Prozessen in einer Electron-Anwendung aufgesetzt. Die ipcMain- und ipcRenderer-Module ermöglichen eine bidirektionale Kommunikation zwischen diesen Prozessen. Die von contextBridge.exposeInMainWorld bereitgestellte "comms"-Schnittstelle ermöglicht es dem Renderer-Prozess, Nachrichten an den Hauptprozess zu senden und auf eingehende Nachrichten vom Hauptprozess zu reagieren. Die ipcMain.on Listener im Hauptprozess ermöglichen es, auf Nachrichten zu reagieren, die vom Renderer-Prozess gesendet werden, und darauf zu antworten.

```
window.comms = {
  sendMessage: (message) => emit(SEND_CHANNEL, message),
  onMessage: (handler) =>
    listen(RECEIVE_CHANNEL, (event) => handler(event.payload)),
  clear: () => {}, // No clear equivalent in the new API
};
```

Quelltext-Auszug 7 Definition von Kanälen in Tauri

```
app.listen_global("asynchronous-message", move |event| {
  match event.payload() {
    Some(payload) => {
      match from_str::<Message>(payload) {
        Ok(message) => {
          // Emit the `asynchronous-reply` event to all webview windows on
the frontend
          if let Err(e) = handle.emit_all("asynchronous-reply", &message) {
              eprintln!("Failed to emit event: {}", e);
          }
        }
        Err(e) => eprintln!("Failed to deserialize payload: {}", e),
      }
    }
    None => eprintln!("Received event with no payload"),
  }
});
```

Quelltext-Auszug 8 Reaktion auf empfangen Nachrichten in Tauri

Auch Bei Tauri kann eine ähnliche Struktur genutzt werden, um zwischen Frontend und Backend zu kommunizieren. Es wird empfohlen diese für die Lifecycle Ereignisse und für Statusänderungen zu verwenden.

Beim Benchmark gibt es zwei verschiedene Modi, sie unterscheiden sich in der Art und Weise, wie die Nachrichten versandt werden. Im Burst-Modus werden alle Nachrichten nahezu gleichzeitig versandt. Es wird eine Nachricht für jede Iteration der Schleife erstellt und versandt. Ein Event-Handler empfängt dann jede Antwortnachricht, registriert deren Ankunftszeit und fügt sie einer Liste von Nachrichten hinzu. Wenn alle Nachrichten empfangen wurden, werden die Ergebnisse berechnet und angezeigt. Im Gegensatz dazu versendet der sequenzielle Modus die Nachrichten nacheinander - jede neue Nachricht wird erst versandt, nachdem die vorherige Antwort empfangen wurde. Jedes Mal, wenn eine Antwortnachricht empfangen wird, wird eine neue Nachricht versandt, bis alle Nachrichten versandt und empfangen wurden.

Metrik (Platform)	**Tauri**	**Electron**
„Burst“ Modus 500 Events (Windows)	117.00ms	17.00ms
„Sequentieller“ Modus 500 Events (Windows)	268.00ms	66.40ms
„Burst“ Modus 5000 Events (Windows)	1192ms	323ms
„Sequentieller“ Modus 5000 Events (Windows)	2522ms	643ms
„Burst“ Modus 500 Events (Linux)	206ms	76ms
„Sequentieller“ Modus 500 Events (Linux)	338ms	169ms

„Burst“ Modus 5000 Events (Linux)	1833ms	527ms
„Sequentieller“ Modus 5000 Events (Linux)	3274ms	1827ms

Tabelle 8 Ergebnisse IPC-Benchmark

Bei einer Analyse des "Burst"-Modus, in dem 500 Ereignisse fast gleichzeitig gesendet werden, ist Electron ungefähr siebenmal schneller als Tauri auf Windows (17 ms gegenüber 117 ms) und etwa 2.7-mal schneller auf Linux (76 ms gegenüber 206 ms). Bei 5000 Ereignissen ist Electron auf Windows etwa 3.7-mal schneller als Tauri (323 ms gegenüber 1192 ms) und auf Linux etwa 3.5-mal schneller (527 ms gegenüber 1833 ms).

