Ying Zhang

Feasibility, Advantage and Challenge of Tauri Cross-Platform Application Development: One Codebase for Mobile and Web

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

7 February 2026

Abstract

Author:	Ying Zhang
Title:	Feasibility, Advantage, and Challenge of Tauri 2.0 Cross-Platform Application Development: One Codebase for Mobile and Web
Number of Pages:	40 pages
Date:	7 February 2026
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Ilkka Kylmäniemi, Senior Lecturer

The objective of the study is to investigate the feasibility of adopting Tauri—a framework for building small, fast, secure, cross-platform applications—as a primary development technology. While the work is motivated by the author's own interests and requirements, its intention is to provide insights that are useful for individual developers and teams evaluating cross-platform solutions.

The study is based on a literature review, theoretical analysis, and a practical case study. The literature review traces the evolution of web and mobile application development and situates Tauri within this landscape. The official Tauri material, such as documentation, source code, and architectural descriptions were analyzed, and a prototype YLE RSS feed application targeting macOS, iOS, and Android was implemented.

The study concludes that Tauri is a viable choice when lightweight binaries and reuse of web technologies are important but that further maturation of the ecosystem is needed. The findings can help practitioners judge whether Tauri fits their own projects and point to future work, such as richer use of plug-ins, larger-scale user studies, and systematic comparisons with other cross-platform frameworks.

Keywords: Tauri, cross-platform application, web development, mobile development

---

Contents

List of Abbreviations

AJAX:	Asynchronous JavaScript and XML
CERN:	The European Organization for Nuclear Research
DOM:	Document Object Model
HTML:	Hypertext Markup Language
HTTP:	Hypertext Transfer Protocol
iOS:	iPhone Operating System
MVP:	Minimum Viable Products
RSS:	Really Simple Syndication
PWA:	Progressive Web Applications
UI:	User Interface
URL:	Uniform Resource Locator
WWW:	World Wide Web
YLE:	Finnish Broadcasting Company

1 Introduction

In the present day, development teams often face the challenge of maintaining multiple separate codebases for native mobile platforms and web applications, which significantly increases development and maintenance costs. To address these challenges, cross-platform development frameworks such as Flutter, React Native, and Xamarin have been introduced. These frameworks leverage the common foundational principles shared by native and web development, enabling developers to build applications on unified underlying frameworks, thereby reducing duplication of effort and facilitating simultaneous deployment across multiple platforms. However, these frameworks often introduce trade-offs in terms of performance, application size, or security. Therefore, there remains a need to assess whether a new emerging framework can offer a sustainable alternative that balances performance, security, and maintainability while preserving development agility and platform compatibility.

Tauri 1.0, released in June 2022, is a framework designed for building lightweight, secure, and high-performance, multi-platform desktop applications targeting operating systems such as Linux, macOS, and Windows [1, p. 1]. Subsequently, in October 2024, Tauri introduced mobile support for iOS and Android platforms as part of its 2.0 version, extending its cross-platform capabilities to include mobile devices [2, p. 1]. Motivated by the emergence of new cross-platform application frameworks, the study aims to investigate the feasibility, advantage and challenges of utilizing Tauri as the primary development tool.

The thesis is structured into six chapters. Following the introductory chapter, Chapter 2 introduces the research question and outlines the research methodology to shape the comprehensive framework of the thesis. Chapter 3 offers an overview of web and mobile development, emphasizing both the advantages and disadvantages inherent to various development tools. Chapter 4 provides a detailed documentary analysis of Tauri resources, focusing on

understanding the framework’s architectural principles and technical design. In Chapter 5, a prototype application is implemented to facilitate an empirical evaluation of developer experience using Tauri. Finally, Chapter 6 presents the main conclusions, summarizing the challenges encountered, the advantages observed, and providing a critical assessment of the feasibility and effectiveness of Tauri for building cross-platform applications.

Although the initiative originates from personal interest and needs, it aims to benefit individuals and development teams seeking cross-platform development solutions. Furthermore, this work may inspire large tech companies, emerging startups, enterprise software providers, and open-source project managers to adopt or contribute to such technologies.

2 Research Framework

In this chapter, the research question and the research methodology of this thesis will be introduced.

2.1 Research Question

The objective of this thesis is to critically evaluate Tauri as a primary tool for cross-platform application development. To contextualize this investigation, it is essential to first review the pain point of the current development world and to answer the following research questions.

1. How has the evolution of web and mobile development shaped the current landscape of application frameworks?
2. What strengths and weaknesses characterize web development tools and native mobile development tools compared to cross-platform technologies?

Answering these questions provides a foundation for analyzing Tauri as an emerging framework. Subsequently, the thesis aims to answer the following research questions.

3. What kind of architecture enables Tauri to guarantee its security and functionality?
4. How feasible is Tauri as a cross-platform development framework in terms of compatibility, scalability, and development workflow?
5. What key advantages and challenges arise when adopting Tauri in practical application development?

2.2 Research Methodology

To evaluate Tauri's feasibility, advantages, and limitations, this thesis employed an approach that combined literature review and theoretical and practical investigation.

The literature review looks at how web and mobile development have evolved, highlighting important milestones and changes from early web technologies to today's mobile platforms. It focused on the pros and cons of popular native mobile tools, such as Kotlin and Swift, and cross-platform frameworks, such as Flutter, React Native, and Xamarin. By comparing these existing solutions, the review identified common challenges faced by developers, which helps frame the evaluation of Tauri.

The theoretical study involved a detailed document analysis of official Tauri resources, including technical documentation, GitHub repositories, architectural specifications, and release notes. This phase aimed to understand the framework's underlying architectural design, process model, and inter-process communication model that emphasizes isolation and minimal system access.

The experimental phase focused on a prototype implementation, in which a functional YLE RSS Feeds application will be developed using Tauri. This case study would serve as a proof of concept to evaluate Tauri's practical development workflow, build efficiency, runtime performance, and platform-specific behaviors. Key evaluation metrics included:

- Application size and performance optimization.
- Build and deployment efficiency.
- Developer experience.

The combination of a literature review, theoretical insights, and experimental results enabled a comprehensive evaluation of Tauri's role and viability in modern cross-platform software development. The findings highlighted

the feasibility, strengths, and limitations of adopting Tauri and propose directions for its future application in development contexts.

3 Modern Development Solution

This chapter presents an overview of the evolution of web and mobile application development, examining native development approaches, Progressive Web Apps (PWAs), and a variety of cross-platform frameworks. Emphasis is given to tools such as Flutter, React Native, and Xamarin, which facilitate efficient multi-platform deployment through shared codebases and modern development practices.

3.1 Web Development

The inception of the World Wide Web dates to 1989, initiated by Tim Berners-Lee's submission of the original Web proposal at CERN, the European Organization for Nuclear Research [3, p. 132]. This proposal introduced key technologies including Uniform Resource Locators (URLs), Hypertext Markup Language (HTML), and the Hypertext Transfer Protocol (HTTP) [3, p. 206]. URLs serve as the text-based addresses for resources on the internet. HTML became the foundational language for creating web pages, allowing developers to define their content and structure. Meanwhile, HTTP established the rules for transferring data—including web pages and images—between clients (such as web browsers) and servers. Initially, the Web was created to facilitate automated information exchange among scientists at universities and research institutions globally [5, p. 1]. While these core technologies established the essential foundation of the Web, they signify merely the beginning phase of its continuous development.

Five years after the advent of the Web, in 1994, Håkon Wium Lie—while working at CERN—conceived Cascading Style Sheets (CSS), introducing a revolutionary approach to address the limitations of HTML's styling capabilities and achieve more control over web pages by separating the content from appearance. This breakthrough established a clear separation between a webpage's visual presentation and its underlying content structure. Instead of embedding layout and styling information within HTML, CSS externalizes these

instructions into dedicated, efficient documents. This enables browsers to render websites with consistent and versatile designs while keeping the core HTML markup clean and semantically meaningful [6, p. 5].

In 1995, Netscape Communications Corporation, a leading web browser company, introduced JavaScript, a scripting language created to bring interactivity to web pages. This language, designed for ease of use, enabled developers to embed small segments of code within the web page's markup [7, p. 2]. At the time, HTML was limited to providing only a simple and static window for defining content and structure, which was inadequate for user interaction. JavaScript was introduced to add dynamic behavior by interpreting and executing these embedded code snippets during page load, thereby enabling dynamic adjustments to the page's layout and behavior in response to user actions [7, p. 2].

In 2005, Asynchronous JavaScript and XML (AJAX) achieved widespread recognition when Google utilized it in prominent applications such as Google Maps and Gmail. This innovation revitalized JavaScript's role, which had previously been limited to form-based applications, especially as Java was often chosen for complex web interactivity [7, p. 48]. Figure 1 below illustrates the distinction between the traditional web development model and the AJAX web development model.

