

Diplomarbeit



Shulker

Ein intelligentes Türschloss für Privatanwender

erstellt von

Alexander Heim
Moritz Laichner



HTBLuVA
Innsbruck Anichstrasse

Betreuer:
Sabo Rubner

2021/22

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Diplomarbeit eingereicht.

Innsbruck, am 08. März 2022

Verfasser/Verfasserinnen:

Alexander Heim

Moritz Laichner

Projektteam



Alexander Heim

Adresse

PLZ Ort

Tel: -

E-Mail: -



Moritz Laichner

Adresse

PLZ Ort

Tel: -

E-Mail: -

Betreuer



Bakk. Sabo Rubner

HTBLuVA Innsbruck Anichstraße

E-Mail: sabo.rubner@htlinn.ac.at

Danksagung

toDO An dieser Stelle möchten wir danke sagen.

Moritz Laichner & Alexander Heim

Innsbruck, 08. März 2022

Gendererklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit durchwegs die Sprachform des generischen Maskulinums angewendet. An dieser Stelle wird darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Sprachform geschlechtsunabhängig verstanden werden soll.

Abstract

The Internet of Things is becoming a larger part of our world day by day. Big name companies often manufacture and develop devices with ease of use in mind, often sacrificing a lot of extensibility and hackability.

Shulker aims to be a software solution that does not sacrifice flexibility for ease of use. This allows for Do-It-Yourself enthusiasts and hackers alike to control access with maximum customizability and an easy to expand on system.

The finished project is a full-on solution providing a Smartphone-App, an API-Server, a fletched out Interface and a fully functioning showpiece.

Shulkers software is fully open source and licensed under the GNU General Public License version 3.

Zusammenfassung

Das Internet der Dinge wird von Tag zu Tag ein größerer Teil unserer Welt. Große Unternehmen entwickeln ihre Produkte oft mit einem starken Fokus auf Benutzerfreundlichkeit, wobei sie meist eine Menge an Erweiterbarkeit und Flexibilität opfern.

Shulker zielt darauf ab, eine Softwarelösung zu sein, die Flexibilität nicht für Benutzerfreundlichkeit eintauscht. So können Do-It-Yourself-Enthusiasten und Hacker das System auf ihre eigenen Anforderungen anpassen und leicht auf das System aufbauen.

Das fertige Projekt ist eine vollwertige Lösung, die eine Smartphone-App, einen API-Server, ein ausgefeiltes Interface und ein voll funktionsfähiges Ausstellungsstück bietet.

Der Quellcode von Shulker ist vollständig Open Source und unter der GNU General Public License version 3 lizenziert.

Inhaltsverzeichnis

I. Intro	12
1. Meta	13
1.1. Hintergrund	13
1.2. Namensherkunft	13
2. Aufbau	14
2.1. Übersicht des Systems	14
2.2. Testumgebung	14
II. Theoretische Grundlagen	15
3. Verwendete Technologien	16
3.1. Rust	16
3.1.1. Warum Rust?	16
3.1.2. Geschichte von Rust	16
3.1.3. Programmierparadigmen	17
3.1.4. Ownership-Model	17
3.1.5. Scope von Variablen	18
3.1.6. Der Drop Trait	19
3.1.7. Immer nur ein Owner	19
3.1.8. move	21
3.1.9. clone	21
3.1.10. Ownership und Funktionen	22
3.1.11. Mutabilität und mutable Referenzen	23
3.1.12. Der Cargo Packagemanager	24
3.1.13. Crates.io	24
3.2. C-Sharp	24
3.2.1. Warum C-Sharp?	25
3.2.2. .NET-Architektur	25
3.2.3. Common Language Runtime	25
3.2.4. ASP.NET Core	25
3.3. Flutter	25
3.3.1. Dart	26
3.3.2. Architektur des Flutter-Frameworks	26
3.3.3. Widgets in Flutter	26
3.3.4. Arten von Flutter Widgets	26
3.3.5. Verwendete Flutter-Pakete	28

Inhaltsverzeichnis

III. Shulker-Core	30
4. Überblick	31
4.1. Allgemeines	31
4.2. Funktionsweise und Aufbau	31
4.3. Konfiguration	32
4.3.1. Mögliche Konfigurationen	33
4.4. Abhängigkeiten	34
5. Der Touchscreen	35
5.1. Funktionsweise von slint	35
5.2. Aufbau der grafischen Oberfläche	35
5.2.1. Menü-Selektor	36
5.2.2. Hauptansicht	36
5.2.3. Unlocked-Ansicht	36
5.3. Die Hardware	37
5.3.1. Vorzeigemodell	37
5.3.2. Mögliche Displays	37
6. Speicherung von Daten	38
6.1. Rustbreak	38
6.2. Funktionsweise	38
6.3. Hashing von Pins	38
6.3.1. Argon2	39
IV. Shulker-Connect	40
7. Allgemeines	41
7.1. Authentifizierung am Server	41
7.2. Verschlüsselung der Anfragen von Shulker-Mobile zum API-Server	41
8. Kommunikation mit Shulker-Core	42
8.1. Was sind POSIX-Sockets?	42
8.2. Aufbau der POSIX-Nachrichten	42
8.3. Ablauf einer Anfrage zu Shulker-Connect	43
8.4. Beispiel	44
8.5. Routen der Interprozesskommunikation	44
8.5.1. CreatePin	44
8.5.2. UseMaster	45
8.5.3. Lock	45
8.5.4. Unlock	45
8.5.5. GetPins	46
8.5.6. DeletePin	46
8.5.7. Status	46
9. Routen der REST-API	47
9.0.1. /api/Credentials	47
9.0.2. /api/Credentials/createPin	48

Inhaltsverzeichnis

9.0.3. /api/Credentials/deletePin/uuid	49
9.0.4. /api/Lock/isLocked	49
9.0.5. /api/Lock/setLockState	50
9.0.6. /api/Session/getToken/secret	50
9.0.7. /api>Status	51
10. Routen der Interprozesskommunikation	52
V. Shulker-Mobile	53
11. Allgemeines	54
11.1. Verbinden des Türschlosses	54
11.1.1. Türschloss manuell verbinden	55
11.1.2. Türschloss mittels QR-Code verbinden	55
11.1.3. Verbindung außerhalb des Lokalen Netzwerkes	55
11.1.4. Authentifizierung bei Start der App	57
11.1.5. Startbildschirm	58
11.1.6. Sidebar	58
11.1.7. PIN-Verwaltung	59
11.1.8. Erstellen eines Pins	61
11.1.9. Einstellungen	64
VI. Vorzeigemodell	68
12. Allgemeines	69
12.1. Planung des Grundgerüsts	69
12.2. Planung der elektronischen Schaltung	70
13. Tür und Box	71
13.1. Herstellung der einzelnen Teile	71
13.1.1. Türblatt	71
13.1.2. Türrahmen	71
13.1.3. Holzbox	71
13.2. Vollendung der Modelltür	72
13.3. Vollendung der Holzbox	73
14. Elektronische Schaltung	74
14.1. Das Schloss	74
14.2. Die Schaltung	74
15. Fertigstellung	76
15.1. Zusammenführung	76
15.2. Das Endprodukt	77

Inhaltsverzeichnis

VII. Appendix	78
16. Appendix	79
16.1. Zeitaufwand	79
16.1.1. Zeitaufwand Alexander Heim	79
16.1.2. Zeitaufwand Moritz Laichner	80
Literaturverzeichnis	82
Abbildungsverzeichnis	84
Code-Snippet-Verzeichnis	86

Teil I.

Intro

1. Meta

1.1. Hintergrund

Das Internet der Dinge wird Tag für Tag größer. Der Einfluss den smarte Geräte in modernen Haushälten haben wird ständig stärker. Die meisten Hersteller von smarten Geräten zielen auf Benutzerfreundlichkeit und Einfachheit ab. Die vielen Do-It-Yourself Enthusiasten rücken für sie in den Hintergrund. Einer wichtigsten und am Häufigsten genutzten Mechanismen in einem Haushalt sind Schlosser. Da nicht jeder der selbst gerne Sachen in die Hand nimmt die Fähigkeiten oder Zeit besitzt, ein sicheres und smartes Türschloss von Grund auf zu entwerfen und aufzubauen, haben wir uns entschieden, ein intelligentes Türschloss zu entwickeln.

1.2. Namensherkunft

Der Name **Shulker** stammt ursprünglich aus dem Videospiel *Minecraft*. In Minecraft ist ein Shulker eine Kreatur, die Verschlossen ist.



Abbildung 1.1.: Die Kreatur **Shulker** aus dem Videospiel *Minecraft*
[1]

2. Aufbau

2.1. Übersicht des Systems

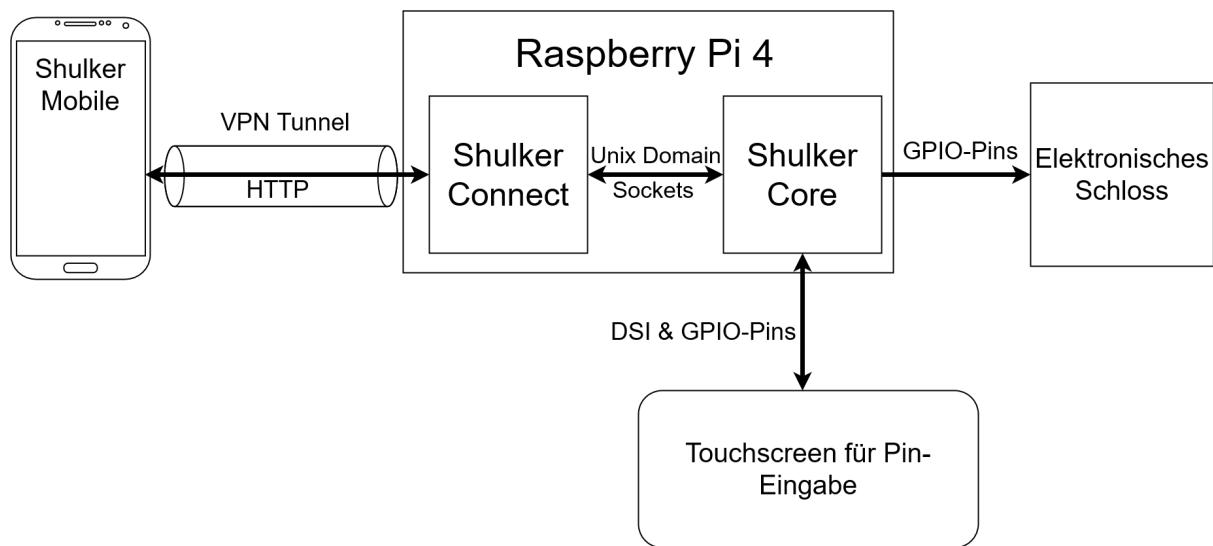


Abbildung 2.1.: Aufbau von **Shulker**

Das Projekt setzt sich aus folgenden **drei** Komponenten zusammen:

- **Shulker-Mobile:** Shulker-Mobile ist eine Smartphone-App, mit der das Türschloss verwaltet werden kann.
- **Shulker-Connect:** Shulker-Connect ist ein auf dem Raspberry-Pi laufender Webserver, der als Schnittstelle für die Kommunikation zum Türschloss dient. Shulker-Mobile kommuniziert über einen IPsec VPN-Tunnel mit Shulker-Connect.
- **Shulker-Core:** Shulker-Core ist eine auf dem Raspberry-Pi laufende Software, die die Benutzeroberfläche des Touchscreens und das elektrische Schloss selbst steuert. Shulker-Core kommuniziert über Unix Domain Sockets mit Shulker-Connect.

Eine genaue Beschreibung der einzelnen Komponenten folgt in späteren Teilen der Diplomschrift.

2.2. Testumgebung

Für Test und Demonstrationszwecke wurde das Türschloss in einer miniatur-Tür eingebaut.

Teil II.

Theoretische Grundlagen

3. Verwendete Technologien

3.1. Rust

Rust ist eine kompilierte Programmiersprache, welche auf mehreren Programmierparadigmen aufbaut. Die Sprache krönt sich damit, dass sie extrem schnelle und sichere Binärprogramme erzeugt. [2] Der gesamte Quellcode des Rust-Ökosystems ist quelloffen. Rust besitzt keinen Garbage-Collector. Arbeitsspeicher wird mithilfe des Ownership-Models verwaltet.

3.1.1. Warum Rust?

Wir haben uns beim Komponent *Shulker-Core*, also bei der Programmierung des Bildschirms und der Hardwareschaltung, sowie der Speicherung von Daten, für Rust entschieden. Diese Entscheidung trafen wir primär aus fünf Gründen:

- Das Ownership-Model und die Einfachheit von nebenläufiger Programmierung in Rust zwingt uns, vor allem im Bereich vom Touchdisplay, zu einer korrekten Lösung und weniger Fehlern
- Vorkenntnisse in *Rust* und dem UI-Toolkit *Slint* erleichtern uns das Arbeiten
- Die hohe Performance des Endprodukts minimiert die Hardwareauslastung
- Die Einfachheit der Kompilierung macht es dem Nutzer leichter
- Da Rust eine Systemprogrammiersprache ist, ermöglicht es hardwarenahe Programmierung trotz hoher Abstraktionen

3.1.2. Geschichte von Rust

Rust entstand im Jahr 2006 als Hobbyprojekt von Graydon Hoare, einem Angestellten bei Mozilla. Im Jahr 2009 begann Mozilla das Projekt zu unterstützen. Die erste offiziell stabile Version (Version 1.0) wurde im Jahr 2015, also ganze neun Jahre danach, veröffentlicht. Im Jahr 2020 wurden 250 Angestellte von Mozilla entlassen. Ein großer Teil des Rust Teams wurde damit entlassen. Das schien eine Bedrohung für die Programmiersprache zu sein. Glücklicherweise wurde im darauffolgenden Jahr die *Rust Foundation* von den Firmen *AWS*, *Google*, *Huawei*, *Microsoft* und *Mozilla* gegründet. Rust wurde in der *Stackoverflow Developer Survey*, einer großen Umfrage für Programmierer, 2016, 2017, 2018, 2019, 2020 und 2021 zur am meisten geliebten Programmiersprache ernannt. Ob das im Jahr 2022 so der Fall bleibt, wird sich zeigen.

3. Verwendete Technologien

3.1.3. Programmierparadigmen

Rust bietet verschiedenste Möglichkeiten, seinen Code zu gestalten. Unter anderem borgt sich die Sprache Ideen von der funktionalen und objektorientierten Programmierung. Auch das nebenläufige Programmieren, unter Rust-Usern als *Fearless Concurrency* bekannt, ist möglich. Man programmiert in Rust primär imperativ, deklaratives Programmieren ist trotzdem auch ein wichtiger Teil der Programmiersprache.

3.1.4. Ownership-Model

Jedes auf einem herkömmlichen Computer ausgeführte Programm muss den eigenen Arbeitsspeicher selbst verwalten. Der Arbeitsspeicher wird in Stack und Heap eingeteilt. [3]

Der Stack funktioniert auf dem *First-In-First-Out*-Prinzip. Man kann das Prinzip mit einem Stapel Teller vergleichen. Will man dem Tellerstapel (also dem Teller-Stack) einen Teller hinzufügen, ist es einfacher, den Teller auf den Stapel dazu zulegen. Will man einen Teller wegnehmen, ist das am einfachsten indem man den obersten Teller wegnimmt. Auf dem Stack kann man also nur einen Wert auf die oberste Stelle abspeichern und eben genau den obersten, also den als letztes hinzugefügten Wert, entfernen. Alle auf dem Stack speicherbaren Daten müssen eine bekannte und konstante Größe aufweisen, also eine genau definierte Anzahl an Bits einnehmen. Zur Speicherung von Daten mit einer zur Kompilierzeit unbekannten oder variablen Größe muss der Heap verwendet werden.

Auf dem Heap herrscht nur minimale Ordnung. Will man Daten auf dem Heap ablegen, muss sich das Computerprogramm einen Teil des Heaps reservieren. Um erst mal einen Teil des Heaps verwenden zu können, muss ein *memory allocator* (ein Teil des Programms) einen nicht verwendeten und groß genug Teilbereich des Heaps finden und als benutzt markieren. Der memory allocator liefert im Anschluss einen Pointer zurück, also eine Speicheradresse, welche auf den Beginn des neu belegten Heapspeicherteils zeigt. Den Pointer kann man dann auf dem Stack speichern, da eine Speicheradresse eine fixe Anzahl an Bits einnimmt. Wichtig: Der Heap erfordert das Freigeben der reservierten Speicherteile, wenn sie nicht mehr in Gebrauch sind. Wird der Speicher nicht befreit, kann es zu Problemen kommen. Zum Beispiel könnte der Arbeitsspeicher bei längerer Ausführung des Programms volllaufen, da immer mehr und mehr Teile des Heaps als benutzt markiert werden.

Bei Sprachen wie *C* und teilweise auch *C++* liegt die Verantwortung des Heaps in den Händen der Programmierer. Der Programmierer muss manuell Heapspeicher anlegen und befreien. Bugs haben so ein leichtes Spiel, da jeder Mensch Fehler macht. Viele höhere Programmiersprachen setzten deshalb zur Vereinfachung und Fehlervermeidung auf einen *Garbage Collector*. Der Garbage Collector scannt regelmäßig nach nicht mehr gebrauchten Speicher. Findet er nicht mehr gebrauchten Speicher, markiert er diesen Teil des Arbeitsspeichers als ungenutzt und ermöglicht so das Wiederverwenden dieses Speicherabschnitts. Ein Garbage Collector hat aber einen großen Nachteil: Er ist ineffizient.

Um dem Problem der Ineffizienz zu entgehen verwendet Rust das *Ownership-Model*. Das Model besteht aus drei Regeln, welche zur *compile time*, also während des Kompilierprozesses, im Quellcode validiert werden. Erfüllt der Rustcode diese Regeln nicht, weist der Compiler auf den Fehler hin und bricht die Kompilation ab. Somit wird das Auftreten von Fehlern in der Speicherverwaltung vollständig behoben. Laut *Google* sind ungefähr 70% aller Sicherheitsbugs

3. Verwendete Technologien

in *Google Chrome* mit inkorrektener Speicherverwaltung verbunden. [4] Das Ownership-Model allein würde einen Großteil dieser Sicherheitsbugs ausschließen.

