

Diplomarbeit



Sokka

Ein modernes Bestellsystem für Restaurants

erstellt von

Nicolaus Rossi
Joshua Winkler



HTBLuVA
Innsbruck Anichstrasse

Betreuer:
Sabo Rubner

2020/21

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 xx.xx.2021

Verfasser/Verfasserinnen:

Nicolaus Rossi

Joshua Winkler

Projektteam



Nicolaus Rossi

Adresse

PLZ Ort

Tel: -

E-Mail: nirossi@tsn.at



Joshua Winkler

Adresse

PLZ Ort

Tel: -

E-Mail: jos.winkler@tsn.at

Betreuer



Bakk. Sabo Rubner

HTBLuVA Innsbruck Anichstraße

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

Danksagung

An dieser Stelle möchten wir uns bei allen bedanken, die uns während der Ausarbeitung unserer Diplomarbeit unterstützt und motiviert haben.

Unser größter Dank gebührt unserem Betreuer Herr Prof. Sabo Rubner. Er hat von Beginn an uns geglaubt, unsere Arbeit regelmäßig begutachtet und uns mit seiner fachlichen Expertise hilfreiche Kritik und Tipps für die Umsetzung sowie einen großen Freiraum für unsere eigenen Ideen und Vorstellungen gegeben.

Wir bedanken uns bei Herr Prof. Andreas Reimair für seine Hilfe bei der Themenfindung unserer Diplomarbeit und für seine Erklärungen des gesamten Ablaufs eines solchen Projekts.

Ebenfalls bedanken wir uns bei unserem guten Freund Timon Schenkenfelder für die Erstellung unseres Projektlogos, für seine Tipps zum Design unserer App sowie bei unserem Freund Kai Fink für das Bereitstellen eines virtuellen Servers, der zum Testen von Sokka verwendet wurde.

Abschließend möchten wir uns bei unseren Eltern aufrichtig bedanken, die uns in gerade dieser besonderen Zeit mit ihrer Unterstützung sowie dem Korrekturlesen unserer Diplomschrift zur Seite standen.

Nicolaus Rossi & Joshua Winkler

Innsbruck, 18. März 2021

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

We live in a society where each year an estimate of 1.3 billion tons of food is wasted. A huge part occurs in restaurants with all-you-can-eat buffets or shops that offer buy-one-get-one-free deals on a frequent basis. [1]

This diploma project revolves around finding a remedy to mitigate food waste in restaurants and canteens using modern technology.

Sokka aims to resolve a part of this major issue by giving customers the opportunity to pre-order their food on the day before by just using their smartphone. This enables chefs to more precisely estimate how much food they need to prepare and hence the number of ingredients that have to be purchased.

The result is a fully functioning ordering and payment system with an intuitive user interface for customers and an easy to use control panel for restaurant management.

Sokka is an open source project (FOSS). The project's source code is available online on GitHub, licensed under the terms of the GNU GPLv3 Open Source license.

github.com/htl-anichstrasse/sokka

Zusammenfassung

Bis zu 1,3 Millionen Tonnen Essen werden jedes Jahr einfach weggeworfen. Ein großer Teil dieser Verschwendungen stammt aus nicht verkauften Lebensmitteln in Restaurants und im Einzelhandel. [1] Diese Diplomarbeit hat das Ziel eine Linderung für dieses schwerwiegende Problem der heutigen Zeit durch die Nutzung moderner Technologie zu finden.

Hinter dem Namen **Sokka** verbirgt sich ein ökologisches Bestellsystem für Kantinen, das den technischen Ansprüchen des 21. Jahrhunderts gerecht wird. *Sokka* erlaubt es Kunden einer Kantine, ihr gewünschtes Essen bereits am Vortag über eine einfache Smartphone-App zu bestellen, um dann am darauffolgenden Tag ihre Bestellung durch das Vorzeigen eines von der App generierten Codes abholen zu können.

Wenn alle Kunden einer Kantine *Sokka* nutzen, ist es den Köchen der Kantine möglich genau einzuschätzen, welche Menge an Essen sie für den kommenden Tag zubereiten müssen. So bleibt im Optimalfall kein Essen übrig und es muss nichts weggeworfen werden.

Sokka ist ein Open-Source-Projekt (FOSS). Der Projektquellcode ist online auf GitHub unter den Lizenzbestimmungen der GNU-GPLv3-Lizenz verfügbar.
github.com/htl-anichstrasse/sokka

Inhaltsverzeichnis

I. Intro	13
1. Meta	14
1.1. Hintergrund	14
1.2. Namensherkunft	14
1.3. Projektlogo	15
2. Aufbau	16
2.1. Komponenten	16
2.2. Testumgebung	16
II. Theoretische Grundlagen	18
3. Verwendete Programmiersprachen	19
3.1. JavaScript	19
3.1.1. Kritik an JavaScript	19
3.2. TypeScript	19
3.3. Dart	20
3.3.1. Was ist Dart?	20
3.3.2. Syntax von Dart	20
4. Haupttechnologien	27
4.1. Docker	27
4.1.1. Images	27
4.1.2. Container	28
4.1.3. Volume	28
4.2. Node.js	29
4.3. React	30
4.3.1. JSX	30
4.3.2. Components	30
4.3.3. States	31
4.4. Flutter	32
4.4.1. Was macht Flutter besonders?	32
4.4.2. Layoutting in Flutter	34
4.4.3. Rendern von Widgets	35
4.4.4. Interaktivität der App	36
4.4.5. Navigation und Routing durch die App	38

Inhaltsverzeichnis

5. Dependencies und Libraries	40
5.1. Gulp.js	40
5.1.1. gulpfile.js	40
5.1.2. Rolle von Gulp in Sokka	40
5.2. Express.js	40
5.2.1. Das Express „Hello world!“	41
5.2.2. Rolle von Express in Sokka	41
 III. Backend	 42
6. Allgemeines	43
6.1. Datenbankverbindung	43
6.2. Konfiguration	43
6.2.1. Verfügbare Konfigurationen	43
6.2.2. Docker Secrets	44
6.3. REST-API	44
6.3.1. Bilder	45
6.3.2. Verifikation	45
7. REST-Route-Dokumentation	46
7.1. Authorization	46
7.2. Rate-Limiting	47
7.3. Root-Routes	48
7.3.1. /image	48
7.3.2. /verify	49
7.4. ACP	50
7.4.1. /acp/acpuser/get	50
7.4.2. /acp/acpuser/create	51
7.4.3. /acp/acpuser/delete	52
7.4.4. /acp/config/get	53
7.4.5. /acp/config/update	54
7.4.6. /acp/group/get	55
7.4.7. /acp/group/create	56
7.4.8. /acp/group/update	57
7.4.9. /acp/group/delete	58
7.4.10. /acp/menu/get	59
7.4.11. /acp/menu/create	60
7.4.12. /acp/menu/update	61
7.4.13. /acp/menu/delete	62
7.4.14. /acp/menu/title/get	63
7.4.15. /acp/menu/category/get	64
7.4.16. /acp/order/get	65
7.4.17. /acp/order/validate	66
7.4.18. /acp/order/invalidate	68
7.4.19. /acp/product/get	69
7.4.20. /acp/product/create	70
7.4.21. /acp/product/update	71

Inhaltsverzeichnis

7.4.22. /acp/product/delete	72
7.4.23. /acp/product/category/get	73
7.4.24. /acp/user/get	74
7.4.25. /acp/user/update	75
7.4.26. /acp/user/delete	76
7.4.27. /acp/login	77
7.4.28. /acp/logout	78
7.4.29. /acp/validate	79
7.4.30. /acp/image	80
7.4.31. /acp/status	81
7.5. Menu	82
7.5.1. /menu/get	82
7.5.2. /menu/title/get	83
7.6. Product	84
7.6.1. /product/get	84
7.7. Order	85
7.7.1. /order/get	85
7.7.2. /order/create	86
7.8. User	87
7.8.1. /user/create	87
7.8.2. /user/login	89
7.8.3. /user/logout	90
7.8.4. /user/validate	91
7.8.5. /acp/user/delete	92
7.8.6. /user/request	93
IV. ACP	94
8. Allgemeines zum ACP	95
8.1. Anmeldung	95
8.2. User-System	96
8.3. Routing	96
9. Seiten	98
9.1. Home	98
9.2. Products	99
9.3. Menus	100
9.4. Users	102
9.5. Groups	103
9.6. Orders	104
9.7. Konfiguration	105
V. Client App	106
10. Der Sokka-Mobile-Client	107
10.1. Was ist der Client?	107

Inhaltsverzeichnis

11. Utility	108
11.1. Allgemeines	108
11.2. Network-Wrapper	108
11.3. Cookie-Storage	110
11.4. Controllers	111
11.4.1. Models für Controller	112
11.4.2. Menu-Controller	113
11.4.3. Product-Controller	113
11.4.4. ShoppingBasket-Controller	113
11.4.5. Order-Controller	113
12. App-Services	114
12.1. Allgemeines	114
12.2. User-Authentication	114
12.2.1. Registrieren eines neuen Nutzers	114
12.2.2. Anmelden eines vorhandenen Nutzers	115
12.2.3. Abmelden eines Nutzers	115
12.2.4. Validieren einer Nutzer-Session	116
12.3. Bearer-Authentication	117
12.3.1. Allgemeines	117
12.3.2. Generieren eines Bearer-Tokens	117
12.4. FetchOrderables-Service	118
12.4.1. Allgemeines	118
12.4.2. Laden von verfügbaren Menüs	119
12.4.3. Laden von verfügbaren Produkten	119
12.5. ManageOrders-Service	120
12.5.1. Allgemeines	120
12.5.2. Generieren einer neuen Bestellung	120
12.5.3. Abfragen der Bestellungen eines Nutzers	122
13. Screens	124
13.1. Allgemeines	124
13.2. Authentication-Screens	124
13.2.1. LoginScreen	124
13.2.2. SignUpScreen	126
13.2.3. LoadingSplashScreen	126
13.3. HomeScreen	127
14. Views	129
14.1. Allgemeines	129
14.2. MenuView	129
14.2.1. Render-Widget für Menü-Informationen	129
14.2.2. Rendern der Menu-Cards für alle verfügbaren Menüs	130
14.3. ProductView	131
14.3.1. Render-Widget für einzelne Produkte	131
14.3.2. Darstellen der Product-Tiles	132
14.4. BasketView	133
14.4.1. Dismissible-Widget für Ware	133

Inhaltsverzeichnis

14.4.2. Darstellen des Warenkorbs	133
14.5. OrderView	136
14.5.1. Render-Widget für Bestellungen	136
14.5.2. Darstellen der Bestellungen im Order-View	139
VI. Admin-Client	140
15. Der Sokka-Admin-Client	141
15.1. Was ist der Admin-Client?	141
16. Admin-Client Utility	142
16.1. Allgemeines	142
16.2. Network-Wrapper & Cookie-Storage	142
16.3. Routes	142
17. Admin-Client Services	143
17.1. Allgemeines	143
17.2. ACPUserAuth	143
17.3. ACPOrderValidation	143
17.3.1. Service-Routes	144
17.3.2. Validieren einer Bestellung	144
17.3.3. Abschließen einer Bestellung	145
18. Admin-Client Screens	146
18.1. ACP-User-Authentication	146
18.2. Admin-Client HomeScreen	146
18.2.1. Scannen von QR-Codes	146
VII. Appendix	149
Zeitaufwand	150
Literaturverzeichnis	152
Abbildungsverzeichnis	154
Code-Snippet-Verzeichnis	156

Teil I.

Intro

1. Meta

1.1. Hintergrund

Wir leben in einer Gesellschaft, in der jährlich ca. 1.3 Milliarden Tonnen an Lebensmitteln, die für den Verzehr geeignet sind, aufgrund ihrer Abweichung von Marktnormen, Überproduktion oder aufgrund von falscher Interpretation des Begriffes „Mindesthaltbarkeitsdatum“ weggeworfen werden.

Besonders Restaurants und Hotels mit einem All-You-Can-Eat-Buffet tragen einen signifikanten Teil zu dieser enormen Verschwendungen essbarer Nahrungsmittel bei.

Genau hier kommt **Sokka** ins Spiel und soll für dringend nötige Veränderungen sorgen. Durch die Möglichkeit, Bestellungen von Kunden einfach verwalten zu können, haben Köche, die auf die Dienste von Sokka zurückgreifen einen besseren Überblick darüber, welche Menge an bestimmten Gerichten zubereitet werden müssen.

Auf diese Weise kann nachhaltig verhindert werden, dass zu viel gekocht und letztendlich dann doch entsorgt wird.

1.2. Namensherkunft

Der Name **Sokka** stammt ursprünglich aus der Serie *Avatar: Der Herr der Elemente*. In jener Serie ist Sokka ein Nebencharakter, der ständig über sein konstantes Bedürfnis zu essen spricht und eine Vorliebe dafür hat, exotische Spezialitäten in besonders großen Mengen zu probieren.

Da sich dieses Projekt ebenso um die verschiedenen Speisen und Gerichte eines Restaurants und deren Verwaltung dreht, wurde das System nach dieser Figur benannt.

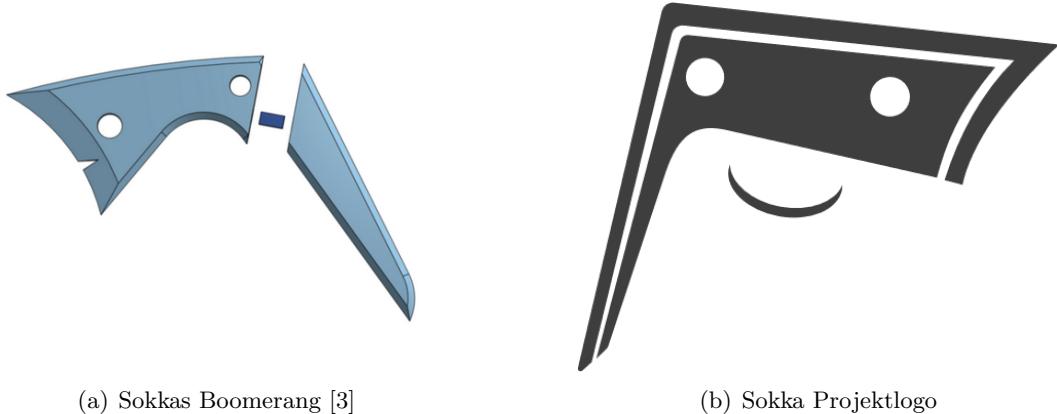


Abbildung 1.1.: Der Charakter **Sokka** aus der Serie *Avatar: Der Herr der Elemente*
[2]

1. Meta

1.3. Projektlogo

Das Markenzeichen des vorher genannten Charakters **Sokka** ist der Boomerang in seiner Hand. Dieser Boomerang wurde dann in Form eines grinsenden Gesichts zum Logo des Projekts.



(a) Sokkas Boomerang [3]

(b) Sokka Projektlogo

Abbildung 1.2.: Sokkas Boomerang und das Projektlogo

2. Aufbau

2.1. Komponenten

Das Projekt besteht aus insgesamt **vier Komponenten**. Nicht alle Komponenten sind für den Endnutzer direkt zugänglich. Einige sind auch dem Personal eines Restaurants, das Sokka verwendet, vorenthalten.

- **Server:** Der Server verwaltet die Daten des Systems und agiert als Kern von Sokka. Alle anderen Komponenten greifen auf den Server zu und stellen Anfragen an diesen.
- **Client:** Der Client ist eine Smartphone-App, welche Kunden des Restaurants installieren können. Durch den Client können verfügbare Produkte eingesehen und bestellt werden.
- **ACP:** Das Admin-Control-Panel ist eine passwortgeschützte Website für das Personal des Restaurants. Über dieses ist es möglich, die Systeminhalte wie Produkte, Menüs und Nutzer zu bearbeiten.
- **Admin-Client:** Der Admin-Client ist eine Smartphone-App, welche vom Personal des Restaurants installiert werden kann. Mit dem Admin-Client können Bestellungen durch Scannen eines QR-Codes der Client-App geprüft und abgeschlossen werden.

Die genaue Funktionsweise der einzelnen Komponenten wird in den folgenden Teilen der Diplomschrift noch genauer beschrieben.

2.2. Testumgebung

Zum Testen der Komponenten wurde ein einfacher virtueller Server vom Hoster *Hetzner Online GmbH* (Frankfurt am Main) verwendet. Dieser fungierte als Host für den *Server* und dem *ACP*. Der *Client* und der *Admin-Client* benötigen keinen Server, da sie als App direkt auf Smartphones laufen.

```
josh@vps:~$ neofetch
,met$$$$$egg.
,g$$$$$$$$$$$$$P.
,g$p"      ""Y$".
,$P'      $$. 
,$$P      ,ggs.    $$b:
`$$`      ,SP"    $$$
$$P      d$'     $$P
$$:      $$.     ,d$$
$$;      Y$b.   ,d$P'
Y$$.      . "Y$$$P"
`$b      "-___
`$$.      Y$$. 
`$$.      `$$b.
`$$.      Y$$b.
`$$.      "Y$b.
`$$.      "``"
```

josh@vps.bemoty.dev
OS: Debian GNU/Linux 10 (buster) x86_64
Host: 1.0
Kernel: 5.4.44-1-pve
Uptime: 34 days, 14 hours, 29 mins
Packages: 547 (dpkg)
Shell: bash 5.0.3
Terminal: /dev/pts/2
CPU: AMD Ryzen 5 3600 6- (2) @ 3.600GHz
Memory: 722MiB / 4096MiB

Abbildung 2.1.: neofetch: Spezifikationen des Test-VServers

2. Aufbau

Um Zugriff auf die verschiedenen Komponenten zu ermöglichen, wurde die Domain `sokka.me` beim US-amerikanischen Domain-Registrar *Namecheap* registriert. Für das Hosten entsprechender Subdomains für die jeweiligen Komponenten sind die Nameserver von *CloudFlare* verwendet worden.

Während des Entwicklungsprozesses kamen folgende Subdomains zur Verwendung:

- **api.sokka.me:** Über diese Domain erfolgte der Zugriff auf den Server und dessen REST-API.
- **acp.sokka.me:** Das Interface für das ACP konnte über diese Domain erreicht werden.
- **pma.sokka.me:** Eine phpMyAdmin-Instanz konnte über diese Domain erreicht werden. Diese war oftmals zum Debugging der Datenbank sehr hilfreich.

Teil II.

Theoretische Grundlagen

3. Verwendete Programmiersprachen

3.1. JavaScript

JavaScript ist eine 1995 erschienene funktionale Skriptsprache mit dynamischem Typensystem, welche ursprünglich für die Erstellung von Websites mit dynamischen Inhalten erstellt wurde. Heutzutage ist *JavaScript* überall im Einsatz und hält den ersten Platz der meistverwendeten Programmiersprachen. [4]

In Sokka findet *JavaScript* nur in Verwendung mit Gulp.js und als kompilierte Version von TypeScript und Dart Anwendung.

3.1.1. Kritik an JavaScript

Obwohl *JavaScript* heutzutage eine der am weitesten verbreiteten Programmiersprachen ist und von vielen anerkannten Firmen aktiv verwendet wird, haben einige Entwickler dennoch Vorbehalte gegen die Nutzung von *JavaScript*. [5]

Die Begründung liegt in der oftmals eigenartigen Verhaltensweise der Sprache, welche man aus anderen Programmiersprachen nicht gewohnt ist. Listen wie *wtfjs* von *Denys Dovhan* zeigen Beispiele, wie komisch *JavaScript* wirklich sein kann. Einige Auszüge davon (siehe denysdovhan/wtfjs auf GitHub für alle Einträge):

```
1 (![] + []) [+[]] +
2  (![] + []) [+!+[]] +
3  ((![] + []) ([])) [+!+[] + [+[]]] +
4  (![] + []) [+![] + !+[]];
```

Code-Snippet 3.1.: „fail“ in *JavaScript*; diese Art von *JavaScript* ist auch bekannt als *JSFuck*

```
1 '3' - 1 // → 2
2 '3' + 1 // → '31'
```

Code-Snippet 3.2.: Lustige Mathematik: Die Typenumwandlung (Coercing) in *JavaScript* ...

3.2. TypeScript

TypeScript ist eine 2012 erschienene Programmiersprache, welche von Microsoft entwickelt wurde und *JavaScript* ein Typensystem hinzufügt. Ansonsten basiert *TypeScript* vollständig auf

3. Verwendete Programmiersprachen

JavaScript und lässt sich deshalb auch zu JavaScript kompilieren.

```
1 class Messenger {  
2     constructor(private message: string) {}  
3  
4     message(): void {  
5         console.log(this.message);  
6     }  
7 }
```

Code-Snippet 3.3.: Eine Klasse mit einer einfachen Funktion in TypeScript

```
1 var Messenger = /** @class */ (function () {  
2     function Messenger(message) {  
3         this.message = message;  
4     }  
5     Messenger.prototype.message = function () {  
6         console.log(this.message);  
7     };  
8     return Messenger;  
9 })();  
10
```

Code-Snippet 3.4.: Eine von TypeScript zu JavaScript kompilierte Klasse

In Sokka wird TypeScript sowohl als primäre Programmiersprache für das Backend, als auch als Programmiersprache für das ACP verwendet.

3.3. Dart

3.3.1. Was ist Dart?

Dart ist eine höhere, clientoptimierte Programmiersprache zur Entwicklung von Apps auf mehreren Plattformen.

Sie wird von *Google* entwickelt und wird hauptsächlich zur Entwicklung von Serversoftware, (nativen) Mobile-, Desktop- oder Web-Apps verwendet.

Wie viele andere Hochsprachen baut Dart strikt auf dem Konzept der Objekt-Orientierung auf und ist eine klassenbasierte Programmiersprache mit eingebautem, automatischen Garbage-Collector.

Dart ist eine kompilierte Sprache und kann sowohl in Maschinencode als auch in Vanilla-JavaScript kompiliert werden, wodurch eine plattformunabhängige Entwicklung ermöglicht wird. [6]

3.3.2. Syntax von Dart

Die Syntax von Dart sieht auf den ersten Blick aus, wie eine Mischung aus *Java* und *JavaScript*.

3. Verwendete Programmiersprachen

Ein *Hello-World*-Programm würde in Dart also folgendermaßen aussehen:

```
1 void main(List<String> arguments) {  
2     print('Hello, World!');  
3 }
```

Code-Snippet 3.5.: Einfaches Hello-World-Programm in Dart

3.3.2.1. Variablen

Die Deklaration von Variablen funktioniert identisch wie in anderen Hochsprachen, so werden Primitiv- und Referenzdatentypen gleich wie in Java deklariert (int, char, byte bzw. Integer, String, Boolean ...).

Alternativ kann auf das Schlüsselwort *var* aus JavaScript zurückgegriffen werden. Dieses ist vergleichbar mit dem *Object*-Datentyp aus Java und „errät“ den benötigten Typ der entsprechenden Variable.

```
1 // Direktes Anlegen eines Integers mit dem Wert 5.  
2 int x = 5;  
3  
4 // Anlegen eines Integers mit dem Wert 7 mithilfe von ''var''.  
5 var y = 7;
```

Code-Snippet 3.6.: Anlegen einfacher Variablen in Dart

Nach Dart-Konventionen ist es allerdings üblich, die Angabe des Datentyps für Variablen im lokalen Scope, beispielsweise innerhalb einer Funktion, durch das *var*-Keyword zu ersetzen. [7]

Für *finale* Variablen mit Initialwert kann sowohl auf dezidierte Datentypen als auch auf *var* verzichtet werden. Bei der Deklaration von Referenzdatentypen kann das *new*-Keyword entfallen.

```
1 final x = 10;
```

Code-Snippet 3.7.: Finale Variable im lokalen Scope

3. Verwendete Programmiersprachen

3.3.2.2. Verzweigungen und Schleifen

Für die Kontrolle des Ablaufs eines Programms werden wie gewohnt *if-else-Statements* und Schleifen wie *for / for in / for each* oder *while* verwendet.

Die Syntax jener Kontrollstrukturen ist mit der von JavaScript quasi ident.

```
1 // Verzweigung mit zwei Bedingungen
2 if (condition 1) {
3     ...
4 } else if (condition 2) {
5     ...
6 } else {
7     ...
8 }
```

Code-Snippet 3.8.: Conditional mit zwei Bedingungen

```
1 final list = [1, 2, 3, 4, 5];
2
3 // Standard for-Schleife
4 for (var i; i < list.length; i++) {
5     ...
6 }
7
8 // for-in-Schleife
9 for (var value in list) {
10    ...
11 }
12
13 // forEach-Schleife mit Callback-Funktion
14 list.forEach((value) => {
15     ...
16 })
```

Code-Snippet 3.9.: Arten von for-Schleifen in Dart

```
1 //while-Schleife
2 while(!condition) {
3     ...
4 }
```

Code-Snippet 3.10.: While-Schleife in Dart

3. Verwendete Programmiersprachen

3.3.2.3. Funktionen

Funktionen in Dart werden auf dieselbe Weise wie in Java deklariert. Die Funktionssignatur bzw. der Funktionskopf benötigt den Rückgabewert der Funktion, den Namen und die Parameterliste.

```
1 // Funktion ohne Rueckgabewert
2 void sayHi() {
3     print('Hi!');
4 }
5
6 // Funktion mit Integer als Rueckgabewert
7 int multiply(final int x, final int y) => x * y;
```

Code-Snippet 3.11.: Deklarieren von Funktionen in Dart

Dart bietet in Bezug auf das Übergeben von Parametern an eine Funktion mehrere Möglichkeiten:

- *positioned, optional parameters*
- *named, optional parameters*

Sogenannte *positionierte und optionale Parameter* werden in der Argumentliste einer Funktion mithilfe eckiger Klammern gekennzeichnet und bewirken, wie der Name bereits verrät, dass jene Parameter optional und in ihrer Position fixiert sind.

```
1 void sendRequest(final String url, final String message,
2                  [ final int port = 80 ] ) {
3     ...
4 }
```

Code-Snippet 3.12.: Funktion mit positioned, optional Parametern

Die Funktion *getHttpUrl()* benötigt neben dem Server und Pfad auch eine Portnummer, welche hier als positionierter, optionaler Parameter mit einem Standartwert von 80 festgelegt wird.

3. Verwendete Programmiersprachen

Wird beim Aufruf dieser Funktion der Port-Integer weggelassen, so wird dieser immer auf 80 gesetzt.

```
1 // Aufruf mit Port 8080
2 getHttpUrl('sokka.me', 'Sokka is awesome!', 8080);
3
4 // Aufruf mit Standartport 80
5 getHttpUrl('sokka.me', 'Sokka is awesome!');
```

Code-Snippet 3.13.: Aufrufen einer Funktion mit positioned Parametern

Neben positionierten Parametern gibt es noch „benannte“, sogenannte *named* Parameter, die mit geschwungenen Klammern in der Parameterliste deklariert werden.

```
1 sendRequest(final String url, { final String message,
2           final int port }){
3   ...
4 }
```

Code-Snippet 3.14.: Aufrufen einer Funktion mit named Parametern

Neben einer URL benötigt die obige Funktion auch eine Nachricht und eine Port-Nummer, um einen gültigen Request zu versenden. Aufgrund ihrer Deklaration innerhalb der geschwungenen Klammern werden sie als benannte Parameter interpretiert und werden wie folgt übergeben.

```
1 sendRequest('api.sokka.me', message: 'Sokka is awesome!',
2             port: 80);
```

Code-Snippet 3.15.: Aufrufen einer Funktion mit *named* Parametern

```
1 // Aufruf mit Port 8080
2 getHttpUrl('sokka.me', 'Sokka is awesome!', 8080);
3
4 // Aufruf mit Standartport 80
5 getHttpUrl('sokka.me', 'Sokka is awesome!');
```

Code-Snippet 3.16.: Aufrufen einer Funktion mit *positioned* Parametern

3. Verwendete Programmiersprachen

3.3.2.4. Klassen

Ähnlich zu Hochsprachen wie Java oder C# können auch in Dart einfach Klassen erstellt werden, wie nachfolgend zu sehen ist.

```
1 class Point {  
2     int _x;  
3     int _y;  
4  
5     Point(this._x, this._y);  
6 }
```

Code-Snippet 3.17.: Simple Klassen in Dart

Wie auch in JavaScript oder Python gibt es in Dart keine Access-Modifier, um den Zugriff auf Felder, Methoden oder Klassen zu regulieren. Felder und Funktionen sind standardmäßig **public** und können lediglich durch einen Unterstrich zu Beginn des Feld- bzw. Funktionsnamen als **private** gekennzeichnet werden.