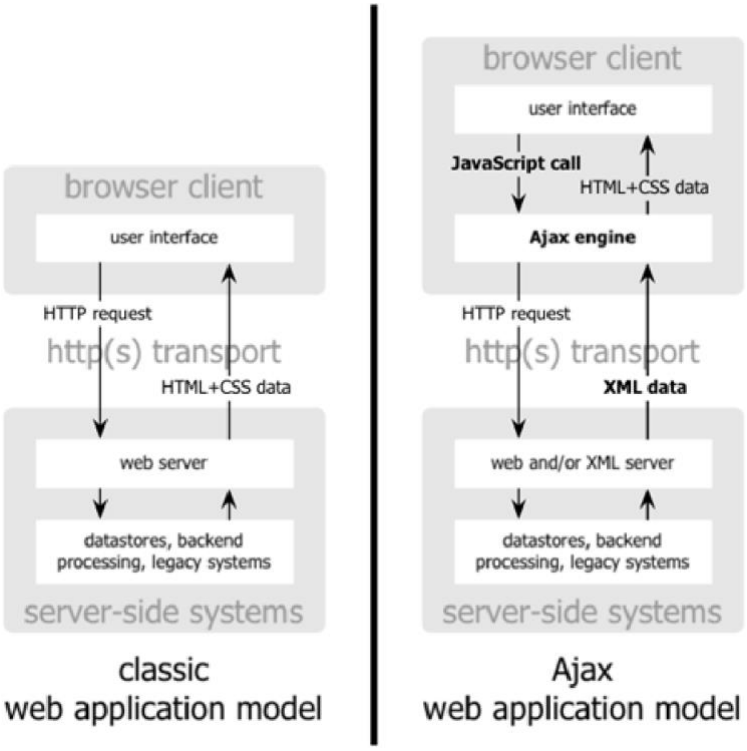


Figure 1: The traditional model for web applications (left) compared to the Ajax model (right).

As illustrated in Figure 1, acting as an intermediary layer between the browser client and the server-side system, AJAX retrieves data asynchronously using the XMLHttpRequest object, which eliminates the need for full-page reloads, saves time, and improves the overall user experience [8, p.1]. The widespread adoption of the AJAX model signaled a major shift in web development, moving beyond the Web's original function as a hypertext medium and establishing user experience as a central focus in application design.

In 2006, the emergence of JavaScript frameworks marked a pivotal shift in web development, addressing the growing complexity brought on by AJAX-style applications and browser interoperability challenges [7, p.49]. Among these frameworks, jQuery rapidly gained popularity due to its simplicity and effectiveness in abstracting browser inconsistencies and enabling Document Object Model (DOM) manipulation [9, p. 1]. Building on this foundation, numerous other JavaScript frameworks soon emerged, each offering unique

architectural patterns and capabilities to better manage increasingly complex web applications.

Building on the core client-server architecture of the World Wide Web [10, p. 1], web development has evolved to encompass client-side frameworks responsible for user interface (UI) rendering and interactivity within the browser, alongside server-side frameworks that manage backend logic and data processing on the server. Well-known client-side frameworks such as AngularJS and Vue.js provide modular and scalable structures for developing dynamic web applications. Nevertheless, React remains the most widely adopted framework, distinguished by its component-based architecture and declarative view methodology [11, p. 1]. According to a 2024 survey conducted by Statista, React maintains its position as the second most utilized framework globally, with a usage rate of 39.5% among respondents [12, p. 1].

On the server side, Node.js serves as the primary JavaScript runtime environment, leading with 40.8% usage as reported in the same survey [12, p. 1], facilitating efficient server-side programming and the development of scalable backend solutions. This robust ecosystem of client- and server-side technologies continues to empower developers to build increasingly efficient and advanced solutions within the web development domain.

Having understood this basis, the focus now shifts to mobile development, a distinct yet equally significant paradigm that responds to the need for optimized, device-specific applications on mobile platforms.

3.2 Mobile Development

Mobile development is closely tied to the evolution of mobile devices, particularly smartphones, and the underlying operating systems. The year 2007 is widely recognized as the commencement of the smartphone era, marked by the launch of Apple’s first iPhone [4, p. 14]. Apple’s innovation not only redefined mobile technology and user interfaces but also catalyzed rapid market

entry by other manufacturers [13, p. 5]. Notably, the introduction of Android-based smartphones, particularly the Samsung Galaxy series, initiated a significant second wave of growth within the industry [4, p. 14]. The distribution of mobile operating system market shares has evolved considerably from 2009 to 2025, as depicted in Figure 2.

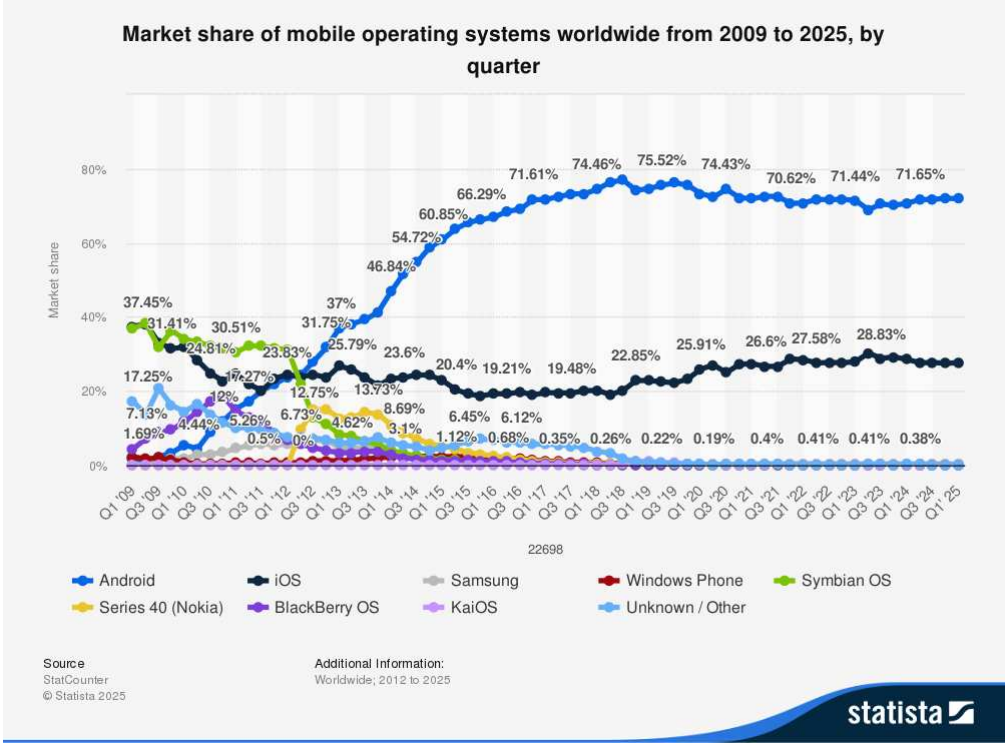


Figure 2. Market share of mobile operating systems worldwide from 2009 to 2025, by quarter

As shown in Figure 2, by the first quarter of 2025, Android had increased from around 28.01% to 71.88% worldwide, while the iPhone Operating System (iOS) had grown from 24.5% to 30.25% since 2012 [14, p. 1]. Together, Android and iOS dominate the industry and are regularly ranked as the top two mobile operating systems globally.

Consequently, mobile development has become a critical domain for developers focused on creating high-performance applications. Developing a mobile application differs substantially from creating a web application, as mobile development must consider factors such as varying screen sizes, distinct user interactions, and native device functionalities. Moreover,

developers often need to acquire new programming languages and develop separate applications for multiple operating systems, leading to increased complexity, higher costs, and extended development timelines. To address these challenges, several development approaches have emerged, including native applications, progressive web applications (PWA), and cross-platform development [15, p. 670].

3.2.1 Native Mobile App Development

The native development paradigm is defined as the process of creating applications that are tailored for individual operating systems—primarily Android and iOS— by utilizing their own programming languages and development tools [15, p. 670]. Concretely, the development tools and programming languages for each major platform are summarized in Table 1.

Table 1. Native development environment for major platform [15, p. 670]

Platform	**Native Development Tools**	**Programming Languages**
Android	Android Studio, IntelliJ IDEA	Kotlin, Java
iOS	Xcode	Swift, Objective-C

As displayed in Table 1, the development tools and programming languages for each major platform are distinct and targeted to their operating systems. Android development primarily uses Android Studio with Kotlin or Java, whereas iOS development relies on Xcode with Swift or Objective-C.

Such an approach enables developers to fully utilize the available features of each platform, including support for Bluetooth, Near Field Communication (NFC), integrated sensors, cameras, and similar native device functionalities

[16, p. 262]. This, in turn, facilitates the optimization of application performance and enhances the user experience across the various device screen sizes [15, p. 670]. A notable limitation of native app development is that each application is tied to a single platform, so expanding availability to additional platforms requires building separate codebases from the ground up. As a result, maintaining multiple codebases increases both development costs and production time. Furthermore, creating more advanced applications necessitates proficiency in platform-specific markup and programming languages [16, p. 262].

3.2.2 Progressive Web App Development

A Progressive Web Application (PWA) is a type of application delivered over the internet, created using foundational web technologies like HTML, CSS, and JavaScript to provide a native-like user experience [17, p. 270].

One of the defining features of PWAs is their install ability, allowing users to add the app directly to their device's home screen without relying on traditional app stores, which leads to superior accessibility and user engagement [17, p. 270]. PWAs also offer high reliability by leveraging service workers to enable offline functionality—critical resources are cached, allowing the app to operate smoothly even when network connectivity is poor or unavailable [18, p. 3]. Another key feature is web push notification support, keeping users informed and actively engaged with timely alerts and updates, similarly to native applications [17, p. 270].

Unlike traditional native app development, which often requires platform-specific tools and languages, PWAs can be developed using a broad range of programming technologies. The front end typically utilizes HTML, CSS, and JavaScript, while the back end is flexible and may include options such as Node.js, Python, PHP, Java, and C#, demonstrating the versatility and broad applicability of PWA architectures [18, p. 1].