Die drei Regeln des Ownership-Modells lauten:

- Jeder Wert ist einer Variable zugewiesen, welche man den *Owner* (Besitzer) nennt
- Es kann immer nur einen Owner geben
- Wenn der Owner out-of-scope fällt, wird der zugewiesene Wert gedropped (der Speicher wird freigegeben)

[5]

3.1.5. Scope von Variablen

Der Scope einer Variable ist der Bereich im Quellcode, in dem eine Variable gültig ist. Ein kleines Beispiel:

```
let x: i32 = 10;
```

Hier wird eine Ganzzahl (ein Integer), die 32 Bit an Information einnimmt, angelegt. Die Variable *x* hat also den Wert *10* vom Typ *i32*; Aus dem Beispiel ist der *Scope* von *x* aber nicht ablesbar. Eine Erweiterung mit Scope:

```
{
    let x: i32 = 10;
}
```

Der Scope von *x* liegt hier innerhalb der geschwungenen Klammern. Ein Zugreifen auf *x* außerhalb des Scopes ist nicht möglich. *x* existiert also nur zwischen den Klammern. Um genauer zu sein: *x* existiert von der Deklaration bis hin zum Ende des Scopes:

```
println!("{}", x); // <- Fehler
{
    println!("{}", x); // <- Fehler
    let x: i32 = 10;
    println!("{}", x); // <- Funktional
}
println!("{}", x); // <- Fehler
```

Dieses Verhalten ist in vielen bekannten Programmiersprachen identisch, also was ist nun das **besondere** an Rust? Da ein 32 Bit Integer eine zur Kompilierzeit bekannte und konstante Größe hat, kann *x* einfach auf dem Stack gespeichert werden. Rusts Vorteile werden beim Umgang mit auf dem Heap gespeicherten Daten erst richtig ersichtlich. Als Veranschaulichung werden wir den *String*-Typen verwenden, da er eine variable Größe hat:

```
{
    let s1 = String::from("Shulker");
    let s2 = String::from("HTML-Anichstrasse");
```

3. Verwendete Technologien

```
}
```

Da der String *s2* mehr Zeichen als der String *s1* hat, braucht *s2* auch mehr Speicherplatz. Strings können auch zur Laufzeit verändert werden. Man könnte zum Beispiel dem *Shulker*-String eine zufällige Zahl hinzufügen. Das verändert natürlich den Speicherbedarf zur Laufzeit. Somit ist klar, dass Strings auf dem Heap abgelegt werden müssen.

3.1.6. Der Drop Trait

Wie es in der letzten Regel des Ownership-Modells beschrieben wird, werden Variablen die out-of-scope fallen gedropped. Gedropped bedeutet, dass Rust eine Funktion namens *drop* ausführt, welche den Speicher der out-of-scope gefallenen Variable befreit. Die *drop*-Funktion wurde in unserem Fall von den Entwicklern vom String-Typ implementiert. Rust regelt also, wann eine Variable gedropped werden kann und sollte. Der Programmierer muss sich keine Gedanken darum machen, den Heap-Speicher zu befreien.

3.1.7. Immer nur ein Owner

Eine Regel des Ownership-Models besagt, dass ein Wert immer nur einen Owner zugeteilt sein darf. Welche Auswirkungen das auf unseren Code hat, kann anhand von wenigen einfachen Beispielen veranschaulicht werden:

```
let a = 10;
let b = a;
```

Im obigen Beispiel ist die Variable *a* der Owner von den Daten mit dem Wert *10*. In der zweiten Zeile sollte dann die Variable *b* der Owner vom Wert in der Variable *a* werden, oder? Wäre das der Fall, würde die zweite Regel des Ownership-Models gebrochen werden. Es würde zwei Owner vom gleichen Speicherort zur gleichen Zeit geben, *a* und *b*.

3.1.7.1. Der Copy Trait

Da der Wert 10 einfach kopiert werden kann, wird er auch kopiert. *b* wird also nicht der Owner von den gleichen Daten wie *a*, sondern von einer Kopie der Daten in *a*. Das obige Beispiel wäre also equivalent zu:

```
let a = 10;
let b = 10;
```

In Rust nennt man dieses Verfahren des einfachen Kopierens *copy*. Wird ein Wert kopiert, so ruft Rust die Funktion *copy* auf. Das ist aber nur möglich, wenn der zu kopierende Wert den Copy-Trait besitzt. Grundsätzlich müssen Daten nur eine Voraussetzung erfüllen, um den Copy-Trait besitzen zu dürfen: Der Wert muss vollständig kopierbar sein.

3. Verwendete Technologien

3.1.7.2. Das Problem mit Copy

Das obige Beispiel hat auf dem Stack gespeicherte, einfach zu kopierende Werte verwendet. Im folgenden Beispiel werden auf dem Heap gespeicherte, nicht ganz so einfach zu kopierende Daten verwendet. Strings um genau zu sein:

```
let a = String::from("Shulker");
let b = a;
```

Man könnte jetzt denken, dass der String *a* einfach mittels copy zu *b* kopiert wird. Das würde aber die Ownership-Regel brechen, dass es nur einen Owner zur gleichen Zeit geben darf. Auf den ersten Blick wird dieser Umstand nicht ganz klar. Dafür muss man sich anschauen, wie Strings in Rust aufgebaut sind:

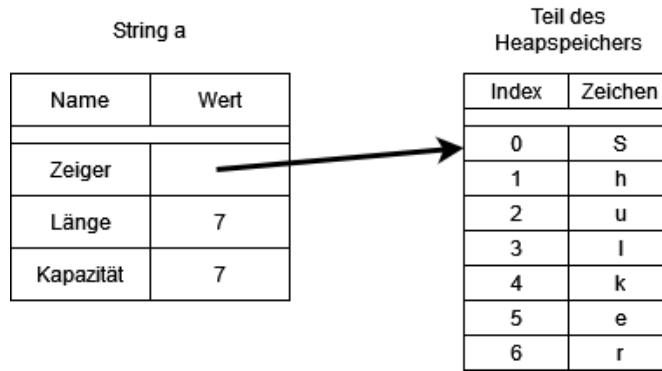


Abbildung 3.1.: Veranschaulichung eines Rust-Strings

Würde man den Copy-Trait nutzen, also einfach Bit für Bit kopieren, würde folgende Situation daraus resultieren:

3. Verwendete Technologien

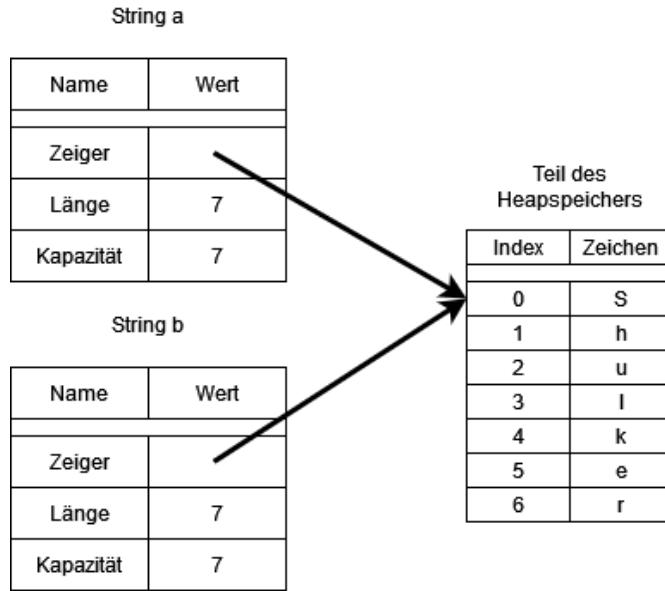


Abbildung 3.2.: Bild zur Veranschaulichung eines kopierten Rust-Strings

3.1.8. move

Wie man sieht, würden beide Variablen auf die gleichen Daten zeigen. Es würde also wieder zwei Owner zur gleichen Zeit geben. Die Regeln des Ownership-Modells würden nicht eingehalten werden. Um dieses Problem zu umgehen, wird die Variable a als nicht mehr gültig angesehen. Wichtig: a wird nicht gedropped, sonst würde b auf nicht existente Daten zeigen. Das nennt man *move*:

```
let a = String::from("Shulker");
let b = a;
println!("{}", a); // <- Fehler
println!("{}", b); // <- funktional, a wurde in b "gemoved"
```

Shallow Copy ist eine wichtige Begrifflichkeit in vielen Sprachen. *Shallow copy* ist dem Verhalten von *move* sehr ähnlich. Einen großen Unterschied gibt es jedoch: Die Invalidierung der ersten Variable. Dieses Verhalten existiert so nur bei *move*.

3.1.9. clone

Will man also korrekt den String a kopieren, verwendet man *clone*. Clone ist im Grunde genommen das gleiche wie eine *deep copy*. Es werden nicht nur die Daten auf dem Stack kopiert, sondern auch die auf dem Heap:

```
let a = String::from("Shulker");
let b = a.clone();
println!("{}", a); // <- funktional
println!("{}", b); // <- funktional
```

3. Verwendete Technologien

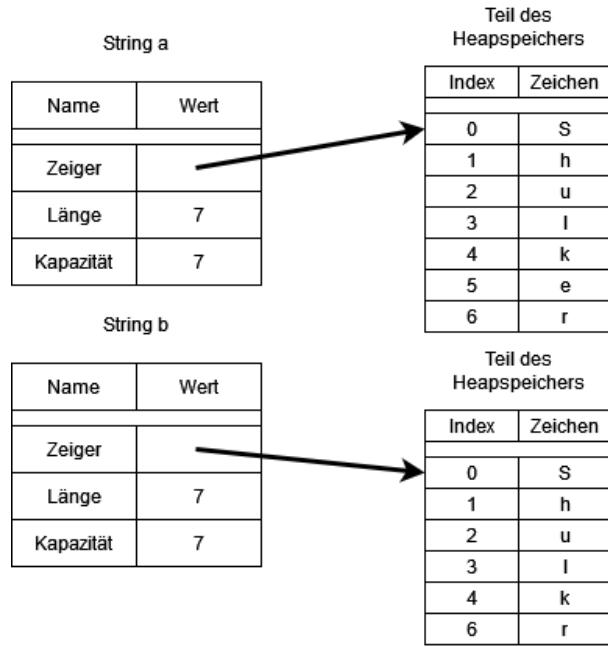


Abbildung 3.3.: Bild zur Veranschaulichung eines geklonten Rust-Strings

3.1.10. Ownership und Funktionen

Eine Funktion ist ein eigener Scope. Daraus resultiert sich, dass Variablen am Ende einer Funktion out-of-scope fallen und gedropped werden müssen:

```
fn sag_hallo(name: String) {
    println!("Hallo {}", name);
} // <- name wird gedropped

fn main() {
    let mein_name = String::from("Alexander");
    sag_hallo(mein_name);
}
```

Es gibt hierbei ein Problem. Man kann die Variable *mein_name* nicht wiederverwenden:

```
fn sag_hallo(name: String) {
    println!("Hallo {}", name);
} // <- name wird gedropped

fn main() {
    let mein_name = String::from("Alexander");
    sag_hallo(mein_name);
    println!("Bis spaeter {}", mein_name);
```

3. Verwendete Technologien

```
// ^ Fehler, da mein_name gedropped wurde
}
```

Anzumerken ist hierbei jedoch, dass Variablen welche den copy trait besitzen, also nicht gemoved werden müssen, einfach kopiert werden. Das Problem gibt es also nur, wenn man komplexere Datentypen einer Funktion als Parameter übergeben will.

Es gibt grundsätzlich zwei Lösungswege:

Man könnte die übergebene Variable wieder zurückgeben:

```
fn sag_hallo(name: String) -> String {
    println!("Hallo {}", name);
    return name;
//   ^ name wird zurueckgegeben und nicht gedropped
}

fn main() {
    let mein_name = String::from("Alexander");
    let mein_name = sag_hallo(mein_name);
//   ^ Rueckgabewert wieder in mein_name uebergeben
    println!("Bis spaeter {}", mein_name);
}
```

Der elegantere Lösungsweg macht sich Referenzen zu nutze. Referenzen sind in Rust so ähnlich wie Pointer, nur dass Referenzen garantieren, dass die Daten auf die gezeigt wird noch valide sind. Referenzen werden durch das Zeichen & gekennzeichnet. Die Lösung ist somit um einiges angenehmer:

```
fn sag_hallo(name: &String) {
    println!("Hallo {}", name);
} // Referenz wird gedropped, Wert aber nicht

fn main() {
    let mein_name = String::from("Alexander");
    sag_hallo(&mein_name);
    println!("Bis spaeter {}", mein_name); // funktioniert
}
```

Referenzen sind ein riesiger Bestandteil von Rust und somit auch des Shulker-Quellcodes.

3.1.11. Mutabilität und `mutable` Referenzen

Alle Variablen in Rust sind grundsätzlich unveränderbar. Um den Wert einer Variable dennoch zu verändern, muss diese als *mutable* gekennzeichnet werden. Dafür gibt es das *mut* Keyword:

```
let mut a = 10;
```

3. Verwendete Technologien

Will man den Wert durch eine Referenz verändern, muss die Referenz und der Owner als veränderbar gekennzeichnet werden:

```
fn veraendern(name: &mut String) {
    name.push('a');
} // Referenz wird gedropped, Veraenderung bleibt

fn main() {
    let mut mein_name = String::from("Alexander");
    // ^ Achtung: mein_name muss mutable sein
    veraendern(&mut mein_name);
    // mein_name ist nun "Alexandera"
}
```

Zu beachten ist:

- Es kann entweder eine mutable Referenz oder unendlich viele immutable Referenzen zur gleichen Zeit geben
- Jede Referenz muss valide sein
- Referenzen dürfen nicht länger als der Owner leben

Um den Quellcode von Shulker-Core nachvollziehen zu können, ist es essentiell, die bereits genannten Konzepte zu verstehen.

3.1.12. Der Cargo Packagemanager

Cargo ist Rusts Packagemanager. Cargo übernimmt das Herunterladen von Packages (Codebibliotheken), das Kompilieren von Quellcode und weiteres. Er ist der Hauptbestandteil des Rust Ökosystems. Cargo lässt sich leicht mit *Rustup* installieren (Rusts offiziellem Installer). Shulker-Core macht starken Gebrauch von Cargo. Die Installation und Nutzung ist besonders simpel, was das Aufsetzen des Raspberry Pis vereinfacht.

3.1.13. Crates.io

crates.io ist die offizielle Sammlung von Rust-Packages, auch *crates* genannt. Cargo greift standardmäßig auf diese Sammlung zu, um benötigte Packages zu installieren und in die Kompilierte Datei einzubinden.

3.2. C-Sharp

C-Sharp ist eine typsichere objektorientierte Allzweck-Programmiersprache. In C-Sharp geschriebene Anwendungen können mittels .NET, CLR und bestimmten *Klassenbibliotheken* ausgeführt werden. Da der C-Sharp Syntax ähnlich zu anderen Sprachen in der C-Sprachfamilie ist, werden die meisten Programmierer keine großen Schwierigkeiten haben, C-Sharp Programmcode zu schreiben bzw. zu lesen. [6]

3. Verwendete Technologien

3.2.1. Warum C-Sharp?

Wir haben uns für C-Sharp entschieden, da C-Sharp über das *ASP.NET Core* Web-Framework eine elegante Lösung bietet, dynamische Webservices zu erstellen.

Die Ausführung von C-Sharp durch *.NET-Core* ermöglicht eine Plattformübergreifende Entwicklung.

3.2.2. .NET-Architektur

Das .NET-Framework (bis 2020 .NET-Core) ist eine freie und offene Entwicklungs-Plattform. Mit .NET können eine breite Menge an Applikationen geschrieben werden, diese können Plattformunabhängig auf diversen Betriebssystemen und Architekturen ausgeführt werden. [7]

3.2.2.1. Programmiersprachen

.NET unterstützt drei Programmiersprachen:

- **C#**: C-Sharp ist eine einfache und moderne Allzweck-Programmiersprache.
- **F#**: F-Sharp ist eine schnell zu schreibende und performante Programmiersprache.
- **Visual Basic**: Visual Basic ist eine Programmiersprache mit einfacherem Syntax, für die keine neuen Funktionen entwickelt werden.

3.2.3. Common Language Runtime

Die virtuelle Laufzeitumgebung von .NET heißt **Common Language Runtime** (CLR), diese wird von Microsoft bereitgestellt und führt den Programmcode aus. CLR ist eine direkte Implementierung Microsofts der *Common Language Infrastructure* (CLI). Die Common Language Infrastructure ist ein internationaler Standard, der sowohl eine Programmiersprachen- als auch Plattformneutrale Entwicklung und Ausführung von Anwendungen ermöglicht.

3.2.4. ASP.NET Core

ASP.NET Core ist ein offenes Plattformunabhängiges Web-Framework, das es einem erlaubt, schnelle und moderne Webanwendungen zu entwickeln. ASP.NET Core wird in .NET ausgeführt.

ASP.NET Core ist eine Neugestaltung von ASP.NET, diese Neugestaltung ermöglichte ein schlankeres und modulareres Framework.

3.3. Flutter

Flutter ist ein UI-Toolkit von Google um Mobile-Applikationen (Apps) zu entwickeln. Wir haben Flutter für die Entwicklung von Shulker-Mobile, der App zur Verwaltung des Türschlosses, verwendet. Einen großen Vorteil, den Flutter für die Entwicklung von Mobilen-Applikationen bietet, ist, dass nur eine einzige Code-Basis für die Entwicklung von sowohl IOS und Android,

3. Verwendete Technologien

als auch Web-Applikationen benötigt wird. Dafür hinaus können Flutter Applikationen auch für Linux, Windows und MacOS kompiliert werden. [8]

3.3.1. Dart

Dart ist eine objektorientierte höhere Programmiersprache, die eine produktive Entwicklung für mehrere Zielplattformen bietet. Das Flutter-Framework wurde für diese Programmiersprache entwickelt. Hierbei stellt Dart die Laufzeit für Flutter Applikationen bereit. In Dart geschriebener Programmcode lässt sich für Web, Mobile und Desktop Zielsysteme kompilieren. Diese Kompilierung erfolgt entweder in Natives-Code, oder für Web-Anwendungen eine Übersetzung in Javascript.