Im sequenziellen Modus, bei dem Nachrichten nacheinander gesendet und empfangen werden, ist Electron auf Windows bei 500 Ereignissen etwa viermal schneller als Tauri (66,4 ms gegenüber 268 ms) und etwa 2-mal schneller auf Linux (169 ms gegenüber 338 ms). Bei 5000 Ereignissen ist Electron auf Windows etwa viermal schneller als Tauri (643 ms gegenüber 2522 ms) und auf Linux etwa 1.8-mal schneller (1827 ms gegenüber 3274 ms).

Diese Daten deuten darauf hin, dass Electron eine effizientere Interprozesskommunikation (IPC) bietet als Tauri, zumindest in den getesteten Szenarien. Unabhängig vom Modus und der Anzahl der Ereignisse übertraf Electron in diesen Tests konsequent Tauri in Bezug auf die Geschwindigkeit.

5.5 Performance Guide für ElectronJS

Die Optimierung der Performance von ElectronJS-Anwendungen umfasst verschiedene Strategien, wie z.B. die Minimierung der Speicher-, CPU- und Festplattennutzung und die Sicherstellung einer schnellen Reaktion der App. Profiling und Messungen sind unerlässlich, um die ressourcenintensivsten Teile der App zu erkennen und zu optimieren. Tools wie die Chrome Developer Tools und das Chrome Tracing Tool sind in diesem Zusammenhang nützlich.

Bevor Sie ein Node.js-Modul in Ihre Anwendung integrieren, müssen Sie unbedingt seine Abhängigkeiten, die benötigten Ressourcen und seine Auswirkungen auf Speicher und CPU bewerten. Die Beliebtheit eines Moduls bedeutet nicht unbedingt, dass es das effizienteste oder kleinste Modul für Ihre Bedürfnisse ist.

Wenn Sie teure Operationen auf die Interaktionen des Benutzers abstimmen, anstatt alle Operationen gleichzeitig auszuführen, kann dies die Leistung verbessern. Vermeiden Sie das Blockieren der Haupt- und Renderer-Prozesse, was Ihre Anwendung zum Stillstand bringen kann. Die Multiprozess-Architektur von Electron kann lange laufende Aufgaben bewältigen, ohne den UI-Thread zu blockieren.

Unnötige Polyfills sollten ebenfalls vermieden werden, da Electron bereits viele Funktionen unterstützt und die Aufnahme dieser Funktionen Ressourcen verschwenden kann. Minimieren Sie Netzwerkanfragen für Ressourcen, die sich nur selten ändern und mit der Anwendung gebündelt werden können.

Schließlich kann die Bündelung des Codes Ihrer Anwendung in einer einzigen Datei den mit dem Aufruf von require() verbundenen Overhead erheblich reduzieren. Tools wie Webpack, Parcel und rollup.js sind effektiv im Umgang mit der einzigartigen Umgebung von Electron, die Node.js und Browser-Umgebungen miteinander verbindet

5.5.1 Unterschiede im Guide von Tauri

Die Strategien zur Leistungsoptimierung unterscheiden sich erheblich zwischen Tauri- und ElectronJS-Anwendungen, wobei jede Plattform ihre einzigartigen Werkzeuge und Methoden empfiehlt. Tauri, beispielsweise, empfiehlt für Rust-Anwendungen den Einsatz von Tools wie cargo-bloat und cargo-expand. Für JavaScript-Abhängigkeiten schlägt es rollup-plugin-visualizer und rollup-plugin-graph vor. Auf der anderen Seite favorisiert Electron die Verwendung der Chrome Developer Tools und des Chrome Tracing Tools für Profiling und Messungen.