Nevertheless, PWAs have several limitations. Currently, PWA technology is primarily supported by Google Chrome and compatible only with iOS devices running version 11.3 or later, excluding older Apple devices. Additionally, the Service Worker API, crucial for PWA capabilities, supports features like push notifications, camera functionality, and geo-location in Chrome; however, it does not provide access to contacts, various low-level hardware sensors, calendar entries, alarm settings, and call management features [19, p.298]. These restrictions limit the ability of PWAs to fully utilize native device features, impacting user experience and applicability in certain contexts.

3.2.3 Cross-Platform App Development

Cross-platform app development refers to the practice of building mobile or web applications that can operate on multiple operating systems—such as iOS and Android—using a single codebase. This approach allows developers to create an app once and deploy it across several platforms, significantly reducing both development time and technical costs [20, p. 1]. Cross-platform applications offer a range of benefits that make them attractive to developers and businesses alike. Because only a single codebase is maintained, updates and bug fixes can be efficiently applied across all platforms, promoting consistency and streamlining ongoing maintenance. This unified method also fosters features such as offline accessibility and supports rapid development of minimum viable products (MVPs) that users can immediately access via an app [16, p. 263].

However, despite these advantages, cross-platform solutions face certain technical limitations. These frameworks are generally less suitable for high-performance applications, such as games or programs that require advanced 3D graphics, due to the constraints of web view rendering and restricted access to certain native device features [16, p. 263]. Notable frameworks used for cross-platform development include Flutter, React Native, and Xamarin [20, p. 1], and they will be discussed in further detail in the following paragraphs.

Flutter, launched in May 2017 by Google, is an open-source software development framework for building visually attractive, natively compiled, multi-platform applications using a single codebase [21, p. 1]. It provides developers with a comprehensive SDK that includes a wide array of pre-built and customizable widgets, making UI creation both efficient and flexible [22, p. 804]. The programming language behind Flutter is Dart, which serves as the foundation for both frontend and backend development. Dart is platform-independent, object-oriented, and compatible with all major operating systems, enhancing versatility for cross-platform projects [22, p. 804].

According to the 2024 Stack Overflow Developer Survey, Flutter ranks impressively among developers, with 60.6% of developers expressing admiration and 12.4% indicating a desire to work with it, reflecting strong professional interest within the developer community [23, p. 1]. It is widely recognized that Flutter’s architecture plays a crucial role in its success, featuring a layered and modular design that enables the development of high-quality, portable mobile applications.

At its core, the Flutter engine, primarily written in C++, acts as a runtime responsible for rendering, animations, network input/output, and accessibility, providing platform-independent low-level services. The Flutter framework, developed in Dart, is built on top of this engine, comprising rich libraries and layers that manage UI components, gestures, animations, and application logic. Central to Flutter's design are widgets, which form the building blocks of the user interface. Each Flutter application is essentially a tree of widgets, with top-level widgets representing the entire app. Flutter provides platform-specific widget sets: Material widgets for Android and Cupertino widgets for iOS, allowing apps to maintain a native look and feel across platforms. This architecture, including the embedder layer that connects Flutter to the underlying OS, ensures flexibility, performance, and ease of use for developers working within the Dart ecosystem, which supports both frontend and backend application components [22, p. 806].

React Native, introduced by Facebook in 2015, is a JavaScript framework that brings the core benefits of React to native mobile development [16, p. 263]. Derived from React, it allows developers to render React components into native platform UI elements, enabling the creation of truly native applications without compromising user experience [24, p. 1]. Since it uses JavaScript, React developers can easily migrate their web applications to mobile platforms without learning new languages or frameworks. According to the 2024 Stack Overflow Developer Survey mentioned above, React Native remains highly regarded, with 56.5% of developers expressing admiration for the framework and 11.7% showing interest in adopting it, highlighting its continued appeal within the developer community [23, p. 1].

React Native offers a core set of platform-agnostic native components, such as View, Text, and Image, which correspond directly to native UI elements on each platform. It employs file-based routing to create stack, modal, drawer, and tab screens with minimal boilerplate by leveraging the filesystem. Developers have access to more than 50 modules to implement native features and extend application capabilities. The Expo Go tool streamlines development and integrates with expo-dev-client, a module that extends Expo’s functionalities for apps requiring native changes [24, p. 1].

Unlike React, which manipulates the UI via a Virtual DOM, React Native runs JavaScript in a separate background thread and communicates with native modules through asynchronous, batched bridges for direct native UI management [25, p. 392].

Xamarin, acquired by Microsoft in February 2016, is a widely used open-source cross-platform development tool that enables developers to create high-performance, native user experiences on multiple mobile platforms using familiar .NET tools, languages, and libraries [26, p. 1]. This strong integration with the .NET ecosystem allowed for significant code sharing across Android, iOS, and Windows applications, streamlining the development process [27, p. 1055]. However, as of May 2024, Microsoft officially ended support for Xamarin

and now recommends migrating to .NET Multi-platform App UI (.NET MAUI), which is the evolution of Xamarin.Forms and offers a unified framework for building applications targeting Android, iOS, macOS, and Windows [26, p. 1].

Based on the 2024 Stack Overflow Developer Survey, Xamarin has garnered admiration from 20.1% of developers and is desired by 1.8%, which likely reflects the impact of its discontinued support. In contrast, .NET MAUI remains more attractive to the developer community, with 53.1% expressing admiration and 5.5% indicating a desire to adopt the framework [23, p. 1].

In summary, cross-platform application development continues to generate significant interest within the developer community, driven by its ability to streamline the development process, reduce costs, and provide consistent user experiences across multiple operating systems.

4 Tauri 2.0

This section provides an in-depth look at Tauri, a modern cross-platform application framework launched in 2022, with particular emphasis on its development roadmap and overview, underlying architecture, key process model, and Inter-Process Communication.

4.1 Overview of Tauri

Tauri is a cross-platform development toolkit that enables developers to build applications for all major desktop systems—Windows, macOS, and Linux—as well as mobile platforms including Android and iOS, using modern web technologies. It supports the integration of any frontend framework that compiles to HTML, JavaScript, and CSS for creating user interfaces, while allowing the use of languages such as Rust, Swift, or Kotlin for backend functionality when required [28, p. 1].

The first stable release, Tauri 1.0, was introduced on June 19, 2022, emphasizing the creation of lightweight, secure, and high-performance desktop applications [1, p. 1]. Since its initial release, Tauri has undergone steady evolution marked by continual enhancements in security, performance, and developer tooling. Early updates focused on strengthening security [29, p. 1] and improving the developer experience through enhanced CLI capabilities [30, p. 1], configuration flexibility [29, p. 1], and improved build processes. Over time, significant structural changes were made to the framework's bundling and code-signing systems to increase reliability and security [31, p. 1].

With the release of Tauri 2.0 in October 2024, the framework expanded to mobile development, enabling seamless deployment of desktop applications to Android and iOS devices. This release also introduced a reworked plugin architecture, advanced Inter-Process Communication (IPC), external security audits, and live Hot-Module Replacement (HMR) for instant frontend updates [2, p. 1].

Over the course of just three years, Tauri's GitHub repository achieved significant community growth, reaching 97,000 stars and 3,100 forks by October 2025 [32, p. 1].

4.2 Tauri Architecture

Tauri is an application toolkit that allows the building of WebView OS applications, neither a virtual machine nor a lightweight kernel wrapper. Its architecture is built around a Rust-powered core process, which works alongside independent frontend processes displayed in native WebView provided by the operating system.

The Rust back end's main tasks include starting and managing application windows, handling system-level interactions, and overseeing inter-process communication (IPC), all of which help maintain security and efficiency throughout the application. The frontend user interface is constructed using standard web technologies, such as HTML, CSS, JavaScript, and any framework that compiles to them, running inside the WebView for a smooth UI experience. Figure 3 below demonstrates the Tauri architecture [33, p. 1].

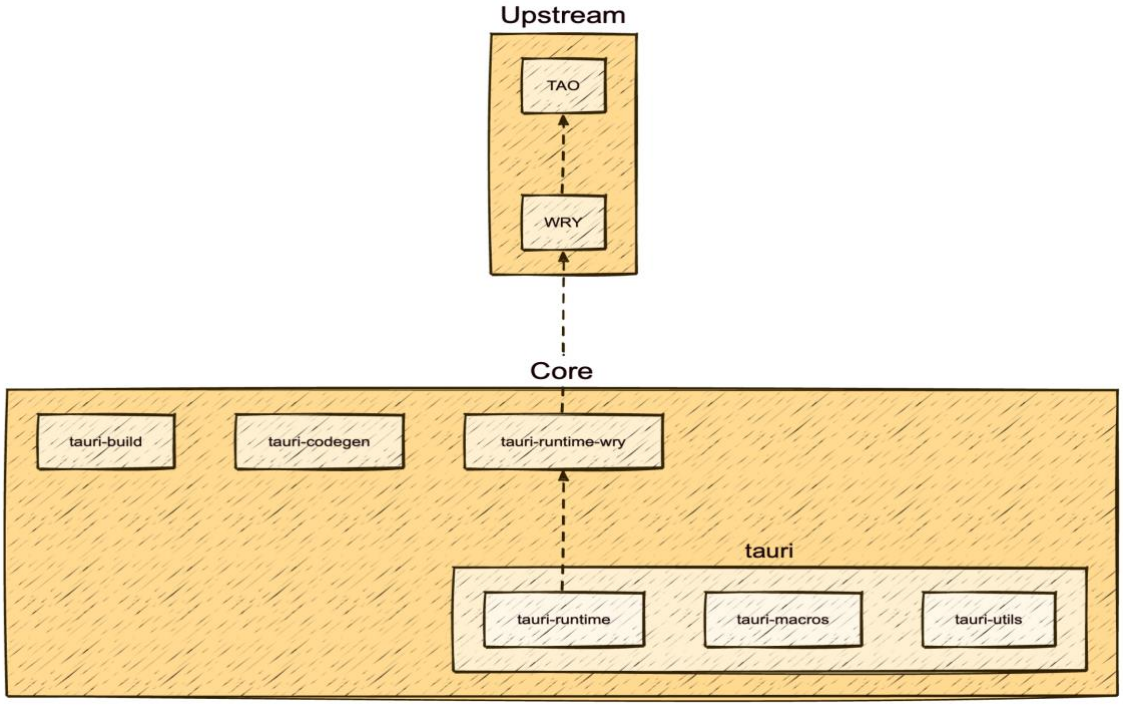


Figure 3. Tauri core architecture. [33, p. 1].