Desweiteren stehen in Dart, ebenso wie in C#, die Schlüsselwörter **get** und **set** zur Verfügung, mit deren Verwendung Funktionen als dezidierte Getter- bzw. Setter-Funktion für eine Membervariable definiert werden können.

```
1 class Foo {  
2     String _bar;  
3  
4     String get getBar => this._bar;  
5  
6     set setBar(final String bar) => this._bar = bar;  
7 }
```

Code-Snippet 3.18.: Getter- und Setter-Funktionen in Dart

In diesem Beispiel werden *getBar* und *setBar* als dezidierte Getter- und Setter-Funktion für das Feld *_bar* festgelegt. Auffällig ist ebenso die JavaScript-ähnliche Arrow-Syntax, die ein Return-Statement zur Gänze ersetzen kann.

3. Verwendete Programmiersprachen

So kann ein *return*, wie es in Java geschrieben wird, durch ein Arrow-Statement wie folgt ersetzt werden.

```
1 // Java
2 public String getName() {
3     return this.name;
4 }
5
6 // Dart
7 String get getName => this._name;
```

Code-Snippet 3.19.: Vergleich einer Getter-Funktion zwischen Java und Dart

3.3.2.5. Vererbungsstrukturen

Abstrakte Klassen Um eine Klasse als abstrakt zu definieren, wird in Dart das *abstract*-Keyword verwendet und mittels *extends*-Keyword an andere Klassen vererbt.

```
1 abstract class Foo {
2     ...
3 }
4
5 class Bar extends Foo {
6     ...
7 }
```

Code-Snippet 3.20.: Erzeugen und Vererben abstrakter Klassen in Dart

Interfaces Dart unterstützt das Prinzip von sogenannten *impliziten Interfaces*. Das heißt, dass jede erzeugte Klasse automatisch als ein Interface genutzt und implementiert werden kann.

4. Haupttechnologien

4.1. Docker

Docker ist eine erstmals 2013 veröffentlichte Technologie, die vom US-amerikanischen Unternehmer *Solomon Hykes* geschaffen wurde. [8] Sie ermöglicht eine einfache Bereitstellung von Software durch das Virtualisieren von isolierten Containern, welche einer Software eine funktionsfähige Laufzeitumgebung und alle benötigten Abhängigkeiten bietet.

Docker wird genutzt, um das Sokka-System schnell und einfach auf neuen Systemen lauffähig zu machen. Dafür existiert eine `docker-compose.yml`-Datei im Root-Verzeichnis des Sokka-Repositories, welche Docker so konfiguriert, dass alle benötigten Container (in der Datei genannt: `services`) vollautomatisch erstellt werden.

4.1.1. Images

Wenn eine Anwendung durch Docker lauffähig gemacht werden soll, muss ein **Docker Image** generiert werden. Dafür wird ein **Dockerfile** erstellt, das Docker Anweisungen zum Generieren des Images gibt. Im Dockerfile wird zum Beispiel festgelegt, welche *Laufzeitumgebung* und welche Version bzw. Implementation dieser Umgebung benötigt wird (beispielsweise eine JVM-Implementation für Java-Anwendungen oder Node.js für JavaScript-Anwendungen). Außerdem werden dem Dockerfile Befehle übergeben, die für die Anwendung benötigte Anwendungen nachlädt und ins Image lädt.

```
1 FROM node:10
2 WORKDIR /usr/src/server
3
4 COPY package*.json .
5
6 RUN npm install
7 COPY .
8 EXPOSE 3000
9
10 ENTRYPOINT [ "node", "main.js" ]
```

Code-Snippet 4.1.: Beispielhaftes Dockerfile für eine Node.js Web-App

4. Haupttechnologien

4.1.2. Container

Ein Docker-Container führt immer ein zuvor generiertes Docker-Image aus. Dieses Image kann entweder selbstgeneriert sein oder von einer **Docker-Registry** wie *Docker Hub* kommen, welches Software verschiedenster Herausgeber als fix fertiges Image zur Verfügung stellt.

4.1.3. Volume

Im Allgemeinen ist es Containern nicht möglich, auf das Dateisystem des Hostsystems zuzugreifen. Manchmal ist dies allerdings notwendig, beispielsweise wenn zur Anwendung hochgeladene Bilder gespeichert werden müssen. Damit diese Daten bei einem Neustart des Containers nicht verloren gehen, können **Volumes** konfiguriert werden, welche es einem Container erlauben, in einem fix festgelegten Ordner auf dem Hostsystem zu lesen und zu schreiben.

4. Haupttechnologien

4.2. Node.js

Node.js ist eine Laufzeitumgebung für die Programmiersprache *JavaScript*, die es ermöglicht, JavaScript-Code außerhalb des Webbrowsers auszuführen. Die Laufzeitumgebung basiert auf der V8-JavaScript-Engine von Google, die ursprünglich für die Verwendung in Kombination mit ihrem Chrome-Browser entwickelt wurde. [9]

Sokka nutzt Node.js, um den aus *TypeScript* kompilierten JavaScript-Code des Servers auszuführen.

4. Haupttechnologien

4.3. React

React (oder *React.js*) ist eine Front-End JavaScript-Library zum Erstellen von Nutzerinterfaces, die von Facebook entwickelt wird. Neben *Vue.js* und *Angular* ist React eine der bekanntesten und am weitest verbreiteten Front-End-Lösungen für moderne Web-Anwendungen. [10]

Sokka nutzt React als Lösung für das ACP.

4.3.1. JSX

Ein Vorteil von React gegenüber anderen bekannten Front-End-Libraries ist **JSX**, in der Langform *JavaScript-XML*. JSX erlaubt es, HTML in JavaScript-Code zu verwenden und vice versa. Beispielsweise kann eine Information von einer API geladen und anschließend mit Hilfe von JSX in HTML angezeigt werden.

```
1 const data = await fetchData();
2 const html = <p>Fetched data from API: {data}</p>;
3
4 ReactDOM.render(
5   html,
6   document.getElementById('root')
7 );
```

Code-Snippet 4.2.: JSX-Beispiel für das Laden und Rendern von Informationen

4.3.2. Components

In React ist alles ein **Component**. Jede Reihe, jede Spalte, jeder Kasten, jedes Eingabefeld und jeder Knopf sind ein eigenes Component.

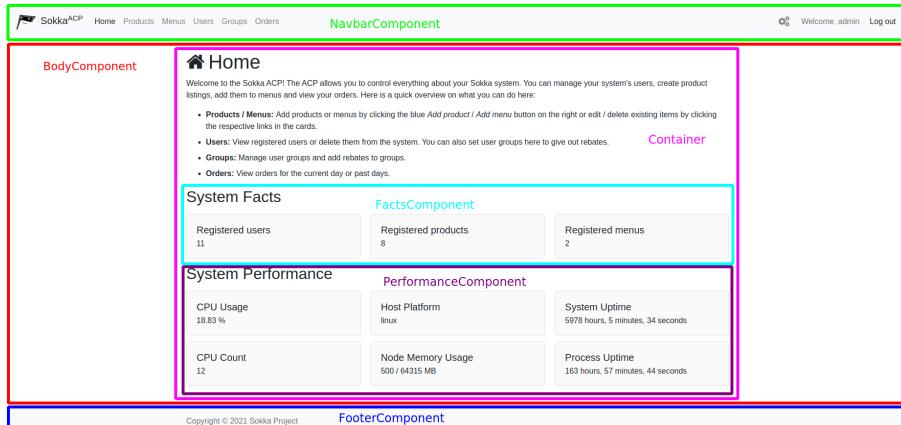


Abbildung 4.1.: JSX-Components am Beispiel des Sokka-ACPs

4. Haupttechnologien

Components können ineinander und so oft wie erforderlich verwendet werden, wodurch eine Menge an redundanten Code eingespart wird. Wenn man das Verhalten eines Components dennoch der Situation entsprechend ändern möchte, können **Component Properties** übergeben werden. Diese Properties oder auch `props` versorgen eine neu erstellte Komponente mit initialen Informationen, die für das Rendern der Component notwendig sind.

```
1 class TooltipButton extends React.Component {
2   render() {
3     return (
4       <button title={this.props.title}>
5         Submit
6       </button>
7     );
8   }
9 }
10
11 ReactDOM.render(
12   <TooltipButton title="Wow, a tooltip!" />,
13   document.getElementById('root')
14 );
```

Code-Snippet 4.3.: Beispiel für ein React-Component für einen Button mit änderbarem Tooltip

4.3.3. States

Oftmals reicht es nicht aus, eine Component nur mit initialen Informationen zu versorgen. Manche Components halten bestimmte Daten, die sich verändern können, weil sie nachgeladen oder durch den Nutzer geändert werden. Für solche Daten gibt es in Components die **Component States**.

```
1 class UnixTime extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {millis: Date.now()};
5   }
6
7   componentDidMount() {
8     setInterval(() => {
9       this.setState({
10         millis: Date.now()
11       });
12     }, 2);
13   }
14
15   render() {
16     return (<p>{this.state.millis}</p>);
17   }
18 }
```

Code-Snippet 4.4.: Beispiel für ein React-Component mit State, welches die UNIX-Zeit rendert

4.4. Flutter

Flutter ist ein von **Google** entwickeltes *User-Interface-Framework* zur Entwicklung plattformabhängiger Apps sowohl für Web, Desktop als auch Mobile.[11]

Neben der Möglichkeit Apps für verschiedene Systeme erstellen zu können, ohne etwas mehrfach programmieren zu müssen, bietet Flutter auch einen integrierten Widget-Katalog voll vorgefertigter UI-Elemente.

Diese können nach Belieben verwendet, umgestaltet und miteinander kombiniert werden.

Flutter bietet hiermit auch Personen mit weniger Erfahrung auf dem Gebiet der Softwareentwicklung die Möglichkeit, relativ einfach eigene Apps zu entwickeln.

4.4.1. Was macht Flutter besonders?

4.4.1.1. Design-Architektur

Der Clou hinter Flutter steckt in seiner Architektur und Design-Struktur, denn in diesem Framework ist **alles** ein Widget, von der eigentlichen App bis hin zum einfachen Text.

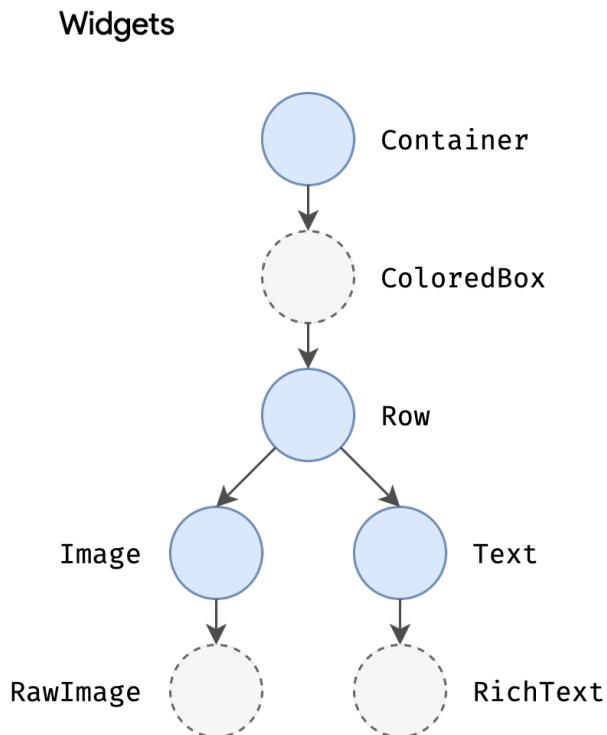


Abbildung 4.2.: Design-Struktur und Widget-Aufbau von Flutter
[12]

Der Aufbau und das Erstellen eines User-Interfaces mithilfe von Flutter erfolgt in erster Linie durch das Kombinieren, Zusammenbauen und Verschachteln von Widgets. Abhängig von ihrem

4. Haupttechnologien

Zustand (*Widget-State*) und ihren Attributen beschreiben jene Widgets, wie das entsprechende View zur Laufzeit aussehen wird.

Ändert sich der Zustand eines Widgets, wird es erneut aufgebaut und gerendert. Die Flutter-Engine vergleicht dabei vorherige Zustände mit der neuen Änderung und rendert nur jene Widgets neu, die sich auch wirklich verändert haben.

Auf diesem Weg kann der Renderprozess von Widgets so schnell wie möglich durchlaufen werden, wodurch auch die Performance der App gesteigert wird.

Auf diesem Prinzip baut auch Flutters *Hot-Rewind*-Funktion bei der Entwicklung auf. Mithilfe dieser können Veränderungen an der App in wenigen Augenblicken auf den Emulator übertragen werden, ohne dass es einen Neustart der App bedingt.

So können selbst die kleinsten Veränderungen schnell ausprobiert und getestet werden, ohne einen erneuten Build von Grund auf notwendig zu machen.

4.4.1.2. Plattformunabhängigkeit

Mithilfe von Flutter können Apps unabhängig von einer Zielplattform entwickelt werden. Das heißt, dass eine Codebase sowohl als Android-App, iOS-App und als Web-App deployed werden kann.

Neben den enormen Zeitersparnissen werden ebenso die nötigen Kenntnisse zweier Programmiersprachen, beispielsweise *Kotlin* für Android und *Swift* für iOS, auf **Dart** reduziert.

Unter anderem wird dies durch den oben erwähnten Widget-Katalog erzielt. Dieser bietet zwei Varianten von Widgets an, einerseits die *Material Components* für Android, andererseits die *Cupertino Components* für iOS.

Das jeweilige Widget-Paket passt sich adaptiv an das Design des entsprechenden Betriebssystems an, um einen möglichst geringen UI-Kontrast zwischen der App und dem Rest des Betriebssystems zu garantieren.



Abbildung 4.3.: Slider-Widget im Material-Design für Android
[13]

4. Haupttechnologien



Abbildung 4.4.: Slider-Widget im Cupertino-Design für iOS
[14]

4.4.2. Layoutting in Flutter

Den Kern von Flutters Layout-System bilden die einzelnen Widgets, deren Attribute und deren kaskadierendes Verhalten zueinander.

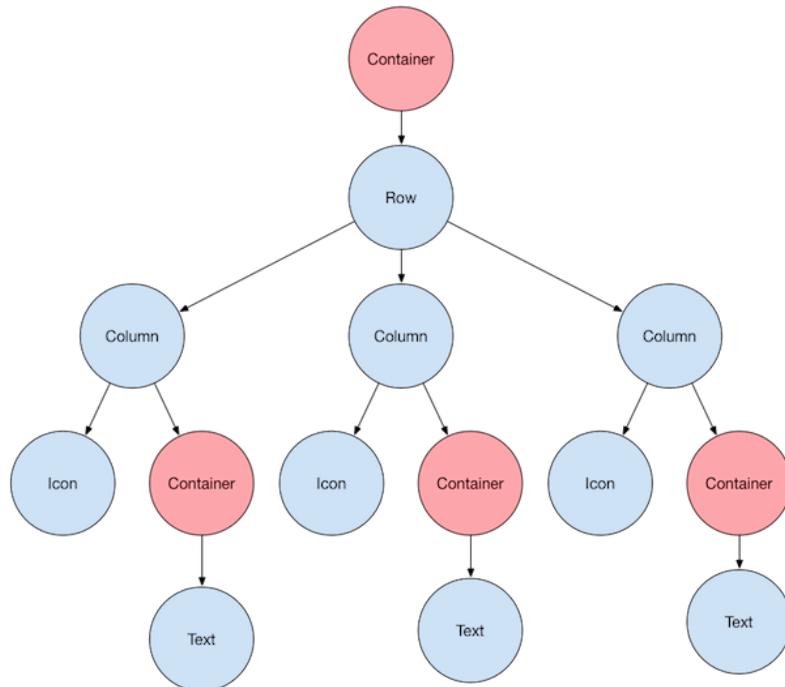


Abbildung 4.5.: Widget-Tree für ein beispielhaftes Layout
[15]

Ähnlich zu anderen Frameworks wie beispielsweise *Bootstrap 4* können unter Einsatz von bekannten Standard-Widgets wie Containern, Rows und Columns bereits simple Layouts erstellt werden.

4. Haupttechnologien

Ein Widget wird in Flutter mithilfe des entsprechenden Objekt-Konstruktors an der gewünschten Stelle erzeugt und mit entsprechenden Attributen befüllt.

```
1 ...
2 new Container(
3   height: 280.0,
4   width: 400.0,
5   padding: new EdgeInsets.only(top: 10.0),
6   child: new Text('Sokka is awesome!')
7 ),
8 ...
```

Code-Snippet 4.5.: Erzeugen eines einfachen Container-Widgets mithilfe von Flutter

Hier wird ein neues Container-Widget mit 280 Pixeln Höhe, 400 Pixeln Breite, einem Top-Padding von zehn Pixeln und einem Text als Kind-Widget erzeugt.

Durch das Kaskadieren unterschiedlicher Widgets können relativ einfach komplexe UI-Strukturen erzeugt werden. Hierfür müssen lediglich die weiteren / inneren Widgets dem `child-` bzw. `children-Property` des äußeren Widgets übergeben werden.

```
1 ...
2 new Container(
3   child: new Row(
4     children: <Widget>[
5       new Icon(
6         Icons.ac_unit,
7         color: Colors.white,
8       ),
9       new Text(
10         'Sokka is awesome!',
11         color: Colors.white,
12       ),
13       new Container(
14         child: new Image( ... )
15       ),
16     ],
17   ),
18 ),
19 ...
```

Code-Snippet 4.6.: Erzeugen eines Containers mit verschachtelten Child-Widgets

4.4.3. Rendern von Widgets

Grundsätzlich erfolgt das Rendern eines Widgets über seine eigene `build()`-Funktion. Diese wird aufgerufen, sobald ein Widget gerendert bzw. neuerendert werden muss und retourniert ein

4. Haupttechnologien

entsprechendes Widget-Objekt, das dem Design-Tree des entsprechenden Kontexts hinzugefügt wird.

```
1 @override
2 Widget build(BuildContext context) {
3     return new Container(
4         child: new Image(
5             image: new AssetImage('path/to/image')
6         ),
7     );
8 }
```

Code-Snippet 4.7.: `build()`-Funktion eines Widgets

Obige Build-Funktion erzeugt einen neuen Container mit einem Bild als Child-Element und übergibt dieses an den Widget-Tree des entsprechenden Layouts.

Hierbei gilt, dass allerdings zwischen zwei Arten von Widgets zu unterscheiden ist: den sogenannten *Stateless Widgets* und *Stateful Widgets*.

Stateless-Widgets haben einen definierten, statischen Zustand und reagieren nicht auf das Verhalten anderer Widgets oder die Interaktion eines Nutzers, beispielsweise Icons, Text oder einfache Buttons.

```
1 class SokkaLogo extends StatelessWidget {
2     @override
3     Widget build(BuildContext build) {
4         return new ListTile(
5             title: new Text('Sokka is awesome!'),
6             leading: new Icon(
7                 Icons.flutter_logo,
8                 color: colors.White,
9             ),
10        );
11    }
12 }
```

Code-Snippet 4.8.: Einfaches Stateless Widget

Stateful-Widgets dagegen haben einen internen Zustand, der durch Interaktion mit dem Nutzer dynamisch verändert werden kann, beispielsweise durch das Tippverhalten des Nutzers.

4.4.4. Interaktivität der App

Um für Interaktion der App zu sorgen werden, wie eben erwähnt, sogenannte Stateful-Widgets benötigt. Diese Widgets sind dynamisch und können beispielsweise ihr Aussehen als Reaktion auf Events, Input durch den Nutzer oder den Erhalt von Daten von einer API verändern.

4. Haupttechnologien

Damit ein solches Widget weiß welchen Zustand es zum aktuellen Zeitpunkt hat, wird dieser in einem *State*-Objekt vom Typ des Widgets gespeichert. Ein solches State-Objekt besteht aus Werten, die den Zustand des entsprechenden Widget beschreiben können. Beispielsweise ob ein ExpansionPanel (aufklappbare Karte) auf- oder zugeklappt ist oder der welchen Wert ein Slider-Widget aktuell hat.

Um den Zustand eines Widgets während der Laufzeit zu verändern und jene Änderung im Anschluss anzeigen zu können, wird hierfür die `setState()`-Funktion verwendet.

Diese überschreibt die entsprechende Variable mit dem neuen Zustandswert und lässt die Flutter-Engine das entsprechende Widget neu rendern, um die Veränderung direkt erkennbar zu machen.

```
1 class LoginScreen extends StatefulWidget {
2     @override
3     _LoginScreenState createState() => _LoginScreenState();
4 }
5
6 class _LoginScreenState extends State<LoginScreen> {
7     @override
8     Widget build(BuildContext context) {
9         return new Scaffold(
10             body: new Center(
11                 ...
12             ),
13         );
14     }
15 }
```

Code-Snippet 4.9.: Ein einfaches Stateful Widget

Wie anhand der Implementation zu erkennen ist, ruft das Widget *LoginScreen* die Build-Funktion seines Zustandwrappers *_LoginScreenState* auf und erzeugt damit ein neues Widget mit dem entsprechenden Zustand.

Ändert sich jener Zustand, so wird die Build-Funktion erneut mit einem anderen Zustand aufgerufen und das Widget wird neu gerendert.

4.4.4.1. App-States und ephemeral States

In Bezug auf die Widget-Architektur von Flutter gibt es zwei Arten von Zuständen:

Ephemeral States Unter sogenannten ephemeralen Zuständen, auch UI-Zustände oder lokale Zustände genannt, versteht man Zustände, welche in einem einzelnen Widget gespeichert werden können. Ein ephemeraler Zustand ist also vergleichbar mit einer veränderbaren, lokalen Variable einer Klasse oder eines Scopes.

4. Haupttechnologien

Einfache Beispiele hierfür sind etwa

- der Fortschritt eines Animationsprozesses
- aktuell ausgewählter Tab in einer BottomNavigationBar
- die aktuelle Zahl an Widgets in einem ListView

App-States Im Gegensatz zu ephemeralen Zuständen werden App-Zustände global über die gesamte App gespeichert und können daher an jedem Ort in der App abgerufen und verändert werden.

Beispiele im konkreten Fall von Sokka sind unter anderem

- E-Mail-Adresse und Session-Token der aktuellen Usersitzung
- Aktuell verfügbare Menüs und Produkte
- Gespeicherte Menüs und Produkte im Warenkorb
- alle Bestellungen des angemeldeten Nutzers

4.4.5. Navigation und Routing durch die App

Um zwischen den verschiedenen Screens und Views der App zu wechseln und zu navigieren, benötigen wir die *Navigator*-Klasse von Flutter.

Hierfür ist es sinnvoll, zuerst eine *HashMap* mit allen möglichen Routes der App zu definieren, um so benannte Routen (*named routes*) verwenden zu können.

```
1 final Map<String, WidgetBuilder> routes = {
2     '/':      (BuildContext buildContext) => new HomeScreen(),
3     '/login': (BuildContext buildContext) => new LoginScreen(),
4     '/signup': (BuildContext buildContext) => new SignUpScreen(),
5     '/loading': (BuildContext buildContext) => new LoadingScreen()
6 };
```

Code-Snippet 4.10.: Routes für die Navigation zwischen Screens

4. Haupttechnologien

Diese Routes müssen im Anschluss dem `routes`-Property des `MaterialApp`-Widgets unserer App übergeben werden.

```
1 return new MaterialApp(  
2     ...  
3     routes: routes  
4     ...  
5 );
```

Code-Snippet 4.11.: Setzen der Routes im App Entry-Point

Nun kann mithilfe der `Navigator`-Klasse und unseren definierten Routes einfach durch die App navigiert werden.

Hierfür stehen einige Methoden zur Verfügung:

```
1 // Wechselt vom aktuellen Screen zum LoginScreen  
2 Navigator.of(context).pushNamed('/login');  
3  
4 // Wechselt vom aktuellen Screen zum SignupScreen und schliesst  
// vorherigen.  
5 Navigator.of(context).popAndPushNamed('/signup');
```

Code-Snippet 4.12.: `Navigator.of()`-Funktion zum Wechseln der Screens per `namedRoutes`

Alternativ kann auch ohne Named-Routes durch die App mithilfe der (umständlicheren) `push()`-Funktion navigiert werden.

```
1 // Erzeugt eine neue, temporaere Route zum HomeScreen und wechselt  
// direkt zu diesem  
2 Navigator.of(context).push(context, new MaterialPageRoute(builder:  
    (context) => new HomeScreen()));  
3  
4 // Schliesst HomeScreen und kehrt zu vorherigen Screen zurueck  
5 Navigator.pop(context);
```

Code-Snippet 4.13.: Navigieren durch die App per `MaterialPageRoute`

5. Dependencies und Libraries

5.1. Gulp.js

Gulp.js ist das GNU/Make für Node.js-Anwendungen. Als Task Runner führt Gulp repetitive Aufgaben im Web-Development wie Linting, Unit-Testing, Code-Optimierung und -Kompilierung aus.

5.1.1. gulpfile.js

Das `gulpfile` ist eine Datei in einem Projekt, welches die einzelnen Aufgaben, deren Auslöser und Ausführung definiert. Aufgaben werden in Gulp mit `gulp.task(name, runner)` definiert. So kann beispielsweise eine Build-Funktion für ein TypeScript-Projekt definiert werden:

```
1 gulp.task('build', gulp.series(() => {
2   return tsProject.src().pipe(tsProject()).js.pipe(gulp.dest(tsConfig.compilerOptions.outDir));
3 }));
```

Code-Snippet 5.1.: Beispielhaftes `gulpfile.js` zum Kompilieren von TypeScript

5.1.2. Rolle von Gulp in Sokka

Das *Sokka-Backend* nutzt Gulp.js für:

- das **Kompilieren** von TypeScript-Code zu JavaScript-Code
- das **Überwachen** von Änderungen im TypeScript-Code (eine Änderung führt automatisch den `build`-Task aus via `gulp-nodemon`-Plugin)
- das **Ausführen** bzw. **Debugging** vom Server

5.2. Express.js

Express ist die wohl bekannteste Library zum Erstellen von Server-Software mit *Node.js*. [16] Dabei hat sie das Ziel durch ihre minimalistische und flexible Art eine schnelle und robuste Grundlage für Serverprojekte jeglicher Art in Node.js zu bieten. Mit Express ist es möglich, einfache APIs zu schreiben, aber auch komplexe Web-Projekte umzusetzen.

5. Dependencies und Libraries

5.2.1. Das Express „Hello world!“

Die wohl einfachste Express-Anwendung gibt bei Aufruf der Root-Route einfach ein „Hello world!“ zurück.

```
1 var app = require('express')();
2
3 app.get('/', (req, res) => {
4   res.send("Hello world!");
5 });
6
7 app.listen(8080);
```

Code-Snippet 5.2.: „Hello world!“ mit Express.js

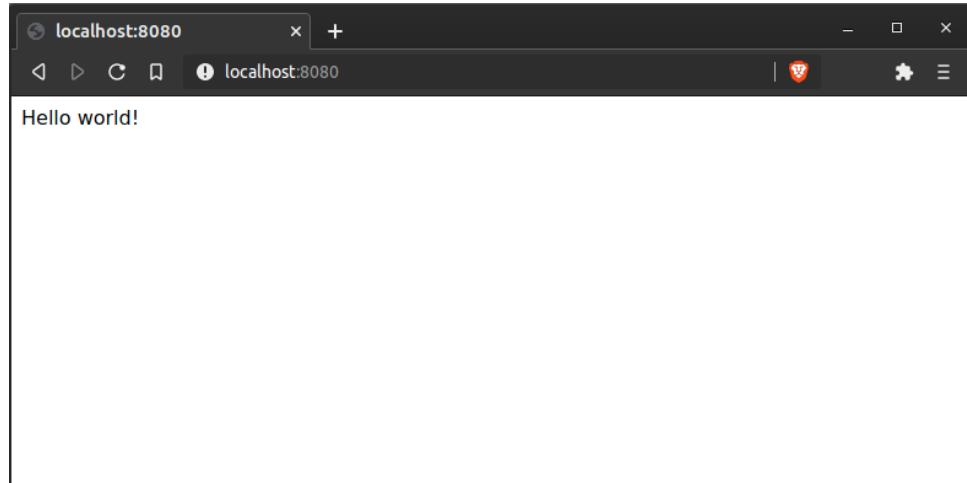


Abbildung 5.1.: Die Ausgabe einer „Hello world!“-Seite mit Express.js

Natürlich ist die Anwendung von Express in Sokka wesentlich komplexer, allerdings sieht man in diesem Beispiel schon sehr gut die grundlegende Funktionsweise der Library .