Darüber hinaus bietet Tauri in seiner Standardkonfiguration weit weniger APIs als ElectronJS, was bedeutet, dass diese explizit in einer sogenannten "Allowlist"-Konfiguration zugelassen werden müssen. Diese bewusste Einschränkung schafft ein Bewusstsein für die Kosten, die mit der Aktivierung verbunden sind. Eine weitere Unterscheidung zwischen den beiden ist, dass Tauri, da es in Rust entwickelt wurde, Rust-spezifische Optimierungen bietet, die auf Electron nicht anwendbar sind. Diese betreffen vor allem Optimierungen bei der Rust-Bauzeit und das sogenannte "Stripping".

Kapitel 6

6 Beispiele aus der Praxis und Fallstudien

ElectronJS wurde aufgrund seiner Flexibilität, seines umfangreichen Funktionsumfangs und der starken Unterstützung durch die Community von zahlreichen bekannten Unternehmen und Projekten übernommen. Zu den bekannten Anwendungen, die mit Electron entwickelt wurden, gehören Visual Studio Code (Microsoft), Slack, Discord, Atom (GitHub) und Trello, neben vielen anderen. (OpenJS Foundation, Electron contributors, 2023) Die Vielzahl erfolgreicher Projekte, die mit Electron entwickelt wurden, zeigt die Fähigkeiten und die Vielseitigkeit des Frameworks bei der Erstellung leistungsstarker plattformübergreifender Desktop-Anwendungen unter Verwendung von Webtechnologien.

Ein Beispiel von Tauri aus der Praxis ist: Xplorer, A Free and Open Source (FOSS) File Explorer. Ursprünglich eine ElectronJS Applikation diese wurde auf Grund eines Community Vorschlages auf Tauri migriert. Durch die Migration konnte der Entwickler wertvolle Informationen sammeln. (Kimlim, 2021)

- Performance und Speicher Verbesserungen wurden festgestellt.
- Das Installationsfile ist 90% Kleiner
- Die Applikation startet schneller (von ~ 1.52 s to ~ 1.28 s)
- Ist Thread sicher
- Einige Features von Electron werden von Tauri nicht unterstützt.
- Interaktionen sind langsamer als auf Electron.

Als neues Framework hat Tauri noch nicht so viele Erfolgsgeschichten vorzuweisen wie ElectronJS. Es gibt jedoch mehrere Projekte, die das Potenzial und die Fähigkeiten von Tauri zeigen diese werden auf dem Disordserver von Tauri und auf GitHub[3] vorgestellt.

- Steam Art Manager: Ein Tool zum Anpassen des Artworks Ihrer Steam-Bibliothek.[4]
- NanoCode: Minimalistischer, quelloffener und leichtgewichtiger Code-Editor in Tauri.[5]

Awesome Tauri[6] ist eine kuratierte Sammlung der besten Dinge aus dem Tauri-Ökosystem und der Community. In dieser Sammlung finden sich einige Anleitungen, Templates für Frontendframeworks Plugins sowie Projekte und Artikel.

Mit ElectronJS wurden zahlreiche erfolgreiche und hochkarätige Projekte erstellt, die seine Vielseitigkeit und Fähigkeiten unter Beweis stellen

[3] https://github.com/tauri-apps/awesome-tauri
[4] https://github.com/Tormak9970/Steam-Art-Manager
[5] https://github.com/azedeveloper/NanoCode
[6] https://github.com/tauri-apps/awesome-tauri