Figure 3 shows that the Tauri toolkit consists of its central core and upstream supporting services. The major crate, tauri, integrates key components such as tauri-runtime, tauri-macros, and tauri-utils. Acting as an intermediary, tauri-runtime connects Tauri with lower-level webview libraries. Tauri-macros facilitates the generation of macros for context, handlers, and commands, utilizing the tauri-codegen crate. Tauri-utils offers a suite of reusable utilities for tasks like configuration file parsing, platform detection, CSP injection, and asset management. At compile time, Tauri reads the tauri.conf.json file to configure application features and settings, while at runtime, it manages script injections, provides APIs for system interaction, and oversees update processes [33, p. 1].

In addition to the principal tauri crate, core components such as tauri-build, tauri-codegen and tauri-runtime-wry play essential roles. The tauri-build crate integrates specialized macros at build time to extend Cargo’s capabilities. It is responsible for compiling the application's configuration as specified in tauri.conf.json, creating the corresponding configuration structure, and embedding, hashing, and compressing assets like icons for both the application and system tray. The tauri-runtime-wry module enables low-level system interactions specifically for the WRY backend, providing functions for printing, monitor detection, and other window management tasks [33, p. 1].

The Tauri-Apps organization curates two fundamental upstream crates integral to Tauri’s ecosystem: TAO and WRY. TAO, a Rust-based library, provides cross-platform support for creating and managing application windows across major operating systems including Windows, macOS, Linux, iOS, and Android. It is a fork of the winit library, extended with Tauri-specific features such as menu bars and system trays. WRY, also written in Rust, is a cross-platform WebView rendering library responsible for interfacing with native WebView on desktop platforms. Within Tauri, WRY abstracts the specific WebView implementations and manages interactions, ensuring consistent cross-platform rendering and communication [33, p. 1].

4.3 Process Model

Tauri utilizes a multi-process architecture like modern web browsers and frameworks like Electron. Each application contains a core process that serves as the entry point and possesses exclusive full access to the operating system. This core process is responsible for creating and managing application windows, system tray menus, and notifications within its permissions. Additionally, all inter-process communication (IPC) is routed through this core, enabling centralized interception, filtering, and manipulation of messages. It also manages global application state, including settings and database connections, facilitating synchronized state across multiple windows while safeguarding sensitive data from exposure in the frontend. The process model is illustrated in Figure 4 [34, p. 1].

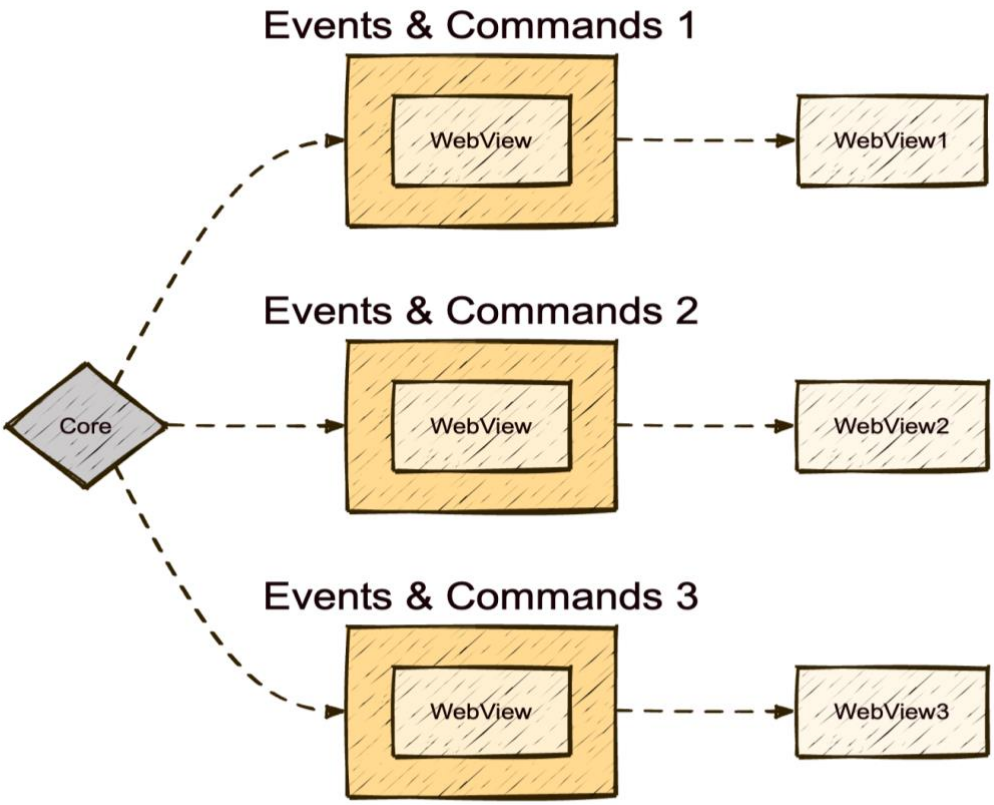


Figure 4. Tauri process model: a single Core process manages one or more WebView processes.

As shown in Figure 4, the Tauri architecture features a single Core process that manages multiple WebView processes. These WebView processes utilize the operating system's native WebView libraries to render the user interface, executing standard web technologies such as HTML, CSS, and JavaScript. This design enables developers to seamlessly migrate their existing frontend frameworks to Tauri. Unlike some other frameworks, Tauri does not bundle these WebView libraries within the final executable; instead, they are dynamically linked at runtime. This approach considerably reduces the application size but requires developers to be mindful of platform-specific behaviors, similar to traditional web development. Currently, Tauri employs Microsoft Edge WebView2 on Windows, WKWebView on macOS, and webkitgtk on Linux [34, p. 1].

4.4 Inter-Process Communication

Inter-Process Communication (IPC) enables secure interaction between separate processes, essential for building advanced applications. In Tauri, IPC utilizes asynchronous message passing, where requests and responses are serialized and transferred through a buffer, ensuring messages are processed only when sufficient resources are available. In contrast to approaches like shared memory or direct function invocation, message passing provides stronger security guarantees. The Tauri Core process can inspect and selectively reject incoming requests, thereby preventing the execution of unauthorized or potentially harmful functions [35, p. 1].

Tauri employs two primary IPC mechanisms: Events and Commands, each following specific communication patterns. The Brownfield approach allows developers to use existing browser-compatible technologies for the frontend without extra adaptation, ensuring seamless integration with standard web practices [36, p. 1]. The Isolation strategy, on the other hand, enforces strict security by filtering and validating all messages from the frontend before they reach the Tauri Core [37, p. 1].

Events operate as one-way communication channels that transfer lifecycle updates and state changes, and they can be triggered by both the frontend and the Tauri Core. Commands, by contrast, are unidirectional messages initiated by the Core backend and handled through designated functions. The central API, invoke, functions similarly to the browser's fetch API—it facilitates interaction with the Rust backend by passing parameters and receiving responses asynchronously [35, p. 1].

5 Project Implementation

This chapter presents the design and implementation of the YLE RSS Feed prototype. First, it outlines the background, requirements analysis, and feature definition, describing how user needs were translated into concrete functionalities. It then introduces the system's high-level architecture, detailing the main components, data flows, and technology choices that underpin the implementation.

5.1 Requirement Analysis and Feature Definition

Although the project was motivated by the author's personal interest, it remains essential to systematically understand user needs and goals through the elicitation and analysis of the requirements.

User story mapping was adopted as a structured technique to bridge user needs and system goals by organizing user activities, steps, and detailed stories in a visual form. This method, commonly used in agile development, supports collaborative exploration of requirements and helps align the envisioned functionality with user-oriented value. The resulting user story map is presented in Figure 5.