Der Syntax von Dart ist mit anderen Programmiersprachen in der C-Sprachfamilie vergleichbar. Auch Dart wird von Google entwickelt. [9]

3.3.2. Architektur des Flutter-Frameworks

Die wichtigsten Komponenten des Flutter-Frameworks bilden folgende Bausteine: [8]

- Dart-Plattform: Die Programmiersprache Dart bildet die Basis für das Flutter-Framework
- Flutter-Engine: Die Flutter-Engine ist eine portable Laufzeitumgebung für das Ausführen von Flutter-Applikationen
- Foundation-Library: Dieser Komponente stellt grundlegende Klassen und Funktionen, die für Flutter-Applikationen benötigt werden, dar.
- Design-specific-widgets: Diese Designspezifische Widgets definieren das grundlegende Aussehen bestimmter grafischer Elemente des Zielsystems (z.B.: IOS, Android).

3.3.3. Widgets in Flutter

Der Grundbaustein für Flutter-Applikationen sind *Widgets*. Widgets sind Komponente in einem grafischen Anzeigesystem. In Flutter beinhalten ein Widget die Darstellung, Logik und Interaktion, die dieses Widget in der Applikation einnimmt.

Flutter stellt eine große Anzahl an vorgefertigten Widgets, die oft benötigt und im Kontext von Grafischen Benutzeroberflächen bekannt sind, bereit. Diese Widgets können von Entwicklern beliebig verwendet werden. Widgets in Flutter folgen den Designrichtlinien der jeweiligen Plattform (z.B.: Material Design für Android), definieren die Darstellung dieser allerdings selbst.

Aus Widgets lassen sich wiederum eigene, benutzerdefinierte Widgets erstellen. Diese Benutzerdefinierten Widgets setzen sich aus beliebigen anderen Widgets zusammen und ermöglichen eine Wiederverwendung von Programmcode. [10]

3.3.4. Arten von Flutter Widgets

Grundsätzlich lassen sich Flutter Widgets in drei Arten unterteilen:

- Stateless widgets

3. Verwendete Technologien

- Stateful widgets
- Inherited widgets

3.3.4.1. Stateless Widgets

Stateless widgets sind statische Widgets die nur bei ihrer Erstellung aktualisiert werden. Das bedeutet, sie können nicht während der Laufzeit aktualisiert oder geändert werden. [11] Beispiele hierfür sind:

- Text-Widget: Einfacher Text der auf dem Bildschirm angezeigt wird.
- Spacer-Widget: Das Spacer-Widget erstellt eine leere Fläche im UI, um einen Abstand zwischen zwei Widgets zu setzen.

Hier ein Beispiel mit gekürztem Programmcode für eine Ladesseite:

```
class LoadingScreen extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            child: Column(  
                children: [  
                    Text("App ladet..."),  
                    CircularProgressIndicator(),  
                ],  
            ),  
        );  
    }  
}
```

Die erste Zeile des Programmcodes definiert eine neue Klasse namens *LoadingScreen*. Diese Klasse erbt von *StatelessWidget* um ein Stateless-Widget zu definieren. Die *build* Methode, mit dem Rückgabewert *Widget*, wird unter anderem bei der erstmaligen Darstellung des Widgets aufgerufen. [11] Diese *build* Methode gibt ein *Scaffold-Widget* zurück. Ein *Scaffold* ist eine Implementierung der grundlegenden visuellen Layout-Struktur. Innerhalb dieses *Scaffolds* wurde ein *body* definiert, diesen zeigt das *Scaffold-Widget* am Bildschirm an. In diesem *body* befindet sich ein *Column-Widget*, dieses ordnet seine Unterobjekte vertikal an. Letztlich befindet sich darin ein einfaches *Text-Widget* und ein *CircularProgressIndicator-Widget*, dieses Widget ist ein sich drehender Ladekreis.

3.3.4.2. Stateful Widget

Stateful Widgets sind Flutter-Widgets, die einen veränderbaren Zustand besitzen. Ein veränderbarer Zustand ist eine Information, die bei der Erstellung eines Widgets gelesen wird, und sich während der Existenz des Widgets ändern kann.

Eine Änderung des Zustandes muss in Flutter mittels

```
State.setState()
```

3. Verwendete Technologien

gekennzeichnet werden. Wird mittels `setState` eine Variable des Zustandes geändert, wird ein neuer Aufbau des dazugehörigen Objektes geplant. So kann z.B.: der Text eines Stateful-Widgets dynamisch geändert werden. Deshalb werden Stateful-Widgets immer dann verwendet, wenn sich Widgets während ihrer Existenz dynamisch ändern sollen.

3.3.5. Verwendete Flutter-Pakete

Für Flutter entwickelte Dart-Pakete werden auf dem Package-Repository `pub.dev` bereitgestellt. Für die Entwicklung von Shulker-Mobile haben wir einige Pakete verwendet, um die Entwicklung der App zu erleichtern.

Um ein Paket in ein Projekt einzubinden, muss eine Referenz in Form des Namens und der Version in die `pubspec.yaml`-Datei eingefügt werden.

3.3.5.1. Dio

Das `Dio`-Paket erlaubt es einem, Http-Anfragen von der Mobilen-Applikation aus zu versenden. Wir haben es in Shulker-Mobile verwendet, um die Kommunikation mit dem Shulker-Connect API-Server zu ermöglichen

3.3.5.2. qr_code_scanner

Das `qr_code_scanner`-Paket ermöglicht der App eine Kamera zu öffnen, in der QR-Codes gescannt und ausgelesen werden können. Wir haben dieses Paket verwendet, um den Verbindungs-Prozess zum Türschloss zu erleichtern. Indem auf dem Touchscreen des Schlosses ein QR-Code angezeigt wird, welcher die lokale IPv4-Adresse des dazugehörigen Raspberry-PI's beinhaltet, kann dieser in der App gescannt werden, um eine manuelle Eingabe dieser Verbindungs-Parameter zu vermeiden, und den Kopplungs-Prozess mit dem Türschloss für den Nutzer zu erleichtern.

3.3.5.3. shared_preferences

Das `shared_preferences`-Paket ermöglicht die Platform-Spezifische dauerhafte Speicherung von Daten auf dem Mobilen-Endgeräten der Nutzer. In Shulker-Mobile kommt es zum Einsatz, um die Verbindungs-Parameter, also die lokale IPv4 des Raspberry-PI's des Türschlosses, zu speichern.

3.3.5.4. check_vpn_connection

Das `check_vpn_connection`-Paket haben wir in Shulker-Mobile verwendet, um zu überprüfen, ob das Smartphone eine aktive VPN-Verbindung hergestellt hat. Dies zu wissen ist wichtig, da für die Verbindung zum Türschloss ein VPN-Tunnel in das Lokale Heimnetzwerk, in dem das Türschloss steht, benötigt wird. Diese VPN-Verbindung ermöglicht eine sichere und verschlüsselte Verbindung, und erlaubt überhaupt erst die Netzwerk-Kommunikation zu dem sich in einem lokalen-Netzwerk befindendem Türschloss.

3. Verwendete Technologien

3.3.5.5. `open_settings`

Das `open_settings`-Paket verwenden wir, um dem Nutzer eine Abkürzung zur Einstellungs-Seite zu bieten, wo die derzeitige VPN-Verbindung verwaltet werden kann. Diese Abkürzung wird dem Nutzer dann angeboten, falls keine Verbindung zum Türschloss hergestellt werden konnte (Das bedeutet, das Smartphone befindet sich nicht im gleichen Netzwerk wie das Türschloss), und keine aktive VPN-Verbindung herrscht. So kann sich der Nutzer komfortabel beim Start der App mittels 2 Klicks mit dem VPN-Server des Routers im lokalen Netzwerk des Raspberry-PI's verbinden.

3.3.5.6. `flutter_localizations`

Das `flutter_localizations`-Paket wird in Shulker-Mobile verwendet, um eine Menü, das zur Auswahl von Daten und Zeitpunkten verwendet wird, auf Deutsch zu setzen.

3.3.5.7. `cupertino_icons`

Dieses Paket wird verwendet, um Cupertino-Icons (also IOS-Icons) in die Applikation einzubinden.

3.3.5.8. `uuid`

Das `uuid`-Paket haben wir in das Projekt eingebunden, um UUID's der 4. Version zu erstellen. Dies wird benötigt, um für neue Pin-Codes Universell eindeutige Bezeichnungen zu generieren.

Teil III.

Shulker-Core

4. Überblick

4.1. Allgemeines

Shulker-Core besteht aus einer *Rust-Binary*. Das bedeutet, dass der gesamte Quellcode von Shulker-Core zu einem einzigen Binärprogramm kompiliert wird. Shulker-Cores gesamter Code befindet sich in einem Unterordner in der Shulker Git-Repository.

Wie der Name schon sagt, ist *Shulker-Core* der Kern von Shulker. Das Binärprogramm ist dafür verantwortlich, Pins zu speichern, Pins zu vergleichen, das Touchdisplay zu steuern, das elektronische Schloss zu steuern und eine Schnittstelle für *Shulker-Connect* bereitzustellen. Um all diese Dinge effizient und sicher bewältigen zu können, wurde Rust als Programmiersprache gewählt. Shulker-Core macht starken Gebrauch vom Cargo-Packagemanager und der offiziellen Package-Registry *crates.io*.

4.2. Funktionsweise und Aufbau

Der Aufbau von Shulker-Core ist recht komplex, da viele Aufgaben möglichst sicher und effizient erledigt werden müssen. In der Programmierung wurde darauf geachtet, Shulker-Core in möglichst verständliche Teilbereiche aufzuteilen, um das Überblicken der gesamten Funktionalität zu vereinfachen.

Diese Teilbereiche sind im Quellcode auf verschiedene Dateien aufgeteilt und somit auch sofort erkennbar. Die jeweiligen Dateien besitzen die Dateiendung *.rs*. Diese Endung kennzeichnet, dass die Datei Rust-Quellcode beinhaltet. Der Quellcode von Shulker-Core wurde auf folgende Dateien aufgeteilt:

- **main.rs:** Der Quellcode in dieser Datei wird beim Start von Shulker-Core als erstes ausgeführt. Sie verwaltet unter anderem das Einlesen der Konfigurationsdatei, die Erstellung des QR-Codes, das Starten der grafischen Benutzeroberfläche und das Initialisieren der ShulkerCore-Struktur, in der alle anderen Teilbereiche residieren.
- **hasher.rs:** Diese Datei enthält den Quellcode für das Hashing der Pins. Zuständig ist sie also dafür, einen Eingabe-Pin zu hashen oder einen Eingabe-Pin mit einem Hash zu vergleichen. Diese Datei ist essentiell zum Verifizieren von Pins.
- **credential_types.rs:** Enthalten ist die Definition eines Pins und einige wenige Pin-Operationen.
- **messaging.rs:** In dieser Datei befinden sich die verschiedenen Typen an Nachrichten, die Shulker-Connect an Shulker-Core senden kann. Quellcode zum Senden und Empfangen solcher Nachrichten ist natürlich auch enthalten.
- **shulker_db.rs:** Der Quellcode in dieser Datei verwaltet die Datenbank zur Speicherung

4. Überblick

und Verwendung von Pins. Sie macht starken Gebrauch von *hasher.rs*, um die Informationen sicher abzuspeichern und zu überprüfen.

- **mainwindow.ui:** Der Großteil des Quellcodes, der die grafischen Benutzeroberfläche definiert, ist hier enthalten.
- **core.rs:** Diese Datei ist die Datei, die die bereits genannten Teilbereiche zusammenführt und bildet somit den Kern von Shulker-Core. In ihr ist Quellcode für die Verwendung der GPIO-Pins, das Öffnen und Schließen des elektronischen Schlosses und das Verarbeiten von Nachrichten enthalten.

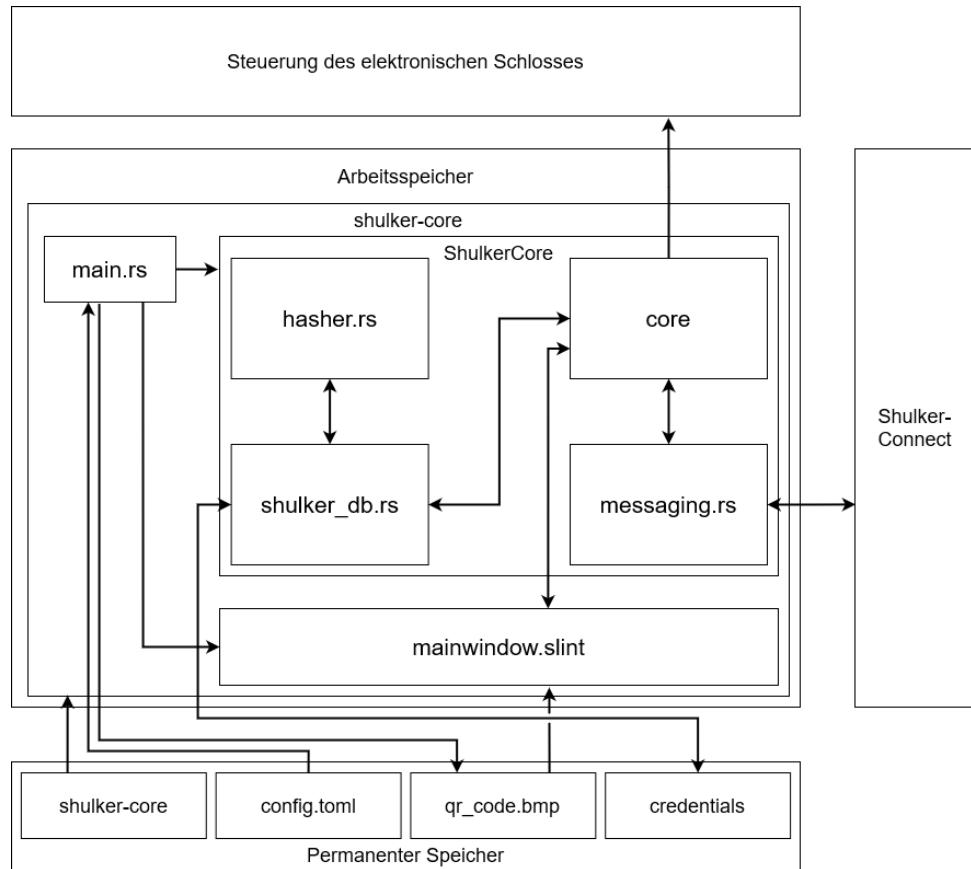


Abbildung 4.1.: Informationsfluss in Shulker-Core

4.3. Konfiguration

Die gesamte Konfiguration von Shulker-Core befindet sich in einer optionalen Textdatei. Da für jede mögliche Einstellung ein Standartwert einprogrammiert ist, muss man nur nötige Änderungen vornehmen. Man muss sich also keine unnötigen Gedanken machen, wie man jeden Wert in der Konfigurationsdatei definieren möchte.

Die Konfigurationsdatei hat das Dateiformat TOML und den Namen **config.toml**. Das Programm sucht im lokalen Ordner nach dieser Datei. Das bedeutet, dass die config.toml-Datei sich

4. Überblick

im gleichen Ordner befinden muss, wie die Binary. Findet Shulker-Core diese Datei nicht, werden alle Standartwerte übernommen. Als Warnung wird eine Warnmeldung an den Standart-Error ausgegeben.

4.3.1. Mögliche Konfigurationen

Es folgt eine Auflistung aller Konfigurationsschlüssel und ihrer Standartwerte:

```
master_password = "master123"
```

Das Master-Passwort dient zur Identifikation der Admins. Das standartmäßige Adminpasswort sollte dringend durch ein Passwort vernünftiger Komplexität ersetzt werden. Es ist das einzige Passwort, das nicht gehashed wird. Dafür gibt es drei Gründe:

1. Wer Zugriff zum Raspberry Pi hat, hat Zugriff zum Schloss.
2. Dem Admin wird die Fähigkeit, ein einzigartiges Passwort zu erstellen, vorrausgesetzt.
3. Bei Vergessen des Passworts kann der Admin es einfach wieder herausfinden.

```
send_socket_path = "/tmp/toShulkerServer.sock"
```

Gibt den Pfad der Unix-Socket an, die zur Kommunikation mit Shulker-Connect dient; In Richtung Shulker-Core zu Shulker-Connect. Um das System möglichst flexibel zu gestalten, wurde diese Konfiguration erschaffen. Man kann so leicht das System anpassen.

```
receive_socket_path = "/tmp/toShulkerCore.sock"
```

Gibt den Pfad der Unix-Socket an, die zur Kommunikation mit Shulker-Connect dient; In Richtung Shulker-Connect zu Shulker-Core. Um das System möglichst flexibel zu gestalten, wurde diese Konfiguration erschaffen. Man kann so leicht das System anpassen.

```
gpio_pin = 27
```

Gibt an, welcher GPIO-Pin des Raspberries für die Ansteuerung des elektronischen Schlosses verwendet werden soll. Zu beachten: soll das Schloss geschlossen sein, ist der GPIO-Pin auf High.

```
autolock_seconds = 24
```

Die Dauer, die der GPIO-Pin bei richtiger Eingabe eines Pins (Passworts) auf High bleibt.

```
qr_code_link = "not setup!"
```

Die Daten, die der QrCode in der M-Ansicht des Core-UIs enkodieren und zugänglich machen soll. Gedacht ist, dass der Admin hier die IP des Raspberry Pis angibt.

```
hash_memory_size = 10
```

Anzahl an Arbeitsspeicher, der für das Hashing der Pins verwendet wird. Genauereres wird später erklärt.

```
hash_iterations = 3
```

Anzahl der Argon2id Durchläufe. Genauereres wird später erklärt.

```
hash_parallelism = 1
```

Anzahl der Hash-Threads. Genauereres wird später erklärt.