6.1 Crates im Vergleich zu NPM-Paketen

Einer der wichtigsten Vorteile von Electron für die Entwicklung von Desktop-Anwendungen ist die Kompatibilität mit jedem Node.js-Paket, wodurch die Funktionalität einer Electron-App erweitert wird. Das liegt an der hybriden Natur von Electron, die Node.js und die Rendering-Engine von Chromium kombiniert und es Entwicklern ermöglicht, die volle Leistung des Node.js Ökosystems zu nutzen. So können beispielsweise Pakete wie express[7] für die Einrichtung lokaler Server, socket.io[8] für die Echtzeitkommunikation und lowdb[9] für die eingebettete Datenspeicherung verwendet werden. Darüber hinaus ermöglichen Pakete wie electron-store[10] eine einfache Datenpersistenz, während electron-updater automatische Updates von freigegebenen Anwendungen ermöglicht. Wenn Ihre Anwendung mit APIs auf Systemebene interagieren muss, bieten Pakete wie node-ffi-napi[11] eine Schnittstelle zu nativem Code. Die große Auswahl an Node.js-Paketen kann also die Möglichkeiten einer Electron-Anwendung erheblich erweitern und bietet mehr Flexibilität und Optionen zur Erstellung robuster und funktionsreicher Desktop-Anwendungen. „Im September 2022 wurden mehr als 2,1 Millionen Pakete in der npm-Registry verzeichnet. Damit ist es das größte Repository für Sprachcode auf der Welt, und Sie können sicher sein, dass es für (fast!) alles ein Paket gibt.“ (OpenJS Foundation, 2023)

So wie Electron von dem umfangreichen Ökosystem der Node.js-Pakete profitiert, macht sich auch Tauri die reichhaltige Sammlung von Rust Crates zunutze. Wie Node.js verfügt auch Rust über ein umfassendes Ökosystem von Bibliotheken, den so genannten Crates, die eingesetzt werden können, um die Funktionalität einer Tauri-Anwendung zu erweitern. Zum Beispiel ist 'serde' eine Crate, die für die Serialisierung und Deserialisierung von Daten verwendet wird, 'reqwest' wird für die Erstellung von HTTP-Anfragen verwendet und 'tokio' vereinfacht das Schreiben von asynchronem Code. Wenn die Anwendung komplexe Berechnungen erfordert, können die Vorteile von Rust hinsichtlich der Leistung und des sicheren Concurrency-Modells mit Crates wie 'rayon' ausgenutzt werden. Ein entscheidender Vorteil von Rust Crates ist, dass sie in der Regel in Maschinencode kompiliert werden, was möglicherweise Leistungsvorteile gegenüber interpretierten oder JIT-kompilierten Sprachen wie JavaScript bietet. Allerdings listet das Crate-Verzeichnis Creates.io[12] im Vergleich zu npm derzeit nur 112.933 Pakete auf.

[7] https://expressjs.com/
[8] https://socket.io/
[9] https://github.com/typicode/lowdb#readme
[10] https://github.com/sindresorhus/electron-store#readme
[11] https://github.com/node-ffi-napi/node-ffi-napi
[12] https://crates.io/

7 Plattformübergreifende Kompatibilität

ElectronJS unterstützt die 3 gängigen Betriebssysteme: Aktuell empfiehlt Electron die Distribution mit Electron Forge. Electron Forge ist eine Sammlung verschiedener Tools die Distribution und den Build-Prozess für die unterstützten Plattformen erleichtern. (OpenJS Foundation and Electron contributors, 2021)

Der Tauri Bundler ist ein Rust-Werkzeug, das die Binärdatei kompiliert, Assets verpackt und das endgültige Bundle für die Bereitstellung vorbereitet. Es identifiziert das Betriebssystem und erstellt ein entsprechendes Bundle. Für Windows unterstützt es die Erstellung von -setup.exe- und .msi-Dateien. Unter macOS erstellt es .app- und .dmg-Dateien. Für Linux erstellt es .deb- und .appimage-Dateien. (Tauri Contributors, 2023)

Mit der Einführung der Version 2.0 von Tauri ist es möglich, IPA-Dateien zu erstellen. Diese Dateien können an Apple übermittelt werden, um eine Freigabe für den App Store zu erreichen. Darüber hinaus bietet Tauri 2.0 auch Unterstützung für Android. Es ermöglicht die Erstellung einer APK-Datei, die zur Installation der App auf einem Android-Betriebssystem verwendet werden kann. (Tauri Contributors, 2023)