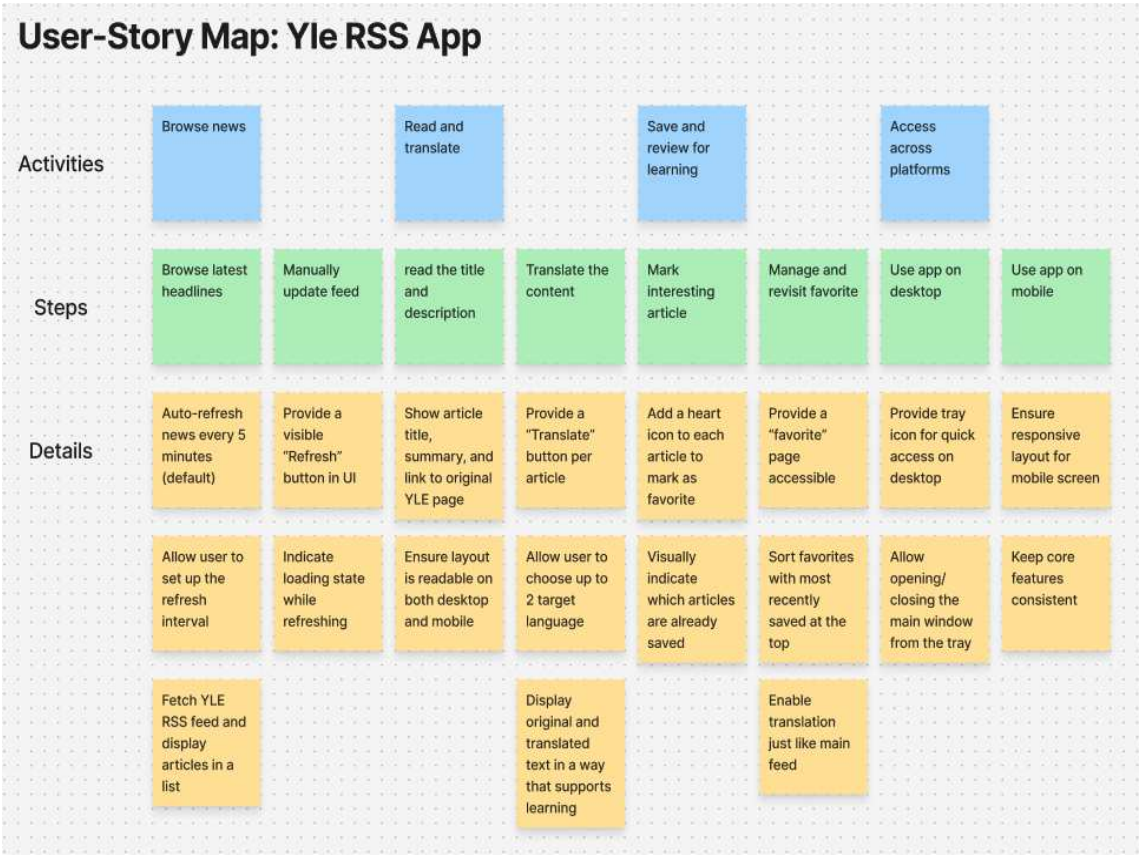


Figure 5. User story map.

As illustrated in Figure 5, the requirements are decomposed into clear, manageable features that combine user interactions with the system and the corresponding detailed objects.

5.2 Application Architecture and Implementation

The YLE RSS Feed application is implemented as a cross-platform system based on Tauri 2.0, combining a React/TypeScript frontend with a Rust backend. The architecture follows Tauri's multi-process model: a WebView-based frontend handles user interaction and presentation, while a Core process written in Rust manages data retrieval, persistence, and platform-specific integration.

5.2.1 Frontend Layer

The frontend layer of the YLE Feed application was implemented using React and TypeScript and was responsible for all user-facing functionality, including rendering the news feed, managing interaction flows, and maintaining local user interface state. It runs inside the Tauri WebView, which allows the same React codebase to be deployed across desktop and mobile platforms while relying on the operating system's native WebView engine. The corresponding frontend file structure is shown in Figure 6. This separation enables the frontend to remain focused on presentation and interaction logic, while delegating system-level operations to the Rust backend.

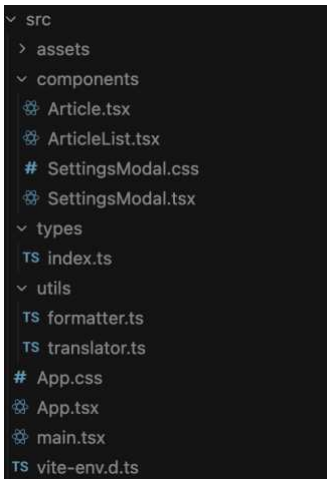


Figure 6. The frontend file structure.

As depicted in Figure 6, the central entry point of the frontend is the App.tsx component, which orchestrates the core views of the application: the main news feed in Figure 7, the saved news view in Figure 8, and the settings modal in Figure 9. It maintains the primary state of the application, including the current RSS feed data, loading and toast states for error handling, cached translations, the user's selected translation languages, the configured refresh interval, the list of saved articles, and the active view (main feed or favorites). Based on this state, App.tsx decides which child components to render and exposes callbacks for actions such as fetching the latest feed, triggering translations, and toggling saved items.

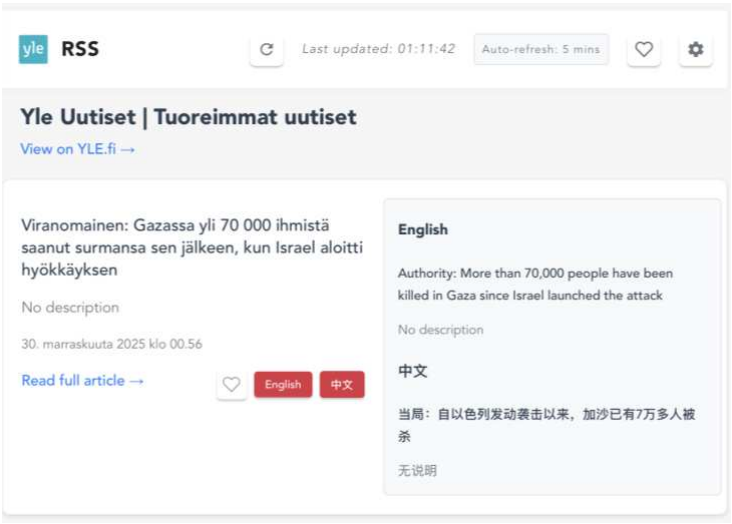


Figure 7. The main view of application allows user to browse the news and read the translation.

As depicted in Figure 7, ArticleList component is responsible for rendering collections of articles, both for the main feed and the saved-items view in Figure 8 below. It receives a list of article objects and maps them to individual Article components, using index-based keys for the main feed and stable link-based keys for saved items in Figure 8. The Article component displays the article's title, description, and metadata, and exposes controls for translating the content and saving management via a heart icon.

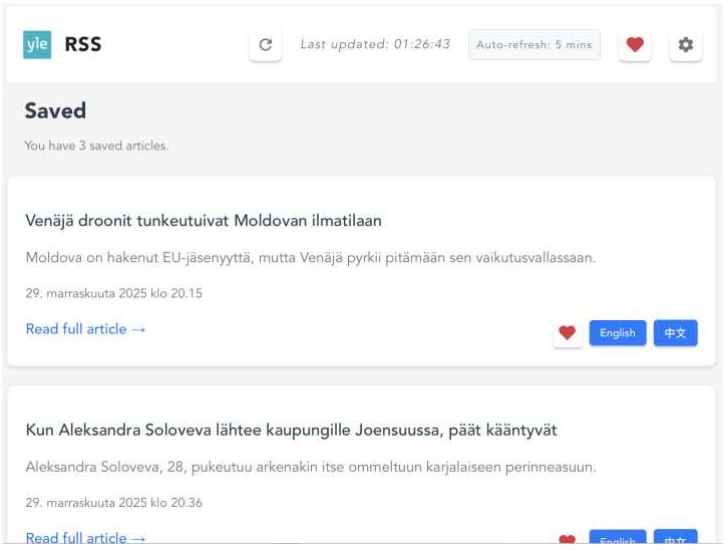


Figure 8. The saved news view allows user to review the news they saved.

As depicted in Figure 8, the saved-items view is identical to the main feed and can be accessed via the heart icon in the navigation bar.

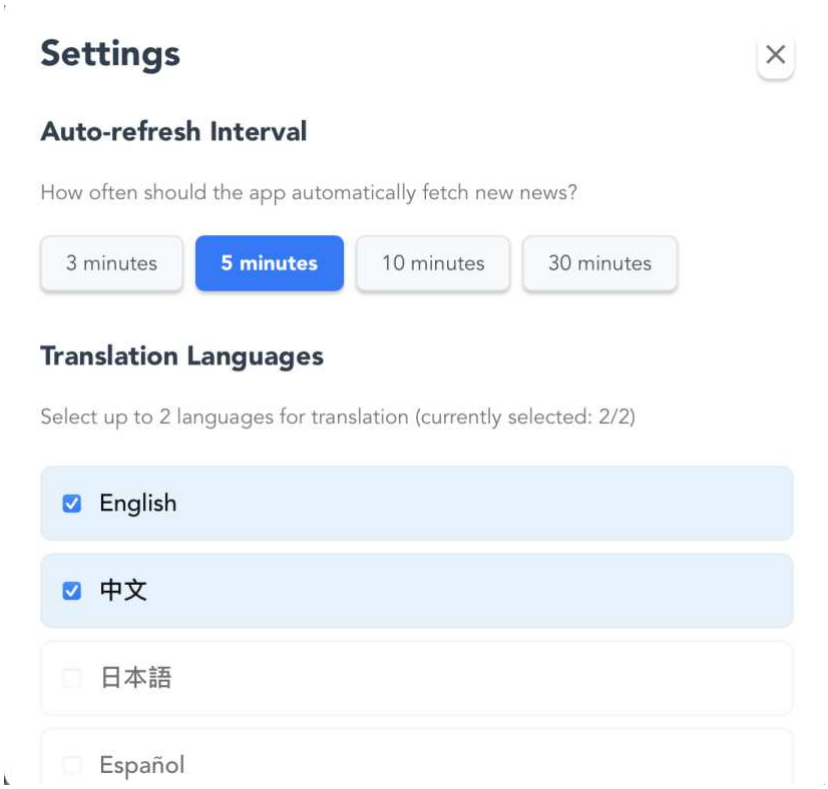


Figure 9. SettingsModal allows user to choose the auto-refresh interval and the translation language.

