Tamás Ádám

ACCESSIBLE AUTHENTICATOR APPLICATION

Investigating Accessibility Barriers and Prototyping an Inclusive MFA Solution

Technology

2026

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Tamás Ádám
Title	Accessible Authenticator Application: Investigating Accessibility Barriers and Prototyping an Inclusive MFA Solution
Year	2026
Language	English
Pages	83
Name of Supervisor	Harri Lehtinen

This thesis examines authenticator applications through the lens of accessibility and proposes a solution to enhance their accessibility, thereby increasing inclusivity.

The theoretical section first explores the concept of authenticator applications, the reasons for their adoption, and how they compare with more traditional solutions (such as SMS or email) in terms of security. It then analyzes the challenges that current authenticator application implementations pose to end users (visual ability and device compatibility) and how they inherently exclude large groups of people.

The implementation section focuses on developing a prototype of an authenticator application that promotes inclusivity while meeting current security requirements. By exploring tools and frameworks that ensure cross-platform compatibility, robust encryption, and accessible design, the thesis demonstrates that authentication systems can be both secure and equitable.

The subsequent evaluation assesses the prototype's usability, accessibility, and cross-platform performance, measuring its success against the stated goals of security and inclusivity. Ultimately, the work argues that accessibility should be treated as a fundamental pillar of security, rather than an afterthought in authenticator app development.

Keywords accessibility, usability, MFA, 2fA, authenticator, application, OTP, TOTP, HOTP, security

CONTENTS

FIGURES

TABLES

ABBREVIATIONS

OTP	One-time password
TOTP	Time-based one-time password
HOTP	HMAC-based one-time password
HMAC	Hash-based message authentication
2FA	Two-factor authentication
MFA	Multi-factor authentication

QR code	Quick response code
API	Application programming interface
CLI	Command-line interface
SQL	Structured Query Language
SVG	Scalable Vector Graphics
JSON	JavaScript Object Notation
UUID	Universally Unique Identifier
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
URI	Uniform Resource Identifier

1 INTRODUCTION

As digital technology seeps into more aspects of everyday life, security becomes increasingly important. Users of digital systems have relied on password authentication from the beginning: to register an account, provide an email address and a password, and then log in with the same email address and password. However, as technology has evolved, this tried-and-tested authentication method has become increasingly vulnerable due to bad password management practices both by users (such as picking inadequate passwords and reusing passwords) and businesses (storing them in an unencrypted, raw form, for instance), social engineering, and man-in-the-middle attacks, to name a few.

To enhance security, service providers have begun implementing various two-factor authentication (2FA) solutions. After the user enters the correct password, a secret code is sent to the user, typically via SMS or email. This way, the user must now complete two separate forms of authentication (hence the name two-factor authentication) before being logged in; however, they still only need to remember one code - their password. While such approaches are considered an improvement in security over regular password-based authentication, they have proven vulnerable to a range of attacks, as explored in the Theoretical Background chapter.

Authenticator applications have emerged as an alternative to other 2FA methods and have quickly gained traction among service providers, addressing the vulnerabilities of email- and SMS-based solutions. However, this comes at a cost: users must complete additional steps, which they may or may not be able to perform. Furthermore, these applications often make assumptions about their end users, such as having a compatible device (typically a smartphone), reading abilities, and being adept at using and understanding contemporary technology. These assumptions exclude large—and typically vulnerable—groups of

people (e.g., older adults), who may need the added layer of security the most.

This thesis aims to develop an inclusive application that does not compromise security. The result should be a cross-platform-compatible, easy-to-use, and understandable application (even for non-technical users) that securely stores sensitive information.

2 THEORETICAL BACKGROUND

Users of digital systems have relied on passwords for authentication since the 1960s. Just a few years later, there was already an incident in which a user gained access to other users' passwords stored in a system's database. To prevent such incidents, passwords were hashed (one-way encrypted) so that, in the event of unauthorized access, attackers would obtain only the hashes, not the unencrypted passwords. Since then, multiple attempts have been made to enhance the traditional username/email and password authentication system, such as by requiring longer, more complex passwords and storing them in more secure formats (Kaczorowski, 2025). However, the Cybersecurity and Infrastructure Security Agency (CISA, n.d.) reports that the most commonly selected password in the United States is still 123456. That is considered an insufficient password because it is too easy to guess, thereby reducing its protection. However, even strong passwords can be compromised if attackers have sufficient time. Relying on a single authentication factor introduces a single point of failure and has long been considered inadequate by security experts. Instead, it is recommended to combine multiple authentication factors for increased security (NIST, n.d.-a).

This chapter discusses what MFA is, the basic principles of authenticator applications, including how they operate, and potential accessibility improvements. Then, a comparison is made between them and alternative MFA solutions, examining accessibility, security, and usability. Finally, various accessibility barriers are discussed; this thesis addresses most of these questions in the Implementation chapter.

2.1 What is Multi-factor Authentication

Multi-factor authentication (MFA) is the practice of combining multiple authentication factors to achieve stronger security. According to

Cloudflare (n.d.), authentication factors may be sorted into the following categories:

- Something the user knows, such as a password
- Something the user has, such as a mobile phone with an authenticator application, or a hardware key
- Some unique traits of the user, such as their fingerprint
- The user's location

The most common form of multi-factor authentication is two-factor authentication (2FA), in which two authentication factors are combined; however, additional factors can be layered on top of each other (Fruhlinger & CSO Staff, 2024). Using multiple authentication factors provides stronger security because the failure of a single factor does not enable unauthorized access. For instance, in the case of an authenticator application, having access to someone else's password is not sufficient; to access their account, the device configured with the authenticator application must also be present to complete the authentication process (NIST, n.d.-b).

MFA has existed for more than 20 years. Although the exact date of origin remains debated, MFA has gained popularity only since the mid-2000s. Before that, security-conscious organizations began employing various systems and methods to enhance security, but many considered them unnecessary, overly complex, and too costly to implement. However, in the mid-2000s, smartphones became increasingly popular, making more user-friendly authentication methods, such as SMS and email codes, available to many users. Still, large-scale data breaches affecting major organizations (such as Sony Pictures Entertainment and the U.S. Office of Personnel Management and Budget) in the early 2010s showed that 2FA adoption was not as widespread and that businesses were not doing enough to protect their customers (de Fremery, 2021). It was around this time that more robust MFA methods emerged: Google Authenticator (Google's authenticator app) was released in 2010, and

push-notification- and hardware-security-key-based MFA also appeared in the mid-2010s (Kaczorowski, 2025).

2.2 Principles of Authenticator Applications

Authenticator applications are software tools that serve as authentication factors and belong to the "things a user has" category. Most are available only for smartphones, but desktop-compatible versions are also available. They work by generating a short-lived one-time password (OTP) that the user can use to authenticate. Such codes are generated based on a secret shared between the authenticator application and the website. The typical authentication flow in an authenticator application consists of the user entering their password, then a valid OTP, and finally completing the authentication. The strength of authenticator applications lies in generating OTPs locally on the user's device, eliminating the need to send them over the internet and reducing the risk of theft by malicious actors. Figure 1 shows the authentication flow using authenticator applications.

Figure 1. Authentication flow using authenticator applications

Users must first link their authenticator application to all accounts they intend to use for MFA. The linking process enables the website and the application to share a secret, ensuring that both parties can independently generate the same OTPs, which prevents the website from having to send codes to the user subsequently. Upon signing in, the user pastes the current OTP from the authenticator app into the website, which then verifies its validity by generating an OTP using the same secret; if the two match, the user-provided code is valid.

Depending on the device on which the application is installed, this linking process either involves scanning a QR code containing the secret (on smartphones) or entering the secret into a text field presented by the application (on desktop PCs) (Trevino, 2025). In both cases, the user then usually needs to enter an OTP into the website. This is done so that the website can verify that the authenticator application is generating valid codes and refuse to link if it is not. Skipping this step could result in users being locked out of their accounts because the server and the application generate different OTPs.

Users can link multiple accounts to the same authenticator application; a list of linked accounts is displayed when the application is opened. For each account, the currently active OTP (and sometimes the next one) is displayed, along with a visual indicator indicating the code's expiry. To complete authentication on a website, the user first enters their password and, when prompted, the current OTP displayed by the application. As previously mentioned, for TOTP, it is recommended to accept a limited number of expired codes; however, these codes are not typically displayed in authenticator applications. Figure 2 shows the main screen of the Proton Authenticator application with a Cloudflare account and a GitHub account linked to it. For each linked account, the website icon and name, the email address (redacted in this picture), the current and next OTPs, and the expiry are displayed to the user.

Figure 2. Proton Authenticator with linked accounts

HMAC-based one-time password (HOTP) is a widely used algorithm for generating OTPs. The two key components of HOTP are a secret, available only to the website that provides it and the application that receives it upon linking, and a counter. This counter is then increased each time a new code is generated. This is referred to as the event-based moving factor (M'Raihi et al., 2005), and it ensures the uniqueness of the generated codes.

Time-based one-time password (TOTP) is another algorithm commonly used by authenticator applications to generate OTPs. It is an extension of the HOTP algorithm that replaces the event-based moving factor with a time-based one, using the current system time and the number of time steps to represent the code's lifetime and to compute OTPs. Because TOTP is a time-based algorithm, it is recommended to account for clock drift by accepting codes that are a few steps out of sync (M'Raihi et al., 2011).

Both algorithms are open standards, meaning their specifications, published by the Internet Engineering Task Force, are publicly available. Both are also based on well-known cryptographic primitives, such as HMAC (for HOTP) or SHA-1. Because both are publicly available, they are free for anyone to use or implement without licensing restrictions or additional costs, meaning any party can create a spec-compliant authenticator application at no cost. TOTP is preferred over HOTP in authenticator applications for practical reasons. If a HOTP authenticator application keeps generating new codes to simulate the short-lived nature of TOTP codes without the user ever pasting them to the server, the application's and the server's counters will eventually go out of sync. This means that a complicated synchronizing mechanism would need to be implemented.

From the server's perspective, authenticator applications are not saved in any way. When the user enables MFA (and selects the appropriate options, "authenticator application" or "TOTP" on most websites), a code

is generated and presented to the user, either embedded in a QR code or as text, as mentioned previously. The only thing the server needs to remember is the secret, and, depending on the server's implementation, the MFA method to use with that secret. This means that the server is not dependent on a specific brand of authenticator application (unless such a restriction is artificially introduced, for example, by requiring a non-standard linking process). This means that users are free to use any authenticator application with any website, so long as the application adheres to appropriate standards. Then, when the user logs in, the server needs to prompt the user for their password and, afterward, for the currently valid OTP code. The server can then verify it by generating a code (or a few codes to account for clock drift), and if the user's code matches, authentication succeeds.

2.3 Comparison with Alternatives

There are various MFA options with varying levels of security and usability. The most common methods, in addition to authenticator applications, include TOTP delivered via email or SMS, MFA via push notifications, and MFA via hardware tokens. This chapter briefly analyzes the aforementioned MFA methods, outlining their strengths and weaknesses over authenticator applications.

2.3.1 Email and SMS

As smartphones became more widespread in the mid-2000s, SMS- and email-based 2FA solutions saw a surge in popularity – having a device that could receive an email or an SMS within seconds proved convenient enough for users to adopt these technologies (de Fremery, 2021). Such options are easy to set up – after the user registers on a website, they usually need to enable it there; for SMS 2FA, they must enter their phone number. Once the setup is complete, each time they log in, after entering their password, they will receive an OTP via email or SMS, depending on which is enabled on the website. They must paste this

code into the appropriate field; the website will verify it and complete the sign-in process, assuming the code is valid. Although these solutions offer greater protection than relying on a single password, both are vulnerable to various attacks.

One of the most common attacks against SMS-based 2FA is SIM swapping. It is a technique used by malicious actors to take over a user's phone number (Cuprik, 2025). There are two primary forms of this attack: either the physical SIM card is stolen from the user, or the attacker uses social engineering to convince the mobile provider to port the user's number to a new SIM card owned by the attacker. The latter approach requires the actor to know the user's phone number beforehand. Regardless of the approach, the outcome is the same: one layer of security has been removed, typically leaving only a single password to protect the user's account and identity.

A common attack against email-based 2FA is phishing. Phishing is an attack in which malicious actors impersonate a trusted company that serves a large number of users to steal their data (Goretsky, 2025). Such attacks usually begin with the user receiving an email containing a link that appears to come from a trusted company, for instance, advising them to update their password. However, the website this link points to may appear identical to the company's official site; it is a spoof designed to steal user data (FBI, n.d.). Phishing can be used to compromise email OTPs by gaining access to the user's email account through a spoofed website. Thus, attackers will have access to OTPs sent via email, thereby eliminating that authentication factor.

Attacks against SMS- and email-based 2FA methods exploit the fact that the OTP used for authentication is not stored on or generated by a user's device. Instead, it should be sent to the user over a network, thereby allowing cybercriminals to steal it. The main strength of authentication applications is that OTPs are generated on the user's device, making them nearly impossible to steal.

2.3.2 Push Notifications

Push notifications are notifications – usually alerts - sent to the user's browser or, more commonly, an application that is installed on their mobile device. They may contain text, emojis, and rich media, depending on their type and the target device. In most cases, a push server delivers such notifications (IBM, n.d.). From an MFA perspective, push notifications are considered an authentication factor.

There are multiple ways to implement MFA via push notifications. A good example of this approach is Google Prompt. It is integrated into Google's account system and, consequently, into Android; however, it is not limited to Android devices. Users can configure it to receive a sign-in authorization push notification when signing in with Google accounts on the device. This method requires minimal user action; users are presented with Yes and No buttons and contextual information, based on which they can accept or deny the sign-in request (Google, n.d.). Figure 3 shows an example sign-in confirmation request from Google Prompt on an Android device (email address and location have been redacted).

