

ENTWICKLUNG EINER FORMULARANWENDUNG MIT
KOMPATIBILITÄTSVALIDIERUNG DER EINFACH- UND
MEHRFACHAUSWAHL-EINGABEFELDER

Vorgelegt von:

Alexander Johr

Friedrichstraße 57-59

Wohnheim 2

Wohneinheit 004

38855 Wernigerode

m27007

Erstprüfer: Prof. Jürgen Singer Ph.D.

Zweitprüfer: Prof. Daniel Ackermann

Datum: 01.09.2021

THEMA UND AUFGABENSTELLUNG DER MASTERARBEIT
MA AI 29/2021

FÜR HERRN ALEXANDER JOHR

ENTWICKLUNG EINER FORMULARANWENDUNG MIT
KOMPATIBILITÄTSVALIDIERUNG DER EINFACH- UND
MEHRFACHAUSWAHL-EINGABEFELDER

Das Thünen-Institut für Ländliche Räume wertet Daten zu Maßnahmen auf landwirtschaftlich genutzten Flächen aus. Dafür müssen entsprechende Maßnahmen bundesweit mit Zeitbezug auswertbar sein und mit Attributen versehen werden. Um die Eingabe für die Wissenschaftler des Instituts zu beschleunigen und um fehlerhafte Eingaben zu minimieren, soll eine spezielle Formularanwendung entwickelt werden. Neben herkömmlichen Freitextfeldern beinhaltet das gewünschte Formular zum Großteil Eingabefelder für Einfach- und Mehrfachauswahl. Je nach Feld kann die Anzahl der Auswahloptionen mitunter zahlreich sein. Dem Nutzer sollen daher nur solche Auswahloptionen angeboten werden, die zusammen mit der zuvor getroffenen Auswahl sinnvoll sind.

Im Wesentlichen ergibt sich die Kompatibilität der Auswahloptionen aus der Bedingung, dass für dasselbe oder ein anderes Eingabefeld eine Auswahlmöglichkeit gewählt bzw. nicht gewählt wurde. Diese Bedingungen müssen durch Konjunktion und Disjunktion verknüpft werden können. In Sonderfällen muss ein Formularfeld jedoch auch die Konfiguration einer vom Standard abweichenden Bedingung ermöglichen. Wird dennoch versucht, eine deaktivierte Option zu selektieren, wäre eine Anzeige der inkompatiblen sowie der stattdessen notwendigen Auswahl ideal.

Die primäre Zielplattform der Anwendung ist das Desktop-Betriebssystem Microsoft Windows 10. Idealerweise ist die Formularanwendung auch auf weiteren Desktop-Plattformen sowie mobilen Endgeräten wie Android- und iOS-Smartphones und -Tablets lauffähig. Die Serialisierung der eingegebenen Daten genügt dem Institut zunächst als Ablage einer lokalen Datei im JSON-Format.

Die Masterarbeit umfasst folgende Teilaufgaben:

- Analyse der Anforderungen an die Formularanwendung
- Evaluation der angemessenen Technologie für die Implementierung
- Entwurf und Umsetzung der Übersichts- und Eingabeoberfläche
- Konzeption und Implementierung der Validierung der Eingabefelder
- Entwicklung von automatisierten Testfällen zur Qualitätskontrolle
- Bewertung der Implementierung und Vergleich mit den Wunschkriterien

Digital unterschrieben von
Juergen K. Singer
o= Hochschule Harz,
Hochschule fuer
angewandte
Wissenschaften, l=
Wernigerode
Datum: 2021.03.23 12:30:
26 MEZ



Prof. Jürgen Singer Ph.D.
1. Prüfer



Prof. Daniel Ackermann
2. Prüfer

Inhaltsverzeichnis

Abbildungsverzeichnis	9
Listingverzeichnis	11
I. Einleitung und Gliederung	15
1. Einleitung	17
1.1. Problemstellung	17
1.2. Gliederung	18
II. Vorbereitung	21
2. Technologie Auswahl	23
2.1. Trendanalyse	23
2.1.1. Stack Overflow Umfrage	23
2.1.2. Google Trends	24
2.1.3. Frameworks mit geringer Relevanz	24
2.1.4. Frameworks mit sinkender Relevanz	26
2.1.5. Frameworks mit steigender Relevanz	27
2.2. Vergleich von React Native und Flutter	28
2.2.1. Vergleich zweier minimaler Beispiele für Formulare und Validierung .	28
2.2.2. Automatisiertes Testen	30
2.3. Fazit und Begründung der Auswahl	32
3. Grundlagen	35
3.1. Flutter	35
3.2. Dart Grundlagen	36
3.2.1. AOT und JIT	37
3.2.2. Set und Map Literale	38
3.2.3. Typen ohne Null-Zulässigkeit	39
3.2.4. Typen mit Null-Zulässigkeit	40
3.2.5. Asynchrone Programmierung	42

III. Implementierung	45
4. Schritt 1 - Formular in Grundstruktur erstellen	47
4.1. Auswahloptionen hinzufügen	48
4.2. Serialisierung einer Maßnahme	53
4.3. Test der Serialisierung einer Maßnahme	56
4.4. Serialisierung der Maßnahmenliste	59
4.5. Test der Serialisierung der Maßnahmenliste	60
4.6. Der Haupteinstiegspunkt	62
4.7. Der Service für den Applikations übergreifenden Zustand	64
4.8. Speichern der Maßnahmen in eine JSON-Datei	66
4.9. Abhängigkeit zum Verwalten der Maßnahmen	68
4.10. Übersichtsbildschirm der Maßnahmen	70
4.10.1. Auflistung der Maßnahmen im Übersichtsbildschirm	72
4.11. Widget MassnahmenTable	73
4.12. Das View Model	77
4.13. Eingabeformular	79
4.13.1. Ausgabe der Formularfelder	82
4.13.2. Eingabefeld für den Maßnahmentitel	83
4.13.3. Speicher-Routine	83
4.14. Widget SelectionCard	85
4.14.1. Bildschirm für die Auswahl der Optionen	88
4.15. Integrations-Test zum Test der Oberfläche	91
5. Schritt 2	99
5.0.1. Integrationstest erweitern	100
5.0.2. Hinzufügen der Auswahloptionen	101
5.0.3. Aktualisierung des Models	101
5.0.4. Aktualisierung der Übersichtstabelle	102
5.0.5. Aktualisierung des ViewModels	103
5.0.6. Aktualisierung der Eingabemaske	104
6. Schritt 3	107
6.1. Einfügen des Form-Widgets	107
6.2. Validierung des Maßnahmentitels	108
7. Schritt 4	115
7.1. Hinzufügen der Bedingungen zu den Auswahloptionen	116
7.2. Hinzufügen der Momentaufnahme aller ausgewählten Optionen im gesamten Formular	118
7.3. Reagieren der Selektionskarte auf die ausgewählten Optionen	119
7.4. Reagieren des Auswahlbildschirms auf die ausgewählten Optionen	122
7.4.1. Hinzufügen der Momentaufnahme zur Validierung	125
8. Schritt 5	127

9. Schritt 6	131
9.1. Integrationstest erweitern	132
9.2. Hinzufügen der Menge der Nebenziele	133
9.3. Aktualisierung des Models	133
9.4. Aktualisierung der Übersichtstabelle	133
9.5. Aktualisierung des ViewModels	134
9.6. Aktualisierung der Eingabemaske	136
9.7. Aktualisierung der Selektionskarte	137
10. Schritt 7	141
IV. Fazit	147
11. Diskussion	149
11.1. Reevaluation des Zustandsmanagements	149
12. Anzeige von fehlerhaften Teilkomponenten der Bedingungen von deaktivierten Auswahloptionen	153
13. Schlussfolgerung-und-Ausblick	155
Literatur	157
Eidesstattliche Erklärung	160

Abbildungsverzeichnis

2.1. Stimmen der Stack Overflow Umfrage von 2013 bis 2020	25
2.2. Suchinteresse der Frameworks mit geringer Relevanz	26
2.3. Stimmen für Cordova und PhoneGap 2013 bis 2020	26
2.4. Stimmen für Xamarin und Cordova	27
2.5. Suchinteresse sinkende und steigende Relevanz	27
2.6. Stimmen für React Native und Flutter	28
4.1. Schritt 1 Übersicht	47
4.2. Schritt 1 Eingabemaske	48
4.3. Schritt 1 Selektions-Bildschirm für Status	48
4.4. UML Diagramme	65
4.5. UML Diagramm	77
5.1. Schritt 2 Übersicht	99
5.2. Schritt 2 Eingabemaske	99
6.1. Schritt 3 Eingabemaske	107
7.1. Schritt 4 Eingabemaske	115
7.2. Schritt 4 Eingabemaske	115
7.3. Schritt 4 Eingabemaske	115
8.1. XXX	129
8.2. XXX	130
8.3. XXX	130
9.1. Schritt 6 Eingabemaske	131
9.2. Schritt 6 Eingabemaske	131
9.3. Schritt 6 Eingabemaske	131
10.1. Schritt 7 Eingabemaske	141
10.2. Schritt 7 Eingabemaske	141
10.3. Schritt 7 Eingabemaske	141
10.4. UML Diagramme	143
10.5. UML Diagramme	144

Listingverzeichnis

3.1. Ein Set	38
3.2. Collection-for in einer Menge	38
3.3. Collection-for in einer Hashtabelle	39
3.4. Collection-if in einer Liste	39
3.5. Collection-if in einer Liste	40
3.6. Collection-if in einer Liste	41
3.7. Collection-if in einer Liste	41
3.8. Collection-if in einer Liste	41
3.9. Collection-if in einer Liste	42
3.10. Collection-if in einer Liste	42
3.11. Collection-if in einer Liste	43
3.12. Collection-if in einer Liste	43
3.13. Collection-if in einer Liste	44
3.14. Collection-if in einer Liste	44
4.1. Schritt 1 Die Klasse LetzterStatus	49
4.2. Schritt 1 Die Klasse Choice	49
4.3. Schritt 1 Die Menge letzterStatusChoices	50
4.4. Schritt 1 Die Klasse Choices	51
4.5. built_value Live Template	53
4.6. Schritt 1 Der Werte-Typ Massnahme	54
4.7. Schritt 1 Der Werte-Typ Identifikatoren	55
4.8. Schritt 1 Der Werte-Typ LetzteBearbeitung	56
4.9. Schritt 1 Der Serialisierer für Massnahme und Storage	56
4.10. Schritt 1 Serialisierung einer Maßnahme Unittest	57
4.11. Schritt 1 Deserialisierung einer Maßnahme Unittest	58
4.12. Schritt 1 Instanzvariable <code>letzteBearbeitung</code> gibt einen <code>LetzteBearbeitungBuilder</code> zurück	59
4.13. Schritt 1 Der Werte-Typ Storage	60
4.14. Schritt 1 Ein automatisierter Testfall überprüft	60
4.15. Schritt 1 Instanzvariable <code>massnahmen</code> gibt einen <code>SetBuilder</code> zurück	61
4.16. Schritt 1 Der Haupteinstiegspunkt	62
4.17. Schritt 1 Der Service AppState	65
4.18. Schritt 1 Die Klasse MassnahmenJsonFile	67
4.19. Schritt 1 Die Klasse MassnahmenModel	69

4.20. Schritt 1 Die Struktur der Klasse MassnahmenMasterScreen	71
4.21. Schritt 1 Die Ausgabe der Maßnahmen	75
4.22. Schritt 1 Die Klasse MassnahmenTable	76
4.23. Schritt 1 Die Klasse MassnahmenFormViewModel	78
4.24. Schritt 1 Klasse MassnahmenDetailScreen Struktur	81
4.25. Schritt 1 Die Ausgabe der Formularfelder	82
4.26. Schritt 1 Die Funktion createMassnahmenTitelTextFormField	83
4.27. Schritt 1 Die Funktion saveRecordAndGoBackToOverviewScreen	84
4.28. Schritt 1 Die Klasse SelectionCard	85
4.29. Schritt 1 Die Build Methode der SelectionCard	87
4.30. Schritt 1 Die Funktion createMultipleChoiceSelectionScreen	90
4.31. Schritt 1 Initialisierung des Integrations Tests	92
4.32. Schritt 1 Initialisierung des Integrations Tests	92
4.33. Schritt 1 Initialisierung des Widgets für den Integrations Tests	93
4.34. Schritt 1 Die Hilfsmethode tabSelectionCard	93
4.35. Schritt 1 Die Hilfsmethode tabConfirmButton	95
4.36. Schritt 1 Die Hilfsmethode tabOption	95
4.37. Schritt 1 Die Hilfsmethode fillTextFormField	96
4.38. Schritt 1 Der Button zum Kreieren einer Maßnahme wird ausgelöst	96
4.39. Schritt 1 Der letzte Status wird ausgewählt	96
4.40. Schritt 1 Der Maßnahmentitel wird eingegeben	97
4.41. Schritt 1 Validierung des Testergebnisses	97
5.1. Schritt 2 Der Integrationstest klickt 5 weitere Karten	100
5.2. Schritt 2 Der Integrationstest klickt 5 weitere Karten	100
5.3. Schritt 2 Die Klasse FoerderklasseChoice	101
5.4. Schritt 2 massnahmenCharakteristika wird Massnahme hinzugefügt	101
5.5. Schritt 2 Der Werte-Typ Massnahmencharakteristika	102
5.6. Schritt 2 Maßnahmencharakteristika werden dem Tabellenkopf hinzugefügt .	102
5.7. Schritt 2 Maßnahmencharakteristika werden dem Tabellenkörper hinzugefügt	103
5.8. Schritt 2 Maßnahmencharakteristika werden dem ViewModel hinzugefügt .	103
5.9. Schritt 2 Konvertierung des Models in das ViewModel	104
5.10. Schritt 2 Konvertierung des ViewModels in das Model	104
5.11. Schritt 2 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	105
5.12. Schritt 2 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	105
5.13. Schritt 2 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	106
5.14. Schritt 2 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	106
6.1. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	108
6.2. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	108
6.3. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	109
6.4. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt .	110
6.5. Schritt 3 errorText wird der SelectionCard hinzugefügt	111
6.6. Schritt 3 errorText wird ausgegeben	111

6.7. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt	112
6.8. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt	112
6.9. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt	113
6.10. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt	114
6.11. Schritt 3 Die Maßnahmencharakteristika Selektionskarten werden ergänzt	114
7.1. Schritt 4 XXXXX	116
7.2. Schritt 4 XXXXX	116
7.3. Schritt 4 XXXXX	117
7.4. Schritt 4 XXXX	118
7.5. Schritt 4 Die Ausgabe der Formularfelder	119
7.6. Schritt 4 XXXX	120
7.7. Schritt 4 XXXX	121
7.8. Schritt 4 XXXX	124
7.9. Schritt 4 Die Ausgabe der Formularfelder	125
8.1. Schritt 5 XXXX	129
9.1. Schritt 6 XXXX	132
9.2. Schritt 6 XXXX	132
9.3. Schritt 6 XXXX	133
9.4. Schritt 6 XXXX	133
9.5. Schritt 6 XXXXX	134
9.6. Schritt 6 XXXXX	134
9.7. Schritt 6 XXXX	134
9.8. Schritt 6 XXXX	135
9.9. Schritt 6 XXXX	135
9.10. Schritt 6 XXXX	136
9.11. Schritt 6 XXXX	136
9.12. Schritt 6 XXXX	137
9.13. Schritt 6 XXXX	138
9.14. Schritt 6 XXXX	139
10.1. Schritt 7 XXXX	142
10.2. Schritt 7 XXXX	144
10.3. Schritt 7 XXXX	145
10.4. Schritt 7 XXXX	145
10.5. Schritt 7 XXXX	146
10.6. Schritt 7 XXX	146
10.7. Schritt 7 XXXX	146
11.1. built_value Live Template	150
11.2. built_value Live Template	150
11.3. built_value Live Template	150
11.4. Die Klasse CartBloc	151

11.5. Die vereinfachte Klasse CartBloc 152

Teil I

EINLEITUNG UND GLIEDERUNG

1. Einleitung

Eine angenehme Erfahrung für den Nutzer einer Software entsteht unter anderem dann, wenn ihm die richtigen Information zur richtigen Zeit präsentiert werden. In Formularen spielen Einfach- und Mehrfachauswahl Felder – im Englischen unter dem Begriff multiple choice Zusammengefasst – eine Rolle.

Die richtigen Informationen zur richtigen Zeit zu präsentieren könnte in diesem Kontext bedeuten, nur solche Auswahloptionen anzubieten, welche mit den bisherigen gewählten Optionen Sinn ergeben. Für die Datenerfassung von Maßnahmen auf landwirtschaftlich genutzten Flächen stellt dies eine Herausforderung dar, denn die Auswahlfelder und Optionen sind zahlreich und ihre Bedingungen komplex. Es lassen sich folgende Probleme ableiten.

1.1. Problemstellung

Das primäre Problem und damit Musskriterium der Formular-Anwendung ist, dass sich die Auswahlfelder untereinander beeinflussen. Wird eine Option in einem Auswahlfeld selektiert, so werden die möglichen Auswahlfelder von potenziell jedem weiteren Auswahlfeld dadurch manipuliert. Es muss eine Möglichkeit gefunden werden, die Abhängigkeiten in einer einfachen Art und Weise für jede Auswahloption zu hinterlegen und bei Bedarf abzurufen.

Das sekundäre Problem, welches sich vom primären Problem ableiten lässt, ist die Laufzeitgeschwindigkeit. Wenn die Auswahl in einem Auswahlfeld die Auswahlmöglichkeiten in potenziell allen anderen Auswahlfeldern manipuliert, so könnte dies zu einer hohen Last beim erneuten Zeichnen der Oberfläche zur Folge haben. Wann immer der Nutzer eine Selektion tätigt, müsste das gesamte Formular neu gezeichnet werden, um sicherzustellen, dass invalide Auswahloptionen gekennzeichnet werden. Bei einem Formular mit wenigen Auswahlfeldern wäre das kein Problem, doch die nötigen Auswahlfelder für das Eintragen von Maßnahmen des Europäischen Landwirtschaftsfonds für die Entwicklung des ländlichen Raums (ELER) sind zahlreich. Ein automatisierter Integrationstest, welcher im Formular Daten einer beispielhaften Maßnahme einträgt, zählt zum Zeitpunkt der Erstellung dieser Arbeit bereits 58 aufgerufene Auswahlfelder und 107 darin selektierte

Auswahloptionen. Das bedeutet, dass bei jedem dieser 107 Selektionen die 58 Auswahlfelder und all ihre Kinder neu gezeichnet werden müssten. Es entstehen also Wartezeiten nach jedem Auswählen einer Option. Das Formular soll in Zukunft zudem noch erweitert und auch für die Eingabe ganz anderer Datensätze mit potenziell noch mehr Auswahlfeldern eingesetzt werden können. Die Dateneingabe wäre mit den Wartezeiten trotzdem möglich. Daher ist es ein Wunschkriterium, dass ein Mechanismus gefunden wird, der nur die Elemente neu zeichnet, die sich wirklich ändern.

Ein weiteres Wunschkriterium ist, dass der Benutzer beim Anwählen einer deaktivierten Auswahloption eine Mitteilung darüber erhält, welche der zuvor ausgewählten Optionen zu der Inkompatibilität mit dem gewünschten Optionen führt.

Ziel dieser Masterarbeit ist es eine geeignete Technologie für die Umsetzung auszuwählen und die Umsetzbarkeit der oben genannten Kriterien zu evaluieren.

1.2. Gliederung

Kapitel 2 evaluiert die Kandidaten der Frontendtechnologien, die für eine nähere Betrachtung infrage kommen. Dazu werden die Umfrageergebnisse der Stack Overflow -Umfragen sowie das relative Suchinteresse dieser Technologien auf Google Trends analysiert. Da die Technologien React Native und Flutter die am verbreitetsten Technologien hervorgingen, werden sie daraufhin einem detaillierteren Vergleich unterzogen.

Da als Frontendtechnologie für die Entwicklung der Formularanwendung Flutter gewählt wurde, beschäftigt sich Kapitel 3 mit den Grundlagen des Frameworks und der zugrunde liegenden Programmiersprache Dart.

Die Kapitel 4 bis 10 dokumentieren die nötigen Entwicklungsschritte, um die einzelnen aufeinander aufbauenden Funktionalitäten hinzuzufügen. Die während der Arbeit im Thünen-Institut entstandene Anwendung wurde zu diesem Zweck auf die für die Problemstellung bedeutsamsten Funktionalitäten reduziert. Die Anzahl der Auswahlfelder beschränkt sich darüber hinaus auf ein Mindestmaß,

welches die Bedingungen der Auswahloptionen untereinander erkennbar macht.

Kapitel 4 stellt die grundlegende Struktur der Anwendung her. Kapitel 5 fügt Hilfsmethoden hinzu, welche das Hinzufügen weiterer Formularfelder in den folgenden Schritten vereinfachen wird.

In Kapitel 6 erhält die Anwendung die grundlegende Funktion, Felder zu validieren. Kapitel 7 erweitert die Validierung schließlich um die Bedingungen der Auswahloptionen. Als

Konsequenz werden alle Formularfelder neu gezeichnet, sollte der Benutzer eine beliebige Auswahloption selektieren. Durch die Validierung geschieht es nach dem Neuzeichnen, dass invalide Auswahlfelder rot markiert werden. Die erforderlichen Änderungen, um nur die Auswahlfelder zu aktualisieren, die ihre Validität oder ihren eigenen Inhalt ändern, wird in Kapitel 8 hinzugefügt.

Kapitel 9 ergänzt die Möglichkeit, Mehrfachauswahlfelder zu verwenden. Kapitel 10 sorgt dafür, dass auch benutzerdefinierte Bedingungen für die Auswahlfelder hinterlegt werden können.

Kapitel 11 setzt sich mit den Erkenntnissen auseinander, die während der Entwicklung der Anwendung gesammelt wurden. Kapitel 13 bewertet die Erkenntnisse, ergänzt sie um einen Ausblick und vergleicht die Ergebnisse der Entwicklung mit den Anforderungen.

Teil II

VORBEREITUNG

2. Technologie Auswahl

Die folgenden drei Kapitel behandeln die Auswahl der Frontend-Technologie für die Umsetzung der Formularanwendung. Dazu werden im ersten Schritt die dafür in Frage kommenden Technologien identifiziert. Anschließend wird der Trend der Popularität dieser Technologien miteinander verglichen. Die daraus resultierenden Kandidaten sollen dann detaillierter untersucht werden. In Hinblick auf die Anforderungen an die Formularanwendung soll dabei die angemessenste Frontend-Technologie ausgewählt werden.

2.1. Trendanalyse

Zwei Quellen wurden für die Analyse der Technologie-Trends ausgewählt: die Ergebnisse der jährlichen Stack Overflow-Umfragen und das Such-Interesse von Google Trends.

2.1.1. Stack Overflow Umfrage

Die Internet-Plattform Stack Overflow richtet sich an Softwareentwickler und bietet ihren Nutzern die Möglichkeiten, Fragen zu stellen, Antworten einzustellen und Antworten anderer Nutzer auf- und abzuwerten.

Besonders für Fehlermeldungen, die häufig während der Softwareentwicklung auftreten, findet man auf dieser Plattform rasch die Erklärung und den Lösungsvorschlag gleich mit. So lässt sich auch die Herkunft des Domain-Namens herleiten:

We named it Stack Overflow, after a common type of bug that causes software to crash – plus, the domain name stackoverflow.com happened to be available.

- Joel Spolsky, Mitgründer von Stack Overflow ¹

Aufgrund des Erfolgsrezepts von Stack Overflow ist die Plattform kaum einem Softwareentwickler unbekannt. Dementsprechend nehmen auch jährlich tausende Entwickler an den von Stack Overflow herausgegebenen Umfragen teil. Seit 2013 beinhalten die Umfragen

¹[35]

auch die Angabe der aktuell genutzten und in Zukunft gewünschten Frontend-Technologien. Stackoverflow erstellt aus diesen gesammelten Daten Auswertungen und Übersichten. Doch gleichzeitig werden die zugrundeliegenden Daten veröffentlicht.²

Um den Trend der Beliebtheit der Frontend-Technologien aufzuzeigen, wurde ein Jupyter Notebook erstellt. Es transformiert die Daten in ein einheitliches Format, da die Umfrageergebnisse von Jahr zu Jahr in einer unterschiedlichen Struktur abgelegt wurden. Anschließend erstellt es Diagramme, die im Folgenden analysiert werden. Das Jupyter Notebook ist im Anhang zu finden.

2.1.2. Google Trends

Suchanfragen die über die Suchmaschine Google abgesetzt werden, lassen sich über den Dienst Google Trends als Trenddiagramm visualisieren. Die Ergebnisse werden normalisiert, um das relative Such-Interesse abzubilden und die Ergebnisse auf einer Skala von 0 bis 100 darstellen zu können.³

Google Trends ist keine wissenschaftliche Umfrage und sollte nicht mit Umfragedaten verwechselt werden. Es spiegelt lediglich das Suchinteresse an bestimmten Themen wider.⁴

Genau aus diesem Grund wird Google Trends im Folgenden lediglich zum Abgleich der Ergebnisse der Stack Overflow Umfrage eingesetzt.

2.1.3. Frameworks mit geringer Relevanz

NativeScript, Sencha (bzw. Sencha Touch) und Appcelerator spielen in den Umfrageergebnissen eine untergeordnete Rolle. Dies ist in den aufsummierten Stimmen von 2013 bis 2020 für alle in der Umfrage auftauchenden Frontend-Technologien zu sehen (Abb. ??).

Auch das Suchinteresse auf Google ist für diese Frameworks äußerst gering. In Abbildung 2.2 werden NativeScript, Sencha, Appcelerator und auch Adobe PhoneGap mit Apache Cordova für das relative Suchinteresse verglichen.

²[36]

³Vgl. [23]

⁴[23]

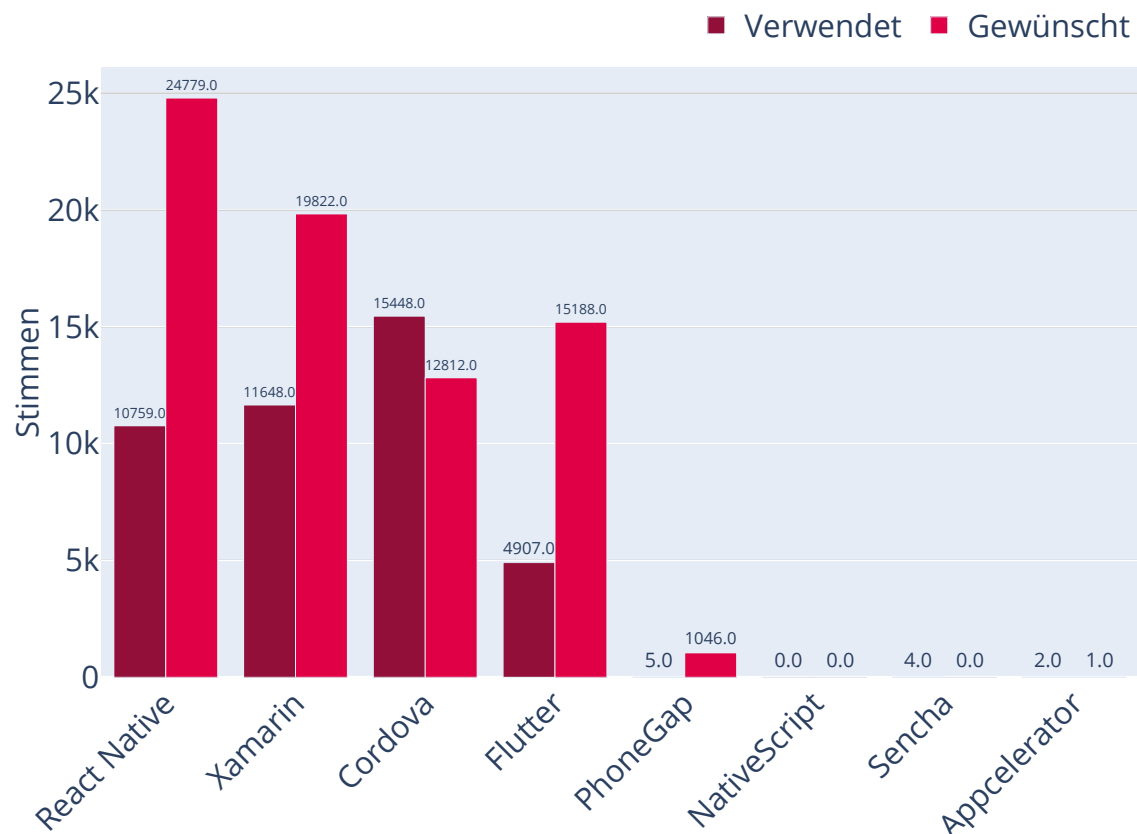


Abbildung 2.1.: Summe der Stimmen der Stack Overflow Umfrage von 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle: **FEHLT!**

Verwandte Technologien zu Apache Cordova

Das Ionic Framework taucht in den Ergebnissen der Stack Overflow Umfragen nicht auf. Ein Grund dafür könnte sein, dass es auf Apache Cordova aufbaut⁶, welches bereits in den Ergebnissen vorkommt. Adobe PhoneGap taucht zwar in den Ergebnissen von 2013 mit 1043 Stimmen auf (Siehe Abbildung 2.3), verliert jedoch in den Folgejahren mit weniger als 10 Stimmen abrupt an Relevanz. Das stimmt nicht mit dem Suchinteresse auf Google überein, da Adobe PhoneGap dort erst ab 2014 anfängt, langsam an Relevanz zu verlieren, wie in Abbildung 2.2 zu sehen ist. 2013 existierte PhoneGap noch als extra Mehrfachauswahlfeld in den Daten, während es ab 2014 nur noch in dem Feld für die sonstigen Freitext Angaben auftaucht⁷. Auch Adobe PhoneGap baut auf Apache Cordova auf⁸. Für diese Auswertung spielen diese verwandten Technologien eine untergeordnete Rolle, da sie auch in den Google Trends weit hinter Apache Cordova zurückbleiben (Abb. 2.2).