Die Unterstützung Für IOS und Android ist noch in der Betaphase, deshalb wird sie in diesem Vergleich nicht hinzugezogen. Falls die Unterstützung aber genauso gut funktioniert, wie bei den Desktop Betriebssystemen könnte das Tauri einen großen Vorteil bieten. Da ElectronJS historisch bedingt die Portierung nicht so ohne erheblichen Aufwand durchführen kann. „ For mobile platform, nearly all of atom-shell's APIs don't apply, so I don't think we will ever support mobile platforms.” (Anon., 2014) Die großen Key User wie Visual Studio Code bereits sich bereits für andere Wege entschieden haben. Wie eine Browser App. (Chan, 2021) Oder wie im Falle von Discord eine Implementierung mit React Native. (Chen, 2016)

7.1 Entwicklererfahrung

7.1.1 Tauri

Tauri zielt darauf ab, eine unkomplizierte und angenehme Erfahrung für Entwickler zu bieten, die den Einstieg relativ leicht macht. Die folgenden Aspekte tragen dazu bei, dass der Einstieg in Tauri leichtfällt:

Wie bereits erwähnt bietet Tauri eine umfassende Dokumentation mit Anleitungen für den Einstieg, API-Referenzen und Best Practices, die es den Entwicklern leicht macht, die Informationen zu finden, die sie für die Arbeit mit dem Framework benötigen. (Tauri Contributors, 2023)

Tauri bietet eine Befehlszeilenschnittstelle (CLI), die die Projekteinrichtung, die Entwicklung und die Erstellung von Projekten vereinfacht. Die CLI bietet eine Vielzahl von Befehlen, mit denen Entwickler Tauri-Anwendungen mühelos erstellen und verwalten können. (Tauri Contributors, 2023)

Da Tauri Webtechnologien für die Erstellung von Anwendungen nutzt, können Entwickler mit Erfahrung in HTML, CSS und JavaScript schnell mit dem Framework arbeiten. Die Verwendung der Programmiersprache Rust für das Backend von Tauri bringt einen zusätzlichen Lernaspekt mit sich, aber die Dokumentation bietet ausreichende Ressourcen, um Entwicklern zu helfen, Rust zu erlernen und effektiv zu nutzen. Weiters waren die Entwickler bemüht die API so zu gestalten, dass Webentwickler auch ohne Rust Kenntnisse Grundlegende Problemstellungen bewältigen können. (Tauri: Building better apps for a better future, 2022)

7.1.2 ElectronJS

ElectronJS verfügt über ein etablierteres und umfangreicheres Ökosystem, was dazu beiträgt, dass der Einstieg in das Framework leichter fällt. Die folgenden Faktoren machen es Entwicklern leicht, mit ElectronJS zu beginnen:

ElectronJS bietet eine gründliche und gut gepflegte Dokumentation, die verschiedene Themen wie die ersten Schritte, die API-Referenz und bewährte Sicherheitsverfahren abdeckt

Electron Fiddle ist eine Sandbox-Anwendung, die mit Electron geschrieben wurde und von den Entwicklern von Electron unterstützt wird. Es empfiehlt sich, diese Anwendung als Lernwerkzeug zu installieren, um mit den APIs von Electron zu experimentieren oder um Funktionen während der Entwicklung zu testen. Fiddle ist auch gut mit unserer Dokumentation integriert, so können Code Beispiel Direkt ausprobiert werden.