5.2.2. Rolle von Express in Sokka

Das Backend von Sokka stützt sich komplett auf die Funktion von Express. Jede API-Route des Servers wird durch Express erstellt und jede Anfrage an den Server wird von Express unterhalten.

Teil III.

Backend

6. Allgemeines

Der **Backend-Server** hat die Aufgabe, durch fest definierte Zugangspunkte den Abruf oder die Veränderung von Daten in der Datenbank des Systems zu ermöglichen. Damit dies funktioniert, besteht der Server aus zwei ganz grundlegenden Teilen:

- **Datenbankverbindung**
- **REST-API**

Für die Implementation des Servers wurde die von Microsoft entwickelte Scriptsprache *TypeScript* gewählt. Der Code davon wird anschließend mit *Gulp.js* zu *JavaScript* kompiliert.

6.1. Datenbankverbindung

Zum Start der Anwendung versucht der Server mit Hilfe der Library `mysql` eine Verbindung zum MySQL-Server aufzubauen, welcher in einem anderen Docker-Container läuft. MySQL nutzt als Hostname den Containernamen, weshalb der Server via `mysql:3306` eine Verbindung zu MySQL herstellen kann. Die Anmelddaten für MySQL werden sicher in *Docker Secrets* gespeichert.

6.2. Konfiguration

Der Backend-Server erlaubt einige Konfigurationsmöglichkeiten. Werte für Konfigurationen werden zuerst aus den Umgebungsvariablen geladen. Falls dort kein Wert gefunden werden kann, wird der Wert in *Docker Secrets* gesucht.

6.2.1. Verfügbare Konfigurationen

`DEBUG: boolean`

→ aktiviert den Debug-Modus. Dieser muss für lokale Tests immer `true` sein! Ist dieser Wert `true`, so werden beispielsweise Bilder in einem lokalen `images`-Ordner gespeichert und nicht in der Docker Volume.

`DEBUG_LOG: boolean`

→ aktiviert den Debug-Log. Ist dieser Wert auf `true` werden beispielsweise alle eingehende Anfragen an den Server im Log mitgeschrieben.

`DOMAIN: string`

→ ist die Domain, unter der der Backend-Server erreichbar ist. Sie wird genutzt, um in Verifikations-E-Mails die richtige Domain zu senden.

6. Allgemeines

MAX_USER_SESSIONS: number

→ gibt an, wie oft sich ein Nutzer (z. B. bei einem neuen Gerät) anmelden kann, bevor eine Session ungültig gemacht wird.

MAX_ACP_USER_SESSIONS: number

→ gibt an, wie oft sich ein ACP-Nutzer (z. B. bei einem neuen Gerät) anmelden kann, bevor eine Session ungültig gemacht wird.

SALT_ROUNDS: number

→ gibt an, wie oft bcrypt ein Passwort oder einen Session-Token hasht, bevor er in die Datenbank gespeichert wird. Je größer dieser Wert ist, desto schwieriger ist es, den Hash zu erraten".

10 ist hierfür meist ein sicherer Wert. [17]

6.2.2. Docker Secrets

Docker Secrets erlaubt es, sensitive Daten wie Passwörter oder Zugangsschlüssel sicher auf einem Server zu speichern. Im Fall von Sokka werden MYSQL_DB, MYSQL_HOST, MYSQL_USERNAME, MYSQL_PASSWORD, VERIFY_EMAIL und VERIFY_EMAIL_PASSWORD gespeichert.

Die Secrets müssen in Textdateien in einem vom VCS ausgenommenen Ordner erstellt werden. Diese Textdateien müssen anschließend bei der Erstellung der Container, in unserem Fall in der docker-compose.yml, eingetragen werden. Beim Start der Container werden alle Secrets in die Container an den Pfad /run/secrets/<key> kopiert, wo sie schlussendlich ausgelesen werden können.

```
1 secrets:
2   MYSQL_DB:
3     file: ./secrets/MYSQL_DB.txt
4   MYSQL_HOST:
5     file: ./secrets/MYSQL_HOST.txt
6   MYSQL_USERNAME:
7     file: ./secrets/MYSQL_USERNAME.txt
8   MYSQL_PASSWORD:
9     file: ./secrets/MYSQL_PASSWORD.txt
10  VERIFY_EMAIL:
11    file: ./secrets/VERIFY_EMAIL.txt
12  VERIFY_EMAIL_PASSWORD:
13    file: ./secrets/VERIFY_EMAIL_PASSWORD.txt
```

Code-Snippet 6.1.: Sokkas Docker Secrets in der docker-compose.yml

6.3. REST-API

Damit die *Client App* und das *ACP* auch auf die Daten der Datenbank zugreifen können stellt der Server dokumentierte Zugangspunkte oder auch **REST-Routes** zur Verfügung. Computer im Internet können auf diese Routes zugreifen, indem sie vorgefertige HTTP-Anfragen an den

6. Allgemeines

Server senden. Wie genau diese Anfragen aussehen müssen, wird im Kapitel *REST-Route-Dokumentation* genauer erläutert.

6.3.1. Bilder

Sowohl Produkte als auch Menüs erhalten in Sokka anpassbare Bilder. Um dies zu realisieren, existiert `/acp/image` zum Hochladen von Bildern und `/image` zum Abrufen von Bildern.

Wenn ein Bild über die ACP-Route hochgeladen wird, legt der Server das Bild mit einer zufallsgenerierten ID (16 Bytes, Hex enkodiert) in einer Docker Volume ab. Diese Bilder können dann später wieder über die zweite Route abgerufen werden.

6.3.2. Verifikation

Bei der Registrierung müssen Nutzer ihre E-Mail-Adresse angeben. Um sicherzugehen, dass diese Adresse auch wirklich gültig ist, wurde mit Hilfe der Library *nodemailer* ein simples E-Mail-Verifikationssystem implementiert. Ein Nutzer kann erst Bestellungen tätigen, wenn sein Nutzerkonto verifiziert ist.

Schon beim Start des Servers wird geprüft, ob die in den *Docker Secrets* gespeicherten Informationen zur Anmeldung bei Google-Mail korrekt sind (`VERIFY_EMAIL` und `VERIFY_EMAIL_PASSWORD`). `VERIFY_EMAIL_PASSWORD` ist hier allerdings nicht wirklich das Passwort für das Google-Mail-Konto, sondern ein generiertes App-Passwort (myaccount.google.com/u/1/apppasswords).

Sofern die gespeicherten Anmelddaten korrekt sind (und der Anmeldevorgang bei Google erfolgreich war), kann der Server vollautomatisch E-Mails an neue Nutzer versenden.

Wenn sich ein neuer Nutzer durch die REST-Route `/user/create` mit einer laut RFC 5322 gültigen E-Mail-Adresse registriert, werden 64 zufällige Bytes generiert, welche anschließend in Base64 enkodiert werden und der entstandene String in Verbindung mit der ID des Nutzerkontos in einer Tabelle abgelegt wird.

Im selben Moment wird an die angegebene E-Mail-Adresse eine Nachricht mit einer URL gesendet, welche den generierten String enthält.

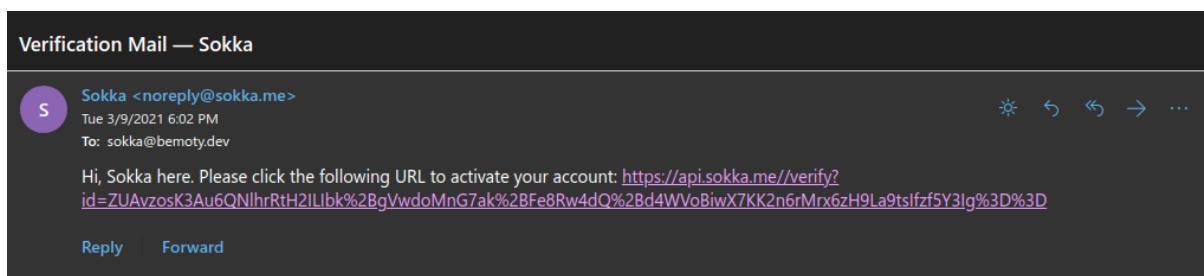


Abbildung 6.1.: Eine E-Mail, welche durch Sokka bei der Verifizierung gesendet wurde

Die URL in der E-Mail führt zur REST-Route `/verify`, welche, wenn der String in der URL gültig ist, das damit verbundene Nutzerkonto verifiziert.

7. REST-Route-Dokumentation

Im folgenden Kapitel werden alle verfügbaren Routes der im Backend-Server eingebauten REST-API dokumentiert. Die Routes der API sind nicht für die öffentliche Verwendung bzw. für die Erstellung nutzerdefinierter Clients gedacht und wurden ausschließlich für die Verwendung in Kombination mit dem **Sokka-Flutter-Client** konzipiert und umgesetzt.

7.1. Authorization

Um die Zugriffe auf gespeicherte Daten und Nutzerkonten zu sichern, sind einige der REST-Routes durch eine *Bearer-Authorization* (auch bekannt als *Token-Authorization*) geschützt. Diese Routes benötigen in dem Header der Anfrage einen Eintrag mit dem Schlüsselwort `Authorization`. Fehlt dieser Eintrag oder ist er ungültig, so wird die Anfrage an die API nicht bearbeitet.

Die Funktionsweise der *Bearer-Authorization* steckt in ihrem Namen. Sie gibt dem „Bearer“, zu Deutsch dem „Inhaber“ eines Zugriff-Tokens, Zugriff auf eine Ressource. [18]

Im Sokka-System gibt es zwei Arten von Tokens, die für die Authorisierung benötigt werden können: `App` und `ACP`. Welche der beiden Arten für eine Route benötigt wird, ist bei den einzelnen Routes als **Authorization-Typ** angegeben.

- Einen `App`-Token erhält man durch eine Anmeldung in der App (siehe `/user/login`)
- Einen `ACP`-Token erhält man durch eine Anmeldung im ACP (siehe `/acp/login`)

Damit die Authorisierung klappt, muss der empfangene Token gemeinsam mit der Nutzer-E-Mail bzw. dem ACP-Nutzernamen im Format `<Name/E-Mail>:<Token>` in Base64-Enkodierung in dem `Authorization`-Header der gewünschten API-Anfrage gesendet werden.

Wenn nun beispielsweise eine Anmeldung mit E-Mail und Passwort an die User-Login-Route (`/user/login`) durchgeführt wird, muss der davon erhaltene Token gemeinsam mit der E-Mail immer in dem Header mitgesendet werden.

```
1 let xhr = new XMLHttpRequest();
2 // btoa() encodiert einen String in Base64
3 xhr.setRequestHeader('Authorization', btoa(email + ':' + token))
4 xhr.open(method, route);
```

Code-Snippet 7.1.: Eine Beispieldokumentation mit Authorization-Header in JavaScript

Wenn zum Beispiel die `email` und der `token` im obigen Code-Beispiel `josh@sokka.me` und `TXJ9Gb9s5gYIdGpPSDYlq6LnTp7rRMb` sind, dann ist der finale Header

7. REST-Route-Dokumentation

```
1 Authorization: Bearer  
am9zaEBzb2tryS5tZTpUWEo5R2I5czVnWUlkR3BQU0RZbHE2TG5UcGc3c1JNYg==
```

7.2. Rate-Limiting

Ein Rate-Limit hindert Nutzer daran, zu viele Anfragen an eine API(-Route) zu stellen. Das ist sinnvoll, wenn eine Anfrage einen serverseitig hohen Ressourcenaufwand verursacht (z. B. bei Anfragen für Berechnungen) oder die Anfrage auf sensitive Ressourcen zugreift (z. B. bei Nutzeranmeldungen).

Bei Sokka wird das Rate-Limit durch eine Open-Source-Library namens `express-rate-limit` realisiert. Die Library erlaubt die Angabe eines Zeitfensters in Millisekunden (`windowMs`) und eine Maximalanzahl an Anfragen von einer IP-Adresse innerhalb dieses Fensters (`max`).

Wenn die Maximalanzahl überschritten wurde, wird ein 421 HTTP-Code mit untenstehender Antwort zurückgegeben. [19]

```
1 const rateLimit = require('express-rate-limit');  
2 const limiter = rateLimit({  
3   windowMs: 1 * 1000 * 60, // 5 minutes  
4   max: 5,  
5   handler: (req, res) => {  
6     res.set('Content-Type', 'application/json');  
7     res.send({ success: false, message: 'Too many failed login attempts, please try again later' });  
8   }  
9 });
```

Code-Snippet 7.2.: Sokka-Implementation eines Rate-Limiters für eine REST-Route

Einige API-Routes verfügen über ein Rate-Limit, um Missbrauch vorzubeugen. Diese haben eine entsprechende Information dazu in der Dokumentation.

7. REST-Route-Dokumentation

7.3. Root-Routes

7.3.1. /image

```
1   GET /image
```

Beschreibung:

Ruft ein Bild via eindeutiger ID aus dem Bildspeicher des Servers ab. Wird genutzt, um Bilder von Produkten und Menüs in der App und im ACP anzuzeigen.

Authorization-Typ:

Keine

7.3.1.1. Parameter

`id: string` (benötigt)

→ Die eindeutige ID des gewünschten Bildes

7.3.1.2. Beispiel

```
1   GET /image?id=52af25abdf2ffced06f863b3c7a0bb42
```

Lädt das Bild mit der angegebenen ID vom Server.

7.3.1.3. Antwort (bei Erfolg)

`Content-Type: image/png`

```
1   Binaerdaten des Bildes
```

7. REST-Route-Dokumentation

7.3.2. /verify

```
1   GET /verify
```

Beschreibung:

Überprüft einen E-Mail-Verifizierungs-Token auf Validität. Diesen erhalten Nutzer nach der Erstellung eines Nutzerkontos via E-Mail.

Authorization-Typ:

Keine

7.3.2.1. Parameter

id: string (benötigt)
→ Der Verifizierungs-Token

7.3.2.2. Beispiel

```
1   GET /verify?id=9n6g7M7ow4qCNXW2nmkpYEavx8M32c%2  
     BRGk1gyYq2ymJ8MRVfx98Bb0bRqt9VrtC40YcqKWpRk86FAShSH4zhQ%3D%3D
```

Prüft den angegebenen Verifizierungstoken

7.3.2.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {  
2       "success": true,  
3       "message": "Successfully verified user"  
4   }
```

7.3.2.4. Antwort (bei ungültigem Token)

Content-Type: application/json

```
1   {  
2       "success": false,  
3       "message": "Invalid token '9n6g7M7ow4qCNXW2nmkpYEavx8M32c%2  
     BRGk1gyYq2ymJ8MRVfx98Bb0bRqt9VrtC40YcqKWpRk86FAShSH4zhQ%3D%3D'"  
4   }
```

7. REST-Route-Dokumentation

7.4. ACP

7.4.1. /acp/acpuser/get

```
1 GET /acp/acpuser/get
```

Beschreibung:

Gibt alle im System gespeicherten ACP-Nutzerkonten zurück. Wird genutzt, um eine Auflistung von ACP-Nutzerkonten im ACP zu erstellen.

Authorization-Typ:

ACP

7.4.1.1. Parameter

Keine

7.4.1.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "users": [
4         {
5             "name": "joshua",
6             "timestamp": "2021-03-02T18:22:02.043Z"
7         },
8         {
9             "name": "nicolaus",
10            "timestamp": "2021-03-02T18:23:08.890Z"
11        }
12    ]
13 }
```

Wobei `timestamp` der Timestamp der Erstellung des Nutzerkontos im ISO-8601-Format ist.

7. REST-Route-Dokumentation

7.4.2. /acp/acpuser/create

```
1 POST /acp/acpuser/create
```

Beschreibung:

Erstellt ein neues ACP-Nutzerkonto. Wird genutzt, um weitere ACP-Nutzerkonten im ACP zu erstellen.

Authorization-Typ:

ACP

7.4.2.1. Payload

`name: string` (benötigt)

→ Der gewünschte Name des neu anzulegenden ACP-Nutzerkontos

`password: string` (benötigt)

→ Das gewünschte Passwort des neu anzulegenden ACP-Nutzerkontos

7.4.2.2. Beispiel

```
1 POST /acp/acpuser/create
2 {
3     "name": "joshua",
4     "password": "sicheresPasswort"
5 }
```

Erstellt ein ACP-Nutzerkonto mit dem Namen „joshua“ und dem Passwort „sicheresPasswort“.

7.4.2.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully created ACP user with username 'joshua'"
4 }
```

7. REST-Route-Dokumentation

7.4.3. /acp/acpuser/delete

```
1 POST /acp/acpuser/delete
```

Beschreibung:

Löscht ein ACP-Nutzerkonto. Wird genutzt, um nicht mehr benötigte ACP-Nutzerkonten im ACP zu löschen.

Authorization-Typ:

ACP

7.4.3.1. Payload

name: string (benötigt)
→ Der Name des zu löschenen ACP-Nutzerkontos

7.4.3.2. Beispiel

```
1 POST /acp/acpuser/delete
2 {
3     "name": "joshua"
4 }
```

Löscht das ACP-Nutzerkonto mit dem Namen „joshua“.

7.4.3.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully deleted ACP user with username 'joshua'"
4 }
```

7. REST-Route-Dokumentation

7.4.4. /acp/config/get

```
1   GET /acp/config/get
```

Beschreibung:

Gibt alle im System gespeicherten Config-Einträge zurück. Wird genutzt, um eine Auflistung der Config-Einträge im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.4.1. Parameter

Keine

7.4.4.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2       "success": true,
3       "configEntries": [
4           {
5               "key": "closingTime",
6               "friendlyName": "Closing Time",
7               "type": "TIME",
8               "value": "20:00"
9           }
10      ]
11  }
```

Wobei type ein Enum aus den Werten INTEGER, STRING und TIME ist.

7. REST-Route-Dokumentation

7.4.5. /acp/config/update

```
1 POST /acp/config/update
```

Beschreibung:

Aktualisiert einen Config-Eintrag, insbesondere dessen Wert. Wird genutzt, um im ACP die Werte der Config-Einträge änderbar zu machen.

Authorization-Typ:

ACP

7.4.5.1. Payload

`key: string` (benötigt)

→ Der Key des zu aktualisierenden Config-Eintrags

`type: string` (optional)

→ Der Typ des zu aktualisierenden Config-Eintrags

`friendlyName: string` (optional)

→ Der Anzeigename des zu aktualisierenden Config-Eintrags

`value: string` (optional)

→ Der Wert des zu aktualisierenden Config-Eintrags

Wobei `type` nur `INTEGER`, `STRING` oder `TIME` sein kann.

7.4.5.2. Beispiel

```
1 POST /acp/config/update
2 {
3     "key": "closingTime",
4     "value": "17:00"
5 }
```

Setzt den Zeitwert vom Config-Eintrag mit dem Key `closingTime` auf den Wert 17:00.

7.4.5.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated config entry"
4 }
```

7. REST-Route-Dokumentation

7.4.6. /acp/group/get

```
1  GET /acp/group/get
```

Beschreibung:

Gibt alle im System gespeicherten Nutzergruppen zurück. Wird genutzt, um eine Auflistung aller Gruppen im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.6.1. Parameter

Keine

7.4.6.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1  {
2      "success": true,
3      "groups": [
4          {
5              "id": 1,
6              "name": "Default",
7              "rebate": 0
8          },
9          {
10             "id": 2,
11             "name": "VIP",
12             "rebate": 10
13         }
14     ]
15 }
```

7. REST-Route-Dokumentation

7.4.7. /acp/group/create

```
1 POST /acp/group/create
```

Beschreibung:

Erstellt eine neue Nutzergruppe. Wird genutzt, um neue Gruppen via ACP zu erstellen.

Authorization-Typ:

ACP

7.4.7.1. Payload

name: **string** (benötigt)

→ Der gewünschte Name des neu anzulegenden ACP-Nutzerkontos

rebate: **number** (benötigt)

→ Das gewünschte Passwort des neu anzulegenden ACP-Nutzerkontos

7.4.7.2. Beispiel

```
1 POST /acp/acpuser/create
2 {
3     "name": "Teachers",
4     "rebate": 25
5 }
```

Erstellt eine Nutzergruppe mit dem Namen „Teachers“ und einem Rabatt von 25 %.

7.4.7.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully created group with id '3'"
4 }
```

7. REST-Route-Dokumentation

7.4.8. /acp/group/update

```
1 POST /acp/group/update
```

Beschreibung:

Aktualisiert den Namen oder den Rabatt einer Gruppe. Wird genutzt, um Gruppen im ACP änderbar zu machen.

Authorization-Typ:

ACP

7.4.8.1. Payload

```
id: number (benötigt)
→ Die ID der zu aktualisierenden Gruppe

name: string (optional)
→ Der Name der zu aktualisierenden Gruppe

rebate: number (optional)
→ Der Rabatt der zu aktualisierenden Gruppe
```

7.4.8.2. Beispiel

```
1 POST /acp/config/update
2 {
3     "id": 3,
4     "rebate": 20
5 }
```

Setzt den Rabatt der Gruppe mit der ID 3 zu 20 %.

7.4.8.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated group"
4 }
```

7. REST-Route-Dokumentation

7.4.9. /acp/group/delete

```
1 POST /acp/group/delete
```

Beschreibung:

Löscht eine Gruppe. Wird genutzt, um nicht mehr benötigte Gruppen im ACP zu löschen.

Authorization-Typ:

ACP

7.4.9.1. Payload

`id: number` (benötigt)
→ Die ID der zu löschenen Gruppe

7.4.9.2. Beispiel

```
1 POST /acp/config/delete
2 {
3     "id": 3
4 }
```

Löscht die Gruppe mit der ID 3.

7.4.9.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated group"
4 }
```

7. REST-Route-Dokumentation

7.4.10. /acp/menu/get

```
1   GET /acp/menu/get
```

Beschreibung:

Gibt alle im System gespeicherten Menüs oder nur eines zurück. Wird genutzt, um eine Auflistung aller Menüs im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.10.1. Parameter

id: number (optional) → Die ID des gewünschten Menüs

7.4.10.2. Beispiel

```
1   GET /acp/menu/get?id=2
```

Gibt das Menü mit der ID 2 zurück.

7.4.10.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "menus": [
4       {
5         "id": 2,
6         "category_id": 3,
7         "name": "Cheesy 'n' Coke",
8         "image_id": "23c94790921c89f56e0629d1282033f0",
9         "price": 5,
10        "entries": [
11          {
12            "id": 2,
13            "product_id": 2,
14            "menu_id": 1,
15            "title_id": 1
16          },
17          {
18            "id": 3,
19            "product_id": 8,
20            "menu_id": 1,
21            "title_id": 3
22          }
23        ]
24      }
25    ]
26 }
```

7. REST-Route-Dokumentation

7.4.11. /acp/menu/create

```
1 POST /acp/menu/create
```

Beschreibung:

Erstellt eine neues Menü. Wird genutzt, um neue Menüs via ACP zu erstellen.

Authorization-Typ:

ACP

7.4.11.1. Payload

name: **string** (benötigt)

→ Der gewünschte Name des neu anzulegenden Menüs

category_id: **number** (benötigt)

→ Die gewünschte Kategorie-ID des neu anzulegenden Menüs

image_id: **string** (benötigt)

→ Die gewünschte Bild-ID des neu anzulegenden Menüs

price: **number** (benötigt)

→ Der gewünschte Preis des neu anzulegenden Menüs

entries: **MenuEntry[]** (optional)

→ Die initialen Produkteinträge des Menüs

7.4.11.2. Beispiel

```
1 POST /acp/menu/create
2 {
3     "name": "Mittagsmenue",
4     "category_id": 25,
5     "image_id": "23c94790921c89f56e0629d1282033f0",
6     "price": 5,
7     "entries": [
8         {
9             "product_id": "3",
10            "title_id": 1
11        }
12    ]
13 }
```

Erstellt ein Menü mit den übertragenen Daten.

7.4.11.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully created menu with id '5'"
4 }
```

7. REST-Route-Dokumentation

7.4.12. /acp/menu/update

```
1 POST /acp/menu/update
```

Beschreibung:

Aktualisiert den Namen, die Kategorie, das Bild, den Preis oder den Inhalt eines Menüs. Wird genutzt, um Menüs im ACP änderbar zu machen.

Authorization-Typ:

ACP

7.4.12.1. Payload

`id: number` (benötigt)
→ Die ID des zu aktualisierenden Menüs

`name: string` (optional)
→ Der Name des zu aktualisierenden Menüs

`image_id: string` (optional)
→ Die Bild-ID des zu aktualisierenden Menüs

`category_id: number` (optional)
→ Die Kategorie-ID des zu aktualisierenden Menüs

`price: number` (optional)
→ Der Preis des zu aktualisierenden Menüs

`entries: MenuEntry[]` (optional)
→ Die Produkteinträge des zu aktualisierenden Menüs

7.4.12.2. Beispiel

```
1 POST /acp/menu/update
2 {
3     "id": 5,
4     "name": "Abendmenue"
5 }
```

Setzt den Namen des Menüs mit der ID 5 zu „Abendmenue“.

7.4.12.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated menu"
4 }
```

7. REST-Route-Dokumentation

7.4.13. /acp/menu/delete

```
1 POST /acp/menu/delete
```

Beschreibung:

Löscht ein Menü. Wird genutzt, um nicht mehr benötigte Menüs im ACP zu löschen.

Authorization-Typ:

ACP

7.4.13.1. Payload

`id: number` (benötigt)
→ Die ID des zu löschen Menü

7.4.13.2. Beispiel

```
1 POST /acp/menu/delete
2 {
3     "id": 5
4 }
```

Löscht das Menü mit der ID 5.

7.4.13.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully deleted menu with id '5'"
4 }
```

7. REST-Route-Dokumentation

7.4.14. /acp/menu/title/get

```
1   GET /acp/menu/title/get
```

Beschreibung:

Gibt alle im System gespeicherten Menütitel zurück. Wird genutzt, um eine Auflistung aller Menütitel im Bearbeitungsdialog für Menüs im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.14.1. Parameter

Keine

7.4.14.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2       "success": true,
3       "menutitles": [
4           {
5               "id": 1,
6               "name": "Starter"
7           },
8           {
9               "id": 2,
10              "name": "Main Dish"
11           },
12           {
13               "id": 3,
14               "name": "Dessert"
15           }
16       ]
17   }
```

7. REST-Route-Dokumentation

7.4.15. /acp/menu/category/get

```
1  GET /acp/menu/category/get
```

Beschreibung:

Gibt alle im System gespeicherten Menükategorien zurück. Wird genutzt, um eine Auflistung aller Menükategorien auf der Bearbeitungsseite für Menüs im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.15.1. Parameter

Keine

7.4.15.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1  {
2      "success": true,
3      "menucategories": [
4          {
5              "id": 1,
6              "name": "For Kids"
7          },
8          {
9              "id": 2,
10             "name": "Diner"
11         },
12         {
13             "id": 3,
14             "name": "Lunch Menus"
15         }
16     ]
17 }
```

7. REST-Route-Dokumentation

7.4.16. /acp/order/get

```
1   GET /acp/order/get
```

Beschreibung:

Gibt alle im System gespeicherten Bestellungen für ein bestimmtes Datum zurück. Wird genutzt, um die Bestellungsübersicht im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.16.1. Parameter

date: string (benötigt)
→ Das gewünschte Bestellungsdatum

7.4.16.2. Beispiel

```
1   GET /acp/order/get?date=2021-03-01
```

Gibt alle Bestellungen, welche am 1. März 2021 getätigten wurden, zurück.