Am Beispiel von Adobe PhoneGap wird deutlich, wie wichtig es ist, auf eine Technologie zu setzen, die weit verbreitet ist. Im schlimmsten Fall wird die Technologie sogar vom Betreiber aufgrund zu geringer Nutzung komplett eingestellt, wie es bei PhoneGap bereits

⁶[30]

⁷Vgl. [36]

⁸Vgl. [1]

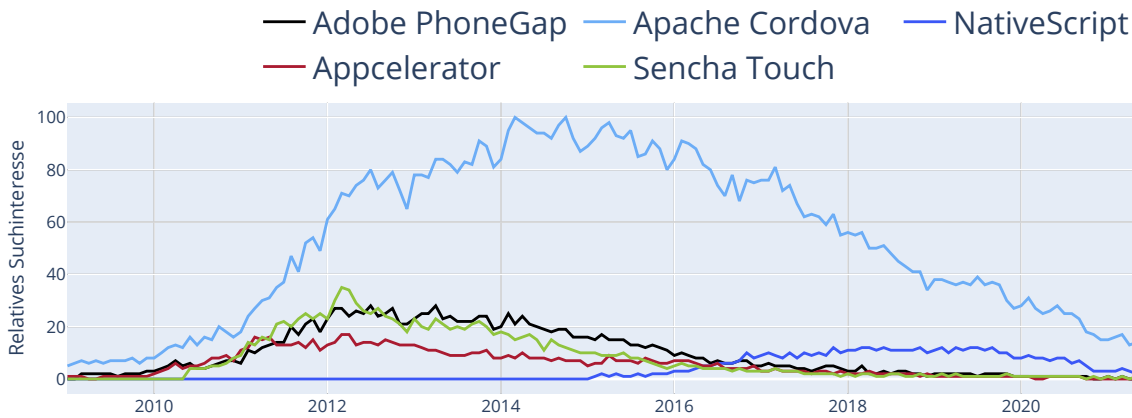


Abbildung 2.2.: Suchinteresse der Frameworks mit geringer Relevanz, Quelle: Eigene Abbildung, Notebook: [Charts/GoogleTrends/GoogleTrends.ipynb](#), Daten-Quelle: Google Trends⁵

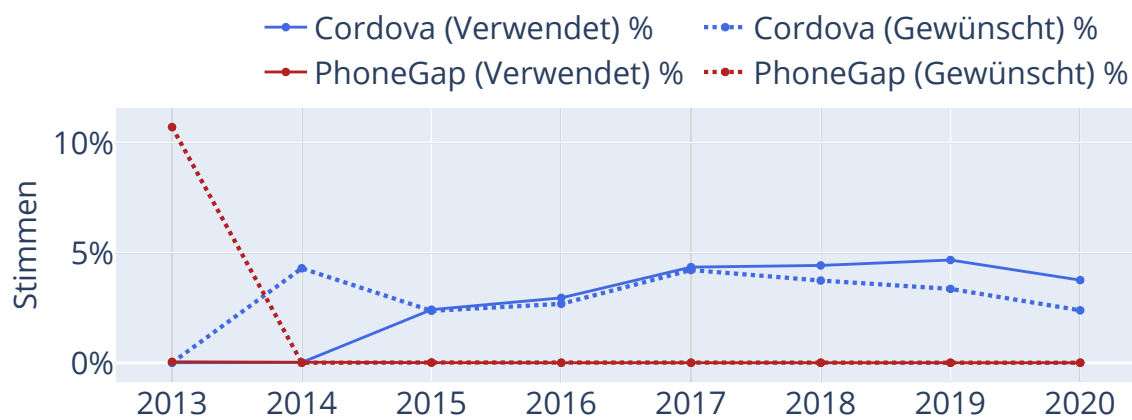


Abbildung 2.3.: Stimmen für Cordova und PhoneGap 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:

FEHLT!

geschehen ist. Adobe gab am 11. August 2020 bekannt, dass die Entwicklung an PhoneGap eingestellt wird und empfiehlt die Migration hin zu Apache Cordova.⁹

2.1.4. Frameworks mit sinkender Relevanz

Die Technologien Xamarin und Cordova zeigen bereits einen abfallenden Trend, wie in Abbildung 2.4 ersichtlich ist. Im Fall von Xamarin gibt es immerhin mehr Entwickler, die sich wünschen, mit dem Framework zu arbeiten, als Entwickler, die tatsächlich mit Xamarin arbeiten. Cordova scheint in diesem Hinblick dagegen eher unbeliebt: Es gibt mehr Entwickler, die mit Cordova arbeiten, als tatsächlich damit arbeiten wollen.

In Abbildung 2.5 ist noch einmal zu sehen, dass Google Trends die Erkenntnisse aus der Stack Overflow Umfrage reflektiert; und es wird auch sichtbar, welche beiden Technologien möglicherweise der Grund für den Rückgang von Xamarin und Cordova sind.

⁹Vgl. [2]

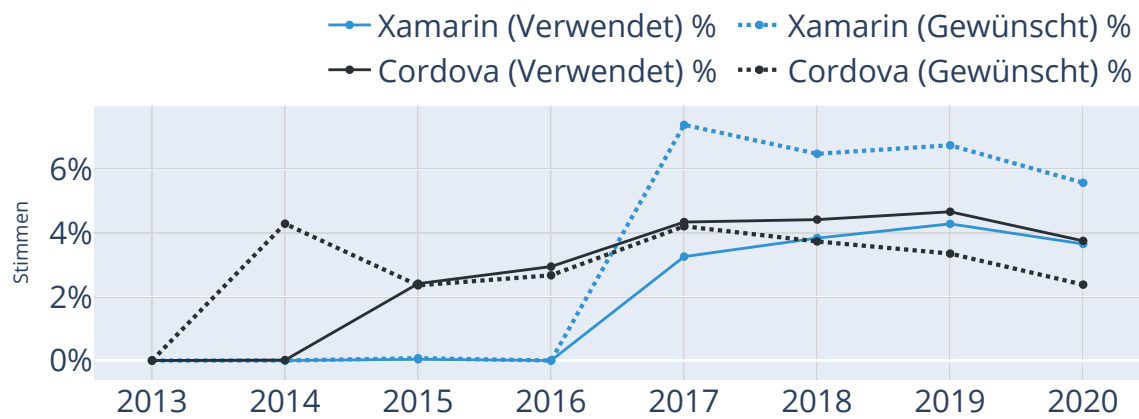


Abbildung 2.4.: Stimmen für Xamarin und Cordova 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:

FEHLT!

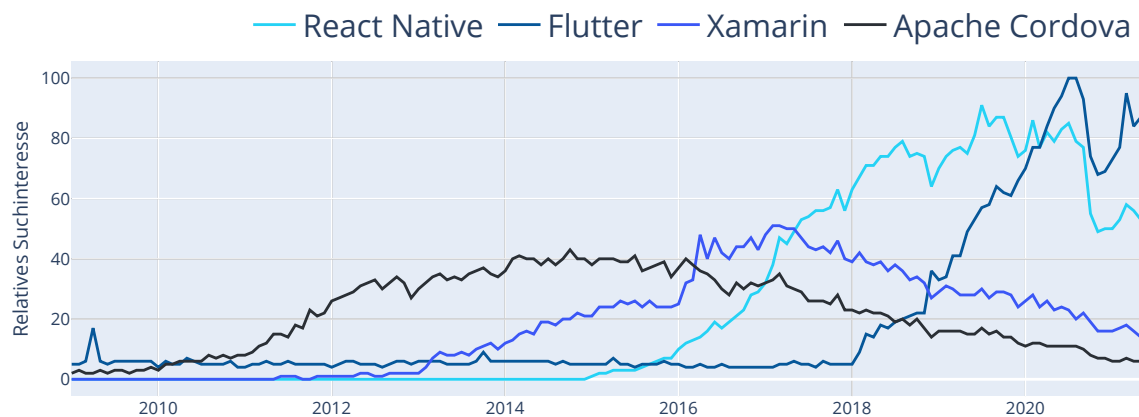


Abbildung 2.5.: Suchinteresse sinkende und steigende Relevanz, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:

FEHLT!

2.1.5. Frameworks mit steigender Relevanz

Besser ist es, auf Technologien zu setzen, die noch einen steigenden Trend der Verbreitung und Beliebtheit zeigen. In Abbildung 2.6 wird sichtbar, dass es sich dabei um Flutter und – immerhin im Hinblick auf die Verbreitung – auch um React Native handelt. Ungünstigerweise wird React Native in der Stack Overflow Umfrage erst seit 2018 als tatsächliches Framework abgefragt. Vorher erschien lediglich das Framework React, welches sich nicht für den Vergleich der Cross-Plattform-Frameworks eignet, da es sich um ein reines Web-Framework handelt. Doch auch die Ergebnisse von Google Trends zeigen einen ähnlichen Verlauf für die Jahre 2019 und 2020 (Abb. 2.5).

Im Vergleich des Jahres 2019 mit 2020 wird sichtbar, dass die Zahl der Entwickler, die sich wünschen, mit React Native zu arbeiten, gesunken ist. Dennoch ist die Anzahl der Entwickler, die mit React Native arbeiten möchten noch weit höher, als die der Entwickler, die tatsächlich mit React Native arbeiten.

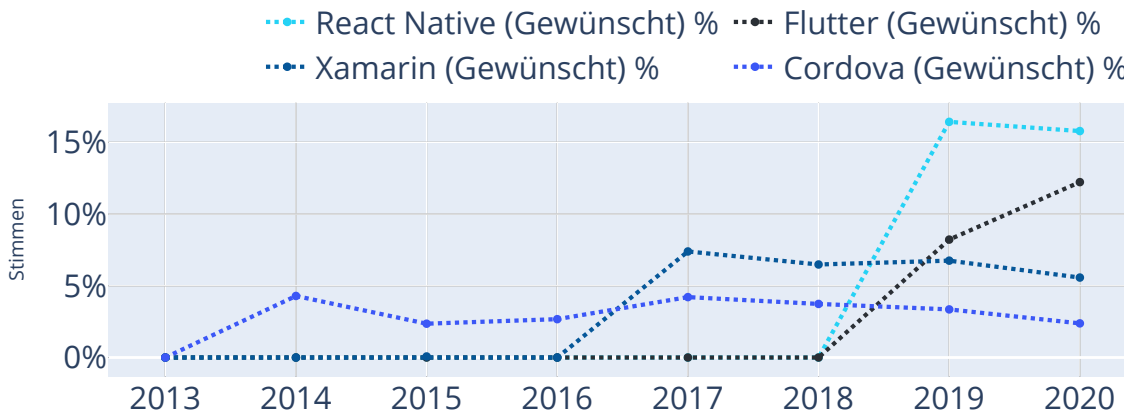


Abbildung 2.6.: Stimmen für React Native und Flutter 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:

FEHLT!

Es ist möglich, dass der abfallende Trend daran liegt, dass die Zahl der Entwickler, die mit Flutter arbeiten möchten im selben Jahr gestiegen ist. React Native hat im Vergleich zu Flutter jedoch noch immer mehr aktive Entwickler und die Tendenz ist steigend. Doch die Anzahl der aktiven Flutter Entwickler zeigt einen noch stärker steigenden Trend. So könnte es sein, dass die Zahl der Flutter Entwickler die der React Native Entwickler in einem der nächsten Jahre überholt. Im Such-Interesse hat sich diese Entwicklung bereits vollzogen (Abb. 2.5).

Nichtsdestotrotz scheinen beide Technologien als Kandidaten für einen detaillierteren Vergleich für dieses Projekt in Frage zu kommen. Im nächsten Kapitel soll evaluiert werden, welches Framework für die Entwicklung der Formular-Anwendung angemessener ist.

2.2. Vergleich von React Native und Flutter

2.2.1. Vergleich zweier minimaler Beispiele für Formulare und Validierung

verweise auf Listings Anhang, erstelle Tabelle mit Zusammenfassung

Es soll eine Formularanwendung mit komplexer Validierung im Rahmen dieser These erstellt werden. Es ist durchaus sinnvoll, die beiden Technologien anhand von Beispielanwendungen, welche Formulare und die Validierung dieser beinhalten, zu vergleichen. Deshalb soll nachfolgend jeweils eine solche Beispielanwendung der jeweiligen Technologie gefunden werden. Die Anwendungen werden sich stark voneinander unterscheiden, weshalb sie im nächsten Schritt vereinfacht und aneinander angeglichen werden. Anschließend wird ersichtlich werden, nach welchen Kriterien sich die Technologien im Hinblick auf die Entwicklung der Formularanwendung vergleichen lassen.

React Native

React native stellt nur eine vergleichsweise geringe Anzahl von eigenen Komponenten zur Verfügung und zu diesen gehören keine, welche die Validierung von Formularen ermöglichen. Doch die im react.js Raum sehr bekannten Bibliotheken Formic, Redux Forms und React Hook Form sind alle drei kompatibel mit React Native.^{10,11,12}

Für die Formularanwendung ist die Validierung komplexer Bedingungen nötig. Die Formular-Validierungs-Bibliotheken bieten in der Regel Funktionen an, welche überprüfen, ob ein Feld gefüllt ist oder der Inhalt einem speziellen Muster entspricht – wie etwa einem regulären Ausdruck. Doch solche mitgelieferten Validierungs-Funktionen reichen nicht aus, um die Komplexität der Bedingungen abzubilden. Stattdessen müssen benutzerdefinierte Funktionen zum Einsatz kommen.

Keiner der drei oben genannten Validierungs-Bibliotheken ist in dieser Hinsicht limitiert. Sie alle bieten die Möglichkeit, eine JavaScript Funktion für die Validierung zu übergeben. Diese Funktion gibt einen Wahrheitswert zurück – wahr, wenn das Feld oder die Felder valide sind, falsch, falls nicht. In React Hook Form ist es die Funktion register, die ein Parameter-Objekt namens Register Options erhält, dessen Eigenschaft validate die JavaScript Funktion zugewiesen werden kann.¹³ In Redux Form ist es die Initialisierungsfunktion reduxForm, die ein Konfigurations-Objekt mit dem Namen config erhält, in welchem die Eigenschaft ebenfalls validate heißt.¹⁴ Auch in Formic ist der Bezeichner validate, und ist als Attribut in der Formic Komponente zu finden.¹⁵

Es ist also absehbar, dass die Formular-Anwendung in React Native entwickelt werden kann. Die nötigen Funktionen werden von den Bibliotheken bereitgestellt. Einziger Nachteil hierbei ist, dass es sich um Drittanbieter Bibliotheken handelt, welche im Verlauf der Zeit an Beliebtheit gewinnen und verlieren können. Möglicherweise geht die Beliebtheit einer der Bibliotheken mit der Zeit zurück, weshalb es weniger Kontributionen wie etwa neue Funktionalitäten oder Fehlerbehebungen, sowie Fragen und Antworten und Anleitungen zu diesen Bibliotheken geben wird, da die Entwickler sich für andere Bibliotheken entscheiden. Die Wahl der Bibliothek kann also schwerwiegende Folgen wie Mangel an Dokumentation oder Limitationen im Vergleich zu anderen Bibliotheken mit sich bringen. Eine Migration von der einen Bibliothek zu einer anderen könnte in Zukunft notwendig werden, wenn diese Limitationen während der Entwicklung auffallen. Aus dem Grund ist es in der Regel von Vorteil, wenn solche Funktionalitäten bereits im Kern der Frontend-Technologie integriert sind. Der Fall, dass die Kernkomponenten an Relevanz verlieren und empfohlen wird, auf externe Bibliotheken zuzugreifen, ist zwar nicht ausgeschlossen, geschieht aber

¹⁰Vgl. [33]

¹¹Vgl. [6]

¹²Vgl. [34]

¹³Vgl. [38]

¹⁴Vgl. [37]

¹⁵Vgl. [8]

im Wesentlichen seltener.

Flutter

Die Flutter Dokumentation stellt in ihrer cookbook Sektion ein Beispiel einer minimalistischen Formularanwendung mit Validierung bereit.¹⁶ Das Rezept ist Teil einer Serie von insgesamt fünf Anleitungen, welche Formulare in Flutter behandeln.¹⁷

Auf Listing im Anhang verweisen

2.2.2. Automatisiertes Testen

Automatisierte Tests in React Native

Die React Native Dokumentation führt genau eine Seite mit einem Überblick über die unterschiedlichen Testarten. Dabei wird das Konzept von Unit Tests, Mocking, Integrations Tests, Komponenten Tests und Snapshot Tests kurz erläutert, jedoch ohne ein Beispiel zu geben oder zu verlinken. Vier Quellcodeschnipsel sind auf der Seite zu finden: Ein Schnipsel zeigt den minimalen Aufbau eines Tests; zwei weitere Schnipsel veranschaulichen beispielhaft, wie Nutzerinteraktionen getestet werden können. Letzteres zeigt die textuelle Repräsentation der Ausgabe einer Komponente, die für einen Snapshottest verwendet wird. Weiterhin wird auf die Jest API Dokumentation verwiesen, sowie auf ein Beispiel für einen Snapshot Test in der Jest Dokumentation.^I

Um die notwendigen Anleitungen für das Erstellen der jeweiligen Tests ausfindig zu machen, ist es notwendig, die Dokumentation von React Native zu verlassen.

Die Dokumentation von Jest enthält mehr Details zum Einsatz der Testbibliothek, welches für mehrere Frontend Frameworks kompatibel ist, die auf JavaScript basieren^{II}. Somit muss zum Erstellen der Unit-Tests immerhin nur dieses Framework studiert werden.

Zum Entwickeln von Tests von React Native Komponenten wird unter anderem auf die Bibliothek React Native Testing Library verwiesen. Anders als der Name vermuten lässt, handelt es sich nicht um eine von React Native bereitgestellte Bibliothek. Im Unterschied zur React Testing Library, von der sie inspiriert ist, läuft sie ebenso wie React Native selbst

¹⁶Vgl. [13]

¹⁷Vgl. [22]

^I<https://jestjs.io/docs/snapshot-testing>

^{II}<https://jestjs.io/docs/getting-started>

nicht in einer Browser-Umgebung.¹⁸ Herausgegeben wird die React Native Testing Library vom Drittanbieter Callstack – einem Partner im React Native Ökosystem.¹⁹

Sie verwendet im Hintergrund den React Test Renderer^{III}, welcher wiederum vom React Team angeboten wird und auch zum Testen von react.js Anwendungen geeignet ist. Der React Test Renderer wird ebenfalls empfohlen, um Komponententests zu kreieren, die keine React Native spezifischen Funktionalitäten nutzen.

Um Integrationstests zu entwickeln - welche die Applikation auf einem physischen Gerät oder auf einem Emulator testen - wird auf zwei weitere Drittanbieter-Bibliotheken verlinkt: Appium^{IV} und Detox^V. Es wird darauf hingewiesen, dass Detox speziell für die Entwicklung von React Native Integrationstests entwickelt wurde. Appium wird lediglich als ein weiteres bekanntes Werkzeug erwähnt.

Es lässt sich damit zusammenfassen, dass der Aufwand der Einarbeitung für automatisiertes Testen in React Native vergleichsweise hoch ist. Die Dokumentation ist auf die Seiten der jeweiligen Anbieter verteilt. Der Entwickler muss sich den Überblick selbst verschaffen und zusätzlich die für das Framework React Native relevanten Inhalte identifizieren. Notwendig ist auch das Erlernen von mehreren APIs um alle Testarten abzudecken. Für einen Anfänger kommt erschwerend hinzu, dass eine Entscheidung für die eine oder andere Bibliothek notwendig wird. Um diese Entscheidung treffen zu können, ist eine Auseinandersetzung mit den Vor- und Nachteilen der Technologien im Vorfeld vom Entwickler zu leisten.

Automatisierte Tests in Flutter

Die Flutter Dokumentation erklärt sehr umfangreich auf 11 Unterseiten die unterschiedlichen Testarten mit Quellcodebeispielen und verlinkt für jede Testart eine bis mehrere detaillierte Schritt-für-Schritt-Anleitungen, wie ein solcher Test erstellt wird.

Eine Seite erklärt den Unterschied zwischen Unit-Tests, Widget-Tests und Integrationstests^{VI}. Eine weitere Seite erklärt Integrationstests detaillierter^{VII}.

Ein sogenanntes Codelab führt durch die Erstellung einer minimalistischen App und zwei Unit-, fünf Widget- und zwei Integrationstests für diese App^{VIII}

¹⁸Vgl. [4]

¹⁹Vgl. [7]

^{III}<https://reactjs.org/docs/test-renderer.html>

^{IV}<http://appium.io/>

^V<https://github.com/wix/detox/>

^{VI}<https://flutter.dev/docs/testing>

^{VII}<https://flutter.dev/docs/testing/integration-tests>

^{VIII}<https://codelabs.developers.google.com/codelabs/flutter-app-testing>

Im sogenannten Kochbuch tauchen folgende Rezepte auf:

- 2 Rezepte für Unit Tests
 - eine grundlegende Anleitung zum Erstellen von Unit-Tests ^{IX}
 - Eine weitere Anleitung zum Nutzen von Mocks in Unit Test mithilfe der Bibliothek mockito ^X
- 3 Rezepte für Widget Tests
 - Eine grundlegende Anleitung zum Erstellen von Widget Tests ^{XI}
 - Ein Rezept mit detaillierteren Beispielen zum Finden von Widgets zur Laufzeit eines Widget Tests ^{XII}
 - Ein Rezept zum Testen von Nutzerverhalten wie dem Tab, dem Drag und dem Eingeben von Text ^{XIII}
- 3 Rezepte für Integrationstests
 - Eine grundlegende Anleitung zum Erstellen eines Integrationstests ^{XIV}
 - eine Anleitung zum Simulieren des Scrollens in der Anwendung während der Laufzeit eines Integrationstests ^{XV}
 - eine Anleitung zum Performance Profiling ^{XVI}

2.3. Fazit und Begründung der Auswahl

Zusammenfassung als Kapitel mit Tabelle und Wahl	Tabelle mit auflisting der Pros und Cons und aufsummierung
--	--

Zusammengefasst: Der Aufwand der Einarbeitung in das Testen in Flutter ist gering. Alle Werkzeuge werden vom Dart- und Flutter-Team bereitgestellt. Die Dokumentation ist

^{IX}<https://flutter.dev/docs/cookbook/testing/unit/introduction>

^X<https://flutter.dev/docs/cookbook/testing/unit/mocking>

^{XI}<https://flutter.dev/docs/cookbook/testing/widget/introduction>

^{XII}<https://flutter.dev/docs/cookbook/testing/widget/finding>

^{XIII}<https://flutter.dev/docs/cookbook/testing/widget/tap-drag>

^{XIV}<https://flutter.dev/docs/cookbook/testing/integration/introduction>

^{XV}<https://flutter.dev/docs/cookbook/testing/integration/scrolling>

^{XVI}<https://flutter.dev/docs/cookbook/testing/integration/performance>

umfangreich, folgt jedoch einem roten Faden. Eine Übersichtsseite fasst die Kerninformationen zusammen und verweist auf die jeweiligen Seiten für detailliertere Informationen und Übungen.

3. Grundlagen

Für die Formular Anwendung wurde die Programmiersprache Dart und das Oberflächen Framework Flutter gewählt. Kapitel Kapitel einfügen erläutert die Entscheidungs-Grundlage dafür.

Nachfolgend soll auf die Grundlagen der beiden Technologien eingegangen werden.

3.1. Flutter

Flutter ist ein Framework von Google zur Entwicklung von Oberflächen. Es unterstützt eine breite Anzahl an Ziel-Systemen. Dazu gehören:

- Desktop:¹
 - Windows:
 - * Win32,
 - * Universal Windows Platform,
 - macOS,
 - Linux,
- Mobile Endgeräte²:
 - Android,
 - iOS,
- und das Web³.

Flutter ist inspiriert durch das Web-Framework react und deren Oberflächenelemente, die Components genannt werden⁴. Die visuellen Oberflächen-Elemente in Flutter werden dagegen Widgets genannt. „*react*“ „*Components*“ verfügen über einen Zustand – „*State*“ genannt – der bei Veränderung das Neuzeichnen der visuellen Repräsentation erwirkt. Flutter unterscheidet allerdings zwischen zwei Arten von Widgets: denen, die einen Zustand pflegen

¹Vgl. 19.

²Vgl. 20.

³Vgl. 25.

⁴Vgl. 21.

3. Grundlagen

– den „*Stateful Widgets*“ – und solchen, die keinen Zustand haben – den „*Stateless Widgets*“.

„*Stateful Widgets*“ pflegen einen Zustand, der mittels der Methode `setState` gesetzt werden kann. Beim Aufrufen der Methode wird das gesamte Widget neu gezeichnet. Der Zustand selbst ist dabei im visuellen Baum als Vater der visuellen Elemente des Widgets verankert und bleibt erhalten, während die dazugehörigen Oberflächenelemente ausgetauscht werden.

„*Stateless Widgets*“ haben dagegen keinen solchen Mechanismus. Wie alle Widgets werden sie neu gezeichnet, wenn es durch das Framework angeordnet wurde. Das kann unter anderem der Fall sein, wenn das Widget zum ersten Mal in der Oberfläche auftaucht, oder das Vater-Element und damit alle Kinder-Elemente neu gezeichnet werden. **oder es von Inherited Widget abhängt?**

„*Stateful Widgets*“ sind nur eine von vielen Möglichkeiten den Zustand des Programms zu verwalten. Die Formular-Anwendung verwendet ausschließlich `StatelessWidget`s, da die Verwaltung des Zustands über das sogenannte BloC Pattern umgesetzt wird. Mehr dazu im Kapitel **Kapitel einfügen**. **BLoC Pattern erklären? Oder einfach bei BehaviourSubject verweisen, dass MVVM bei Google B**

Inherited Widgets?

3.2. Dart Grundlagen

Flutter-Anwendungen werden in der Programmiersprache Dart geschrieben. Nachfolgend soll auf eine Reihe von Besonderheiten von Dart im Vergleich zu anderen objektorientierten Programmiersprachen eingegangen werden.

Dart ist eine Hochsprache, die hauptsächlich für die Entwicklung von Oberflächen entwickelt wurde, sich jedoch ebenso dazu eignet, Programme für das Back-End zu entwickeln.

Ein Hauptaspekt bei dem Design der Sprache ist die Produktivität des Entwicklers. Mechanismen wie das „*hot reload*“ verkürzen die Entwicklungszyklen erheblich. Das „*hot reload*“ ermöglicht es, während eine Anwendung im Debug-Modus ausgeführt wird, Änderungen an dessen Quellcode vorzunehmen. Daraufhin werden nur die Teile der laufenden Applikation aktualisiert, die tatsächlich verändert wurden. Währenddessen bleibt die Anwendung in der gleichen Ansicht, anstatt zum Hauptbildschirm zurückgesetzt zu werden, von der aus der Entwickler erneut zur gewünschten Ansicht zurücknavigieren müsste.

3.2.1. AOT und JIT

Nicht nur für die reibungslose Entwicklung sondern auch für das Laufzeitverhalten der finalen Applikation wurde die Sprache optimiert. Für die Ziel-Architekturen ARM32, ARM64 und x86_64 wird Dart in Maschinencode kompiliert⁵.

Dementsprechend kommt während der Entwicklung eine virtuelle Maschine - die Dart VM - über Just-in-time-Kompilierung (JIT) zum Einsatz. Für die Kompilierung in Maschinencode wird dagegen Ahead-of-time-Kompilierung (AOT) eingesetzt.

tree shaking

Für die Minimierung der Dateigröße des resultierenden Kompilats wird das sogenannte „*tree shaking*“ eingesetzt. Das Hauptprogramm importiert über das Schlüsselwort `import` Funktionalitäten aus weiteren .dart-Dateien oder sogar ganzen Bibliotheken. Diese Dateien importieren wieder Weitere. Dadurch wird ein Baum aufgespannt. Das „*tree shaking*“ identifiziert, welche Funktionalitäten tatsächlich vom Programm verwendet werden und welche nicht. Dies bringt aber eine wichtige Einschränkung mit sich. Die Metaprogrammierung (der Zugriff auf sprachinterne Eigenschaften, wie etwa Klassen und ihre Attribute) ist damit stark eingeschränkt.

Meta-Programmierung

Bei der Kompilierung werden die Original-Bezeichner durch Symbole ersetzt, welche minimalen Speicherbedarf haben. Aber nicht nur das, denn durch das „*tree shaking*“ werden auch etwaige Eigenschaften und Funktionalitäten entfernt, die nicht verwendet werden. Die sogenannte „*Reflexion*“ oder „*Introspektion*“ versucht auf solche Meta-Informationen während der Laufzeit zuzugreifen. Da die Eigenschaften aber nicht mehr verfügbar sind, ist „*Reflexion*“ nicht anwendbar. Dart greift daher auf eine andere Variante der Meta-Programmierung zurück: die Quellcode Generierung.