4. Überblick

4.4. Abhängigkeiten

Shulker-Core baut auf vielen *crates*, also Codebibliotheken auf. Alle direkten Abhängigkeiten sind in der Cargo.toml-Datei nachvollziehbar. Es folgt eine kurze Auflistung und Erklärung der wichtigsten Abhängigkeiten:

- **slint**; früher bekannt als SixtyFPS, ist ein User-Interface-Toolkit. Es ermöglicht relativ simples und plattformübergreifendes Programmieren von grafischen Benutzeroberflächen.
- **serde**; steht für **Serialize** und **Deserialize**. Es ist das am bekannteste Framework, um Rust-Strukturen zu serialisieren und zu entserialisieren. Benutzt wird es, um die Kommunikation über Unix-Sockets zu ermöglichen.
- **argon2**; ist ein kryptographischer Passwort-Hashing Algorithmus. Das *argon2*-crate implementiert diesen Algorithmus 100%-ig mit Rust-Code. Die Variante Argon2id wird für das Unkenntlichmachen der Pins benutzt.
- **gpio-cdev**; wird verwendet, um den GPIO-Pin zu benutzen. Dabei wird auf die Linux-Kernel-API namens *GPIO character device ABI* aufgebaut.
- **config**; ist ein crate, welches das Einlesen der Konfigurationsdatei erleichter.
- **chrono**; stellt Code zur Verfügung, um den Umgang mit Zeit und Datum zu erleichtern.
- **interprocess**; erleichtert die Verwendung von Unix-Sockets.
- **rustbreak**; eine einfache, in sich geschlossene Datenbank. Wird zur Speicherung von Daten verwendet.
- **qr_code**; ermöglicht schnelles Erstellen von QR-Codes. Wird verwendet um die Verbindung mit Shulker-Mobile zu erleichtern. Es ist das einzige crate, welches nicht von crates.io heruntergeladen wird. Das crate wird direkt von der Git-Repository auf Github kopiert.

5. Der Touchscreen

Der Touchscreen von Shulker kann grundsätzlich jeder Touchscreen mit akzeptabler Größe sein. Die Steuerung des Touchscreens wird von Shulker-Core mithilfe des crates **slint** (Version 0.2) übernommen.

5.1. Funktionsweise von **slint**

Um mit **slint** eine grafische Benutzeroberfläche bereitstellen zu können, wird empfohlen, eine **.slint**-Datei erstellen. In dieser Datei wird mittels der **.slint**-Sprache eine Benutzeroberfläche beschrieben und programmiert. Diese Datei wird vom **slint**-Compiler in Binärkode umgewandelt. Da die **.slint**-Datei und der eigentliche Anwendungscode also getrennt sind, muss man sich mittels Callbacks, Models und Eigenschaften verständigen.

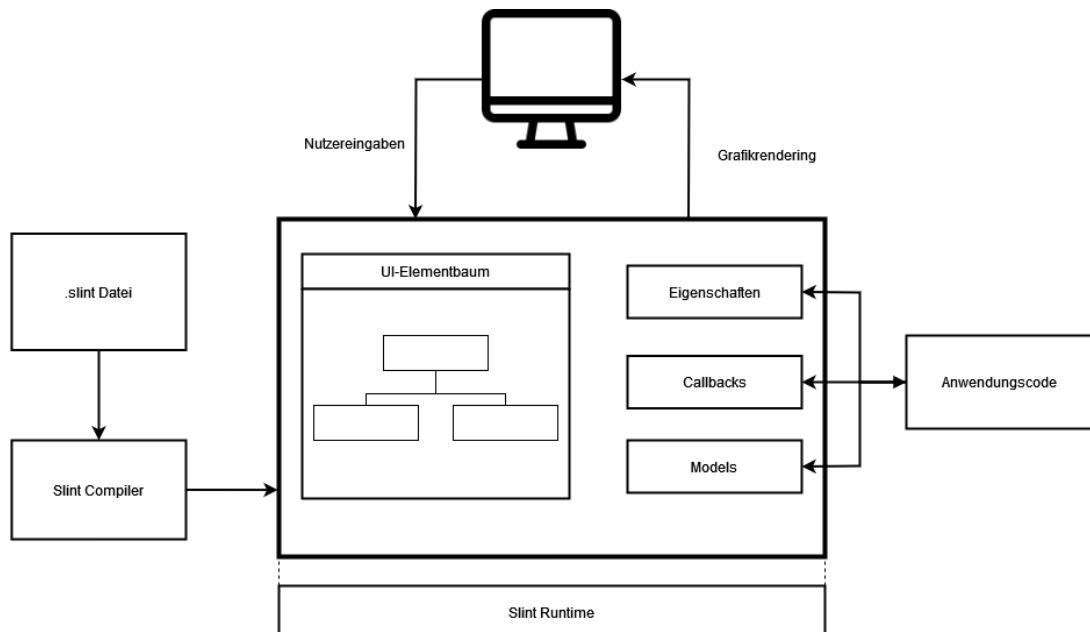


Abbildung 5.1.: Aufbau und Funktionsweise von **slint** [12]

5.2. Aufbau der grafischen Oberfläche

Shulker-Core verwendet eine **.slint**-Datei, welche den Namen *mainwindow.slint* trägt. Diese Datei befindet sich im Unterordner namens *ui* in der Git-Repository von Shulker. Das Design der

5. Der Touchscreen

Oberfläche konzentriert sich darauf, möglichst leicht lesbar und stromsparend zu sein. Deshalb entschieden wir uns für die Farben Schwarz und Weiß.

Die grafische Oberfläche besteht aus drei Teilen:

- Menü-Selektor
- Hauptsicht
- Unlocked-Ansicht

5.2.1. Menü-Selektor

Der Menü-Selektor ist grundsätzlich immer zu sehen. Die einzige Ausnahme bildet das Entsperrtsein des elektronischen Schlosses. Der Menü-Selektor ermöglicht das Wechseln der Hauptansicht. Es gibt drei Wahlmöglichkeiten:

- **Pin:** zur Eingabe von Ziffernpins
- **PW:** zur Eingabe von Passwörtern, also Pins mit Buchstaben
- **M:** zur Verbindung mit Shulker-Mobile

5.2.2. Hauptansicht

Die Hauptansicht nimmt den Großteil des Bildschirms in Anspruch. In ihr wird dem Nutzer das Eingeben von Pins und Passwörtern ermöglicht. Ein QR-Code zur Verbindung mit Shulker-Mobile lässt sich auch anzeigen.

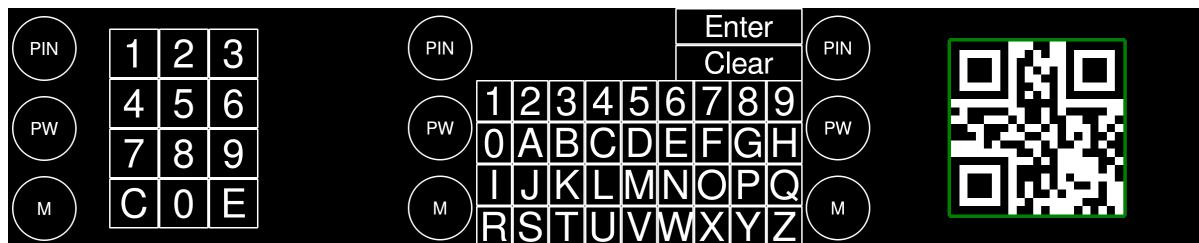


Abbildung 5.2.: Core-UI mit den verschiedenen Hauptansichten: PIN, PW und M

5.2.3. Unlocked-Ansicht

Die Unlocked-Ansicht zeigt dem Nutzer, dass das Schloss momentan offen ist. Sie nimmt den ganzen Bildschirm in Anspruch und weicht als einzige Komponente vom standart Farbschema ab; gretles Grün vermittelt ein klares Signal. Auf ihr ist auch ersichtlich, wie lange das Schloss noch offen bleibt. Die Ansicht kann über mehrere Wege in den Vordergrund geschoben werden, grundsätzlich aber dann, wenn an Shulker-Core ein Öffnen-Signal oder ein korrekter Pin gesendet wurde.

5. Der Touchscreen

5.3. Die Hardware

5.3.1. Vorzeigemodell

Im Vorzeige Modell wird das offizielle *Raspberry Pi Touch Display* verwendet. Das Display besitzt eine Auflösung von 800x480 Pixeln und hat eine diagonale Größe von sieben Zoll. Natürlich unterstützt es auch Berührungseingaben, um die Nutzbarkeit von Shulker zu ermöglichen.

5.3.2. Mögliche Displays

Grundsätzlich kann Shulker-Core mit jedem Display, das mit dem Raspberry Pi kompatibel ist, arbeiten. Sollte ein besonders kleiner Bildschirm gewählt werden, ist es äußerst wahrscheinlich, dass die Standard-Implementation des User-Interfaces nicht mehr den Verhältnissen entspricht. Da das gesamte Projekt Open-Source und die Slint-Sprache nicht besonders komplex ist, kann in solchen Fällen der Admin selbst eine eigenen Implementation der grafischen Benutzeroberfläche nutzen. Das ist natürlich auch von Vorteil, wenn man zusätzliche Funktionen hinzugefügen will.

6. Speicherung von Daten

Shulker-Core speichert die vom Admin angelegten Pins ab. Alle Pins werden in einer Datei namens *credentials* abgespeichert. Es wird keine vollwertige Datenbank verwendet, um das System nicht zu groß zu gestalten und den Resourcenverbrauch auf dem Raspberry möglichst gering zu halten.

6.1. Rustbreak

Shulker-Core verwendet das crate *rustbreak* zum Speichern von Daten. Rustbreak ist eine von der Ruby-Datenbank *daybreak* inspirierte Datenbank, welche Daten nur in einer Datei speichert. Es wird die Version 2.0 von rustbreak verwendet.

Der größte Vorteil von Rustbreak gegenüber einer herkömmlichen Datenbank ist, dass die gesamte Rustbreak-Datenbank in den Arbeitsspeicher kopiert wird. Veränderungen an der Datenbank erfolgen dann im Arbeitsspeicher. Dem Programmierer ist es überlassen, wann er die Datenbank im Arbeitsspeicher auf den permanenten Speicher überträgt und somit permanent abspeichert. Das ist von großen Vorteil, da die meisten Raspberry Pi Setups eine sehr langsame SD-Karte als permanentes Speichermedium nutzen. Somit können Verzögerungen in der Anwendung minimiert werden.

6.2. Funktionsweise

Shulker-Core lädt die gesamte *credentials*-Datei beim Start in den Arbeitsspeicher. Wird ein eingegebener Pin mit der Datenbank abgeglichen, muss so nur die geladene Datenbank abgefragt werden. Um Änderung an der Datenbank permanent zu machen, wird die geladene Datei nach jeder Änderung wieder auf die Festplatte kopiert. Das System ist so konfiguriert, dass ein Fehler beim Abspeichern der Datei keine Probleme verursacht. Die Ursprungsdatei bleibt in solchen Fällen erhalten. Es kommt zu keinem Datenverlust.

6.3. Hashing von Pins

Da die Nutzer von Shulker aus Einfachheit möglicherweise Passwörter nutzen wollen, die sie an anderen Orten bereits verwenden, werden alle Pins gehashed abgespeichert. Somit ist es dem Admin und einem möglichen Hacker nicht möglich, die originalen Pins nachzuvollziehen und somit möglicherweise in anderen Bereichen Schaden anzurichten.

6. Speicherung von Daten

6.3.1. Argon2

Als Hashing-Algorithmus wird Argon2, genauer Argon2id verwendet. Argon2 wurde gewählt, da eine reine Rustimplementation des Algorithmus existiert und der Algorithmus sich als Gewinner der *Password Hashing Competition* behaupten.[13]

6.3.1.1. PHC-String

Die unkenntlich gemachten Pins werden in der Form von PHC-Strings abgespeichert. Ein PHC-String besteht aus:

- **id:** Kennzeichnung des verwendeten Hash-Algorithmus
- **version:** Version des Algorithmus
- **param:** mehrere Parameter, die dem Algorithmus gefüttert werden
- **salt:** zufällige Zeichenkette, welche das Schützen des Passworts weiter verbessert
- **hash:** der eigentliche Hash, also der unkenntlich gemachte Pin

```
$argon2id$v=19$m=65536,t=2,p=1$gZiV/M1gPc22E1AH/Jh1Hw$CWorkoo7oJBQ/iyh7uJ0L02aLEfrHwIWl1SAxT0zRno
```

Abbildung 6.1.: Beispiel eines PHC-Strings

[14]

6.3.1.2. Konfiguration von Argon2id

Shulker-Core nimmt drei Parameter für Argon2id an:

- **Memory:** Die Menge an Arbeitsspeicher die der Algorithmus verwenden soll. Je höher dieser Wert ist, desto schwieriger ist es, den Hash zu berechnen. Somit wird es einem Angreifer ebenfalls schwerer, viele Hashes zu generieren und somit das originale Passwort herauszufinden. Wird in der Shulker-Core-Konfigurationsdatei als *hash_memory_size* definiert.
- **Iterations:** Wie oft sich der Algorithmus durch die ausgewählte Größe an Arbeitsspeicher durcharbeiten soll. Je höher, desto sicherer die Hashes. Wird in der Shulker-Core-Konfigurationsdatei als *hash_iterations* definiert.
- **Parallelism:** Wie viele Threads der Algorithmus verwenden soll. Desto höher dieser Wert, desto sicherer ist der Hash vor großen parallelen Attacken (zum Beispiel das Cracken mit Grafikkarten). Wird in der Shulker-Core-Konfigurationsdatei als *hash_parallelism* definiert.

Teil IV.

Shulker-Connect

7. Allgemeines

Shulker-Connect ist die Softwarelösung, um von externen Ressourcen aus mit das Türschloss zu steuern.

Dies haben wir mittels eines ASP.NET Core Web API Servers umgesetzt. Dieser Server stellt eine REST-API zur Verfügung, auf die von beliebigen Applikationen aus zugegriffen werden kann, um Shulker-Core Befehle mitzuteilen und Abfragen zu stellen.

In der Shulker-Zutrittsmanagement-Lösung wird der Shulker-Connect API-Server von Shulker-Mobile verwendet. Durch die Implementierung von Shulker-Connect könnten allerdings auch beliebige andere Applikationen, wie z.B.: eine Steuerungs-Website, das Türschloss steuern. So könnten später auf einfacher weise die Steuerungsmöglichkeiten des Schlosses erweitert werden.

7.1. Authentifizierung am Server

Immer dann, wenn Shulker-Mobile auf Routen von Shulker-Connect zugreifen möchte, muss sich der Client Authentifizieren. Diese Authentifizierung haben wir mittels eines Session-Systems umgesetzt.

Um dies am Server umzusetzen, haben wir eine *SessionManager*-Klasse programmiert. Diese Klasse folgt dem *Singleton Pattern* und verwaltet alle aktiven Sessions. In Shulker bleibt eine Session 20 Minuten valide, dann muss eine neue generiert werden.

Der Session-String selbst setzt sich aus 32 kryptografisch zufällig ausgewählten Buchstaben bzw. Zahlen zusammen.

7.2. Verschlüsselung der Anfragen von Shulker-Mobile zum API-Server

Da es sich bei einem Haustürschloss um eine sehr wichtige und sichere Vorrichtung handelt, muss natürlich auch die Kommunikation zu diesem auf sicherem Wege erfolgen. Das Türschloss soll schließlich mittels App von der ganzen Welt aus steuerbar sein.

Um dieses Problem zu lösen und eine definitiv sichere und verschlüsselte Kommunikation zu gewährleisten, haben wir uns dazu entschieden, den Nutzer zu zwingen, einen VPN-Tunnel zum lokalen Netzwerk, in dem auch das Türschloss steht, herstellen zu müssen, falls dieser sich nicht im lokalen Netzwerk befindet. So sicher wie das VPN-Protokoll, dass zur Verbindung genutzt wird ist, ist somit auch die Verbindung zum Türschloss.

Als zusätzlichen Vorteil bietet diese Lösung auch, dass das Türschloss nicht vom öffentlichen Internet aus erreichbar sein muss. Dies sichert das Gesamtsystem noch einmal zusätzlich ab und schließt einen wichtigen Angriffsvektor.

8. Kommunikation mit Shulker-Core

Shulker-Connect muss mit *Shulker-Core* kommunizieren, sodass eine API-Anfrage an Shulker-Connect tatsächlich auch das Türschloss steuern kann. Eine lokale Kommunikation zwischen mehreren Software-Lösungen kann mittels *inter process communication* umgesetzt werden. Um eine stabile und effiziente Kommunikation mit der auf dem Raspberry PI laufenden Rust-Software Shulker-Core zu gewährleisten, haben wir auf beiden Endpunkten *POSIX-Sockets* implementiert.

Beim Start von Shulker-Connect werden zwei neue Threads erstellt: Ein Listener-Thread und ein Sender-Thread. Beide Threads erstellen ein *socket*-Objekt, das auf eine Verbindung von *Shulker-Core* wartet. Wird diese hergestellt, sind die Threads bereit, Daten zu senden bzw. zu empfangen. In Shulker werden alle Nachrichten im *UTF8*-Format gesendet.

8.1. Was sind POSIX-Sockets?

POSIX local inter-process communication sockets (auch Unix Domain Sockets oder IPC Sockets genannt) ermöglichen eine bidirektionale Kommunikationsverbindung für die Interprozesskommunikation (IPC) auf UNIX basierenden Systemen. Hierbei wird von beiden Kommunikationspartnern eine Datei vereinbart, über diese die Kommunikation erfolgt. [15] Die Kommunikation zwischen Shulker-Connect und Shulker-Core muss nicht verschlüsselt werden, da eine Komprimierung des Raspberry-Pi's die einzige Möglichkeit darstellt, diese Kommunikation mitzulegen.

8.2. Aufbau der POSIX-Nachrichten

Da Shulker-Connect und Shulker-Core über String-Nachrichten miteinander kommunizieren, haben wir ein Format festgelegt, an welchen sich alle Nachrichten der Kommunikation halten müssen.

Alle gesendeten Nachrichten werden im JSON-Format gesendet. Jede JSON-Nachricht beinhaltet immer ein Attribut namens "*method*", dieses ist der Identifikator der Anfrage und lässt sich mit Routen von Web-Diensten vergleichen. Zusätzlich können beliebige andere Attribute, die für die jeweilige Route relevant sind, in der Nachricht mitgesendet werden. Zu guter Letzt wird ein Nachrichtentrenner an das Ende einer jeder Nachricht gesetzt. Wir haben uns für "\n" entschieden.

8. Kommunikation mit Shulker-Core

8.3. Ablauf einer Anfrage zu Shulker-Connect

Um zu demonstrieren, wie der Ablauf einer Anfrage an Shulker-Connect intern gehandhabt wird, schauen wir uns das Generieren eines Session-Tokens an.