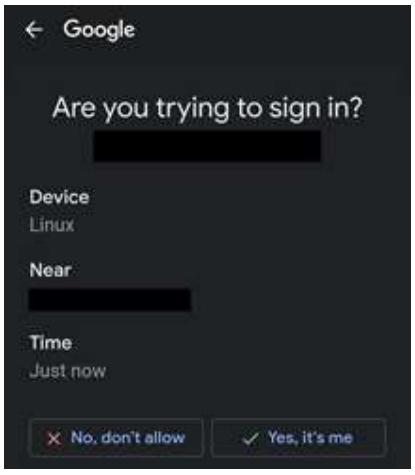


Figure 3. Google Prompt confirmation request

Another popular method uses the number-matching pattern. When the user taps a push notification to review a sign-in request, they are prompted to enter a number displayed in the browser (Microsoft, n.d.). GitHub's GitHub Mobile application (GitHub, n.d.) and Microsoft's Authenticator are good examples of a number-matching pattern implementation. Figure 4 shows an example of a sign-in confirmation request using GitHub's number-matching pattern implementation.

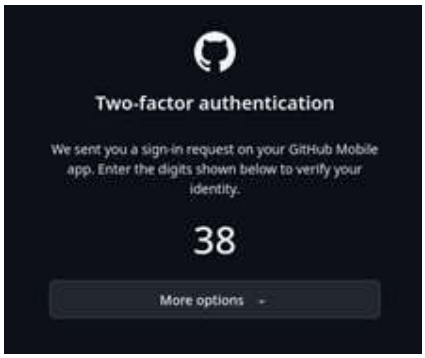


Figure 4. GitHub confirmation request

The main issue with push notification-based MFA is that the push server and the application that allows the user to authorize sign-in requests must be connected – after all, the push server needs to know which device to send the push notification to. This means users would have to download and install multiple applications for the same purpose, making them increasingly difficult to manage and quickly filling storage space. However, that is not the only problem. Because the application and push server must be connected, the device with the application installed must be connected to the internet to receive push notifications.

2.3.3 Hardware Security Keys

A hardware security key is a small physical device that contains a secure chip used for authentication. This chip employs protocols such as FIDO2 to generate reliable, secure tokens that verify user identity. Because the

user possesses the physical hardware used for authentication, this method is considered one of the strongest (D'Andrea, 2025).

Hardware security keys provide a seamless way to authenticate. The user does not need to enter an OTP; they only need to connect their security key to the device they are authenticating with when prompted. Depending on the physical device, this connection is established via USB, NFC, or Bluetooth (Cybersecurity411, 2025).

The strengths and weaknesses of hardware keys lie in their status as a separate physical device. Because they are separate devices, they are immune to man-in-the-middle attacks, phishing, or SIM-swapping – previously discussed MFA methods were vulnerable to such techniques. However, this also means users must always carry an additional device for authentication. The loss of such a device may result in the complete loss of access to accounts if proper backup methods were not configured. Lastly, the user must purchase such a device, making hardware keys a paid form of authentication (Kim, 2025).

2.4 Usability and Accessibility

The World Health Organization (2011) reports that more than 1 billion adults live with some form of disability, 185 million of whom live with severe disability – a statistic that underlines the importance of authentication usability and accessibility. The study argues that disability is a human-rights concern since people with disabilities "experience inequalities", "are subject to violations of dignity", and "are denied autonomy". Article 21 of the Convention on the Rights of Persons with Disabilities (CRPD) (United Nations, 2006) requires States Parties to ensure freedom of expression and access to information for persons with disabilities. While it does not explicitly address authentication in digital systems, an inaccessible authentication system can prevent users with disabilities from effectively using these systems, potentially denying them free access to information.

According to the International Organization for Standardization (2018), usability is defined as the "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". In digital systems, this goal typically refers to a task users intend to perform; however, if that task requires authentication, authentication itself can impede execution. For this reason, it is crucial to evaluate authentication systems from a usability perspective, as poor usability can lead users to develop coping mechanisms that ultimately undermine security (Temoshok et al., 2025). Such coping mechanisms include using weak passwords, reusing passwords across multiple platforms and accounts, and failing to use MFA.

Accessibility in digital systems refers to system design that ensures people with disabilities, including auditory, cognitive, neurological, physical, speech, and visual impairments, can use them. However, it is not only people with disabilities who benefit from accessibility – elderly people, or people with "temporary disabilities" such as a broken arm, are also positively affected by it (W3C Web Accessibility Initiative, 2026). The current standard for measuring the accessibility of digital systems is the Web Content Accessibility Guidelines (WCAG) 2.2, which addresses the aforementioned disabilities. However, not all needs of affected people can be addressed (World Wide Web Consortium, 2024). WCAG defines three conformance levels, ranging from A (lowest) to AAA (highest); legislation typically requires AA conformance (Leahy & Olivera, 2024). While WCAG primarily defines guidelines for web content accessibility, some of these guidelines may also be applied in non-web contexts (World Wide Web Consortium, 2025).

WCAG 2.2 defines four principles of accessibility:

- Perceivable: the presented information needs to be perceivable by at least some of the user's senses.
- Operable: the interface needs to be operable by the user

- Understandable: the user needs to be able to understand the presented information and interface operation
- Robust: the information needs to be robust enough so that a wide range of user agents can interpret it reliably

If any of the four principles are not met, the content is not accessible. A set of guidelines is provided for each of the 13 principles to present information in ways that meet the needs of as many people as possible. Success criteria are provided for each guideline describing what is required to conform to it. A success criterion is a statement that can be either true or false when content is checked against it. Some of these criteria can be tested by automated software; however, others require manual testing (W3C Web Accessibility Initiative, 2025, July 15).

The quick reference for meeting WCAG 2.2 lists all 13 guidelines. The first principle (perceivable) has 4 supporting guidelines, which require text alternatives to non-text content, alternatives for time-based media, adaptability of the presented content without it losing structure (for instance, using a simpler layout), and making it easier for users to see the presented content (by better separating foreground from background, for instance). The second principle (operable) has five supporting guidelines that require all functionality to be available from the keyboard, providing users sufficient time to read and use content, safe design of content (designs that are known to cause seizures and physical reactions are prohibited), providing users various ways to navigate, locate where they are, and find content, and by providing additional ways to operate the website besides keyboard input. The third principle (understandable) has 3 supporting guidelines: readable, understandable text content; a predictable website appearance and operation; and help for users to correct and avoid mistakes. Finally, the last principle (robust) has 1 supporting guideline that requires maximizing compatibility with current and future user agents and assistive technologies (W3C Web Accessibility Initiative, 2025, September 22).

2.4.1 Barriers to Usability and Accessibility

The usability and accessibility of a particular MFA method depend largely on the method itself and on its implementation. For instance, screen reader support in an authenticator application increases both. In most cases, a context switch occurs during authentication: first, the user enters their password; then, they complete the second factor, which often involves retrieving an OTP from another application, such as an authenticator app, an email account, or SMS. Switching between applications may distract users with various neurological conditions (Leahy & Olivera, 2024).

Authenticator applications make several assumptions about their potential end users. First, end users are expected to have a reasonable understanding of how MFA and, in particular, authenticator applications work – mainly to be aware of the short-lived nature of the generated OTPs. Second, the most popular authenticator applications are available only for smartphones, excluding individuals who do not possess such devices. Another barrier they pose is the time limit – a core aspect of how authenticator applications operate; users may find it challenging to copy and paste the OTP into the website within a short timeframe (Di Campi & Luccio, 2025; Kumar, 2024).

A usability study by Acemyan et al. (2018) evaluated four 2FA products provided by Google. It was found that although 2FA provides security benefits, the setup and authentication process remain challenging for users. In the study, users first needed to set up 2FA by downloading and installing the required software and linking it to a Google account provided by the study. The next step was performing an actual authentication. The mean percentage of users who were able to set up the inspected 2FA methods was 68%, ranging from 55% (Google Prompt/push notification) to 75% (SMS and Google Authenticator). However, the analysis found that the Google products offered better usability during authentication, with an average of 95% of users

successfully authenticating, ranging from 85% (Google Authenticator) to 100% (USB security key and Google Prompt/push notification). The study concluded that usability – especially during setup – needs further improvement to achieve greater adoption of 2FA, since overly complex setup or authentication processes may discourage users from adopting it, despite its security benefits.

An online survey conducted by VocalEyes (n.d.) with blind and partially sighted participants on the accessibility of quick response (QR) codes and short-number SMS found that 66% of participants could not see QR codes at all. The overall net "ease of use" score for QR codes was -42%, with 60% of participants reporting a negative-to-neutral experience when using them independently, and many commenting negatively on the scanning process. The U.S. General Services Administration (2024) highlights similar problems: blind or partially sighted people may find locating or scanning them challenging, people with mobility impairments may struggle with holding the mobile device steady while scanning a QR, and people with cognitive disabilities might struggle to understand the usage of QR codes and interpret next steps. Suggested mitigations include providing appropriate alt text and offering alternatives when possible.

Moreno et al. (2023) conducted an online survey on the accessibility of authentication for blind and partially sighted people in the Spanish-speaking world. 34 people participated, 13 women and 21 men. Participants were presented with a questionnaire with three categories: passwords, other authentication systems (including 2FA), and demographic. A total of 31 participants (91.2%) had previously attempted to use 2FA, but only 52.9% were successful. Common issues included timeouts (likely due to the short-lived nature of OTPs), QR code scanning, and managing multiple devices when the authentication process was initiated on a device other than the one generating the OTP. The study concludes that most of the reported problems would have been mitigated if WCAG guidelines had been followed appropriately. It

also notes that more than half of blind participants successfully used 2FA, but only for certain accessible products on the market.

In conclusion, the usability and accessibility of MFA and authenticator applications, in particular, need improvement. The most frequently reported issues concern QR code scanning, short-lived OTPs, and switching between devices. The implementation chapter of this thesis proposes solutions to these problems.

3 SOFTWARE DEVELOPMENT BACKGROUND

The software development background chapter provides the necessary theoretical background for the subsequent implementation chapter. The objectives chapter discusses the findings from the barriers-to-usability-and-accessibility chapter and proposes solutions to these barriers. The selection of tools and frameworks chapter then describes how to select the most suitable tools, frameworks, and libraries to meet the requirements outlined in the objectives chapter.

3.1 Objectives

Chapter 2.4.1 of this thesis found three major concerns when developing an authenticator application. The objectives chapter proposes solutions to all of them. These issues are as follows:

- Short-lived OTPs
- Scanning of QR codes
- Switching between devices

The short-lived nature of OTPs has been reported as a usability concern in authenticator applications. Because they are a core component of authenticator applications, their short-lived nature must be preserved. However, this issue could be addressed by displaying at least one upcoming code, thereby doubling its lifetime and making application use with a screen reader more seamless.

Using QR codes is often seamless for most people, as it is far easier than typing a website's secret URL into a text field. For that reason, QR code scanning for linking needs to be retained; however, a screen that lets users select their preferred linking method needs to be implemented. That way, users who prefer QR codes can still use them, while those who have trouble scanning them can enter the secret directly.

The third major problem was the need to switch devices when authenticating with an authenticator application, assuming the application runs on a separate device from the one the user is currently using. One solution to this would be to develop separate versions of the application for different platforms – a benefit of this would be that each version can be specifically optimized for its platform. Because this approach involves increased development time, and performance benefits would be negligible as authenticator applications are not resource-intensive, a single application will be developed using a framework that enables cross-platform compatibility. Target operating systems include major desktop platforms such as Windows and macOS, as well as the two major mobile operating systems, Android and iOS. Chapter 3.2 discusses frameworks that enable the development of such cross-platform applications.

Another important consideration is how adept the application's users are with such technologies, or with modern applications in general. A common practice is to replace button labels with well-understood icons; for instance, a "Settings" button is often replaced with a simple gear icon. Although the intention might be to offer a cleaner UI by reducing the amount of text presented to the user, older adults may struggle to understand the meaning of UI elements that are not clearly labeled. For this reason, the application should use clear labels instead of icons, even if it results in a less clean-looking interface.

Finally, developing a production-ready application is not the goal of this thesis. The application needs to be a usable prototype that demonstrates best practices in usability, accessibility, and authentication, and can later be developed into a production-ready application if necessary. This is because developing such an application is outside the scope of this thesis and would not provide sufficient benefits to justify the increased development time. The evaluation chapter will discuss potential further improvements of the application.

3.2 Selection of Tools and Frameworks

The first major tool used throughout the application's development process is Git, a version control system. Version control is the practice of recording every change made to a codebase so that, if needed, any previous version can be easily restored (Git, n.d.). To make sharing these changes across multiple devices seamless, Git provides the concept of "remotes". A Git remote is a server that stores a copy of the local files (and serves as the source of truth), so that files can be downloaded (cloned, per Git terminology) onto multiple devices and changes can be synchronized easily. GitHub, the most popular Git hosting service with over 150 million users (Lisowski, 2025), will be the choice of Git hosting service.

As discussed in Chapter 3.1, one of the most crucial aspects of the application framework used throughout the development process is cross-platform compatibility. A variety of frameworks offer this feature, each with its own set of benefits and drawbacks. Table 1 presents some widely adopted frameworks, their languages, and whether they support desktop and mobile.

Table 1. Application framework candidates

Name	**Languages**	**Desktop**	**Mobile**
Electron	JavaScript/TypeScript	Yes	Possible
.NET	C#	Yes	Yes
Kotlin Multiplatform	Kotlin	Yes	Yes
Flutter	Dart	Yes	Yes
Tauri	Rust + JavaScript/TypeScript	Yes	Yes

The table above lists the application frameworks examined during this selection process. Desktop and mobile compatibility are split into two columns because some frameworks officially support only one or the other. If the documentation of a framework explicitly mentions the support of a specific platform, then that is marked as “Yes”. If it does not, but research reveals that applications written in the framework can run on some platform, then it is marked as “Possible”. Otherwise, it is marked as “No”.

Electron is a mature application framework that emulates the browser application programming interface (API) and environment for desktop applications, allowing users to use web frontend technologies, such as React, to design the application. This is achieved by running a Chromium instance under the hood, which is what the browser Google Chrome (and its forks) is built on. Not only do these features allow developers to use tools and frameworks they are familiar with to build applications, but because of Electron’s cross-platform nature, the resulting applications are compatible with Windows, Linux, and macOS desktop operating systems (Electron, n.d.). However, Electron’s documentation does not mention support for mobile devices. While additional research shows that it is possible to run Electron applications on mobile platforms, doing so requires extra work.