Vertrautheit mit Webtechnologien: ElectronJS verwendet Webtechnologien wie HTML, CSS und JavaScript für die Erstellung von Anwendungen, so dass Entwickler mit Erfahrung in diesen Sprachen leicht auf das Framework umsteigen können. Darüber hinaus integriert ElectronJS Node.js für sein Backend, was für viele JavaScript-Entwickler eine vertraute Umgebung ist. (OpenJS Foundation and Electron contributors, 2021)

7.2 Vergleich von Benutzerfreundlichkeit und Tools

Sowohl Tauri als auch ElectronJS sind bestrebt, ein angenehmes und unkompliziertes Entwicklererlebnis zu bieten. Wenn Sie die Benutzerfreundlichkeit, die Werkzeuge und die Erfahrung der Entwickler zwischen den beiden Frameworks vergleichen, sollten mehrere Faktoren berücksichtigen werden:

Dokumentation: Zwar bieten beide Frameworks eine umfassende Dokumentation, doch ElectronJS verfügt aufgrund seiner größeren und etablierteren Community über eine umfangreichere Sammlung von Ressourcen. Die Dokumentation von Tauri ist jedoch gut strukturiert und zugänglich, so dass es für Entwickler einfach ist, relevante Informationen zu finden.

CLI und Projekteinrichtung: Beide Frameworks bieten Befehlszeilenschnittstellen und Tools, die die Einrichtung und Entwicklung von Projekten vereinfachen. ElectronJS verfügt über eine breitere Palette von Tools wie Electron-forge und Electron-builder, die den Bereitstellungsprozess vereinfachen können. Die CLI von Tauri bietet eine Vielzahl von Befehlen zur Verwaltung von Tauri-Anwendungen und trägt so zu einer reibungslosen Entwicklung bei. Dadurch, dass Tauri noch recht jung ist, gibt das Entwicklerteam und die enge Opensourcecommunity noch viel vor. Bei Electron ist die Motivation eigene Tools zu entwickeln viel großer da es bereits einen Markt dafür gibt. So muss das Team hinter Electron viele dieser Tools zu einem Best-Practice-.Tool wie Electron Forge zusammenfassen. Während das Tauri Team zurzeit nur einen Weg besitzt Projekte zu initialisieren, und zwar mit der Tauri CLI

Vertrautheit mit Webtechnologien: Sowohl Tauri als auch ElectronJS nutzen Webtechnologien, so dass sie für Entwickler mit Erfahrung in HTML, CSS und JavaScript zugänglich sind. Während ElectronJS Node.js für sein Backend verwendet, setzt Tauri Rust ein, was für einige Entwickler einen zusätzlichen Lernaspekt darstellen kann. Die Tauri-Dokumentation bietet jedoch genügend Ressourcen, um Entwicklern zu helfen, Rust zu erlernen und effektiv zu nutzen.

Beide Frameworks bieten eine angenehme Erfahrung für Entwickler, mit einer umfassenden Dokumentation, CLI-Tools und Optionen für die Projekteinrichtung. Während ElectronJS aufgrund seiner größeren Community über ein umfangreicheres Angebot an Ressourcen verfügt, ist Tauri

aufgrund seines Schwerpunkts auf Einfachheit und Leistung eine attraktive Wahl für Entwickler, die an einer schlanken und sicheren Alternative interessiert sind.

Bei der Wahl zwischen Tauri und ElectronJS sollten Entwickler ihre bisherigen Erfahrungen mit Webtechnologien, Backend-Programmiersprachen (Rust vs. Node.js) und ihre Präferenzen in Bezug auf Tools und Dokumentation berücksichtigen. Beide Frameworks bieten eine solide Grundlage für die Erstellung plattformübergreifender Desktop-Anwendungen mit Webtechnologien, wobei die Komplexität und die Lernkurve je nach Hintergrund und Vertrautheit des Entwicklers mit den jeweiligen Technologien unterschiedlich sind.

8 Schlussfolgerung

Dieses Dokument bietet einen umfassenden Vergleich zwischen Tauri und ElectronJS. Dabei werden Aspekte wie Einführung, Architektur, Funktionen, Sicherheit, Lernressourcen, plattformübergreifende Kompatibilität, Lernkurve, Erfahrungen von Entwicklern, reale Anwendungsfälle und Zukunftsaussichten behandelt.