As depicted in Figure 9, configuration and preference management are handled by the SettingsModal. This component allows the user to select up to two translation languages from six languages and to configure the automatic refresh interval (3, 5, 10, or 30 minutes). When the user confirms the settings, the component passes the updated configuration back to App.tsx, which updates its state and persists with the values to local storage so that they are restored across application restarts.

Shared utility functionality, such as translator.ts, formatter.ts, is implemented in small modules. The translator module wraps the MyMemory translation API and provides a simple interface for translating article titles and descriptions into the user's selected languages, including basic caching of translation results in the frontend state. The formatter module centralizes date and time formatting, ensuring consistent presentation of publication dates.

5.2.2 Backend Layer

The backend layer is implemented in Rust as the Tauri Core process. It is responsible for starting the Tauri application, registering command handlers, and providing all system-level functionality that the frontend cannot access directly. This includes fetching and parsing the YLE RSS feed, managing the system tray on desktop platforms, and controlling application windows. The core backend functionality fetch_yle_feed command is shown in Figure 10.

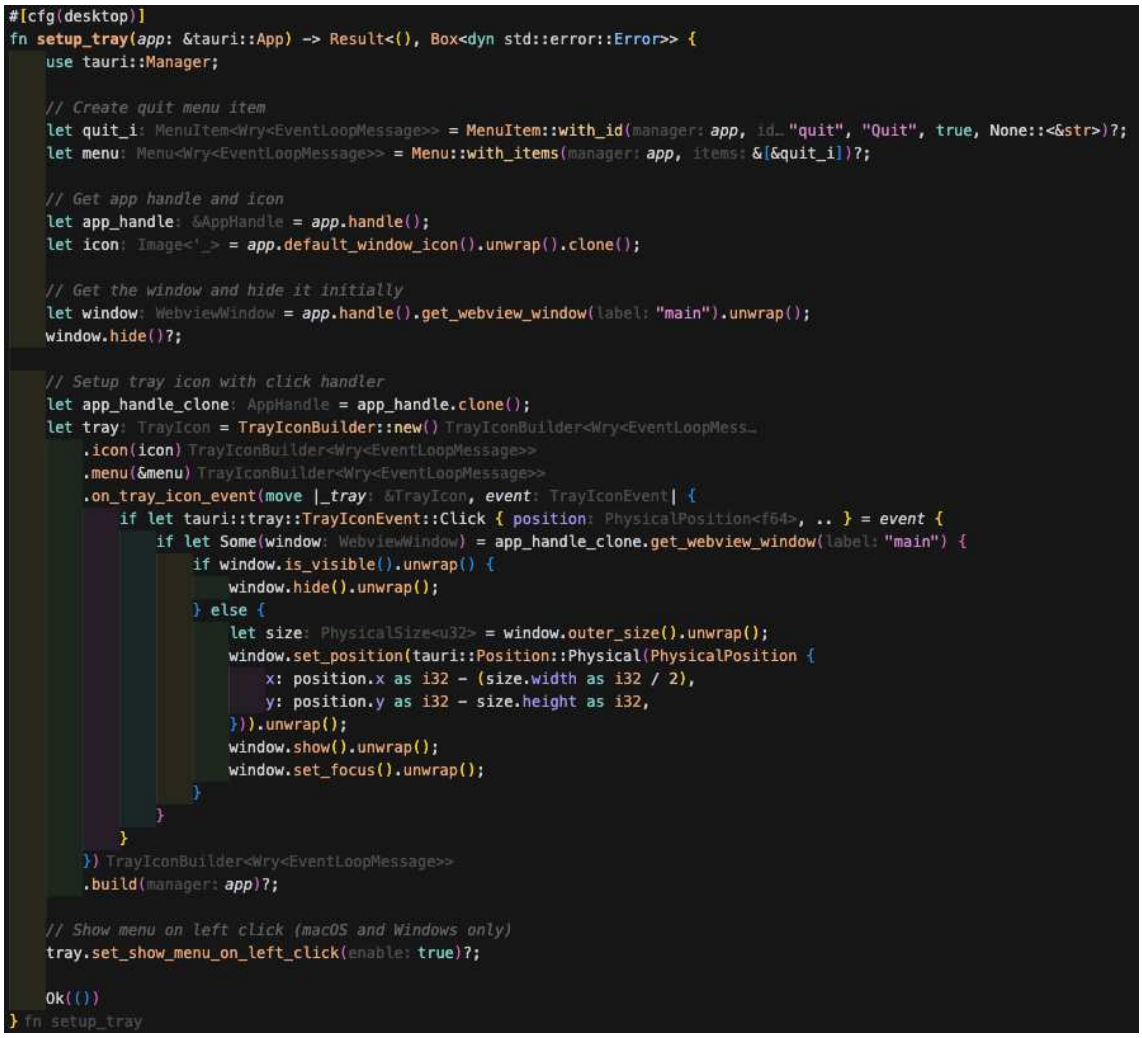
```rust
#[tauri::command]
async fn fetch_yle_feed() -> Result<RssFeed, String> {
    let client: Client = reqwest::Client::new();
    let response: Response = client Client
        .get("https://feeds.yle.fi/uutiset/v1/recent.rss?
        publisherIds=YLE_UUTISET")
        .send()
        .await Result<Response, Error>
        .map_err(|e: Error| e.to_string())?;

    if response.status() != 200 {
        return Err("Could not download YLE RSS feed".to_string());
    }

    let body: String = response.text().await.map_err(|e: Error| e.to_string
    ())?;

    // Parse RSS
    let channel: Channel = Channel::read_from(body.as_bytes()).map_err(|
    e: Error| {
        println!("RSS parsing error: {}", e);
        e.to_string()
    })?;

    let items: Vec<RssItem> = channel Channel
        .items()
        .iter()
        .map(|item: &Item| RssItem {
            title: item.title().unwrap_or("No title").to_string(),
            link: item.link().unwrap_or("").to_string(),
            description: item.description().unwrap_or("No description").
            to_string(),
            pub_date: item.pub_date().map(|d: &str| d.to_string()),
            author: item.author().map(|a: &str| a.to_string()),
        })
        .collect();

    let feed: RssFeed = RssFeed {
        title: channel.title().to_string(),
        description: channel.description().to_string(),
        link: channel.link().to_string(),
        items,
    };

    Ok(feed)
} fn fetch_yle_feed
```

Figure 10. The Rust fetch_yle_feed command to issue the YLE RSS endpoint and return the data.

As depicted in Figure 10, this command issues an HTTP request to the YLE Really Simple Syndication (RSS) endpoint, parses the XML response using the rss crate, and maps the result into custom Rust types when invoked from the

frontend. These types, such as RssFeed and RssItem, represent the feed metadata and individual news articles in a strongly typed form. The structures are annotated for serialization and deserialization, allowing them to be converted into JSON and returned to the frontend via Tauri's command system.

Platform-specific behavior is encapsulated inside the Core process to keep the frontend uniform across targets. On desktop platforms (Windows, macOS, Linux), it configures a system tray icon and manages window visibility in Figure 11.



```rust
#[cfg(desktop)]
fn setup_tray(app: &tauri::App) -> Result<(), Box<dyn std::error::Error>> {
    use tauri::Manager;

    // Create quit menu item
    let quit_i: MenuItem<Wry<EventLoopMessage>> = MenuItem::with_id(manager: app, id… "quit", "Quit", true, None::<&str>)?;
    let menu: Menu<Wry<EventLoopMessage>> = Menu::with_items(manager: app, items: &[&quit_i])?;

    // Get app handle and icon
    let app_handle: &AppHandle = app.handle();
    let icon: Image<'_> = app.default_window_icon().unwrap().clone();

    // Get the window and hide it initially
    let window: WebviewWindow = app.handle().get_webview_window(label: "main").unwrap();
    window.hide()?;

    // Setup tray icon with click handler
    let app_handle_clone: AppHandle = app_handle.clone();
    let tray: TrayIcon = TrayIconBuilder::new() TrayIconBuilder<Wry<EventLoopMess…
        .icon(icon) TrayIconBuilder<Wry<EventLoopMessage>>
        .menu(&menu) TrayIconBuilder<Wry<EventLoopMessage>>
        .on_tray_icon_event(move |_tray: &TrayIcon, event: TrayIconEvent| {
            if let tauri::tray::TrayIconEvent::Click { position: PhysicalPosition<f64>, .. } = event {
                if let Some(window: WebviewWindow) = app_handle_clone.get_webview_window(label: "main") {
                    if window.is_visible().unwrap() {
                        window.hide().unwrap();
                    } else {
                        let size: PhysicalSize<u32> = window.outer_size().unwrap();
                        window.set_position(tauri::Position::Physical(PhysicalPosition {
                            x: position.x as i32 - (size.width as i32 / 2),
                            y: position.y as i32 - size.height as i32,
                        })).unwrap();
                        window.show().unwrap();
                        window.set_focus().unwrap();
                    }
                }
            }
        }) TrayIconBuilder<Wry<EventLoopMessage>>
        .build(manager: app)?;

    // Show menu on left click (macOS and Windows only)
    tray.set_show_menu_on_left_click(enable: true)?;

    Ok(())
} fn setup_tray
```

Figure 11. The Rust command setup_tray to manage the quick access to application and window visibility.