Quellcode-Generierung

Das Package „*source_gen*“ erlaubt das Auslesen der Meta-Informationen und ermöglicht das Generieren von Quellcode, der von diesen Eigenschaften abgeleitet werden kann. So verwendet beispielsweise das Package „*built_value*“ die Quellcode-Generierung. Zunächst werden Eigenschaften wie Klassennamen und Instanzvariablen mit ihren Bezeichnern und

⁵Vgl. 18.

Datentypen gelesen. Die Eigenschaften können dann genutzt werden, um unveränderliche Werte-Typen und dazugehörige sogenannte „*Builder*“-Objekte des Erbauer-Entwurfsmusters, sowie Funktionen zum Serialisieren und Deserialisieren von Objekten zu generieren. [Referenzen](#)

3.2.2. Set und Map Literale

Dart erlaubt es Listen (`List`), Mengen (`Set`) und Hashtabellen (`Map`) als sogenannte Literale zu deklarieren. Ein Literal ist die textuelle Repräsentation eines Wertes eines speziellen Datentyps. Beispielsweise ist `"Text"` ein String-Literal für eine Zeichenkette mit den Elementen „T“, „e“, „x“, „t“. So ist auch `{"Text"}` ein Literal für eine Menge (`Set`). Eine Menge mit den gleichen Werten könnte genauso auch wie in Listing 3.1 erstellt werden.

```
1 var menge = Set();  
2 menge.add("Text");
```

Listing 3.1.: Ein Set, Quelle: Eigenes Listing

Es entfällt also die Instanziierung einer Liste, einer Menge oder einer Hashtabelle über den Klassennamen und der darauffolgenden Zuweisung der einzelnen Werte. Stattdessen startet das `Set` und `Map` Literal mit einer öffnenden geschweiften Klammer und endet mit einer schließenden geschweiften Klammer. Innerhalb der Klammern werden die Werte im Fall eines Sets mit `,` getrennt nacheinander aufgeführt (`{1,2}`). Im Fall einer Map werden der Schlüssel und der Wert durch einen `:` voneinander getrennt und die Schlüssel-Wertepaare wiederum durch `,` getrennt nacheinander aufgelistet (`{1: "erster Wert", 2: "zweiter Wert"}`). Eine Liste wiederum wird mit eckigen Klammern geöffnet und geschlossen. Die Werte werden erneut mit `,` getrennt voneinander angegeben (`[1,2]`).

Collection for

Dart erlaubt es Schleifen innerhalb von Listen-, Mengen- und Hashtabellen-Literalen zu verwenden. Dabei darf die Schleife jedoch keinen Schleifen-Körper besitzen. Lediglich der Schleifen-Kopf wird dazu im Literal geschrieben. Darauf folgt der Wert, der bei jedem Schleifendurchlauf hinzugefügt werden soll. Dabei kann der Wert von der Schleifenvariable genutzt oder davon abgeleitet werden. Listing 3.2 geht beispielsweise durch die Liste der Temperatur-Angaben 97.7, 105.8, die in Fahrenheit gelistet sind.

```
1 var gradCelsiusTemperaturen = {  
2   for (var f in [97.7, 101.3, 105.8])  
3     (f - 32) * 5 / 9  
4 };
```

Listing 3.2.: Collection-for in einer Menge, Quelle: Eigenes Listing

Für jeden Schleifendurchlauf wird die Schleifen-Variable `f` mit der entsprechenden Formel

in Grad Celsius umgewandelt. Das Ergebnis ist somit äquivalent mit dem `Set`-Literal `{36.5, 38.5, 41}`.

Gleiches gilt für Hashtabellen. Hierbei wird ein Schlüssel-Werte-Paar übergeben. Links von einem `:` ist der Schlüssel und rechts davon der Wert. In Listing 3.3 wird durch die gleiche Liste von Temperaturen in Fahrenheit iteriert.

```
1 var celsiusUndFahrenheit = {
2   for (var f in [97.7, 101.3, 105.8])
3     (f - 32) * 5 / 9 : f
4 };
```

Listing 3.3.: Collection-for in einer Hashtabelle, Quelle: Eigenes Listing

Für jede Schleifenvariable `f` wird für das resultierende Schlüssel-Wörter-Paar das Ergebnis in Grad Celsius als Schlüssel und das Ergebnis als Wert eingetragen. Das Ergebnis von `celsiusUndFahrenheit` ist dementsprechend eine `Map` mit dem Wert: `{36.5: 97.7, 38.5: 101.3, 41: 105.8}`

Collection-if

Neben dem Collection-for ist auch die Nutzung von Fallunterscheidungen in Kollektionen erlaubt. Vor dem Wert, der in die Kollektion aufgenommen werden soll oder nicht, kann das Schlüsselwort `if` mit einer darauffolgenden Bedingung in Klammern gesetzt werden. Listing 3.4 iteriert durch eine Anzahl von Temperaturen in Grad Celsius.

```
1 var fieberTemperaturen = [
2   for (var c in [36.5, 38.5, 41])
3     if (c >= 38.5) c
4 ];
```

Listing 3.4.: Collection-if in einer Liste, Quelle: Eigenes Listing

Nur in dem Fall, dass die Temperatur der Schleifen-Variable `c` größer oder gleich 38,5 ist, wird die Temperatur der Liste zugefügt. Das Ergebnis der Liste `fieberTemperaturen` ergibt also `[38.5, 41]`.

3.2.3. Typen ohne Null-Zulässigkeit

Im Vergleich zu vielen anderen Programmiersprachen - wie beispielsweise Java - wird in Dart zwischen gewöhnlichen Typen und nullable Typen unterschieden. In Java ist es nur bei atomaren Datentypen wie `int` und `float` vorgeschrieben einen Wert anzugeben. `null` ist bei diesen primitiven Datentypen nicht als Wert erlaubt. Doch nicht atomare Datentypen erlauben immer die Angabe von `null` als Wert. `null` drückt dabei immer das Nicht-Vorhandensein von Daten aus. Ab Dart 2.12 kann allen Datentypen standardmäßig kein

Null-Wert zugewiesen werden. Das hat den Vorteil, dass der Compiler sich darauf verlassen kann, dass eine Variable niemals den Wert `null` haben kann. Das ist besonders dann nützlich, wenn auf einem Objekt eine Methode aufgerufen wird. Ist das Objekt in Wahrheit `null`, so gibt es erst zur Laufzeit einen Fehler, da die Methode auf der Referenz `null` nicht aufgerufen werden kann. Damit ein Laufzeitfehler geworfen werden kann, muss vor jedem Aufruf einer Methode auf einer Referenz überprüft werden, ob die Referenzen nicht `null` sind. Würde diese Überprüfung nicht stattfinden, so könnte kein Laufzeitfehler geworfen werden und das Programm würde ohne Fehlermeldung abstürzen. Handelt es sich allerdings um eine Referenz, die niemals den Wert `null` annehmen kann, so kann der Compiler die Überprüfung auf Null-Werte für diese Referenzen überspringen. Damit erhöht sich zusätzlich die Ausführungsgeschwindigkeit, da die Überprüfung Zeit in Anspruch nimmt. Vor allem aber ist es vorteilhaft für den Entwickler, da der Compiler Fehlermeldungen und Warnungen mitteilen kann, wenn Operationen auf Variablen mit potenziellen Null-Werten verwendet werden. Die Abwesenheit von Daten ist jedoch bei der Entwicklung sehr wichtig. Nicht alle Variablen können immer einen Wert haben. Aus diesem Grund gibt es in Dart auch die Typen, die Null-Werte zulassen. Allerdings gelten besondere Regeln für diese Typen.

3.2.4. Typen mit Null-Zulässigkeit

Wird in Dart hinter einem Typen ein `?` angegeben, so kann die Variable nicht nur Werte annehmen, die dieser Datentyp zulässt, sondern zusätzlich auch noch den Wert `null`. Methoden auf Objekten mit Null-Zulässigkeit aufzurufen ist nicht ohne Weiteres möglich.

Im Listing 3.5 wird versucht, auf die Variable `fahrenheitTemperature` den Operator `-` anzuwenden, um sie mit `32` zu subtrahieren.

```
1 void printTemperatureInCelsius(int? fahrenheitTemperature) {  
2     print((fahrenheitTemperature - 32) * 5 / 9);  
3 }
```

Listing 3.5.: Collection-if in einer Liste, Quelle: Eigenes Listing

Der Compiler liefert jedoch einen Fehler, da der Wert der Variablen `null` sein kann, wie die Notation `int?` anzeigt. Solange nicht feststeht, dass die Variable zur Laufzeit tatsächlich nicht `null` ist, kann das Programm nicht kompiliert werden.

Zu diesem Zweck macht Dart von der sogenannten „*Type Promotion*“ - deutsch Typ Beförderung - Gebrauch. Mithilfe einer Fallunterscheidung kann vor Anwenden der Operation nachgesehen werden, ob der Wert der Variablen nicht `null` ist. Innerhalb des Körpers der Fallunterscheidung wird der Typ der Variablen automatisch in einen Typ ohne Null-Zulässigkeit befördert. Der Code in Listing 3.6 lässt sich daher wieder kompilieren.

Eine Besonderheit stellen dabei allerdings Instanzvariablen dar. In Dart wird syntaktisch


```

1 void printTemperatureInCelsius(int? temperature) {
2   if (temperature != null) {
3     print((temperature - 32) * 5 / 9);
4   }
5 }

```

Listing 3.6.: Collection-if in einer Liste, Quelle: Eigenes Listing

nicht zwischen dem Aufruf einer Getter-Methode oder einer Instanzvariable unterschieden. In Listing 3.7 könnte sich hinter den Aufrufen von `temperature` in den Zeilen 6 und 7 die Instanzvariable verbergen, die in Zeile 2 deklariert ist.

```

1 class Patient {
2   num? temperature;
3   Patient({this.temperature});
4
5   void printTemperatureInCelsius() {
6     if (temperature != null) {
7       print((temperature - 32) * 5 / 9);
8     }
9   }
10 }

```

Listing 3.7.: Collection-if in einer Liste, Quelle: Eigenes Listing

Genauso könnte es aber auch sein, dass eine Klasse von `Patient` erbt und das Feld `temperature` mit einer gleichnamigen Getter-Methode überschreibt. Auch wenn es sehr unwahrscheinlich ist, könnte es trotzdem vorkommen, dass der Aufruf von `temperature` in Zeile 6 einen Wert zurückgibt, der nicht `null` ist und der darauffolgende Aufruf in Zeile 7 `null` liefert. So provoziert es die Klasse `UnusualPatient` im Listing 3.8.

```

1 class UnusualPatient extends Patient {
2   int counter = 0;
3
4   num? get temperature {
5     counter++;
6     if (counter.isOdd) {
7       return 97.7;
8     } else {
9       return null;
10    }
11  }
12 }

```

Listing 3.8.: Collection-if in einer Liste, Quelle: Eigenes Listing

Beim ersten Aufruf von `temperature` wird die Zähl-Variable `counter` von 0 auf 1 erhöht. Die Abfrage, ob es sich bei dem Wert von `counter` um eine ungerade Zahl handelt, ist erfolgreich (Z. 6), weshalb mit 97,7 ein valider Wert zurückgegeben wird. Beim zweiten Aufruf erhöht sich `counter` allerdings auf 2. Die gleiche Abfrage schlägt dieses Mal fehl. Deshalb liefert die Getter-Methode nun `null` (Z. 9). Ein solches Szenario ist schon sehr unwahrscheinlich, doch die Typ-Überprüfung des Compilers arbeitet mit Beweisen. Im Fall von Instanzvariablen kann nicht bewiesen werden, dass zur Laufzeit ein solcher Fall

ausgeschlossen werden kann.

Sollte sich der Entwickler sicher sein, dass die Variable nicht `null` sein kann, so kann er mit einem nachgestellten `!` erzwingen, dass die Variable als nicht `null` angesehen wird (Listing 3.9, Z. 3).

```
1 void printTemperatureInCelsius() {  
2     if (temperature != null) {  
3         print((temperature! - 32) * 5 / 9);  
4     }  
5 }
```

Listing 3.9.: Collection-if in einer Liste, Quelle: Eigenes Listing

Sollte es dann dennoch passieren, dass die Variable `null` ist, so wird eine Fehlermeldung beim Aufruf der Variable geworfen.

Eine noch sicherere Variante ist es, die Instanzvariable zuvor in eine lokale Variable zu speichern (Listing 3.10, Z. 2).

```
1 void printTemperatureInCelsius() {  
2     num? temperature = this.temperature;  
3     if (temperature != null) {  
4         print((temperature - 32) * 5 / 9);  
5     }  
6 }
```

Listing 3.10.: Collection-if in einer Liste, Quelle: Eigenes Listing

Die lokale Variable hat keine Möglichkeit zwischen den zwei Aufrufen einen unterschiedlichen Wert anzunehmen. Somit kann auch das Suffix `!` weggelassen werden (Z. 4).

3.2.5. Asynchrone Programmierung

Wird auf eine externe Ressource zugegriffen - wie zum Beispiel das Abrufen einer Information von einem Webserver, oder das Lesen einer Datei im lokalen Dateisystem - so handelt es sich um asynchrone Operationen.

Im Sprachkern stellt Dart Schlüsselwörter und Datentypen für die asynchrone Programmierung bereit. Das sind unter anderem die Datentypen `Future` und `Stream` sowie die Schlüsselwörter `async` und `await`.

Future

Ein `Future`-Objekt repräsentiert einen potenziellen einmaligen Wert, der erst in der Zukunft bereit steht. Er gleicht damit dem sogenannten `Promise` - deutsch Versprechen – in

JavaScript⁶.

Das Listing 3.11 zeigt mit dem Lesen einer Datei ein Beispiel für den Aufruf einer asynchronen Operation.

```
1 var fileContent = file.readAsString();
```

Listing 3.11.: Collection-if in einer Liste, Quelle: Eigenes Listing

Anders als erwartet, befindet sich in der Variablen `fileContent` in Wahrheit kein Text mit dem Inhalt der Datei. Stattdessen hat die Variable den Datentyp `Future<String>` und ist lediglich ein sogenannter „*Handle*“ - deutsch Referenzwert - für das potenzielle und zukünftige Ergebnis der Operation.

Mit der Übergabe einer Funktion, die bei Vollendung der Operation aufgerufen wird, kann der Wert ausgewertet werden. Man nennt diese Operation auch „*Callback-Funktion*“ - deutsch Rückruffunktion. Listing 3.12 zeigt, wie auf den Dateiinhalt zugegriffen werden kann.

```
1 fileContent.then((text) {
2   print("Der Datei-Inhalt ist: $text");
3 });
```

Listing 3.12.: Collection-if in einer Liste, Quelle: Eigenes Listing

Über die Methode `then` wird eine Funktion übergeben, die genau einen Parameter hat. In diesem Parameter wird der Text der gelesenen Datei bei Vollendung der Operation übergeben.

Der Einsatz von „*Callback-Funktionen*“ kann den Quellcode stark verkomplizieren. Man spricht von der sogenannten „*callback hell*“ - deutsch Rückruffunktionen-Hölle -, wenn solche „*Callback-Funktionen*“ über etliche Level hinweg ineinander verschachtelt sind.

Um genau das zu verhindern, existieren in Dart die Schlüsselwörter `async` und `await`. Genauso heißen sie auch in anderen Sprachen wie etwa C# ab Version 4.5 und JavaScript ab Version ES2017^{7,8}.

Listing 3.13 zeigt, dass das Anwenden des Schlüsselwortes `await` vor der Operation `file.readAsString` dafür sorgt, dass der zukünftige Wert direkt in `fileContent` gespeichert wird.

Ganz ohne „*Callback-Funktion*“ kann der Dateiinhalt in der darauffolgenden Zeile ausgegeben werden.

⁶Vgl. 31.

⁷Vgl. 32.

⁸Vgl. 5.

```
1 printFileContent() async {  
2   var fileContent = await file.readAsString();  
3   print("Der Datei-Inhalt ist: $fileContent");  
4 }
```

Listing 3.13.: Collection-if in einer Liste, Quelle: Eigenes Listing

Doch jede Funktion, die auf andere Funktionsaufrufe wartet, muss selbst als asynchron gekennzeichnet werden. Dazu dient das `async` Schlüsselwort vor Beginn des Methodenkörpers.

Streams

Streams liefern nicht nur einen Wert – wie im Fall eines `Future` – sondern eine Serie von Werten, die in der Zukunft geliefert werden. Listing 3.14 zeigt wie auf einen solchen Stream gehorcht werden kann.

```
1 var countStream = Stream<num>.periodic(const Duration(seconds: 1), (count) {  
2   return count;  
3 });  
4  
5 countStream.listen((count) {  
6   print("Gezählte Sekunden: $count");  
7 });
```

Listing 3.14.: Collection-if in einer Liste, Quelle: Eigenes Listing

`countStream` liefert jede Sekunde einen neuen Wert, nämlich die aktuelle Sekunde - von 0 beginnend. Mit `countStream.listen` kann eine Funktion übergeben werden, die immer dann ausgeführt wird, wenn dem `countStream` ein neuer Wert hinzugefügt wurde. Der erste Parameter ist dabei der hinzugefügte Wert.

Es wird zwischen zwei Arten von Streams unterschieden. Solche, die genau einen Empfänger haben - „*single subscription streams*“ - und solche, die beliebig viele Empfänger haben können - „*broadcast streams*“.

Für die Formularanwendung sind ausschließlich „*broadcast streams*“ zu berücksichtigen. Die Streams sollen verwendet werden, um Änderungen in der Eingabemaske zu behandeln. Die Oberflächenelemente horchen auf diese Änderungen. Teile der Oberfläche und damit die Oberflächenelemente, welche auf die Streams horchen, werden immer wieder neu gezeichnet. Dabei werden die Elemente entfernt und durch neu konstruierte ersetzt. So melden sich immer wieder Zuhörer vom „*Stream*“ ab und neue Elemente melden sich an. Aufgrund dessen kommen nur „*broadcast streams*“ infrage.

Teil III

IMPLEMENTIERUNG

4. Schritt 1 - Formular in Grundstruktur erstellen

Im ersten Schritt soll die Formular-Anwendung in ihrer Grundstruktur entwickelt werden. Das beinhaltet alle drei Oberflächen, welche in den darauf folgenden Schritten lediglich erweitert werden. Das Formular erhält noch keine Validierung. Somit sind alle Eingaben oder nicht kompatible Selektionen erlaubt. Die erste Ansicht, welche der Benutzer sieht, soll die Übersicht der bereits eingetragenen Maßnahmen sein (Abb. 4.1).



Zuletzt bearbeitet am	Maßnahmentitel
2021-8-4 13:20	Maßnahme 1

Zuletzt bearbeitet am	Maßnahmentitel
2021-8-4 13:17	Maßnahme 2

Abbildung 4.1.: Der Übersicht-Bildschirm zeigt in Schritt 1 zunächst nur die Maßnahmen mit ihrem Titel und Bearbeitungsdatum in den Kategorien „Abgeschlossen“ und „In Bearbeitung“, Quelle: Eigene Abbildung

Die Auflistung der Maßnahmen erfolgt in den Kategorien „In Bearbeitung“ und „Abgeschlossen“. Innerhalb dieser Rubriken werden die Maßnahmen in einer Tabelle angezeigt. Mit einem Klick auf den Button unten rechts im Bild wird der Benutzer auf die zweite Ansicht weitergeleitet: die Eingabemaske (Abb. 4.2).

Sie ermöglicht die Eingabe des Maßnahmen-Titels über ein simples Eingabefeld. Darüber hinaus ist die Selektions-Karte für den Status zu sehen. Mit einem Klick auf diese Karte öffnet sich der Selektions-Bildschirm. Er ermöglicht die Auswahl der Auswahloptionen, in diesem Fall die Optionen „in Bearbeitung“ und „abgeschlossen“ (Abb. 4.3).

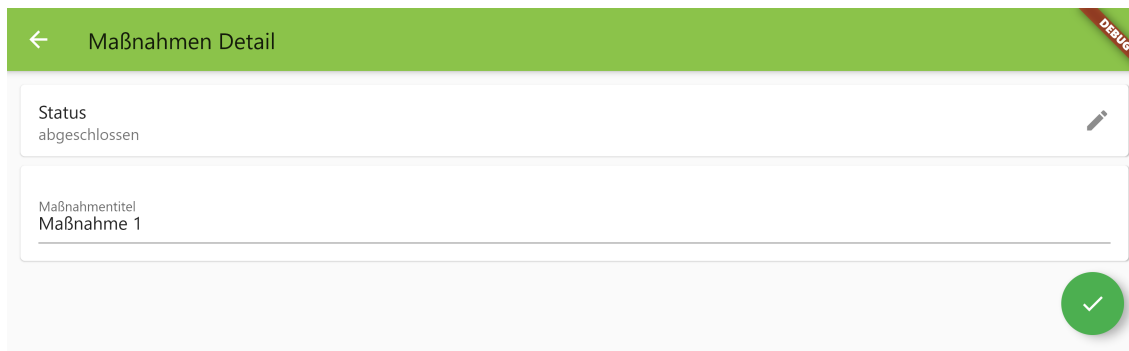


Abbildung 4.2.: Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung



Abbildung 4.3.: Der Selektions-Bildschirm für das Feld Status erlaubt die Auswahl der Optionen „in Bearbeitung“ und „abgeschlossen“, Quelle: Eigene Abbildung

4.1. Auswahloptionen hinzufügen

Dart verfügt – anders als beispielsweise Java¹ – nicht über Aufzählungstypen mit zusätzlichen Eigenschaften. Das Schlüsselwort `enum` in Dart erlaubt lediglich die Auflistung konstanter Symbole². Für die Auswahloptionen ist es jedoch notwendig, dass es zwei Eigenschaften gibt:

- die Abkürzung, die in der resultierenden Datei gespeichert werden soll
- und der Beschreibungstext, welcher in der Oberfläche angezeigt wird.

Das hat den Hintergrund, dass die Abkürzungen weniger Speicherplatz einnehmen und die Beschreibung sich in Zukunft auch ändern darf. Würde anstatt der Abkürzung die Beschreibung als Schlüssel verwendet werden, so würde eine Datei, die mit einer älteren Version des Formulars erstellt wurde, nicht mehr von neueren Versionen der Applikation eingelesen werden können. Der alte Beschreibungstext würde nicht mehr mit dem Text übereinstimmen, der als Schlüssel in der Anwendung verwendet wird.

Die beiden Zustände „in Bearbeitung“ und „abgeschlossen“ werden daher in Listing ?? als statische Klassenvariablen deklariert (Z. 6-7). Die beiden Konstruktor-Aufrufe übergeben dabei als erstes Argument die Abkürzung und als zweites Argument die Beschreibung. Der Konstruktor selbst (Z. 9-10) deklariert die beiden Parameter als positionale Parameter.

¹Vgl. 26, S. 321.

²Vgl. 17, S. 74f.


```

5 class LetzterStatus extends Choice {
6   static final bearb = LetzterStatus("bearb", "in Bearbeitung");
7   static final fertig = LetzterStatus("fertig", "abgeschlossen");
8
9   LetzterStatus(String abbreviation, String description)
10    : super(abbreviation, description);
11 }

```

Listing 4.1.: Die Klasse `LetzterStatus`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/choices.dart](#)

Positionale Parameter Im Vergleich zu den benannten Parametern ist bei den positionalen Parametern nur ihre Reihenfolge in der Parameterliste ausschlaggebend. Das Argument für die `abbreviation` steht dabei also immer an erster Stelle und das Argument für `description` immer an der zweiten (Z. 6-7). Positionale Parameter sind vorgeschrieben. Werden sie ausgelassen, so gibt es einen Compilerfehler.³

Die Klasse `LetzterStatus` erbt von der Basisklasse `Choice` (Z. 5). Der Konstruktor der Klasse (Z. 9) übergibt beide Parameter als Argumente an den Konstruktor der Klasse `Choice`. Genau wie in Java wird mithilfe des Schlüsselwortes `super` (Z. 10) der Konstruktor der Basisklasse aufgerufen. Doch anders als in Java erfolgt der Aufruf des `super` Konstruktors nicht in der ersten Zeile des Konstruktor-Körpers⁴. Weil das Aufrufen des Konstruktors der Basisklasse zum statischen Teil der Objekt-Instanziierung gehört, muss der Aufruf von `super` in der Initialisierungsliste erfolgen. Die Initialisierungsliste wird mit dem `:` nach der Parameterliste eingeleitet (Z. 10)⁵.

Die Basisklasse `Choice` (Listing 4.2) deklariert lediglich die beiden Felder `description` und `abbreviation` jeweils als `String` (Z. 4-5). Beide sind mit `final` gekennzeichnet, was sie zu unveränderlichen Instanzvariablen macht. Nach der Initialisierung können sie keine anderen Werte annehmen.⁶ Die Initialisierung der beiden Variablen muss im statischen Kontext der Instanziierung erfolgen. Mit der abgekürzten Schreibweise `this.abbreviation` und `this.description` im Konstruktor (Z. 7) werden die Parameter den Feldern zugewiesen.

```

3 class Choice {
4   final String description;
5   final String abbreviation;
6
7   const Choice(this.abbreviation, this.description);

```

Listing 4.2.: Die Klasse `Choice`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/base/choice.dart](#)

Dies erübrigt sowohl die Angabe des Parametertyps mittels `(String abbreviation, String description)`, denn der Typ des Parameters kann bereits durch Angabe des Typs in der Instanzvariablen-Deklaration (Z. 4-5) abgeleitet werden. Außerdem entfällt auch die Zuweisung, die man an-

³Vgl. 17, S. 74f.

⁴Vgl. 26, S. 310.

⁵Vgl. 17, S. 42.

⁶Vgl. 17, S. S16.

sonsten in der Form `this.abbreviation = abbreviation` und `this.description = description` in der Initialisierungsliste erreichen würde.⁷

Die Variable `letzterStatusChoices` (Listing 4.3, Z. 13) fasst die beiden statischen Klassenvariablen als eine Kollektion zusammen. Da es sich um eine solche Kollektion handelt, in der jedes Element nur ein einziges Mal vorkommen darf, ist hier von einer Menge zu sprechen. Auffällig hierbei ist, dass das Schlüsselwort `new` fehlt. In Dart ist das Schlüsselwort für die Konstruktion von Instanzen optional. Die Klasse, die zur Konstruktion dieser Menge verwendet wird, ist die selbst erstellte Klasse `Choices`. Über das Typargument `LetzterStatus` wird erreicht, dass ausschließlich Variablen dieses Typs in der Menge eingefügt werden dürfen. Wird stattdessen eine Variable eingefügt, die weder vom selben Typ, noch von einem Typ, der von `letzter Status` erbt, so gibt es einen Compilerfehler. Dies dient einzig und allein dem Zweck, dem Fehler vorzubeugen, dass aus Versehen falsche Optionen in der Menge eingetragen werden. Über den Parameter `name` ist es möglich, dieser Menge die Beschriftung `"Status"` hinzuzufügen. Es handelt sich hier um einen benannten Parameter.

```
13 final letzterStatusChoices = Choices<LetzterStatus>(  
14     {LetzterStatus.bearb, LetzterStatus.fertig},  
15     name: "Status");
```

Listing 4.3.: Die Menge `letzterStatusChoices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/choices.dart](#)

⁷Vgl. 17, S. 40f.

Listing 4.4 zeigt die Klasse `Choices`. Sie erbt von `UnmodifiableSetView` und erlaubt damit die Erstellung einer eigenen Menge - auch `Set` genannt **Referenz**. Methoden, die man von einem `Set` erwartet, lassen sich somit direkt auf Instanzen der Klasse `Choices` aufrufen. Darunter unter anderem die `contains` Methode, welche erlaubt, das Vorhandensein eines Objekts im `Set` zu überprüfen **Referenz**.

```

10 class Choices<T extends Choice> extends UnmodifiableSetView<T> {
11     final String name;
12     final Map<String, T> choiceByAbbreviation;
13
14     T? fromAbbreviation(String? abbreviation) => choiceByAbbreviation[abbreviation];
15
16     Choices(Set<T> choices, {required this.name})
17         : choiceByAbbreviation = {
18         for (var choice in choices) choice.abbreviation: choice,
19         },
20         super(choices);
21 }

```

Listing 4.4.: Die Klasse `Choices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/base/choice.dart](#)

Instanzvariable `name` (Z. 11) wird im Konstruktor 16 zugewiesen. Auffällig hierbei ist, dass der Parameter in geschweiften Klammern geschrieben steht und das Schlüsselwort `required` vorangestellt ist. Das macht den Parameter zu einem vorgeschriebenen benannten Parameter.

Vorgeschriebene benannte Parameter Gewöhnliche benannte Parameter sind optional. Wird ihnen das Schlüsselwort `required` vorangestellt, so müssen sie gesetzt werden, denn sonst gibt es einen Compilerfehler. An dieser Stelle ist das `required` Schlüsselwort sinnvoll, denn es handelt sich um den Datentyp `String`, der nicht den Wert `null` annehmen kann. Würde der Parameter aber optional sein, so wäre es möglich, das Programm zu kompilieren, auch wenn bei Aufrufen des Konstruktors kein Argument für den Parameter übergeben wurde. Doch in diesem Fall gäbe es keinen Initialwert für `name` und somit müsste der Instanzvariablen `null` zugewiesen werden. In der statischen Analyse wird daher sichergestellt, dass Instanzvariablen durch benannte Parameter nur dann mit absoluter Sicherheit initialisiert werden, wenn diese durch `required` als verpflichtend gekennzeichnet sind und damit unter keinem Umstand ausgelassen werden können. **Kürzer und einfacher** Dürfte `name` den Wert `null` annehmen, so würde es sich um den Datentyp `String` mit Null-Zulässigkeit – also mit der Notation `String?` – handeln.