Beide Frameworks bieten eine umfassende Dokumentation, plattformübergreifende Kompatibilität und ein angenehmes Entwicklererlebnis. Während Tauri sich mehr auf die Leistung und ein natives Benutzererlebnis konzentriert, bietet ElectronJS eine bewährte Lösung mit reichlich Ressourcen und einer langen Liste von erfolgreichen, hochkarätigen Projekten.

8.1 Abschließende Gedanken zum Vergleich

Zum aktuellen Zeitpunkt gibt es noch keine kommerziell erfolgreiche Anwendung mit Tauri aus denen man Rückschlüsse ziehen könnte. Um eine bestehende Webanwendung als Desktop Anwendung zugänglich zu machen ist Tauri mit Sicherheit eine sehr gute Option.

Electron ist aktuell schneller, was die Kommunikation zwischen den Prozessen betrifft, damit ist es sicher die bessere Wahl für Applikationen, die hohen Gebrauch von IPC machen. Auch ist die Auswahl an fertigen Lösungen in Form für Bibliotheken viel größer, sodass häufig gewünschte Features schnell eingebaut werden können.

8.2 Zusammenfassung der wichtigsten Ergebnisse

Electron	**Tauri**
Verfügbare Bibliotheken	Rust
Robustheit	Installationsfile Größe
Größe der Community	Größe im Dateisystem
NodeJs	Automatische WebView2 Updates
Schnelle IPC	
Kontrollierte Chromium Version	

Sowohl Tauri als auch ElectronJS einzigartige Vorteile bieten und unterschiedliche Bedürfnisse in der plattformübergreifenden Desktop-Anwendungsentwicklung abdecken. Entwickler sollten bei der Auswahl des am besten geeigneten Frameworks für ihre Projekte ihre spezifischen Anforderungen, Prioritäten und ihre Vertrautheit mit den jeweiligen Technologien abwägen. Sowohl Tauri als auch ElectronJS haben vielversprechende Zukunftsaussichten und werden die Landschaft der Web-App-Entwicklung wahrscheinlich weiter prägen, da sie sich weiterentwickeln und an die Bedürfnisse von Entwicklern und Endbenutzern anpassen.

Es kann auch hilfreich sein, eine Liste mit "Must-have"- und "Nice-to-have"-Funktionen und -Kriterien für Ihr Projekt zu erstellen. Diese Liste kann Ihnen als Leitfaden dienen, wenn Sie die beiden Frameworks vergleichen und Ihre Entscheidung treffen.

8.3 Ausblick

Tauri und ElectronJS haben beide das Potenzial, die Landschaft der Web-App-Entwicklung erheblich zu beeinflussen. Wenn mehr Entwickler diese Frameworks einsetzen, könnte die Grenze zwischen Webanwendungen und nativen Desktop-Anwendungen zunehmend verschwimmen. Dies könnte zu

einer größeren Nachfrage nach Webtechnologien und -kenntnissen führen, da Entwickler ihr vorhandenes Wissen nutzen können, um plattformübergreifende Anwendungen zu erstellen.

Die Ausweitung von Tauri auf mobile Plattformen könnte sich auch auf die Landschaft der Web-App-Entwicklung auswirken, da Entwickler mit einer einzigen Codebasis Anwendungen für Desktop- und mobile Geräte erstellen können. Dies würde die Rolle der Webtechnologien bei der plattformübergreifenden Anwendungsentwicklung weiter festigen.

Zusammenfassend lässt sich sagen, dass die Zukunftsaussichten sowohl für Tauri als auch für ElectronJS vielversprechend sind, da sie ständig weiterentwickelt werden und der Schwerpunkt auf der Verbesserung der Leistung, der Sicherheit und der Erfahrung der Entwickler liegt. Diese Frameworks werden wahrscheinlich weiterhin die Landschaft der Web-App-Entwicklung prägen und die Übernahme von Webtechnologien für die plattformübergreifende Anwendungsentwicklung fördern.

9 Works Cited

OpenJS Foundation, Electron contributors, 2021. *electronjs.org/.* [Online]
Available at: https://www.electronjs.org/docs/latest/
[Accessed 27 04 2023].