.NET is a cross-platform, open-source application framework developed by Microsoft. It offers a wide range of APIs and supports a variety of operating systems out of the box, including desktop platforms Windows, macOS, and Linux, as well as mobile platforms Android and iOS. The primary programming language for .NET applications is C#, but other languages (such as F# and Visual Basic) are also supported (Microsoft, 2024).

Kotlin Multiplatform (KMP) is an open-source application framework developed by JetBrains and used in production by a variety of large

organizations. It is designed to work across Android, iOS, and any platform compatible with the Java Virtual Machine (JVM). Since the JVM supports Windows, Linux, and macOS, applications developed in KMP meet the operating system requirements outlined in Chapter 3.1. KMP offers a native feel and performance on mobile devices, even on platforms where running a virtual machine is suboptimal. For user interface (UI) development, KMP offers Compose Multiplatform. The primary programming language for KMP applications is Kotlin (Kotlin, n.d.).

Flutter is an open-source cross-platform application framework developed by Google. It takes a different approach to bundling applications compared to other frameworks. Instead of generating native code for platforms, the Flutter rendering engine (written in C++) and the Flutter Framework (written in Dart, the same language used to develop applications in Flutter) are bundled with the app, and only minimal native code is included, which is enough to get Flutter running on a specific platform. The benefit of this approach is speed; the downside is increased bundle size – an "empty" application is typically around 6.7MB in size (Amadeo, 2018). Flutter was originally designed as a cross-platform mobile application framework for Android and iOS; however, desktop platform support has since been added (Flutter, n.d.).

Tauri is an open-source, cross-platform application framework written in Rust designed to enable developers to build performant, yet small (in size), applications. Any frontend framework that compiles to HTML, CSS, and JavaScript is supported for UI development. At the same time, more resource-intensive computations and tasks are handled by a backend written in Rust, Swift, or Kotlin. Tauri, under the hood, uses each system's native webview, resulting in a smaller bundle size. Because it supports many frontend frameworks, it offers great flexibility. Finally, because Tauri is written in Rust, a language designed with safety in mind, it enables the development of secure applications (Tauri, n.d.).

After careful consideration, Tauri was chosen for the application framework. The primary reason was that it is compatible with React.js for frontend development, a framework the development team is well-versed in. Electron would also enable writing the frontend in React.js; however, it has limited mobile support out of the box, making Tauri a better fit. Although its documentation mentions that Rust, Swift, and Kotlin are all supported for backend development, Rust is the default option. For that reason, and because of the security and speed it offers, Rust was selected. This means the application will be built with Tauri, using React.js for the frontend and Rust for the backend.

To reduce the application's UI development time, a component library is needed: a collection of predefined components that typically require no styling, though individual components can be styled if needed. For this application, Shadcn/ui will be used – "a set of beautifully-designed, accessible components and a code distribution platform" (Shadcn, n.d.). It differs from most component libraries: while most offer a variety of components available as a single package, Shadcn provides a command-line interface (CLI) that lets developers install specific components from its registry. This way, only the components used in the application are installed (Shadcn, n.d.). Shadcn components are built on top of Radix Primitives, a low-level component library with an emphasis on accessibility. Radix promises that developers can use its components without implementing accessibility features themselves, which is crucial for this application. Increased accessibility is achieved by conforming to WAI-ARIA, a specification on the semantics of common UI patterns published by the World Wide Web Consortium (W3C) (Radix, n.d.).

Radix UI, behind the scenes, uses Tailwind CSS, a CSS framework that, unlike other such frameworks, does not provide prebuilt components; instead, it offers a large set of predefined classes that can be applied to HTML elements. These predefined classes come with specific styles; for example, the class "bg-yellow-50" sets the element's background to a

specific shade of yellow. If used properly, TailwindCSS can lead to cleaner code and more consistent styling across the application.

To simplify the fetching of linked account data from the Rust backend, TanStack Query will be used. It is a React.js library that simplifies fetching, caching, and state management by providing an opinionated approach to these areas (TanStack, n.d.). Although using such a library is not strictly required, it will simplify the development of support for linking new accounts to the application and for periodically fetching all linked accounts and their associated data to display in the application's UI.

Although it was found that scanning QR codes can reduce usability and accessibility, this application will still support QR code scanning. When the user adds a new account, they will be prompted to choose between manual entry and QR code scanning. This way, those who prefer scanning QR codes and those who prefer manually entering account data will have a suitable option. For this reason, a QR code scanner library is required. A popular library called "qr-scanner" will be used.

As discussed in Chapter 2.2, the two prevalent OTP generation algorithms are HOTP and TOTP; the latter being a time-based extension of HOTP. For the website and the application to generate the same code, they need to use the same algorithm. TOTP has been preferred over HOTP by many implementors because it is easier to implement. HOTP uses a counter incremented by 1 each time a code is generated. This means that validating servers need to account for users generating codes without submitting them to the server by opening the application and immediately closing it, causing the application's counter to become out of sync with the server's counter. To account for this, the server would have to accept multiple codes, which is suboptimal. Not only that, but HOTP codes do not have an expiry date; they remain valid until they are used (Lumburovska et al., 2021). Since TOTP is based on the current Unix time, which is split into time steps, users cannot force the

application to go out of sync with the validating server, and the generated codes will expire due to TOTP's nature. For these reasons, the application to be developed alongside this thesis will use TOTP. The frontend will be responsible for generating these codes. While in a typical web application, the backend would have to perform such tasks so as not to expose sensitive data (in this case the secret to generate codes from) to parts of the application, that users can access, a Tauri application (even if it is possible to split it into multiple components) is compiled to a single binary, that runs on the users device as a single unit. By generating codes on the frontend, the frontend does not have to poll the backend every few seconds to check whether new codes have been generated, resulting in better performance. A popular OTP generator package, "totp-generator", is going to be used.

Storage is another important aspect of authenticator applications. Storing account data alongside its respective secrets in a remote database would defeat the purpose of an authenticator application; after all, one of its strong suits is that codes are generated locally on the user's device without first fetching a secret or code from a server. This means the database must be local to the device running the application. SQLite is a C implementation of a structured query language (SQL) engine that is, among other things, "small", "fast", and "self-contained" (SQLite, n.d.). SQLite is built into most mobile phones and computers; its self-contained nature makes it an ideal candidate for this application's database. To enable the backend to communicate with an SQLite database, rusqlite, a Rust wrapper for SQLite, will be used.

3.3 Application Architecture

The application architecture is shown in Figure 5.

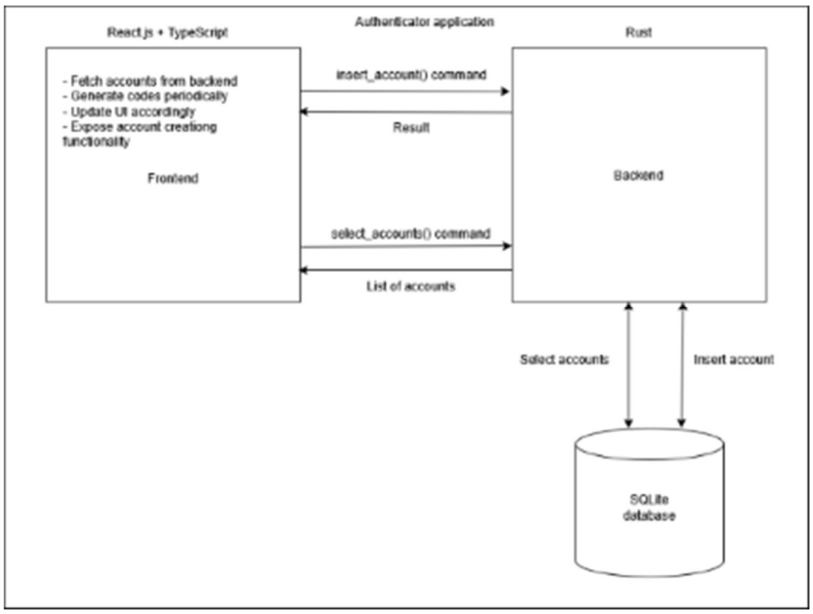


Figure 5. Application architecture

The application is split into a backend and a frontend, as has been discussed before. The backend's responsibility is to communicate with the local SQLite database, handle inserting new accounts, and query all existing accounts upon request. It is also responsible for exposing functions (in the form of Tauri commands, discussed in more detail in Chapter 4.5) to the frontend, enabling the frontend to store accounts and retrieve a list of existing accounts. The backend also serves as the application's system-level entry point.

The frontend is where the majority of the logic lives in this application. It is where the UI logic is, and since accessibility is inherently coupled to the UI, accessibility features are also located in the frontend. There is a repeating task running in the frontend that updates each code's expiry in the UI every second and generates new codes when needed. Since the frontend generates these codes, there is no need to poll the backend every few seconds for a new set, thereby improving performance. The frontend includes a screen listing linked accounts with

their respective codes and expirations, and another screen for linking new accounts to the application.

In a Tauri application, the frontend and backend are compiled into a single binary; however, each runs in its own process. The backend is the main system-level entry point; upon startup, it initializes the UI and starts the frontend. Tauri provides two ways for these components to communicate. The frontend can call the backend via the use commands (explained in more detail in Chapter 4.5), and the backend can emit events that the frontend can listen to and act upon. Since the latter mechanism is not used in this application, it will not be further explained.

4 IMPLEMENTATION

This chapter discusses the implementation of a prototype authenticator application that aims to address accessibility and usability issues outlined in previous chapters. First, the empty "skeleton" project is explained in detail. Then, the installation of project dependencies is described. Finally, the application's backend and frontend are discussed separately, detailing the thought process and choices made during implementation, along with the reasoning behind them.

4.1 Project Structure

Tauri provides a utility called create-tauri-app that generates the basic structure and files for any Tauri application, enabling developers to select from a variety of frontend frameworks and package managers. This utility is available for a variety of operating systems and package managers. Because the development takes place on a Windows computer, the PowerShell command provided by Tauri is used. Running the command generates the files shown in Figure 6.

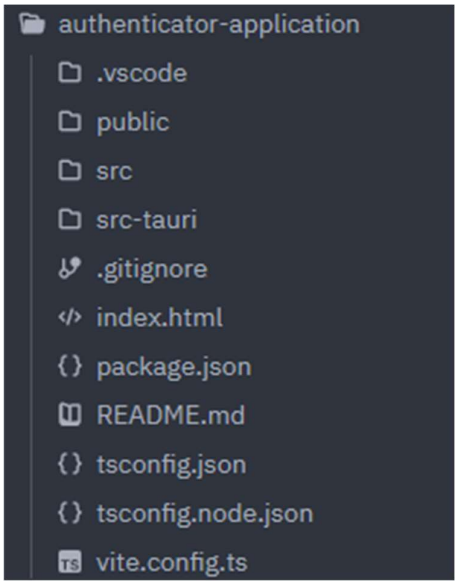


Figure 6. Tauri starter project

The generated file structure is very similar to that of a React.js application, except that it includes the "src-tauri" directory. Two folders can be removed right away, as they are not needed:

- ".vscode", which contains extension recommendations for Microsoft's text editor, VSCode
- "README.md", which contains basic instructions and recommendations for Tauri

The "public" folder contains scalable vector graphics (SVG) images used by the default application. These will be replaced during the development process. The "src" folder contains the frontend's React.js code, while the backend code lives in "src-tauri", which is specific to Tauri applications. The rest of the files are relatively standard for any JavaScript/TypeScript project. The file ".gitignore" is used by git to determine which files should be ignored; that is, which files should not be tracked or pushed to the remote repository. Index.html contains only the most basic HTML structure as well as a "mounting point", where the frontend's React.js code will be programmatically inserted. The file "package.json" contains a list of JavaScript/TypeScript dependency metadata, project metadata, and utility commands, called scripts, for running the application, type checker, tests, or the linter. Both "tsconfig.json" and "tsconfig.node.json" are related to TypeScript; they contain rules used by the type checker. Lastly, "vite.config.ts" is used by Vite, a bundler; it contains information on bundling and running the application.

By default, "package.json" contains React.js and Tauri dependencies; however, they are not automatically installed upon running the create-tauri-app utility. They can be installed by running "pnpm install" (or "pnpm i" for short) in the project's root folder. As a result, two more files are generated:

- "pnpm-lock.yaml" contains dependency-related information
- "node_modules" contains the actual dependency files

The “node_modules” folder is ignored by default, which is correct, but “pnpm-lock.yaml” is included in the remote repository.

4.2 Project Dependencies

The project uses two separate package managers because the JavaScript/TypeScript ecosystem is different from that of Rust. The most popular JavaScript/TypeScript package manager is called “Node Package Manager” (npm), but this project uses pnpm, which stands for “performant npm”. As the name implies, it provides, among other things, performance benefits over npm. Frontend dependencies are installed using this package manager. On the other hand, backend dependencies are installed using Cargo, Rust’s built-in package manager. This means that two different package managers will track project dependencies: one for frontend dependencies, and the other for backend dependencies.

Chapter 3.2 covered the process of selecting tools, libraries, and frameworks for this application that need to be installed at this stage. Shadcn, totp-generator, qr-scanner, and TanStack Query are all frontend dependencies and should be installed via pnpm; however, the installation process differs for each. TanStack Query and totp-generator are installed like any other dependency, by running “pnpm i @tanstack/react-query totp-generator qr-scanner”. However, as discussed in Chapter 3.2, Shadcn is not an actual project dependency, but a CLI tool designed to simplify the installation of specific Shadcn/UI components. For that reason, installing Shadcn differs from installing most JavaScript/TypeScript packages. First, two TailwindCSS packages need to be installed: “pnpm i tailwindcss @tailwindcss/vite”. Then, a new cascading stylesheet (CSS) file needs to be created in “src” and named “main.css” and imported in “main.tsx”. TailwindCSS needs to be imported into the newly created CSS file by adding “@import ‘tailwindcss’;”. This file will store various imports from TailwindCSS and Shadcn/UI, as well as a few custom styles generated by the latter. After

that, the tailwind CSS Vite plugin needs to be added to the plugin list in "vite.config.ts". Then, a new entry needs to be added to "tsconfig.json" under "compilerOptions", as shown in Figure 7.