Neben `name` wird mit `choiceByAbbreviation` eine weitere Instanzvariable deklariert (Z. 12). Es handelt sich um den Datentyp `Map` - eine Kollektion die Daten mittels Schlüssel-Werte-Paaren ablegen kann. Als Schlüssel wird die Abkürzung mit dem Datentyp `String` verwendet. Als Wert ist der generische Typ-Parameter `T` angegeben. Er ist in Zeile 10 deklariert und muss mindestens von der Klasse `Choice` erben. In `choiceByAbbreviation` werden also die Auswahlmöglichkeiten über ihre Abkürzung abgelegt und können über dieselbe wieder

referenziert werden. Da es sich auch hier um eine unveränderliche Instanzvariable handelt, muss sie schon in der Initialisierungsliste initialisiert werden (Z. 17-19). Dabei wird zunächst mit der öffnenden geschweiften Klammer (Z. 17) ein sogenanntes Literal einer Map begonnen, welches mit einer schließenden geschweiften Klammer (Z. 19) endet. Mehr zu Map Literalen in dem Grundlagenkapitel **Kapitel einfügen**.

Auffällig ist jedoch, dass in Zeile 18 dem Set lateral keine einfache Auflistung von Werten übergeben wird. Stattdessen wird das mit dem sogenannten „*collection for*“ eine Wiederholung verwendet.

In Zeile 18 wird durch die Menge aller Auswahloptionen `choices` iteriert und dabei in jedem Schleifendurchlauf die Auswahloption in die Variable `choice` gespeichert. Während des Schleifendurchlaufs wird dann ein Schlüssel-Werte-Paar gebildet, wobei `choice.abbreviation` der Schlüssel ist und das Objekt `choice` der Wert.

Die Map `choiceByAbbreviation` erlaubt es nach der Initialisierung mit Hilfe der Methode `fromAbbreviation` (Z. 14) über die Abkürzung das dazugehörige `Choice`-Objekt abzurufen. Beispielsweise gibt der Befehl `letzterStatusChoices.fromAbbreviation("fertig")` das Objekt `LetzterStatus("fertig", "abgeschlossen")` zurück. Auffällig dabei ist, dass der Parameter `abbreviation` mit dem Typ `String?` und der generische Rückgabetypp mit `T?` gekennzeichnet ist. Der Suffix `?` macht beide zu Typen mit Null-Zulässigkeit.

Die Methode `fromAbbreviation` soll für die Deserialisierung genutzt werden. Sollten im Formular Auswahlfelder leer gelassen worden sein, so haben entsprechende Variablen den Wert `null`. Wenn nun das Formular abgespeichert wird, so tauchen auch in der abgespeicherten JSON-Datei keine Werte für das Feld auf. Aus der JSON-Datei werden ausschließlich die Abkürzungen der Auswahloptionen gelesen. Die Methode `fromAbbreviation` wandelt sie wieder in die entsprechenden Objekte des Datentyps `Choice` um. Sollte jedoch kein Wert hinterlegt sein, so wird `letzterStatusChoices.fromAbbreviation(null)` aufgerufen. Dadurch wird klar, dass der Parameter `null` zulassen muss. Es impliziert auch, dass potenziell `null` zurückgegeben werden kann, da für den Schlüssel `null` kein Wert in der Map hinterlegt sein kann. Deshalb erlaubt auch der Rückgabetypp `T?` Null-Werte.

4.2. Serialisierung einer Maßnahme

Damit die Daten angezeigt und verändert werden können, müssen sie zunächst serialisierbar sein, sodass sie auf einen Datenträger geschrieben und von dort auch wieder gelesen werden können. Die zwei bekanntesten Bibliotheken zum Serialisieren in Dart heißen `json_serializable` und `built_value`. Beide haben gemeinsam, dass sie Quellcode generieren, welcher die Umwandlung der Objekte in JSON übernimmt. „*built_value*“ bietet im Gegensatz zu „*JSON Serializable*“ jedoch die Möglichkeit unveränderbare Werte-Typen – sogenannte „*immutable value types*“ – zu erstellen. Da diese unveränderbaren Werte noch bei der Erstellung des sogenannten ViewModels – mehr dazu im Kapitel **Einfügen** – hilfreich werden, wurde sich für diese Bibliothek entschieden.

Ein Werte-Typ für „*built_value*“ erfordert einige Zeilen Boilerplate-Code, um den generierten Quellcode mit der selbstgeschriebenen Klasse zu verknüpfen. Entwicklungsumgebungen wie Visual Studio Code und Android Studio erlauben solchen Boilerplate Code generieren zu lassen und dabei nur die erforderlichen Platzhalter einzugeben. In Visual Studio Code werden diese Templates „*Snippets*“ genannt, in Android Studio heißen sie „*Live Templates*“. Listing 4.5 zeigt, wie das live Template für das Generieren eines Wertetyps für `built_value` aussieht. Templates für „*built_value*“ wie dieses und weitere müssen nicht vom Nutzer eingegeben werden, sondern existieren bereits als Plugin für die beiden Entwicklungsumgebungen^{XVII}, ^{XVIII}.

```

6  part '$file_name$.g.dart';
7
8  abstract class $ClassName$ implements Built<$ClassName$, $ClassName$Builder> {
9      $todo$
10
11      $ClassName$. _();
12      factory $ClassName$([void Function($ClassName$Builder) updates]) = _$$$ClassName$;
13  }

```

Listing 4.5.: Live Template für die Erstellung von `built_value` Boilerplate-Code in Android Studio, Quelle: JetBrains Marketplace Built Value Snippets Plugin

`$ClassName$` Wird dabei jeweils durch den gewünschten Klassennamen ersetzt. Android Studio erlaubt, dass beim Einfügen des „*live templates*“ der Klassename einmalig eingegeben werden muss. Anschließend wird mithilfe des „*live templates*“ der Boilerplate Code generiert.

In Listing 4.6 ist der Werte-Typ `Massnahme` zu sehen. Die Zeilen 11 bis 13, sowie 23 bis 28 wurden dabei automatisch erstellt. Die Zeilen 14 bis 21 wurden hinzugefügt. Zunächst soll die Maßnahme über die `guid` eindeutig identifiziert werden können.

^{XVII}<https://plugins.jetbrains.com/plugin/13786-built-value-snippets>

^{XVIII}<https://marketplace.visualstudio.com/items?itemName=GiancarloCode.built-value-snippets>

```

6 part 'massnahme.g.dart';
7
8 abstract class Massnahme implements Built<Massnahme, MassnahmeBuilder> {
9   String get guid;
10
11   LetzteBearbeitung get letzteBearbeitung;
12
13   Identifikatoren get identifikatoren;
14
15   static void _initializeBuilder(MassnahmeBuilder b) =>
16     b..guid = const Uuid().v4();
17
18   Massnahme._();
19
20   factory Massnahme([void Function(MassnahmeBuilder) updates]) = _$Massnahme;
21
22   static Serializer<Massnahme> get serializer => _$massnahmeSerializer;
23 }

```

Listing 4.6.: Der Werte-Typ `Massnahme`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Globally Unique Identifier Ein GUID – Kurzform von Globally Unique Identifier – ist eine Folge von 128 Bits, die zur Identifikation genutzt werden kann. Eine solche GUID hat eine textuelle Repräsentation wie beispielsweise die folgende: `'f81d4fae-7dec-11d0-a765-00a0c91e6bf6'`

Die Attribute `letzteBearbeitung` und `identifikatoren` sind im Gegensatz zu dem String-Attribut `guid` zusammengesetzte Datentypen, die im Folgenden weiter beleuchtet werden.

Auffällig ist, dass es sich hier um eine abstrakte Klasse handelt und die drei Attribute jeweils Getter-Methoden ohne Implementierung sind. Eine solche Getter-Methode speichert keinen Wert, sondern gibt lediglich den Wert eines Feldes zurück. Die dazugehörigen Felder, Setter-Methoden, die konkrete Klasse und der restliche generierte Code ist in der gleichnamigen Datei mit der Endung `.g.dart` (Zeile 11) zu finden.

Die Klassen-Methode `_initializeBuilder` kann in jedem Werte-Typ hinterlegt werden, um Standardwerte für Felder festzulegen. Die Methode wird intern von „*built_value*“ aufgerufen. Bei dem Feld „*guid*“ handelt es sich um einen String, der keine Null-Werte zulässt. Könnte das Feld auch Null-Werte annehmen, so wäre die Notation in Dart stattdessen `String? get guid;` „*built_value*“ erwartet also immer einen Wert für dieses Feld. Sollte die Datei gelesen werden, welche die Maßnahmen enthält, so enthält jede Maßnahme bei der Deserialisierung den abgespeicherten Wert für die `guid` und somit wird das Feld gefüllt. Doch sollte eine leere Maßnahme über einen Konstruktor erstellt werden, so wäre das Feld „*guid*“ leer und „*built_value*“ würde einen Fehler auslösen. Aus diesem Grund wird in der Zeile 21 für das Feld `guid` ein Standardwert festgelegt: nämlich eine zufällige generierte ID, die dem Standard Uuid der Version 4 entspricht. Zu diesem Zweck wird das „*Builder*“-Objekt verwendet. Die Klasse `MassnahmeBuilder` gehört dabei zu dem von „*built_value*“ generierten Quellcode. Der Parametername wird hier – wie so häufig im builder pattern –

mit einem `b` für „*Builder*“ abgekürzt. Die Syntax `=>` leitet einen sogenannten „*arrow function body*“ ein. Dabei handelt es sich schlicht um einen Funktions-Körper, der genau eine Anweisung ausführt. Deshalb muss er nicht von geschweiften Klammern umgeben werden.⁸ Auf dem „*Builder*“-Objekt können dann die Eigenschaften so gesetzt werden, als wären sie die Eigenschaften von dem Objekt `Massnahme`. In Wahrheit werden sie aber nur auf den „*Builder*“-Objekt angewendet. Ebenfalls auffällig ist die Syntax `b..guid`. Statt dem Punkt zum Zugriff auf Attribute des Objekts wird hier der sogenannte Kaskadierungs-Operator benutzt.

Der Kaskadierungs-Operator Durch Eingabe von zwei aufeinanderfolgenden Punkten `..` statt nur einem `.` können mehrere Operationen an einem Objekt ausgeführt werden, ohne das Objekt zuvor einer Variablen zuzuweisen oder die Operationen über dessen Namen wiederholt aufzurufen.⁹ Beispiel: die Aufrufe `objekt.tueEtwas();` `objekt.tueEtwasAnderes();` und `objekt..tueEtwas()..tueEtwasAnderes();` sind äquivalent.

Da der Kaskadierungs-Operator jedoch dazu verwendet wird, mehrere Operationen auf einem Objekt auszuführen, hat er in Zeile 16 keine Funktion. Doch bei Änderung eines Objekts über das „*Builder Pattern*“ werden für gewöhnlich mehrere Operationen am gleichen „*Builder*“-Objekt ausgeführt, weshalb der Einheitlichkeit wegen der Kaskadierungs-Operator immer im Zusammenhang mit dem „*Builder*“-Objekt verwendet werden soll.

Die Attribute `letzteBearbeitung` und `identifikatoren` (Z. 11, 13) erhalten dagegen ganz automatisch Standardwerte in Form von Instanzen der dazugehörigen Klassen. Diese wiederum konfigurieren ihre eigenen Felder und deren initiale Werte.

Der Werte-Typ `Identifikatoren` ist in Listing 4.7 zu sehen. Er enthält das Attribut `massnahmenTitel`, welcher im Eingabeformular durch das Texteingabefeld gefüllt werden wird.

```

25 abstract class Identifikatoren
26     implements Built<Identifikatoren, IdentifikatorenBuilder> {
27     String get massnahmenTitel;
28
29     static void _initializeBuilder(IdentifikatorenBuilder b) =>
30         b..massnahmenTitel = "";

```

Listing 4.7.: Der Werte-Typ `Identifikatoren`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Schließlich enthält der Werte-Typ `LetzteBearbeitung` in Listing 4.8 noch die Attribute `letztesBearbeitungsDatum` in Zeile 43 und `letzterStatus` in Zeile 50. Im Eingabeformular wird der Selektions-Bildschirm den Inhalt des Feldes `letzterStatus` bestimmen. Der initiale Wert wird in Zeile 54 auf einen konstanten Wert gesetzt, der dem Zustand `'in Bearbeitung'` entspricht - mehr dazu im Kapitel **Kapitel einfügen**.

⁸Vgl. 17, S. 18f., 234.

⁹Vgl. 17, S. 149f.

4. Schritt 1 - Formular in Grundstruktur erstellen

```
41 abstract class LetzteBearbeitung
42     implements Built<LetzteBearbeitung, LetzteBearbeitungBuilder> {
43     DateTime get letztesBearbeitungsDatum;
44
45     String get formattedDate {
46         final date = letztesBearbeitungsDatum;
47         return "${date.year}-${date.month}-${date.day} ${date.hour}:${date.minute}";
48     }
49
50     String get letzterStatus;
51
52     static void _initializeBuilder(LetzteBearbeitungBuilder b) => b
53         ..letztesBearbeitungsDatum = DateTime.now().toUtc()
54         ..letzterStatus = LetzterStatus.bearb.abbreviation;
```

Listing 4.8.: Der Werte-Typ `LetzteBearbeitung`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Das Attribut `letztesBearbeitungsDatum` ist dagegen nicht im Formular änderbar, sondern wird einmalig in Zeile 53 auf den aktuellen Zeitstempel gesetzt. Zugehörig zu diesem Attribut gibt es noch eine abgeleitete Eigenschaft namens `formattedDate` (Z. 45-48). Es ist eine Hilfsmethode, die das letzte Bearbeitungsdatum in ein für Menschen lesbares Datumsformat umwandelt. In dem Übersichts-Bildschirm Abbildung 4.1 ist das Datumsformat sichtbar.

Da diese Getter-Methode eine Implementierung besitzt, wird für sie von „*built_value*“ kein Quellcode für die Serialisierung generiert.

Bevor die Werte-Typen serialisiert werden können, muss `built_value` jedoch noch mitgeteilt werden, für welche Werte-Typen Serialisierungs-Funktionen generiert werden sollen. Dazu werden über die Annotation `@SerializersFor` die gewünschten Klassen aufgelistet (Listing 4.9, Z. 10). Die Zeilen 11 und 12 sind dabei immer gleich, es sei denn, es ist ein anderer Serialisierung Algorithmus gewünscht. In diesem Fall wird das `StandardJsonPlugin` verwendet.

```
10 @SerializersFor([Massnahme, Storage])
11 final Serializers serializers =
12     (_$serializers.toBuilder()..addPlugin(StandardJsonPlugin())).build();
```

Listing 4.9.: Der Serialisierer für `Massnahme` und `Storage`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/serializers.dart](#)

Wird nun der Befehl `flutter pub run build_runner build` ausgeführt, so wird der Quellcode generiert und die Werte-Typen können für die Serialisierung genutzt werden.

4.3. Test der Serialisierung einer Maßnahme

Das Ergebnis der Serialisierung wird im dazugehörigen Unit-Test ersichtlich (Listing 4.10). In Zeile 8 wird ein Objekt der Klasse `Massnahme` instanziiert. Anders als bei gewöhnli-

chen Datentypen lassen sich bei diesem unveränderlichen Datentyp keine Attribute nach der Erstellung anpassen. Die einzige Möglichkeit besteht darin, ein neues Objekt mit dem gewünschten Attributwert zu erstellen und die restlichen Werte des alten Objekts zu übernehmen. Dies ist mit Hilfe des sogenannten „Builder“-Entwurfsmuster möglich, welches in „*built_value*“ Anwendung findet.

```

6 test('Massnahme serialises without error', () {
7   var massnahme = Massnahme();
8   massnahme = massnahme
9     .rebuild((b) => b.identifikatoren.massnahmenTitel = "Massnahme 1");
10
11   var actualJson = serializers.serializeWith(Massnahme.serializer, massnahme);
12
13   var expectedJson = {
14     'guid': massnahme.guid,
15     'letzteBearbeitung': {
16       'letztesBearbeitungsDatum': massnahme
17         .letzteBearbeitung.letztesBearbeitungsDatum.microsecondsSinceEpoch,
18       'letzterStatus': 'bearb'
19     },
20     'identifikatoren': {'massnahmenTitel': 'Massnahme 1'}
21   };
22
23   expect(actualJson, equals(expectedJson));

```

Listing 4.10.: Serialisierung einer Maßnahme Unittest, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/test/data_model/massnahme_test.dart](#)

Erbauer-Entwurfsmuster Das Erbauer-Entwurfsmuster - englisch builder pattern - ist ein Erzeugungsmuster, welches die Konstruktion komplexer Objekte von ihrer Repräsentation trennt. Es gehört zu der Serie von Entwurfsmustern der Gang of Four.¹⁰ Im Fall von *built_value* trennt es die unveränderlichen Objekte von ihrer Konstruktion. Über den „Builder“ lassen sich Änderungen an diesen unveränderlichen Objekten vornehmen, wodurch eine Kopie dieses unveränderlichen Objekts mit der gewünschten Änderung zurückgegeben wird.

In den Zeilen 9 bis 10 wird so ein neues Objekt von der Klasse *Maßnahme* mit Hilfe der Methode `rebuild` erzeugt und anschließend der Referenz `massnahme` zugewiesen, wodurch sie ihren alten Wert verliert. Über die generierte Methode `serializers.serializeWith` kann das Objekt in JSON übersetzt werden. Der erste Parameter `Massnahme.serializer` gibt dabei an, wie diese Serialisierung erfolgen soll. Auch das `serializer`-Objekt wurde von „*built_value*“ generiert. Der zweite Parameter ist die tatsächliche `massnahme`, die in JSON umgewandelt werden soll. Die Zeilen 13 bis 21 erstellen das JSON-Dokument, mit dem das serialisierte Ergebnis am Ende verglichen werden soll. Dabei werden die gleichen Eigenschaften eingetragen. So etwa die `guid` (Z. 14), welche bei der Initialisierung der *Maßnahme* automatisch und zufällig erstellt wurde. Außerdem das letzte Bearbeitungsdatum, welches den Zeitstempel erhält, zu dem die *Maßnahme* generiert wurde. Da *built_value* bei der Serialisierung die Datumswerte in Mikrosekunden umwandelt, muss für das erwar-

¹⁰Vgl. 11, S. 119.

tete JSON-Dokument das Gleiche passieren (Z. 16-17). Der `'letzterStatus'` (Z. 18) wird hierbei auf den Standardwert `'bearb'` gesetzt und der `'massnahmenTitel'` (Z. 20) auf den gleichen Wert, der in Zeile 9 übergeben wurde. Schließlich vergleicht die Methode `expect` das tatsächlich serialisierte JSON-Dokument mit dem, welches zuvor zum Vergleich aufgebaut wurde. Der zweite Parameter ist ein sogenannter `Matcher` und die Variante mit dem Namen `equals` überprüft auf absolute Gleichheit.

Analog zur Serialisierung testet der Unit-Test in Listing 4.11 auch die Deserialisierung. Das JSON-Dokument ist dabei sehr ähnlich und unterscheidet sich lediglich in zwei Details. Der `'guid'` wird auf einen festen Wert festgelegt (Z. 38), statt - wie zuvor - durch den in dem Initialisierungsprozess der Maßnahme zufällig generiert zu werden. Außerdem wird auch das `letztesBearbeitungsDatum` festgesetzt, nämlich auf die Microsekunde 0 (Z. 40).

```

36 test('Massnahme deserialises without error', () {
37   var json = {
38     'guid': "test massnahme id",
39     'letzteBearbeitung': {
40       'letztesBearbeitungsDatum': 0,
41       'letzterStatus': 'bearb'
42     },
43     'identifikatoren': {'massnahmenTitel': 'Massnahme 1'}
44   };
45
46   var expectedMassnahme = Massnahme((b) => b
47     ..guid = "test massnahme id"
48     ..identifikatoren.massnahmenTitel = "Massnahme 1"
49     ..letzteBearbeitung.update((b) {
50       b.letztesBearbeitungsDatum =
51         DateTime.fromMicrosecondsSinceEpoch(0, isUtc: true);
52     }));
53   var actualMassnahme =
54     serializers.deserializeWith(Massnahme.serializer, json);
55
56   expect(actualMassnahme, equals(expectedMassnahme));

```

Listing 4.11.: Deserialisierung einer Maßnahme Unittest, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/test/data_model/massnahme_test.dart](#)

Zum Vergleich wird in den Zeilen 46 bis 52 eine Maßnahme über das „*Builder*“-Entwurfsmuster generiert und die gleichen festen Werte werden für die Eigenschaften übergeben. Dabei ist darauf zu achten, dass die Instanzvariable `letzteBearbeitung` keinen Wert über den Zuweisungs-Operator `=` erhält, sondern stattdessen die Methode `update` darauf aufgerufen wird.

Da es sich bei der Instanzvariablen `letzteBearbeitung` genauso um ein Objekt eines Wertetypen - handelt, ist sie ebenso unveränderlich. Deshalb kann sie nur über einen „*Builder*“ manipuliert werden. Ein Blick in den generierten Quellcode offenbart, dass es sich in Wahrheit um einen „*Builder*“ handelt (Listing 4.12, Z. 224-225).

Außerdem müssen die Microsekunden für das Datum zunächst in ein Objekt von `DateTime` umgewandelt werden. Dafür wird der benannte Konstruktor `fromMillisecondsSinceEpoch`

```

216 class MassnahmeBuilder implements Builder<Massnahme, MassnahmeBuilder> {
217   _$Massnahme? _$v;
218
219   String? _guid;
220   String? get guid => _$this._guid;
221   set guid(String? guid) => _$this._guid = guid;
222
223   LetzteBearbeitungBuilder? _letzteBearbeitung;
224   LetzteBearbeitungBuilder get letzteBearbeitung =>
225     _$this._letzteBearbeitung ??= new LetzteBearbeitungBuilder();
226   set letzteBearbeitung(LetzteBearbeitungBuilder? letzteBearbeitung) =>
227     _$this._letzteBearbeitung = letzteBearbeitung;

```

Listing 4.12.: Instanzvariable `letzteBearbeitung` gibt einen `LetzteBearbeitungBuilder` zurück, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.g.dart](#)

von `DateTime` (Z. 51) aufgerufen.

Benannte Konstruktoren In Programmiersprachen wie beispielsweise Java können Methoden überladen werden, indem ihre Methodensignatur geändert wird. Beim Aufruf der Methode kann über die Anzahl und die Typen der übergebenen Argumente die gewünschte Methode gewählt werden. Das Gleiche gilt für Konstruktoren. Wird ein weiterer Konstruktor für eine Klasse in Java benötigt, so besteht einzig und allein die Möglichkeit darin, den Konstruktor zu überladen. Sowohl überladene Methoden als auch überladene Konstruktoren existieren in Dart nicht. Wird also in Dart ein alternativer Konstruktor gewünscht, so muss er einen Namen bekommen. Beim Aufruf des Konstruktors wird dieser Name dann mit einem `.` nach dem Klassennamen angegeben, um den gewünschten Konstruktor zu benennen.

Ganz ähnlich wie bei der Serialisierung wird nun mit dem Befehl `serializers . deserializeWith` unter Angabe des Objekts, welches die Deserialisierung übernehmen soll – nämlich wiederum `Massnahme.serializer` – das JSON-Dokument in ein Objekt des Werte-Typs `Massnahme` deserialisiert (Z. 53-54). Schließlich wird in Zeile 56 das Ergebnis der Deserialisierung mit dem gewünschten Ergebnis verglichen.

Gibt man in der Kommandozeile den Befehl `flutter test test /data_model /massnahme_test.dart` ein, so werden die Tests in der Testdatei ausgeführt. Die Ausgabe `00:01 +2: All tests passed!` teilt mit, dass beide Tests erfolgreich ausgeführt wurden und beide Ergebnisse mit den verglichenen Werten übereinstimmen.

4.4. Serialisierung der Maßnahmenliste

Damit alle Maßnahmen - statt nur einer einzigen - in einer Datei zusammengefasst werden können, müssen die Maßnahmen zunächst zu einer Menge zusammengefasst werden, die

ebenfalls serialisierbar ist. Der Werte-Typ `Storage` ist dafür vorgesehen (Listing 4.13). Er deklariert allein das `BuiltSet massnahmen` (Z. 10). Ein `BuiltSet` ist die Abwandlung eines gewöhnlichen Sets, jedoch unter anderem mit der Möglichkeit, es mit einem „*Builder*“ zu erstellen und das Set zu serialisieren. Die Übergabe des Typarguments `<Massnahme>` gewährleistet, dass keine anderen Objekte eingefügt werden können, die weder eine Instanz der Klasse `Massnahme` sind, oder einer Klasse, die von `Massnahmen` erbt.

```
9 abstract class Storage implements Built<Storage, StorageBuilder> {  
10   BuiltSet<Massnahme> get massnahmen;
```

Listing 4.13.: Der Werte-Typ `Storage`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/storage.dart](#)

Der Befehl `flutter pub run build_runner build` stößt erneut die Quellcodegenerierung für den Werte-Typen `Storage` an.

4.5. Test der Serialisierung der Maßnahmenliste

Nun soll noch überprüft werden, ob die Menge von Maßnahmen mit genau einer eingetragenen Maßnahme korrekt serialisiert. Auch das wird von einem Unit Test überprüft (Listing 4.14). In Zeile 8 wird das leere Objekt `storage` erstellt. In Zeile 9 wird es dann wiederverwendet, um aufbauend auf der Kopie Änderungen mithilfe der `rebuild`-Methode durchzuführen.

```
7 test('Storage with one Massnahme serialises without error', () {  
8   var storage = Storage();  
9   storage = storage.rebuild((b) => b.massnahmen.add(  
10     Massnahme((b) => b.identifikatoren.massnahmenTitel = "Massnahme 1")));  
11  
12   var actualJson = serializers.serializeWith(Storage.serializer, storage);  
13  
14   var expectedJson = {  
15     "massnahmen": [  
16       {  
17         "guid": storage.massnahmen.first.guid,  
18         "letzteBearbeitung": {  
19           "letztesBearbeitungsDatum": storage  
20             .massnahmen  
21             .first  
22             .letzteBearbeitung  
23             .letztesBearbeitungsDatum  
24             .microsecondsSinceEpoch,  
25           "letzterStatus": "bearb"  
26         },  
27         "identifikatoren": {"massnahmenTitel": "Massnahme 1"}  
28       }  
29     ]  
30   };  
31   expect(actualJson, equals(expectedJson));
```

Listing 4.14.: Ein automatisierter Testfall überprüft, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/test/data_model/storage_test.dart](#)

Bei der Instanzvariable `massnahmen` der Klasse `Storage` handelt es sich um ein `BuiltSet`. Der Aufruf von `b.massnahmen` gibt allerdings nicht dieses `BuiltSet` zurück. Wäre es so, so könnte die Operation `add` nicht darauf angewendet werden. Ein `BuiltSet` stellt keine Methoden zur Manipulation des Sets zur Verfügung. In Wahrheit gibt der Ausdruck `b.massnahmen` einen `SetBuilder` zurück. Das kann im generierten Quellcode nachgesehen werden (Listing 4.15, Z. .)

```

91 class StorageBuilder implements Builder<Storage, StorageBuilder> {
92   _$Storage? _$v;
93
94   SetBuilder<Massnahme>? _massnahmen;
95   SetBuilder<Massnahme> get massnahmen =>
96     _$this._massnahmen ??= new SetBuilder<Massnahme>();

```

Listing 4.15.: Instanzvariable `massnahmen` gibt einen `SetBuilder` zurück, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/storage.g.dart](#)

Der `SetBuilder` wiederum erlaubt es, Änderungen am Set vorzunehmen und stellt dafür die - für ein Set übliche - Methode `add` bereit. Im Aufruf von `add` wird dann ein Objekt des Werte-Typs `Maßnahme` konstruiert (Z. 10). Dazu wird dieses Mal die anonyme Funktion zum Konstruieren der `Maßnahme` gleich im Konstruktor übergeben.

Diesmal konstruiert die Methode `serializers.serializeWith` mit dem Serialisierer `Storage.serializer` ein weiteres JSON-Objekt (Z. 12). Genau wie zuvor wird ein JSON-Dokument vorbereitet (Z. 14-30), welches der `Matcher equals` mit dem serialisierten Dokument des soeben konstruierten Objekts `storage` vergleicht (Z. 31). Das JSON-Dokument unterscheidet sich nur darin, dass es einen Knoten namens `'massnahmen'` enthält, der als Wert eine Liste hat. Die Liste hat nur ein Element. Weil dieses Mal das Objekt des Typs `Massnahme` nicht direkt zugreifbar ist, muss es zunächst über die Liste der `Maßnahmen` aus dem `storage`-Objekt abgerufen werden. Das ist mit dem Befehl `first` möglich, der das erste Objekt - und in diesem Fall einzige Objekt - der Kollektion zurückgibt (Z. 17, 21). Darüber kann erneut die `guid` und das `letztesBearbeitungsDatum` abgerufen werden.