As shown in Figure 11, the setup_tray command provides quick access to the application and command actions and manages window visibility, for example

by showing or hiding the main window in response to tray menu events, and it can integrate a native menu if required. These responsibilities are implemented using Tauri's window and tray APIs, ensuring that desktop-specific logic remains in the Rust layer and does not leak into the React codebase.

The same backend codebase is reused for all supported platforms, including iOS and Android, with conditional compilation applied only where necessary for desktop-specific features such as the tray icon. For non-desktop builds, the Core process still exposes the same commands for fetching the RSS feed and returning structured data to the WebView, but tray and menu functionality are omitted. This design allows the application to share a single Rust backend while adapting gracefully to the capabilities of each platform.

5.2.3 External Services

The system integrates two external services: the YLE RSS feed as the primary news source and the MyMemory translation API for on-demand translation shown in Figure 12.

```
/**
 * Translate text from Finnish to target language using MyMemory Translation API
 */
export const translateText = async (text: string, targetLang: Language): Promise<string> => {
  try {
    const response = await fetch(`https://api.mymemory.translated.net/get?q=${encodeURIComponent(text)}&langpair=fi|${targetLang}`);

    if (!response.ok) {
      throw new Error(`Translation failed: ${response.statusText}`);
    }

    const data = await response.json();

    if (data.responseStatus === 200) {
      return data.responseData.translatedText;
    } else {
      throw new Error('Translation service error');
    }
  } catch (error) {
    console.error('Translation error:', error);
    // Fallback: return original text if translation fails
    return text;
  }
};
```

Figure 12. The translator util function uses MyMemory API to translate the text.

As depicted in Figure 12, it receives the source text and a Language enum value and returns a Promise resolving to the translated string. User preferences

and application state that must persist across sessions—such as selected translation languages, refresh interval, and the list of saved articles—are stored in browser-based local storage on the frontend, enabling lightweight persistence without a separate backend server.

5.2.4 Data Flows

Data flows through the system in three main scenarios. For news retrieval, the frontend invokes a Tauri command, which triggers the Rust backend to fetch and parse the RSS feed before returning structured data to the WebView for rendering. For translation, the frontend sends a request to the translation utility, which calls the external translation API and returns the translated text to be displayed alongside the original content. For saved news and settings, the frontend updates local storage whenever the user modifies preferences or marks an article as a saved news and restores this state when the application starts.

5.3 Evaluation

Since the purpose of the prototype is to explore a new cross-platform development framework, the evaluation assesses Tauri across multiple dimensions. The analysis addresses cross-platform behavior on macOS, iOS, and Android, and key quantitative metrics including application size, runtime performance, build and deployment efficiency, and development challenges encountered.

In the test environment, the prototype ran successfully on macOS, iOS, and Android, and demos of the application running in the iOS and Android simulators are shown in Figure 13 and Figure 14.

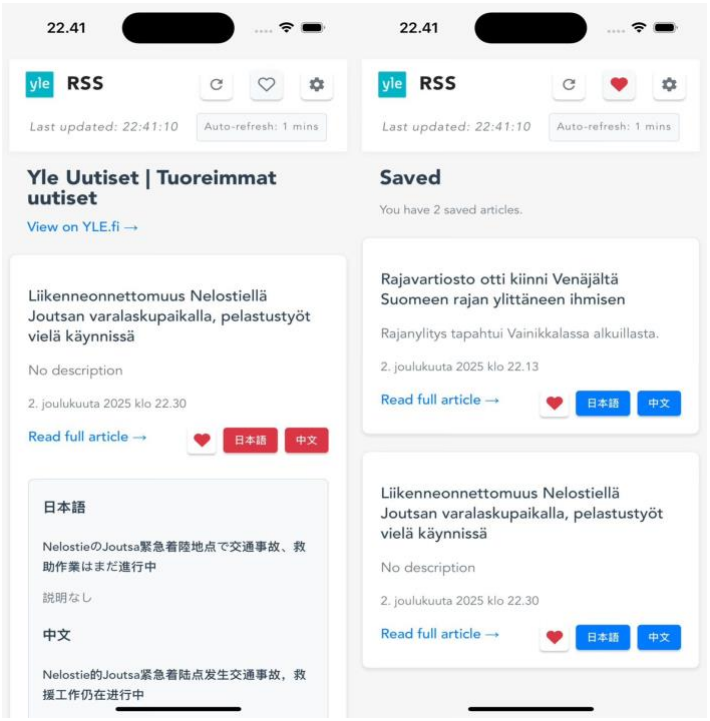


Figure 13. The demo of application running in the iOS simulator.

As depicted in Figure 13, users were able to browse news items, view translations, save articles for later reading, adjust the refresh interval, and switch between different target languages on the iOS platform.

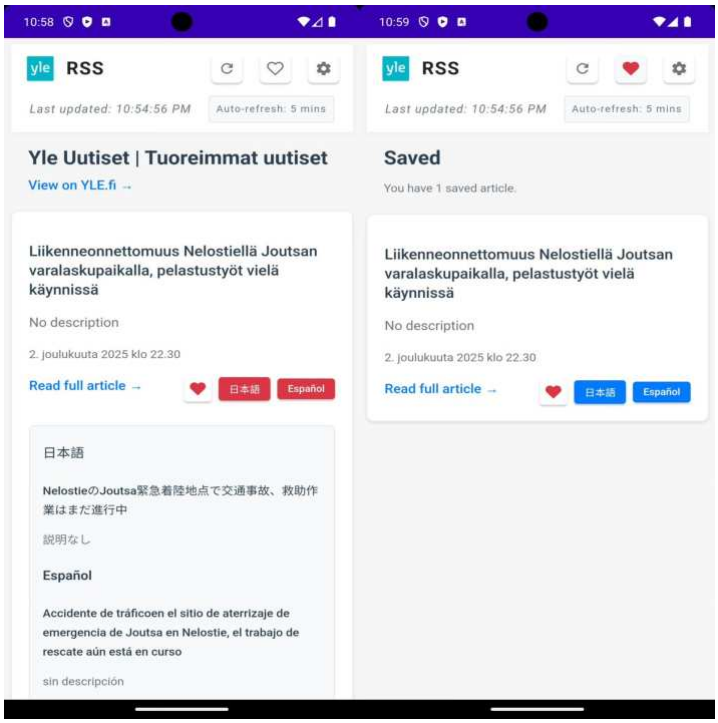


Figure 14. The demo of application running in the Android simulator.

As depicted in Figure 14, the application has the same features on the Android platform. The desktop-specific tray-icon access is only available on macOS and is intentionally omitted from the mobile builds. Because the current version does not implement authentication or cloud synchronization, saved items remain local to each device; for example, articles saved on iOS are not visible on macOS. Introducing a shared user account and synchronized storage would, therefore, be a natural direction for future enhancement.

The application intentionally uses very few external dependencies. Only a single Tauri plug-in (tauri-plugin-opener) is included, reserved for potential future use in tasks such as improved error handling, programmatic control, and opening external files. In its current form, the prototype relies primarily on Tauri's core functionality, including the system tray integration, which illustrates a lean dependency strategy that simplifies maintenance and reduces potential attack surface.

Application size and bundle composition were first assessed on macOS and iOS by inspecting the platform-specific artifacts (.app, .dmg, and .ipa). The macOS release binary is approximately 4.4 MB, resulting in an installed desktop bundle of about 4.6 MB and a compressed distribution image of 2.7 MB, which is markedly smaller than the 100–200 MB range often reported for comparable cross-platform frameworks that embed their own runtime and browser engine. Code-splitting further reduced the frontend payload to roughly 188 KB, with around 140 KB attributable to the React framework and about 20 KB to application-specific logic, thereby minimizing parsing and loading overhead in the WebView.

Runtime behavior was evaluated primarily through memory usage. During normal operation, the desktop application maintains a memory footprint of approximately 59.19 MB, which is well below the 150 MB target adopted for this study and indicates efficient resource utilisation enabled by Rust on the backend and React's component-based architecture on the frontend. On mobile, an Android debug build produced a 173 MB APK; however, this figure

reflects an unoptimized development configuration and is expected to decrease substantially in a release build with symbol stripping and size-oriented optimizations. Together, these results suggest that a single Tauri codebase can target macOS, iOS, and Android while preserving relatively small bundles and modest memory consumption.

During the implementation of the prototype, several practical challenges were encountered. Initially, difficulties arose from framework documentation and ecosystem maturity, particularly regarding desktop-specific features. Implementing the tray-icon integration required substantial effort, as existing documentation referenced outdated APIs and renamed modules following framework updates. Consequently, considerable time was spent reconciling inconsistencies between documentation, examples, and the current API, detracting from creative development. This suggests that, while Tauri offers robust technical capabilities, the accuracy and timeliness of official documentation and community support warrant improvement for developers leveraging emerging platform features.