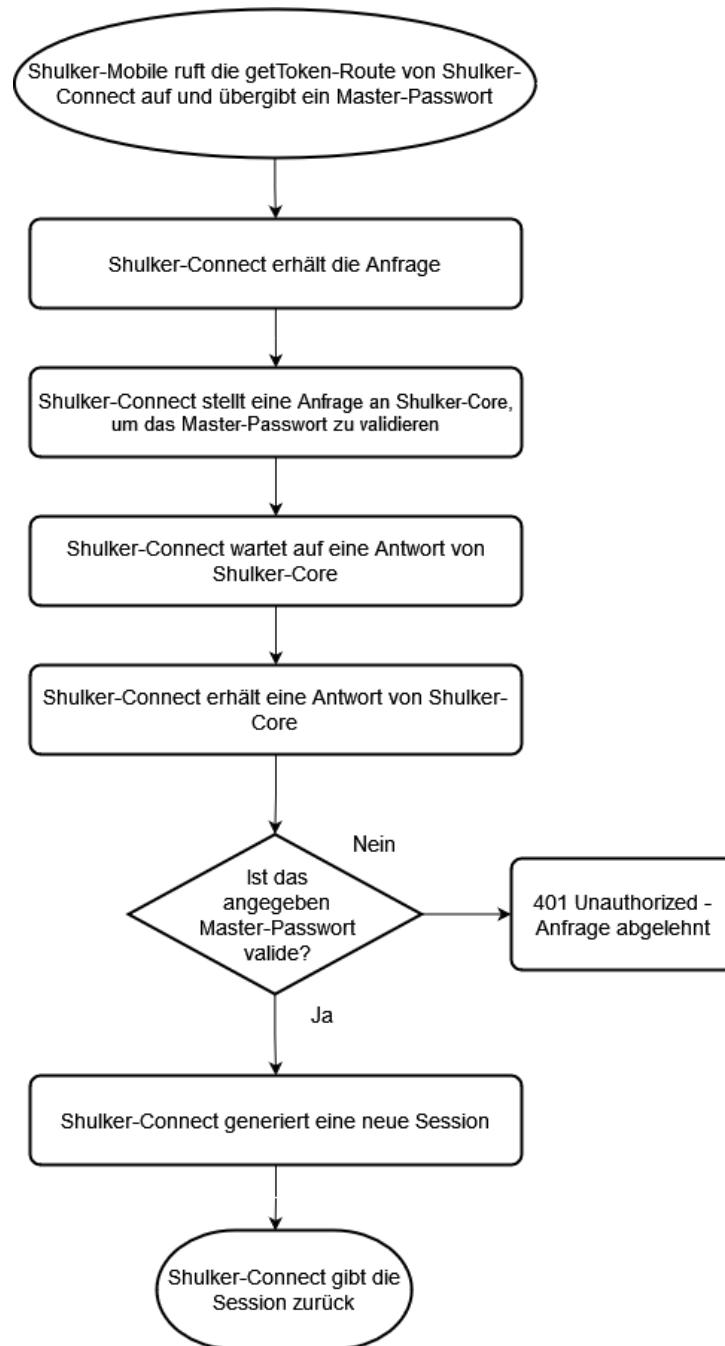


Abbildung 8.1.: Ablauf der Abfrage zum generieren einer neuen Session

Als erstes ruft Shulker-Mobile beim Start der App die Route `/api/Session/getToken/secret`

8. Kommunikation mit Shulker-Core

auf. Diese Route nimmt einen Parameter namens *secret* - das ist das Master-Passwort des Türschlosses.

Shulker-Connect nimmt die Anfrage in der *getToken* Methode innerhalb des *Session Controllers* entgegen, wie der Namen des Controllers verdeutlicht, handhabt dieser die Sessions.

Im Anschluss stellt Shulker-Connect über den *POSIX-Socket* in Richtung Shulker-Core eine Anfrage, um zu überprüfen, ob das Master-Passwort Valide ist. Genaueres zum Aufbau der Anfragen folgt später. Shulker-Connect wartet währenddessen asynchron auf die Antwort von Shulker-Core.

Sobald eine Antwort von Shulker-Core über den POSIX-Socket empfangen wurde, läuft die *getToken* Methode weiter. Dort wird anschließend, abhängig davon ob das angegebene Master-Passwort valide ist, eine neue Session erstellt und zurückgegeben, oder die Anfrage über den HTTP-Error *401 Unauthorized* abgelehnt.

8.4. Beispiel

Shulker-Connect sendet folgenden String an Shulker-Core.

```
{"method": "GetPins"}
```

Shulker-Core erhält die Nachricht, und schaut sich die *method* des JSON-Strings an. Die *method GetPins* definiert, dass Shulker-Connect eine Liste alle Pins anfordert.

Shulker-Core sendet nun Beispielhaft folgende Nachricht an Shulker-Connect zurück.

```
{
  "method": "PinList",
  "pins": [
    {
      "label": "Oma",
      "uuid": "7cb46a16-9794-486c-89b5-997b32f9b81c",
      "start_time": "2022-03-19T16:40:34.098Z",
      "end_time": "2022-05-01T15:00:00Z",
      "uses_left": -1
    }
  ]
}
```

Somit kann Shulker-Connect anhand der *method PinList* erkennen, dass es sich bei dieser Nachricht um die Liste aller Pins, also einer Antwort der vorherigen Anfrage, handelt.

8.5. Routen der Interprozesskommunikation

In Shulker-Connect und Shulker-Core haben wir folgende Routen für die lokale Interprozess-kommunikation implementiert:

8.5.1. CreatePin

8. Kommunikation mit Shulker-Core

```
{  
    "method": "CreatePin",  
    "pin": <Credential>,  
}
```

Beschreibung:

Erstellt einen neuen Pin

8.5.1.1. Parameter

[Erforderlich] pin

Typ: Credential

Beschreibung: Der zu erstellende Pin.

8.5.2. UseMaster

```
{  
    "method": "UseMaster",  
    "secret": <Master-Passwort>,  
}
```

Beschreibung:

Überprüft die Korrektheit des Master-Pins.

8.5.2.1. Parameter

[Erforderlich] secret

Typ: string

Das zu validierende Master-Passwort

8.5.3. Lock

```
{  
    "method": "Lock"  
}
```

Beschreibung:

Sperrt das Türschloss.

8.5.4. Unlock

```
{  
    "method": "Unlock"  
}
```

8. Kommunikation mit Shulker-Core

Beschreibung:

Entsperrt das Türschloss.

8.5.5. GetPins

```
{  
    "method": "GetPins"  
}
```

Beschreibung:

Fragt alle auf dem Türschloss erstellten Pins ab.

Antwort von Shulker-Connect

Liste aller erstellten Pins.

8.5.6. DeletePin

```
{  
    "method": "DeletePin",  
    "uuid": <uuid>  
}
```

Beschreibung:

Löscht einen Pin von der Datenbank von Shulker-Core.

8.5.7. Status

```
{  
    "method": "Status"  
}
```

Beschreibung:

Fragt ab, ob das Türschloss zurzeit geschlossen oder geöffnet ist.

Antwort von Shulker-Connect

Nachricht mit *method Locked* oder *Unlocked*.

9. Routen der REST-API

Im folgenden Kapitel werden die Routen beschrieben, die Shulker-Connect als REST-API bereitstellt.

9.0.1. /api/Credentials

```
GET /api/Credentials
```

Beschreibung:

Gibt eine Liste aller Pins zurück.

Rückgabe:

Liste aller Pins im JSON-Format.

```
[Credential{
    uid          string
    start_time   string($date-time)
    end_time     string($date-time)
    uses_left    integer($int32)
    secret       string
    label        string
}]
```

Abbildung 9.1.: Rückgabe der Route

9.0.1.1. Parameter

[Erforderlich] session

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

9.0.1.2. Beispiel

```
GET /api/Credentials?session=duHSYvYpBMzaekLGrutLosFECRTIKweC
```

Beispielhafte Antwort

```
[  
 {
```

9. Routen der REST-API

```
        "uuid": "58813b36-d3d9-4255-b212-6376e0b9fe17",
        "start_time": "2022-03-19T16:40:34.098Z",
        "end_time": "9999-01-01T15:00:00Z",
        "uses_left": -1,
        "secret": null,
        "label": "Mama"
    }
]
```

9.0.2. /api/Credentials/createPin

```
POST /api/Credentials/createPin
```

Beschreibung:

Erstellt einen neuen Pin.

Rückgabe:

Bei Erfolg: *HTTP 200 Success*

Bei ungültiger Anfrage: *HTTP 400 Bad Request*

9.0.2.1. Parameter

[*Erforderlich*] **session**

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

[*Erforderlich*] **Credential**

Typ: **Credential**

Beschreibung: Das zu erstellende Credential.

9.0.2.2. Beispiel

```
POST      /api/Credentials/createPin
{
    "uuid": "58813b36-d3d9-4255-b212-6376e0b9fe17",
    "start_time": "2022-03-19T16:40:34.098Z",
    "end_time": "9999-01-01T15:00:00.000Z",
    "uses_left": -1,
    "secret": "14205735",
    "label": "Mama"
}
```

Beispielhafte Antwort

HTTP 200 Success

9. Routen der REST-API

9.0.3. /api/Credentials/deletePin/uuid

```
DELETE /api/Credentials/deletePin/{uuid}
```

Beschreibung:

Löscht einen existierenden Pin.

9.0.3.1. Parameter

[*Erforderlich*] **session**

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

[*Erforderlich*] **uuid**

Typ: **string**

Beschreibung: UUID des zu löschenen Pins.

9.0.3.2. Beispiel

```
POST /api/Credentials/deletePin/{uuid}
{
  "session": "BzPwGtFJgCeXdOxxtiWSWott4u7AxaNk",
  "uuid": "58813b36-d3d9-4255-b212-6376e0b9fe17"
}
```

Beispielhafte Antwort

HTTP 200 Success

9.0.4. /api/Lock/isLocked

```
GET /api/Lock/isLocked
```

Beschreibung:

Gibt zurück, ob das Türschloss gerade geöffnet oder geschlossen ist.

Rückgabe:

boolean: true/false

9.0.4.1. Parameter

[*Erforderlich*] **session**

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

9. Routen der REST-API

9.0.4.2. Beispiel

```
GET /api/Lock/isLocked
```

Beispielhafte Antwort

true

9.0.5. /api/Lock/setLockState

```
POST /api/Lock/setLockState
```

Beschreibung:

Öffnet bzw. schließt das Türschloss

9.0.5.1. Parameter

[Erforderlich] **session**

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

[Erforderlich] **closed**

Typ: **boolean**

Soll das Schloss geschlossen sein.

9.0.5.2. Beispiel

```
POST /api/Lock/setLockState
{
    "closed": "false"
}
```

Öffnet das Türschloss

Beispielhafte Antwort

HTTP 200 Success

9.0.6. /api/Session/getToken/secret

```
GET /api/Session/getToken/{secret}
```

Beschreibung:

Generiert eine neue Session für die Authentifikation.

9. Routen der REST-API

9.0.6.1. Parameter

[*Erforderlich*] **session**

Typ: **string**

Beschreibung: Session-String zur Authentifikation.

[*Erforderlich*] **secret**

Typ: **string**

Das Master-Passwort des Türschlosses.

9.0.6.2. Beispiel

```
GET /api/Session/getToken/master123
```

Beispielhafte Antwort

AJr4O2VE8bT3DeCW5rG4yyB1Esy1vXJY

9.0.7. /api/Status

```
GET /api/Status
```

Beschreibung:

Kann verwendet werden, um zu überprüfen, ob die Kommunikation zum API-Server funktioniert.

Rückgabe:

HTTP 200 Success

9.0.7.1. Beispiel

```
GET /api/Status
```

Beispielhafte Antwort

HTTP 200 Success

10. Routen der Interprozesskommunikation

toDO

Teil V.

Shulker-Mobile

11. Allgemeines

Shulker-Mobile ist eine Mobile-Applikation zur Steuerung des Türschlosses. Shulker-Mobile wurde in Flutter entwickelt. Die Funktionalität von Shulker-Mobile beinhaltet das öffnen bzw. schließen des Türschlosses und das erstellen neuer Pins, die am Touchscreen des Türschlosses verwendet werden können, um dieses zu öffnen.

11.1. Verbinden des Türschlosses

Beim ersten Start der Applikation muss sich diese die Verbindungsparameter für die Kommunikation mit dem Türschloss speichern. Für diesen Vorgang bietet Shulker-Mobile 2 Möglichkeiten:

- Türschloss manuell verbinden
- Türschloss mittels QR-Code verbinden

Nachdem der Nutzer das Türschloss erstmalig verbunden hat, werden die Verbindungsparameter über das *shared_preferences*-Paket lokal auf dem Smartphone gespeichert.



Abbildung 11.1.: Optionen zum verbinden des Türschlosses

11. Allgemeines

11.1.1. Türschloss manuell verbinden

Um die Applikation manuell mit dem Türschloss zu verbinden, muss sowohl die lokale IP-Adresse des Raspberry PI's, als auch der Port des vom Shulker-Connect laufenden Web-Servers angegeben werden.



Abbildung 11.2.: Eingabefelder um das Türschloss manuell zu verbinden

11.1.2. Türschloss mittels QR-Code verbinden

Alternativ stellt das Shulker-Türschloss-System auch eine leichtere, empfohlene Option, bereit, um die App mit dem Türschloss zu verbinden.

ToDo: Bild einfügen

Hierzu kann der Nutzer auf dem Touchscreen des Türschlosses einen QR-Code anzeigen lassen, welche die Verbindungsparameter beinhaltet und in der App gescannt werden kann. Die ausgelesenen Daten werden dann automatisch in das gleiche Formular, das auch bei der manuellen Verbindung verwendet wird, eingefügt. So muss der Nutzer keine Verbindungsparameter manuell in die App eintragen.

11.1.3. Verbindung außerhalb des Lokalen Netzwerkes

Natürlich soll das Türschloss nicht nur vom Lokalen Netzwerk des Raspberry PI's, sondern auch vom Internet aus gesteuert werden können. Um dies zu ermöglichen, muss auf dem Router des Heimnetzes ein VPN-Server eingerichtet werden. Da die Konfiguration eines VPN-Servers von Router zu Router unterschiedlich ist, gehen wir auf dies nicht genauer ein. Im Internet finden sich viele Anleitungen, die dies für diverse Router erklären.

11. Allgemeines

11.1.3.1. Überprüfung der Verbindung in der Applikation

Beim Start der Applikation müssen einige Dinge überprüft werden, um sicherzustellen, dass die Kommunikation mit dem Türschloss reibungslos funktioniert. Dies ist in folgendem Ablaufdiagramm ersichtlich:

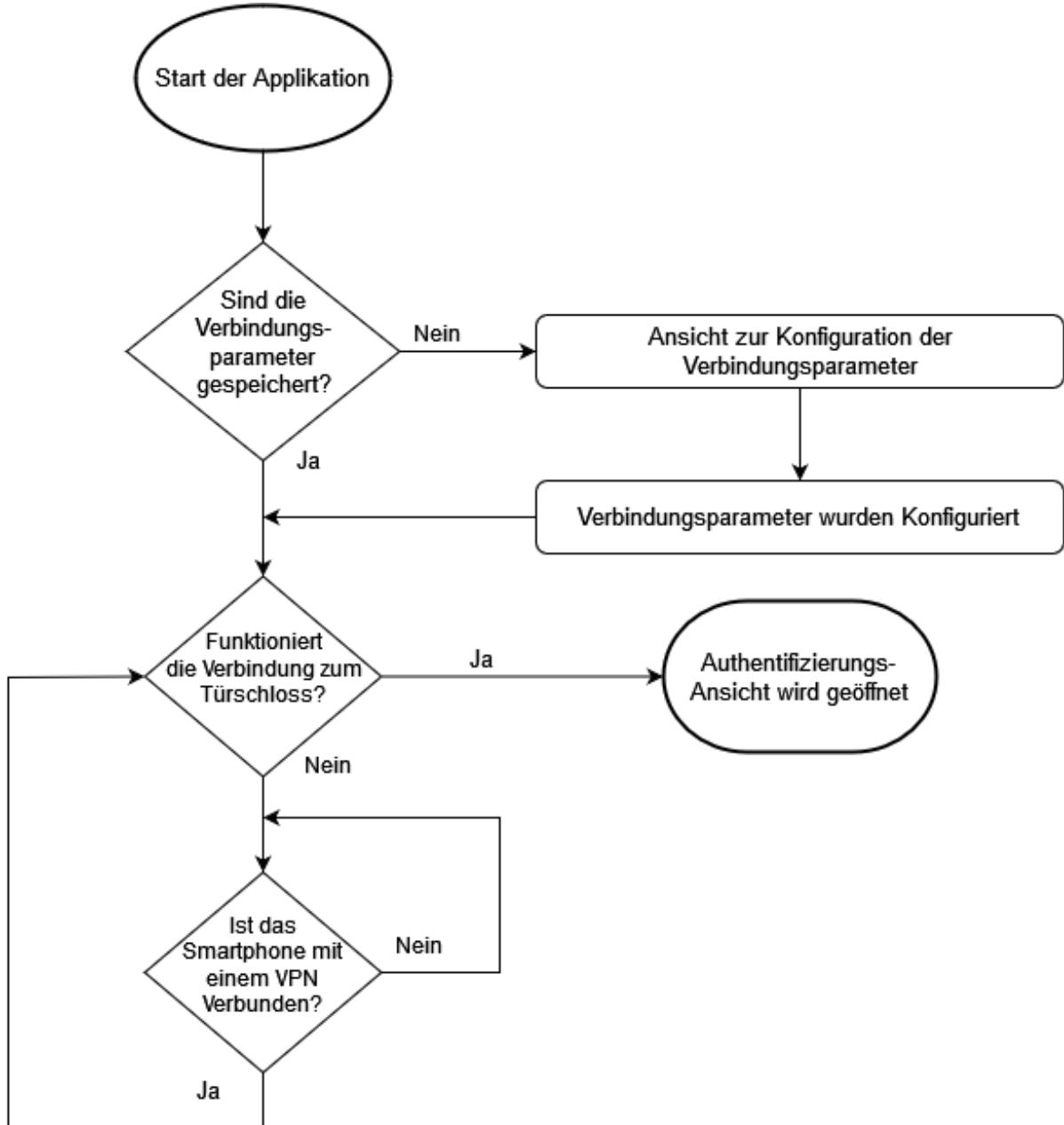


Abbildung 11.3.: Schritte der Überprüfung um eine Verbindung zum Türschloss herzustellen

Als erstes überprüft die Applikation nach ihrem Start, ob die Verbindungsparameter bereits lokal gespeichert sind. Ist das nicht der Fall, öffnet sich die Ansicht, um die Verbindungsparameter

11. Allgemeines

zu konfigurieren.