7.4.16.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "orders": [
4       {
5         "id": 60,
6         "user_id": 26,
7         "timestamp": "2021-03-01T08:29:34.000Z"
8         "state": "VALID",
9         "rebate": 10,
10        "total": 6,
11        "menuOrders": [
12          {
13            "menu_id": 1,
14            "quantity": 1,
15            "menu": <Menu Object>
16          },
17          {
18            "productOrders": [
19              {
20                "product_id": 1,
21                "quantity": 1,
22                "product": <Product Object>
23              }
24            ]
25          }
26        ]
27      }
28    }
```

7. REST-Route-Dokumentation

7.4.17. /acp/order/validate

```
1 GET /acp/order/validate
```

Beschreibung:

Prüft die Validität einer Bestellung. Wird genutzt, um im Admin-Client die Bestellung aus einem gelesenen QR-Code zu überprüfen.

Authorization-Typ:

ACP

7.4.17.1. Parameter

order: string (benötigt)

→ Die zu validierende Bestellung im Format „userId:orderId“

7.4.17.2. Beispiel

```
1 GET /acp/order/get?validate?order=25:10
```

Validiert die Bestellung mit der ID 10 vom Nutzer 25.

7.4.17.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "valid": true,
4     "order": {
5         "id": 60,
6         "user_id": 26,
7         "timestamp": "2021-03-01T08:29:34.000Z"
8         "state": "VALID",
9         "rebate": 10,
10        "total": 6,
11        "menuOrders": [
12            "menu_id": 1,
13            "quantity": 1,
14            "menu": <Menu Object>
15        ],
16        "productOrders": [
17            "product_id": 1,
18            "quantity": 1,
19            "product": <Product Object>
20        ]
21    }
22}
```

7. REST-Route-Dokumentation

7.4.17.4. Antwort (bei ungültiger Bestellung)

Content-Type: application/json

```
1  {
2      "success": true,
3      "valid": false,
4      "reasons": [
5          "Order does not belong to user",
6          "Order was not created yesterday",
7          "Order has already been invalidated"
8      ]
9  }
```

Wobei `reasons` immer ein String-Array mit allen zutreffenden Begründungen ist, weshalb der übertragene Code ungültig ist.

7. REST-Route-Dokumentation

7.4.18. /acp/order/invalidate

```
1 POST /acp/order/invalidate
```

Beschreibung:

Invalidiert eine Bestellung. Wird genutzt, um eingelöste Bestellungen bzw. QR-Codes an der Kasse zu invalidieren. Achtung: Diese Route prüft nicht, ob die Bestellung einlösbar ist. Dafür ist /acp/order/validate zu nutzen.

Authorization-Typ:

ACP

7.4.18.1. Payload

```
order: string (benötigt)  
→ Die zu invalidierende Bestellung im Format „userId:orderId“
```

7.4.18.2. Beispiel

```
1 POST /acp/menu/invalidate  
2 {  
3     "order": "25:10",  
4 }
```

Invalidiert die Bestellung mit der ID 10.

7.4.18.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {  
2     "success": true,  
3     "message": "Successfully invalidated order"  
4 }
```

7. REST-Route-Dokumentation

7.4.19. /acp/product/get

```
1 GET /acp/product/get
```

Beschreibung:

Gibt alle im System gespeicherten Produkte oder nur eines zurück. Wird genutzt, um eine Auflistung aller Produkte im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.19.1. Parameter

`id: number` (optional) → Die ID des gewünschten Produkts. Es können auch mehrere Produkte abgefragt werden (mit , getrennt)

7.4.19.2. Beispiel

```
1 GET /acp/product/get?id=6
```

Gibt das Produkt mit der ID 6 zurück.

7.4.19.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "products": [
4         {
5             "id": 1,
6             "category_id": 2,
7             "name": "Coca-Cola",
8             "image_id": "52af25abdf2ffced06f863b3c7a0bb42",
9             "price": 2.5,
10            "hidden": false
11        }
12    ]
13 }
```

7. REST-Route-Dokumentation

7.4.20. /acp/product/create

```
1 POST /acp/product/create
```

Beschreibung:

Erstellt eine neues Produkt. Wird genutzt, um neue Produkte via ACP zu erstellen.

Authorization-Typ:

ACP

7.4.20.1. Payload

name: **string** (benötigt)

→ Der gewünschte Name des neu anzulegenden Produkts

category_id: **number** (benötigt)

→ Die gewünschte Kategorie-ID des neu anzulegenden Produkts

image_id: **string** (benötigt)

→ Die gewünschte Bild-ID des neu anzulegenden Produkts

price: **number** (benötigt)

→ Der gewünschte Preis des neu anzulegenden Produkts

hidden: **boolean** (optional)

→ **true**, wenn das Produkt versteckt sein soll, **false** andernfalls

7.4.20.2. Beispiel

```
1 POST /acp/product/create
2 {
3     "name": "Coke",
4     "category_id": 2,
5     "image_id": "8c255a9642248b20f1f1bbe13c2acf14",
6     "price": 3,
7     "hidden": false
8 }
```

Erstellt ein Produkt mit den übertragenen Daten.

7.4.20.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully created product with id '4'"
4 }
```

7. REST-Route-Dokumentation

7.4.21. /acp/product/update

```
1 POST /acp/product/update
```

Beschreibung:

Aktualisiert den Namen, die Kategorie, das Bild, den Preis oder den Versteckt-Status eines Produkts. Wird genutzt, um Produkte im ACP änderbar zu machen.

Authorization-Typ:

ACP

7.4.21.1. Payload

`id: number` (benötigt)
→ Die ID des zu aktualisierenden Produkts

`name: string` (optional)
→ Der Name des zu aktualisierenden Produkts

`image_id: string` (optional)
→ Die Bild-ID des zu aktualisierenden Produkts

`category_id: number` (optional)
→ Die Kategorie-ID des zu aktualisierenden Produkts

`price: number` (optional)
→ Der Preis des zu aktualisierenden Produkts

`hidden: boolean` (optional)
→ Der Versteckt-Status des zu aktualisierenden Produkts

7.4.21.2. Beispiel

```
1 POST /acp/product/update
2 {
3     "id": 3,
4     "price": "4.5"
5 }
```

Setzt den Preis des Produkts mit der ID 3 zu 4.50 Euro.

7.4.21.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated product"
4 }
```

7. REST-Route-Dokumentation

7.4.22. /acp/product/delete

```
1 POST /acp/product/delete
```

Beschreibung:

Löscht ein Produkt. Wird genutzt, um nicht mehr benötigte Produkte im ACP zu löschen.

Authorization-Typ:

ACP

7.4.22.1. Payload

`id: number` (benötigt)
→ Die ID des zu löschen Produkts

7.4.22.2. Beispiel

```
1 POST /acp/product/delete
2 {
3     "id": 3
4 }
```

Löscht das Menü mit der ID 3.

7.4.22.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully deleted product with id '3'"
4 }
```

7. REST-Route-Dokumentation

7.4.23. /acp/product/category/get

```
1   GET /acp/product/category/get
```

Beschreibung:

Gibt alle im System gespeicherten Produktkategorien zurück. Wird genutzt, um eine Auflistung aller Produktkategorien auf der Bearbeitungsseite für Produkte im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.23.1. Parameter

Keine

7.4.23.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "productcategories": [
4       {
5         "id": 1,
6         "name": "Beverages"
7       },
8       {
9         "id": 2,
10        "name": "Snacks"
11      },
12      {
13        "id": 3,
14        "name": "Burgers"
15      }
16    ]
17 }
```

7. REST-Route-Dokumentation

7.4.24. /acp/user/get

```
1   GET /acp/user/get
```

Beschreibung:

Gibt alle im System gespeicherten Nutzer zurück. Wird genutzt, um eine Auflistung aller Nutzer im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.24.1. Parameter

Keine

7.4.24.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1  {
2      "success": true,
3      "users": [
4          {
5              "id": 1,
6              "email": "jos.winkler@tsn.at",
7              "verified": 1,
8              "blocked": 0,
9              "group_id": 3,
10             "timestamp": "2021-01-19T19:42:13.000Z",
11             "password": "$2b$10$MZ2nJ.
12                                         Vvp1EZCNRRViTbduQp6ZIoU4rkv4paetQdkEwMDRLDpqSQ6"
13         }
14     ]}
```

7. REST-Route-Dokumentation

7.4.25. /acp/user/update

```
1 POST /acp/user/update
```

Beschreibung:

Aktualisiert den E-Mail-Adresse, Verifikationsstatus oder Gruppe eines Nutzerkontos. Wird genutzt, um Nutzerkonten im ACP änderbar zu machen.

Authorization-Typ:

ACP

7.4.25.1. Payload

`id: number` (benötigt)

→ Die ID des zu aktualisierenden Nutzerkontos

`email: string` (optional)

→ Die E-Mail-Adresse des zu aktualisierenden Nutzerkontos

`verified: boolean` (optional)

→ Der Verifikationsstatus des zu aktualisierenden Nutzerkontos

`group: number` (optional)

→ Die Gruppen-ID des zu aktualisierenden Nutzerkontos

7.4.25.2. Beispiel

```
1 POST /acp/user/update
2 {
3     "id": 2,
4     "verified": true
5 }
```

Setzt den Verifikationsstatus des Nutzerkontos mit der ID 2 auf `true`.

7.4.25.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully updated users"
4 }
```

7. REST-Route-Dokumentation

7.4.26. /acp/user/delete

```
1 POST /acp/user/delete
```

Beschreibung:

Löscht ein Nutzerkonto. Wird genutzt, um nicht mehr benötigte oder ungewollte Nutzerkonten im ACP zu löschen.

Authorization-Typ:

ACP

7.4.26.1. Payload

`id: number` (benötigt)
→ Die ID des zu löschenen Nutzerkontos

7.4.26.2. Beispiel

```
1 POST /acp/user/delete
2 {
3     "id": 5
4 }
```

Löscht das Nutzerkonto mit der ID 5.

7.4.26.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully deleted user with id '3'"
4 }
```

7. REST-Route-Dokumentation

7.4.27. /acp/login

```
1 POST /acp/login
```

Beschreibung:

Führt eine Anmeldung im ACP bei einem ACP-Nutzerkonto durch.

Authorization-Typ:

Keine

7.4.27.1. Payload

name: **string** (benötigt)

→ Der Name des ACP-Nutzerkontos, bei dem die Anmeldung durchgeführt werden soll

password: **string** (benötigt)

→ Das Passwort des ACP-Nutzerkontos, bei dem die Anmeldung durchgeführt werden soll

7.4.27.2. Beispiel

```
1 POST /acp/login
2 {
3     "name": "admin",
4     "password": "securepassword123"
5 }
```

Führt eine Anmeldung für das ACP-Nutzerkonto admin mit dem Passwort securepassword123 durch und retourniert einen generierten Session-Token.

7.4.27.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Credentials validated",
4     "token": "7sGCYBi5796mvPYe1Jg5pybMuJXBGa%2B4",
5     "name": "admin"
6 }
```

7.4.27.4. Rate Limit

Diese API-Route verfügt über ein Rate-Limit, um Missbrauch vorzubeugen. Es können maximal 5 Anmeldungen innerhalb von 5 Minuten von derselben IP-Adresse durchgeführt werden.

7. REST-Route-Dokumentation

7.4.28. /acp/logout

```
1 POST /acp/logout
```

Beschreibung:

Inaktiviert einen Session-Token eines ACP-Nutzerkontos und führt damit eine Abmeldung eines ACP-Nutzerkontos im ACP durch.

Authorization-Typ:

Keine

7.4.28.1. Payload

```
name: string (benötigt)
→ Der Name des ACP-Nutzerkontos, bei dem die Abmeldung durchgeführt werden soll token
: string (benötigt)
→ Der Session-Token des ACP-Nutzerkontos, der inaktiviert werden soll.
```

7.4.28.2. Beispiel

```
1 POST /acp/logout
2 {
3     "name": "admin",
4     "token": "7sGCYBi5796mvPYelJg5pybMuJXBGa%2B4"
5 }
```

Führt eine Anmeldung für das ACP-Nutzerkonto `admin` mit dem Passwort `securepassword123` durch und returniert einen generierten Session-Token.

7.4.28.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully invalidated session"
4 }
```

7. REST-Route-Dokumentation

7.4.29. /acp/validate

```
1 POST /acp/validate
```

Beschreibung:

Prüft, ob der Session-Token eines ACP-Nutzerkontos gültig ist. Wird genutzt, um bei Aufruf des ACPs einen vorhandenen Session-Cookie zu prüfen.

Authorization-Typ:

Keine

7.4.29.1. Payload

```
name: string (benötigt)
→ Der Name des ACP-Nutzerkontos, bei dem die Validierung durchgeführt werden soll token:
  string (benötigt)
→ Der Session-Token des ACP-Nutzerkontos, bei dem die Validierung durchgeführt werden soll
```

7.4.29.2. Beispiel

```
1 POST /acp/validate
2 {
3     "name": "admin",
4     "token": "7sGCYBi5796mvPYelJg5pybMuJXBGa%2B4"
5 }
```

Validiert den Session-Token `7sGCYBi5796mvPYelJg5pybMuJXBGa%2B4` für das ACP-Nutzerkonto mit dem Namen `admin`.

7.4.29.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "ACP token for this email is valid"
4 }
```

7.4.29.4. Antwort (bei ungültigen Token)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Could not validate ACP token for username 'admin'"
4 }
```

7. REST-Route-Dokumentation

7.4.30. /acp/image

```
1 POST /acp/image
```

Beschreibung:

Lädt ein neues Bild hoch. Wird genutzt, um die Bilder-Upload-Funktion im ACP zu ermöglichen.

Authorization-Typ:

ACP

7.4.30.1. Payload

buffer: ArrayBuffer (benötigt)

→ Der Uint8Array Buffer aus dem das hochzuladende Bild besteht

7.4.30.2. Beispiel

```
1 POST /acp/image
2 {
3     "buffer": {
4         "type": "Buffer",
5         data: [<Binary Data>]
6     }
7 }
```

Lädt ein Bild mit den Binärdaten im übergebenen ArrayBuffer hoch.

7.4.30.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Image saved successfully"
4 }
```

7. REST-Route-Dokumentation

7.4.31. /acp/status

```
1   GET /acp/status
```

Beschreibung:

Gibt aktuelle Statistiken und Auslastungsdaten des Sokka-Systems zurück. Werte werden maximal alle 5 Sekunden ausgelesen.

Authorization-Typ:

ACP

7.4.31.1. Parameter

Keine

7.4.31.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2       "success": true,
3       "status": [
4           "users": 11,
5           "ordersToday": 2,
6           "registeredProducts": 25,
7           "registeredMenus": 5,
8           "sysUptime": 21069220,
9           "procUptime": 138350.661,
10          "totalMem": 64314.59375,
11          "freeMem": 515.703125,
12          "platform": "linux",
13          "cpuCount": 12,
14          "cpuUsage": 0.209398186314922169
15      ]
16  }
```

sysUptime und procUptime sind in Sekunden, totalMem und freeMem in Bytes und cpuUsage in Prozent.

7. REST-Route-Dokumentation

7.5. Menu

7.5.1. /menu/get

```
1 GET /menu/get
```

Beschreibung:

Gibt alle im System gespeicherten Menüs oder nur eines zurück. Wird genutzt, um eine Auflistung aller Menüs im Client zu ermöglichen.

Authorization-Typ:

App

7.5.1.1. Parameter

`id: number` (optional) → Die ID des gewünschten Menüs.

7.5.1.2. Beispiel

```
1 GET /menu/get?id=4
```

Gibt das Menü mit der ID 4 zurück.

7.5.1.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "data": [
4         {
5             "id": 6,
6             "category_id": 4,
7             "name": "Cheeseburger Menu",
8             "image_id": "23c94790921c89f56e0629d1282033f0",
9             "price": 3,
10            "entries": [
11                {
12                    "id": 2,
13                    "product_id": 2,
14                    "menu_id": 1,
15                    "title_id": 1
16                }
17            ]
18        ]
19    }
20 }
```

7. REST-Route-Dokumentation

7.5.2. /menu/title/get

```
1   GET /menu/title/get
```

Beschreibung:

Gibt einen im System gespeicherten Menütitel zurück. Wird genutzt, um den Produkten in Menüs im Client den richtigen Titel zuzuordnen.

Authorization-Typ:

App

7.5.2.1. Parameter

id: number (benötigt) → Die ID des gewünschten Menütitels.

7.5.2.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "data": {
4       "id": 1,
5       "name": "Starter"
6     }
7 }
```

7. REST-Route-Dokumentation

7.6. Product

7.6.1. /product/get

```
1 GET /product/get
```

Beschreibung:

Gibt alle im System gespeicherten Produkte oder nur eines zurück. Wird genutzt, um eine Auflistung aller Produkte im Client zu ermöglichen.

Authorization-Typ:

App

7.6.1.1. Parameter

`id: number` (optional) → Die ID des gewünschten Produkts.

7.6.1.2. Beispiel

```
1 GET /product/get?id=4
```

Gibt das Produkt mit der ID 4 zurück.

7.6.1.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "data": [
4         {
5             "id": 1,
6             "category_id": 2,
7             "name": "Coca-Cola",
8             "image_id": "52af25abdf2ffced06f863b3c7a0bb42",
9             "price": 2.5,
10            "hidden": false
11        }
12    ]
13 }
```

7. REST-Route-Dokumentation

7.7. Order

7.7.1. /order/get

```
1   GET /order/get
```

Beschreibung:

Gibt alle Bestellungen des angemeldeten Nutzerkontos zurück, oder nur jene, welche an einem angegebenen Datum getätigt wurden. Wird genutzt, um getätigte Bestellungen im Client aufzulisten.

Authorization-Typ:

App

7.7.1.1. Parameter

`date: string (optional)` → Das Datum der abzurufenden Bestellungen.

7.7.1.2. Beispiel

```
1   GET /product/get?date=2021-03-03
```

Gibt alle Bestellungen des angemeldeten Nutzerkontos zurück, welche am 3. März 2021 getätigten wurden.

7.7.1.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "orders": [
4       {
5         "id": 60,
6         "user_id": 26,
7         "timestamp": "2021-03-01T08:29:34.000Z"
8         "state": "VALID",
9         "rebate": 10,
10        "total": 6,
11        "menuOrders": [
12          "menu_id": 1,
13          "quantity": 1,
14          "menu": <Menu Object>
15        ],
16        "productOrders": [
17          "product_id": 1,
18          "quantity": 1,
19          "product": <Product Object>
20        ]
21      }
22    ]
23 }
```

7. REST-Route-Dokumentation

7.7.2. /order/create

```
1 POST /order/create
```

Beschreibung:

Erstellt eine neue Bestellung. Wird genutzt, um neue Bestellungen im Client zu t tigen.

Authorization-Typ:

App

7.7.2.1. Payload

```
products: ProductOrder[] (ben tigt)
→ Der Produktbestellungen dieser Bestellung

menus: MenuOrder[] (ben tigt)
→ Die Men bestellungen dieser Bestellung
```

7.7.2.2. Beispiel

```
1 POST /order/create
2 {
3     "products": [
4         {
5             "product_id": 3,
6             "quantity": 2
7         }
8     ],
9     "menus": [
10    {
11        "menu_id": 2,
12        "quantity": 1
13    }
14 ]
15 }
```

T tigt eine Bestellung von 2x Produkt mit ID 3 und 1x Men  mit ID 2.

7.7.2.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Order successfully created"
4 }
```

7.8. User

7.8.1. /user/create

```
1 POST /user/create
```

Beschreibung:

Erstellt ein neues Nutzerkonto. Wird genutzt, um neue Registrierungen für Nutzerkonten im Client zu ermöglichen.

Authorization-Typ:

Keine

7.8.1.1. Payload

`email: string` (benötigt)

→ Die E-Mail-Adresse des zu erstellenden Nutzerkontos

`password: string` (benötigt)

→ Das Passwort des zu erstellenden Nutzerkontos

`tos: boolean` (benötigt)

→ Signalisiert eine Zustimmung zu den allgemeinen Nutzungsbedingungen des Sokka-Systems (muss `true` sein)

`privacypolicy: boolean` (benötigt)

→ Signalisiert eine Zustimmung zu der Datenschutzerklärung des Sokka-Systems (muss `true` sein)

7.8.1.2. Beispiel

```
1 POST /user/create
2 {
3     "email": "meineEmail@sokka.me",
4     "password": "test123"
5 }
```

Erstellt ein neues Nutzerkonto mit der E-Mail-Adresse `meineEmail@sokka.me` und dem Passwort `test123`.

7.8.1.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully created user",
4     "token": "eYiKOYqtihKgVknurEqlluc3mhawIxAo"
5 }
```

7. REST-Route-Dokumentation

Bei Erfolg wird ein Session-Token generiert und zurückgegeben. Außerdem wird eine Verifikations-E-Mail an die übergebene E-Mail-Adresse gesendet.

7.8.1.4. Rate Limit

Diese API-Route verfügt über ein Rate-Limit, um Missbrauch vorzubeugen. Es können maximal 5 Registrierungen innerhalb von 10 Minuten von derselben IP-Adresse durchgeführt werden.

7. REST-Route-Dokumentation

7.8.2. /user/login

```
1 POST /user/login
```

Beschreibung:

Führt eine Anmeldung bei einem existenten Nutzerkonto durch. Wird genutzt, um neue Anmeldungen für Nutzerkonten im Client zu ermöglichen.

Authorization-Typ:

Keine

7.8.2.1. Payload

email: string (benötigt)

→ Die E-Mail-Adresse des Nutzerkontos, bei dem die Anmeldung durchgeführt werden soll.

password: string (benötigt)

→ Das Passwort des Nutzerkontos, bei dem die Anmeldung durchgeführt werden soll.

7.8.2.2. Beispiel

```
1 POST /user/login
2 {
3     "email": "meineEmail@sokka.me",
4     "password": "test123"
5 }
```

Führt eine Anmeldung beim Nutzerkonto mit der E-Mail-Adresse `meineEmail@sokka.me` und dem Passwort `test123` durch.

7.8.2.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Credentials validated",
4     "token": "eYiKOYqtihKgVknurEq1luc3mhawIxAo"
5 }
```

Bei Erfolg wird ein Session-Token generiert und zurückgegeben.

7.8.2.4. Rate Limit

Diese API-Route verfügt über ein Rate-Limit, um Missbrauch vorzubeugen. Es können maximal 5 Anmeldungen innerhalb von 5 Minuten von derselben IP-Adresse durchgeführt werden.

7. REST-Route-Dokumentation

7.8.3. /user/logout

```
1 POST /user/logout
```

Beschreibung:

Führt eine Abmeldung bei einem existenten Nutzerkonto durch. Wird genutzt, um Abmeldungen von Nutzerkonten im Client zu ermöglichen.

Authorization-Typ:

App

7.8.3.1. Payload

email: string (benötigt)

→ Die E-Mail-Adresse des Nutzerkontos, bei dem die Abmeldung durchgeführt werden soll.

token: string (benötigt)

→ Der Token des Nutzerkontos, bei dem die Abmeldung durchgeführt werden soll.

7.8.3.2. Beispiel

```
1 POST /user/logout
2 {
3     "email": "meineEmail@sokka.me",
4     "token": "eYiKOYqtihKgVknurEq1luc3mhawIxAo"
5 }
```

Führt eine Abmeldung beim Nutzerkonto mit der E-Mail-Adresse `meineEmail@sokka.me` für die Session mit dem Token `eYiKOYqtihKgVknurEq1luc3mhawIxAo` durch.

7.8.3.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Successfully invalidated session"
4 }
```

7. REST-Route-Dokumentation

7.8.4. /user/validate

```
1 POST /user/validate
```

Beschreibung:

Prüft, ob der Session-Token eines Nutzerkontos gültig ist. Wird genutzt, um bei Aufruf des Clients einen vorhandenen Session-Cookie zu prüfen.

Authorization-Typ:

App

7.8.4.1. Payload

email: string (benötigt)

→ Die E-Mail-Adresse des Nutzerkontos, bei dem die Validierung durchgeführt werden soll

token: string (benötigt)

→ Der Token des Nutzerkontos, bei dem die Validierung durchgeführt werden soll

7.8.4.2. Beispiel

```
1 POST /user/validate
2 {
3     "email": "meineEmail@sokka.me",
4     "token": "eYiKOYqtihKgVknurEq1luc3mhawIxAo"
5 }
```

Validiert den Session-Token eYiKOYqtihKgVknurEq1luc3mhawIxAo für das Nutzerkonto mit der E-Mail-Adresse meineEmail@sokka.me.

7.8.4.3. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "Token for this email is valid"
4 }
```

7.8.4.4. Antwort (bei ungültigen Token)

Content-Type: application/json

```
1 {
2     "success": false,
3     "message": "Could not validate token for email 'meineEmail@sokka.me'"
4 }
```

7. REST-Route-Dokumentation

7.8.5. /acp/user/delete

```
1 POST /acp/user/delete
```

Beschreibung:

Löscht das aktuell angemeldete Nutzerkonto. Wird genutzt, um DSGVO-konform Nutzern die Löschung ihres Kontos und ihrer Daten zu ermöglichen.