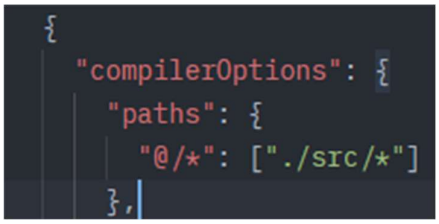


Figure 7. Updated tsconfig file

Finally, Shadcn can be installed by running "pnpm dlx shadcn@latest init -t vite". The flag "-t vite" is used to specify that this application is using Vite instead of other bundlers.

The backend has 2 primary dependencies: totp_rs and rusqlite, which need to be installed via Cargo, Rust's dependency manager. Since Cargo is not available from the project's root directory, navigating to "src-tauri" is necessary: "cd src-tauri". Finally, the dependencies are installed via "cargo add rusqlite totp_rs".

4.3 Frontend Structure

The frontend-related files are located in the "src" folder. The layout is that of a typical React.js application. The "assets" folder typically contains various assets; in this case, it contains only a single SVG React.js logo. The "components/ui" directory is where components installed via the Shadcn cli tool are placed. The button component was downloaded automatically when Shadcn was installed. Shadcn also generated the "lib" folder alongside "utils.ts," which is inside it. By default, "utils.ts" contains TailwindCSS-related utility functions, but it is possible to expand it as needed. The main CSS file in this application is "App.css", which contains pre-generated CSS rules that will be removed

later. Finally, "App.tsx" exports the root React.js component, called App, which is imported in "main.tsx". "main. tsx" is responsible for programmatically mounting the HTML nodes generated by the React.js code in "index.html" in the project's root directory. Figure 8 shows the files automatically generated by the create-tauri-app utility for the frontend.

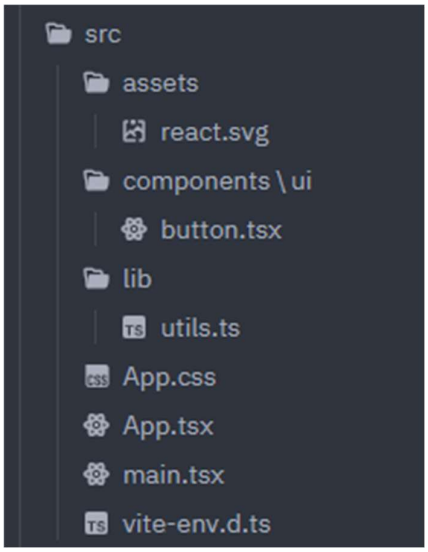


Figure 8. Frontend structure

4.4 Backend Structure

The backend-related files are located in the "src-tauri" directory. The "capabilities" folder contains permissions granted or denied to the window or webview in which the application runs. There are two folders, "gen/schema" and "target", that were auto-generated by Cargo and Tauri, and are ignored by default. They contain schema files for auto-completion, compiled files, and executables. There is an "icons" folder containing Tauri icons of various sizes, which will be removed later. The "src" folder contains the backend logic written in Rust; by default, "main.rs" and "lib.rs" are generated, with the first serving as the application entry point and the second containing the actual backend logic. This is where the backend code will be placed. There is a separate ".gitignore" file that ignores the aforementioned "gen/schema" and

"target" directories. Files "build.rs", "Cargo.lock", and "Cargo.toml" are part of the Rust ecosystem and are responsible for defining build logic and managing dependencies. Finally, "tauri.conf.json" contains Tauri-specific configuration. Figure 9 shows the auto-generated backend-related files and directories.

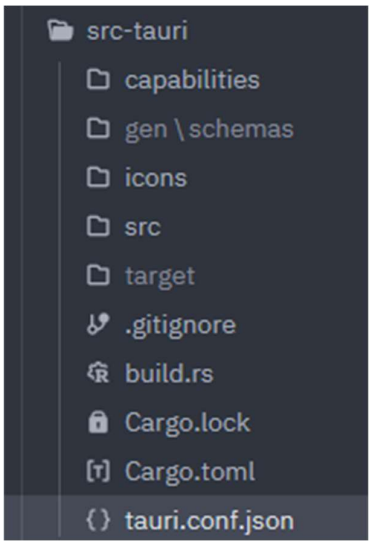


Figure 9. Backend structure

4.5 Frontend-backend Communication

Tauri applications are split into a frontend and backend, a terminology inherited from typical web applications. The frontend is responsible for displaying information to users and enabling user interaction with the application. Proper usability and accessibility – which are key aspects of this application – are also the responsibility of the frontend. On the other hand, the backend contains logic to link accounts by inserting them into the database and returning their information to the frontend for display.

Communication between these two components is made possible by a Tauri concept called commands, which are a form of inter-process communication (IPC). A command is a Rust function defined in the

backend component, annotated with the "#[tauri::command]" attribute. Figure 10 shows an example Tauri command.

```rust
#[tauri::command]
fn greet(name: &str) -> String {
    format!("Hello, {}! You've been greeted from Rust!", name)
}
```

Figure 10. Example Tauri command

The command shown in Figure 8 is named "greet", accepts a single string parameter, and returns a string value. This command can be invoked by the frontend component via the "invoke" function exposed by Tauri, as shown in Figure 11.

```javascript
async function fetchGreetingFromBackend() {
  return invoke<string>(cmd: "greet", args: { name: "John Doe" });
}
```

Figure 11. Tauri command invocation

The "invoke" function is exposed to the frontend by Tauri. It is an async function that accepts three parameters (although in the above example, only two are passed): the command name, which is the Rust function's name, an argument list, where the name and type of the arguments have to match those in the Rust function's argument list, and an options object, that is not set in the above example. The "invoke" function is also generic, because the TypeScript code has no way of determining the return type of the Rust function. In the above example, the Rust function returns a string, so the generic parameter of "invoke" needs to be set to the same thing. Tauri can translate types from the frontend to the backend and vice versa, so a Rust "String" or "&str" becomes a TypeScript "string", for instance.

4.6 Implementation Details

For each linked account, the following information is stored: a unique ID, the website name in a customizable format, the email address of the account, the issuer of the code (often the website's name), the secret to generate codes from, the number of digits of the OTP, time step at which new codes need to be issued, and the algorithm used by TOTP. The reason for storing both the issuer and the title is to help users easily differentiate between accounts linked to the same website (such as a work and a personal account). They can set the name accordingly, and it appears correctly in the UI. The issuer could not be used for this, however, as it needs to be the actual name of the website or organization issuing the code.

4.6.1 Backend Implementation

In this project, the backend is far less complicated than the frontend; it only exposes basic functionality via Tauri commands: one to insert a new account into the database, one to delete an account, and one to fetch all linked accounts. This means that, contrary to the more usual setups, most of the application logic lives in the frontend. The reason for this is that, usually, in web applications, the frontend and the backend are separate from each other; one runs on a server, while the other runs in the user's browser. This means that critical information, such as database credentials or business logic, cannot leave the backend. However, even though a Tauri application technically consists of a frontend and a backend, it still counts as a single application, so keeping some (or even most) of the business logic can make sense in certain situations.

The backend consists of five Rust files, as shown in Figure 12.

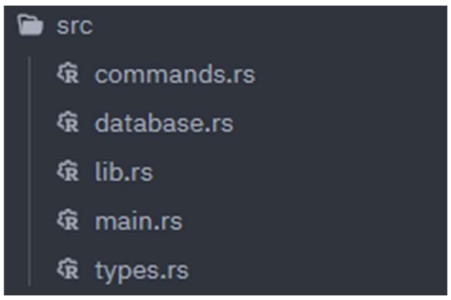


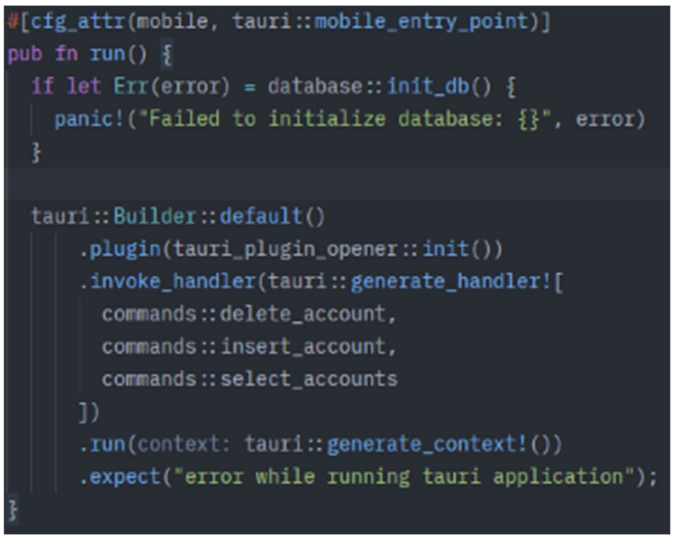
Figure 12. Backend custom file structure

The file “commands.rs” contains the definition for every command exposed to the frontend, “database.rs” contains the main database logic, “lib.rs” is where the Tauri application is configured, “main.rs” is the main entry point of the application, and “types.rs” contains type definitions used by the backend. Every file seen in Figure 10, besides “lib.rs” and “main.rs”, is called a module, and needs to be “registered” in one file before it can be used anywhere. In this application, “lib.rs” is the file that registers these custom modules, as shown in Figure 13.

```
mod commands;
mod database;
mod types;
```

Figure 13. Module registration

Additionally, “lib.rs” contains only the Tauri entry point. This is where (among other things) commands can be registered. This application initializes the database before Tauri launches, then registers Tauri commands. If database initialization fails, the application “panics” (crashes). This behavior would not be ideal in a production-ready application, but it is acceptable in a prototype. Figure 14 shows the Tauri entry point of the application.



```rust
#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
  if let Err(error) = database::init_db() {
    panic!("Failed to initialize database: {}", error)
  }

  tauri::Builder::default()
      .plugin(tauri_plugin_opener::init())
      .invoke_handler(tauri::generate_handler![
        commands::delete_account,
        commands::insert_account,
        commands::select_accounts
      ])
      .run(context: tauri::generate_context!())
      .expect("error while running tauri application");
}
```

Figure 14. Tauri entry point

This entry point is called from “main.rs”, which is the typical entry point of a Rust application. Most of “lib.rs” and all “main.rs” were auto-generated by Tauri when creating the project; the only custom things in “lib.rs” are the registration of custom modules and commands. Figure 15 shows the application’s entry point, “main.rs”.

```rust
// Prevents additional console window on Windows in release, DO NOT REMOVE!!
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]

fn main() {
    authenticator_application_lib::run()
}
```

Figure 15. Application entry point

As mentioned previously, “types.rs” contains custom types used in the application. The information to store for an account is described in Chapter 4.6. In the backend, the account type is implemented as seen in Figure 16.

```rust
#[derive(Serialize, Deserialize)]
pub struct Account {
  pub id: String,
  pub secret: String,
  pub title: String,
  pub issuer: Option<String>,
  pub digits: Option<u8>,
  pub period: Option<u8>,
  pub algorithm: Option<Algorithm>
}
```

Figure 16. Rust account type

The properties "id", "secret", and "title" are all required, whereas the rest of them are optional, hence the use of Rust's "Option" type. Properties "id", "secret", "title", and "issuer" all hold a String value, "digits" and "period" all hold a byte (unsigned integer of 8 bits, u8), and "algorithm" is a custom Rust enum as seen in Figure 17.

```rust
#[derive(Serialize, Deserialize)]
pub enum Algorithm {
  SHA1,
  SHA256,
  SHA384,
  SHA512
}
```

Figure 17. Rust algorithm enum

Both of the above types have the "#[derive(…)]" attribute, which tells the Rust compiler to generate a basic implementation for the provided traits, "Serialize" and "Deserialize" in this case. This enabled the frontend to pass in an account formatted as a JavaScript Object Notation (JSON) string, and the backend will receive it as an instance of the above

“Account” struct. To access these traits, they need to be imported first. Figure 18 shows the imports found in “types.rs”.

```
use rusqlite::{Error, ToSql, types::{ToSqlOutput, Value}};
use serde::{Serialize, Deserialize};
```

Figure 18. Imports in "types.rs"

Other than the discussed types and imports, two implementation blocks are located in “types.rs”. The first is the “rusqlite::ToSql” implementation for the “Algorithm” enum, which enables rusqlite to translate those enum values to SQL types when used in SQL queries. This functionality is used in “database.rs”. The other attaches the function “from” to “Algorithm”, which is also used in “database.rs”. It enables parsing an “Option<String>” into an “Option<Algorithm>” enum value, if possible. These implementation blocks are shown in Figure 19.

```
impl ToSql for Algorithm {

  fn to_sql(&self) -> Result<ToSqlOutput<'_>, Error> {
    let name = match &self {
      Algorithm::SHA1 => "SHA1",
      Algorithm::SHA256 => "SHA256",
      Algorithm::SHA384 => "SHA384",
      Algorithm::SHA512 => "SHA512"
    };

    Ok(ToSqlOutput::Owned(Value::Text(name.to_string())))
  }
}
```

```
impl Algorithm {

  pub fn from(string: Option<String>) -> Option<Algorithm> {
    string.and_then(|it| match it.as_str() {
      "SHA1" => Some(Algorithm::SHA1),
      "SHA256" => Some(Algorithm::SHA256),
      "SHA384" => Some(Algorithm::SHA384),
      "SHA512" => Some(Algorithm::SHA512),
      _ => None
    })
  }
}
```

Figure 19. Implementation blocks in “types.rs”

All database-related logic is located in “database.rs”, although the initialization is actually invoked in “lib.rs”, as seen previously. Database logic includes creating the database table if it does not yet exist, inserting and deleting accounts, and fetching all accounts. SQL commands are stored in one place as constants. The upside is that they

do not clutter the code of the function that uses them. Figure 20 shows these SQL commands.

```
const DB_FILE: &str = "authenticator.db";
const SQL_CREATE_TABLE: &str = "CREATE TABLE IF NOT EXISTS `accounts`(\
`id` VARCHAR(36) NOT NULL PRIMARY KEY,\
`secret` VARCHAR(255) NOT NULL,\
`title` VARCHAR(36) NOT NULL,\
`issuer` VARCHAR(50),\
`digits` INTEGER,\
`period` INTEGER,\
`algorithm` VARCHAR(10)\
);";
const SQL_INSERT_ACCOUNT: &str = "INSERT INTO `accounts` VALUES (?1, ?2, ?3, ?4, ?5, ?6, ?7);";
const SQL_SELECT_ACCOUNTS: &str = "SELECT * FROM `accounts`;";
const SQL_DELETE_ACCOUNT: &str = "DELETE FROM `accounts` WHERE `id` = ?1;";
```

Figure 20. SQL commands

The “accounts” database table stores account properties, including their names and types. The columns “id”, “secret”, “title”, “issuer”, and “algorithm” are of type VARCHAR, a textual type, with a set maximum length. In most cases, the actual length of the content stored in these columns is unknown in advance, except for “id,” which is always a universally unique identifier (UUID) with a fixed length of 36 characters. Since the lengths of the other properties are unknown, a reasonable maximum limit was set to avoid issues later. SQLite offers a similar datatype, called TEXT. The main difference between the two is that VARCHAR has a maximum length, whereas TEXT does not, making it ideal for longer descriptions or comments. Since TEXT does not have a maximum length, SQLite needs to allocate more space for it, making it less storage-efficient. For that reason, VARCHAR was chosen for these columns. The INTEGER type used for the columns “digits” and “period” can hold numeric values. Finally, required values are marked as “NOT NULL”, indicating that null values are not allowed by the database. Since all the other properties of an account are optional, the columns have not been marked “NOT NULL” to be in line with that.