Ein weiterer Unit-Test überprüft, ob auch die Deserialisierung eines `storage`-Objekts erfolgreich ist. Er ist in Listing 2.1 im Anhang ?? zu finden. Auch dieser Test ist der Deserialisierung des Objekts des Typs `Massnahme` sehr ähnlich. **Weg?** Er unterscheidet sich nur darin, dass das `Massnahme`-Objekt in der Liste `massnahmen` des `storage`-Objekts enthalten ist.

4.6. Der Haupteinstiegspunkt

Das Listing 4.16 zeigt den Haupteinstiegspunkt des Programms. Darin ist erkennbar, dass sich die Applikation in drei Rubriken einteilen lässt:

- das Model (Z. 27-30)
- der View (Z. 41-44)
- das ViewModel. (Z. 25-26)

```

18 class MassnahmenFormApp extends StatelessWidget {
19   const MassnahmenFormApp({Key? key}) : super(key: key);
20
21   @override
22   Widget build(BuildContext context) {
23     return AppState(
24       model: MassnahmenModel(MassnahmenJsonFile()),
25       viewModel: MassnahmenFormViewModel(),
26       child: MaterialApp(
27         title: 'Maßnahmen',
28         theme: ThemeData(
29           primarySwatch: Colors.lightGreen,
30           accentColor: Colors.green,
31           primaryIconTheme: const IconThemeData(color: Colors.white),
32         ),
33         initialRoute: MassnahmenMasterScreen.routeName,
34         routes: {
35           MassnahmenMasterScreen.routeName: (context) =>
36             const MassnahmenMasterScreen(),
37           MassnahmenDetailScreen.routeName: (context) =>
38             const MassnahmenDetailScreen()
39         },
40       ));
41   }
42 }

```

Listing 4.16.: Der Haupteinstiegspunkt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/main.dart](#)

Model View ViewModel Das ModelViewViewModel Entwurfsmuster wurde zunächst von John Gossman für die Windows Presentation Foundation beschrieben. Das Model beschreibt die Datenzugriffs-Komponente welche die Daten in relationalen Datenbanken oder hierarchischen Datenstrukturen wie XML oder JSON ablegt. Der View beschreibt die Oberflächenelemente wie Texteingabefelder und Buttons. Diese beiden Komponenten sind auch aus dem ModelViewController Entwurfsmuster bekannt. Das ModelViewViewModel Entwurfsmuster ist eine Weiterentwicklung davon und integriert das sogenannte ViewModel. Es ist dafür zuständig als Schnittstelle zwischen View und Model zu fungieren. Die Daten des Models lassen sich in der Regel nicht direkt mit Oberflächen Elementen verknüpfen. Denn es kann es notwendig sein, dass die Oberfläche weitere temporäre Daten benötigt, die aber nicht mit den Daten des Models gespeichert werden sollen. Das ViewMo-

del übernimmt diese Arbeit, indem es die Daten des Models abrufen und sie in veränderter Form den Oberflächen-Elementen zur Verfügung steht. Andersherum formt es die Eingaben in der Nutzeroberfläche so um, dass sie im strikten Datenmodell des Models Platz finden.¹¹

`MassnahmenModel` (Z. 29) verwaltet die eingegebenen Daten der Maßnahmen und nutzt die Abhängigkeit `MassnahmenJsonFile` (Z. 27) um die Daten auf einem Datenträger als eine JSON-Datei zu speichern. Somit gehören diese beiden Klassen dem Model an.

`MassnahmenFormViewModel` (Z. 25) greift die Daten des Models ab und formt diese um, sodass sie von dem View `MassnahmenDetailScreen` (Z. 43) verändert werden können. Sollen die Daten gespeichert werden, so stellt `MassnahmenFormViewModel` ebenfalls Methoden zur Verfügung um die Daten wieder in das Format des Models einpflegen zu können.

`MassnahmenMasterScreen` (Z. 41) stellt eine Ausnahme dar, denn dieser View präsentiert die Daten aus dem Model ohne eine Schnittstelle über ein ViewModel. Das ist möglich, weil die Daten nicht manipuliert, sondern nur angezeigt werden müssen.

Damit sowohl ViewModel als auch Model von jedem View heraus abrufbar sind, werden sie in eine Art Service eingefügt (Z. 23). Das Widget `AppState` ist dieser Service. Er erhält das Model (Z. 24) und das ViewModel (Z. 25) im Konstruktor. Die Abhängigkeit zum Schreiben des Models in eine JSON Datei `MassnahmenJsonFile` bekommt das Model ebenfalls im Konstruktor übergeben (Z. 24). `AppState` ist das erste Element, welches im Widget-Baum auftaucht. Die gesamte restliche Applikation ist als Kind-Element hinterlegt (Z. 26). Damit können alle Widgets auf den Service zugreifen.

Service Locator und Dependency Injection Das Service Locator Entwurfsmuster folgt dem Umsetzungsparadigma Inversion of Control – deutsch Umkehrung der Steuerung. Frameworks folgen diesem Muster, indem sie als erweiterbare Skelett-Applikationen fungieren. Anstatt, dass die Applikation den Programmfluss steuert und dabei selbst Funktionen aufruft, wird die Programmflusssteuerung an das Framework abgegeben und mit Hilfe von Ereignissen ermöglicht, dass das Framework Funktionen des Nutzers aufruft.¹² Im Service Locator Entwurfsmuster werden Komponenten darüber hinaus zentral registriert und über dieses Register anderen Komponenten zur Interaktion zur Verfügung gestellt.¹³ Anstatt die Komponenten direkt miteinander zu verknüpfen, werden Sie für den Zugriff von praktisch überall vorbereitet. Vor allem für automatisierte Tests ist dies von Vorteil, da solche Abhängigkeiten ausgetauscht werden können, um ganz spezielle Teil-Funktionalitäten eines Programms zu testen. Mehr dazu im Kapitel `Kapitel einfügen`.

¹¹Vgl. 27.

¹²Vgl. 29.

¹³Vgl. 9.

Anders als der Name vermuten lässt, steuert `MaterialApp` nicht nur das Aussehen der Applikation im Material Design Look. Darüber hinaus stellt das Widget auch Grundfunktionalitäten einer App, wie etwa den Navigator bereit. Damit hat die Applikation die Möglichkeit – ähnlich wie bei einer Webside – auf Unterseiten zu navigieren. Hat der Benutzer die Arbeit in der Unterseite vollendet, so kann der Navigator gebeten werden, zur vorherigen Ansicht zurückzukehren. Mit dem Parameter `routes` (Z. 34-39) erfolgt die Angabe der Unterseiten, die besucht werden können. Über `initialRoute` (Z. 39) kann die Startseite angegeben werden.

4.7. Der Service für den Applikations übergreifenden Zustand

Um Daten an alle Kind Elementen im Widgets mitzugeben, finden die sogenannten „*InheritedWidgets*“ Anwendung. Der Service `AppState` (Listing 4.17) ist genauso ein solches.

Im Konstruktor erhält er zunächst bei den Parameter des Typs `key` (Z. 7). Es ist gängige Praxis in Flutter, jedem Widget im Konstruktor zu ermöglichen, einen solchen Schlüssel zu übergeben. Es ist jedoch optional. Ein solcher Schlüssel kann genutzt werden, um das Widget eindeutig zu identifizieren und es unter anderem über den Schlüssel wiederzufinden. In den Zeilen 8 und 9 werden das Model und das ViewModel dem Objekt im Konstruktor übergeben. In den Zeilen 14 und 15 sind sie deklariert. Das letzte Element im Konstruktor ist das `child`. Ihm muss der Widget-Baum übergeben werden, dem der Zustand verfügbar gemacht werden soll.

Der Aufruf des Basis-Konstruktors mit den Argumenten `key` und `child` ist in Zeile 11 zu sehen. Die Basisklasse von `InheritedWidget` ist `ProxyWidget` und erhält exakt dieselben Argumente. Das `ProxyWidget` verwendet das Kindelement, um es im WidgetBaum unterhalb von sich selbst zu zeichnen. Eine eigene Methode zum Zeichnen muss also nicht für das `InheritedWidget` implementiert werden. Die einzige Methode, welche implementiert werden muss, ist `updateShouldNotify` (Z. 24). Immer dann, wenn das `InheritedWidget` selbst aktualisiert wird, kann es alle Widgets, die davon abhängig sind, benachrichtigen. In dem Fall werden diese Widgets ebenfalls neu gezeichnet. Für die Formular-Applikation ist das allerdings nicht gewünscht. Die Aktualisierung der Oberfläche soll in den nachfolgenden Schritten selbst kontrolliert werden. Deshalb erfolgt die Rückgabe `false`, da in Zukunft nicht gewünscht ist, den Applikations-Zustand komplett auszutauschen. Um die Aktualisierung der Oberfläche kümmern sich sowohl Model als auch ViewModel.

Damit ein Widget Abhängigkeit von dem `AppState` anmelden kann, verwendet es in seiner eigenen `build`-Methode die Methode `dependOnInheritedWidgetOfExactType<AppState>()`. Der Aufruf der Methode erfolgt auf dem Objekt vom Typ `BuildContext`. Weil dieser Kontext bei jedem Zeichnen allen Kindern übergeben wird, kann jedes Kind darüber die Vater-Elemente wiederfinden.

Damit der Aufruf leichter lesbar und kürzer ist, empfiehlt das Flutter-Team eine eigene Klassenmethode zu erstellen, welche die Methode für den Benutzer aufruft (Z. 16-17). Auch eine Fehlermeldung kann bei dieser Auslagerung geworfen werden, sollte im Kontext kein Objekt des gewünschten Typs vorhanden sein (Z. 18). Das Widget, welches auf den `AppState` zugreifen möchte, kann es mit der einfachen Schreibweise `AppState.of(context)` abrufen.

Abbildung 4.4 zeigt die Beziehung zwischen den Bildschirmen und dem `AppState` auf. Sowohl `MassnahmenMasterScreen` und `MassnahmenDetailScreen` müssen auf `MassnahmenModel` und `MassnahmenFormViewModel` zugreifen können. Zu diesem Zweck erstellt `MassnahmenFormApp` den `AppState`. Er enthält sowohl `ViewModel` als auch `Model`. Über ihn können beide Bildschirme auf `Model` und `ViewModel` zugreifen.

```

5 class AppState extends InheritedWidget {
6   const AppState({
7     Key? key,
8     required this.model,
9     required this.viewModel,
10    required Widget child
11  }) : super(key: key, child: child);
12
13   final MassnahmenFormViewModel viewModel;
14   final MassnahmenModel model;
15
16   static AppState of(BuildContext context) {
17     final AppState? result = context.dependOnInheritedWidgetOfExactType<AppState>();
18     assert(result != null, "Kein AppState im 'context' gefunden");
19     return result!;
20   }
21
22   @override
23   bool updateShouldNotify(covariant AppState oldWidget) => false;
24 }

```

Listing 4.17.: Der Service `AppState`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/app_state.dart](#)

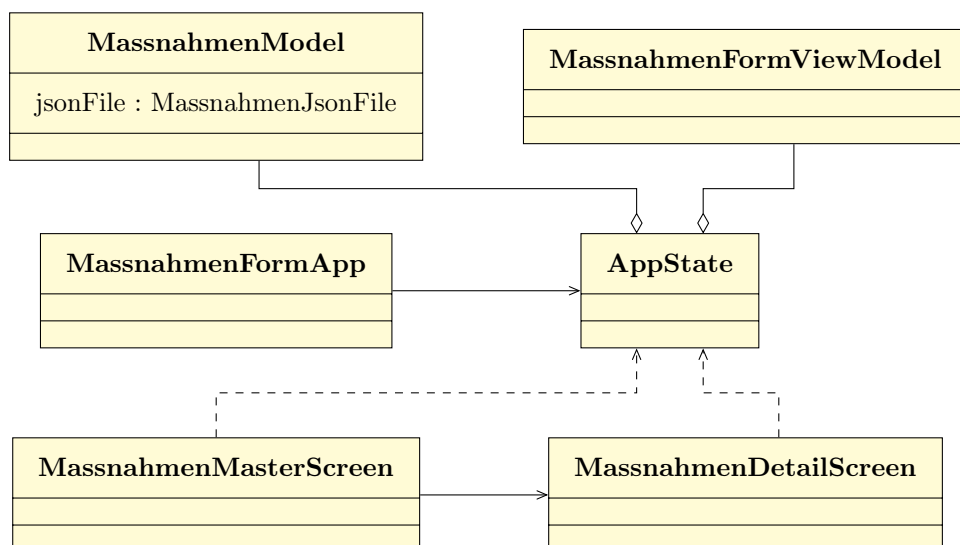


Abbildung 4.4.: UML Diagramme, Quelle: Eigene Abbildung

4.8. Speichern der Maßnahmen in eine JSON-Datei

Das Model wird durch die Klasse `MassnahmenJsonFile` in eine JSON-Datei gespeichert (Listing 4.18). Der Dateipfad wird dabei durch die Methode `_localMassnahmenJsonFile` (Z. 8-11) abgerufen. Die Hilfsmethode `getApplicationSupportDirectory` (Z. 9) gibt aus dem Nutzerverzeichnis des aktuellen Nutzers den zur Applikation zugeordneten Datei-Ordner zurück. Auf Windows-Betriebssystemen wäre das beispielsweise `C:\Users\AktuellerNutzer\AppData\Roaming\com.example\conditional_form`.

Dadurch, dass dem Methoden-Bezeichner `_localMassnahmenJsonFile` ein Unterstrich vorangestellt ist, ist die Methode privat und kann nur innerhalb der Klasse aufgerufen werden. Dart hat damit eine Konvention zum Standard werden lassen. In Programmiersprachen wie beispielsweise C++ wurde der Unterstrich zusätzlich den Bezeichnern von Instanz Attributen vorangestellt, die mit dem `private` Schlüsselwort gekennzeichnet sind, damit sie überall im Quellcode als private Attribute identifizierbar sind, ohne dazu die Klassendefinition ansehen zu müssen. In Dart gibt es dagegen das `private` Schlüsselwort nicht. Stattdessen wird der Unterstrich vor dem Bezeichner verwendet, um ein Instanzattribut privat zu deklarieren.

Die Getter-Methode `_localMassnahmenJsonFile` hat den Rückgabetypp `Future<File>` und ist zudem mit dem Schlüsselwort `async` gekennzeichnet. Asynchron muss die Methode deshalb sein, weil sie auf den Aufruf `getApplicationSupportDirectory` warten muss, der ebenfalls asynchron abläuft.

Der Funktion `saveMassnahmen` (Z. 13-16) wird ein JSON Objekt in Form einer Hashtabelle übergeben. Sie ruft die Hilfs-Getter-Methode `_localMassnahmenJsonFile` (Z. 14) auf und schreibt den Dateinhalt in die Datei des abgefragten Pfades (Z. 15). Zuvor wird dazu das JSON-Objekt in eine textuelle Repräsentation überführt. Dazu dient die Funktion `jsonEncode`.

Das Äquivalent dazu stellt die Methode `readMassnahmen` (Z. 18-30) dar. Auch sie ruft den Dateipfad ab (Z. 19), überprüft allerdings im nächsten Schritt, ob die Datei bereits existiert (Z. 21). Sollte das der Fall sein, so wird die Datei eingelesen (Z. 23). Die textuelle Repräsentation aus der Datei wird mittels Methode `jsonDecode` in ein JSON-Objekt in der Form einer Hashtabelle gespeichert (Z. 24) und schließlich zurückgegeben (Z. 26). Sollte die Dateien nicht existieren, führt das zu einer Ausnahme (Z. 28), welche von der aufrufenden Funktion behandelt werden kann.

```

7 class MassnahmenJsonFile {
8   Future<File> get _localMassnahmenJsonFile async {
9     var directory = await getApplicationSupportDirectory();
10    return File("${directory.path}/Maßnahmen.json");
11  }
12
13  Future<void> saveMassnahmen(Map<String, dynamic> massnahmenAsJson) async {
14    var file = await _localMassnahmenJsonFile;
15    await file.writeAsString(jsonEncode(massnahmenAsJson));
16  }
17
18  Future<Map<String, dynamic>> readMassnahmen() async {
19    var file = await _localMassnahmenJsonFile;
20
21    var fileExists = await file.exists();
22    if (fileExists) {
23      final fileContent = await file.readAsString();
24      final jsonObject = jsonDecode(fileContent) as Map<String, dynamic>;
25
26      return jsonObject;
27    } else {
28      throw MassnahmenFileDoesNotExistException("$file was not found");
29    }
30  }
31 }

```

Listing 4.18.: Die Klasse MassnahmenJsonFile, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/persistence/massnahmen_json_file.dart](#)

4.9. Abhängigkeit zum Verwalten der Maßnahmen

Die Art und Weise, wie die Maßnahmen abgerufen werden, sollte nach Möglichkeit abstrahiert werden. Das erlaubt, den Mechanismus in Zukunft auszutauschen, ohne dabei den Rest der Applikation verändern zu müssen. So wäre es beispielsweise denkbar, statt einer JSON-Datei eine direkte Verbindung zu einer relationalen Datenbank herzustellen. Auch das Austauschen der Abhängigkeit mit einem Platzhalter, der lediglich die Aufrufe der Methoden zählt, ist damit möglich. Ein solches Platzhalterobjekt wird „Mock“ genannt und für automatisiertes Testen eingesetzt (siehe Kapitel **Kapitel einfügen**). Ebenso abstrahiert werden soll der Umgang mit Ausnahmen. Sollte die Datei nicht verfügbar sein, so muss die Oberfläche davon nicht zwingend betroffen sein. Stattdessen kann der Service sich entscheiden, eine leere Liste von Maßnahmen zurückzugeben. Sobald die Liste manipuliert wird, kann eine neue Datei angelegt werden und sie mit den eingegebenen Daten beschreiben. Die Klasse `MassnahmenModel` (Listing 4.19) tut genau das.

Sie bekommt `MassnahmenJsonFile` im Konstruktor übergeben (Z.11). Daraufhin ruft der Konstrukteur gleich die `init` auf (Z. 12), welche in den Zeilen 15-22 deklariert ist. Darin wird der Stream `storage` (Z. 19) initialisiert. Es handelt sich um eine Erweiterung eines „broadcast streams“ mit dem Namen `BehaviorSubject` (Z. 9). Es entstammt dem Paket `rx.dart`, welches die Streams in Dart um eine Reihe von weiteren Funktionalitäten erweitert. Ein `BehaviorSubject` hat die Besonderheit, dass es den Wert des letzten Ereignisses zwischenspeichern. Die „broadcast streams“ haben für gewöhnlich den Nachteil, dass neue Zuhörer des Streams nur die neuen Ereignisse erhalten. Alle in der Vergangenheit erfolgten Ereignisse sind nicht mehr verfügbar. Vor allem dann, wenn in der Oberfläche der letzte Wert eines Streams verwendet werden soll, um Elemente zu zeichnen, ist das von einem besonderen Nachteil. Denn wenn der Stream zuvor initialisiert wurde, so gibt es keine Daten zu dem Zeitpunkt, wenn die Oberfläche gezeichnet wird. Sollte die Oberfläche jedoch gezeichnet werden, bevor der Stream initialisiert wurde, so existieren ebenfalls keine Daten. Hier kommt das `BehaviorSubject` ins Spiel. Sobald die Oberfläche gezeichnet wird und der Stream bereits initialisiert ist, kann dennoch auf den zuletzt übertragenen Wert zurückgegriffen werden. Anschließend überträgt der Stream die folgenden Aktualisierungen für die Oberfläche mit jedem neuen Ereignis, so wie es für Streams üblich ist.

Der Stream kann nicht bereits in der Initialisierungsliste des Konstruktors mit den Daten aus der JSON-Datei gefüllt werden. Das liegt daran, dass die JSON-Daten dazu zunächst gelesen werden müssen, was nur durch eine Reihe von asynchronen Operation möglich ist. In einer Initialisierungsliste können allerdings keine asynchronen Operationen ausgeführt werden. Deshalb wird `init` erst im Konstruktor-Körper aufgerufen (Z. 7).

Damit der Stream anfangs nicht leer ist, füllt ihn der benannte Konstruktor `seeded` mit einem leeren Objekt des Typs `Storage` (Z. 9). Sobald die Datei gelesen (Z. 17) und anschließend deserialisiert wurde (Z. 20), erhält der Stream über die Setter-Methode `value`

ein neues Ereignis mit dem gelesenen Wert (Z. 19).

Die Initialisierung ist von einem `try`-Block umgeben. Sollte die Initialisierung verschlagen, weil die JSON-Datei nicht existiert, wird die entsprechende Fehlerbehandlung ausgeführt (Z. 21). Diese ist leer, da sich im Stream bereits ein leeres `Storage`-Objekt befindet. Mit diesem leeren Objekt kann die Oberfläche weiterarbeiten. In Zukunft könnte es sinnvoll sein, innerhalb der Fehlerbehandlung eine Meldung an den Benutzer zu geben, um darüber zu informieren, dass eine neue Datei angelegt wurde.

Mit `putMassnahmeIfAbsent` (Z. 24-33) steht eine Methode bereit, um gleichzeitig sowohl die Oberfläche, als auch die JSON-Datei zu aktualisieren. Sollte die eigetragene Maßnahme schon existieren, wird sie zunächst gelöscht (Z. 26). In jedem Fall wird die neue Maßnahme dem Stream hinzugefügt (Z. 27). Durch Austauschen des gesamten Objekts mit der Zuweisung von `storage.value` (Z. 25) erhält der Stream erneut ein neues Ereignis, womit er die Oberfläche benachrichtigen kann, sich neu zu zeichnen. Außerdem wird die Serialisierung des `Storage`-Objekts und angestoßen (Z. 29-30) und die neue Liste von Maßnahmen im darauffolgenden Schritt zurück in die JSON-Datei gespeichert (Z. 32).

```

7 class MassnahmenModel {
8   final MassnahmenJsonFile jsonFile;
9   final storage = BehaviorSubject<Storage>.seeded(Storage());
10
11   MassnahmenModel(this.jsonFile) {
12     init();
13   }
14
15   init() async {
16     try {
17       final massnahmenAsJson = await jsonFile.readMassnahmen();
18
19       storage.value =
20         serializers.deserializeWith(Storage.serializer, massnahmenAsJson)!;
21     } on MassnahmenFileDoesNotExistException {}
22   }
23
24   putMassnahmeIfAbsent(Massnahme massnahme) async {
25     storage.value = storage.value.rebuild((b) => b.massnahmen
26       ..removeWhere((m) => m.guid == massnahme.guid)
27       ..add(massnahme));
28
29     var serializedMassnahmen =
30       serializers.serializeWith(Storage.serializer, storage.value);
31
32     await jsonFile.saveMassnahmen(serializedMassnahmen as Map<String, dynamic>);
33   }
34 }

```

Listing 4.19.: Die Klasse `MassnahmenModel`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_access/massnahmen_model.dart](#)

4.10. Übersichtsbildschirm der Maßnahmen

Der erste Bildschirm - die Übersicht der Maßnahmen - kann auf das im letzten Schritt erstellten Model zugreifen. In Listing 4.20 ist die Struktur des Übersicht-Bildschirms zu sehen. Über die Route `/massnahmen_master` ist der Bildschirm erreichbar (Z. 16). Die `build`-Methode zeichnet die Oberfläche 21-111. Da ein Objekt des Typs `MassnahmenPool` im zentralen Register der Provider hinterlegt wurde, kann mit der Methode `Provider.of` darauf zugegriffen werden.

Mittels `AppState.of(context)` ist nun der Zugriff auf sowohl Model als auch ViewModel möglich. Zur einfacheren Verwendung sind sie als lokale Variablen zwischengespeichert (Z. 20-21).

Das Widget `Scaffold` - deutsch Gerüst - stellt ein grundlegendes Layout mit einer Überschrift und einem Bereich für den Inhalt bereit (Z. 23). Das `Scaffold` kann auch Mitteilungen an den Benutzer am unteren Bildschirmrand einblenden.

Die Überschrift wird in der sogenannten `AppBar` hinterlegt (Z. 24). Sie unterstützt weitere Funktionalitäten. Sollte es sich bei der aktuell besuchten Route um eine Unterseite handeln, taucht links von der Titel-Überschrift einen Button zum Zurücknavigieren auf. Weiterhin können rechts von der Titelleiste Aktionsbuttons hinzugefügt werden, welches für die Formular Anwendung allerdings nicht nötig ist.

Zusätzlich kann dem `Scaffold` ein Button für die primäre Aktion auf diesem Bildschirm hinzugefügt werden: der sogenannte `FloatingActionButton` (Z. 88-97). Bei Aktivierung des Buttons navigiert die Applikation zur Eingabemaske, um eine neue Maßnahme anzulegen (Z. 98).

Das Eingabeformular sollte den Benutzer auffordern, tatsächlich leere Eingabefelder zu füllen. Deshalb muss die Aktivierung des Buttons auch das ViewModel neu initialisieren. Dies geschieht durch Zuweisung einer leeren Maßnahme zu zur Setter-Methode `vm.model` (Z. 95). Ohne die Neuinitialisierung würde die Eingabemaske immer die zuletzt eingetragene Maßnahme enthalten. Dies würde große Verwirrung beim Benutzer stiften.

Der `FloatingActionButton` erhält den Schlüssel `createNewMassnahmeButtonKey` (Z. 89). Er ist als `GlobalKey` deklariert (Z. 11). Er findet beim Integrationstest Anwendung, um den Button zu finden (Siehe Kapitel **Kapitel einfügen**).

Der Inhaltsbereich des `Scaffold` beinhaltet das Widget `StreamBuilder` (Z. 27). Er kann auf Streams horchen, die Ereignisse des Typs `Storage` übermitteln. Er horcht auf Änderungen im Model, um genau zu sein auf Änderungen des Streams `model.storage` (Z. 28). Sobald der `StreamBuilder` ein Ereignis erhält, so führt er die Methode aus, die als Argument des

Parameters `builder` hinterlegt ist. Alle Widgets außerhalb davon, wie etwa das `Scaffold`, erhalten dabei keine Aufforderung zum Neuzeichnen, sobald eine Maßnahme hinzugefügt wird. Das wirkt sich positiv auf die Laufzeit-Geschwindigkeit aus.

```

11 final createNewMassnahmeButtonKey = GlobalKey();
12
13 class MassnahmenMasterScreen extends StatelessWidget {
14   static const routeName = '/massnahmen_master';
15
16   const MassnahmenMasterScreen({Key? key}) : super(key: key);
17
18   @override
19   Widget build(BuildContext context) {
20     final model = AppState.of(context).model;
21     final vm = AppState.of(context).viewModel;
22
23     return Scaffold(
24       appBar: AppBar(
25         title: const Text('Maßnahmen Master'),
26       ),
27       body: StreamBuilder<Storage>(
28         stream: model.storage,
29         builder: (context, _) {
30           return SingleChildScrollView(
31             // ...
32           );
33         },
34       ),
35       floatingActionButton: FloatingActionButton(
36         key: createNewMassnahmeButtonKey,
37         child: const Icon(
38           Icons.post_add_outlined,
39           color: Colors.white,
40         ),
41         onPressed: () {
42           vm.model = Massnahme();
43           Navigator.of(context).pushNamed(MassnahmenDetailScreen.routeName);
44         },
45       ),
46     );
47   }
48 }

```

Listing 4.20.: Die Struktur der Klasse `MassnahmenMasterScreen`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_master.dart](#)

4.10.1. Auflistung der Maßnahmen im Übersichtsbildschirm

Der Inhalt der `builder`-Methode ist in Listing 4.21 dargestellt. Das erste Widget ist ein `SingleChildScrollView` (Z. 30). Das Argument `scrollDirection` ist nicht gefüllt, weshalb die Standardoption - die vertikale Scrollrichtung - gewählt wird. Sollte die Liste der Maßnahmen die Höhe des Fensters überschreiten, so kann der Benutzer vertikal über die Liste scrollen.

Das Kind des Scrollbereichs ist ein `Column`-Widget (Z. 31). Sie zeichnet Widgets, die als Argument des Parameters `children` gesetzt sind, von oben nach unten. Der Parameter `crossAxisAlignment` gibt an, wie die Kindelemente ausgerichtet sein sollen. `crossAxis` bedeutet dabei die zur Anzeige-Richtung entgegengesetzte Richtung. Da die `Column` vertikal zeichnet, ist mit `crossAxis` die horizontale Achse gemeint. `CrossAxisAlignment.start` beschreibt, dass Elemente entlang der horizontalen Achse an dessen Startpunkt auszurichten sind. Dadurch sind alle Elemente der Liste linksbündig.

Zuerst kommt die Auflistung der abgeschlossenen Maßnahmen. Die Überschrift `"Abgeschlossen"` (Z. 37), soll einen Abstand von jeweils 16 Pixel in alle Richtungen haben. Das ermöglicht das Widget `Padding` (Z. 35-40) und das Argument `EdgeInsets.all(16.0)`. Nach der Überschrift erscheint als zweites Element in der `Column` ein weiterer `SingleChildScrollView` (Z. 41-57), allerdings dieses Mal mit horizontaler Scroll-Richtung (Z. 42). Sollten die Informationen der Maßnahmen die Breite des Fensters überschreiten, kann der Nutzer von links nach rechts scrollen.