Authorization-Typ:

App

7.8.5.1. Payload

Keiner

7.8.5.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1 {
2     "success": true,
3     "message": "User deleted successfully"
4 }
```

7. REST-Route-Dokumentation

7.8.6. /user/request

```
1   GET /user/request
```

Beschreibung:

Gibt alle Daten zurück, welche über den angemeldeten Nutzer im System gespeichert sind. Wird genutzt, um einen Datenexport nach DSGVO zu ermöglichen.

Authorization-Typ:

App

7.8.6.1. Parameter

Keine

7.8.6.2. Antwort (bei Erfolg)

Content-Type: application/json

```
1   {
2     "success": true,
3     "data": {
4       "userData": {
5         ...
6       },
7       "orderData": {
8         ...
9       },
10      "sessionData": {
11        ...
12      },
13      "verificationData": {
14        ...
15      }
16    }
17 }
```

Hierbei stellt `userData` alle Nutzer-Metadaten wie E-Mail-Adresse oder Zeitpunkt der Registrierung dar. `orderData` beinhaltet alle Daten über Bestellungen, welche dem Nutzer zugeordnet werden können. `sessionData` alle Daten über Sessions, welche dem Nutzer zugeordnet werden können und `verificationData` alle Daten über die Verifizierung des Nutzers (beispielsweise die generierte Verifizierungs-URL).

7.8.6.3. Rate Limit

Diese API-Route verfügt über ein Rate-Limit, um Missbrauch vorzubeugen. Es kann maximal eine Abfrage innerhalb von 24 Stunden von derselben IP-Adresse durchgeführt werden.

Teil IV.

ACP

8. Allgemeines zum ACP

Das **ACP** (*Admin Control Panel*) stellt die zentrale Verwaltungsstelle für das Sokka-System dar. Der allgemeine Status des gesamten Systems, die registrierten Nutzer, die eingestellten Produkte, die Menüs etc. können im ACP eingesehen und bearbeitet werden.

8.1. Anmeldung

Der Zugriff zum ACP ist durch eine Anmeldemaske mit Nutzernamen und Passwort geschützt. Es können beliebig viele Nutzeraccounts für das ACP erstellt werden. So ist es möglich, neben einem ausgewählten Administrator beispielsweise auch den Kassierer darauf zugreifen zu lassen.

Bei einem Zugriff auf das ACP wird nach den Cookies `sokka_token` und `sokka_username` gesucht. Sind diese vorhanden, werden sie überprüft und bei Erfolg wird die Startseite des ACPS gerendert. Sind sie nicht vorhanden oder ungültig, so wird die Login-Seite gerendert.

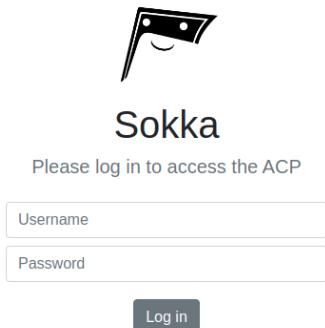


Abbildung 8.1.: Die Sokka-ACP-Anmeldeseite

Es liegt in der Natur einer React-App, welche nicht mit *SSR (Server Side Rendering)* entwickelt wurde, dass Layoutdaten auch ohne Login sofort beim Client landen, da der gesamte React-Code bereits sofort geladen wird. Das stellt allerdings kein Problem dar, da für das Abrufen von (kritischen) Daten aus der API ein valider Session-Token nötig ist.

Name	Value	Domain	Path	Expire...	Size	HttpO...	Secure	SameS...	Priority
<code>sokka_token</code>	eig0uz7e7p6g5pHCrd2cRl2KYPYDNgLN	acp.so...	/	Session	43				Medium
<code>sokka_username</code>	admin	acp.so...	/	Session	19				Medium

Abbildung 8.2.: Cookies in den Chromium-Dev-Tools, welche durch das ACP gesetzt wurden

8.2. User-System

Die Verwaltung der ACP-Nutzerkonten läuft über die Konfigurationsseite des ACPs. Hier können neue Nutzerkonten unter **Create a new ACP user** erstellt werden oder nicht mehr benötigte Nutzerkonten in **Manage ACP users** gelöscht werden. Das Passwort für ein ACP-Nutzerkonto muss mindestens *5 Zeichen lang* sein.

The screenshot displays two panels of the Sokka-ACP User Management interface. The top panel, titled 'Create a new ACP user', contains fields for 'Username' and 'Password' with a 'Create' button. The bottom panel, titled 'Manage ACP users', lists existing users with columns for '#', 'Username', and 'Delete'. The users listed are admin, Nicolaus, and test1, each with a red delete icon.

#	Username	Delete
0	admin	
1	Nicolaus	
2	test1	

Abbildung 8.3.: Die Nutzerverwaltung im Sokka-ACP

8.3. Routing

Da die React-App für das Sokka-ACP aus mehreren Unterseiten besteht, wird **Routing** benötigt. Dieses ist dafür verantwortlich, dass URLs wie `acp.sokka.me/products` auch auf die richtige Seite, wie in diesem Fall auf die Produktübersicht, verweisen.

Die Implementation des Routings im ACP geschieht durch die Library `react-router-dom`. Sie prüft beim Zugriff auf die Website, auf welche Route zugegriffen wurde und lädt diese.

8. Allgemeines zum ACP

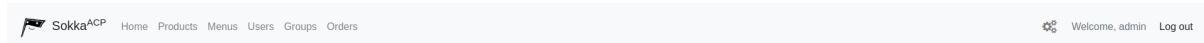
```

1  if (loggedIn) {
2    logIn();
3    return (
4      <Router>
5        <Navbar />
6        <Switch>
7          <Route path="/" exact component={HomePage} />
8          <Route path="/config" exact component={ConfigPage} />
9          <Route path="/products" exact component={ProductsPage} />
10         <Route path="/products/add" exact component={AddProductPage} />
11         <Route path="/products/:id" exact component={EditProductPage} />
12         <Route path="/menus" exact component={MenusPage} />
13         <Route path="/menus/add" exact component={AddMenuPage} />
14         <Route path="/menus/:id" exact component={EditMenuPage} />
15         <Route path="/groups" exact component={GroupsPage} />
16         <Route path="/users" exact component={UsersPage} />
17         <Route path="/orders" exact component={OrdersPage} />
18         <Route path="**" component={FourZeroFour} />
19       </Switch>
20     </Router>
21   );
22 } else {
23   return (
24     <Router>
25       <Route path="/" exact component={LoginPage} />
26       <Route path="**" component={() => <Redirect to="/" />} />
27     </Router>
28   );
29 }

```

Code-Snippet 8.1.: React-Router-Implementation des Sokka-ACPs

Die App prüft zuerst, wie in *Anmeldung* erwähnt, die Existenz und Gültigkeit von `sokka_token` und `sokka_username` Cookies. Wenn die Anmeldung gültig ist, lädt der Router normal alle konfigurierten Unterseiten oder, wenn die angeforderte URL nicht gefunden werden kann, die **404-Seite**. Falls die Anmeldung ungültig ist, wird der Nutzer auf die Anmeldungsseite geleitet.



404

Seems like you have reached a dead end. Sorry!

[Back to home](#)

Abbildung 8.4.: Aufrufen einer nicht existierenden URL im Sokka-ACP

Besonders sind die Routes `/products/:id` und `/menus/:id`. Diese Routes erwarten anstelle von `:id` eine ID, in diesem Fall die ID des gewünschten Produkts bzw. Menüs, um die Bearbeitungsseite desselbigen laden zu können. Wird ein Produkt oder ein Menü aufgerufen, das nicht existiert, so erscheint ein Fehler, der besagt, dass das gewünschte Objekt nicht gefunden werden konnte.

9. Seiten

9.1. Home

Die Startseite des ACPs gibt eine allgemeine Schnellerklärung über alle Seiten des ACPs und ist die erste Seite auf der man nach der Anmeldung landet. Neben den allgemeinen Erklärungen stehen auf der Startseite Statistiken und Systeminformationen wie CPU-Auslastung, Laufzeit des Servers, RAM-Nutzung usw.

The screenshot shows the Sokka ACP Home page. At the top, there is a header with a house icon and the text "Home". Below the header, a welcome message reads: "Welcome to the Sokka ACP! The ACP allows you to control everything about your Sokka system. You can manage your system's users, create product listings, add them to menus and view your orders. Here is a quick overview on what you can do here:". Below this message, there is a bulleted list of actions:

- **Products / Menus:** Add products or menus by clicking the blue *Add product / Add menu* button on the right or edit / delete existing items by clicking the respective links in the cards.
- **Users:** View registered users or delete them from the system. You can also set user groups here to give out rebates.
- **Groups:** Manage user groups and add rebates to groups.
- **Orders:** View orders for the current day or past days.

Below the list, there is a section titled "System Facts" containing three cards:

Registered users 12	Registered products 8	Registered menus 2
------------------------	--------------------------	-----------------------

Below the "System Facts" section is another section titled "System Performance" containing six cards arranged in two rows:

CPU Usage 17.99 %	Host Platform linux	System Uptime 6029 hours, 14 minutes, 21 seconds
CPU Count 12	Node Memory Usage 472 / 64315 MB	Process Uptime 19 hours, 10 minutes, 51 seconds

Abbildung 9.1.: Die Startseite des Sokka-ACPs

Die Informationen unter **System Facts** und **System Performance** aktualisieren sich alle fünf Sekunden automatisch. Für das Auslesen der Performance-Daten wird die Library `os-utils` verwendet.

9.2. Products

Die Produktseite erlaubt es, Produkte im Sokka-System hinzuzufügen, zu bearbeiten oder zu löschen.

The screenshot shows a list of products with their names, prices, and edit/delete buttons. The products are:

- Coca-Cola, 0.3L (2,50 €)
- Small Fries (2,50 €)
- Pizza Salami (5,50 €)

Abbildung 9.2.: Die Produkteseite des Sokka-ACPs

Die Seite zum Hinzufügen von Produkten (erreichbar über den „Add product“-Button) und dem Bearbeiten von Produkten (erreichbar über den „Edit“-Button in den einzelnen Produktkarten) baut auf derselben React-Component auf, weshalb das Aussehen der beiden Seiten nahezu identisch ist.

Die verfügbaren und einstellbaren Eigenschaften für Produkte sind:

- **Name:** Der Name des Produkts
- **Price:** Der Preis des Produkts in Euro
- **Category:** Die Kategorie des Produkts
- **Hidden:** Ob das Produkt dem Nutzer angezeigt werden soll
- **Image:** Das Vorschaubild des Produkts

The screenshot shows the edit form for the Coca-Cola product. The fields are:

- Name: Coca-Cola, 0.3L
- Price: 2.5
- Category: Beverages
- Hidden: (checkbox checked)
- Image: Choose File (No file chosen), with a small preview image of a Coca-Cola bottle.

At the bottom is a "Save changes" button.

Abbildung 9.3.: Die Bearbeitungsseite von Produkten im Sokka-ACP

9.3. Menus

Die Menüseite erlaubt es, Menüs im Sokka-System hinzuzufügen, zu bearbeiten und zu löschen.

The screenshot shows a list of two menu items:

- Cheesy & Coke**: 5,00 €. Actions: Edit, Delete.
- Coke 'n' Fries**: 3,50 €. Actions: Edit, Delete.

A blue button labeled "Add menu" is visible in the top right corner.

Abbildung 9.4.: Die Menüseite des Sokka-ACPs

Genau wie bei den Produkten, baut die Seite zum Hinzufügen von Menüs (erreichbar über den „Add menu“-Button) und dem Bearbeiten von Menüs (erreichbar über den „Edit“-Button in den einzelnen Menükarten) auf derselben React-Component auf, weshalb das Aussehen der beiden Seiten nahezu identisch ist.

Die verfügbaren und einstellbaren Eigenschaften für Menüs sind:

- **Name**: Der Name des Menüs
- **Price**: Der Preis des Menüs in Euro
- **Category**: Die Kategorie des Menüs
- **Image**: Das Vorschaubild des Menüs

Ein Menü im Sokka-System besteht immer aus Produkten im Sokka-System. Jedem Produkt in einem Menü kann ein Titel zugewiesen werden, um die Rolle des Produkts im Menü anzuzeigen (beispielsweise „Starter“ für Vorspeisen oder „Main Dish“ für Hauptgänge)

The screenshot shows the edit form for the menu item "Cheesy & Coke".

Name	<input type="text" value="Cheesy & Coke"/>
Price	<input type="text" value="5"/>
Category	<input type="text" value="Small Menus"/>
Image	<input type="button" value="Choose File"/> No file chosen

Below the form are two cards showing associated products:

- Coca-Cola, 0.3L**: Beverage — 2,50 €. Actions: Edit Title, Delete.
- Cheeseburger**: Main Dish — 6,50 €. Actions: Edit Title, Delete.

At the bottom are buttons for "Add product" and "Save changes".

Abbildung 9.5.: Die Bearbeitungsseite von Menüs im Sokka-ACP

9. Seiten

Produkte in einem Menü können über den „Delete“-Button entfernt werden oder deren Titel über den „Edit Title“-Button bearbeitet werden. Um ein neues Produkt in das Menü hinzuzufügen, kann der „Add product“-Button verwendet werden.

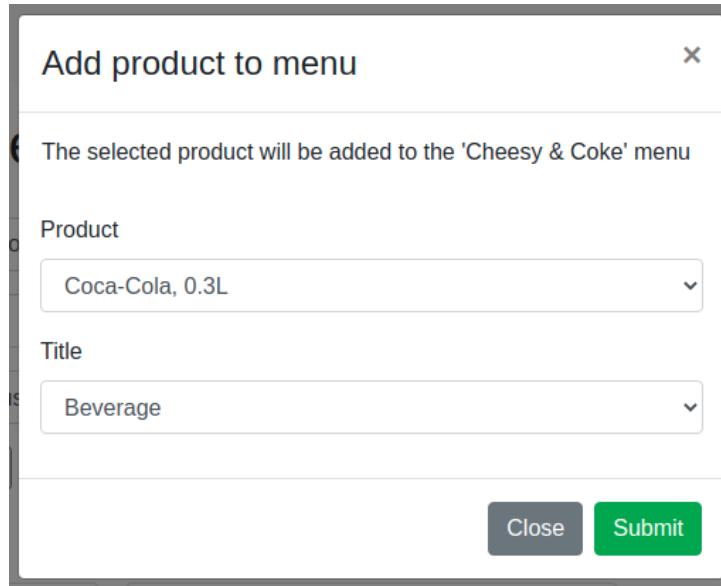
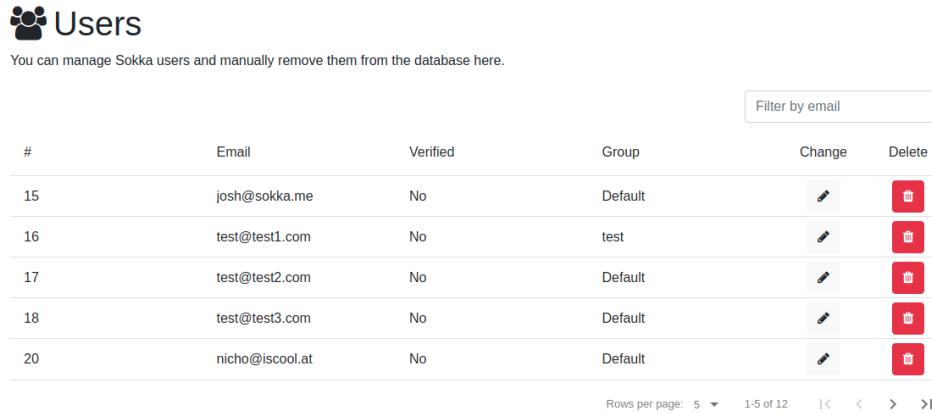


Abbildung 9.6.: Dialog zum Hinzufügen neuer Produkte in einem Menü im Sokka-ACP

9.4. Users

Die Nutzerseite erlaubt es, registrierten Nutzern im Sokka-System eine Rabattgruppe zuzuweisen oder sie komplett aus der Datenbank zu entfernen. In der Suchleiste kann nach Nutzer-E-Mail gefiltert werden.



The screenshot shows a table titled "Users" with the following data:

#	Email	Verified	Group	Change	Delete
15	josh@sokka.me	No	Default		
16	test@test1.com	No	test		
17	test@test2.com	No	Default		
18	test@test3.com	No	Default		
20	nicho@iscool.at	No	Default		

Below the table, there is a "Filter by email" input field and a pagination section indicating "Rows per page: 5" and "1-5 of 12".

Abbildung 9.7.: Die Nutzerseite des Sokka-ACPs

ist die interne und eindeutige Nutzer-ID, **E-Mail** die vom Nutzer angegebene E-Mail-Adresse, **Verified** ist der Status der Verifizierung des Nutzers (genaueres in Sektion „*Verifikation*“ im Backend-Teil), **Group** die festgelegte Rabattgruppe (standardmäßig *Default*) und Change bzw. Delete sind Aktionsspalten.

Der *Change*-Button in jeder Zeile erlaubt das Ändern der Rabattgruppe eines Nutzers. Rabattgruppen müssen zuerst im Reiter „*Groups*“ konfiguriert werden. Danach können erstellte Gruppen im Change-Dialog ausgewählt werden.

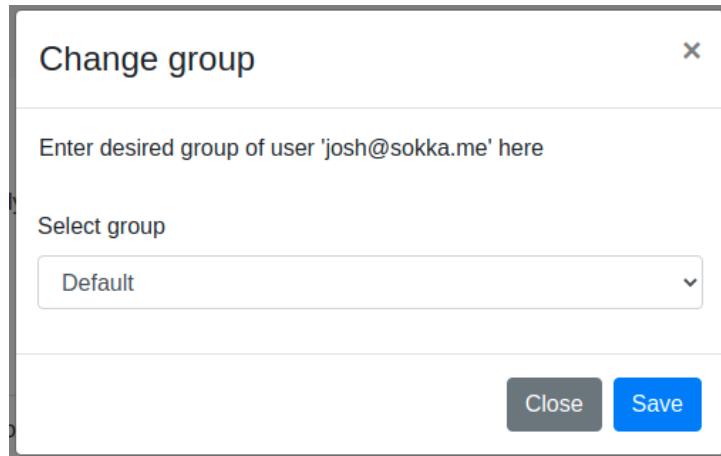


Abbildung 9.8.: Der Dialog zum Ändern von Rabattgruppen von Nutzern im Sokka-ACP

9.5. Groups

Die Gruppenseite erlaubt es, neue Rabattgruppen zu erstellen, existente zu bearbeiten und nicht mehr benötigte zu löschen. In der Suchleiste kann nach Gruppennamen gefiltert werden.

The screenshot shows a table titled "Groups" with the following data:

#	Name	Rebate	Change	Delete
1	Default	27		
11	test	25		
12	test2	24		
14	testoas	34		

At the top right is a blue "Create group" button. Below the table is a "Filter by name" input field. At the bottom are pagination controls: "Rows per page: 5", "1-4 of 4", and navigation arrows.

Abbildung 9.9.: Die Gruppenseite des Sokka-ACPs

Wenn man für eine bestimmte Gruppe von Nutzern einen allgemein gültigen Rabatt für Bestellungen von Menüs sowie Produkten hinterlegen möchte, so kann man für diese Nutzer mit einem Klick auf den „Create group“-Button eine Rabattgruppe erstellen.

Alle Rabattgruppen, außer die Default-Gruppe, sind lösbar. Der Name und der Rabatt der Gruppe kann über den „Change“-Button via Slider angepasst werden.

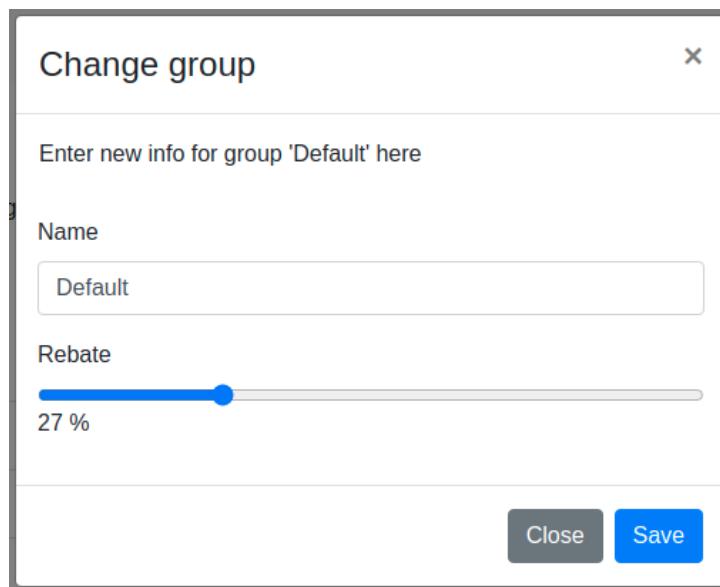


Abbildung 9.10.: Der Dialog zum Ändern des Rabatts von Rabattgruppen im Sokka-ACP

9.6. Orders

Die Orderseite erlaubt es, getätigte Bestellungen von Nutzern pro Tag einzusehen.

The screenshot shows the 'Orders' section of the Sokka-ACPs. At the top, it says 'You can view orders for specific dates here.' and has a date input field set to '03/02/2021'. Below this, there's a QR code. To its right, under 'Menu Orders:', it says 'No menu orders'. Under 'Product Orders:', it lists '1x Coca-Cola, 0.3L (2,50 €)' and '1x Chicken Nuggets (3,50 €)'. At the bottom left, it shows 'Order: #57, User ID: 26' and 'From 3/2/2021, 4:16:37 PM'. On the right, it says 'This order is VALID' and provides a breakdown: 'Subtotal: 6,00 €', 'Group Rebate: 27 %', and 'Total: 4,38 €'.

Abbildung 9.11.: Die Bestellungsseite des Sokka-ACPs

Die Order-Seite des ACPs ist rein informativ. Hier können keine Änderungen an den Bestellungen vorgenommen werden, beispielsweise können Bestellungen im ACP nicht invalidiert werden. Für das Bearbeiten von Bestellungen existiert der *Admin-Client*.

Das ACP zeigt folgende Informationen pro Bestellung an:

- Die ID der Bestellung
- Die ID des Nutzers, der die Bestellung getätigkt hat
- Den Zeitpunkt der Bestellung
- Den Inhalt der Bestellung
- Den Preis und Rabatt der Bestellung
- Die Validität der Bestellung (**VALID** bedeutet, dass die Bestellung noch nicht eingelöst wurde, **INVALIDATED** bedeutet, dass die Bestellung an der Kasse eingelöst wurde)

Der QR-Code ergibt sich aus der Bestellungs-ID und der Nutzer-ID im Format <userID>:<orderID> und ist derselbe QR-Code, der auch im Client angezeigt wird.

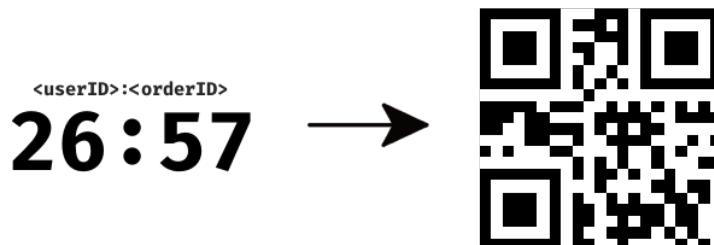


Abbildung 9.12.: QR-Code-Erstellung der Bestellungen

9.7. Konfiguration

Die Konfigurationsseite erlaubt es neben der ACP-Nutzerkontoverwaltung ebenfalls ganz grundlegende Einstellungswerte für das Sokka-System festzulegen.

The screenshot shows two main sections of the configuration interface:

- User Creation:** A form titled "Create a new ACP user" with fields for "Username" and "Password". Below it is a "Create" button.
- User Management:** A table titled "Manage ACP users" listing existing users with columns for "#", "Username", and "Delete". The users listed are admin, Nicolaus, and test1, each with a delete icon.

Abbildung 9.13.: Die Konfigurationsseite des Sokka-ACPs

Die Funktionsweise der Nutzerverwaltung wird in der Sektion *User-System* genauer erläutert.

Zum Zeitpunkt des Schreibens gibt es auf dieser Seite nur eine Einstellung. Diese ist die `closingTime`, welche jene Uhrzeit ist, zu der keine Bestellungen für den folgenden Tag zulässig sind. Ist die `closingTime` beispielsweise *9:00 PM*, so ist es Nutzern maximal bis 21:00 Uhr möglich Bestellungen zu tätigen, andernfalls erhalten sie im Client eine Fehlermeldung.

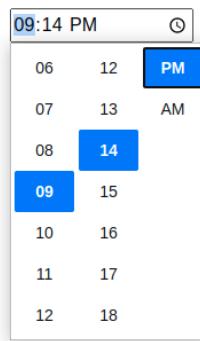


Abbildung 9.14.: Die Zeitauswahl auf der Konfigurationsseite im Sokka-ACP

Das Format der Zeit im Input-Feld passt sich dem Format des Computers an, von dem auf das ACP zugegriffen wird.

Teil V.

Client App

10. Der Sokka-Mobile-Client

10.1. Was ist der Client?

Der **Sokka-Client** bildet die Schnittstelle zwischen Nutzern und den Diensten von **Sokka**.

Nach erfolgreicher Registrierung des Accounts kann der Nutzer die verschiedenen Angebote und Waren eines Restaurants, welches auf die Dienste von Sokka zurückgreift, einsehen und nach freiem Belieben für den nächsten Tag Bestellungen platzieren.

Für jede getätigte Bestellung erhält der Nutzer einen einzigartigen QR-Code, der am Folgetag im Restaurant durch das Personal eingescanned wird, um die Bestellung zu bestätigen und diese zu kassieren.

Einfache Nutzung durch Intuitivität Dank des intuitiv gestalteten User-Interfaces und die für den Nutzer optimierte User-Experience ist der Sokka-Client für jede Person einfach und verständlich nutzbar.

Statt komplexer Verschachtelung von Buttons und Schaltflächen wird auf simples aber modernes Design gesetzt, wodurch jeder Nutzer, völlig unabhängig der Altersgruppe, schnell und einfach den Umgang mit der App erlernt und meistert.

11. Utility

11.1. Allgemeines

Die Utility-Klassen dienen als allgemeine Helfer-Klassen und bieten Funktionen, die in der gesamten App benötigt werden, beispielsweise für das Versenden von Requests an die Sokka-API, für das persistente Abspeichern von Cookies auf dem Gerät oder zum Verarbeiten bestimmter Aktionen durch den Nutzer.

11.2. Network-Wrapper

Um in einer Flutter-App Requests sowie Responses über das *HTTP*-Protokoll versenden und empfangen zu können muss das von Dart bereitgestellte *http*-Package zu den Dependencies im `pubspec.yaml`-File hinzugefügt werden.

```
1 dependencies:  
2   http: ^0.12.2
```

Code-Snippet 11.1.: Hinzufügen des **http**-Packages zum `pubspec.yaml`-File

Mithilfe dieses Packages können nun *GET*- und *POST*-Requests an eine bestimmte URL, in unserem Fall an die Sokka-API gesendet werden, wobei die erhaltene Antwort per `jsonDecode()`-Funktion in ein lesbares JSON-Format umgewandelt werden muss.

11. Utility

```
1 import 'dart:convert';
2 import 'package:http/http.dart' as http;
3
4 void main (List<String> arguments) async {
5     final response = await http.post(
6         'https://api.sokka.me/user/login',
7         headers: {
8             ...
9         },
10        body: jsonEncode({
11            'email': 'email',
12            'password': 'password'
13        }),
14    );
15 }
```

Code-Snippet 11.2.: Simples Beispiel für einen POST-Request

Um das Versenden und Verarbeiten solcher HTTP-Requests möglichst einfach zu gestalten bietet jene Klasse zwei Wrapper-Funktionen für GET- und POST-Requests an.

Jene Funktionen erhalten die *Ziel-URL* und entsprechenden *Request-Headers* und *-Body* in Form einer *HashMap* als Argumente, welche direkt per `jsonEncode()` umgewandelt werden.

Tritt bei einem durchgeföhrten Request ein Fehler auf - ein Statuscode der Antwort unter 200 bzw. über 400 oder ein leeres JSON-Objekt - wirft die Funktion eine Exception mit entsprechendem HTTP-Code.

Erfolgt der Request fehlerlos, so wird dessen Body per `jsonDecode()` umgewandelt und returniert.

11.2.0.1. NetworkWrapper.GET

```
1 Future<dynamic> get(final String url, { final Map<String, String> headers }) async {
2     final response = await http.get(
3         url,
4         headers: headers,
5     );
6
7     if (response.statusCode < 200 || response.statusCode > 400 || json == null) {
8         throw new Exception("Error fetching data from $url with ${response.statusCode}");
9     }
10    return this._jsonDecoder.convert(response.body);
11 }
```

Code-Snippet 11.3.: GET-Request-Wrapper der NetworkWrapper-Klasse

11.2.0.2. NetworkWrapper.POST

```

1 Future<dynamic> post(final String url, { final Map<String, String> headers, body, encoding }) async {
2     final response = await http.post(
3         url,
4         headers: headers,
5         body: this._jsonEncoder.convert(body),
6         encoding: encoding
7     );
8
9     if (response.statusCode < 200 || response.statusCode > 400 || json == null) {
10        throw new Exception("Error fetching data from $url");
11    }
12    return this._jsonDecoder.convert(response.body);
13 }
```

Code-Snippet 11.4.: POST-Request-Wrapper der NetworkWrapper-Klasse

11.3. Cookie-Storage

Damit sich ein User des Clients nicht nach jedem App-Neustart erneut mit seinen Logindaten anmelden muss, werden dessen E-Mail-Adresse sowie der generierte Session-Token, der nach einem erfolgreichem Login erhalten wird, in Form von Cookies im externen Speichers des Geräts abgespeichert.

Damit dies plattformunabhängig funktioniert, wird das `shared-preferences`-Package von Flutter verwendet. Dieses Plugin wrapped die `NSUserDefaults` von iOS und die `SharedPreferences` von Android, wodurch Key-Value-Paare persistent unabhängig vom Betriebssystem abgespeichert werden können.

Damit die Daten, die im Cookie-Storage gespeichert sind, global in der App verfügbar sind, wird in jener Klasse das *Singleton-Design-Pattern* angewendet. Dieses ermöglicht, dass immer dieselbe, einzige Instanz dieser Klasse aufgerufen wird, anstatt immer neue Objekte per Konstruktor anzulegen.

Dart hat hierfür eine eigene syntaktische Lösung:

```

1 class CookieStorage {
2     // Eigentliche Instanz des Cookie-Storages
3     static CookieStorage _instance = new CookieStorage.internal();
4
5     // Singleton-Konstruktor, der _instance statt einer neuen Objektinstanz
6     // zu retournieren
7     factory CookieStorage() => _instance;
8
9     // Klasseninterner Konstruktor
10    CookieStorage.internal();
11 }
```

Code-Snippet 11.5.: Dart's syntaktische Lösung für die Erstellung eines Singletons

Innerhalb der Klasse besitzt der Cookie-Storage zusätzlich eine Map `Map<String, String>`, in

11. Utility

der die Key-Value-Paare nochmals abgespeichert werden, damit sie in dafür notwendigen Situationen synchron abgefragt werden können.

Das Speichern und Abfragen von Werten aus dem externen Speicher des Geräts erfolgt über eine Instanz der *SharedPreferences*:

```
1 import 'package:shared_preferences/shared_preferences.dart';
2
3 ...
4
5 // Retourniert den String, der mit entsprechendem Key gespeichert wurde
6 Future<String> getString(final String key) async {
7     return await SharedPreferences.getInstance().getString(key);
8 }
9
10 // Speichert den Argument-String mit entsprechendem Key ab
11 Future<void> storeString(final String key, final String value) async {
12     await SharedPreferences.getInstance().setString(key, value);
13 }
```

Code-Snippet 11.6.: Speichern von Key-Value-Pairs in den externen Gerätespeichern

Damit die gespeicherten Daten auch in der App persistent abgerufen werden können, gibt es klassenintern im Cookie-Storage vorgefertigte, statische Keys sowohl für den Session-Token als auch für die User-E-Mail-Adresse:

```
1 static const TOKEN_KEY = 'token';
2 static const EMAIL_KEY = 'email';
```

Code-Snippet 11.7.: Definieren von statischen Key-Strings für erhöhte Typsicherheit

So können Fehler, die durch Tippfehler oder falsche Angaben des Keys beim Abspeichern bzw. Abfragen von Werten entstehen, zur Gänze verhindert werden.

Anstatt den Key also manuell anzugeben, kann dieser nun simpel über die statische Klassenvariable aufgerufen werden.

11.4. Controllers

Mithilfe der Controller-Klassen können benötigte Werte oder Informationen eines bestimmten Views global in der App gespeichert, dynamisch geändert und angezeigt werden, beispielsweise

```

1 Future<String> getEmail() async {
2     return await SharedPreferences.getInstance()
3         .getString(CookieStorage.EMAIL_KEY);
4 }
5
6 Future<void> storeEmail(final String email) async {
7     await SharedPreferences.getInstance()
8         .setString(CookieStorage.EMAIL_KEY, email);
9 }
```

Code-Snippet 11.8.: Verwalten von Cookie-Daten mithilfe oben definierter Static-Keys

die Produkte, die sich im Warenkorb des Nutzers befinden.

11.4.1. Models für Controller

Menu Im Menu-Model werden alle relevanten Daten zu einem Menü abgespeichert.

Integer	_menuID	ID des Menüs
String	_name	Name des Menüs
List<MenuEntry>	_entries	die Liste mit allen Menü-Einträgen
Double	_price	Preis des Menüs
NetworkImage	_image	Bild des Menüs
Boolean	_isHidden	Status, ob das Menü im View angezeigt werden soll

MenuEntry Verknüpft alle Produkte mit dem Menü, in dem sie enthalten sind.

Integer	_titleID	Name des Produkts im Menü
Integer	_menuID	ID des Menüs
Product	_product	das im Menü enthaltene Produkt

Product Im Product-Model werden alle relevanten Daten zu einem Produkt abgespeichert.

Integer	_productID	ID des Produkts
String	_name	Name des Produkts
Double	_price	Preis des Produkts
NetworkImage	_image	Bild des Produkts
Boolean	_isHidden	Status, ob das Produkt im View angezeigt werden soll

11. Utility

Order Im Order-Model werden alle relevanten Daten zu einer Bestellung abgespeichert.

Integer	_orderID	ID der Bestellung
Integer	_userID	ID des Nutzers, der die Bestellung getätigt hat
String	_timestamp	Zeitpunkt, an dem die Bestellung erstellt worden ist
Integer	_rebate	Rabatt, der vom Gesamtpreis der Bestellung abgezogen wird
Double	_subTotal	Preis der Bestellung, ohne den Rabatt zu berücksichtigen
Boolean	_total	Preis der Bestellung mit abgezogenem Rabatt
Map<String, int>	_menuOrders	Map mit allen bestellten Menüs und ihrer Anzahl
Map<String, int>	_productOrders	Map mit allen bestellten Produkten und ihrer Anzahl
String	_qrData	String, aus dem der QR-Code der Bestellung generiert wird

11.4.2. Menu-Controller

Der Menu-Controller speichert alle zum aktuellen Zeitpunkt verfügbaren Menüs in einer internen Liste, die zu App-Start per API-Call befüllt wird, ab.

Ebenso wie der *CookieStorage* ist auch der Menu-Controller - sowie alle weiteren Controller - als Singleton implementiert.

11.4.3. Product-Controller

Im Product-Controller werden alle zum Kauf verfügbaren Produkte abgespeichert. Auch dieser wird bei App-Start durch einen Aufruf der API initialisiert.

11.4.4. ShoppingBasket-Controller

In der Liste des ShoppingBasket-Controllers werden all jene Menüs und Produkte zwischengespeichert, die vom Nutzer zu seinem Warenkorb hinzugefügt worden sind.

Im Gegensatz zu den anderen Controllern ist dieser beim Start der App leer und wird erst durch die Interaktion des Users befüllt.

11.4.5. Order-Controller

Der Order-Controller verwaltet alle vom Nutzer getätigten Bestellungen und wird genauso wie der Menu- und Product-Controller zum Startzeitpunkt der App über die Sokka-API befüllt.

12. App-Services

12.1. Allgemeines

Die verschiedenen Services der App sind dafür zuständig, jegliche Requests an die Sokka-API durchzuführen und, sofern diese erfolgreich sind, die entsprechenden Responses zu verarbeiten und in der App zu verwalten.

Für die folgenden Anwendungsfälle wird jeweils ein entsprechender Service benötigt:

- Registrierung neuer bzw. Anmeldung vorhandener User
- Generieren von *Bearer*-Token, die als Authorisierung für Menü- und Produkt-Requests benötigt werden
- Abfragen der verfügbaren Menüs und Produkte
- Erstellen und Verwalten der Bestellungen eines Nutzers

12.2. User-Authentication

12.2.1. Registrieren eines neuen Nutzers

Für die Registrierung eines neuen Nutzers müssen seine E-Mail-Adresse inklusive seines gewählten Passwords sowie seine Zustimmung zur Datenschutzerklärung sowie den Nutzungsbedingungen an die API gesendet werden.

Jene Daten werden vom Nutzer über die Inputs im Signup-Screen angegeben.

12.2.1.1. API-Routes

Die benötigten API-Routen werden als statische Konstanten deklariert.

```
1 static const String LOGIN_ROUTE = 'https://api.sokke.me/user/login';
2 static const String LOGOUT_ROUTE = 'https://api.sokke.me/user/logout';
3 static const String SIGNUP_ROUTE = 'https://api.sokke.me/user/create';
4 static const String VALIDATE_ROUTE = 'https://api.sokke.me/user/validate';
```

Code-Snippet 12.1.: Benötigte /user-Routes der Sokka-API

12. App-Services

```
1 Future<String> signUpUser(final String email, final String password) async {
2     final response = await this._networkWrapper.post(
3         SIGNUP_ROUTE,
4         headers: {
5             'Content-Type': 'application/json',
6         },
7         body: {
8             'email': email,
9             'password': password,
10            'tos': true,
11            'privacypolicy': true,
12        }
13    );
14    return response['token'];
15 }
```

Code-Snippet 12.2.: Funktion zum Registrieren eines neuen Nutzers

Ist der Request erfolgreich, wird ein neuer Nutzer in der Datenbank erzeugt. Als Antwort wird der Token für die damit generierte Nutzer-Session übermittelt, welcher in weiterer Folge im Cookie-Storage abgespeichert wird.

12.2.2. Anmelden eines vorhandenen Nutzers

Zur Anmeldung eines bestehenden Nutzers in der App müssen dessen Logindaten an die /user/login-Route der Sokka-API gesendet werden.

Gelingt der Request, wird als Antwort der Token für die neu erzeugte Nutzer-Session gesendet.

```
1 Future<String> loginUser(final String email, final String password) async {
2     final response = await this._networkWrapper.post(
3         LOGIN_ROUTE,
4         headers: {
5             'Content-Type': 'application/json',
6         },
7         body: {
8             'email': email,
9             'password': password,
10        }
11    );
12    return response['token'];
13 }
```

Code-Snippet 12.3.: Funktion zum Anmelden eines bestehenden Nutzers

12.2.3. Abmelden eines Nutzers

Wenn sich ein Nutzers manuell in der App abmeldet, wird ein Request an /user/logout mit dem gespeicherten Session-Token und der E-Mail des Nutzers gesendet.

12. App-Services

Jener Session-Token wird in weiterer Folge ungültig gemacht und aus dem Cookie-Storage der App gelöscht, wodurch der Nutzer beim nächsten App-Start wieder zum Login-Screen weitergeleitet wird.

```
1 Future<void> logoutUser(final String sessionToken, final String email) async {
2     await this._networkWrapper.post(
3         LOGOUT_ROUTE,
4         headers: {
5             'Content-Type': 'application/json',
6         },
7         body: {
8             'token': sessionToken,
9             'email': email
10        },
11    );
12 }
```

Code-Snippet 12.4.: Funktion zum Abmelden eines Nutzers in der App

12.2.4. Validieren einer Nutzer-Session

Damit sich ein Nutzer der Sokka-App nicht bei jedem Start der App mit seinen Logindaten neu anmelden muss, werden die E-Mail-Adresse und der Session-Token nach einer erfolgreichen Anmeldung im Cookie-Storage abgespeichert.

Mit jenen Daten im externen Gerätespeicher kann nun bei jedem weiteren App-Start ein Request an die /user/validate-Route mit Session-Token und E-Mail zur Validierung der Nutzer-Session gesendet werden.

Wenn der Session-Token nachwievor gültig ist, wird der Nutzer automatisch zum Home-Screen weitergeleitet. Ist die Session abgelaufen, erscheint der Login- bzw. Signup-Screen.

```
1 Future<bool> validateSessionToken(final String sessionToken, final String email) async {
2     final response = await this._networkWrapper.post(
3         VALIDATE_ROUTE,
4         headers: {
5             'Content-Type': 'application/json',
6         },
7         body: {
8             'token': sessionToken,
9             'email': email
10        },
11    );
12
13    return response['success'];
14 }
```

Code-Snippet 12.5.: Funktion zur Validierung eines gespeicherten Nutzer-Session-Tokens

12. App-Services

12.2.4.1. Wrapper-Funktion zur Session-Validierung

Damit der Check der Validität des gespeicherten Session-Tokens einfach bei Start der App ausgeführt werden kann, gibt es folgende Wrapper-Funktion, die automatisch benötigte Werte aus dem Storage nimmt und einen entsprechenden Request an die Sokka-API sendet.

```
1 Future<bool> validateSession() async {
2     String email = await this._cookieStorage.getEmail();
3     String token = await this._cookieStorage.getSessionToken();
4
5     return await this.validateSessionToken(token, email);
6 }
```

Code-Snippet 12.6.: Wrapper-Funktion für die einfache Validierung eines Session-Tokens

12.3. Bearer-Authentication

12.3.1. Allgemeines

Um die aktuell verfügbaren Menüs und Produkte sowie die Bestellungen eines Nutzers per API-Request abfragen zu können, wird ein *Bearer-Authentication-Token* benötigt.

12.3.2. Generieren eines Bearer-Tokens

Dieser setzt sich aus der E-Mail-Adresse des Nutzers und dem Token der aktuellen Nutzer-Session zusammen.

```
1 import 'dart:convert';
2
3 String createBearerAuthToken(final String email, final String token) {
4     return 'Bearer ${base64Encode(utf8.encode('$email:$token'))}';
5 }
```

Code-Snippet 12.7.: Generieren eines Bearer-Authrizaton-Tokens

12. App-Services

Rufen wir testweise oben angeführte Funktion auf,

```
1 String testEmail = 'sokka@testuser.me';
2 String testToken = 'Ee4IHqoSXYMsu24VDVL/SHhgBHKc31K';
3
4 print(createBearerAuthToken(testEmail, testToken));
```

Code-Snippet 12.8.: Erstellen eines Test-Bearer-Tokens

erhalten wir folgenden Authorization-Token:

```
>>> 'Bearer c29ra2FAdGVzdHVzZXIubWU6RWU0SUhpcW9TWH1Nc3UyNFZEVkwvU0hoZ0JIS2MzMUs='
```

Code-Snippet 12.9.: Test-Bearer-Authorization-Token

Ein solcher Token wird als `Authorization`-Parameter bei Requests des `FetchOrderables`- oder des `ManageOrders`-Service benötigt.

12.4. FetchOrderables-Service

12.4.1. Allgemeines

Damit der Nutzer alle angebotenen Menüs und Produkte in der App sehen und in weiterer Folge auch bestellen kann, müssen diese bei App-Start per API-Requests geladen werden. Hierfür gibt es nachfolgende Service-Funktionen.

12.4.1.1. API-Routes

```
1 static const MENU_ROUTE = 'https://api.sokka.me/menu/get';
2 static const PRODUCT_ROUTE = 'https://api.sokka.me/product/get';
3 static const IMAGE_ROUTE = 'https://api.sokka.me/image';
```

Code-Snippet 12.10.: Benötigte `/menu-`, `/product-`, und `/image`-Routes der Sokka-API

12. App-Services

12.4.2. Laden von verfügbaren Menüs

Die API übermittelt alle Menüs per JSON-Format mit nur einem einzigen gesendeten Request, wodurch durch eine einfache Iteration alle Menüs schnell in die App geladen werden können.

```
1 Future<void> appendMenus() async {
2     final auth = this._bearerAuth.createBearerAuthToken();
3     final response = await this._networkWrapper
4         .get(
5             '$MENU_ROUTE',
6             headers: {
7                 'Authorization': auth
8             },
9         );
10    final data = response['data'];
11
12    for (var i = 0; i < data.length; i++) {
13        var menu = data[i];
14        List<MenuEntry> entries = new List<MenuEntry>();
15
16        for (var entry in data[i]['entries']) {
17            entries.add(new MenuEntry(titleID: entry['title_id'], menuID: entry['menu_id'],
18                product: await this.getProduct(entry['product_id'])));
19        }
20
21        this._menuController.appendToMenus(new Menu(menuID: menu['id'], name: menu['name'], entries: entries,
22            image: await this.getImage(menu['image_id']), price: menu['price'].toDouble(),
23            isHidden: menu['hidden'] == 1));
24    }
25 }
```

Code-Snippet 12.11.: Abrufen und Speichern der verfügbaren Menüs

Aus den erhaltenen Response-Daten werden direkt neue Objekte vom `Menu`-Model erzeugt, dem `Menu`-Controller hinzugefügt und im `Menu`-View angezeigt.

12.4.3. Laden von verfügbaren Produkten

Das Abrufen der angebotenen Produkte erfolgt nach demselben Prinzip wie die Abfrage der Menüs. Per API-Call werden alle Produkte übermittelt und können durch Iterieren über das JSON-Objekt in die App geladen werden.

```
1 Future<void> appendProducts() async {
2     final response = await this._networkWrapper
3         .get(
4             '$PRODUCT_ROUTE',
5             headers: {
6                 'Authorization': this._bearerAuth.createBearerAuthToken(),
7             },
8         );
9     final data = response['data'];
10
11    for (var product in data) {
12        this._productController.appendToProducts(new Product(productID: product['id'], name: product['name'],
13            price: product['price'].toDouble(),
14            image: await this.getImage(product['image_id']), isHidden: product['hidden'] == 1));
15    }
16 }
```

Code-Snippet 12.12.: Abrufen und Speichern der verfügbaren Produkte

12.4.3.1. Wrapper-Funktionen für das Initialisieren der Menüs und Produkte

Wie beim User-Authentication-Service gibt es auch für die Initialisierung aller Waren eine Wrapper-Funktion, die beim Start der App aufgerufen wird.

```
1 Future<void> initializeMenus() async{  
2     await this.appendMenus();  
3 }  
4  
5 Future<void> initializeProducts() async {  
6     await this.appendProducts();  
7 }
```

Code-Snippet 12.13.: Abrufen und Speichern der verfügbaren Menüs

12.5. ManageOrders-Service

12.5.1. Allgemeines

Der Manage-Orders-Service bietet die Funktionalitäten, die benötigt werden, damit der Nutzer in der App neue Bestellungen erzeugen und vergangene für die Zahlungsabwicklung vor Ort abfragen und deren QR-Code vorzeigen kann.

12.5.1.1. API-Routes

```
1 static const ORDER_CREATE = 'https://api.sokka.me/order/create';  
2 static const ORDER_GET = 'https://api.sokka.me/order/get';
```

Code-Snippet 12.14.: Benötigte Routes der Sokka-API zur Verwaltung von Nutzer-Bestellungen

12.5.2. Generieren einer neuen Bestellung

Sobald ein Nutzer die in seinem Warenkorb befindlichen Waren bestellt, wird ein API-Request abgesendet, der in der Datenbank einen neuen Bestell-Eintrag erzeugt und abrufbar macht.

12. App-Services

Hierfür müssen alle Menüs und Produkte, die der Nutzer bestellen möchte, als Payload an den Request übergeben werden.

Diese werden per `loadMenus()`- sowie `loadProducts()`-Funktion in den Body des Requests geschrieben.

```
1 Future<List<Map<String, int>>> loadMenus() async {
2     final List<Menu> menus = this._basketController.getAllMenus();
3     final Map<int, int> menuIDToQuantity = new Map<int, int>();
4     List<Map<String, int>> payload = new List<Map<String, int>>.filled(menus.length, new Map());
5
6     menus.forEach((menu) => menuIDToQuantity[menu.getID] = !menuIDToQuantity.containsKey(menu.getID)
7         ? 1
8         : menuIDToQuantity[menu.getID] + 1);
9
10    final List<int> menuIDs = menuIDToQuantity.keys.toList();
11    final List<int> quantities = menuIDToQuantity.values.toList();
12
13    for (var i = 0; i < menuIDs.length; i++) {
14        payload[i] = { 'menu_id': menuIDs[i], 'quantity': quantities[i] };
15    }
16    payload = payload.where((menu) => menu['menu_id'] != null).toList();
17
18    return payload;
19 }
```

Code-Snippet 12.15.: `loadMenus()`-Funktion zum Laden bestellter Menüs in den Request-Body

```
1 Future<List<Map<String, int>>> loadProducts() async {
2     final List<Product> products = this._basketController.getAllProducts();
3     final Map<int, int> productIDtoQuantity = new Map<int, int>();
4     List<Map<String, int>> payload = new List<Map<String, int>>.filled(products.length, new Map());
5
6     products.forEach((product) => productIDtoQuantity[product.getID] = !productIDtoQuantity.containsKey(product.getID)
7         ? 1
8         : productIDtoQuantity[product.getID] + 1);
9
10    final List<int> productIDs = productIDtoQuantity.keys.toList();
11    final List<int> quantities = productIDtoQuantity.values.toList();
12
13    for (var i = 0; i < productIDs.length; i++) {
14        payload[i] = { 'product_id': productIDs[i], 'quantity': quantities[i] };
15    }
16    payload = payload.where((product) => product['product_id'] != null).toList();
17
18    return payload;
19 }
```

Code-Snippet 12.16.: `loadProducts()`-Funktion zum Laden bestellter Produkte in den Request-Body

12. App-Services

Jene Funktionen können nun in der `createOrder()`-Funktion aufgerufen werden.

```
1 Future<dynamic> createOrder() async {
2     final response = await this._networkWrapper.post(
3         ORDER_CREATE,
4         headers: {
5             'Content-Type': 'application/json',
6             'Authorization': this._bearerAuth.createBearerAuthToken(),
7         },
8         body: {
9             'products': await this.loadProducts(),
10            'menus': await this.loadMenus(),
11        },
12    );
13
14    return response;
15 }
```

Code-Snippet 12.17.: `loadProducts()`-Funktion zum Laden bestellter Produkte in den Request-Body

Ist der Request erfolgreich, wird eine neue Bestellung in der Datenbank erzeugt und im `order-view` dargestellt, damit der QR-Code der eben aufgegebenen Bestellung direkt zum Abruf bereit steht.

12.5.3. Abfragen der Bestellungen eines Nutzers

Damit ein Nutzer Einblick über seine vergangenen Bestellungen hat und diese zum Scan bei der Zahlungsabwicklung vorzeigen kann, werden alle getätigten Bestellungen des Nutzers bei App-Start der Liste des `Order-Controllers` übergeben, um im `order-View` angezeigt werden zu können.

Dies geschieht mithilfe der `appendOrders()`-Funktion des Services.

12. App-Services

```
1 Future<void> appendOrders() async {
2     final response = await this._networkWrapper.get(
3         ORDER_GET,
4         headers: {
5             'Authorization': this._bearerAuth.createBearerAuthToken(),
6         },
7     );
8     final data = response['orders'];
9
10    for (var order in data) {
11        final Map<String, int> menuOrders = new Map<String, int>();
12        final Map<String, int> productOrders = new Map<String, int>();
13
14        for (var menu in order['menuOrders']) {
15            menuOrders[menu['menu']['name']] = menu['quantity'];
16        }
17
18        for (var product in order['productOrders']) {
19            productOrders[product['product']['name']] = product['quantity'];
20        }
21
22
23        this._orderController.appendToOrders(order: new Order(orderID: order['id'],
24            userID: order['user_id'], rebate: order['rebate'], subTotal: order['total'].toDouble(),
25            timestamp: this._dateFormatter.format(DateTime.parse(order['timestamp']))),
26            menuOrders: menuOrders, productOrders: productOrders));
27    }
28 }
```

Code-Snippet 12.18.: Funktion zum Abfragen aller vom Nutzer getätigten Bestellungen

13. Screens

13.1. Allgemeines

Die verschiedenen Screens der App bilden mit den auf ihnen angezeigten Views die Hauptinteraktionsfläche zwischen der App und dem Nutzer.

13.2. Authentication-Screens

Für die Registrierung eines neuen Nutzers, das Anmelden eines vorhandenen Nutzers und für den Zeitraum, in dem ein vorhandener Session-Token validiert wird, sowie für die Dienste nach einer erfolgreichen Anmeldung steht in der App jeweils ein eigener Screen bereit.

13.2.1. LoginScreen

Der `Login-Screen` dient als - wie der Name schon verrät - Anmelde-Maske der App, über welche sich bereits bestehende Nutzer anmelden können, um auf die Dienste von Sokka zuzugreifen und Bestellungen aufzugeben zu können.

Wie jedes Widget, mit dem der Nutzer interagieren soll, ist auch der `Login-Screen` ein `Stateful Widget`. Schließlich sollen hier die benötigten Anmelddaten eingegeben und per Druck eines Buttons zur Verifizierung an den Server gesendet werden.

Für die Eingabe eines vom Nutzer abhängigen Textes werden ein `TextField`-Widget und ein sogenannter `TextEditingController`, der die Eingabe des Feldes überwacht, benötigt.

Damit der Text innerhalb des Screens als Wert verfügbar und verwendbar ist, wird dieser vom entsprechenden Controller in die passende Klassenvariable geschrieben.

So können die eingegebenen Daten in weiterer Folge per Request an den Server gesendet und überprüft werden.

13. Screens

Bei erfolgreicher Anmeldung werden sowohl der Cookie-Storage als auch der Menu- und Product-Controller per Funktion initialisiert.

```
1 class LoginScreen extends StatefulWidget {
2     @override
3         _LoginScreenState createState() => _LoginScreenState();
4 }
5
6 class _LoginScreenState extends State<LoginScreen> {
7     ...
8     final TextEditingController _emailController = new TextEditingController();
9     final TextEditingController _passwordController = new TextEditingController();
10
11    String _email;
12    String _password;
13
14    @override
15    Widget build(BuildContext context) {
16        ...
17        new TextFormField(
18            controller: this._passwordController,
19            obscureText: true,
20            keyboardAppearance: Brightness.dark,
21            decoration: new InputDecoration(
22                labelText: 'PASSWORD',
23            ),
24        ),
25        ...
26    }
27
28    Future<void> _initialize() async {
29        await this._cookieStorage.initializeCache();
30        await this._fetchOrderables.initializeMenus();
31        await this._fetchOrderables.initializeProducts();
32    }
33 }
```

Code-Snippet 13.1.: Login-Screen als Stateful Widget mit TextFormField und TextEditingController für E-Mail und Passwort

13.2.2. SignUpScreen

Auf dieselbe Weise wie der Login-Screen ist auch der Signup-Screen und dessen Funktionsweise aufgebaut.

Für das Anlegen eines neuen Nutzers muss dieser seine E-Mail-Adresse inklusive seines Passworts angeben. Um zu verhindern, dass in das Passwort ungewünschte Tippfehler eingebaut werden, muss dieses zweimal eingegeben werden.

Durch Drücken des *Submit*-Buttons auf dem Screens wird unter der Voraussetzung, dass keines der Felder leer ist oder die beiden Passwörter ungleich sind, ein Request an die Sokka-API gesendet.

Ist dieser erfolgreich, wird der neue Nutzer mit gewünschten Daten in der Datenbank angelegt und automatisch in der App angemeldet.

13.2.3. LoadingSplashScreen

Der Loading-Splash-Screen dient als zeitliche Überbrückung der Ladesequenz beim Starten der App und wird solange angezeigt, bis der gespeicherte Session-Token validiert worden ist.

Wenn der gespeicherte Token gültig ist, wird der Nutzer auf die Hauptseite weitergeleitet. Andernfalls wird der Login- bzw. Signup-Screen angezeigt.

Da dieser Screen frei von Nutzer-Interaktion ist, wird dieser als Stateless Widget deklariert, als Status-Anzeige wird ein `CircularProgressIndicator`-Widget verwendet.

```

1 class LoadingSplashScreen extends StatelessWidget {
2     @override
3     Widget build(BuildContext context) {
4         ...
5         new Container(
6             alignment: Alignment.center,
7             child: new SizedBox(
8                 width: 200.0,
9                 height: 200.0,
10                child: new CircularProgressIndicator(
11                    strokeWidth: 10.0,
12                    valueColor: new AlwaysStoppedAnimation<Color>(
13                        new Color(0xFF00E5C6)
14                    ),
15                ),
16            ),
17        ),
18    ...
19 }
20 }
```

Code-Snippet 13.2.: Loading-Splash-Screen in Form eines Stateless Widgets

13. Screens

Dieser muss per MediaQuery als Alternativ-Route zu Login bzw. Signup und dem Home-Screen im App-Entry-Point im `main.dart`-File gesetzt werden.