Im zweiten Schritt sendet die Applikation eine Testanfrage an die `/api>Status` Route, um die Verbindung zum Türschloss zu überprüfen. Befindet sich das Smartphone aktuell im selben Netzwerk wie der Raspberry PI, wird diese Testanfrage erfolgreich sein, und somit kann nun der Authentifizierungs-Bildschirm geöffnet werden.

Falls diese Anfrage scheitert, und sich das Smartphone somit nicht im Heimnetzwerk befindet, wird nun eine Ansicht geöffnet, in der der Nutzer mittels eines Buttons die Smartphone-Einstellungsseite öffnen kann, in der die aktive VPN-Verbindung verwaltet werden kann. Die App überprüft währenddessen laufend, ob sich der Nutzer mit einem VPN verbunden hat. In diesem Fall überprüft die Applikation anschließend wieder die Verbindung mittels einer Testanfrage.

11.1.4. Authentifizierung bei Start der App

Nachdem die App eine Verbindung zum Türschloss sichergestellt hat, öffnet sich die Authentifizierungs-Ansicht. Dort wird der User aufgefordert, das Master-Passwort des Türschlosses einzugeben.

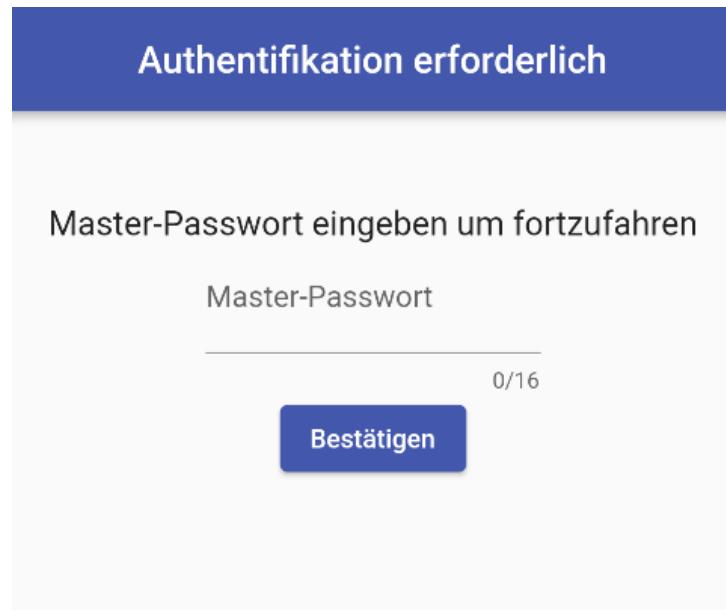


Abbildung 11.4.: Ansicht zur Authentifikation

Das eingegebene Master-Passwort wird dazu verwendet, um auf der `/api/Session/getToken/{secret}`-Route eine neue Session anzufragen. Zusätzlich dient diese Eingabe auch als Schutz der App vor unbefugtem Zugriff. Die App ist somit kein Angriffsvektor, da ohne eine aktive Session keine Befehle an das Türschloss gesendet werden können.

11. Allgemeines

11.1.5. Startbildschirm

Der Startbildschirm der Applikation wird nach erfolgreichem anfragen der Session aufgerufen. Dort lässt sich das Türschloss jederzeit öffnen bzw. schließen. Nachdem das Türschloss geöffnet wurde, bleibt es 24 Sekunden geöffnet, bevor es sich wieder automatisch schließt. Links befindet sich ein kleiner Knopf, mit dem eine Sidebar geöffnet werden kann, welche weitere Ansichten öffnen kann.



Abbildung 11.5.: Der Startbildschirm von Shulker-Mobile

11.1.6. Sidebar

Die Sidebar öffnet sich links am Bildschirm und gibt eine Übersicht der verschiedenen Ansichten, die geöffnet werden können.

11. Allgemeines



Abbildung 11.6.: Die Sidebar zeigt verschiedene Ansichten von Shulker-Mobile an.

11.1.7. PIN-Verwaltung

Die PIN-Verwaltungs-Ansicht bietet einen Überblick aller erstellten Pins.

11. Allgemeines

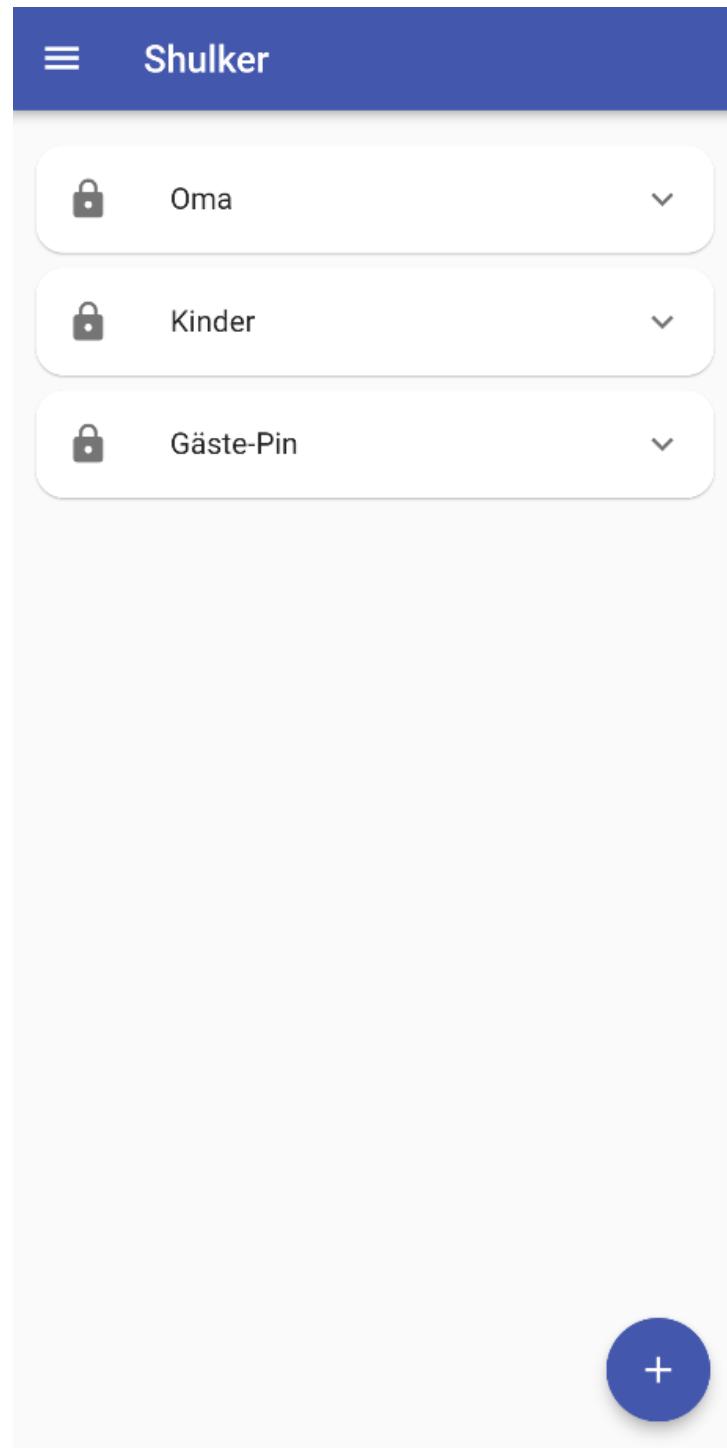


Abbildung 11.7.: Die PIN-Verwaltungs-Ansicht von Shulker-Mobile

Einzelne Pins können angeklickt werden, um weitere Informationen über diese anzuzeigen. Nachdem ein Pin angeklickt wurde, öffnet sich dieser. Nun lässt sich auslesen, wie oft dieser Pin noch verwendet werden kann, und in welchem Zeitraum dieser gültig ist. Außerdem lassen sich Pins von diesem Menü aus löschen.

11. Allgemeines

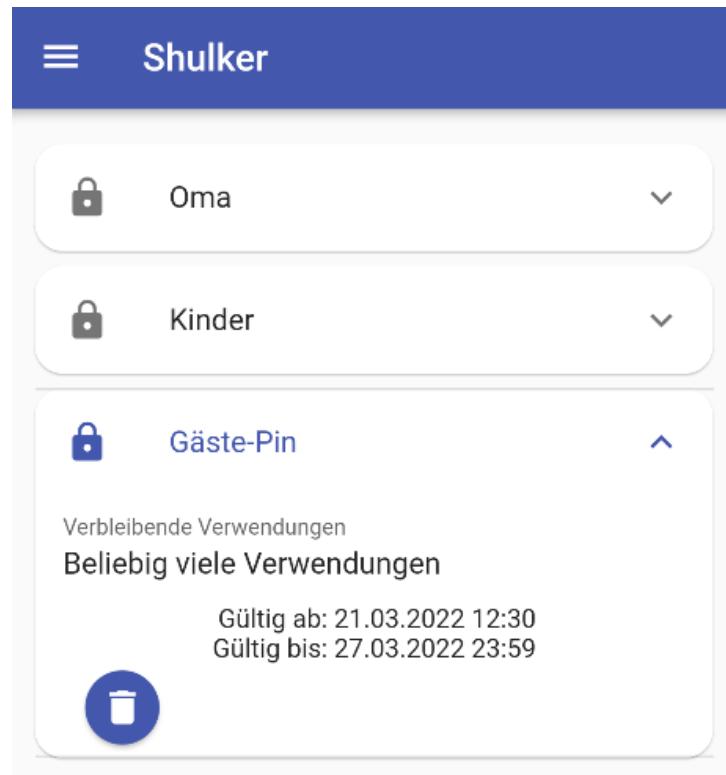


Abbildung 11.8.: Geöffnetes Menü nachdem ein Pin angeklickt wurde

11.1.8. Erstellen eines Pins

In der rechten unteren Ecke der PIN-Verwaltungs-Ansicht befindet sich ein *Floating Action Button*, der ein Fenster öffnet, um neue Pins zu erstellen.



Abbildung 11.9.: Knopf zum erstellen eines Pins

Nachdem dieser gedrückt wurde, öffnet sich ein Fenster, das einem Formular beinhaltet, um neue Pins zu erstellen.

11. Allgemeines

The screenshot shows a mobile application interface for adding a pin. At the top, there is a blue header bar with a back arrow, the title "Pin hinzufügen" (Add Pin) in white, and a blue button labeled "Erstellen" (Create). Below the header, there are several input fields:

- Name:** A text input field with a placeholder "Name". Below it, a small text indicates "0/30".
- Schlüssel:** A text input field with a placeholder "PIN Code". Below it, a small text indicates "0/16".
- PIN Code wiederholen:** A text input field with a placeholder "PIN Code wiederholen". Below it, a small text indicates "0/16".
- Gültigkeit:** A section for defining the validity period. It shows "vom 20.03.2022" and "für immer" (valid until never), with a time entry of "00:00". There is also a checked checkbox next to "für immer".
- Verwendungen:** A section for specifying usage settings. It includes "Beliebig viele Verwendungen" (Unlimited uses) with a checked checkbox and "Verbleibende Verwendungen" (Remaining uses).

Abbildung 11.10.: Fenster zum erstellen eines Pins

Dort muss dem Pin einen Namen gegeben werden. Dann kann der Schlüssel des Pins eingetragen werden, dieser muss zwei mal eingegeben werden, um sicherzustellen, dass sich der Nutzer bei der Eingabe nicht verschrieben hat. Alle Felder dieses Formulars beinhalten eine Validierung der Eingabe.

Zusätzlich kann dort noch die Gültigkeit und die Anzahl der Verwendungen angegeben werden. Um einen Zeitpunkt für die Gültigkeit zu definieren, muss dieser einfach angeklickt werden. Anschließend öffnet sich ein Fenster, in dem Datum bzw. Uhrzeit ausgewählt werden kann. Nachdem ein Pin hinzugefügt wurde, erscheint dieser in der Pin-Verwaltungs-Ansicht.

11. Allgemeines

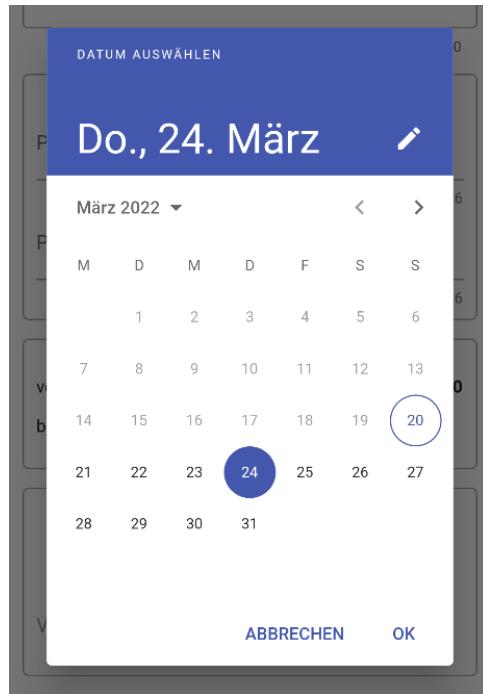


Abbildung 11.11.: Fenster zur Auswahl eines Datums.

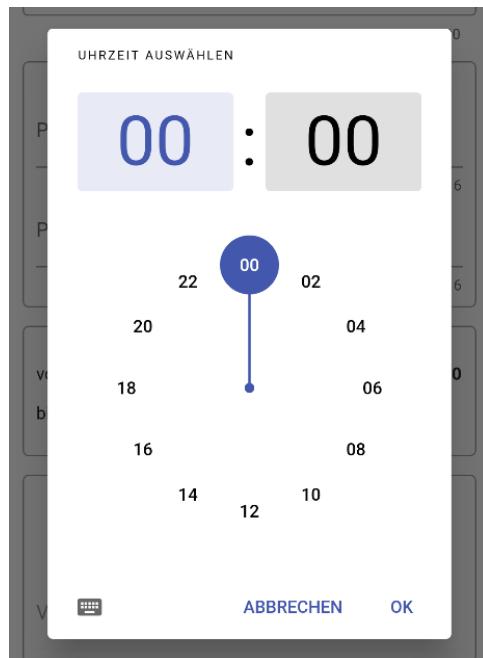


Abbildung 11.12.: Fenster zur Auswahl einer Uhrzeit.

11. Allgemeines

11.1.9. Einstellungen

Die Einstellungs-Ansicht ist von der Sidebar aus aufrufbar. Dort befindet sich ein Knopf, um das Türschloss neu zu Verbinden, und ein weiterer, um Informationen über die App, wie Versionsnummer und Lizenzen der verwendeten Software, anzuzeigen.

11. Allgemeines

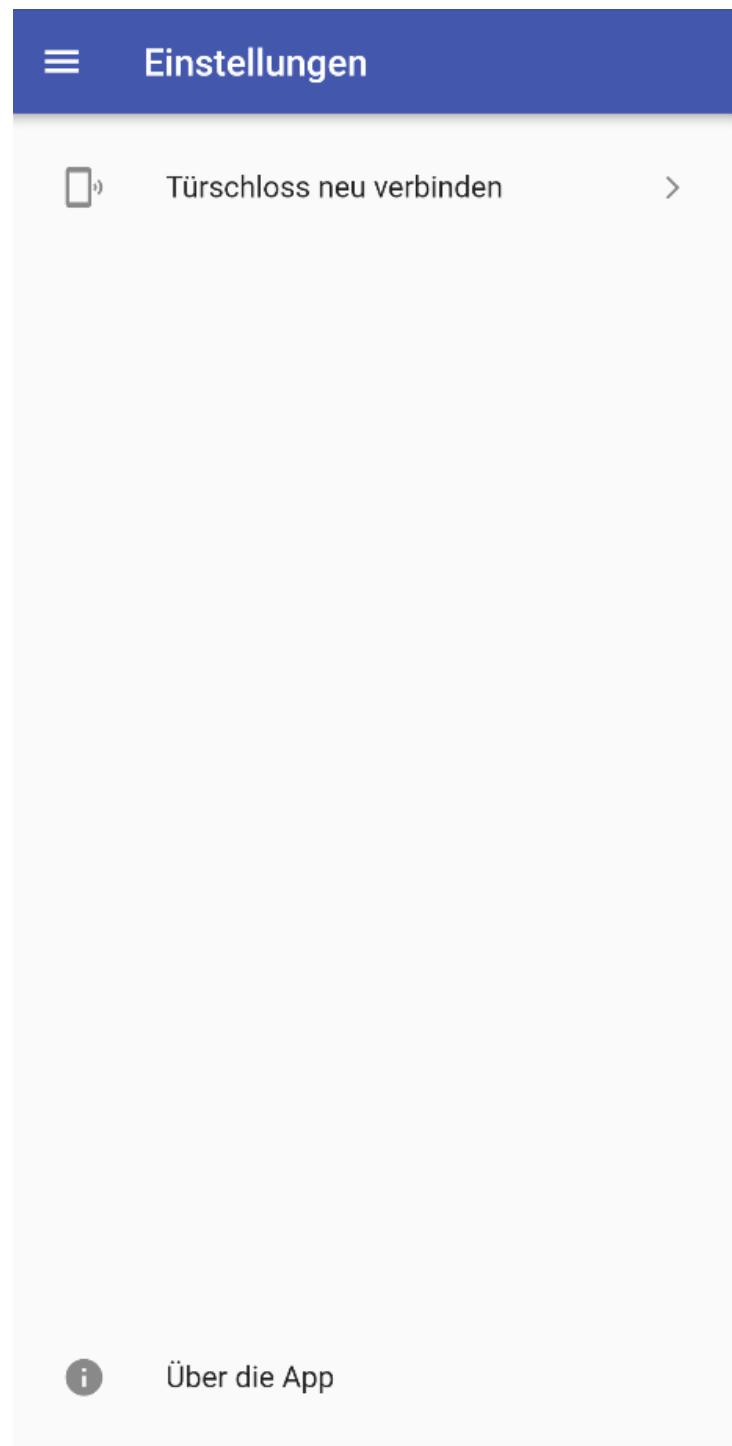


Abbildung 11.13.: Einstellungs-Ansicht

11. Allgemeines

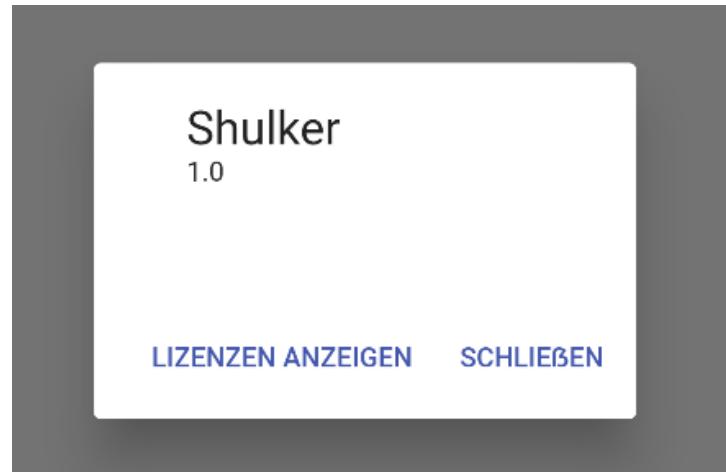


Abbildung 11.14.: Fenster, das Informationen über die App anzeigt. Wird von Flutter bereitgestellt.