The remaining SQL commands demonstrate rusqlite's approach to handling dynamic SQL parameters. The command only contains a reference to the parameter (formatted as ?number, starting from one, so the first parameter is expressed as ?1), and the actual list of parameters is passed to rusqlite as a Rust tuple. The previously mentioned "rusqlite::ToSql" trait implementation is needed in order to pass an "Option<Algorithm>" type as a parameter to rusqlite. Figure 21 shows the usage of SQL parameters in rusqlite.

```
pub fn insert_account(account: &Account) → Result<usize, Error> {
  use_db(handler: |conn| conn.execute(SQL_INSERT_ACCOUNT, params: (
    &account.id,
    &account.secret,
    &account.title,
    &account.issuer,
    &account.digits,
    &account.period,
    &account.algorithm
  )))
}
```

Figure 21. Parameters in rusqlite

The function shown in Figure 19 is used to insert a new account into the database. It takes a single parameter, an Account reference, inserts it into the database, and returns a "Result<usize, Error>" type. Since Rust does not have the concept of exceptions, the Result type is the standard type used to either return a success ("Ok") or an error ("Err") value from a function. In this case, usize is the number of rows affected by the SQL command, whereas Error is a standard rusqlite error indicating a database failure.

The "use_db" function used by "insert_account", as depicted in Figure 21, is a utility function that expects a single parameter: a function that takes a rusqlite::Connection and uses it. This function opens a

connection to the appropriate database files, passes the connection to the function provided as the only argument, and then automatically drops the connection.

Another use of this function is in the "select_accounts" function, which, as the name implies, returns all entries from the database. The function returns a success value of a list of accounts (vector, or vec for short, to adhere to Rust's terminology), or a standard "rusqlite::Error" in case of a database error. Each returned tow is mapped to a successful account value (the "Algorithm::from" function, which was discussed earlier, is used by this function), after which the iterator is "collected" into a vector. Figure 22 shows the function "select_accounts".

```
pub fn select_accounts() → Result<Vec<Account>, Error> {
  use_db(|conn| {
    let mut stmt = conn.prepare(SQL_SELECT_ACCOUNTS)?;
    let rows = stmt.query(params: [])?;

    let accounts: Vec<Account> = rows
      .map(|row| Ok(
        Account {
          id: row.get(idx: 0)?,
          secret: row.get(idx: 1)?,
          title: row.get(idx: 2)?,
          issuer: row.get(idx: 3)?,
          digits: row.get(idx: 4)?,
          period: row.get(idx: 5)?,
          algorithm: Algorithm::from(string: row.get(idx: 6)?)
        }
      ))
      .collect()?;

    Ok(accounts)
  })
}
```

Figure 22. Select accounts function

Finally, the commands located in "commands.rs" expose these database functions to the frontend. The error handling in these commands is acceptable for a prototype: errors are only logged to the console, and

the frontend never receives that information to inform the user. In a production-grade application, better error handling would be required. Figure 23 shows the exposed Tauri commands.

```rust
use crate::database;
use crate::types::{Account};

#[tauri::command]
pub fn insert_account(account: Account) -> Result<usize, String> {
  database::insert_account(&account).map_err(|it| format!("Failed to insert account into database: {}", it))
}

#[tauri::command]
pub fn select_accounts() -> Result<Vec<Account>, String> {
  database::select_accounts().map_err(|it| format!("Failed to select accounts from database: {}", it))
}
```

Figure 23. Custom commands

4.6.2 Frontend Implementation

There are a few accessibility-related considerations to take into account. The first, and probably the most important, rule is to use semantic HTML tags, which is considered best practice. For instance, an HTML " " component can act as a button but lacks the same properties, leading to confusion for screen reader users. Sometimes, visible content does not translate well to screen readers, the data structure might be confusing, or the data itself might be ambiguous. In such cases, adding the "aria" attribute can solve this by providing better names or descriptions. Decorative elements should be hidden using the "aria-hidden" attribute, as they are only visual and may be considered unnecessary noise by a screen reader user. It is also important that the UI is navigable solely with the keyboard, without a mouse, which is mostly the case when proper HTML semantics are followed. Though not all, most of these considerations are addressed by Radix UI.

The UI framework of choice in the frontend is React.js. It is based on the concept of components, where each component renders only a small part of the UI. This way, if a state changes and the UI needs to be updated, it suffices to re-render only that specific component.

Reusability is another important aspect of components. For instance, a button with specific styling can be extracted into its own component and reused throughout the project, rather than using a generic button and applying the same style in each instance.

React.js components are written in the JSX language (or TSX, if TypeScript is used instead of JavaScript). JSX combines standard JavaScript code and HTML-like document structure. Each component is defined as a standard JavaScript function that can then be used like an XML tag. Such functions can take an object of some shape, called "props", which are passed to the component as attributes, and return an HTML-like object. Figure 24 shows an example React.js component.

```tsx
import * as React from "react";

export type NoContentProps = {
  title: string;
  description: string;
  Icon: React.ComponentType<{ className: string }>;
};

export function NoContent(props: NoContentProps) {
  return (
    <div className="flex flex-col items-center justify-center text-center p-10 gap-4" role="status" aria-live="polite">
      <div className="w-16 h-16 rounded-full bg-muted flex items-center justify-center" aria-hidden="true">
        <props.Icon className="w-8 h-8 text-muted-foreground" />
      </div>

      <h2 className="text-lg font-semibold">{props.title}</h2>

      <p className="text-sm">{props.description}</p>
    </div>
  );
}
```

Figure 24. Example React.js component definition

In the above example, the "NoContent" component is defined. It takes a "props" object whose type is defined in "NoContentProps. The "title" and "description" props are standard strings; however, "Icon" is actually another React.js component. The function's name is written in PascalCase as opposed to camelCase to be in line with React's naming convention. This example component can then be used like an HTML tag, as seen in Figure 25.

```
<NoContent
  title="No accounts yet"
  description="Use the add button or scan a QR code to get started."
  Icon={Inbox}
/>
```

Figure 25. Example React.js component usage

The frontend of this application contains a few custom components, and some that were installed via Shadcn. Most components do not have complicated logic worth describing; therefore, only the more complex ones will be explained in detail. Figure 26 shows an overview of the frontend files.

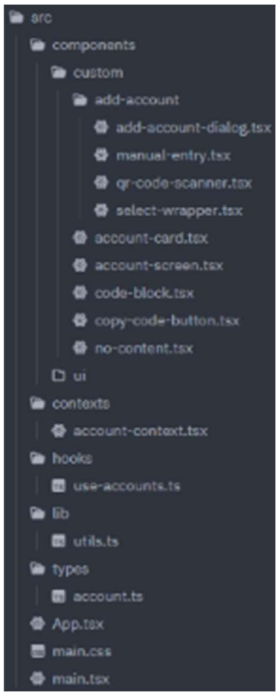


Figure 26. Custom React.js components

As discussed before, “main.tsx” is the React entry point of this application; the only thing that is performed in that file is mounting the

"<App />" component. The main CSS file (literally named "main.css") houses basic styling as well as the import of TailwindCSS styles. "App.tsx" sets up context providers and the main screen; this will be explained in more detail later. The "components" directory includes two other directories, "custom" and "ui". The first houses components developed for this application; the latter is where Shadcn components are installed via the command-line tool. The "contexts" folder contains custom React.js context; similarly, the "hooks" folder contains custom React.js hooks. The "lib" folder contains a single utility file, "utils.ts," and the "types" directory contains the custom type definitions.

A React context is a special type of component that typically consists of a context and a provider. The goal of contexts is to eliminate the phenomenon of "prop drilling"; if component A has a child, called B, who also has a child, called C, and C requires some property, often A and B would also have to add that their respective "props" list so that they can pass it down to component C. A context eliminates this problem. The actual context can hold data, and any component that is a descendant of the corresponding context provider can use its value. Figure 27 shows an example.

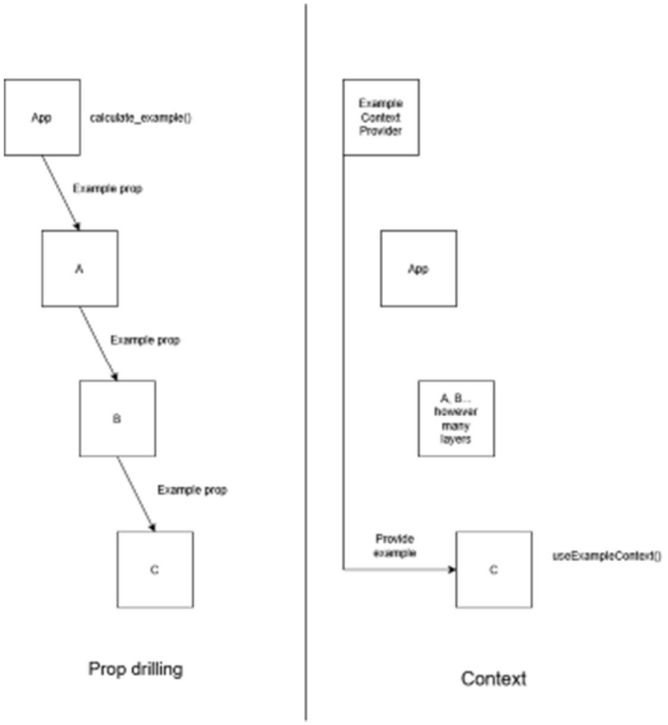


Figure 27. Prop drilling vs. context

To show up-to-date expiry values and codes in the UI, a heartbeat functionality is implemented in the "account-context.tsx" context file. It is a recurring task that runs every second, recalculates code expiry, and generates new codes as needed. If the backend handled code generation, the frontend would have to call the backend to obtain new codes, wasting resources. This way, the backend is only called when the application starts, and an up-to-date list of accounts needs to be queried, and when a new account is created. Figure 28 shows the "tick" function, which runs every second.

```
const tick = React.useEffectEvent(callback: async () => {
  const input = computed.length === 0 ? accounts.map(callbackfn: it => withDefaults(account: it)) : computed;

  const result = await Promise.all(
    values: input.map(callbackfn: async it => {
      const codes: Codes = {
        current: it.codes.current,
        upcoming: it.codes.upcoming,
        expiry: it.codes.expiry - 1
      };

      if (codes.expiry <= 0) {
        const current = await generateTotp(account: it);
        const upcoming = await generateTotp(account: it, options: { upcoming: true });

        codes.current = current.code;
        codes.upcoming = upcoming.code;
        codes.expiry = current.expiry;
      }

      return { ...it, codes };
    })
  );

  setComputed(value: result);
});
```

Figure 28. The "tick" function

The above function is wrapped in a “React.useEffectEvent” call. Effect events always get access to the most up-to-date state values, which is required in this instance. The state value “computed” is a list of accounts fetched from the backend, but expanded to contain the current and upcoming codes, as well as the current code’s expiry. When the first cycle runs at startup, the “accounts” list, fetched directly from the backend, will already be available, and default values for the aforementioned properties are set (empty strings for the current and next codes, and 0 for expiry). Then, expiry is calculated from the current value. Since 30-, 60-, and sometimes 15-second expiry is accepted, having just one task running every 30 seconds to recalculate codes would not suffice. The expiry is calculated by subtracting 1 from the current value; in this case, the expiry serves as a time-to-live value measured in seconds (which is ideal because it can be shown directly in the UI without further conversions). If the result is less than or equal to zero, the code has expired, so a new current and upcoming code is

generated. Technically, the new current code is the previous upcoming code, so that this section could be further optimized. If new codes were generated because the previous codes expired, the "expiry" value is reset to its default. Because the "generateTotp" function is asynchronous, the array of accounts is mapped to an array of "Promise<Account>", then wrapped in a "Promise.all" call and awaited. This way, once all asynchronous computation has finished, the new list of accounts with up-to-date codes and expiry values is ready, and the "setComputed" function is called, updating the "computed" state value and triggering a re-render. This way, the UI is immediately refreshed, and the new codes and expiry values are shown to the user.

The "tick" function is set up in a "React.useEffect" call. Effects run when the component is first rendered, and when any of their dependencies change. JavaScript provides a function called "setInterval" that accepts two parameters: a function and an interval in milliseconds. The provided function will then be invoked periodically at the provided interval. The "setInterval" function returns a task ID that can be used to stop this interval by passing it to "clearInterval". Figure 29 shows how the heartbeat is set up and stopped.

```
React.useEffect(effect: () => {
  tick();
  const heartbeat = setInterval(callback: tick, delay: 1000);

  return () => {
    clearInterval(timeout: heartbeat);
  };
}, deps: []);
```

Figure 29. Interval setup and stop

First, the "tick" function is called, then the heartbeat is set up via "setInterval". This is done so that "tick" runs immediately rather than

after a 1-second (1000 millisecond) delay. A React.js effect can return a function, which is used to perform cleanup after that effect. This is called when the component unmounts. In the code snippet above, the heartbeat is cleared, so it does not run any more cycles.