Die Informationen der Maßnahmen werden in einer Tabelle angezeigt. Dies übernimmt das selbstgeschriebene „Widget“ `MassnahmenTable` (Z. 45). Als erstes Argument erfolgt die Übergabe der anzuzeigenden Maßnahmen aus dem Model. `storage.value.massnahmen` gibt den aktuellen Wert des Streams des `storage`-Objekts zurück und greift auf die Liste der Maßnahmen zu. Mit der Methode `where` (Z. 47) kann ein Filter auf die Liste angewendet werden. Die übergebene anonyme Funktion (Z. 47-49) überprüft, ob der letzte Status auf fertig gesetzt ist. Dazu reicht der Vergleich der Abkürzung. Nur wenn die Bedingung erfüllt ist, bleibt die Maßnahme in der gefilterten Kollektion zurück. Ein solcher Filter gibt ein sogenanntes „lazy“ `Iterable` zurück. Erst beim Zugriff auf das Ergebnis findet der Filter Anwendung. Doch es gibt keinen Zwischenspeicher für die gefilterten Elemente. Jeder Zugriff filtert die Elemente also neu. Der Aufruf `toSet` bewirkt allerdings das Speichern der Ergebnisse in einer Menge (Z. 50). Das Resultat erhält das Widget `MassnahmenTable` zur Anzeige.

Ein weiterer Parameter ist `onSelect` (Z. 50). Als Argument kann eine Funktion mit genau einem Parameter gesetzt werden. Sollte der Benutzer in der Tabelle eine Maßnahme auswählen, so löst er damit die Funktion aus. Der erste Parameter enthält dann die ausgewählte Maßnahme. Daraufhin soll sich wieder die Eingabemaske öffnen (Z. 55-56). Dann

beinhalten die Eingabefehler jedoch die Werte der ausgewählten Maßnahme. Um das zu erreichen, reicht eine Zuweisung der Maßnahme an das `ViewModel` (Z. 51). Allerdings soll die Maßnahme zuvor ein neues letztes Bearbeitungsdatum mit dem aktuellen Zeitstempel erhalten (Z. 51-53).

Unterhalb der Rubrik der finalen Maßnahmen, listed die Übersicht die Maßnahmen, welche sich noch im Entwurf befinden (Z. 59-83). Daher ist das dritte Element der `Column` wiederum eine Überschrift: `"In Bearbeitung"` (Z. 62) gefolgt von einem weiteren horizontalen Scrollbereich (Z. 66-83) mit einer Tabelle von Maßnahmen (Z. 70-82). Der einzige Unterschied hier: die Bedingung der Filterfunktion. Dieses Mal filtert die Kollektion auf Maßnahmen in Bearbeitung (Z. 73-74).

4.11. Widget `MassnahmenTable`

Die `MassnahmenTable` ist ein `StatelessWidget` (Listing 4.22, Z. 6). Zur Anzeige eignet sich das Widget `Table` (Z. 15-31).

Im Verlauf der Erstellung der Arbeit, wurde versucht das Widget `DataTable` zu verwenden. Doch im Gegensatz zur `DataTable` erlaubt es die `Table`, unterschiedlich hohe Zeilen zu zeichnen. Die Höhe der Zeile wird dazu in Abhängigkeit von dem benötigten Inhalt der Zellen berechnet. Die Breite und Ausrichtung der Spalten kann konfiguriert werden. Die Eigenschaft `IntrinsicColumnWidth` sorgt dafür, dass die Spalten immer genau so groß sind, wie der Inhalt es benötigt (Z. 17). Zeilenumbrüche für die Texte in den Spalten sind somit nicht notwendig. `TableCellVerticalAlignment.middle` lässt die Tabelle die Inhalte zentriert darstellen (Z. 18).

Der Parameter `children` erhält als Argument eine Liste von `TableRow` Elementen (Z. 20-30). Die erste Tabellenzeile 20-23 beinhaltet die Spalten-Bezeichnungen. Jede `TableRow` hat wiederum den Parameter `children`. Das Argument bezieht sich hier auf die Zellen in der Zeile. Dabei ist wichtig, dass jede `TableRow` die gleiche Anzahl von Zellen hat. Weicht nur eine Zeile davon ab, zeichnet sich die gesamte Tabelle nicht und eine Ausnahme wird ausgelöst. Für die Spaltenbezeichnungen wurde eine Hilfsmethode kreiert: `_buildColumnHeader` (Z. 34-37). Sie zeichnet die Spalten mit einem Abstand von 8 Pixel in alle Richtungen.

Nach den Spaltenbezeichnungen folgen die Zeilen für die Daten der Maßnahmen (Z. 24-29). Die Methode `map` (Z. 24) ermöglicht es dazu durch die Liste der Maßnahmen zu iterieren und für jede Maßnahme ein Element eines völlig anderen Typs - in diesem Fall `TableRow` - zurückzugeben. Bei den vorangestellten Punkten `...` in Zeile 24 handelt sich um den spread operator. Die Filtermethode `map` und die darauffolgende Methode `toList` liefert eine Liste von `TableRow` Elementen. Die umgebende Liste der Zeilen `children` (Z. 19-30) erwartet jedoch Elemente des Typs `TableRow` und keine Elemente des Typs `List`. Der spread

operator ermöglicht alle Elemente der inneren Liste in die äußere Liste einzufügen.¹⁴

Eine weitere Hilfsmethode `_buildSelectableCell` erstellt Zellen, die anklickbar sind (Z. 39-51). Das Widget `TableRowInkWell` (Z. 41-51) kann in Tabellen verwendet werden, um einen anklickbaren Bereich zu erstellen. Beim Anklicken breitet sich ausgehend von der Position des Klicks ein Tintenklecks aus. Dabei überschreitet der Tintenklecks nicht den Bereich, der von der umgebenden Zeile begrenzt ist. Bei auslösen des Ereignisses `onTap` erfolgt die Ausführung des Callbacks `onSelect` (Z. 44) mit der ausgewählten Maßnahme. Doch zuvor muss überprüft werden, ob der Callback auch initialisiert wurde (Z. 43). Wie hier zu sehen ist, reicht es nicht aus, abzufragen, ob `onSelect` gesetzt ist. Trotzdem erfolgt keine Typ-Beförderung zu einem Typen ohne Null-Zulässigkeit, denn es handelt sich um eine Instanzvariable. Deshalb muss der Suffix `!` gesetzt sein (Siehe Grundlagenkapitel 3.2.4 Typen mit Null-Zulässigkeit).

Bei `onSelect` handelt es sich um einen Callback. An diesem Beispiel kann das Inversion of Control Entwurfsmuster visualisiert werden. Abbildung 4.5 zeigt wie die Akteure zusammenarbeiten. Der `MassnahmenMasterScreen` verwendet die `MassnahmenTable`. Die Tabelle enthält ein Objekt namens `onSelect`. Dabei handelt es sich um einen Funktions-Objekt. Anstatt eine neue Klasse mit einer beinhaltenden Funktion zu deklarieren, kann das gleiche über eine Abkürzung erreicht werden: dem Schlüsselwort `typedef` (Z. 4). Hier erlaubt es eine Funktionssignatur als eigenen Typ zu deklarieren. Der `MassnahmenMasterScreen` wiederum instanziiert genauso so ein Funktions-Objekt als anonyme Funktion (Listing 4.21, Z. 75-82). Weil es der Signatur der Typdefinition von `OnSelectCallback` entspricht, kann es der Tabelle als Argument für den Parameter `onSelect` übergeben werden.

¹⁴Vgl. 16.

```

30 return SingleChildScrollView(
31   child: Column(
32     crossAxisAlignment: CrossAxisAlignment.start,
33     children: [
34       const Padding(
35         padding: EdgeInsets.all(16.0),
36         child: Text(
37           "Abgeschlossen",
38           style: TextStyle(fontSize: 20),
39         ),
40     ),
41     SingleChildScrollView(
42       scrollDirection: Axis.horizontal,
43       child: Padding(
44         padding: const EdgeInsets.all(16.0),
45         child: MassnahmenTable(
46           model.storage.value.massnahmen
47             .where((m) =>
48               m.letzteBearbeitung.letzterStatus ==
49                 LetzterStatus.fertig.abbreviation)
50             .toSet(), onSelect: (selectedMassnahme) {
51               vm.model = selectedMassnahme.rebuild((m) => m
52                 ..letzteBearbeitung.letztesBearbeitungsDatum =
53                   DateTime.now().toUtc());
54             },
55             Navigator.of(context)
56               .pushNamed(MassnahmenDetailScreen.routeName);
57           ),
58       ),
59       const Padding(
60         padding: EdgeInsets.all(16.0),
61         child: Text(
62           "In Bearbeitung",
63           style: TextStyle(fontSize: 20),
64         ),
65     ),
66     SingleChildScrollView(
67       scrollDirection: Axis.horizontal,
68       child: Padding(
69         padding: const EdgeInsets.all(16.0),
70         child: MassnahmenTable(
71           model.storage.value.massnahmen
72             .where((m) =>
73               m.letzteBearbeitung.letzterStatus ==
74                 LetzterStatus.bearb.abbreviation)
75             .toSet(), onSelect: (selectedMassnahme) {
76               vm.model = selectedMassnahme.rebuild((m) => m
77                 ..letzteBearbeitung.letztesBearbeitungsDatum =
78                   DateTime.now().toUtc());
79             },
80             Navigator.of(context)
81               .pushNamed(MassnahmenDetailScreen.routeName);
82           ),
83       ),
84     ],
85   ),
86 );

```

Listing 4.21.: Die Ausgabe der Maßnahmen, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_master.dart](#)

```

4 typedef OnSelectCallback = void Function(Massnahme selectedMassnahme);
5
6 class MassnahmenTable extends StatelessWidget {
7   final Set<Massnahme> _massnahmenToDisplay;
8   final OnSelectCallback? onSelect;
9
10  const MassnahmenTable(this._massnahmenToDisplay, {this.onSelect, Key? key})
11    : super(key: key);
12
13  @override
14  Widget build(BuildContext context) {
15    return Table(
16      border: TableBorder.all(width: 3),
17      defaultColumnWidth: const IntrinsicColumnWidth(),
18      defaultVerticalAlignment: TableCellVerticalAlignment.middle,
19      children: [
20        TableRow(children: [
21          _buildColumnHeader(const Text("Zuletzt bearbeitet am")),
22          _buildColumnHeader(const Text("Maßnahmentitel"))
23        ]),
24        ..._massnahmenToDisplay.map((m) {
25          return TableRow(children: [
26            _buildSelectableCell(m, Text(m.letzteBearbeitung.formattedDate)),
27            _buildSelectableCell(m, Text(m.identifikatoren.massnahmenTitel)),
28          ]);
29        }).toList(),
30      ],
31    );
32  }
33
34  Widget _buildColumnHeader(Widget child) => Padding(
35    padding: const EdgeInsets.all(8.0),
36    child: child,
37  );
38
39  Widget _buildSelectableCell(Massnahme m, Widget child,
40    {double padding = 8.0}) =>
41    TableRowInkWell(
42      onTap: () {
43        if (onSelect != null) {
44          onSelect!(m);
45        }
46      },
47      child: Padding(
48        padding: EdgeInsets.all(padding),
49        child: child,
50      ),
51    );
52 }

```

Listing 4.22.: Die Klasse MassnahmenTable, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/massnahmen_table.dart](#)

Das Inversion of Control Entwurfsmuster ist auch unter dem Namen „*hollywood pattern*“ bekannt, da es ähnlich wie die typische Antwort auf eine Bewerbung für einen Hollywood Film - don't call us, we'll call you - funktioniert.¹⁵

Und genauso arbeiten der Übersichts-Bildschirm und die Tabelle zusammen. Der Übersichts-Bildschirm verwendet die Tabelle, welche nicht wissen muss, wofür sie eingesetzt wird. Sobald die Tabelle eine Selektion des Benutzers bemerkt, kommuniziert sie wieder mit dem Übersicht Bildschirm. Nun greift der Übersicht-Bildschirm über den Service Locator auf das ViewModel zu, um die selektierte Maßnahme zu übergeben.

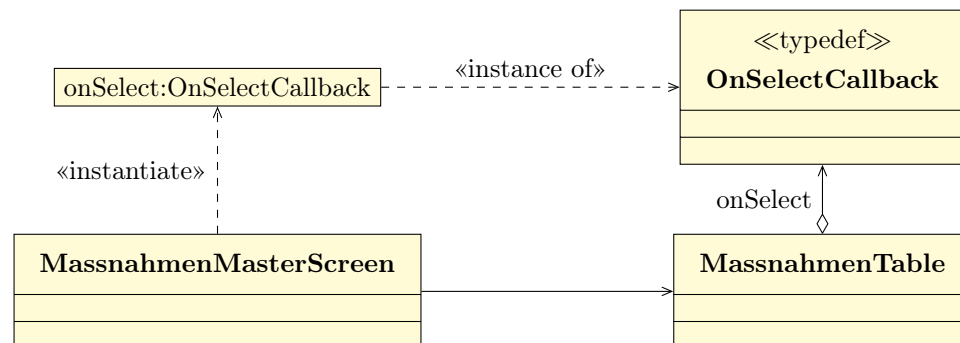


Abbildung 4.5.: UML Diagramm, Quelle: Eigene Abbildung

4.12. Das View Model

Listing 4.23 zeigt das ViewModel. Im ersten Schritt enthält es nur drei Streams vom Typ **BehaviorSubject**. Eines für den letzten Status (Z. 6), eines für die „*guid*“ (Z. 8) und eines für den Titel der Maßnahme (Z. 10). Anhand dessen wird offensichtlich, warum ein ViewModel nötig ist. Die Daten, die in der Oberfläche angezeigt werden, sind Streams, die neue Werte annehmen können. Wann immer sich ein Wert ändert, löst der Stream ein neues Ereignis aus. Auf dieses Ereignis kann der View reagieren. Das Model bietet die Eigenschaften der Maßnahmen dagegen nicht als Stream an.

Weil sich das Model und das ViewModel in ihrer Struktur unterscheiden, gibt es zwei Methoden, die die Konvertierung in beide Richtungen vornehmen. Die Setter-Methode **model** (Z. 12-18) erhält ein Objekt des Wertes des Typs **Massnahme** - das Format des Models. Die einzelnen Eigenschaften werden dann in das Format des ViewModels umgewandelt: in Streams. Darüber wird der Setter-Methode **value** von jedem **BehaviorSubject** der entsprechende Wert aus dem Model zugewiesen. Besonders ist auch, wie die Auswahloptionen sich im Model und im ViewModel unterscheiden. Im ViewModel sind es abgeleitete Objekte der Basisklasse **Choice**, wie z.B. **LetzterStatus**. Im Gegensatz dazu speichert das Modell die Optionen lediglich über die Abkürzung als String ab. Mit Hilfe der Methode **fromAbbreviation** kann anhand der Abkürzung wieder das entsprechende Objekt wiedergefunden werden (Z. 16).

¹⁵Vgl. 10.

Die Getter-Methode dagegen konvertiert in das exakte Gegenteil. Die aktuellen Werte von jedem `BehaviorSubject` werden über die Getter-Methode `value` ausgelesen und anschließend der entsprechenden Eigenschaft des Objekts vom Werte-Typ `Massnahme` gespeichert. Die Auswahloption, die für den letzten Status hinterlegt wurde, wird dabei wiederum nur als Abkürzung eingetragen. Dementsprechend ist bloß die Eigenschaft `abbreviation` abzufragen (Z. 22).

Allerdings kann bei Auswahlfeldern auch keine Option gewählt sein. Die Getter-Methode `value` kann daher also auch `null` zurück geben. Der Compiler gibt einen Fehler aus, wenn versucht wird, auf `value` eine Operation auszuführen, sollte es sich um einen Typ mit Null-Zulässigkeit handeln. So ist es bei dem Aufruf von `abbreviation` der Fall (Z. 22). Der Fehler kann nur damit behoben werden, indem das Prefix `?` der Operation vorangestellt wird. In diesem Fall wird die Methode aufgerufen, sollte `value` nicht `null` sein. Ist `value` dagegen `null`, so wird die Operation nicht ausgeführt und der gesamte Ausdruck gibt direkt `null` zurück.

```

5 class MassnahmenFormViewModel {
6   final letzterStatus = BehaviorSubject<LetzterStatus?>.seeded(null);
7
8   final guid = BehaviorSubject<String?>.seeded(null);
9
10  final massnahmenTitel = BehaviorSubject<String>.seeded("");
11
12  set model(Massnahme model) {
13    guid.value = model.guid;
14
15    letzterStatus.value = letzterStatusChoices
16      .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
17    massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
18  }
19
20  Massnahme get model => Massnahme((b) => b
21    ..guid = guid.value
22    ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
23    ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
24    ..identifikatoren
25      .update((b) => b..massnahmenTitel = massnahmenTitel.value));
26 }

```

Listing 4.23.: Die Klasse `MassnahmenFormViewModel`, Quelle: Eigenes Listing, Datei: `Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart`

4.13. Eingabeformular

Das soeben erstellte ViewModel kann nun für die Eingabemaske verwendet werden. Listing 4.24 zeigt die grundlegende Struktur der Klasse `MassnahmenDetailScreen`.

Wiederum werden das ViewModel und das Model über das „*InheritedWidget*“ `AppState` abgerufen und in die jeweiligen lokalen Variablen gespeichert (Z. 16, 17). Nachfolgend werden zwei Hilfsfunktionen innerhalb der `build`-Methode deklariert. Solche sogenannten *nested functions* - deutsch verschachtelten Funktionen - sind im Dart erlaubt, was zu einer weiteren Besonderheit führt. Der Sichtbarkeitsbereich von Variablen ist in Dart lexikalisch. Die Bindung der Variablen ist also durch den umgebenden Quelltext bestimmt. Die lokalen Variablen `model` und `vm` sind also im gesamten Bereich sichtbar, der durch die öffnenden und schließenden geschweiften Klammern der Methode `build` aufgespannt wird (Z. 15-103). Damit sind sie auch innerhalb der beiden verschachtelten Funktionen verfügbar. Innerhalb der Funktionen kann auf `model` und `vm` zugegriffen werden, ohne sie über einen Parameter übergeben zu müssen.

Das erste Widget im Inhaltsbereich des Scaffold ist ein `WillPopScope`. Es erlaubt das Verlassen einer Route an eine Abhängigkeit zu knüpfen. Bei dem Eingabeformular handelt es sich um eine Unterseite. Dadurch erscheint in der `AppBar` (Z. 47-48) links von der Überschrift ein Button, der ermöglicht, zur letzten Ansicht zurück zu navigieren (Abb. 4.2). Dabei stellt sich jedoch die Frage, was mit der bis zu diesem Zeitpunkt eingetragenen Maßnahme passieren soll. Für die Formular-Anwendung soll in diesem Fall die Maßnahme im aktuellen Zustand abgespeichert werden. Dazu wird dem Parameter `onWillPop` als Argument die Funktion `saveRecord`.

Anders als im Übersicht-Bildschirm erhält das `Scaffold` kein Argument für den Parameter `floatingActionButton`. Der Hintergrund dafür ist, dass auf diesem Bildschirm in den nächsten Schritten nicht nur ein, sondern zwei solcher Buttons zur Verfügung stehen sollen. Daher muss der Button *manual* angelegt werden. Das ist nur mit Hilfe eines `Stack`-Widgets möglich, welcher als Kind des `WillPopScope` eingetragen ist. Ein `Stack` erlaubt es mehrere Ebenen in der Tiefe anzulegen. Das unterste Element soll die Auflistung der Eingabefelder sein. Der `SingleChildScrollView` (Z. 54-79) bietet einen vertikalen Scrollbereich an, in dem die Eingabefelder in einer `Column` (Z. 58-76) untereinander aufgelistet sind. Die Ebene, die über den Eingabefeldern eingeblendet wird, soll die beiden Aktions-Buttons zeichnen. Das Widget `Align` erlaubt in dieser Ebene festzulegen, wo die Elemente angeordnet sein sollen (Z. 80-99). Wie für den `FloatingActionButton` üblich wurde die untere rechte Bildschirm-Ecke gewählt (Z. 81). Die Buttons sollen in Zukunft übereinander angeordnet sein, weshalb ein `Column`-Widget zum Einsatz kommt. Zum ersten Mal taucht der Parameter `mainAxisSizeAuf`. Mit dem Argument `MainAxisSize.min` nimmt die `Column` in der Höhe nur so viel Platz ein, wie durch die Kindelemente notwendig. Als bisher einziges Element in der `Column` taucht nun der `FloatingActionButton` auf (Z. 87-95), der die aktuell eingetra-

4. Schritt 1 - Formular in Grundstruktur erstellen

genen Daten abspeichern (Z. 92) und zur Übersicht zurückkehren soll (Z. 93). Wenn der Nutzer den Mauszeiger über diesen Button bewegt, wird ein Tooltip angezeigt: "Validiere und speichere Massnahme"(Z. 88). Der Tooltip ist als Konstante angelegt (Z. 7). Das hat vor allem den Grund, dass er auch für den folgenden Integrationstest genutzt wird. Elemente können darin über einen beinhaltenden Text oder Tooltip gefunden werden.


```

7  const saveMassnahmeTooltip = "Validiere und speichere Massnahme";
8
9  class MassnahmenDetailScreen extends StatelessWidget {
10   static const routeName = '/massnahmen-detail';
11
12   const MassnahmenDetailScreen({Key? key}) : super(key: key);
13
14   @override
15   Widget build(BuildContext context) {
16     final vm = AppState.of(context).viewModel;
17     final model = AppState.of(context).model;
18
19     Future<bool> saveRecord() {
...     // ...
28   }
29
30   Widget createMassnahmenTitelTextFormField() {
...     // ...
44   }
45
46   return Scaffold(
47     appBar: AppBar(
48       title: const Text('Maßnahmen Detail'),
49     ),
50     body: WillPopScope(
51       onWillPop: () => saveRecord(),
52       child: Stack(
53         children: [
54           SingleChildScrollView(
55             child: Center(
56               child: Padding(
57                 padding: const EdgeInsets.all(8.0),
58                 child: Column(
...                   // ...
76                 ),
77               ),
78             ),
79           ),
80           Align(
81             alignment: Alignment.bottomRight,
82             child: Padding(
83               padding: const EdgeInsets.all(16.0),
84               child: Column(
85                 mainAxisAlignment: MainAxisAlignment.min,
86                 children: [
87                   FloatingActionButton(
88                     tooltip: saveMassnahmeTooltip,
89                     heroTag: 'save_floating_action_button',
90                     child: const Icon(Icons.check, color: Colors.white),
91                     onPressed: () {
92                       saveRecord();
93                       Navigator.of(context).pop();
94                     },
95                   ),
96                 ],
97               ),
98             ),
99           ),
100         ],
101       ),
102     ));
103   }
104 }

```

Listing 4.24.: Die Struktur des Bildschirms MassnahmenDetailScreen, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart

4.13.1. Ausgabe der Formularfelder

Listing 4.25 zeigt die Ausgabe der Formularfelder in einer Column 58. Das Auswahlfeld für den letzten Status verwendet ein selbstgeschriebenes `Widget` namens `SelectionCard` (Z. 61-72). Da die Menge der Auswahloptionen auch den Namen der Liste enthält, kann er als Titel der Selektionskarte verwendet werden (Z. 62). In diesem Fall ist das der Text „Status“. Die Auswahloptionen, welche der Auswahlbildschirm anzeigen soll, sind dem Parameter `allChoices` hinterlegt 63. `allChoices` hinterlegt 63.

Die Selektionskarte soll ihren eigenen Zustand pflegen. Sie erhält dazu lediglich den initialen Wert, der aktuellen im ViewModel gespeichert ist. Bei allen Änderungen, die innerhalb der Selektionskarte erfolgen, sollen die gleichen Änderungen auch im ViewModel nachgepflegt werden. Sollte also der Wert des letzten Status im ViewModel verfügbar sein (Z. 65), so wird er als Startwert dem Parameter `initialValue` (Z. 64-67) übergeben. Dabei ist zu beachten, dass das Argument eine Menge ist. Sie wird mit den öffnenden und schließenden geschweiften Klammern erstellt. Das `collection-if` wird hier verwendet, um genau ein Element diesem `Set`-Literal hinzuzufügen, sollte es nicht `null` sein. Ist das Element allerdings `null`, so bleibt das `Set`-Literal einfach leer. Für mehr Informationen zum `Set`-Literal und dem `collection-if` siehe [Kapitel einfügen](#).

Wenn der Benutzer einer Auswahloptionen selektiert, so wird die dementsprechende anonyme Funktion aufgerufen. Sie ist für den Parameter `onSelect` hinterlegt, (Z. 68-69). Das gleiche gilt für Auswahloptionen, welche deselektiert werden (Z. 70-71). Das Auswahlfeld erlaubt nur einen Wert. Deshalb reicht es aus, den Wert bei Selektion zu ersetzen und ihn bei Deselektion zu leeren, also ihn auf `null` zu setzen.

```

58 child: Column(
59   crossAxisAlignment: CrossAxisAlignment.start,
60   children: [
61     SelectionCard<LetzterStatus>(
62       title: letzterStatusChoices.name,
63       allChoices: letzterStatusChoices,
64       initialValue: {
65         if (vm.letzterStatus.value != null)
66           vm.letzterStatus.value!
67       },
68       onSelect: (selectedChoice) =>
69         vm.letzterStatus.value = selectedChoice,
70       onDeselect: (selectedChoice) =>
71         vm.letzterStatus.value = null,
72     ),
73     createMassnahmenTitelTextFormField(),
74     const SizedBox(height: 64)
75   ],
76 ),

```

Listing 4.25.: Die Ausgabe der Formularfelder, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

4.13.2. Eingabefeld für den Maßnahmentitel

Unterhalb der ersten Selektionskarte soll das Eingabefeld für den Maßnahmentitel erscheinen (Z. 73). Listing 4.26 zeigt die Implementierung der verschachtelten Funktion zum Zeichnen dieses Eingabefeldes. Es handelt sich um das Widget `TextFormField` (Z. 34-41).

```

30 Widget createMassnahmenTitelTextFormField() {
31   return Card(
32     child: Padding(
33       padding: const EdgeInsets.all(16.0),
34       child: TextFormField(
35         initialValue: vm.massnahmenTitel.value,
36         decoration: const InputDecoration(
37           hintText: 'Maßnahmentitel', labelText: 'Maßnahmentitel'),
38         onChanged: (value) {
39           vm.massnahmenTitel.value = value;
40         },
41       ),
42     ),
43   );
44 }
```

Listing 4.26.: Die Funktion `createMassnahmenTitelTextFormField`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Hier wird klar, wovon die Selektionskarte inspiriert ist. Denn auch das `TextFormField` erhält einen initialen Wert über den Parameter `initialValue`. Sobald sich der Wert des Formularfeldes ändert, kann der neue Wert im ViewModel über die anonyme Funktion aktualisiert werden, welche dem Parameter `onChanged` übergeben wurde.

4.13.3. Speicher-Routine

Die Funktion, die dem Parameter `onWillPop` des `WillPopScope` übergeben wurde, ist in Listing 4.27 zu sehen. Die Voraussetzung für diese Funktion ist, dass ihr Rückgabotyp ein `Future<bool>` ist. Das erlaubt der Methode asynchron zu sein. Der `Future`, der von der Funktion zurückgegeben werden soll, muss in der Zukunft den Wert `true` zurückgeben, wenn dem Navigator erlaubt werden soll, zurück zu navigieren. Da die Implementierung der Methode allerdings nicht asynchron ist, soll der Wahrheitswert direkt zurückgegeben werden. Mit dem benannten Konstruktor `value` der Klasse `Future` ist es möglich, genau das zu tun 27. Der Wahrheitswert ist damit in einem `Future`-Objekt gekapselt und steht ohne Verzögerung zur Verfügung. Aktuell soll die Maßnahme lediglich abgespeichert werden (Z. 25), da noch keine Validierung erfolgt.

Der Benutzer erhält noch eine Mitteilung, dass die Maßnahme erstellt wurde. Das aktuelle `Scaffold`-Objekt kann über `ScaffoldMessenger.of` adressiert werden (Z. 20). Sollte bereits eine Mitteilung vorliegen, wird diese wieder versteckt, um Platz für die neue zu machen (Z.