Anon., 2014. *github.com.* [Online]
Available at: https://github.com/electron/electron/issues/562#issuecomment-51735074
[Accessed 21 03 2023].

Anon., 2023. *tauri.app.* [Online]
Available at: https://tauri.app/v1/references/architecture/
[Accessed 02 05 2023].

Anon., 2023. *twitter.com.* [Online]
Available at: https://twitter.com/electronjs
[Accessed 06 04 2023].

Bulajic, A. S. S. &. S. R., 2013. *An Effective Development Environment Setup for System and Application Software..* Informing Science Institute, Proceedings of Proceedings of the Informing Science and Information Technology Education Conference.

Chan, S., 2021. *github.com.* [Online]
Available at: https://github.com/microsoft/vscode/issues/70764#issuecomment-954596653
[Accessed 21 03 2023].

Chen, F., 2016. *discord.com.* [Online]
Available at: https://discord.com/blog/using-react-native-one-year-later
[Accessed 21 03 2023].

Daniel Attevelt, P. K. T. W. M. B., 2022. *ROS - The Tauri Programme,* s.l.: s.n.

Fernandez, S., 2019. *msrc.microsoft.com.* [Online]
Available at: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/
[Accessed 05 01 2023].

Greif, S., 2023. *stateofjs.com.* [Online]
Available at: https://2021.stateofjs.com/en-US/libraries/mobile-desktop/
[Accessed 4 03 2023].

Kimlim, J. M., 2021. *dev.to.* [Online]
Available at: https://dev.to/kimlimjustin/xplorer-a-modern-file-explorer-that-was-written-using-typescript-has-its-performance-improved-on-the-recent-release-4f99
[Accessed 06 05 2023].

MSEdgeTeam, mikehoffms, nishitha-burman, vchapel, champnic,liminzhu,zoherghadyali,jasonstephen15,captainbrosset,jm-trd-ms,Reezaali,pagoe-msft,peiche-jessica,JoshuaWeber, 2023. *learn.microsoft.com.* [Online]
Available at: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution#evergreen-distribution-mode
[Accessed 02 05 2023].

Nokes, C., 2016. *cameronnokes.com.* [Online]
Available at: https://cameronnokes.com/blog/deep-dive-into-electron's-main-and-renderer-processes/
[Accessed 03 04 2023].

OpenJS Foundation and Electron contributors., 2023. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/context-isolation#what-is-it
[Accessed 15 05 2023].

OpenJS Foundation and Electron contributors, 2021. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/quick-start#access-nodejs-from-the-renderer-with-a-preload-script
[Accessed 22 03 2023].

OpenJS Foundation and Electron contributors, 2021. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/tutorial-adding-features
[Accessed 05 01 2023].

OpenJS Foundation and Electron contributors, 2021. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/api/ipc-main
[Accessed 09 05 2023].

OpenJS Foundation and Electron contributors, 2021. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/blog/webview2#architecture-overview
[Accessed 01 05 2023].

OpenJS Foundation and Electron contributors, 2021. *https://www.electronjs.org/.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/quick-start#package-and-distribute-your-application
[Accessed 16 05 2023].

OpenJS Foundation and Electron contributors, 2023. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/security
[Accessed 14 05 2023].

OpenJS Foundation, Electron contributors, 2021. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/docs/latest/tutorial/process-model
[Accessed 27 04 2023].

OpenJS Foundation, Electron contributors, 2023. *electronjs.org.* [Online]
Available at: https://www.electronjs.org
[Accessed 20 04 2023].

OpenJS Foundation, Electron contributors, 2023. *electronjs.org.* [Online]
Available at: https://www.electronjs.org
[Accessed 20 03 2023].

OpenJS Foundation, Electron contributors, 2023. *electronjs.org.* [Online]
Available at: https://www.electronjs.org/apps
[Accessed 25 03 2023].