Fetching the current list of accounts and adding specific accounts can be done by calling the backend via Tauri commands. As discussed in Chapter 4.6.1, the backend exposes two commands: "select_accounts" to query all accounts from the database, and "insert_account", which expects an account as a parameter and inserts it into the database. To simplify command invocation and, especially, the management of data returned by the backend, TanStack Query, a lightweight data-fetching and state-management library, was installed. Although it offers many more features, in this application, it is used to easily integrate asynchronous "invoke" calls into a typical React.js context. Figure 30 shows the invocation of these commands.

```
const { data: accounts } = useSuspenseQuery({
  queryKey: ["acounts"],
  queryFn: () => invoke<Account[]>("select_accounts")
});

const addAccountMutation = useMutation({
  mutationKey: ["add", "account"],
  mutationFn: (account: Account) => invoke("insert_account", { account })
});
```

Figure 30. Command invocation

In the above example, two commands are invoked. The first one, "select_accounts", is a query because it only fetches data from the backend. It is wrapped in a "useSuspenseQuery" call. A suspense query takes advantage of React's "suspense" feature, which allows the application to show a user-defined fallback while waiting for an asynchronous operation to complete. In this application, the

asynchronous operation is fetching accounts from the backend, while the fallback component is a spinner.

The second command in Figure 29 is "insert_account", which is wrapped in a "useMutation" call. A mutation is when data is changed in the backend (for instance, by creating, updating, or deleting data). The "useMutation" call returns a "UseMutationResult" object with a "mutate" method, which needs to be called to actually perform the mutation. By default, this "mutate" method does not take any arguments; however, the "mutationFn" property in this case is set to a function that does take a single argument of type "Account". Because of this, the "mutate" function will also take an argument of the same type.

The account context exposes three things: a list of accounts with their current up-to-date codes and code expiry dates, a function to add an account, and another to remove accounts. The account list is what the heartbeat "tick" function operates on, ensuring that components working with this account list always receive up-to-date data. The add function calls the mutation and, to avoid refetching the account list, adds the newly created account with default values to the "computed" accounts list. Although the account context exposes the remove account functionality, it is not wired up and is not used in the application. Figure 31 shows the add account function.

```
const add = (account: Account) => {
  addAccountMutation.mutate(account);
  setComputed([...computed, withDefaults(account)]);
};
```

Figure 31. Add account function

Finally, the account context exports the actual context object and a context provider component. The context provider is then imported into the "App" component, where it is added to the component tree along

with TanStack Query's "QueryClientProvider". Meanwhile, the context object is imported by the "use-accounts" hook. This hook exports a React hook function that can be used in other components to get access to the account context value. Figures 32 and 33 show the usage of "AccountContextProvider" and the account context object, respectively.

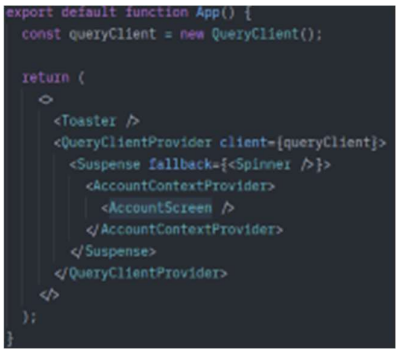


Figure 32. Context provider usage

```
export function useAccounts() {
  const context = useContext(context: AccountContext);
  if (context === null) {
    throw new Error(message: "useAccounts() called outside provider.");
  }

  return context;
}
```

Figure 33. Accounts hook

React forbids accessing context values outside context providers. The error message in the above example reflects that. The context value will be null if this hook is used outside of the corresponding context provider. "Outside" could mean two things: either the context provider is only rendered conditionally (for instance, when the user is authenticated, a different screen with different context providers is available compared to the unauthenticated screen), or the hook is called somewhere in a

standard JavaScript/TypeScript function. In this application, only the latter could theoretically happen; the context providers are always part of the component tree.

The application's main screen is divided into two sections: the navbar and the account list. The navbar contains an “Authenticator” logo, a search bar, and an “Add” button to add accounts. The navbar is positioned at the top of the application in desktop mode, but the search bar and the “Add” button are moved to the bottom on mobile devices for better ergonomics. This is achieved by having two similar navbars in the code, but rendering only one based on the device's screen size. This is achieved by using Tailwind’s media query classes combined with visibility modifier classes. The navbar at the top (desktop navbar) has the following Tailwind classes: “hidden md:flex”. This means that initially, the “display” CSS property is set to “none”, while on screens with a width of at least 48 rem, “display” is set to “flex”. A similar approach is used for the mobile navbar, but in reverse. It has a class of “md:hidden”, which means that it is visible by default; however, on screens with at least 48 rem of width, “display” is set to “none”. Although there is a “visibility” property in CSS, the “hidden” class in Tailwind changes the “display” property, because “visibility” does not truly hide an element. The space it would take up would be reserved (and empty), but the element itself would not be visible, whereas elements, whose “display” property is set to “none”, behave as though they are not part of the DOM. Tailwind defines five utility classes that correspond to various screen sizes. The “md” class can be translated into CSS as “@media (width >= 48rem)”; 48 rem is 768 pixels. By combining a resolution class with another Tailwind utility class, the second class's style applies only when the screen size matches the first class's media query. Figure 34 shows an example of such conditional styling.

```
{/* Desktop search bar */}
<div className="hidden md:flex flex-1">
```

Figure 34. Conditional styling

The account list renders each account's information grouped in a grid with one, two, or three columns, depending on the screen size. To get access to the account list, the previously described "useAccounts()" hook is called in this component. However, this list cannot be rendered directly, because the search functionality would not work that way. Instead, the current search term (referred to as "filter" in the code) is stored in React state, and a new variable, "filteredAccounts," is introduced to contain only matching accounts. An account is considered a match if its title or issuer contains the search term (case-insensitive). This way, only the information from the matching accounts is shown to the user. Figure 35 shows the use of the "useAccounts()" hook and the search implementation.

```
const { accounts, add } = useAccounts();
const [filter, setFilter] = React.useState<string | null>(initialState: null);
const filteredAccounts = filter ? accounts.filter(predicate: it => accountMatches(account: it, search: filter)) : accounts;

const search = (filter: string) => setFilter(value: filter.length === 0 ? null : filter);
```

Figure 35. Account search

The "search" function is assigned to the input's "onChange" handler, which is invoked when the input's value changes (e.g., when a user types or deletes the input's content). When the user types something, the filter value is automatically updated. Because the "filteredAccounts" value depends on the "filter" state variable, if the latter changes, so does "filteredAccounts", which causes the account screen to be re-rendered, leading to "live" search results; as the user types, the UI is updated accordingly, without having to press a search button. Figure 36 shows the desktop search input.

```
{/* Desktop search bar */}
<div className="hidden md:flex flex-1">
  <Input
    placeholder="Search accounts..."
    aria-label="Search authentication accounts"
    className="w-full"
    name="Desktop search bar"
    onChange={event => search(filter: event.target.value)}
  />
</div>
```

Figure 36. Desktop search

To add a meaningful fallback when the search yielded no results, or when no accounts have been added to the application yet, certain components are conditionally rendered. The account list is only rendered if the "filteredAccounts" list contains at least one account. Otherwise, appropriate fallback text is shown to the user. If "filteredAccounts" is empty but the "accounts" list is not, it means the search yielded no results, even though accounts were added. If both lists are empty, no accounts have been added yet. Figure 37 shows the rendering logic in the account screen.

```
<div id="accounts-container" className="pb-20 md:pb-0">
  {filteredAccounts.length > 0 && (
    <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6 p-6" role="list">
      {filteredAccounts.map(callbackFn: account => (
        <AccountCard key={account.id} account={account} />
      ))}
    </div>
  )}
</div>

{filteredAccounts.length === 0
  && (accounts.length === 0 ? (
    <NoContent
      title="No accounts yet"
      description="Use the add button or scan a QR code to get started."
      Icon={Inbox}
    />
  ) : (
    <NoContent
      title="No accounts matched"
      description="No account was found with this title or issuer."
      Icon={SearchX}
    />
  ))}
```

Figure 37. Account rendering logic

The logic for code generation is in the "utils.ts" file, where all other utility functions are defined. The only required parameter to generate a TOTP is the secret. However, there are a couple of optional parameters: the number of digits, the time step, the hashing algorithm, and the timestamp. By default, a code consists of 6 digits, has a 30-second time step, is generated using the SHA-1 algorithm, and uses the current timestamp. There can be some variations, however. Although less common, 7- and 8-digit codes also exist; the time step can be 60, and other hashing algorithms can be used, such as SHA-256, SHA-384, or SHA-512. Figure 38 shows the code generator function.

```
export async function generateTotp(account: Account, options?: { upcoming: boolean }) {
  const digits = account.digits ?? DEFAULT_DIGITS;
  const algorithm = account.algorithm ?? DEFAULT_ALGORITHM;
  const period = account.period ?? DEFAULT_PERIOD;

  const timestamp = options?.upcoming ? Date.now() + period * 1000 : Date.now();

  const { otp } = await TOTP.generate(key: account.secret, options: {
    digits,
    algorithm: hyphenateAlgorithm(input: algorithm),
    period,
    timestamp
  });

  return { code: otp, expiry: period };
}
```

Figure 38. Code generation function

To avoid wasting storage, these default values are not stored in the database. For that reason, the variables "digits", "algorithm", and "period" are taken from the "account" variable, and if they are null, they are replaced with the aforementioned default values. The "options" variable passed to this function contains only one property, called "upcoming". Setting this property to true will cause the upcoming code to be generated, which is not yet valid (though servers may already accept it to account for clock drift). The "timestamp" variable is calculated based on whether the code is supposed to be the current, in

which case the current datetime is used, or the upcoming, in which case the period (expressed in milliseconds) is added to the current datetime (which is also expressed in milliseconds). The "hyphenateAlgorithm" whose result is passed to the "algorithm" property of the "generate" function converts the hashing algorithm's name from the standard format ("SHA1") to a hyphenated format ("SHA-1"). This is required; the algorithm property would not be taken into account otherwise.

The standard OTP auth uniform resource identifier (URI) is formatted as "otpauth://<type>/<title>?parameters". Type is a required parameter whose value can be either "hotp" or "totp", in lowercase. Since this application only supports TOTP, if the type is not "totp", an error is thrown. Title (officially called label) is the title that the application should display for that specific account. Parameters can contain a few options properties, such as "digits", "period", "algorithm", and "issuer".

A function called "parseAccountFromUri" is located in "utils.ts", which can parse a standard OTP auth URI into an account. When the user scans a QR code to add an account, the QR code contains an OTP auth URI, which this function extracts and parses. Figure 39 shows this function.

```
export function parseAccountFromUri(url: URL): Account {
  const { protocol, pathname, searchParams } = url;
  if (protocol !== "otpauth:") throw new Error(message: "Unsupported protocol");

  if (url.host !== "totp") throw new Error(message: "HOTP is not supported by this application");

  const title = pathname.slice(start: 1);
  if (!title) throw new Error(message: "Empty title");

  const secret = searchParams.get(name: "secret");
  if (!secret) throw new Error(message: "Missing secret");

  const issuer = searchParams.get(name: "issuer") ?? title;
  const digits = parseSafeNumber(input: searchParams.get(name: "digits")) ?? undefined;
  const period = parseSafeNumber(input: searchParams.get(name: "period")) ?? undefined;
  const algorithm = validateAlgorithm(input: searchParams.get(name: "algorithm")) ?? undefined;

  return {
    id: crypto.randomUUID(),
    secret,
    title,
    issuer,
    digits,
    period,
    algorithm
  };
}
```

Figure 39. URI parser function

First, the "protocol", "pathname", and "searchParams" properties are extracted from the passed-in URL object. Protocol is assumed to be "otpauth:", so if it is not, an error is thrown. Since the "host" property is required (that is, what contains the "type" parameter described before), it is assumed that it will exist and that its value is going to be "totp". These assumptions are generally safe for a prototype, but in a production-ready application, this should be revisited and changed. Then, "title" and "secret" are extracted. If either of them does not exist, an error is thrown. The previous remark on throwing errors applies here, too. After that, all optional parameters are extracted, with appropriate defaults assigned if they are missing. The "parseSafeNumber" function ensures that the number passed to it is a safe number (in this context, that means a positive, finite number). The hashing algorithm is also checked against a list of supported algorithms (SHA-1, SHA-256, SHA-

384, and SHA-512). Finally, a new UUID is generated for this account, and the result is returned.

The last part of the application worth going in depth on is the account linking functionality. As mentioned previously, scanning a QR code can significantly reduce usability for certain users. For this reason, this application offers both manual entry and QR code scanning. Since not all devices on which this application can run may have a camera to scan QR codes, the application must first check whether a camera is available. If not, then only manual entry will be possible. If it does, the user will be prompted to choose. Figure 40 shows the camera detection logic.

```
const { isOpen, setOpen, action, buttonClassName } = props;
const [isCameraAvailable, setCameraAvailable] = React.useState(initialState: false);
const [choice, setChoice] = React.useState<"qr" | "manual" | null>(initialState: null);

React.useEffect(effect: () => {
  async function isCameraAvailable() {
    try {
      const devices = await navigator.mediaDevices.enumerateDevices();
      return devices.some(predicate: it => it.kind === "videoinput");
    } catch (error) {
      return false;
    }
  }

  isCameraAvailable().then(onfulfilled: it => setCameraAvailable(value: it));
}, deps: [setCameraAvailable]);
```

Figure 40. Camera detection

The above code snippet is from the “add-account-dialog.tsx” file, which contains the “Add” account button, and most of the dialog skeleton used by the QR scanner and the manual entry form dialogs. A media device of type “videoinput” is a camera. If such devices exist, the corresponding “isCameraAvailable” state value will be set to true; otherwise, it will remain false. The “choice” state value is used only when a camera is available, and the user needs to choose how to add an account. Initially, the value is null, indicating that the user has not yet made a selection.