21). Anschließend wird eine sogenannte `SnackBar` mit dem entsprechenden Text angezeigt (Z. 22-23).

```
19 Future<bool> saveRecord() {  
20   ScaffoldMessenger.of(context)  
21     ..hideCurrentSnackBar()  
22     ..showSnackBar(  
23       const SnackBar(content: Text('Massnahme wird gespeichert ...')));  
24  
25   model.putMassnahmeIfAbsent(vm.model);  
26  
27   return Future.value(true);  
28 }
```

Listing 4.27.: Die Funktion `saveRecordAndGoBackToOverviewScreen`, Quelle: Eigenes Listing,
Datei: `Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/
massnahmen_detail.dart`

4.14. Widget SelectionCard

Das Listing 4.28 zeigt die Struktur des Widgets SelectionCard. Die Klasse hat einen generischen Typparameter (Z. 15). `<ChoiceType extends Choice>` bedeutet, dass die SelectionCard nur für Typen verwendet werden kann, die von `Choice` erben. Das ist eine wichtige Voraussetzung, da auf den übergebenen Werten Operationen ausgeführt werden sollen, die nur `Choice` unterstützt. Alle Parameter, die dem Konstrukt übergeben werden, leiten ebenso von diesen Typparameter ab. Einzige Ausnahme dabei ist der `title` 16.

```

7 typedef OnSelect<ChoiceType extends Choice> = void Function(
8     ChoiceType selectedChoice);
9
10 typedef OnDeselect<ChoiceType extends Choice> = void Function(
11     ChoiceType selectedChoice);
12
13 const confirmButtonTooltip = 'Auswahl übernehmen';
14
15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16     final String title;
17     final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
18     final Choices<ChoiceType> allChoices;
19     final OnSelect<ChoiceType> onSelect;
20     final OnDeselect<ChoiceType> onDeselect;
21
22     SelectionCard(
23         {required this.title,
24         required Iterable<ChoiceType> initialValue,
25         required this.allChoices,
26         required this.onSelect,
27         required this.onDeselect,
28         Key? key})
29         : selectionViewModel = BehaviorSubject<BuiltSet<ChoiceType>>.seeded(
30             BuiltSet.from(initialValue)),
31           super(key: key);

```

Listing 4.28.: Die Klasse SelectionCard, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

Mit dem Stream `selectionViewModel` verwaltet die `SelectionCard` ihren eigenen Zustand. Der Stream ist mit dem generischen Typen `BuiltSet<ChoiceType>` konfiguriert. Das macht es unmöglich, den aktuell hinterlegten Wert anzupassen, ohne das Gesamtobjekt auszutauschen. Der Tausch des Objekts wiederum bewirkt, dass ein Ereignis über den Stream ausgelöst wird. Über dieses Ereignis zeichnet die SelectionCard Teile seiner Oberfläche neu. Allerdings erhält der Konstruktor kein Argument des Typs `BehaviorSubject` sondern stattdessen vom `Iterable<ChoiceType>` (Z. 24). Damit wird der Benutzer nicht darauf eingeschränkt, einen Stream zu übergeben. Er kann auch eine gewöhnliche Liste oder Menge setzen. Die Umwandlung der ankommenden Kollektion erfolgt in der Initialisierungsliste 29-30. Nur so ist es möglich, die Instanzvariable mit `final` als unveränderbar zu kennzeichnen. Initialisierungen solcher Variablen müssen im statischen Kontext der Objekterstellung geschehen. Der Konstruktor-Körper gehört dagegen nicht mehr zur statischen Teil. Im Konstruktor-Körper können Operationen der Instanz verwendet werden, denn das Objekt existiert bereits. Der Versuch eine mit `final` gekennzeichnete Instanzvariable im

Konstruktor-Körper zu setzen, führt zu einem Compilerfehler in Dart. Der Konstruktor `seeded` des `BehaviorSubject` wird mit einem `BuiltSet` gefüllt (Z. 29). Dieses wiederum wird mit dem benannten Konstruktor `from` von `BuiltSet` mit der Kollektion aufgerufen (Z. 30). Er wandelt die Liste in eine unveränderbare Menge um. Die Liste aller Auswahloptionen `allChoices` (Z. 18) gewährleistet über den generischen Typ-Parameter, das nicht aus versehen Auswahloptionen übergeben werden, die nicht zum Typ der `SelectionCard` passen. Die Rückruf-Funktionen (Z. 19, 20) die bei Selektion und Deselektion von Optionen ausgelöst werden, bieten einen besonderen Vorteil, dadurch, dass sie mit dem generischen Typen konfiguriert sind. Die Signaturen der Rückruf-Funktionen (Z. 7-8, 10-11) geben nämlich vor, dass der erste Parameter vom Typen `ChoiceType` sein muss. Wenn nun der Benutzer der `SelectionCard` einen Typ wie etwa `LetzterStatus` für den Typparameter übergibt, so erhält er auch eine Rückruffunktion, dessen erster Parameter vom Typ `LetzterStatus` ist. Ohne eine Typumwandlung - englisch type casting - von (Z. Choice) in `LetzterStatus`, können Operationen auf das Objekt angewendet werden, die nur `LetzterStatus` unterstützt.

Das erste Element, welches von der `build`-Methode zurückgeben wird, ist ein `StreamBuilder` (Listing 4.29, Z. 47). Er horcht auf das `selectionViewModel` (Z. 48). Sobald also eine Selektion getätigt wurde, aktualisiert sich auch die dazugehörige Karte. Das Aussehen einer Karte wird durch das Widget `Card` erreicht (Z. 51). Dadurch erhält es abgerundete Ecken und einen Schlagschatten, der es vom Hintergrund abgrenzt. Ein `ListTile` Widget erlaubt es dann, den übergebenen `title` als Überschrift zu setzen (Z. 54) und die aktuell ausgewählten Selektionen als Untertitel anzuzeigen (Z. 56). Zu diesem Zweck wandelt die Methode `map` alle Elemente von `selectedChoices` in `String`-Objekte um, indem es von dem `Choice`-Objekt lediglich den Beschreibungstext `description` verwendet. Anschließend sammelt der Befehl `join` die resultierende `String`-Objekte ein, formt sie in einen gemeinsamen `String` zusammen und trennt sie darin jeweils mit einem `", "` voneinander.

Das `ListTile` erhält ein `FocusNode`-Objekt (Z. 53), damit der Benutzer beim Zurücknavigieren von der Unterseite im Formular wieder in der gleichen vertikalen Position der Karte landet, die er zuvor ausgewählt hat. Der Benutzer würde ansonsten in Formular wieder an der obersten Position herauskommen. Der `FocusNode` wird einmal zu Anfang der `build`-Methode erstellt (Z. 35). Damit ist er außerhalb des `StreamBuilder` und bleibt somit beim Neuzeichnen der Karte erhalten.

```

34 Widget build(BuildContext context) {
35   final focusNode = FocusNode();
36
37   navigateToSelectionScreen() async {
38     focusNode.requestFocus();
39
40     Navigator.push(
41       context,
42       MaterialPageRoute(
43         builder: (context) =>
44           createMultipleChoiceSelectionScreen(context)));
45   }
46
47   return StreamBuilder(
48     stream: selectionViewModel,
49     builder: (context, snapshot) {
50       final selectedChoices = selectionViewModel.value;
51       return Card(
52         child: ListTile(
53           focusNode: focusNode,
54           title: Text(title),
55           subtitle:
56             Text(selectedChoices.map((c) => c.description).join(", ")),
57           trailing: const Icon(Icons.edit),
58           onTap: navigateToSelectionScreen,
59         ),
60       );
61     });
62 }

```

Listing 4.29.: Die Build Methode der SelectionCard, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

Klickt der Benutzer die Karte an, navigiert er schließlich zur Unterseite, wo er die Auswahloptionen präsentiert bekommt. Die verschachtelte Funktion `navigateToSelectionScreen` kommt dafür zum Einsatz (Z. 37-45). Da das Wechseln zu Unterseite bevorsteht, fordert der `focusNode` den Fokus für das angeklickte `ListTile` an (Z. 38). Schließlich navigiert der Benutzer mit `Navigator.push` zur Unterseite. Es handelt sich um den Auswahlbildschirm, auf dem der Benutzer die gewünschte Option anwählen kann. Die Besonderheit dieses Mal: die Route ist nicht als Widget deklariert und wird nicht über einen Namen aufgerufen, so wie es bei dem Übersichtsbildschirm und der Eingabemaske war. Stattdessen baut eine Funktion bei jedem Aufruf die Seite neu gebaut. Das dynamische Bauen der Seite hat einen besonderen Vorteilen, der am Listing 4.30 erklärt wird.

4.14.1. Bildschirm für die Auswahl der Optionen

Die Funktion `createMultipleChoiceSelectionScreen` (Listing 4.30) gibt einen `Scaffold` zurück, der die gesamte Seite enthält (Z. 65). Das erste Kind des `Scaffold` ist wiederum ein `StreamBuilder` (Z. 69). Hier wird der Vorteil der dynamischen Erzeugung der Seite offensichtlich: die Unterseite kann das gleiche ViewModel wiederverwenden, welches auch von der `SelectionCard` genutzt wird. Auch alle weiteren Instanzvariablen der `SelectionCard` können wiederverwendet werden. Würde es sich stattdessen um eine weitere Route handeln, so müssten alle diese Informationen über den Navigator zur neuen Unterseite übergeben werden. Sollte der Nutzer die Auswahl beenden, so müsste auch ein Mechanismus für das Zurückgeben der selektierten Daten implementiert werden. Dadurch, dass die `SelectionCard` und der Auswahlbildschirm sich das gleiche ViewModel teilen, kann sogar ein weiterer Vorteil in Zukunft genutzt werden: in einem zweispaltigen Layout könnte auf der linken Seite die Eingabemaske und auf der rechten Seite der Bildschirm der Auswahloptionen eingeblendet werden. Sobald sich Auswahloptionen im rechten Auswahl Bildschirm verändern, so würden sich die Änderungen auf der linken Seite für den Benutzer direkt widerspiegeln.

Innerhalb des `StreamBuilder` werden die Auswahloptionen gebaut. Dazu speichert die lokale Variable `selectedChoices` die aktuellen Selektionen des Streams zunächst zwischen (Z. 72). Die Optionen werden in einem `ListView` präsentiert (Z. 73). Er ermöglicht es, Listen-Elemente in einem vertikalen Scrollbereich darzustellen. Die Funktion `map` konvertiert alle Objekte in der Liste aller möglichen Optionen `choices` in Elemente des Typs `CheckboxListTile` (Z. 74-98). In der Standard-Variante sind die Checkboxes rechtsbündig. Der Parameter `controlAffinity` kann genutzt werden, um dieses Verhalten zu überschreiben (Z. 80).

Das `CheckboxListTile` erhält einen Titel, der aus dem Beschreibungstext `description` des `Choice`-Objekts gebildet wird (Z. 81). Ob eine Option aktuell bereits ausgewählt ist, kann mit dem Parameter `value` übertragen werden (Z. 82). Sollte sich die Selektion ändern, erfolgt die Mitteilung über die Rückruffunktion `onChanged` (Z. 83-94). Der erste Parameter der anonymen Funktion gibt dabei die ausgewählte Selektion an. Eine Fallunterscheidung

überprüft zunächst, ob der Parameter `selected` nicht `null` ist, denn sein Parametertyp `bool?` lässt Null-Werte zu. Durch die Typ-Beförderung ist `selected` innerhalb des Körpers der Fallunterscheidung dann vom Typ `bool` (Z. 84-94).

Darin wird zunächst der Zustand des ViewModels der `SelectionCard` aktualisiert. Die `replace`-Methode des „*Builder*“-Objekts kann die gesamte Kollektion im `BuiltSet` austauschen, ungeachtet dessen, dass es sich beim Argument selbst nicht um ein `BuiltSet` handelt. Die `replace`-Methode wandelt das Argument dafür automatisch um. Durch Zuweisung des neuen Wertes erhält das ViewModel der `SelectionCard` ein neues Ereignis. Damit wird die `SelectionCard` und der dazugehörige Auswahlbildschirm aktualisiert. Während der Erstellung dieser Arbeit wurde versucht, die `SelectionCard` als ein `StatefulWidget` zu erstellen. Mittels `setState` sollte dafür gesorgt werden, dass sowohl `SelectionCard` als auch der Auswahlbildschirm aktualisiert werden. Doch bei diesem Vorgehen zeichnet sich nur die `SelectionCard` neu. Der Auswahlbildschirm bleibt unverändert, denn er wird zwar von der `SelectionCard` gebaut, doch ist er nicht tatsächlich Kind der `SelectionCard`. In Wahrheit ist der Auswahlbildschirm ein Kind von `MaterialApp` - genau wie `MassnahmenMasterScreen` und `MassnahmenDetailScreen`.

Neben dem ViewModel der `SelectionCard` muss jedoch auch das ViewModel der Eingabemaske aktualisiert werden. Mit den Rückruffunktionen `onSelect` (Z. 90) und `onDeselect` (Z. 92) hat die aufrufende Ansicht die Möglichkeit, auf Selektionen zu reagieren.

Schließlich ist noch der `FloatingActionButton` Teil der Unterseite (Z. 99-103). Mit einem Klick darauf gelangt der Benutzer zurück zur Eingabemaske (Z. 100).

```

64 Widget createMultipleChoiceSelectionScreen(BuildContext context) {
65   return Scaffold(
66     appBar: AppBar(
67       title: Text(title),
68     ),
69     body: StreamBuilder(
70       stream: selectionViewModel,
71       builder: (context, snapshot) {
72         final selectedChoices = selectionViewModel.value;
73         return ListView(children: [
74           ...allChoices.map((ChoiceType c) {
75             bool isSelected = selectedChoices.contains(c);
76
77             return CheckboxListTile(
78               key: Key(
79                 "valid choice ${allChoices.name} - ${c.abbreviation}"),
80               controlAffinity: ListTileControlAffinity.leading,
81               title: Text(c.description),
82               value: isSelected,
83               onChanged: (selected) {
84                 if (selected != null) {
85                   selectionViewModel.value =
86                     selectionViewModel.value.rebuild((b) {
87                       b.replace(isSelected ? [] : [c]);
88                     });
89                 if (selected) {
90                   onSelect(c);
91                 } else {
92                   onDeselect(c);
93                 }
94               }
95             ));
96           }).toList(),
97         ]);
98     },
99     floatingActionButton: FloatingActionButton(
100       onPressed: () => Navigator.of(context).pop(),
101       tooltip: confirmButtonTooltip,
102       child: const Icon(Icons.check),
103     ),
104   );
105 }
106 }

```

Listing 4.30.: Die Funktion `createMultipleChoiceSelectionScreen`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

4.15. Integrations-Test zum Test der Oberfläche

Ein automatisierter Integrationstest soll verifizieren, dass die Oberfläche wie vorgesehen funktioniert. Der Integrationstest simuliert einen Benutzer, der die Applikation verwendet, um eine Maßnahme einzutragen. Bei Abschluss des Tests soll überprüft werden, ob die eingegebenen Daten mit den Inhalten der JSON-Datei übereinstimmen.

Flutter erlaubt über einen eigenen Testtreiber solche Integrationstest durchzuführen. Dabei wird die Applikation zur Ausführung gebracht, und jeder Schritt so visualisiert, wie es bei der Ausführung der realen Applikation der Fall wäre. Der Entwickler hat damit die Möglichkeit, die Eingaben und Interaktionen zu beobachten und gegebenenfalls zu bemerken, warum ein Testfall nicht korrekt ausgeführt wird.

Das Ergebnis des Integrationstests soll allerdings nicht mit der tatsächlich geschriebenen JSON-Datei überprüft werden. Der Test soll nicht tatsächlich Daten auf der Festplatte speichern. Das würde die Gefahr bergen, dass vergangene Eingaben manipuliert werden. Stattdessen soll der Test in einer Umgebung stattfinden, die keine Auswirkung auf die Haupt-Applikation oder zukünftige Tests haben soll. Zu diesem Zweck können sogenannte Mocks genutzt werden. Das Paket „*mockito*“ erlaubt über Annotationen solche Mocks für die gewünschten Klassen über Quellcode-Generierung zu erstellen.

Integrationstests werden im Ordner `integration_test` angelegt. Während des Zeitpunkts der Erstellung dieser Arbeit war es in der Standardkonfiguration der Quellcode-Generierung und dem Paket „*mockito*“ nicht möglich, Mocks auch im `integration_test` Ordner zu generieren. Lediglich innerhalb des `test` Ordners, der für die Unit-Tests vorgesehen ist, hat die Annotation `generate mocks` funktioniert. Zu diesem Fehlverhalten existiert ein entsprechendes Issue im GitHub Repository des Mockito packages. [Ref](#) Um das Generieren von Mocks auch für Integrationstest verfügbar zu machen, hat der Autor dieser Arbeit einen entsprechenden Lösungsansatz recherchieren und im Issue beschrieben. [Ref](#)

Damit der `integration_test` Ordner für die Quellcode-Generierung der Mocks integriert wird, muss ein entsprechender Eintrag in der Build-Konfiguration vorgenommen werden. Damit das Paket „*source_gen*“ die entsprechenden Dateien analysiert, müssen sie in der Rubrik `sources` angegeben werden (Listing 4.32, Z. 3-8). Wird der Ordner `integration_test` darin eingefügt (Z. 8), bezieht „*source_gen*“ den Ordner in der Quellcode-Generierung mit ein. Zusätzlich dazu muss die Rubrik `generate_for` von dem `mockBuilder` des „*mockito*“-Pakets (Z. 11-13) um die gleiche Angabe des Ordners ergänzt werden (Z. 13).

Anschließend kann mit der Annotation `and generate mocks` (Listing 4.32, Z. 20) ein Mock für `MassnahmenJsonFile` angefordert werden. In der Kommandozeile ist `flutter pub run build_runner build` einzugeben, damit der entsprechende Quellcode generiert wird. Mit dem Mock kann der Integrationstest ausgeführt werden, ohne dass befürchtet werden muss, dass die JSON-Datei

4. Schritt 1 - Formular in Grundstruktur erstellen

```
1 targets:
2   $default:
3     sources:
4       - $package$
5       - lib/$lib$
6       - lib/**/*.dart
7       - test/**/*.dart
8       - integration_test/**/*.dart
9     builders:
10      mockito|mockBuilder:
11        generate_for:
12          - test/**/*.dart
13          - integration_test/**/*.dart
```

Listing 4.31.: Initialisierung des Integrations Tests, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/build.yaml](#)

tatsächlich beschrieben wird. Stattdessen kann darauf gehorcht werden, wenn Operationen auf dem Objekt ausgeführt werden.

```
18 const durationAfterEachStep = Duration(milliseconds: 1);
19
20 @GenerateMocks([MassnahmenJsonFile])
21 void main() {
22   testWidgets('Can fill the form and save the correct json', (tester) async {
23     final binding = IntegrationTestWidgetsFlutterBinding.ensureInitialized()
24       as IntegrationTestWidgetsFlutterBinding;
25     binding.framePolicy = LiveTestWidgetsFlutterBindingFramePolicy.fullyLive;
26
27     final massnahmenJsonFileMock = MockMassnahmenJsonFile();
28     when(massnahmenJsonFileMock.readMassnahmen()).thenAnswer((_) async => {});
```

Listing 4.32.: Initialisierung des Integrations Tests, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Die Funktion `testWidgets` startet den Test und erhält als ersten Parameter das `tester`-Objekt (Z. 22). Darüber ist die Interaktion mit der Oberfläche während des Tests möglich. In den Zeilen 22 bis 25 wird der Testtreiber initialisiert. [Ref](#). Anschließend wird ein Objekt der generierten Klasse `MockMassnahmenJsonFile` erstellt. Wenn das Model nun während der Applikation versucht, aus der JSON-Datei zu lesen, soll der Mock eine leere Liste von Maßnahmen zurückgeben (Z. 28). Dazu wird die entsprechende Methode `when` verwendet. Als erster Parameter wird die Methode `readMassnahmen` des Mocks übergeben. Im darauffolgenden Aufruf `thenAnswer` kann angegeben werden, welche Rückgabe die Methode liefern soll.

Über den `tester` kann mit Hilfe der Methode `pumpWidget` ein beliebiges Widget in der Test-Ausführung konstruiert werden. In diesem Fall ist es die gesamte Applikation, die getestet werden soll. Dementsprechend ist hier erneut der komplette Haupteinstiegspunkt angegeben (Listing 4.33). Doch der Konstruktor von (Z. `MassnahmenModel`) erhält dieses Mal nicht das `MassnahmenJsonFile`, sondern den entsprechenden Mock (Z. 31).

Weil während des Integrationstest immer wieder die gleichen Operationen wie das Selektieren einer Selektions-Karte, das Auswählen einer Option, das Anklicken des Buttons zum

```

30 await tester.pumpWidget(AppState(
31   model: MassnahmenModel(massnahmenJsonFileMock),
32   viewModel: MassnahmenFormViewModel(),
33   child: MaterialApp(
34     title: 'Maßnahmen',
35     theme: ThemeData(
36       primarySwatch: Colors.lightGreen,
37       accentColor: Colors.green,
38       primaryIconTheme: const IconThemeData(color: Colors.white),
39     ),
40     initialRoute: MassnahmenMasterScreen.routeName,
41     routes: {
42       MassnahmenMasterScreen.routeName: (context) =>
43         const MassnahmenMasterScreen(),
44       MassnahmenDetailScreen.routeName: (context) =>
45         const MassnahmenDetailScreen()
46     },
47   ));

```

Listing 4.33.: Initialisierung des Widgets für den Integrations Tests, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Akzeptieren der Auswahl und das Füllen eines Eingabefeldes auftauchen, wurden entsprechende Hilfsfunktionen erstellt.

Der Funktion `tabSelectionCard` (Listing 4.35) benötigt lediglich die Liste der Auswahloptionen `choices`, die ihr hinterlegt ist.

```

49 Future<void> tabSelectionCard(Choices choices) async {
50   final Finder textLabel = find.text(choices.name);
51   expect(textLabel, findsWidgets);
52
53   final card = find.ancestor(of: textLabel, matching: find.byType(Card));
54   expect(card, findsOneWidget);
55
56   await tester.ensureVisible(card);
57   await tester.tap(card);
58   await tester.pumpAndSettle(durationAfterEachStep);
59 }

```

Listing 4.34.: Die Hilfsmethode `tabSelectionCard`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Um Objekte während des Testens in Oberfläche zu finden, stellt die Klasse `Finder` nützliche Funktionalitäten zur Verfügung. `Finder`-Objekte können über Fabrikmethoden des Objekts `find` abgerufen werden.

Fabrikmethoden Bei der Fabrikmethode handelt es sich um ein klassenbasiertes Erzeugungsmuster. Anstatt ein Objekt einer Klasse direkt über einen Konstruktor zu erstellen, erlaubt ein Erzeuger das Objekt zu konstruieren. Dabei entscheidet der Erzeuger darüber, welche Implementierung der Klasse zurückgegeben wird. Der aufrufende Kontext muss die konkrete Klasse dazu nicht kennen.¹⁶ Er arbeitet lediglich mit der Schnittstelle. In diesem

¹⁶Vgl. 11, S. 107–116.

Fall ist `find` dieser Erzeuger. Über die Fabrikmethode `text` wird ein `_TextFinder` konstruiert, jedoch über die Schnittstelle `Finder` zurückgegeben. Eine weitere Fabrikmethode ist `ancestor`. Sie gibt einen `_AncestorFinder` zurück, welcher ebenso hinter der Schnittstelle `Finder` versteckt wird. **Ref.** Die Fabrikmethoden werden hier deshalb verwendet, weil sie die Lesbarkeit verbessern. Anstatt `Finder titel = new _TextFinder("Maßnahmentitel")` ist `Finder titel = find.text("Maßnahmentitel")` deutlich leichter zu erfassen.

Um die Selektions-Karten zu finden, wird lediglich der Titel- Text benötigt. Angenommen der Test ruft `tabSelectionCard` mit dem Argument `letzterStatusChoices` auf, so entspricht `choices.name` dem String `"Status"`. Der Ausdruck `find.text("Status")` lokalisiert den Titel innerhalb der Selektions-Karte (Z. 50).

Die Funktion `expect` erwartet als ersten Parameter einen `Finder` und als zweiten einen sogenannten „*Matcher*“ (Z. 51). Der Aufruf von `expect` mit dem entsprechenden `Finder`-Objekt und dem `Matcher` `findsWidgets` verifiziert, dass mindestens ein entsprechendes Text Element gefunden wurde.

Wurde das Text-Element gefunden, so muss noch den Vater gesucht werden, der vom Typ `Card` ist (Z. 53). Das kann mit `find.ancestor` erfolgen. Über den Parameter `of` erhält er den `Finder` des Kind-Elements und der Parameter `matching` erhält als Argument die Voraussetzung, die vom Vater-Objekt erfüllt werden soll, als weiteren `Finder`. `find.byType(Card)` sucht also alle Elemente vom Typ `Card`. `find.ancestor` sucht anschließend alle Entsprechungen, in der eine `Card` ein Vater des `Finder titleLabel` ist. Wiederum überprüft die Funktion `expect`, dass die Karte gefunden wurde. Doch dieses Mal muss es genau ein Widget sein, welches mit dem „*Matcher*“ `findsOneWidget` verifiziert werden kann (Z. 54). Sollte mehr als nur eine Karte gefunden werden, so wäre nicht klar, welche geklickt werden soll.

Um eine Karte tatsächlich anzuwählen muss sie im sichtbaren Bereich sein. Die Methode „*ensureVisible*“ scrollt den Bildschirm zur entsprechenden Position, damit die Karte sichtbar ist (Z. 56). Schließlich sorgt `tab` mit dem `Finder card` dafür, dass die Karte ausgewählt wird. `pumpAndSettle` (Z. 58) ist eine obligatorische Methode, die nach jeder Aktion durchgeführt werden muss. Sie sorgt dafür, dass der Test so lange pausiert, bis alle Aktionen in der Oberfläche und damit auch alle angestoßenen Animationen vorüber sind. Zusätzlich kann eine Dauer angegeben werden, die darüber hinaus gewartet werden soll.

`tabConfirmButton` funktioniert ähnlich (Listing 4.35). Das Finden des Buttons ist jedoch einfacher, da es nur einen Button zum Akzeptieren auf jeder Oberfläche gibt. Der Button enthält keinen Text, lässt sich aber auch über seinen Tooltip lokalisieren (Z. 62). Die Hilfsfunktion klickt den Button (Z. 63) und wartet dann erneut auf Vollendung aller angestoßenen Animationen (Z. 64).

Ist der Integrationstest aktuell in dem Auswahlbildschirm, so sorgt `tabOption` dafür, dass Auswahloptionen gewählt wird (Listing 4.36). Dazu wird die gewünschte Option dem Pa-

```

61 Future<void> tabConfirmButton() async {
62   var confirmChoiceButton = find.byTooltip(confirmButtonTooltip);
63   await tester.tap(confirmChoiceButton);
64   await tester.pumpAndSettle(durationAfterEachStep);
65 }

```

Listing 4.35.: Die Hilfsmethode `tabConfirmButton`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

parameter `choice` übergeben. Um die Checkbox der Option zu finden, muss jedoch zunächst der Text der Auswahloption gefunden werden (Z. 68). Erst wenn verifiziert wurde, dass auch nur genau ein Label mit diesem Text existiert, läuft der Test weiter (Z. 69).

```

67 Future<void> tabOption(Choice choice, {bool tabConfirm = false}) async {
68   final choiceLabel = find.text(choice.description);
69   expect(choiceLabel, findsOneWidget);
70
71   await tester.ensureVisible(choiceLabel);
72   await tester.tap(choiceLabel);
73   await tester.pumpAndSettle(durationAfterEachStep);
74
75   if (tabConfirm) {
76     await tabConfirmButton();
77   }
78 }

```

Listing 4.36.: Die Hilfsmethode `tabOption`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Ein Klick auf das Text-Label reicht bereits aus, denn damit wird das Vater-Element - das `CheckboxListTile` - ebenfalls getroffen. Der `tester` holt es in den sichtbaren Bereich 71, klickt es 72 und wartet auf Abschluss aller Animationen (Z. 73). Sollte der optionale Parameter `tabConfirm` auf `true` gesetzt sein (Z. 75), so wird der Auswahlbildschirm anschließend direkt wieder geschlossen, nachdem die Option ausgewählt wurde (Z. 76).

Schließlich kann mit der Hilfsfunktionen `fillTextFormField` ein Formularfeld über dessen Titel gefunden und der entsprechende übergebende Text eingetragen werden (Listing 4.37). Sie findet das `TextFormField`, indem es zunächst nach dem Titel mit `find.text(title)` und anschließend dessen Vater-Element vom Typ `TextFormField` sucht (Z. 83). Sollte sowohl der Hinweistext als auch der Titel den gleichen Text enthalten, so kann es sein, dass zwei solche Elemente gefunden werden. In Wahrheit ist es aber zwei Mal dasselbe `TextFormField`. Mit `.first` wird lediglich das erste Element geliefert (Z. 85). Nachdem feststeht, dass das Element existiert (Z. 85) und es in den sichtbaren Bereich gescrollt wurde (Z. 87), gibt der Integrationstest den gewünschten Text in das Eingabefeld ein (Z. 88). Anschließend wird erneut auf Abschluss aller Animationen gewartet (Z. 89).

Während der Integrationstest startet, öffnet sich als Erstes der Übersichts-Bildschirm. Zunächst wird gewartet, dass alle Widgets korrekt initialisiert wurden (Listing 4.38, Z. 92). Es folgt der Klick auf den Button zum Erstellen einer neuen Maßnahme (Z. 95). Dazu wird der Button über den entsprechenden `key` gefunden (Z. 94). Vor allem jetzt ist das Abwar-

4. Schritt 1 - Formular in Grundstruktur erstellen

```
80 Future<void> fillTextFormField(  
81     {required String title, required String text}) async {  
82     final textFormField = find  
83         .ancestor(of: find.text(title), matching: find.byType(TextFormField))  
84         .first;  
85     expect(textFormField, findsOneWidget);  
86  
87     await tester.ensureVisible(textFormField);  
88     await tester.enterText(textFormField, text);  
89     await tester.pumpAndSettle(durationAfterEachStep);  
90 }
```

Listing 4.37.: Die Hilfsmethode `fillTextFormField`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

ten mittels `pumpAndSettle` (Z. 96) unablässig, denn es wird auf einen anderen Bildschirm navigiert. Angenommen der Test wartet nicht ab, so würden die Aktionen noch immer auf den Elementen des alten Bildschirms Anwendung finden.

```
92 await tester.pumpAndSettle(durationAfterEachStep);  
93  
94 var createNewMassnahmeButton = find.byKey(createNewMassnahmeButtonKey);  
95 await tester.tap(createNewMassnahmeButton);  
96 await tester.pumpAndSettle(durationAfterEachStep);
```

Listing 4.38.: Der Button zum Kreieren einer Maßnahme wird ausgelöst, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Der Integrationstest öffnet nun den Auswahl-Bildschirm, in dem die Selektions-Karte zum Setzen des letzten Statuses angewählt wird (Listing 4.39, Z. 98). Anschließend fällt die Wahl auf die Option für „*abgeschlossen*“ (Z. 98). Dabei sorgt `tabConfirm: true` für die sofortige Rückkehr zum Eingabeformular nach der Auswahl.

```
98 await tabSelectionCard(letzterStatusChoices);  
99 await tabOption(LetzterStatus.fertig, tabConfirm: true);
```

Listing 4.39.: Der letzte Status wird ausgewählt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Nachfolgend soll der Test das Eingabefeld für den Maßnahmen-Titel überprüfen (Listing 4.40). Es erfolgt die Erstellung eines beispielhaften Titels anhand des aktuellen Datums und der aktuellen Uhrzeit (Z. 101, 102). Der erstellte Text dient als Eingabe für das Eingabefeld (Z. 104).