Subsequently, the iOS build pipeline presented notable challenges. The build process failed when Xcode attempted to compile the Rust backend, and the diagnostic output available through Tauri’s toolchain was insufficient for troubleshooting. Investigation identified several root causes, including a non-operational build daemon, Node.js version conflicts, and daemon connection issues. To resolve these problems, the build script was refactored to bypass the Tauri CLI daemon entirely, invoking the Rust compiler directly via Cargo, thereby stabilizing the iOS build workflow.

6 Conclusion

The main objective of this project was to investigate the feasibility of using Tauri as a primary cross-platform development framework. Theoretical analysis combined with a working prototype indicates that Tauri can deliver lightweight, secure applications that target desktop and mobile platforms from a shared codebase. At the same time, the work highlighted practical challenges related to ecosystem maturity, particularly the quality of documentation, community support, and tooling integration, which can slow down development and debugging.

The study has several limitations. Owing to time constraints, the prototype did not systematically explore the wider ecosystem of Tauri plug-ins and thus does not fully reflect the capabilities available for features such as deep linking, clipboard access, or advanced native integration. Furthermore, the prototype has so far been evaluated only by the author, without systematic user studies, so conclusions about usability and robustness remain preliminary.

These limitations also suggest clear directions for future work. A logical next step is to extend the prototype with additional Tauri plug-ins in order to leverage richer platform features and to reassess application size, performance, and complexity under those conditions. In parallel, the application should be evaluated with a larger and more diverse group of users in realistic scenarios, both to validate its usability across platforms and to generate empirical feedback that can guide further design and implementation improvements.

Despite these constraints, the results provide a concrete case study of Tauri’s strengths and weaknesses in practice and can inform future work on building multi-platform applications with Tauri or comparing it empirically against alternative frameworks such as Electron, React Native, or Flutter.

References

1 Thompson-Yvetot D. Tauri 1.0 release [Internet]. June 19, 2022 [cited September 15, 2025]. Available from: https://v2.tauri.app/blog/tauri-1-0/

2 Nogueira L. Tauri 2.0 Stable Release [Internet]. October 2, 2024 [cited September 18, 2025]. Available from: https://v2.tauri.app/blog/tauri-20/

3 Gillies J, Cailliau R. How the Web was born: The story of the World Wide Web. Oxford: Oxford University Press; 2000.

4 Miller D, Abed Rabho L, Awondo P, de Vries M, Duque M, Garvey P, et al. The Global Smartphone: Beyond a Youth Technology. London: UCL Press; 2021.

5 Berners-Lee T. Information Management: A Proposal. CERN; 1990.

6 Lie HW, Bos B. Cascading style sheets: designing for the Web. 3rd ed. Boston: Addison-Wesley; 2005.

7 Wirfs-Brock A, Eich B. JavaScript: The First 20 Years. Proc. ACM Program. Lang. June 2020;4(HOPL):77:1–77:189. https://doi.org/10.1145/3386327.

8 Garrett JJ. Ajax: A New Approach to Web Applications [Internet]. Adaptive Path; February 2005 [cited September 1, 2025]. Available from: https://designftw.mit.edu/lectures/apis/ajax_adaptive_path.pdf

9 jQuery. What is jQuery? [Internet]. jQuery.com; October 6, 2021 [cited September 10, 2025]. Available from: https://jquery.com

10 Saternos C. Client-Server Web Apps with JavaScript and Java. Sebastopol (CA): O'Reilly Media, Inc. 2014. Chapter 1, The Nature of the Web. Available from: **https://learning.oreilly.com/library/view/client-server-web-apps/9781449369323/ch01.html

11 React – A JavaScript Library for Building User Interfaces [Internet]. Reactjs.org; [cited September 10, 2025]. Available from: https://legacy.reactjs.org

12 Stack Overflow. Most Used Web Frameworks Among Developers Worldwide, as of 2024 [Internet]. Statista; July 24, 2024 [cited Sep 15, 2025]. Available from: **https://www-statista-com.ezproxy.metropolia.fi/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/**

13 Sedov N. Global Android Smartphone Market: Strategy Recommendations for Competitors in the Market. Helsinki: Metropolia University of Applied Sciences; 2020. Available

from: **https://finna.fi/Record/theseus_lab.10024_344477?sid=5136822232**

14 StatCounter. (March 11, 2025). Market Share of Mobile Operating Systems worldwide from 2009 to 2025, by quarter [Graph]. In *Statista*. Retrieved September 15, 2025, from https://www-statista-com.ezproxy.metropolia.fi/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

15 Pinto CM, Coutinho C. From Native to Cross-platform Hybrid Development. In: Proceedings of the 2018 International Conference on Intelligent Systems (IS); November 7-9, 2018; Funchal, Portugal. New York: IEEE; 2018. p. 669-676. doi: 10.1109/IS.2018.8710545. Available from: https://dl.acm.org/doi/10.1109/IS.2018.8710545

16 Koram N, Garg R. Review on Mobile App Development: Tools and Techniques. In: 2023 IEEE World Conference on Applied Intelligence and Computing (AIC); November 06-08, 2023; Sonbhadra, India. New York (NY): IEEE; 2023. p. 260-6. doi: 10.1109/AIC57670.2023.10263908. Available from: https://ieeexplore.ieee.org/document/10263908

17 Gaikwad SM, Kulkarni KR. Data Collection System Based on PWA (Progressive Web App) as SaaS. In: 2022 5th International Conference on Advances in Science and Technology (ICAST); December 16-17, 2022; Mumbai, India. New York (NY): IEEE; 2022. p. 270-3. doi: 10.1109/ICAST55766.2022.10039543. Available from: https://ieeexplore.ieee.org/document/10039543

18 Azhar MAH Bin, Mohan JT. Progressive Web App for Real-time Doctor-patient Communication and Searchable Health Conditions. In: 2022 E-Health and Bioengineering Conference (EHB); September 22-24, 2022; Iasi, Romania. New York (NY): IEEE; 2022. p. 1-5. doi: 10.1109/EHB55594.2022.9991288. Available from: https://ieeexplore.ieee.org/document/9991288

19 Gambhir A, Raj G. Analysis of Cache in Service Worker and Performance Scoring of Progressive Web Application. In: Proceedings of the 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE); May 21-23, 2018; Paris, France. New York (NY): IEEE; 2018. p. 294-9. doi: 10.1109/ICACCE.2018.8441715. Available from: https://www.semanticscholar.org/paper/Analysis-of-Cache-in-Service-Worker-and-Performance-Gambhir-Raj/2c07d349f445e7d2ca7a3f647541e1dcd965d87a

20 Kumar N, Mali SK, Kaur P. Navigating Cross-platform App Development: Frameworks and Best Practices. In: Proceedings of the 2024 International Conference on Advances in Computing, Communication and Materials (ICACCM); April 12-15, 2024; Dehradun, India. New York (NY): IEEE; 2024. p. 1-6. doi: 10.1109/ICACCM61117.2024.11059163. Available from:

https://www.researchgate.net/publication/393335030_Navigating_Cross-Platform_App_Development_Frameworks_and_Best_Practices

21 Flutter. Flutter - Build Apps for Any Screen [Internet]. Mountain View (CA): Google; [cited September 28, 2025]. Available from: **https://flutter.dev/

22 Bhagat SA, Dudhalkar SG, Kelapure PD, Kokare AS, Bachwani SA. Review on Mobile Application Development Based on Flutter Platform. Int J Res Appl Sci Eng Technol [Internet]. January 2022;10(1):803-809. Available from: https://www.ijraset.com/best-journal/review-on-mobile-application-development-based-on-flutter-platform

23 Stack Overflow. Technology | 2024 Stack Overflow Developer Survey [Internet]. New York (NY): Stack Overflow; 2024. [cited September 28, 2025]. Available from: https://survey.stackoverflow.co/2024/technology#admired-and-desired-misc-tech-desire-admire

24 Meta. React Native · Learn Once, write Anywhere [Internet]. Menlo Park (CA): Meta Platforms, Inc.; 2015 [cited September 28, 2025]. Available from: **https://reactnative.dev/

25 Azizah AH, Faidah SZ, Ulum MB, Handayani P. Exploration of React Native Framework in Designing a Rule-based Application for Healthy Lifestyle Education. In: Proceedings of the 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI); October 14-16, 2021; Jakarta, Indonesia. New York (NY): IEEE; 2021. p. 391-394. doi: 10.1109/ICCSAI53272.2021.9609763. Available from: https://www.researchgate.net/publication/356517699_Exploration_of_React_Native_Framework_in_designing_a_Rule-Based_Application_for_healthy_lifestyle_education

26 Microsoft. Mobile development with Xamarin | .NET [Internet]. Redmond (WA): Microsoft; [cited September 29, 2025]. Available from: **https://dotnet.microsoft.com/en-us/apps/xamarin

27 Javed M, Estep M. Teaching Undergraduate Software Engineering: Xamarin Mobile App Development During the Covid-19 Pandemic. In: Proceedings of the 2021 International Conference on Computational Science and Computational Intelligence (CSCI); December 15-17, 2021; Las Vegas, NV, USA. New York (NY): IEEE; 2021. p. 1055-60. doi: 10.1109/CSCI54926.2021.00224. Available from: https://ieeexplore.ieee.org/document/9798912