Selecting QR code scanning sets this value to “qr”; similarly, selecting manual entry sets it to “manual”. Figure 41 shows the conditional rendering of the correct dialog based on camera availability and the user’s choice.

```
<DialogContent>
  {/* If choice is null and there is an available camera, user need to pick */}
  {isCameraAvailable && choice === null && (
    <>
      <DialogTitle>Add account</DialogTitle>
      <DialogDescription>Select how you would like to add your account.</DialogDescription>
      <Button onClick={() => setChoice(value: "qr")}>Scan QR code</Button>
      <p className="justify-self-center">or</p>
      <Button onClick={() => setChoice(value: "manual")}>Enter manually</Button>
    </>
  )}

  {/* Camera is not available, force manual entry */}
  {((!isCameraAvailable && choice === null) || choice === "manual") && (
    <ManualEntry action={action} close={close} />
  )}

  {/* Camera is available and has been selected by the user, open QR-code scanner */}
  {isCameraAvailable && choice === "qr" && <QrCodeScanner action={action} close={close} />}
</DialogContent>
```

Figure 41. Rendering of the add account dialog

If a camera is available and the “choice” variable is null, the user has not yet selected how to add the account, so the selection dialog needs to be shown. If the camera is not available and “choice” is null, show the manual entry form. In this case, the selection dialog will never be shown to the user, so “choice” will remain null. If “choice” is set to “manual”, however, the camera is available on the user’s device. However, the user has already selected manual entry in the selection prompt, so the manual entry dialog needs to be shown. Lastly, if the camera is available, and “choice” is set to “qr”, then open the QR code scanner.

4.7 Overview of the Finished Application

The application has been designed to be responsive and support a wide range of screen sizes. The main screen can be divided into two segments: the navbar at the top of the screen, consisting of the application logo, a search bar, and the add account button; and the main view, which shows linked accounts and their data. On devices with screens narrower than 768 pixels, the application logo stays at the top of the screen, while the search bar and the add account button are moved to the bottom for easier access. The main screen showing accounts is also adjusted based on screen size: if it is below 768px, accounts are organized into a single column; otherwise, accounts are shown in a grid with 2 columns on devices with a screen width less than 1024px and 3 columns on devices with a screen width greater than 1024px.

For each account, a card is rendered: the account title is the card's title, the issuer is shown beneath the title, and the code expiry is displayed next to the title. Both the current and upcoming codes are displayed simultaneously, each in its separate box with buttons to copy the code easily. To make codes easier to comprehend, increased letter spacing was applied to them, and they are separated into two blocks of numbers, the size of each block depending on the number of digits in the code: 6 digits are divided into 2 * 3 digits, 7 digits are divided into 3 and 4 digits, and 8 digits are divided into 2 * 4 digits. Figures 42, 43, and 44 show the main application screen on devices with widths greater than 1024 pixels, between 1024 and 768 pixels, and less than 768 pixels, respectively.

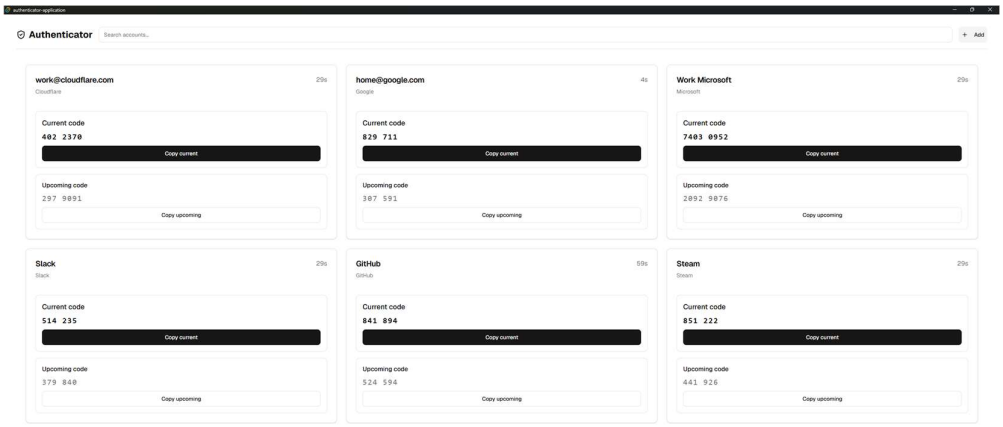


Figure 42. UI on a screen width of 1024 pixels

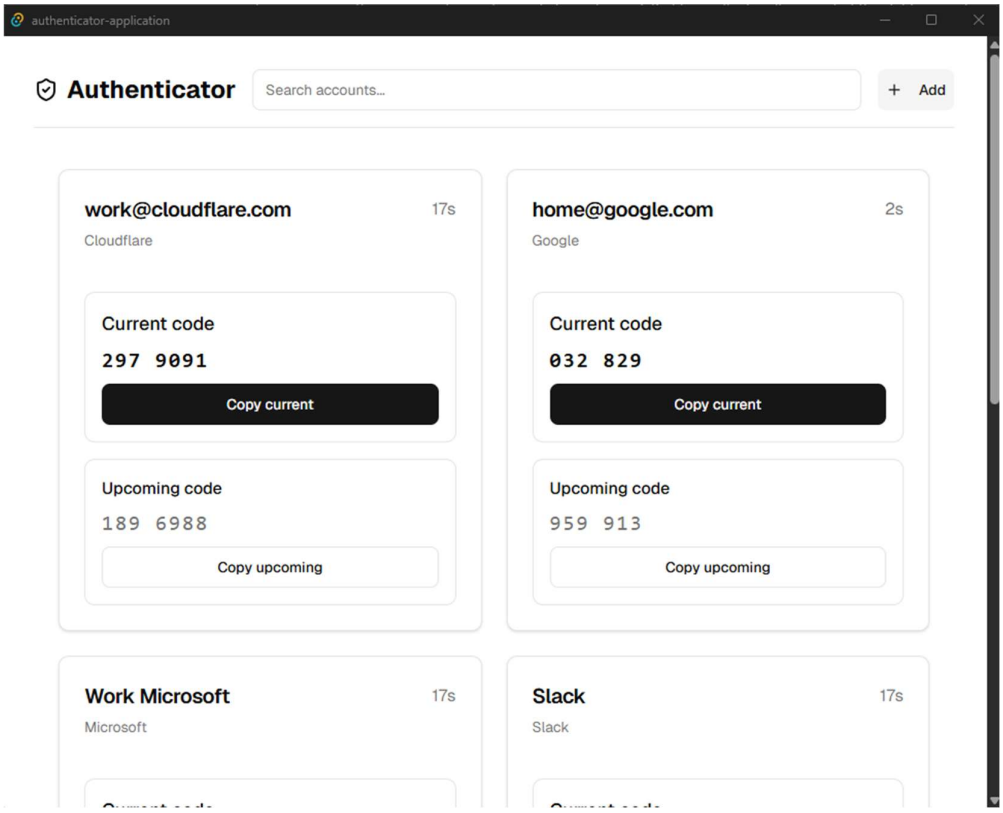


Figure 43. UI on a screen width of less than 1024 pixels

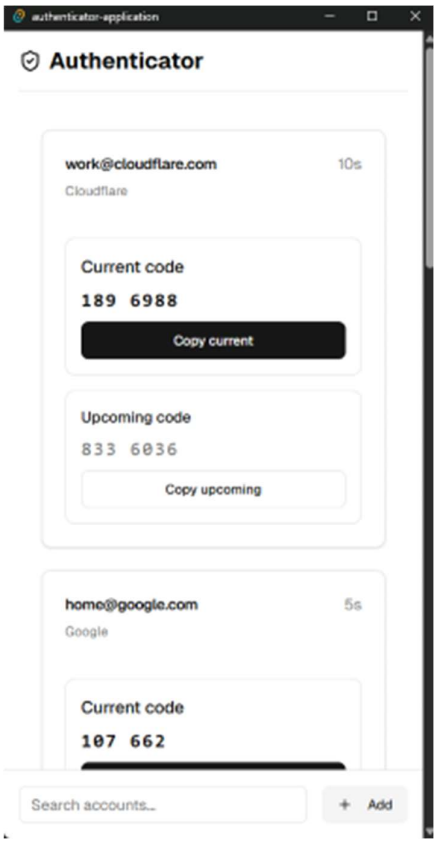


Figure 44. UI on a screen width of less than 768 pixels

The various dialogs for adding accounts remain the same across all devices, although on smaller screens, the manual entry form is scrollable to prevent content from being cut off at the top or bottom of the screen. Because the flow for adding an account was described in the previous chapter, it will not be explained here; instead, screenshots of all relevant dialogs are included. Figures 45, 46, and 47 show the selection dialog, manual entry dialog, and QR scanner dialog (the actual camera view has been obscured for privacy reasons), respectively.

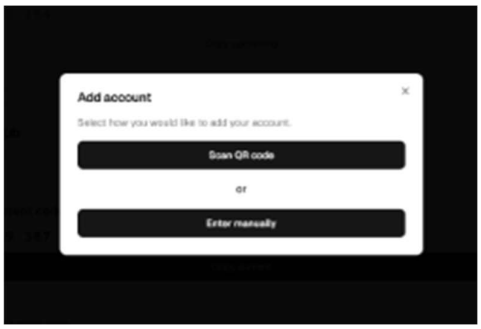


Figure 45. Selection dialog

Figure 46. QR code scanner dialog

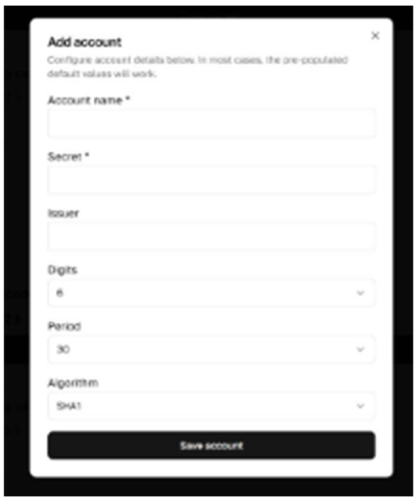


Figure 47. Manual entry dialog

5 EVALUATION OF RESULTS

This chapter discusses the testing and evaluation process of the application after its completion. Both automated and manual testing were conducted, which will be described in detail. After that, the testing findings are summarized, with some concerns highlighted.

5.1 Automated Testing

For automated testing, two tools were used: axe Devtools, an accessibility check for Microsoft Edge, and ESLint, a popular linting library for JavaScript/TypeScript. The latter does not automatically include accessibility-related checks (rules, according to ESLint terminology), so an additional plugin, "eslint-plugin-jsx-a11y", had to be installed. Axe Devtools found a few elements with a "role" attribute, but no "aria-label" or "aria-labelledby" attribute. Such issues were fixed by adding proper labels to these elements. ESLint found only a single issue: an incorrect use of the "autoFocus" attribute. The problem with it is that automatically shifting focus may disorient users, especially those who use assistive technology. This issue was fixed by removing the problematic "autoFocus" attribute.

All in all, these automated tools did not find many issues with the code, and the ones they did find were fixable with little effort. This is likely because HTML semantics have been followed when possible, and proper "aria" attributes have been added to elements where necessary. It is important to note, however, that such static analysis tools only examine the code and cannot analyze it in context. The appropriate guidelines may be followed, but poor document structure can still hinder users who use assistive technology. For this reason, manual testing is also required.

5.2 Manual Testing

Manual testing consisted of two things: checking if the application is operable only using the keyboard (which is a WCAG requirement), and then using it with a screen reader. The application was mostly operable with a keyboard alone, but a few minor adjustments were needed. The "tabIndex" attribute was used more liberally than it should have been. This led to cards and cards within other cards being focusable. The problem with that is that there is no reason for them to be focusable in the first place. Only important, actionable elements should be focusable, for instance, an input field or a button. The default HTML elements already have focus configured, so a "<button>" element, or a custom button that, under the hood, uses the HTML default button, is focusable without extra effort. The tab flow of the application after removing unnecessary "tabIndex" properties is that each tab press moves to the next actionable element: first, the search bar; then the add account button; then the copy current code button for the first account; then the copy upcoming code button for the first account; and so forth. Pressing enter on any of these elements performs the action associated with the specific element. Search in the account list (although technically that is done just by typing), open the add account dialog, or copy the current or upcoming code for an account.

The final, and perhaps most important, test was using the application with a screen reader. The screen reader of choice was NVDA. Screen readers are operated by pressing various keys on the keyboard (which is why keyboard-only operability is very important). "H" navigates to the next header, "B" navigates to the button, and " tab navigates to the next focusable element. In contrast, the arrow keys navigate between elements directly related to each other (parent/child). Pressing the shift key with "H", "B", or tab navigates to the previous header, button, or focusable element, respectively.

Testing with a screen reader revealed that, while the application is operable, the document structure needs improvement: too many headings made navigation cumbersome. Not only that, sometimes labels were misleading or just ambiguous, and important context was missing in some cases. The ideal flow would be that pressing "H" once moves to the next account card, and pressing "B" moves to the copy current code button. The copy upcoming code button, when pressed with the arrow keys in an account card, would read out all necessary information, which is also displayed in the UI. After the initial testing was completed, a few modifications were made to the codebase to improve the experience.

First, the account title heading was updated. The CardTitle Shadcn component, in which the account's title was placed, was a " " element by default, which meant that the jump next header functionality ignored this header completely. This was fixed by setting its role attribute to "heading" and its "aria-level" attribute to 3, which made the component behave as an HTML "<h3>" would. Additionally, a more informative "aria-label" attribute was set, so that the screen reader would read out more meaningful information. Then, the expiry section was updated. In the UI, only the actual expiry number is visible, for example, 30 for 30 seconds. However, the number 30 alone is not very informative, so it was surrounded by screen reader-only "<p>" elements, making the text read out by the screen reader "Expires in 30 seconds". This, combined with the context provided by the header, resulted in better usability; now, moving "down" the DOM with the arrows would read something along the lines of "Example account codes. Expires in 30 seconds". The issuer applied the "aria-hidden" property to hide it from screen readers because it contained only redundant data; the title was sufficient to identify the account clearly.