```
1 return new MediaQuery(  
2   data: new MediaQueryData.fromWindow(UI.window),  
3   child: new Directionality(  
4     textDirection: TextDirection.ltr,  
5     child: new LoadingSplashScreen(),  
6   ),  
7 );
```

Code-Snippet 13.3.: Loading-Screen als Alternativ-Route per MediaQuery

13.3. HomeScreen

Der Home-Screen bildet die zentrale Interaktionsebene zwischen dem Nutzer und Sokka.

Er wird aus einem `DefaultTabController`-Widget aufgebaut und besteht aus vier verschiedenen Tabs, die per Swipe gewechselt werden können und die Parent-Elemente für alle Views bilden.

Für die Navigation zwischen den Views des Home-Screens wird ein `TabBar`-Widget benötigt, welches dem `bottomNavigationBar`-Property übergeben werden muss.

Mithilfe einer solchen Tab-Bar kann neben einer Beschriftung inklusive Icon der einzelnen Tabs auch ein Indicator eingebaut werden, der anzeigt, auf welchem Tab (und damit View) sich der Nutzer gerade befindet.

13. Screens

```
1 class LoginScreen extends StatefulWidget {
2     @override
3         _LoginScreenState createState() => _LoginScreenState();
4 }
5
6 class _LoginScreenState extends State<LoginScreen> {
7     ...
8     final TextEditingController _emailController = new TextEditingController();
9     final TextEditingController _passwordController = new TextEditingController();
10
11    String _email;
12    String _password;
13
14    @override
15    Widget build(BuildContext context) {
16        ...
17        new TextFormField(
18            controller: this._passwordController,
19            obscureText: true,
20            keyboardAppearance: Brightness.dark,
21            decoration: new InputDecoration(
22                labelText: 'PASSWORD',
23            ),
24        ),
25        ...
26    }
27
28    Future<void> _initialize() async {
29        await this._cookieStorage.initializeCache();
30        await this._fetchOrderables.initializeMenus();
31        await this._fetchOrderables.initializeProducts();
32    }
33 }
```

Code-Snippet 13.4.: DefaultTabController mit vier Tabs und Tab-Bar zur Navigation

14. Views

14.1. Allgemeines

Die verschiedenen Views der App, die auf dem Home-Screen angezeigt werden, stellen die eigentlichen Dienste von Sokka zur Verfügung.

Neben dem Anzeigen der verfügbaren Produkte und Menüs finden auch hier die Kaufabwicklung und Bestellung gewünschter Waren sowie die Verwaltung vergangener Bestellungen statt.

Aufgrund ihrer benötigten Interaktivität sind alle vier Views als Stateful Widget aufgebaut und bilden je nach Zweck des Views die entsprechenden Inhalte ihres Controllers ab.

14.2. MenuView

Im Menu-View werden alle verfügbaren Menüs, die im Menu-Controller gespeichert sind, in Form einer Liste bestehend aus Card-Widgets angezeigt.

14.2.1. Render-Widget für Menü-Informationen

Damit ein Nutzer schnell einen Überblick über alle verfügbaren Menüs des Anbieters erhält, werden alle relevanten Informationen eines jeweiligen Menüs, beispielsweise die inkludierten Produkte oder der Preis, in einem individualisierten Card-Widget dargestellt.

Hierfür muss dem Menu-Card-Konstruktor ein Objekt der Menü-Klasse übergeben werden. Aus diesem werden in der `build()`-Funktion des Custom-Widgets alle Daten ausgelesen und in entsprechend weiteren Widgets angezeigt.

So befinden sich folgende Inhalte eines Menüs auf dessen Menu-Card:

- der Name des Menüs als Titel des Card-Widgets
- das Bild des Menüs
- alle inkludierten Produkte, welche als Liste angeführt werden
- der Preis des Menüs

Unter eben genannten Informationen befindet sich ein Button, der auf Druck das entsprechende Menü zum Warenkorb des Nutzers hinzufügt und zum Kauf bereit macht.

14. Views

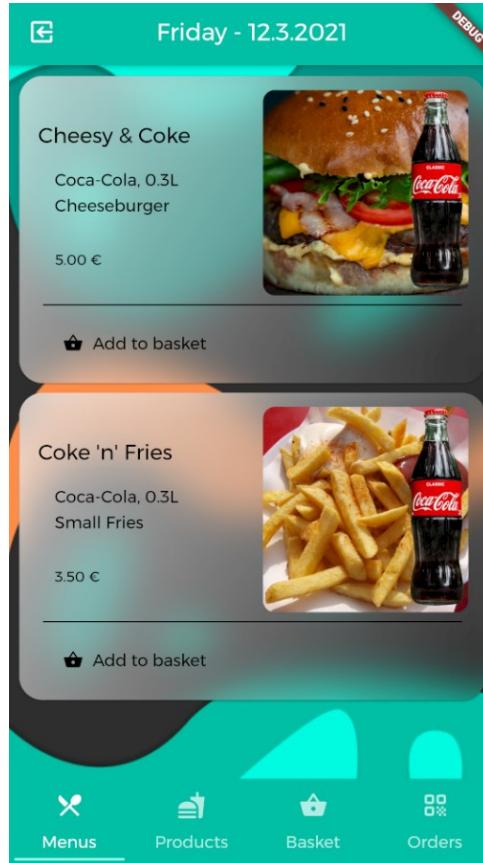


Abbildung 14.1.: Menu-View mit darauf abgebildeten Menu-Cards

14.2.2. Rendern der Menu-Cards für alle verfügbaren Menüs

Um für jedes bestellbare Menü eine solche Menu-Card im eigentlichen View anzuzeigen, wird das `ListView.builder`-Widget verwendet.

Dieses Widget erzeugt eine Liste mit einer definierten Anzahl, in unserem Fall die Anzahl an Menü-Objekten im Menu-Controller des übergebenen Child-Elements.

```
1 new ListView.builder(
2   itemCount: this._menuController.getMenus().length,
3   itemBuilder: (BuildContext context, int index)
4     => new MenuCard(this._menuController.getMenus()[index]),
5 )
```

Code-Snippet 14.1.: `ListView.builder`-Widget zum Erzeugen und Darstellen aller Menu-Cards

14.3. ProductView

Im Product-View werden alle zum Kauf erhältlichen Produkte, die bei App-Start in der Liste des Product-Controllers gespeichert werden, in Form von individualisierten Card-Widgets angezeigt.

14.3.1. Render-Widget für einzelne Produkte

Ein Produkt wird im Product-View mittels einer Kachel dargestellt. Auf dieser befindet sich das Bild des Produkts, dessen Name sowie sein Preis.

Hierfür wird erneut das Card-Widget des Material-Katalogs verwendet.

Nach bereits bekanntem Prinzip muss dem Konstruktor des Custom-Widgets ein Produkt-Objekt übergeben werden, aus welchem alle erforderlichen Daten wie

- das Bild des Produkts
- der Name des Produkts
- der Preis des Produkts

entnommen werden.

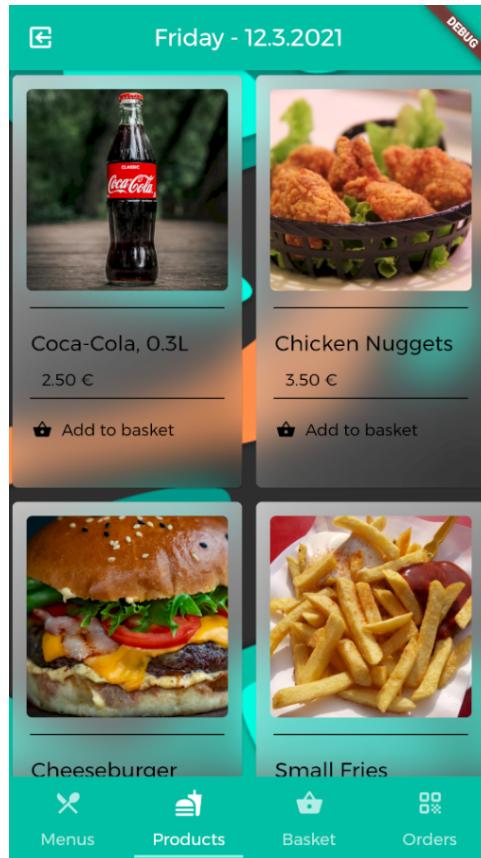


Abbildung 14.2.: Product-View mit Kachel für das jeweilige Produkt

14.3.2. Darstellen der Product-Tiles

Für das Darstellen der einzelnen Product-Tiles wird das Open-Source *flutter_staggered_grid_view*-Package verwendet. [20]

Mithilfe dieser Library lässt sich einfach ein Grid-View mit dynamischen Reihen- und Spalten-dimensionen definieren.

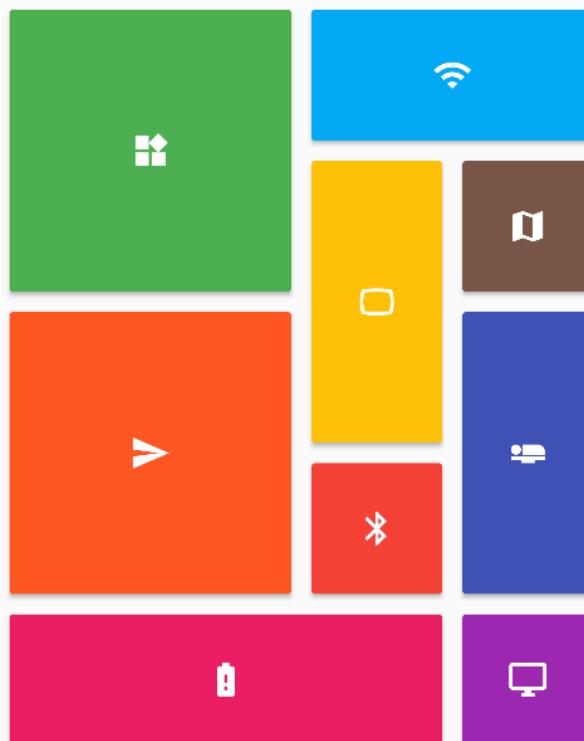


Abbildung 14.3.: Beispiel für ein Staggered-Grid-View [21]

Nun kann man im eigentlichen View für die Produktansicht mithilfe eines `StaggeredGridView.countBuilder`-Widgets die diversen Kacheln für alle vorrätigen Produkte erstellen und anzeigen.

```

1 new ListView.builder(
2   itemCount: this._menuController.getMenus().length,
3   itemBuilder: (BuildContext context, int index)
4     => new MenuCard(this._menuController.getMenus()[index]),
5 )

```

Code-Snippet 14.2.: `StaggeredGridView.countBuilder`-Widget zum Erzeugen und Darstellen der Product-Tiles

14.4. BasketView

Der Basket-View bildet alle Menüs und Produkte ab, die zuvor vom Nutzer zu dessen Warenkorb hinzugefügt worden sind.

Hierfür wird für jede Ware ein `Dismissible`-Widget verwendet. Dieses erlaubt dem Nutzer, dass ungewünschte Menüs und Produkte aus dem Warenkorb entfernt werden. [22]

14.4.1. Dismissible-Widget für Ware

Mithilfe eines Dismissible-Widets kann der Nutzer einfach seinen Warenkorb verwalten und hinzugefügte Menüs und Produkte mithilfe eines kurzen Swipes über das Widget wieder entfernen.

Damit dies funktioniert muss einerseits jedes Dismissible-Widget einen *Unique Key* erhalten, damit die Engine weiß, welches Card-Widget vom Widget-Tree entfernt werden soll.

Andererseits ist eine Implementation der `onDismissed`-Funktion erforderlich. In dieser wird festgelegt, was nach Entfernen eines Dismissibles geschehen soll.

In unserem Fall soll das bestimmte Produkt bzw. Menü vom Warenkorb und damit auch vom ShoppingBasket-Controller entfernt werden und der Gesamtpreis des Warenkorbs aktualisiert werden.

```

1 new Dismissible(
2   key: new UniqueKey(),
3   onDismissed: (DismissDirection direction) => {
4     setState(() => {
5       this._shoppingBasketController.getBasket().removeAt(index),
6       this._updateTotalPrice(),
7     })
8   }
9 );
10 ...
11 ...
12
13 void _updateTotalPrice() {
14   this._totalPrice = 0.0;
15   this._shoppingBasketController.getBasket().forEach((orderable) => {
16     this._totalPrice += orderable.getPrice
17   });
18 }
```

Code-Snippet 14.3.: Dismissible-Widget im Basket-View mit `_updateTotalPrice`-Funktion zum Aktualisieren des Gesamtpreises

14.4.2. Darstellen des Warenkorbs

Wie bereits im MenuView und ProductView verwendet, kommt auch hier wieder das `ListView.builder`-Widget zum Einsatz, um für jede Ware im Warenkorb ein entsprechendes Dismissible-Widget zu generieren und anzuzeigen.

14. Views

Der aktuelle Gesamtpreis des Warenkorbs wird mithilfe eines Textfelds auf einem `BottomAppBar`-Widget dargestellt und bei jeder Veränderung der vorübergehend gespeicherten Waren aktualisiert, wie in Snippet 14.3 zu sehen ist.

Um eine Bestellung aufzugeben, wird ein `FloatingActionButton` an das Bottom-App-Bar-Widget gebunden, welcher auf Druck ein `ModalBottomSheet`-Widget öffnet und die verschiedenen Kaufoptionen anzeigt.

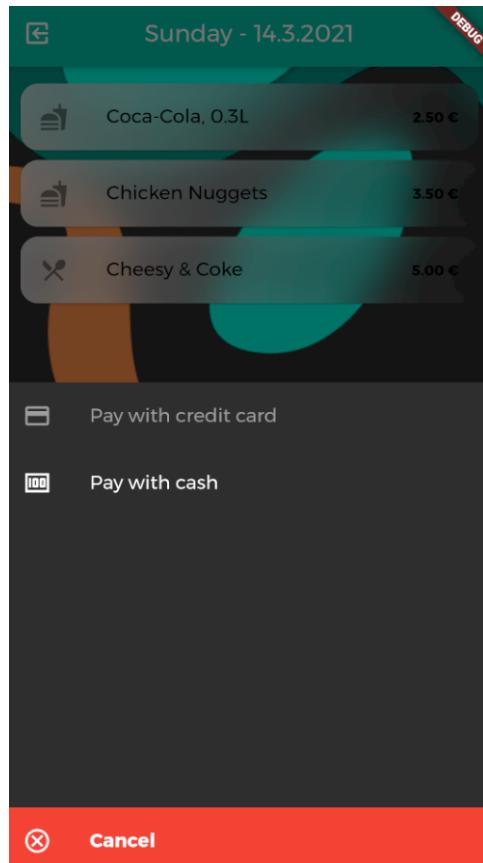


Abbildung 14.4.: Modal-Bottom-Sheet im Basket-View zum Wählen der Bezahloption

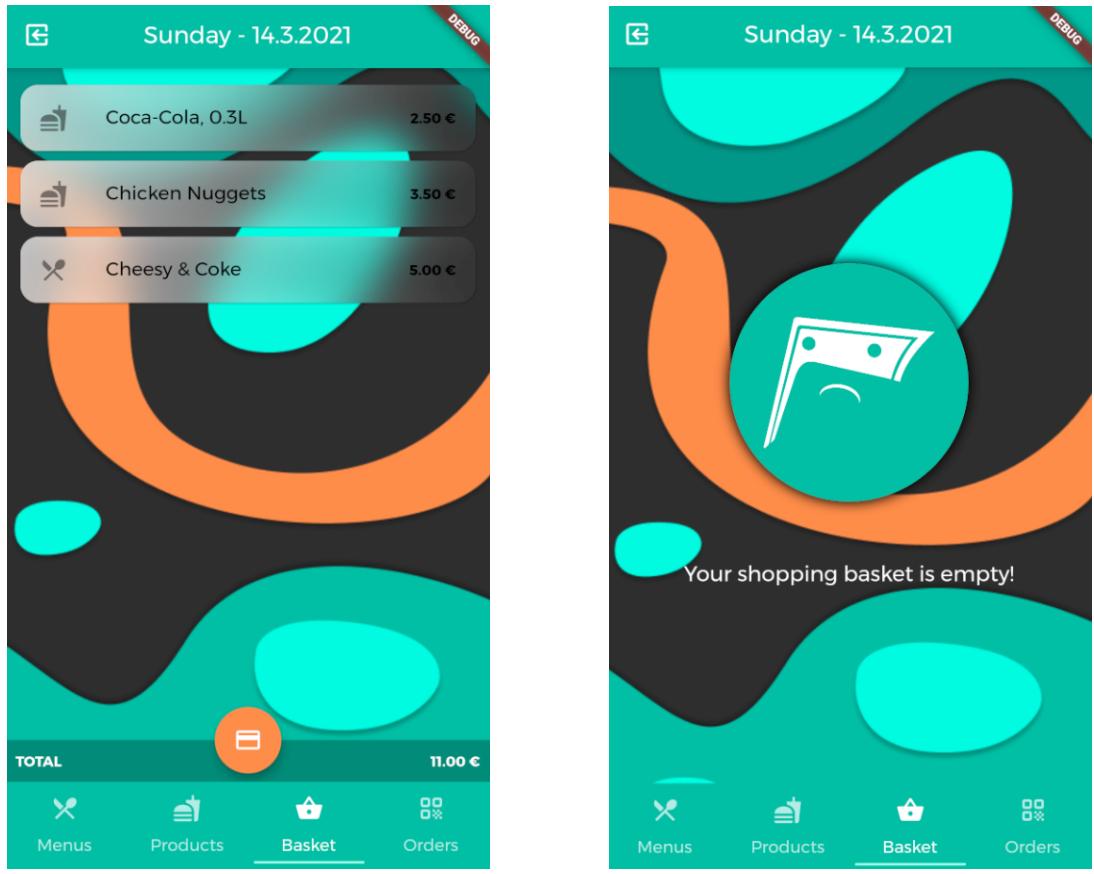
Ist der Warenkorb des Nutzers leer, so soll als Alternative zu einem leeren List-View-BUILDER das Sokka-Logo als Platzhalter angezeigt werden.

14. Views

```
1 @override
2 Widget build(BuildContext context) {
3     return this._shoppingBasketController.getBasket().isEmpty
4         ? new Scaffold(
5             // Anzeigen des Warenkorbs
6         )
7         : new Scaffold(
8             // Sokka-Logo als Alternative
9         );
10 }
```

Code-Snippet 14.4.: StaggeredGridView.countBuilder-Widget zum Erzeugen und Darstellen der Product-Tiles

Dies lässt sich mithilfe obigen Ternary-Operators in der `build()`-Funktion des Views einfach realisieren. Dieser soll abhängig vom Warenkorb zwei verschiedene Scaffolds erzeugen und führt zu folgendem Resultat:



(a) Basket-View mit verschiedenen Waren

(b) Leerer Basket-View

Abbildung 14.5.: Basket-View in befüllter und leerer Form

14.5. OrderView

Im Order-View werden alle vom Nutzer getätigten Bestellungen angezeigt und können damit jederzeit abgerufen werden.

14.5.1. Render-Widget für Bestellungen

Eine Bestellung wird, wie ein Menü, mithilfe eines individualisierten Card-Widgets dargestellt. Anstatt der bekannten Informationen des Menüs werden hier die wichtigen Daten der entsprechenden Bestellung angezeigt

Dazu gehören:

- das Datum der Bestellung
- die bestellten Waren
- der scanbare QR-Code der Bestellung
- der zu zahlende Betrag vor und nach Abzug des möglichen Rabattes

Jener QR-Code wird an der Kassa des Anbieters vom Admin-Client eingescannt, um einerseits die bestellte Waren zu überprüfen und andererseits jene Bestellung abzuschließen und den Code ungültig zu machen.



Abbildung 14.6.: Order-View mit Card-Widgets für jede Bestellung

14. Views

Damit das Scannen des QR-Codes an der Kassa vor Ort verbessert wird, kann der Code einer Bestellung durch Antippen vergrößert werden. Hierfür wird ein `ModalBottomSheet` mit weißem Hintergrund geöffnet, auf dem das QR-Image in vergrößerter Form abgebildet und damit leichter einscanbar ist.

Da ein `QrImage`-Widget allerdings kein definiertes `onTap`-Property hat, muss das `onTap`-Verhalten des QR-Codes mithilfe eines `GestureDetectors` implementiert werden.

```
1 new SizedBox(
2   child: new GestureDetector(
3     onTap: () => showModalBottomSheet(
4       context: context,
5       builder: (context)
6         => new Container(
7           child: new Center(
8             child: new QrImage(
9               ...
10            ),
11            ),
12            ),
13            ),
14            ),
15 );
```

Code-Snippet 14.5.: Implementation der `onTap`-Funktionalität des QR-Codes

14. Views

Nun kann durch kurzes Antippen der QR-Codes für den Scan vergrößert werden und sieht wie folgt in der App aus:

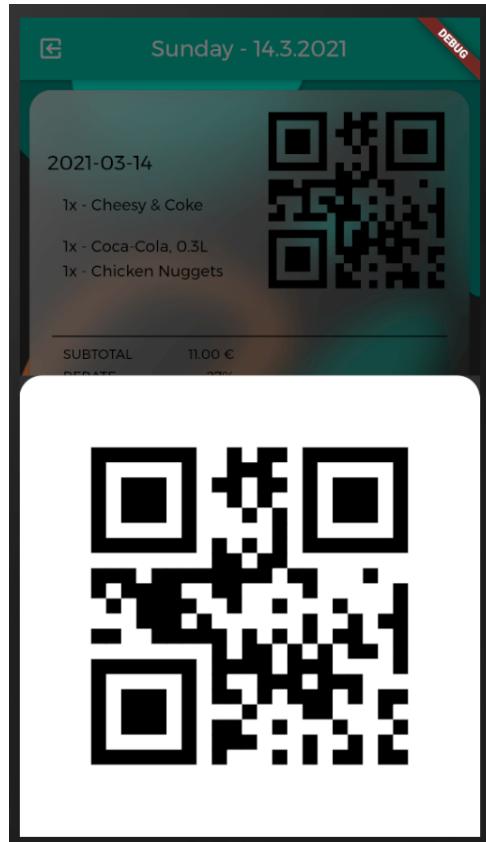


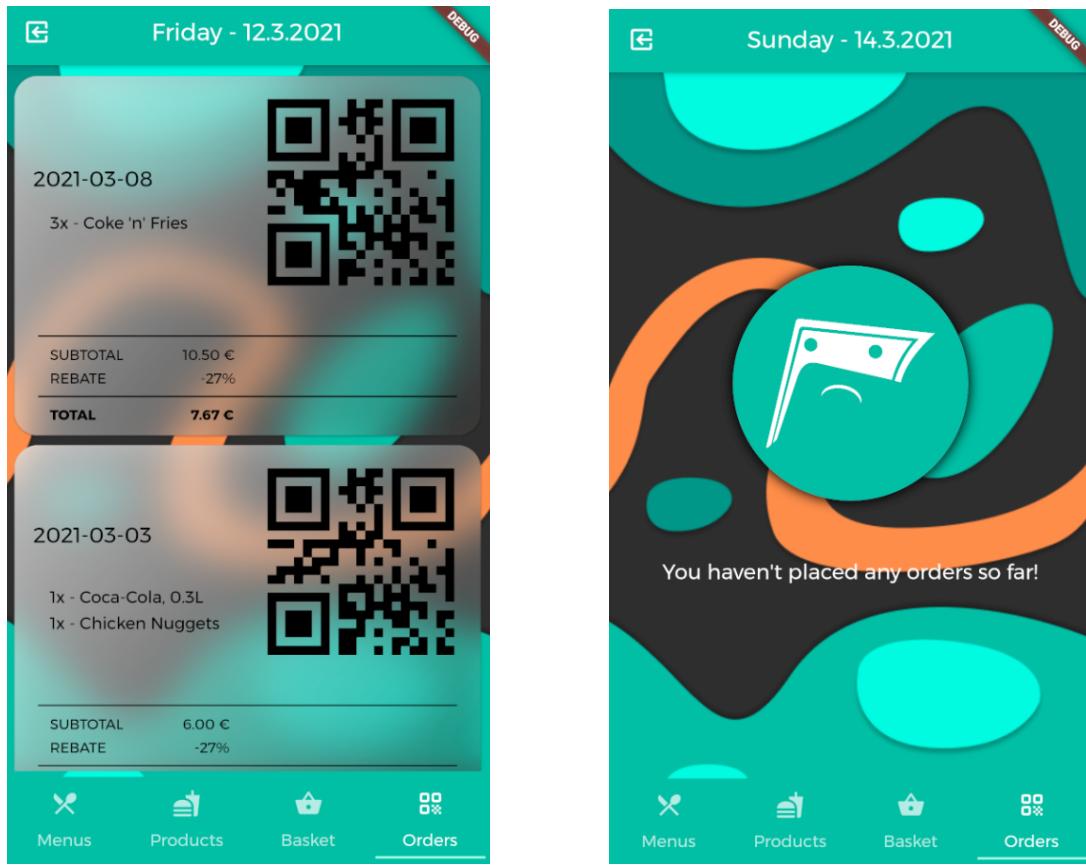
Abbildung 14.7.: Per Modal-Bottom-Sheet vergößerter QR-Code zum Erleichtern des Scans

14. Views

14.5.2. Darstellen der Bestellungen im Order-View

Dem Prinzip des BasketViews folgend, werden auch im Order-View alle Bestellungen des Nutzers in Form von Order-Panels generiert und mittels ListView.builder-Widget angezeigt.

Hat der Nutzer hingegen noch keine Bestellungen getätigt wird, auch hier das Sokka-Logo mit entsprechender Meldung angezeigt.



(a) Order-View mit diversen vom Nutzer getätigten Bestellungen

(b) Leeres Order-View

Abbildung 14.8.: Leeres als auch befülltes Order-View

Teil VI.

Admin-Client

15. Der Sokka-Admin-Client

15.1. Was ist der Admin-Client?

Der **Admin-Client** dient zur Validierung von Bestellungen durch das Personal des Restaurants bzw. der Kantine, das die Sokka-Dienste zur Bestellungsverwaltung nutzt. Somit steht der Admin-Client dem normalen Nutzer nicht zur Verfügung.

Validieren und Abschließen von Bestellungen Mithilfe des eingebauten QR-Scanners werden die QR-Codes von Bestellungen, die der Nutzer mittels Client vorlegt, eingescannt.

Hierdurch kann die Gültigkeit der Bestellung überprüft, deren Inhalte und Bezahlstatus eingesehen und die Bestellung als abgeschlossen markiert werden.

Auf diese Weise wird der QR-Code der entsprechenden Bestellung ungültig gemacht und kann künftig nicht mehr vorgezeigt werden.

16. Admin-Client Utility

16.1. Allgemeines

Wie auch im Client dienen die Utility-Klassen des Admin-Clients für generelle Funktionen wie die Kommunikation mit der Sokka-API oder das Sichern von Daten im externen Speicher des Geräts.

16.2. Network-Wrapper & Cookie-Storage

Sowohl der Network-Wrapper als auch der Cookie-Storage sind aufgrund ihrer Wiederverwendbarkeit direkt vom Client übernommen und im Admin-Client implementiert worden.

16.3. Routes

Die Routes zur Navigation durch den Admin-Client unterscheiden sich kaum von denen des normalen Clients.

Aufgrund seines Zwecks benötigt diese Version des Clients keine Signup-Route, da die Authentifizierung mit ACP-Nutzern, die zuerst im ACP erzeugt werden müssen, erfolgt.

Hierfür wird auf bekannte Weise eine HashMap mit den benötigten Routen erzeugt:

```
1 final Map<String, WidgetBuilder> routes = {  
2     '/':      (BuildContext buildContext) => new HomeScreen(),  
3     '/login':  (BuildContext buildContext) => new LoginScreen(),  
4     '/loading': (BuildContext buildContext) => new LoadingSplashScreen()  
5 };
```

Code-Snippet 16.1.: Routes des Admin-Clients

17. Admin-Client Services

17.1. Allgemeines

Die verschiedenen Services des Admin-Clients sind für die Anfragen an die Sokka-API zur Authentifikation von ACP-Nutzern, das Erzeugen eines Bearer-Tokens, der als Authorization-Parameter für Abfragen an die Order-Route benötigt wird, sowie für das Validieren einer bestimmten Bestellung zuständig.

17.2. ACPUUserAuth

Damit der Admin-Client vom Personal eines Restaurants benutzt werden kann, muss dieses sich mit einem gültigen ACP-Nutzer anmelden.

Anders als beim normalen Client kann hier kein ACP-Nutzer erstellt werden, dies muss über das ACP geschehen.

Dementsprechend sind auch zur Verwendung dieses Services die entsprechenden ACP-User-Routes nötig, die, wie bereits bekannt, als statische Variablen innerhalb der Klasse gespeichert werden.

```
1 static const String ACP_LOGIN_ROUTE = 'https://api.sokka.me/acp/login';
2 static const String ACP_LOGOUT_ROUTE = 'https://api.sokka.me/acp/logout';
3 static const String ACP_VALIDATE_ROUTE = 'https://api.sokka.me/acp/validate';
```

Code-Snippet 17.1.: API-Routes für den User-Auth-Service

Jene Routen werden in den jeweiligen Funktionen aufgerufen, um einen entsprechenden Request an die den Server zu senden.

17.3. ACPOrderValidation

Der *ACP-Order-Validation*-Service bildet das Fundament für die Hauptfunktion des Admin-Clients - dem Validieren von Bestellungen durch Einscannen des entsprechenden QR-Codes.

Hierfür wird ein Bearer-Token als Authorization-Parameter - gleich wie für die Verwendung des FetchOrderables-Service des Clients - benötigt.

Dieser wird mithilfe des Bearer-Authentication-Services, der aus dem Client übernommen worden ist, generiert und setzt sich aus dem Namen und dem Passwort des angemeldeten ACP-Nutzers zusammen.

17.3.1. Service-Routes

```
1 static const ACP_ORDER_VALIDATE = 'https://api.sokka.me/acp/order/validate';
2 static const ACP_ORDER_INVALIDATE = 'https://api.sokka.me/acp/order/invalidate';
```

Code-Snippet 17.2.: API-Routes für den Order-Validation-Service

17.3.2. Validieren einer Bestellung

Wird der QR-Code einer Bestellung mithilfe des Admin-Clients eingescannt, muss als erster Schritt dessen Validität in Hinblick auf

- den Zeitpunkt der Bestellung (diese muss am Vortag vor der „Closing Time“ aufgegeben worden sein)
- Zugehörigkeit der Bestellung zum eingeloggten Nutzer
- Gültigkeit des QR-Codes

überprüft werden.

Hierfür wird mit nachfolgend angeführter Funktion ein entsprechender Request nach dem Scan eines Codes an die API gesendet.