Das Menü, das nach einem Klick auf den „Über die App“-Knopf aufgerufen wird, wird von Flutter bereitgestellt. Der Knopf „Lizenzen Anzeigen“ listet die Lizenzen der im Projekt verwendeten Software.

11. Allgemeines

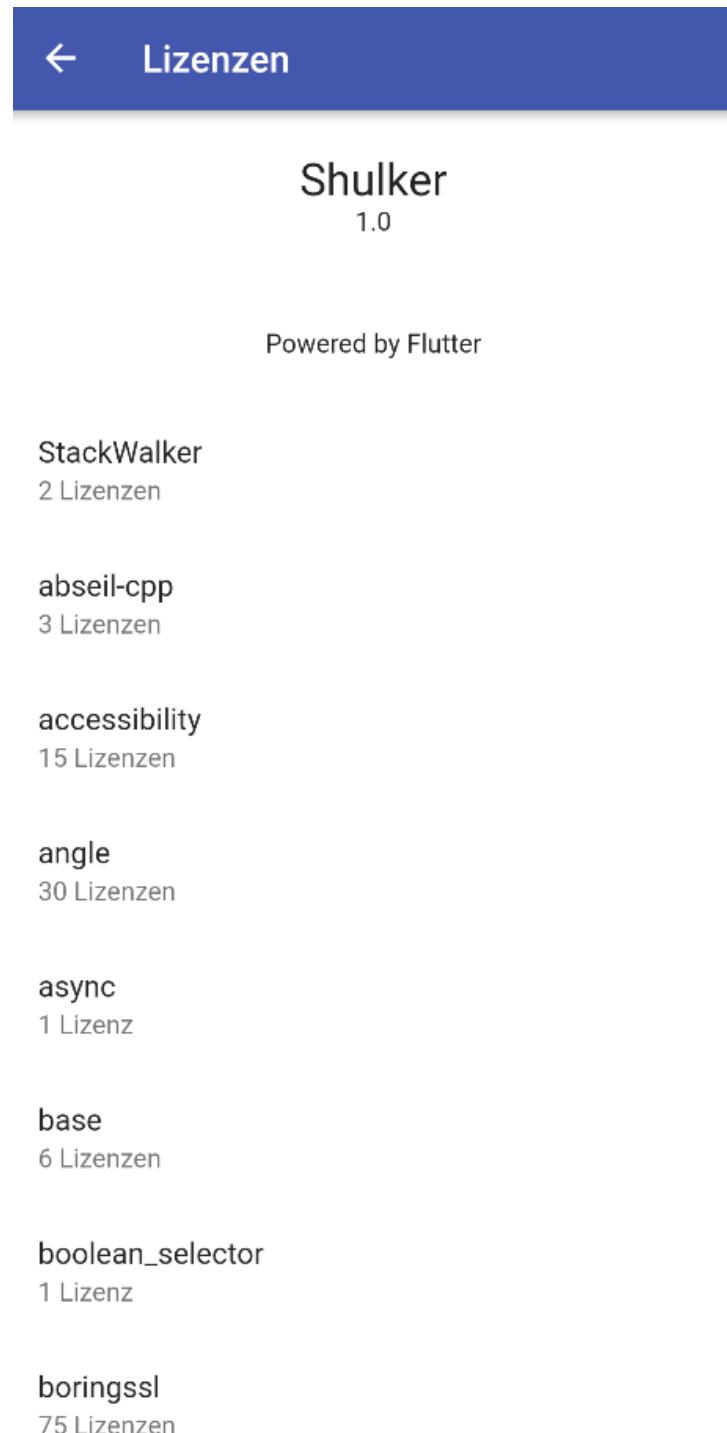


Abbildung 11.15.: Fenster, das Informationen über die App anzeigt. Wird von Flutter bereitgestellt.

Teil VI.

Vorzeigemodell

12. Allgemeines

Das Vorzeigemodell dient zur Veranschaulichung des gesamten Projekts. Es ist nicht als tatsächliche praktische Implementation gedacht, sondern als eine kleine Visualisierung. Natürlich dient es auch als Beweis dafür, dass das gesamte Shulker-System voll funktionsfähig und in der Realität anwendbar ist.

Das Modell wird nach Fertigstellung des Diplomprojekts in die HTL-Anichstraße gebracht und dort vorgestellt.

12.1. Planung des Grundgerüsts

In die Planung des Vorzeigemodells flossen verschiedene Faktoren ein. Die wichtigsten Entscheidungsgrundlagen lauten:

- Das Modell muss irgendwo gelagert und physisch in die HTL-Anichstraße gebracht werden. Um die Umständlichkeit dieses Vorgehens zu minimieren, wurde die Größe des Türrahmens in etwas kleineren Dimensionen realisiert.
- Das Modell dient nur als Veranschaulichung und Beweis der Funktionalität von Shulker. Daher müssen keine besonders aufwendige Sicherheitsmaßnahmen getroffen werden.
- Das Modell soll mit wenig Aufwand anpassbar und abänderbar sein. Die Wahl der Baumaterialien ist daher mit Bedacht zu wählen.
- Die Kosten des Modells müssen sich in Grenzen halten.

Aus den oben genannten Gründen entstanden folgende Spezifikationen für das Grundgerüst:

- Der Rahmen und das Türblatt der Modelltür wird aus Holz gefertigt.
- Der Rahmen hat Außenmaße von rund 80x50cm und eine Breite von rund 5cm.
- Das Türblatt sitzt bündig im Rahmen.
- Der Raspberry Pi und Teile der elektronischen Schaltung werden sich in einer Holzbox befinden.
- An der Vorderseite der Holzbox sitzt der Touchscreen.
- Die Holzbox wird seitlich an den Türrahmen befestigt.
- Die Außenmaße der Holzbox sind gleich den Außenmaßen des Touchscreens.

12.2. Planung der elektronischen Schaltung

Da Shulker allein nur die anliegende Spannung eines GPIO-Pins verwaltet, musste für das Vorzeigemodell eine elektronische Schaltung zur Ansteuerung eines elektronischen Schlosses entworfen und gebaut werden.

Die Planung der Schaltung war kein großer Aufwand. Es wurden nur grundlegende Dinge festgelegt:

- Es soll wenn möglich nur ein Netzteil für Schloss und Raspberry Pi zugleich benötigt werden.
- Das Schloss darf als Eingangsspannung nicht mehr als 12V benötigen.
- Der Raspberry Pi soll das Schloss mittels eines Relais ansteuern.
- Die Kosten der Komponenten müssen sich in Grenzen halten.
- Die Schaltung muss so simpel wie möglich gestaltet sein.

13. Tür und Box

13.1. Herstellung der einzelnen Teile

13.1.1. Türblatt

Das Türblatt besteht aus einer einzelnen Holzplatte. Die genauen Maße dieser Holzplatte betragen 35x47cm. Diese Holzplatte wurde aus dem Holzverschnitt einer herkömmlichen Baumarkt-Handelskette erworben. Die Kosten des Türblatts betragen 6€.

Es mussten keine weiteren Änderungen am Türblatt vorgenommen werden. Wir entschieden uns, den Rahmen an das Türblatt anzupassen.

13.1.2. Türrahmen

Um den Türrahmen herzustellen, wurde zuerst weiteres Holz besorgt. Dieses Holzstück hatte bereits weitgehend geeignete Maße. Es war ein langes Brett, aus dem die vier Seiten des Türrahmens gebildet werden konnten. Um vom Ausgangsholz bis hin zum fertigen Türrahmen zu gelangen, wurden folgende Schritte getätigt:

1. **Vierkanthobel:** Zuerst wurde das Werkstück an allen vier Seiten gleichzeitig mithilfe eines Vierkanthobels bearbeitet. Dadurch wurde die Oberfläche der Werkstücke geglättet und abgerichtet.
2. **Stufe schneiden:** Da das Türblatt mit dem Türrahmen bündig sein soll, also ein stumpf einschlagendes Türblatt sein soll, muss in den Türrahmen eine Stufe geschnitten werden. Ansonsten würde nichts das Türblatt beim Schließen der Tür stoppen.
3. **Zuschneiden:** Als nächstes wurde das Werkstück in vier Teile auf Maß zugeschnitten. Aus Schönheitsgründen wurden die Werkstücke in Gehrung geschnitten. Das bedeutet, dass die zwei Eckstücke mit 45° zugeschnitten werden, damit sie dann genau aneinander liegen können, wenn sie zu einem Rahmen zusammengeführt werden.
4. **Verleimen und Verschrauben:** Die vier Teile des Rahmens wurden nun zusammengeführt und verleimt. Zusätzlich wurden sie mit Holzbauschrauben gesichert, um das ganze Konstrukt robuster zu machen.

Der Türrahmen war somit fertig.

13.1.3. Holzbox

Da die Holzbox ein recht kleines Werkstück ist, entschieden wir uns dafür, kleine Sperrholzplatten als Ausgangsmaterial zu verwenden. Diese Sperrholzplatten hatten die Maße 20x30x4cm und waren für die Bearbeitung mit einer Laubsäge geeignet. Wir entschieden uns aus Genauigkeits-

13. Tür und Box

gründen aber gegen die Verwendung einer Laubsäge. Wir entschieden uns für das Verwenden einer Kreissäge.

Die Herstellung der Holzbox kann ebenfalls in einfache Schritte unterteilt werden:

1. **Eroieren der Maße:** Als erstes mussten wir die genauen Maße der Holzbox festlegen. Da das Touchdisplay direkt als Vorderseite der Box fungieren soll, mussten wir die Maße auf das Display anpassen. Das Display hat die Maße 19,5*12,5cm.
2. **Aufzeichnen der Schnitte:** Bevor wir die Sperrholzplatten zuschneiden konnten, mussten wir natürlich die gewollten Schnitte auf das Holz aufzeichnen, um für eine gewisse Genauigkeit zu sorgen. Dies setzten wir mit einem schwarzen Adding-Stift um.
3. **Ausschneiden:** Das Ausschneiden mit der Kreissäge war nun ein Kinderspiel. Trotzdem ist beim Arbeiten mit einer Kreissäge enorm auf die eigene Sicherheit zu achten.
4. **Zusammenleimen:** Nun leimten wir die ausgeschnittenen Teile zu einer Box zusammen. Da die Holzplatten sehr dünn sind, machte ein Gehrungsschnitt keinen Sinn. Wir leimten sie normal aufeinander zusammen. Das Ergebnis ist eine Holzbox ohne Boden und Decke. Ein Loch für den Touchscreen, das andere um in die Box greifen zu können.
5. **Hinzufügen der Klappe:** Um die Box öffnen und schließen zu können, mussten wir noch eine Holzklappe hinzufügen. Das machten wir einfach mit einer weiteren Holzplatte. Als Scharnier fungiert hierbei starkes Klebeband, da es völlig ausreichend ist.

Nun fehlte nurnoch das Einsetzen des Touchscreens.

13.2. Vollendung der Modelltür

Zur Vollendung der Modelltür mussten nur noch das Zusammenfügen von Türrahmen und Türblatt erfolgen. Da das Türblatt bündig mit dem Türrahmen ist, ließen sich leicht herkömmliche Scharniere aus Metall als Türscharniere verwenden.

Um die Tür auch ohne Strom schließbar zu machen, haben wir einen kleinen Türriegel hinzugefügt. Alle Komponenten wurden mit Schrauben auf dem Türrahmen und dem Türblatt befestigt.

Die Modelltür wurde somit erfolgreich produziert.

13. Tür und Box

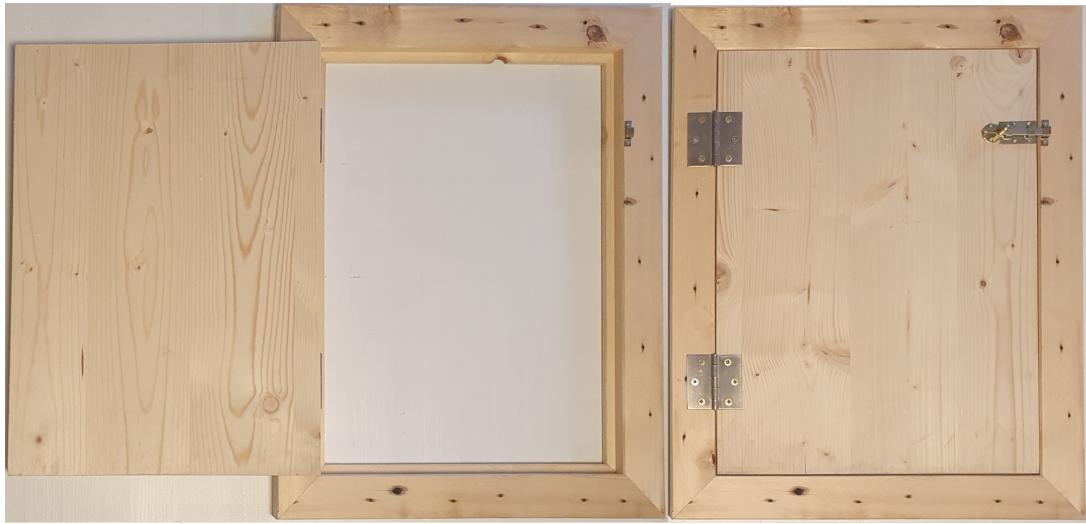


Abbildung 13.1.: Modelltür; geöffnet und geschlossen

13.3. Vollendung der Holzbox

Um das Aussehen der Holzbox etwas aufzuschmücken, haben wir schwarzes Klebeband um die Box gewickelt. Dies dient auch zur Verbesserung der Stabilität. Das Display wurde mithilfe von doppelseitigen Klebeband befestigt. Dies ermöglicht leichtes wieder-abnehmen des Bildschirms. Das ist äußerst nützlich, falls später noch etwas am Display geändert werden muss. Zusätzlich ist der Touchscreen innen mit Klebeband befestigt, um stärkeren Halt zu bieten.

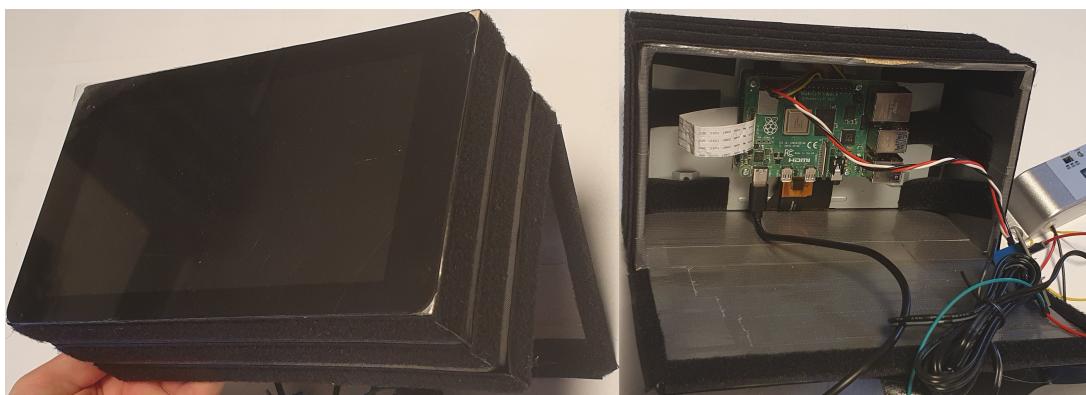


Abbildung 13.2.: Box in zwei Ansichten

14. Elektronische Schaltung

Die elektronische Schaltung im Vorzeigemodell dient dazu, das Signal des Raspberry Pi GPIO-Pins in das Öffnen und Schließen eines elektronischen Schlosses zu verwandeln.

14.1. Das Schloss

Bevor man mit der Entwicklung der Schaltung beginnen kann, muss man sich für ein geeignetes Schloss entscheiden.

Als elektronisches Schloss wurde ein Bolzenschloss von der Marke "LIBO Smart Home" gewählt. Das Schloss arbeitet mit 12V Gleichstrom und ist somit für unser Vorzeigemodell geeignet. Gefertigt ist es aus einer Aluminiumlegierung. Die vorgefertigten Löcher machen es leicht, das Schloss an das Türblatt und den Türrahmen zu befestigen.

Das Bolzenschloss schließt, wenn es an elektrischen Strom anliegt. Das bedeutet, dass das Schloss bei einem Stromausfall öffnen würde. Das ist vollkommen in Ordnung, da das Vorzeigemodell keine hohe Sicherheit gewährleisten muss.

14.2. Die Schaltung

Damit der Raspberry Pi mit einem 5V GPIO-Pin ein 12V Schloss ansteuern kann, wird ein Relais verwendet. Ein Relais ist ein Schalter, welcher durch elektrischen Strom betätigt wird. Damit kann der Raspberry Pi den Stromkreis des elektronischen Schlosses schließen und öffnen und somit auch das Schloss.

Da es um einiges angenehmer ist, nur ein Netzteil in eine Steckdose einstecken zu müssen, haben wir uns entschieden, ein Netzteil mit 12V Ausgangsspannung für das Schloss und den Raspberry Pi zu verwenden. Da der Raspberry Pi aber mit 5V arbeitet, muss ein Step-Down-Konverter zwischengeschaltet werden, der die 12V Spannung in eine 5V Spannung umwandelt.