A new live search announcement field was also added, only visible to screen readers. The problem with the original search functionality was that the only feedback to the search was visible. If there were matching accounts, the UI was updated accordingly; otherwise, an appropriate

error message was displayed to the user. However, the screen reader read this error message aloud only if the user pressed the tab key after searching. However, because the search input was live, there was no non-visual feedback that the search had occurred, making it unlikely that users would press the tab key, thereby hiding the error feedback from them. The search announcement is implemented to provide automatic feedback as the user types. Either a meaningful success message (for instance, "15 accounts were found") or a "no accounts were found" message is announced. To make the screen reader actually read out the feedback, the aria-live attribute was set to polite. This way, if the element's content changes, the screen reader announces it to the user, and this announcement does not interrupt the user (a more forceful option, called "assertive", exists, but it was not necessary in this case).

A new screen reader-only heading was added to the application that provides feedback on the number of accounts loaded at startup. The screen reader's virtual cursor is initially positioned at the "Authenticator" heading. Moving it forward with the "H" button will immediately take it to this new heading, prompting the screen reader to read "15 accounts in total" to the user. This is vital because there is no other non-visual indication in the application about the number of linked accounts.

The manual entry describes which account parameters are required and explains that the pre-populated default values are sufficient in most cases. However, since this was contained in a DialogDescription component, which uses an HTML "<p>" tag, the screen reader would only read it out if the user was navigating with the arrow keys. To fix that, this description received the "tabIndex" property with a value of 0. This way, pressing tab in the manual entry dialog would jump to the description field first, preventing the user from accidentally skipping the instructions.

Both the QR scanner component and the manual entry form provided feedback on adding the account by displaying the new account card in the grid. That is visual feedback; however, when adding accounts using a screen reader and the keyboard, there was virtually no feedback. To fix this, Shadcn's toast functionality was used to show an "Account added successfully" message. Because the screen reader automatically reads such toast messages, this provides enough feedback.

Finally, the "Current code" and "Upcoming code" headings were changed to regular HTML "<p>" tags. This meant that the "<p>" tag had to receive extra styling to appear like headings. The reason for this change is that the presence of these headings disrupted the previously discussed navigation flow and only provided redundant information to the user.

5.3 Assessment of Results

With the changes outlined in chapters 5.1 and 5.2, the original goals have been achieved. The application is accessible to a reasonable extent; all implemented features are usable by various user groups, including those who rely on assistive technology. The common issues with the authenticator application described in Chapter 3.1 have also been addressed. The application runs on a variety of devices, so users can install it on the devices they usually use for authentication, eliminating the distraction of switching between multiple devices. The application also supports manual entry of account data, so if someone struggles with QR codes, they have a usable alternative. Upcoming codes are also displayed next to the currently active codes, so users can copy them instead of the current ones and have more time to paste them on the website. Lastly, while the application uses a few icons, there is always a clear label or description next to them, providing necessary context.

However, there are a few things to take into account. Perhaps the most important consideration is the development team's limited experience with assistive technology. While manual testing provided useful data about the applications usability and accessibility, which was then used to improve the overall experience for all kinds of groups of users using the application, people using assistive technology such as screen readers on a day-to-day basis could most likely supply additional insight into how this application could be further improved, and about any shortcomings, that the development team could not identify.

Another important consideration is that the application can be as usable and accessible as possible, but it cannot make the authentication process completely accessible. If, for instance, the website the user is authenticating on is not generally accessible, the authentication process may still fail because of the website. For example, the user might be able to open this application and copy the code in time. However, if the website does not support proper keyboard navigation, the user might be able to paste the copied code into it. This means that, to make authentication fully accessible, all parties involved have emphasized accessibility, as no single party can fix the shortcomings of the others.

This application is merely a prototype, and although it proved to be usable and the outlined goals were achieved, certain modifications would be required before releasing it to the public. Perhaps the most important of them all is localization support, which is crucial for accessibility; information needs to be presented to the user in a language that they understand. However, this application presents everything in English, excluding people who do not understand it. Two additional accessibility-related improvements would be high-contrast mode and adjustable font size.

A few other future improvement ideas concern usability and security. A production-grade application must have better error handling; this was explained in the Implementation chapter, where applicable. Account

updating and deletion would also improve the application's overall usability; for example, if there is a typo in an account title, there is currently no way to fix it, nor is updating or deleting supported. Another common feature of authenticator applications that this prototype lacks is the ability to import and export account data. This is useful when the user migrates between applications. Exporting account data enables the user to import it into another application, while the import feature allows the user to import data originally exported from another authenticator application. Finally, security should also be improved. At the moment, there is no password or biometric protection on the application, which would be necessary in a production-ready application.

6 CONCLUSION

This thesis aimed to explore the usability and accessibility challenges of MFA and authenticator applications more specifically. It also set out to provide a usable and accessible alternative to current solutions.

To achieve this, the currently adopted MFA solutions were first examined from the perspectives of usability and accessibility. Then, relevant studies and surveys were analyzed to gather more information about the shortcomings of current authentication flows and authenticator applications. During this process, a few key issues have been highlighted that plague authenticator applications.

Using the gathered data, a prototype application was planned to address these concerns. The tools, libraries, and frameworks to be used in the application were selected for their ability to improve accessibility, where applicable. Then the application architecture was designed and presented. After that, the implementation process began by first detailing the installation of project dependencies and the initial setup of the selected tools. Then, both the frontend and the backend implementation were explained in depth, providing reasoning for the various implementation choices when necessary. Finally, an overview of the finished application was presented. It was demonstrated that the application supports multiple screen sizes well.

After finishing the initial implementation of the application, both automated and manual testing and evaluation were conducted. Various issues identified by automated tools and during manual testing have been fixed to improve usability and accessibility. Finally, concerns regarding the reliability of the conducted tests and potential future improvements have been outlined.

Overall, the application did achieve the original goals. The UI meets the requirements identified in the analysis of existing studies, and most usability and accessibility concerns have also been mitigated.

REFERENCES

Acemyan, C. Z., Kortum, P. T., Xiong, J., & Wallach, D. S. (2018). *2FA might be secure, but it is not usable: A summative usability assessment of Google's two-factor authentication (2FA) methods.* In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 62*(1), 1141–1145. https://doi.org/10.1177/1541931218621262

Amadeo, R. (2018, February 27). *Google starts a push for cross-platform app development with Flutter SDK*. https://arstechnica.com/gadgets/2018/02/google-starts-a-push-for-cross-platform-app-development-with-flutter-sdk/

CISA. (n.d.). *Multi-factor authentication*. https://www.cisa.gov/topics/cybersecurity-best-practices/multifactor-authentication

Cloudflare. (n.d.). *What is two-factor authentication?*. https://www.cloudflare.com/learning/access-management/what-is-two-factor-authentication/

Cuprik, R. (2025). *SIM swapping exposed: What is it and how to stay safe?* ESET. https://www.eset.com/blog/en/home-topics/privacy-and-identity-protection/sim-swapping-exposed-stay-safe/

Cybersecurity411. (2025, February 13). *The role of hardware security keys: Why USB and NFC-based MFA are gaining popularity*. https://cybersecurity.industry411.com/2025/02/13/the-role-of-hardware-security-keys-why-usb-and-nfc-based-mfa-are-gaining-popularity

D'Andrea, A. (2025). *What is a hardware security key and how does it work?.* Keeper Security. https://www.keepersecurity.com/blog/2023/05/09/what-is-a-hardware-security-key-and-how-does-it-work/

de Fremery, R. (2021). *The evolution of multi-factor authentication.* LastPass. https://blog.lastpass.com/posts/the-evolution-of-multi-factor-authentication

Di Campi, A. M., & Luccio, F. L. (2025). *Accessible authentication methods for persons with diverse cognitive abilities. Universal Access in the Information Society, 24*, 2195–2217. https://doi.org/10.1007/s10209-025-01189-4

Electron. (n.d.). *What is Electron?*. https://www.electronjs.org/docs/latest/

FBI. (n.d.). *Spoofing and phishing.* https://www.fbi.gov/how-we-can-help-you/scams-and-safety/common-frauds-and-scams/spoofing-and-phishing

Flutter. (n.d.). *Flutter on desktop*. https://flutter.dev/multi-platform/desktop

Fruhlinger, J., & CSO Staff. (2024). *Two-factor authentication (2FA) explained: How it works and how to enable it*. CSO Online. https://www.csoonline.com/article/563753/two-factor-authentication-2fa-explained.html

Git. (n.d.). *Getting started – about version control*. https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

GitHub. (n.d.). *Verifying with GitHub Mobile*. https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/accessing-github-using-two-factor-authentication#verifying-with-github-mobile

Google. (n.d.). *Sign in with Google prompts*. https://support.google.com/accounts/answer/7026266

Goretsky, A. (2025). *Understanding phishing: definition and tactics.* ESET. https://www.eset.com/blog/en/what-is/understanding-phishing-definition-and-tactics/

IBM. (n.d.). *What is a push notification?*. https://www.ibm.com/think/topics/push-notifications

International Organization for Standardization. (2018). *ISO 9241-11:2018 ergonomics of human-system interaction — Part 11:*

Usability: Definitions and concepts. https://www.iso.org/standard/63500.html

Kaczorowski, M. (2025, August 13). *The evolution of authentication, from passwords to passkeys.* Oblique Security. https://oblique.security/blog/history-of-authentication/

Kim, J. (2025). *The pros and cons of different MFA methods.* Keeper Security. https://www.keepersecurity.com/blog/2025/03/31/the-pros-and-cons-of-different-mfa-methods/

Kotlin. (2025, November 18). *What is Kotlin Multiplatform*. https://kotlinlang.org/docs/multiplatform/kmp-overview.html

Kumar, S. (2024). *Accessibility challenges with digital security solutions*. Digitala11y. https://www.digitala11y.com/accessibility-challenges-with-digital-security-solutions/#multifactor-authentication-mfa

Leahy, M., Olivera, G. (2024). *Accessibility considerations for authentication experiences*. CapTech. https://www.captechconsulting.com/articles/accessibility-considerations-for-authentication-experiences

Lisowski, T. (2025, December 30). *Top Git hosting service for 2026*. GitProtect. https://gitprotect.io/blog/top-git-hosting-services/

Lumburovska, L., Dobreva, A., Andonov, S., Trpcheska, H. M., & Dimitrova, V. (2021). *A comparative analysis of HOTP and TOTP authentication algorithms. Which one to choose?*. International Scientific Journals. https://stumejournals.com/journals/confsec/2021/4/131

Microsoft. (n.d.). *How number matching works in MFA push notifications for Authenticator - Authentication methods policy.* https://learn.microsoft.com/en-us/entra/identity/authentication/how-to-mfa-number-match

Microsoft. (2024, January 10). *Introduction to .NET*. https://learn.microsoft.com/en-us/dotnet/core/introduction

M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., & Ranen, O. (2005). *HOTP: An HMAC-based one-time password algorithm (RFC 4226)*. Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc4226

M'Raihi, D., Machani, S., Pei, M., & Rydell, J. (2011). *TOTP: Time-based one-time password algorithm* (RFC 6238). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc6238

Moreno, L., Petrie, H., & Schmeelk, S. (2023). *Accessibility barriers with authentication methods for blind and partially sighted people in the Spanish-speaking world. In 36th Annual British HCI Conference* (pp. 189–198). BCS Learning and Development Ltd. https://doi.org/10.14236/ewic/BCSHCI2023.22

NIST. (n.d.-a). *How do I create a good password*. https://www.nist.gov/cybersecurity/how-do-i-create-good-password

NIST. (n.d.-b). *Multi-factor authentication*. https://www.nist.gov/itl/smallbusinesscyber/guidance-topic/multi-factor-authentication

Radix. (n.d.). *Accessibility*. https://www.radix-ui.com/primitives/docs/overview/accessibility

Shadcn. (n.d.). *Introduction*. https://ui.shadcn.com/docs

SQLite. (n.d.). *SQLite home page*. https://sqlite.org/index.html

TanStack. (n.d.). *Overview.* https://tanstack.com/query/v4/docs/framework/react/overview

Tauri. (2025, August 1). *What is Tauri?*. https://tauri.app/start/

Temoshok, D., Fenton, J. L., Choong, Y., Lefkovitz, N., Regenscheid, A., Galluzzo, R., Richer, J. P., (2025)., *Digital identity guidelines: Authentication and lifecycle management (NIST Special Publication 800-63B-4).* https://doi.org/10.6028/NIST.SP.800-63B-4

Trevino, A. (2025). *What are authenticator apps and how do they work?*. Keeper Security.

https://www.keepersecurity.com/blog/2023/07/20/what-are-authenticator-apps-and-how-do-they-work/

United Nations. (2006). *Convention on the Rights of Persons with Disabilities.* https://www.ohchr.org/en/instruments-mechanisms/instruments/convention-rights-persons-disabilities

U.S. General Services Administration. (2024). *Accessibility bytes: QR codes*. https://www.section508.gov/blog/accessibility-bytes/qr-codes/

VocalEyes. (n.d.). *Digital accessibility: QR codes and short number SMS*. https://vocaleyes.co.uk/research/digital-accessibility-qr-codes-and-short-number-sms/

World Health Organization. (2011). *World report on disability 2011*. https://iris.who.int/handle/10665/44575

World Wide Web Consortium. (2024). *Web Content Accessibility Guidelines (WCAG) 2.2*. https://www.w3.org/TR/WCAG22/

World Wide Web Consortium. (2025). *Guidance on applying WCAG 2 to on-web information and communications technologies (WCAG2ICT)*. https://www.w3.org/TR/wcag2ict-22/

World Wide Web Consortium Web Accessibility Initiative. (2025, July 15). *Introduction to understanding WCAG 2.2*. https://www.w3.org/WAI/WCAG22/Understanding/intro

World Wide Web Consortium Web Accessibility Initiative. (2026, February 3). *Introduction to web accessibility.* World Wide Web Consortium. https://www.w3.org/WAI/fundamentals/accessibility-intro/

World Wide Web Consortium Web Accessibility Initiative. (2025, September 22). *How to meet WCAG (Quick Reference)*. https://www.w3.org/WAI/WCAG22/quickref/