```
1 Future<dynamic> validateOrder({ final String orderKey }) async {
2     final response = await this._networkWrapper
3         .get(
4             '$ACP_ORDER_VALIDATE?order=$orderKey',
5             headers: {
6                 'Content-Type': 'application/json',
7                 'Authorization': await this._bearerAuth.createBearerAuthToken(),
8             },
9         );
10    if (response['success'] && response['valid']) {
11        final order = response['order'];
12
13        final Map<String, int> menuOrders = new Map<String, int>();
14        final Map<String, int> productOrders = new Map<String, int>();
15
16        for (var menu in order['menuOrders']) {
17            menuOrders[menu['menu']['name']] = menu['quantity'];
18        }
19        for (var product in order['productOrders']) {
20            productOrders[product['product']['name']] = product['quantity'];
21        }
22
23        return new Order(orderID: order['id'], userID: order['user_id'], rebate: order['rebate'],
24                         subTotal: order['total'].toDouble(),
25                         timestamp: _dateFormatter.format(DateTime.parse(order['timestamp'])),
26                         menuOrders: menuOrders, productOrders: productOrders);
27
28    } else if (response['success'] && !response['valid']) {
29        return response['reasons'];
30    }
31    return response['message'];
32 }
```

Code-Snippet 17.3.: Überprüfen der Gültigkeit einer Bestellung

17. Admin-Client Services

Ist die Bestellung gültig und der Request damit erfolgreich, so werden die vom Nutzer bestellten Waren, deren Preis, der Rabatt und der Gesamtpreis nach Abzug des Rabatts in einem Dialog-Fenster angezeigt.

Das Personal kann nun den entsprechenden Betrag vom Nutzer kassieren und die Bestellung abschließen und den QR-Code invalidieren.

17.3.3. Abschließen einer Bestellung

Wird eine Bestellung durch den Admin-Client als gültig erkannt und in weiterer Folge durch den Restaurantangestellten kassiert muss die Bestellung und deren QR-Code ungültig gemacht werden, um nicht mehrmals vorgezeigt werden zu können.

Dies kann über den Druck eines Buttons im oben erwähnten Dialog-Fenster, in dem die relevanten Daten der Bestellung aufgelistet sind, durchgeführt werden.

Dabei wird folgende Anfrage an die API gesendet:

```
1 Future<bool> invalidateOrder({ final String orderKey }) async {
2     final response = await this._networkWrapper
3         .post(
4             ACP_ORDER_INVALIDATE,
5             headers: {
6                 'Content-Type': 'application/json',
7                 'Authorization': await this._bearerAuth.createBearerAuthToken(),
8             },
9             body: {
10                 'order': orderKey,
11             },
12         );
13     return response['success'];
14 }
```

Code-Snippet 17.4.: Invalidieren einer Bestellung

18. Admin-Client Screens

18.1. ACP-User-Authentication

Grundsätzlich ist die Authentifikation eines Nutzers zur Verwendung des Admin-Clients gleich aufgebaut, wie jene des normalen Clients.

Die Anmeldedaten werden über Textfelder vom Nutzer eingegeben und per Druck des Submit-Buttons zur Überprüfung an den Server gesendet.

Ist der Anmeldevorgang erfolgreich, wird ein Session-Token übermittelt, der als Cookie im Gerät abgespeichert wird.

18.2. Admin-Client HomeScreen

Über den Home-Screen des Admin-Clients kann per Druck eines Buttons der Scan-Vorgang von QR-Codes gestartet werden. Dabei wechselt die App zur Frontkamera des Gerätes und startet den Scanner.

18.2.1. Scannen von QR-Codes

Für das reibungslose Einscannen eines QR-Codes wird das *flutter_barcode_scanner*-Package verwendet.[23]

Mithilfe dieses Packages kann dynamisch ein neues View mit der Frontkamera, mit welcher auch ein QR-Code eingescannt wird, geöffnet werden.

Dies geschieht auf folgende Weise:

```
1 Future<void> _scanQR() async {
2     final qrCode = await FlutterBarcodeScanner.scanBarcode(
3         '#FF6666',
4         'Cancel',
5         true,
6         ScanMode.QR,
7     );
8     setState(() => this._qrOrderKey = qrCode);
9 }
```

Code-Snippet 18.1.: Öffnen des QR-Scanners

18. Admin-Client Screens

Wird nun obige Funktion mittels Antippen des Buttons im Screen ausgeführt, öffnet sich die Kamera-App des Geräts und beginnt mit dem Scavorgang, so muss der Fokus nur noch auf einen QR-Code gerichtet werden.

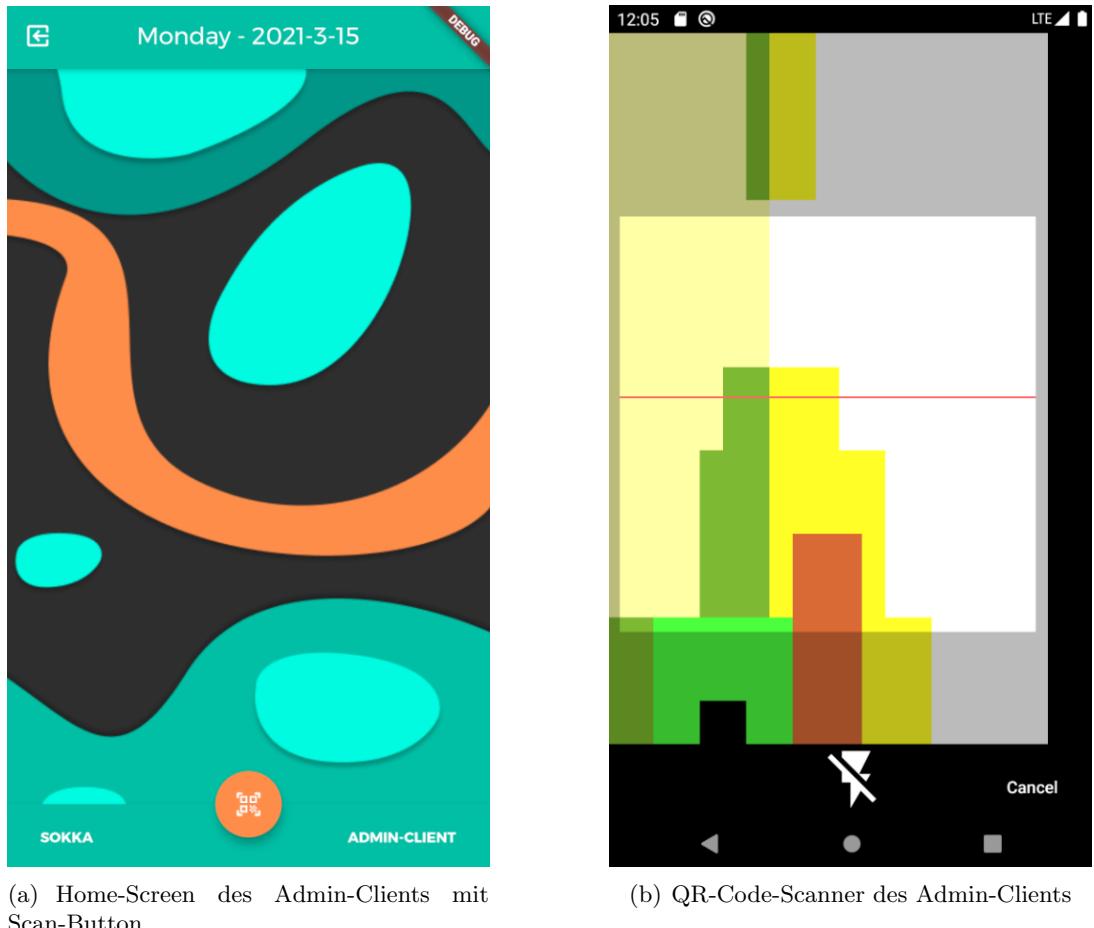


Abbildung 18.1.: Wechsel von Home-Screen zum QR-Code-Scanner

Nach einem erfolgreichen Scan-Vorgang wird automatisch ein Request an die API gesendet und die Gültigkeit des QR-Codes überprüft.

Stellt sich dabei heraus, dass der Code aus folgenden Gründen

- Bestellung wurde nicht am Vortag aufgegeben
- Order-ID und User-ID passen nicht zusammen
- Bestellung wurde bereits invalidiert

nicht gültig ist, wird über einen Alert-Dialog eine entsprechende Fehlermeldung im Admin-Client angezeigt.

Wenn der Code als valide gilt wird ein Alert-Dialog geöffnet, in dem alle relevanten Informationen der Bestellung wie bestellte Waren und deren Preis aufgelistet.

Über dieses Dialogfenster lässt sich in weiterer Folge die Bestellung per Button-Druck, der

18. Admin-Client Screens

einen Invalidierungs-Request an die API auslöst, ungültig machen und damit als abgeschlossen markieren.

Teil VII.

Appendix

Zeitaufwand

Arbeitsprotokoll - Nicolaus Rossi

Projekt	Arbeitsaufwand	Prozentueller Anteil
Planung	8:01:12	3.73%
Frontend	149:19:17	69.43%
Dokumentation	57:43:08	26.84%
Summe	215:03:37	100.00%

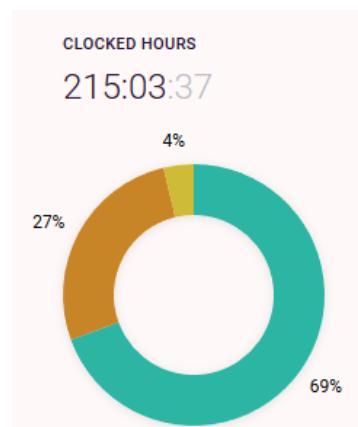


Abbildung 18.2.: Verteilung von Nicolaus' Arbeitsstunden

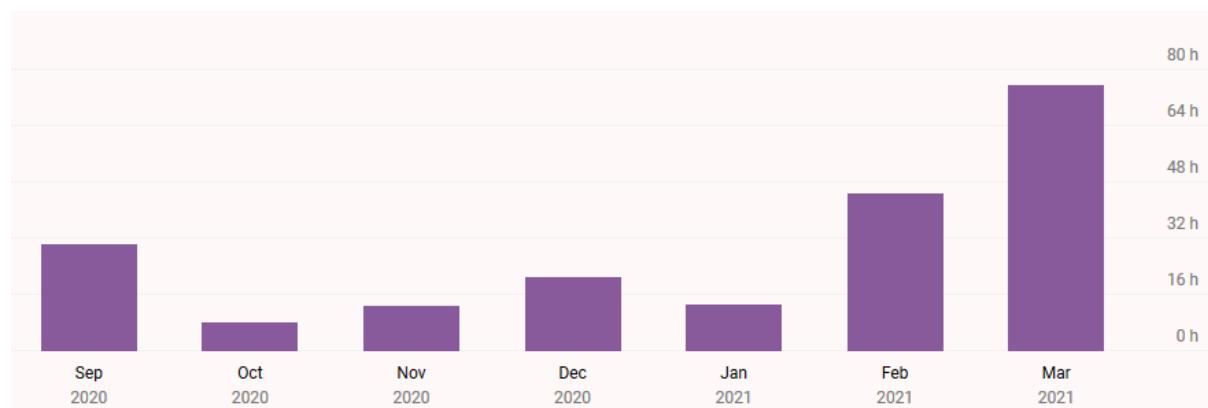


Abbildung 18.3.: Nicolaus' Arbeitsverlauf vom 14.09.2020 bis zum 26.03.2021

Arbeitsprotokoll - Joshua Winkler

Projekt	Arbeitsaufwand	Prozentueller Anteil
Planung	8:52:30	4.81%
Backend	77:07:14	41.82%
ACP	66:08:50	35.87%
Dokumentation	32:16:37	17.50%
Summe	184:25:11	100.00%

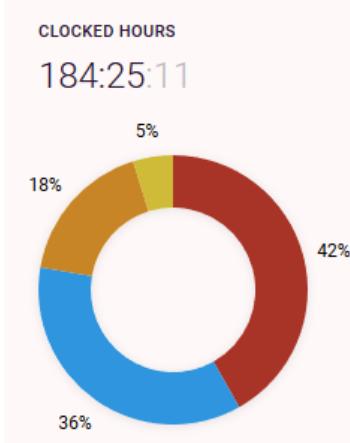


Abbildung 18.4.: Verteilung von Joshuas Arbeitsstunden

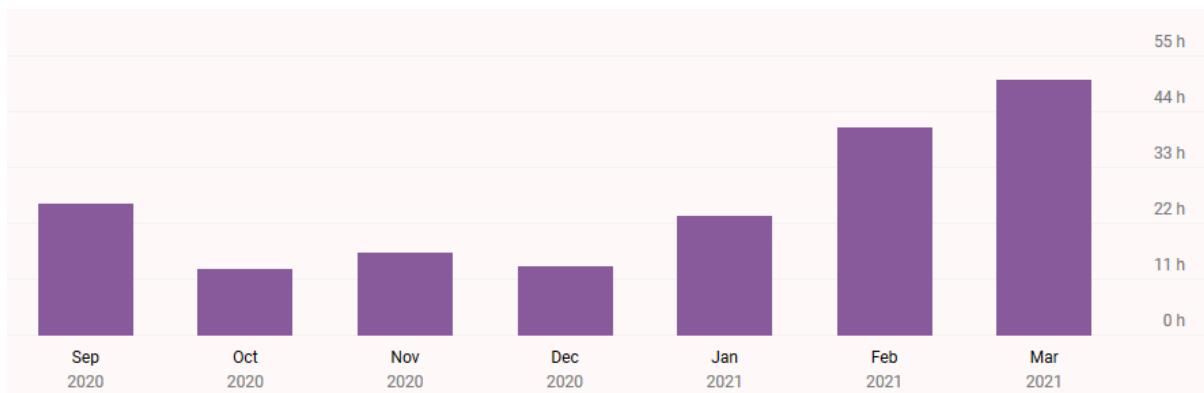


Abbildung 18.5.: Joshuas Arbeitsverlauf vom 14.09.2020 bis zum 26.03.2021

Literaturverzeichnis

- [1] Laura Depta. Global Food Waste and its Environmental Impact. <https://en.reset.org/knowledge/global-food-waste-and-its-environmental-impact-09122018>, 2018. Abgerufen am 17. März 2021.
- [2] Nickelodeon Studios. Sokka. <http://en.nickelodeonarabia.com/shows/avatar/characters/sokka/975mtb>, 2005. Abgerufen am 19. März 2021.
- [3] Aguilar Workshop. Sokka's Boomerang - Avatar The Last Airbender. <https://www.thingiverse.com/thing:4607453>, 2020. Abgerufen am 19. März 2021.
- [4] Technostacks. Which are the most popular programming languages in 2021? <https://technostacks.com/blog/most-popular-programming-languages>, 2021. Abgerufen am 9. März 2019.
- [5] Richard Kenneth Eng. Javascript is a dysfunctional programming language. <https://medium.com/javascript-non-grata/javascript-is-a-dysfunctional-programming-language-a1f4866e186f>, 2016. Abgerufen am 16. März 2021.
- [6] Dart. Dart overview. <https://dart.dev/overview>, 2018. Abgerufen am 3. März 2021, vom Verfasser übersetzt.
- [7] Dart Team. Effective dart: Design. <https://dart.dev/guides/language/effective-dart/design#dont-type-annotate-initialized-local-variables>, 2021. Abgerufen am 9. März 2021.
- [8] Solomon Hykes. Au revoir. <https://www.docker.com/blog/au-revoir/>, 2018. Abgerufen am 9. März 2021.
- [9] Sebastian Springer. Node.js: Das javascript-framework im Überblick. <https://t3n.de/magazin/serverseitige-javascript-entwicklung-nodejs-einsatz-231152/>, 2013. Abgerufen am 9. März 2021.
- [10] Ketankumar Rathod Rahiman Nadaf, Mohammad Afzal Khan. Angular vs. react vs. vue. <https://www.jadeglobal.com/blog/angular-vs-react-vs-vue>, 2020. Abgerufen am 9. März 2021.
- [11] Flutter. Flutter overview. <https://flutter.dev/>, 2018. Abgerufen am 4. März 2021, vom Verfasser übersetzt.
- [12] Flutter. Flutter-Design-Architektur. <https://flutter.dev/docs/resources/architectural-overview>, 2018. Abgerufen am 4. März 2021.

Literaturverzeichnis

- [13] Flutter. Slider-Widget im Material-Design für Android. <https://flutter.github.io/assets-for-api-docs/assets/material/slider.png>, 2018. Abgerufen am 4. März 2021.
- [14] Flutter. Slider-Widget im Cupertino-Design für iOS. <https://flutter.dev/images/widget-catalog/cupertino-slider.png>, 2018. Abgerufen am 4. März 2021.
- [15] Flutter. Widget-Tree eines beispielhaften UI-Layouts. <https://flutter.dev/docs/resources/architectural-overview>, 2018. Abgerufen am 4. März 2021.
- [16] Paul Serby. Case study: How & why to build a consumer app with node.js. <https://venturebeat.com/2012/01/07/building-consumer-apps-with-node/>, 2012. Abgerufen am 15. März 2021.
- [17] Martin Stoeckli. What are salt rounds and how are salts stored in bcrypt? <https://stackoverflow.com/a/46713082/6663020>, 2017. Abgerufen am 9. März 2019.
- [18] Guy Levin. 4 Most Used REST API Authentication Methods. <https://blog.restcase.com/4-most-used-rest-api-authentication-methods/>, 2019. Abgerufen am 2. März 2021.
- [19] nfriedly. Express rate limit. <https://www.npmjs.com/package/express-rate-limit>, 2021. Abgerufen am 8. März 2021.
- [20] Romain Rastel. flutter_staggered_grid_view. https://pub.dev/packages/flutter_staggered_grid_view, 2021. Abgerufen am 12. März 2021.
- [21] Romain Rastel. staggered_grid_view-Image. https://raw.githubusercontent.com/letsar/flutter_staggered_grid_view/master/doc/images/example_01.PNG, 2021. Abgerufen am 12. März 2021.
- [22] Flutter. Implement swipe to dismiss. <https://flutter.dev/docs/cookbook/gestures/dismissible>, 2018. Abgerufen am 12. März 2021.
- [23] Amol Gangadhare. flutter_barcode_scanner. https://pub.dev/packages/flutter_barcode_scanner, 2021. Abgerufen am 15. März 2021.

Abbildungsverzeichnis

1.1.	Der Charakter Sokka aus der Serie <i>Avatar: Der Herr der Elemente</i>	14
1.2.	Sokkas Boomerang und das Projektlogo	15
2.1.	neofetch: Spezifikationen des Test-VServers	16
4.1.	JSX-Components am Beispiel des Sokka-ACPs	30
4.2.	Design-Struktur und Widget-Aufbau von Flutter	32
4.3.	Slider-Widget im Material-Design für Android	33
4.4.	Slider-Widget im Cupertino-Design für iOS	34
4.5.	Widget-Tree für ein beispielhaftes Layout	34
5.1.	Die Ausgabe einer „Hello world!“-Seite mit Express.js	41
6.1.	Eine E-Mail, welche durch Sokka bei der Verifizierung gesendet wurde	45
8.1.	Die Sokka-ACP-Anmeldeseite	95
8.2.	Cookies in den Chromium-Dev-Tools, welche durch das ACP gesetzt wurden . .	95
8.3.	Die Nutzerverwaltung im Sokka-ACP	96
8.4.	Aufrufen einer nicht existierenden URL im Sokka-ACP	97
9.1.	Die Startseite des Sokka-ACPs	98
9.2.	Die Produkteseite des Sokka-ACPs	99
9.3.	Die Bearbeitungsseite von Produkten im Sokka-ACP	99
9.4.	Die Menüseite des Sokka-ACPs	100
9.5.	Die Bearbeitungsseite von Menüs im Sokka-ACP	100
9.6.	Dialog zum Hinzufügen neuer Produkte in einem Menü im Sokka-ACP	101
9.7.	Die Nutzerseite des Sokka-ACPs	102
9.8.	Der Dialog zum Ändern von Rabattgruppen von Nutzern im Sokka-ACP . . .	102
9.9.	Die Gruppenseite des Sokka-ACPs	103
9.10.	Der Dialog zum Ändern des Rabatts von Rabattgruppen im Sokka-ACP . . .	103
9.11.	Die Bestellungsseite des Sokka-ACPs	104
9.12.	QR-Code-Erstellung der Bestellungen	104
9.13.	Die Konfigurationsseite des Sokka-ACPs	105
9.14.	Die Zeitauswahl auf der Konfigurationsseite im Sokka-ACP	105
14.1.	Menu-View mit darauf abgebildeten Menu-Cards	130
14.2.	Product-View mit Kachel für das jeweilige Produkt	131
14.3.	Beispiel für ein Staggered-Grid-View [21]	132
14.4.	Modal-Bottom-Sheet im Basket-View zum Wählen der Bezahloption	134
14.5.	Basket-View in gefüllter und leerer Form	135
14.6.	Order-View mit Card-Widgets für jede Bestellung	136
14.7.	Per Modal-Bottom-Sheet vergößerter QR-Code zum Erleichtern des Scans . .	138

Abbildungsverzeichnis

14.8. Leeres als auch gefülltes Order-View	139
18.1. Wechsel von Home-Screen zum QR-Code-Scanner	147
18.2. Verteilung von Nicolaus' Arbeitsstunden	150
18.3. Nicolaus' Arbeitsverlauf vom 14.09.2020 bis zum 26.03.2021	150
18.4. Verteilung von Joshuas Arbeitsstunden	151
18.5. Joshuas Arbeitsverlauf vom 14.09.2020 bis zum 26.03.2021	151

Code-Snippet-Verzeichnis

3.1.	„fail“ in JavaScript; diese Art von JavaScript ist auch bekannt als <i>JSFuck</i>	19
3.2.	Lustige Mathematik: Die Typenumwandlung (Coercing) in JavaScript	19
3.3.	Eine Klasse mit einer einfachen Funktion in TypeScript	20
3.4.	Eine von TypeScript zu JavaScript kompilierte Klasse	20
3.5.	Einfaches Hello-World-Programm in Dart	21
3.6.	Anlegen einfacher Variablen in Dart	21
3.7.	Finale Variable im lokalen Scope	21
3.8.	Conditional mit zwei Bedingungen	22
3.9.	Arten von for-Schleifen in Dart	22
3.10.	While-Schleife in Dart	22
3.11.	Deklarieren von Funktionen in Dart	23
3.12.	Funktion mit <i>positioned</i> , optional Parametern	23
3.13.	Aufrufen einer Funktion mit <i>positioned</i> Parametern	24
3.14.	Aufrufen einer Funktion mit <i>named</i> Parametern	24
3.15.	Aufrufen einer Funktion mit <i>named</i> Parametern	24
3.16.	Aufrufen einer Funktion mit <i>positioned</i> Parametern	24
3.17.	Simple Klassen in Dart	25
3.18.	Getter- und Setter-Funktionen in Dart	25
3.19.	Vergleich einer Getter-Funktion zwischen Java und Dart	26
3.20.	Erzeugen und Vererben abstrakter Klassen in Dart	26
4.1.	Beispielhaftes Dockerfile für eine Node.js Web-App	27
4.2.	JSX-Beispiel für das Laden und Rendern von Informationen	30
4.3.	Beispiel für ein React-Component für einen Button mit änderbarem Tooltip	31
4.4.	Beispiel für ein React-Component mit State, welches die UNIX-Zeit rendert	31
4.5.	Erzeugen eines einfachen Container-Widgets mithilfe von Flutter	35
4.6.	Erzeugen eines Containers mit verschachtelten Child-Widgets	35
4.7.	<code>build()</code> -Funktion eines Widgets	36
4.8.	Einfaches Stateless Widget	36
4.9.	Ein einfaches Stateful Widget	37
4.10.	Routes für die Navigation zwischen Screens	38
4.11.	Setzen der Routes im App Entry-Point	39
4.12.	<code>Navigator.of()</code> -Funktion zum Wechseln der Screens per <i>namedRoutes</i>	39
4.13.	Navigieren durch die App per MaterialPageRoute	39
5.1.	Beispielhaftes <code>gulpfile.js</code> zum Kompilieren von TypeScript	40
5.2.	„Hello world!“ mit Express.js	41
6.1.	Sokkas Docker Secrets in der <code>docker-compose.yml</code>	44
7.1.	Eine Beispielanfrage mit Authorization-Header in JavaScript	46

Code-Snippet-Verzeichnis

7.2. Sokka-Implementation eines Rate-Limiters für eine REST-Route	47
8.1. React-Router-Implementation des Sokka-ACPs	97
11.1. Hinzufügen des http -Packages zum <code>pubspec.yaml</code> -File	108
11.2. Simples Beispiel für einen POST-Request	109
11.3. GET-Request-Wrapper der NetworkWrapper-Klasse	109
11.4. POST-Request-Wrapper der NetworkWrapper-Klasse	110
11.5. Dart's syntaktische Lösung für die Erstellung eines Singletons	110
11.6. Speichern von Key-Value-Pairs in den externen Gerätespeichern	111
11.7. Definieren von statischen Key-Strings für erhöhte Typsicherheit	111
11.8. Verwalten von Cookie-Daten mithilfe oben definierter Static-Keys	112
12.1. Benötigte <code>/user</code> -Routes der Sokka-API	114
12.2. Funktion zum Registrieren eines neuen Nutzers	115
12.3. Funktion zum Anmelden eines bestehenden Nutzers	115
12.4. Funktion zum Abmelden eines Nutzers in der App	116
12.5. Funktion zur Validierung eines gespeicherten Nutzer-Session-Tokens	116
12.6. Wrapper-Funktion für die einfache Validierung eines Session-Tokens	117
12.7. Generieren eines Bearer-Authorization-Tokens	117
12.8. Erstellen eines Test-Bearer-Tokens	118
12.9. Test-Bearer-Authorization-Token	118
12.10. Benötigte <code>/menu</code> -, <code>/product</code> -, und <code>/image</code> -Routes der Sokka-API	118
12.11. Abrufen und Speichern der verfügbaren Menüs	119
12.12. Abrufen und Speichern der verfügbaren Produkte	119
12.13. Abrufen und Speichern der verfügbaren Menüs	120
12.14. Benötigte Routes der Sokka-API zur Verwaltung von Nutzer-Bestellungen	120
12.15. <code>loadMenus()</code> -Funktion zum Laden bestellter Menüs in den Request-Body	121
12.16. <code>loadProducts()</code> -Funktion zum Laden bestellter Produkte in den Request-Body	121
12.17. <code>loadProducts()</code> -Funktion zum Laden bestellter Produkte in den Request-Body	122
12.18. Funktion zum Abfragen aller vom Nutzer getätigten Bestellungen	123
13.1. Login-Screen als Stateful Widget mit <code>TextFormFields</code> und <code>TextEditingController</code> s für E-Mail und Passwort	125
13.2. Loading-Splash-Screen in Form eines Stateless Widgets	126
13.3. Loading-Screen als Alternativ-Route per <code>MediaQuery</code>	127
13.4. <code>DefaultTabController</code> mit vier Tabs und Tab-Bar zur Navigation	128
14.1. <code>ListView.builder</code> -Widget zum Erzeugen und Darstellen aller Menu-Cards	130
14.2. <code>StaggeredGridView.countBuilder</code> -Widget zum Erzeugen und Darstellen der Product-Tiles	132
14.3. <code>Dismissible</code> -Widget im Basket-View mit <code>_updateTotalPrice</code> -Funktion zum Aktualisieren des Gesamtpreises	133
14.4. <code>StaggeredGridView.countBuilder</code> -Widget zum Erzeugen und Darstellen der Product-Tiles	135
14.5. Implementation der <code>onTap</code> -Funktionalität des QR-Codes	137
16.1. Routes des Admin-Clients	142
17.1. API-Routes für den User-Auth-Service	143

Code-Snippet-Verzeichnis

17.2. API-Routes für den Order-Validation-Service	144
17.3. Überprüfen der Gültigkeit einer Bestellung	144
17.4. Invalidieren einer Bestellung	145
18.1. Öffnen des QR-Scanners	146