14. Elektronische Schaltung

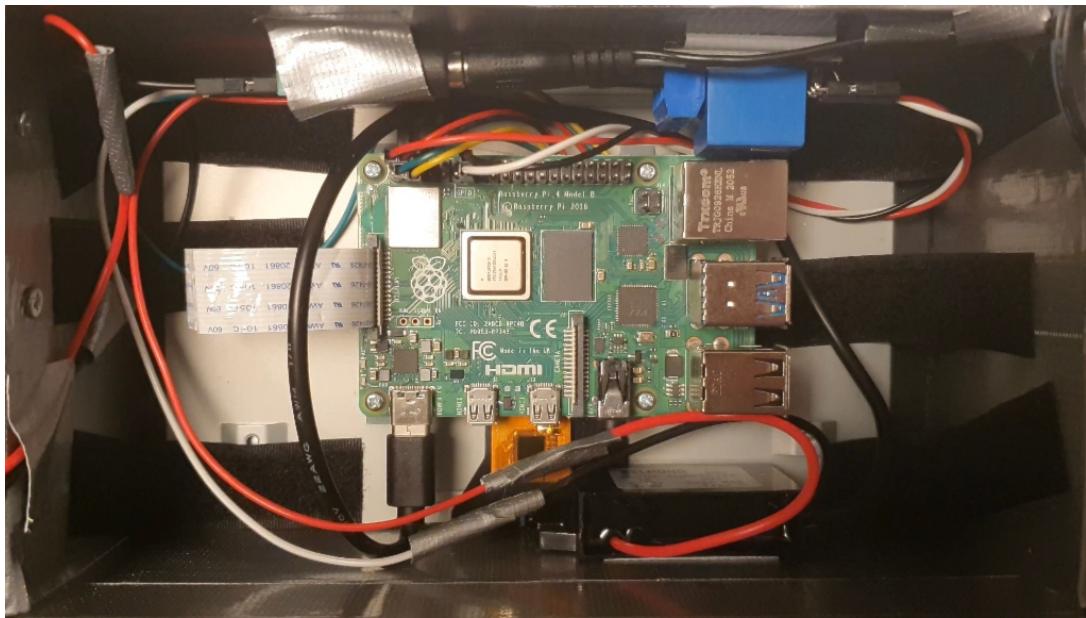


Abbildung 14.1.: Die elektronische Schaltung innerhalb der Box

15. Fertigstellung

Alle einzelnen Komponenten sind nun bereits fertiggestellt. Der Raspberry Pi sitzt sicher hinter dem Touchscreen, welcher sicher auf der Box liegt. Der Rahmen und das Türblatt bilden die Modelltür und die elektronische Schaltung ist ebenso voll funktionsfähig.

15.1. Zusammenführung

Das Schloss wurde mit Schrauben an den Türrahmen und das Türblatt befestigt. Der nächste Schritt war es, mit kleinen Schrauben die Elektronik-beinhaltende Box seitlich an den Türrahmen zu montieren. Als letztes musst das elektronische Schloss an den GPIO-Pin des Raspberry Pis angeschlossen werden.

Das Vorzeigemodell war somit vollendet und voll funktionsfähig.

15. Fertigstellung

15.2. Das Endprodukt

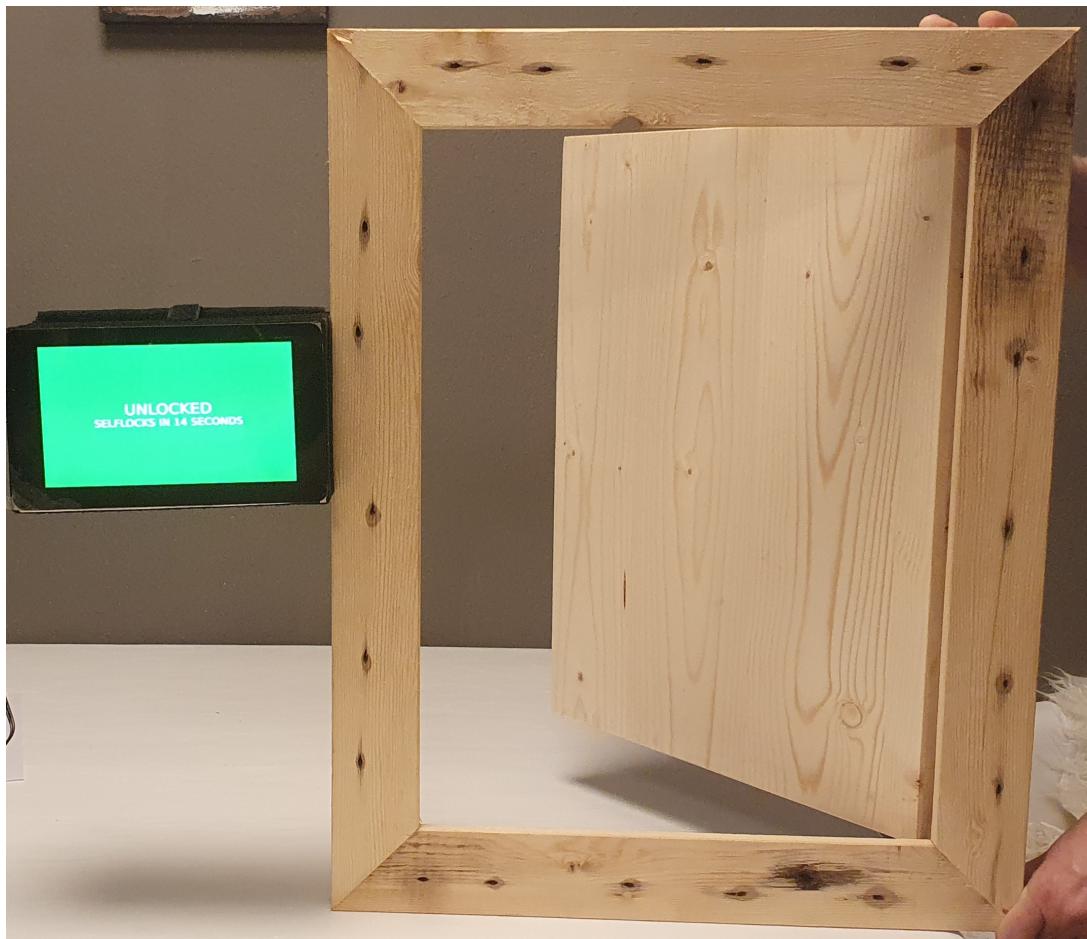


Abbildung 15.1.: Das Vorzeigemodell

Teil VII.

Appendix

16. Appendix

16.1. Zeitaufwand

Der gearbeitete Zeitaufwand für Shulker ist in folgenden Diagrammen ersichtlich:

16.1.1. Zeitaufwand Alexander Heim

Zeitaufwand

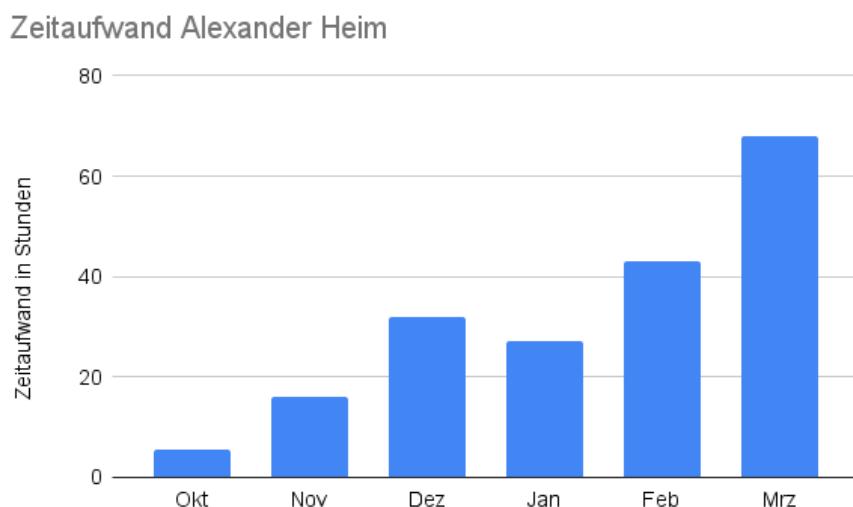


Abbildung 16.1.: Arbeitsstunden von Alexander Heim verteilt auf die Monate.

Verteilung der Arbeitszeit

16. Appendix

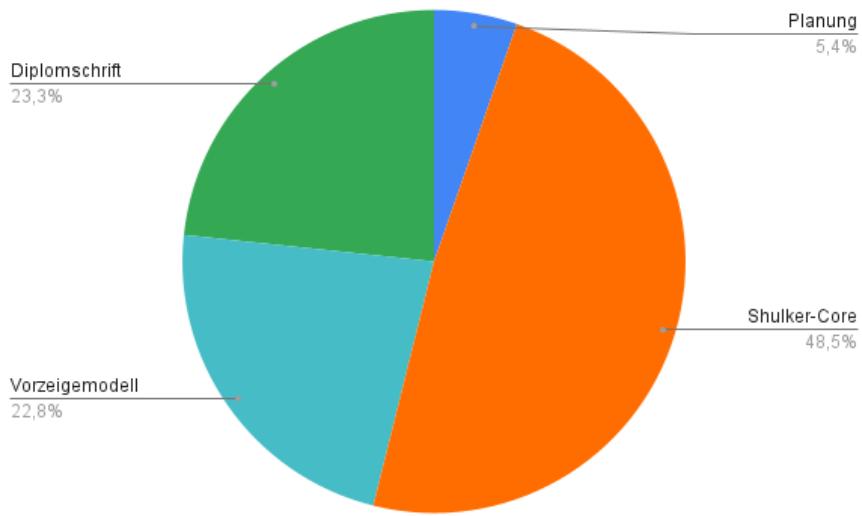


Abbildung 16.2.: Verteilung der Arbeitszeit vom Alexander Heim

16.1.2. Zeitaufwand Moritz Laichner

Zeitaufwand

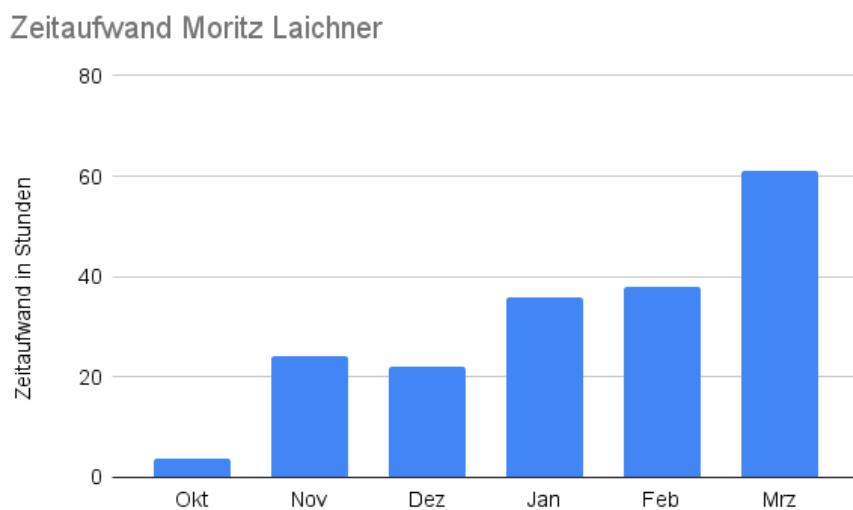


Abbildung 16.3.: Arbeitsstunden von Moritz Laichner verteilt auf die Monate.

Verteilung der Arbeitszeit

16. Appendix

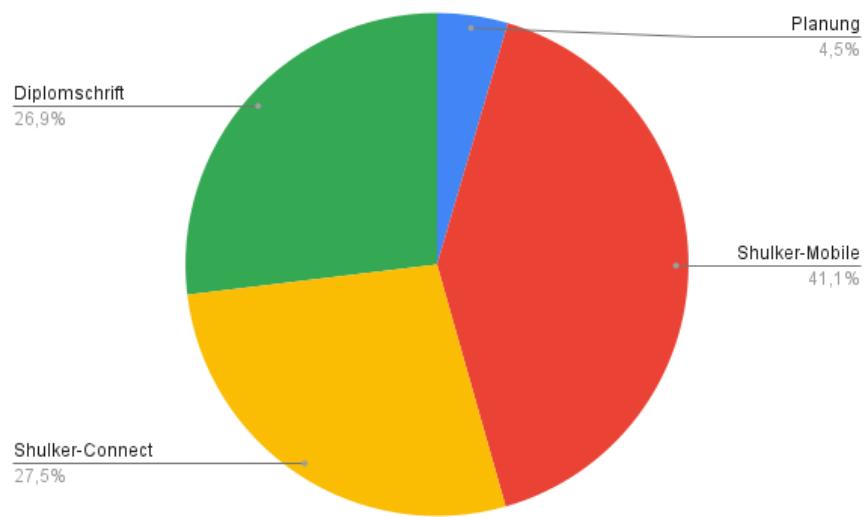


Abbildung 16.4.: Verteilung der Arbeitszeit vom Moritz Laichner

Literaturverzeichnis

- [1] Minecraft Fandom Wiki User-10384423. Violetter Shulker. https://minecraft.fandom.com/de/wiki/Shulker?file=Violetter_Shulker.png, 2015. Abgerufen am 08. März 2022.
- [2] Rust Community Steve Klabnik, Carol Nichols. Rust programmiersprache. <https://doc.rust-lang.org/book/ch00-00-introduction.html>, 2022. Abgerufen am 15. März 2022.
- [3] Rust Community Steve Klabnik, Carol Nichols. Rust book ownership. <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>, 2022. Abgerufen am 15. März 2022.
- [4] The Chromium Project. Chromium speicherbugs. <https://www.chromium.org/Home/chromium-security/memory-safety/>, 2022. Abgerufen am 15. März 2022.
- [5] Rust Community Steve Klabnik, Carol Nichols. Rust book ownership-regeln. <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-rules>, 2022. Abgerufen am 15. März 2022.
- [6] Microsoft. A tour of the c sharp language. <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>, 2022. Abgerufen am 14. März 2022.
- [7] Microsoft. What is dotnet. <https://docs.microsoft.com/en-us/dotnet/core/introduction>, 2022. Abgerufen am 14. März 2022.
- [8] Wikipedia. Flutter software. [https://en.wikipedia.org/wiki/Flutter_\(software\)](https://en.wikipedia.org/wiki/Flutter_(software)), 2022. Abgerufen am 15. März 2022.
- [9] Wikipedia. Dart programming language. [https://en.wikipedia.org/wiki/Dart_\(programming_language\)](https://en.wikipedia.org/wiki/Dart_(programming_language)), 2022. Abgerufen am 15. März 2022.
- [10] Wikipedia. Flutter software. [https://de.wikipedia.org/wiki/Flutter_\(Software\)](https://de.wikipedia.org/wiki/Flutter_(Software)), 2022. Abgerufen am 16. März 2022.
- [11] Flutter. StatelessWidget class. <https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html>, 2022. Abgerufen am 16. März 2022.
- [12] Kontributoren Simon Hausmann, Olivier Goffart. Slint funktionsweise. <https://github.com/slint-ui/slint>, 2022. Abgerufen am 17. März 2022.
- [13] PHC-Team. Password hashing competition. <https://www.password-hashing.net/>, 2019. Abgerufen am 18. März 2022.
- [14] PHC-Team. Phc-string-format. <https://github.com/P-H-C/phc-string-format/blob/master/phc-sf-spec.md>, 2021.

Literaturverzeichnis

- [15] Wikipedia. Berkeley sockets. https://en.wikipedia.org/wiki/Berkeley_sockets#Client-server_example_using_TCP, 2022. Abgerufen am 18. März 2022.
- [16] Wikipedia. C-Sharp. <https://de.wikipedia.org/wiki/C-Sharp>, 2022. Abgerufen am 09. März 2022.
- [17] Flutter. StatefulWidget class. [https://api.flutter.dev/flutter/widgets/ StatefulWidget-class.html](https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html), 2022. Abgerufen am 17. März 2022.

Abbildungsverzeichnis

1.1.	Die Kreatur Shulker aus dem Videospiel <i>Minecraft</i>	13
2.1.	Aufbau von Shulker	14
3.1.	Veranschaulichung eines Rust-Strings	20
3.2.	Bild zur Veranschaulichung eines kopierten Rust-Strings	21
3.3.	Bild zur Veranschaulichung eines geklonten Rust-Strings	22
4.1.	Informationsfluss in Shulker-Core	32
5.1.	Aufbau und Funktionsweise von slint [12]	35
5.2.	Core-UI mit den verschiedenen Hauptansichten: PIN, PW und M	36
6.1.	Beispiel eines PHC-Strings	39
8.1.	Ablauf der Abfrage zum generieren einer neuen Session	43
9.1.	Rückgabe der Route	47
11.1.	Optionen zum verbinden des Türschlosses	54
11.2.	Eingabefelder um das Türschloss manuell zu verbinden	55
11.3.	Schritte der Überprüfung um eine Verbindung zum Türschloss herzustellen	56
11.4.	Ansicht zur Authentifikation	57
11.5.	Der Startbildschirm von Shulker-Mobile	58
11.6.	Die Sidebar zeigt verschiedene Ansichten von Shulker-Mobile an.	59
11.7.	Die PIN-Verwaltungs-Ansicht von Shulker-Mobile	60
11.8.	Geöffnetes Menü nachdem ein Pin angeklickt wurde	61
11.9.	Knopf zum erstellen eines Pins	61
11.10	Fenster zum erstellen eines Pins	62
11.11	Fenster zur Auswahl eines Datums.	63
11.12	Fenster zur Auswahl einer Uhrzeit.	63
11.13	Einstellungs-Ansicht	65
11.14	Fenster, das Informationen über die App anzeigt. Wird von Flutter bereitgestellt.	66
11.15	Fenster, das Informationen über die App anzeigt. Wird von Flutter bereitgestellt.	67
13.1.	Modelltür; geöffnet und geschlossen	73
13.2.	Box in zwei Ansichten	73
14.1.	Die elektronische Schaltung innerhalb der Box	75
15.1.	Das Vorzeigemodell	77
16.1.	Arbeitsstunden von Alexander Heim verteilt auf die Monate.	79

Abbildungsverzeichnis

16.2. Verteilung der Arbeitszeit vom Alexander Heim	80
16.3. Arbeitsstunden von Moritz Laichner verteilt auf die Monate.	80
16.4. Verteilung der Arbeitszeit vom Moritz Laichner	81

List of Codes