Die nötigen Eingaben sind erfolgt. Daher kann der Test nun den Klick auf den Button zum Speichern simulieren (Listing ??, Z. 106-108). Dadurch würde in der Anwendung nun das Speichern der Maßnahmen in der JSON-Datei erfolgen. Doch da stattdessen ein Mock verwendet wurde, passiert dies nicht. Das Model ruft aber dennoch die entsprechenden Methoden - wie zum Beispiel `saveMassnahmen` - auf. Die Methoden haben nur nicht die ursprüngliche Funktion. Stattdessen protokollieren sie sowohl die Aufrufe, als auch die übergebenen Argumente. Durch die Methode `verify` (Z. 111) kann überprüft werden, ob die entsprechende Methode `saveMassnahmen` ausgeführt wurde. Der „*Matcher*“ `captureAny`


```

101 final now = DateTime.now();
102 var massnahmeTitle =
103     "Test Maßnahmen ${now.year}-${now.month}-${now.day} ${now.hour}:${now.minute}";
104 await fillTextFormField(title: "Maßnahmentitel", text: massnahmeTitle);

```

Listing 4.40.: Der Maßnahmentitel wird eingegeben, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

ermöglicht die Überprüfung auf irgendeine Übergabe und stellt die übergebenen Argumente über den Rückgabewert bereit.

```

106 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
107 await tester.tap(saveMassnahmeButton);
108 await tester.pumpAndSettle(durationAfterEachStep);
109
110 var capturedJson =
111     verify(massnahmenJsonFileMock.saveMassnahmen(captureAny)).captured.last;
112
113 var actualMassnahme = capturedJson['massnahmen'][0] as Map;
114 actualMassnahme.remove("guid");
115 actualMassnahme["letzteBearbeitung"].remove("letztesBearbeitungsDatum");
116
117 var expectedJson = {
118     'letzteBearbeitung': {'letzterStatus': 'fertig'},
119     'identifikatoren': {'massnahmenTitel': massnahmeTitle},
120 };
121
122 expect(actualMassnahme, equals(expectedJson));

```

Listing 4.41.: Validierung des Testergebnisses, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Die Rückgabe ist vom Typ `VerificationResult` und enthält eine Getter-Methode mit dem Namen `captured`. Dabei handelt es sich um eine Liste aller Argumente, die in den vergangenen Aufrufen übergeben wurden. Mit `last` lässt sich auf das Argument des letzten Aufrufes zurückgreifen.

Nun soll sich zeigen, ob das übergebene Argument mit dem erwarteten Wert übereinstimmt. Weil das Ergebnis eine Liste mit lediglich einer Maßnahme ist, soll auch ausschließlich diese Maßnahme verglichen werden. Der Schlüssel `'massnahmen'` greift auf die Liste zurück und der Schlüssel `0` auf die erste und einzige Maßnahme. Die lokale Variable `actualMassnahme` speichert sie zwischen (Z. 113).

Es ist unklar, welche zufällige `guid` bei der Erstellung der Maßnahme generiert wurde. Auch der Zeitstempel hinter dem Schlüssel `"letzteBearbeitung"` ist unbekannt. Eine mögliche Lösung wären weitere Mocks, welche die Erstellung der `guid` und des Datums überwachen und - anstelle einer zufälligen - immer die gleiche Zeichenkette zurückgibt. Es ist jedoch auch möglich, die Vergleiche der `guid` und des Zeitstempels auszuschließen. Dazu reicht es die entsprechenden Schlüssel-Werte-Paare über die Schlüssel `"guid"` und `"letztesBearbeitungsDatum"` aus der Ergebnis-Hashtabelle zu entfernen (Z. 114-115).

Die lokale Variable `expectedJson` speichert das erwartete Ergebnis der eingegebenen Maß-

4. Schritt 1 - Formular in Grundstruktur erstellen

nahme (Z. 117-120). Die Methode `expect` und der „Matcher“ `equals` überprüfen beide Objekte auf Gleichheit (Z. 122).

Der Befehl `flutter test integration_test/app_test.dart` startet den Test. Die App öffnet sich und der Ausführung des Tests kann zugesehen werden Punkt am Enderfolg in dem Terminal die Ausgabe des Ergebnisses: `All tests passed!`

5. Schritt 2

Maßnahmen Master

Abgeschlossen

Zuletzt bearbeitet am	Maßnahmentitel	Förderklasse	Kategorie	Zielfläche	Zieleinheit	Hauptzielsetzung Land
2021-8-4 13:15	Maßnahme 1	aukm_ohne_vns	extens	al	ha	biodiv

In Bearbeitung

Zuletzt bearbeitet am	Maßnahmentitel	Förderklasse	Kategorie	Zielfläche	Zieleinheit	Hauptzielsetzung Land
2021-8-4 13:17	Maßnahme 2					

Abbildung 5.1.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

← Maßnahmen Detail

Maßnahmencharakteristika

Förderklasse

Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE Vertragsnaturschutz

Kategorie

Extensivierung

Zielsetzung

Zielfläche

AL

Zieleinheit

ha

Hauptzielsetzung Land

Biodiversität

Abbildung 5.2.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

In diesem Schritt sollen weitere Selektions-Karten für die Einzelauswahlfelder hinzugefügt werden. Es handelt sich um die Einzelauswahlfelder für Förderklasse, Kategorie, Zielfläche, Zieleinheit und Zielsetzung.

Darüber hinaus soll das Erstellen der Selektions-Karten in einer Methode abstrahiert werden. Das ermöglicht die Konfiguration der Selektions-Karten in der aufrufenden Eingabemaske, ohne dafür die Klasse `SelectionCard` ändern zu müssen.

99

5.0.1. Integrationstest erweitern

Noch vor der Implementierung der Änderungen soll zunächst der Integrationstest um die zusätzlichen Selektionen erweitert werden (Listing 5.1). Nach den letzten Eingaben und bevor der Button zum Speichern ausgelöst wird, erfolgt die Selektion der fünf Optionen (Z. 106-119).

```

106 await tabSelectionCard(foerderklasseChoices);
107 await tabOption(FoerderklasseChoice.aukm_ohne_vns, tabConfirm: true);
108
109 await tabSelectionCard(kategorieChoices);
110 await tabOption(KategorieChoice.extens, tabConfirm: true);
111
112 await tabSelectionCard(zielflaecheChoices);
113 await tabOption(ZielflaecheChoice.al, tabConfirm: true);
114
115 await tabSelectionCard(zieleinheitChoices);
116 await tabOption(ZieleinheitChoice.ha, tabConfirm: true);
117
118 await tabSelectionCard(hauptzielsetzungLandChoices);
119 await tabOption(ZielsetzungLandChoice.biodiv, tabConfirm: true);
120
121 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
122 await tester.tap(saveMassnahmeButton);
123 await tester.pumpAndSettle(durationAfterEachStep);

```

Listing 5.1.: Der Integrationstest klickt 5 weitere Karten, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/integration_test/app_test.dart](#)

Nach der Auswahl und der anschließenden Serialisierung sollen die entsprechenden Werte auch in der Json-Datei auftauchen. Die Json-Datei erhält ein neues Schlüssel-Werte-Paar mit dem Schlüssel `'massnahmenCharakteristika'` und einem Objekt für die fünf neuen Werte (Listing 5.2, Z. 135-141).

```

132 var expectedJson = {
133   'letzteBearbeitung': {'letzterStatus': 'fertig'},
134   'identifikatoren': {'massnahmenTitel': massnahmeTitle},
135   'massnahmenCharakteristika': {
136     'foerderklasse': 'aukm_ohne_vns',
137     'kategorie': 'extens',
138     'zielflaeche': 'al',
139     'zieleinheit': 'ha',
140     'hauptzielsetzungLand': 'biodiv'
141   },
142 };

```

Listing 5.2.: Der Integrationstest klickt 5 weitere Karten, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/integration_test/app_test.dart](#)

Der Integrationstest ist damit aktualisiert. Die Implementierung ist jedoch noch gar nicht erfolgt. Die Selektions-Karten können nicht geklickt werden, da sie in der Oberfläche noch nicht auftauchen. Die neuen Schlüssel-Werte-Paare können nicht in der Hash-Tabelle auftauchen, da sie dem entsprechenden Werte-Typ noch nicht hinzugefügt wurden. Der Integrationstest kann also unmöglich erfolgreich sein. Der Quellcode kann noch nicht einmal kompilieren, da die entsprechenden Symbole – wie zum Beispiel `FoerderklasseChoice` –

fehlen. Das hier angewendete Vorgehensmodell wird Test-Driven Development – deutsch Testgetriebene Entwicklung – genannt.

„Development is driven by tests. You test first, then code. Until all the tests run, you aren’t done. When all the tests run, and you can’t think of any more tests that would break, you are done adding functionality.“

— Kent Beck¹

Es folgt das Hinzufügen der fehlenden Symbole, damit der Quellcode wieder kompiliert werden kann. Anschließend erfolgt die Weiterentwicklung des Models, ViewModels und Views damit der Integrationstest erneut erfolgreich abschließt.

5.0.2. Hinzufügen der Auswahloptionen

Der Integrationstest selektiert unter anderem die Förderklasse mit der Abkürzung `aukm_ohne_vns`. Sie wird den Auswahloptionen hinzugefügt, wie in Listing 5.3 zu sehen ist. Die Liste aller hinzugefügten Auswahloptionen in diesem Schritt ist in Anhang ?? auf den Seiten ?? bis ?? zu finden.

```
11 static final aukum_ohne_vns = FoerderklasseChoice("aukm_ohne_vns",
12     "Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE
    ↳ Vertragsnaturschutz");
```

Listing 5.3.: Die Klasse FoerderklasseChoice, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

5.0.3. Aktualisierung des Models

Damit der Hash-Tabelle der Schlüssel `'massnahmenCharakteristika'` hinzugefügt wird, muss der entsprechende Eintrag im Werte-Typ `Massnahme` hinzugefügt werden. Die Getter-Methode `massnahmenCharakteristika`, die das Paket „*built_value*“ dazu veranlaßt, den Quellcode für die Eigenschaft zu generieren, wird unterhalb der Getter-Methode `identifikatoren` hinzugefügt (Listing 5.4, Z. 15).

```
13 Identifikatoren get identifikatoren;
14
15 MassnahmenCharakteristika get massnahmenCharakteristika;
```

Listing 5.4.: massnahmenCharakteristika wird Massnahme hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/data_model/massnahme.dart](#)

Bei dem Datentyp handelt es sich um einen weiteren Werte-Typ: `MassnahmenCharakteristika`, welcher in Listing 5.5 zu sehen ist. Die darin enthaltenen Getter-Methoden sind dagegen

¹3, S. 9.

lediglich gewöhnliche Zeichenketten, da sie die Abkürzungen der ausgewählten Optionen abspeichern. Da sie auch im Entwurfsmodus auch nicht gefüllt sein können, wird ihnen mit dem Suffix `?` erlaubt, auch Null-Werte anzunehmen (Z. 70-74).

```

67 abstract class MassnahmenCharakteristika
68     implements
69         Built<MassnahmenCharakteristika, MassnahmenCharakteristikaBuilder> {
70   String? get foerderklasse;
71   String? get kategorie;
72   String? get zielflaeche;
73   String? get zieleinheit;
74   String? get hauptzielsetzungLand;

```

Listing 5.5.: Der Werte-Typ `MassnahmenCharakteristika`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/data_model/massnahme.dart](#)

Der Werte-Typ wurde hinzugefügt. Der Befehl `flutter pub run build_runner build` generiert den Quellcode für die Serialisierung und die Builder-Methoden.

5.0.4. Aktualisierung der Übersichtstabelle

Der Übersichtsbildschirm bzw. die Übersichtstabelle können auf das Model ohne den Umweg über das ViewModel zugreifen. Der Tabellenkopf listet die Überschriften der hinzugefügten Werte auf (Listing 5.6, Z. 23-27).

```

22 _buildColumnHeader(const Text("Maßnahmentitel")),
23 _buildColumnHeader(const Text("Förderklasse")),
24 _buildColumnHeader(const Text("Kategorie")),
25 _buildColumnHeader(const Text("Zielfläche")),
26 _buildColumnHeader(const Text("Zieleinheit")),
27 _buildColumnHeader(const Text("Hauptzielsetzung Land")),

```

Listing 5.6.: Maßnahmencharakteristika werden dem Tabellenkopf hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/widgets/massnahmen_table.dart](#)

Für jede Zeile der Tabelle werden weitere selektierbare Zellen generiert (Listing 5.7, Z. 33-42). Im Unterschied zur Zelle des Maßnahmen-Titels können die Getter-Methoden der Maßnahmen-Charakteristika jedoch Null-Werte enthalten. Doch das `Text`-Widget akzeptiert keine Null-Werte als Argument. Deshalb wird der Operator `??` verwendet. Dabei handelt es sich um die „*If-null Expression*“. Sie überprüft den Ausdruck links vom Operator `??`. Ist er `null`, so wird der Wert rechts vom Operator verwendet. Ist der dagegen nicht `null`, so wird der Wert links vom Operator `??` genutzt.² Ist der Wert also nicht gefüllt, so wird in allen Fällen der leere String `""` als Argument übergeben.

²Vgl. 17, S. 165.

```

32 _buildSelectableCell(m, Text(m.identifikatoren.massnahmenTitel)),
33 _buildSelectableCell(
34   m, Text(m.massnahmenCharakteristika.foerderklasse ?? "")),
35 _buildSelectableCell(
36   m, Text(m.massnahmenCharakteristika.kategorie ?? "")),
37 _buildSelectableCell(
38   m, Text(m.massnahmenCharakteristika.zielflaeche ?? "")),
39 _buildSelectableCell(
40   m, Text(m.massnahmenCharakteristika.zieleinheit ?? "")),
41 _buildSelectableCell(m,
42   Text(m.massnahmenCharakteristika.hauptzielsetzungLand ?? "")),

```

Listing 5.7.: Maßnahmencharakteristika werden dem Tabellenkörper hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/widgets/massnahmen_table.dart](#)

5.0.5. Aktualisierung des ViewModels

Damit die Eingabefelder die neuen Werte eintragen können, muss das ViewModel oder die beobachtbaren Subjects bereitstellen (Listing 5.8, Z. 12-17). **Subjects und Observer in Schritt 1 erklären**

```

5 class MassnahmenFormViewModel {
6   final letzterStatus = BehaviorSubject<LetzterStatus?>.seeded(null);
7
8   final guid = BehaviorSubject<String?>.seeded(null);
9
10  final massnahmenTitel = BehaviorSubject<String>.seeded("");
11
12  final foerderklasse = BehaviorSubject<FoerderklasseChoice?>.seeded(null);
13  final kategorie = BehaviorSubject<KategorieChoice?>.seeded(null);
14  final zielflaeche = BehaviorSubject<ZielflaecheChoice?>.seeded(null);
15  final zieleinheit = BehaviorSubject<ZieleinheitChoice?>.seeded(null);
16  final hauptzielsetzungLand =
17    BehaviorSubject<ZielsetzungLandChoice?>.seeded(null);

```

Listing 5.8.: Maßnahmencharakteristika werden dem ViewModel hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

Die Konvertierung des Models in das ViewModel erfolgt wie gewohnt über das Herausuchen des korrekten Objektes aus der Menge der Auswahloptionen über die Abkürzung (Listing 5.9, Z. 29-36).

Wenn in jeder Zeile der Ausdruck `model.massnahmenCharakteristika` stehen würde, wäre die Leserlichkeit stark eingeschränkt. Das würde für weitere Zeilenumbrüche sorgen. Deshalb speichert die lokale Variable `mc` den Ausdruck zwischen und kann in den folgenden Zeilen verwendet werden (Z. 27). Damit die variable `mc` jedoch nur Gültigkeit für die folgenden Zeilen hat, begrenzen die öffnenden und schließenden geschweiften Klammern den Sichtbarkeitsbereich (Z. 26,37).

Bei der Konvertierung des Models in das ViewModel wurde bereits beim letzten Schritt die Methode `update` verwendet, um das Objekt des geschachtelten Wertetyps `Identifikatoren` anzupassen (Listing 5.10, Z. 44). So ist es auch für den geschachtelten Wertetyp `MassnahmenCharakteristika`

```

19 set model(Massnahme model) {
20   guid.value = model.guid;
21
22   letzterStatus.value = letzterStatusChoices
23     .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
24   massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
25
26   {
27     final mc = model.massnahmenCharakteristika;
28
29     foerderklasse.value =
30       foerderklasseChoices.fromAbbreviation(mc.foerderklasse);
31     kategorie.value = kategorieChoices.fromAbbreviation(mc.kategorie);
32
33     zielflaeche.value = zielflaecheChoices.fromAbbreviation(mc.zielflaeche);
34     zieleinheit.value = zieleinheitChoices.fromAbbreviation(mc.zieleinheit);
35     hauptzielsetzungLand.value =
36       hauptzielsetzungLandChoices.fromAbbreviation(mc.hauptzielsetzungLand);
37   }
38 }

```

Listing 5.9.: Konvertierung des Models in das ViewModel, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

der Fall. Der Unterschied: Es handelt sich um Auswahloptionen, weshalb nur die Abkürzungen abgespeichert werden (Z. 46-50), so wie es auch schon bei `letzterStatus` geschah (Z. 42).

```

40 Massnahme get model => Massnahme((b) => b
41   ..guid = guid.value
42   ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
43   ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
44   ..identifikatoren.update((b) => b..massnahmenTitel = massnahmenTitel.value)
45   ..massnahmenCharakteristika.update((b) => b
46     ..foerderklasse = foerderklasse.value?.abbreviation
47     ..kategorie = kategorie.value?.abbreviation
48     ..zielflaeche = zielflaeche.value?.abbreviation
49     ..zieleinheit = zieleinheit.value?.abbreviation
50     ..hauptzielsetzungLand = hauptzielsetzungLand.value?.abbreviation));

```

Listing 5.10.: Konvertierung des ViewModels in das Model, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

5.0.6. Aktualisierung der Eingabemaske

Nach der Anpassung des ViewModels kann schließlich die Eingabemaske erweitert werden.

Im letzten Schritt nahm die Selektionskarte für den letzten Status 11 Zeilen ein **R**. Das wäre für jede weitere Karte nun auch der Fall. Damit die Übersichtlichkeit darunter nicht leidet, soll nun zunächst eine Methode erstellt werden, welche die Erstellung der Selektionskarten abstrahiert und damit den Aufruf auf 3 Zeilen reduziert. Dies erlaubt auch die

Konfiguration der Selektionskarten außerhalb der Klasse `SelektionCard`. In den folgenden Schritten soll diese Konfigurationsmöglichkeit genutzt werden, um weitere Funktionalitäten hinzuzufügen, ohne die Klasse selbst zu manipulieren. Die Methode `buildSelectionCard` bekommt dazu nur die Argumente für die Liste aller Auswahloptionen `allChoices` (Listing 5.13, Z. 49) und das Subject `selectionViewModel` (Z. 50) übergeben. Nun übernimmt die Methode die Übergabe der Argumente an den Konstruktor der `SelektionCard`. Dazu verwendet die `SelektionCard` wie zuvor den Namen der Menge der Auswahloptionen als Titel (Z. 52). Außerdem wird dieselbe Menge unverändert an die `SelektionCard` weitergegeben (Z. 53).

```
48 Widget buildSelectionCard<ChoiceType extends Choice>(  
49     {required Choices<ChoiceType> allChoices,  
50     required BehaviorSubject<ChoiceType?> selectionViewModel}) {  
51     return SelektionCard<ChoiceType>(  
52         title: allChoices.name,  
53         allChoices: allChoices,  
54         initialValue: {  
55             if (selectionViewModel.value != null) selectionViewModel.value!  
56         },  
57         onSelect: (selectedChoice) => selectionViewModel.value = selectedChoice,  
58         onDeselect: (selectedChoice) => selectionViewModel.value = null,  
59     );  
60 }
```

Listing 5.11.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Der Grund, warum die Klasse `SelektionCard` den Titel aus der Menge der Auswahloption nicht selbständig extrahiert ist, dass die Klasse auf diese Weise auch für mehrere Anwendungsgebiete genutzt werden kann. Es muss nicht immer der Fall sein, dass der Titel auf diese Art und Weise ausgelesen werden kann. Somit erlaubt die Methode `buildSelectionCard` nun den Aufruf trotzdem zu vereinfachen und die Anwendbarkeit der Klasse `SelektionCard` durch dessen direkte Veränderung nicht einzuschränken.

Das betrifft auch das ViewModel. Durch die Methode `buildSelectionCard` muss lediglich das `BehaviorSubject` übergeben werden. Die Methode kümmert sich bei Initialisierung der Selektionskarte um das Auslesen des aktuellen Wertes (Z. 54-56) und die Aktualisierung dessen über die Methoden `onSelect` (Z. 57) `onDeselect` (Z. 58). Damit ist die Erstellung der Selektionskarte für den letzten Status mit 3 Zeilen (Listing 5.12) nun deutlich kürzer als die ursprüngliche Variante mit 11 Zeilen (siehe Seite 82).

```
77 buildSelectionCard(  
78     allChoices: letzterStatusChoices,  
79     selectionViewModel: vm.letzterStatus),
```

Listing 5.12.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Unterhalb des Eingabefeldes für den Maßnahmen-Titel können nun die weiteren Selekti-

onskarten ergänzt werden, die jeweils ebenfalls bloß 3 Zeilen einnehmen und damit eine hohe Übersichtlichkeit gewährleisten (Listing 5.13, Z. 82-98).

```

80 buildSectionHeadline("Identifikatoren"),
81 createMassnahmenTitelTextFormField(),
82 buildSectionHeadline("Maßnahmencharakteristika"),
83 buildSelectionCard(
84     allChoices: foerderklasseChoices,
85     selectionViewModel: vm.foerderklasse),
86 buildSelectionCard(
87     allChoices: kategorieChoices,
88     selectionViewModel: vm.kategorie),
89 buildSubSectionHeadline("Zielsetzung"),
90 buildSelectionCard(
91     allChoices: zielflaecheChoices,
92     selectionViewModel: vm.zielflaeche),
93 buildSelectionCard(
94     allChoices: zieleinheitChoices,
95     selectionViewModel: vm.zieleinheit),
96 buildSelectionCard<ZielsetzungLandChoice>(
97     allChoices: hauptzielsetzungLandChoices,
98     selectionViewModel: vm.hauptzielsetzungLand),

```

Listing 5.13.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Auffällig hierbei sind Überschriften (Z. 80, 82) und eine Zwischenüberschrift (Z. 89) über den Selektionskarten. Sie sorgen für sichtbare Gruppierungen in der Oberfläche.

Die Hilfsmethode `buildSectionHeadline` und `buildSubSectionHeadline` bauen die Überschriften (Listing 5.14, Z. 131-134) bzw. Zwischenüberschriften (Z. 136-139) mit unterschiedlichen Abständen zur Außenkante (Z. 132, 137) und unterschiedlicher Schriftgröße (Z. 133, 138). Der benannte Konstruktor `fromLTRB` der Klasse `EdgeInsets` erlaubt die Abstände zur Außenkante im Uhrzeigersinn für jede Seite festzulegen. Die Abkürzung `LTRB` steht dabei für `left, top, right, bottom` – deutsch links, oben, rechts, unten.

```

131 Widget buildSectionHeadline(String text) => Padding(
132     padding: const EdgeInsets.fromLTRB(0, 24, 0, 8),
133     child: Text(text, style: const TextStyle(fontSize: 22)),
134 );
135
136 Widget buildSubSectionHeadline(String text) => Padding(
137     padding: const EdgeInsets.fromLTRB(4, 12, 0, 4),
138     child: Text(text, style: const TextStyle(fontSize: 14)),
139 );

```

Listing 5.14.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Damit ist die Implementierung für Schritt 2 beendet.

Der Integrationstest kann nun verifizieren, dass die Eingaben erfolgen und in der Json-Datei auftauchen werden.

6. Schritt 3

← Maßnahmen Detail

Identifikatoren

Maßnahmentitel
Maßnahme 1

Dieser Maßnahmentitel ist bereits vergeben

Maßnahmencharakteristika

Förderklasse

Feld Förderklasse enthält keinen Wert!

Fehler im Formular trotz Status "abgeschlossen" Entwurf speichern?

Abbildung 6.1.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

In diesem Schritt soll die grundlegende Validierungsfunktion hinzugefügt werden. Maßnahmen, die als abgeschlossen markiert sind, dürfen keine leeren Eingabefelder enthalten und der Maßnahmentitel darf nicht doppelt belegt sein. `Flutter` stellt das Widget `Form` für die Validierung von Eingabefeldern bereit.

6.1. Einfügen des Form-Widgets

Das Widget `Form` ist ein Container, welcher die Validierung für alle Kinderelemente des Typs `FormField` ausführt. Damit es alle Eingabefelder im Formular umgibt, wird es oberhalb des `Stack` eingefügt (Listing 6.1, Z. 161). Das `Form`-Widget muss über einen `key` registriert werden (Z. 162), damit auf die Validierungsfunktionen zurückgegriffen werden kann.

Die Erstellung des `formKey` findet zu Beginn der `build`-Methode des Eingabeformulars statt (Listing 6.2, Z. 20). Der `GlobalKey` identifiziert ein Element, welches durch ein Widget gebaut wurde, über die gesamte Applikation hinweg. Es erlaubt darüber hinaus auf das `State`-Objekt zuzugreifen, welches mit dem `StatefulWidget` verknüpft ist. Ohne Angabe eines Typparameters kann nur Zugriff auf Funktionen des Typs `State` gewährt werden. Doch die gewünschte Methode `validate` ist nur Teil des Typs `FormState`. Damit das Element,

```

161 child: Form(
162   key: formKey,
163   child: Stack(
164     children: [
165       SingleChildScrollView(
166         child: Center(

```

Listing 6.1.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

welches über den `GlobalKey` registriert wurde, auch den `FormState` liefert, kann der entsprechende Typparameter `<FormState>` bei der Erstellung des `GlobalKey` übergeben werden.

```

17 Widget build(BuildContext context) {
18   final vm = AppState.of(context).viewModel;
19   final model = AppState.of(context).model;
20   final formKey = GlobalKey<FormState>();

```

Listing 6.2.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

6.2. Validierung des Maßnahmentitels

Das Eingabefeld für den Maßnahmen-Titel ist ein `TextFormField` (Listing 6.3, Z. 88). Es erbt vom Typ `FormField` und wird daher mit dem Vatterelement `Form` verknüpft. Es beinhaltet bereits einen Parameter für die Validierungsfunktion namens `validator` (Z. 93). Die übergebene Funktion erhält im ersten Parameter den für das Textfeld eingetragenen Wert. Die Funktion soll `null` zurückgeben, wenn keine Fehler in der Validierung geschehen sind. In jedem anderen Fall soll der Text zurückgegeben werden, der als Fehlermeldung angezeigt werden soll.

Sollte der Parameter `null` sein oder aber ein leerer String (Z. 94), so wird die entsprechende Fehlermeldung `'Bitte Text eingeben'` angezeigt (Z. 96). Damit der Benutzer direkt zu dem fehlerhaften Eingabefeld geführt wird, kann ein Objekt der Klasse `FocusNode` verwendet werden. Er wird vor der Konstruktion der Karte erstellt (Z. 84) und dem Parameter `focusNode` des `TextFormField` übergeben (Z. 89). Sollte ein Fehler bei der Validierung gefunden werden, kann mit der Methode `requestFocus` angeordnet werden, den Cursor in das betreffende Feld zu setzen (Z. 95). Das sorgt auch dafür, dass das Eingabefeld in den sichtbaren Bereich gerückt wird.

Sollte das Textfeld nicht leer sein, so soll noch überprüft werden, ob der Maßnahmen-Titel bereits vergeben ist. Über das Model kann die Liste der Maßnahmen angefordert werden (Z. 99). Die Funktion `any` akzeptiert als Argument eine Funktion, die für alle Elemente der Liste ausgeführt wird (Z. 99-102). Wenn die Rückgabe der Funktion auch nur in einem

```

83 Widget createMassnahmenTitelTextFormField() {
84   final focusNode = FocusNode();
85   return Card(
86     child: Padding(
87       padding: const EdgeInsets.all(16.0),
88       child: TextFormField(
89         focusNode: focusNode,
90         initialValue: vm.massnahmenTitel.value,
91         decoration: const InputDecoration(
92           hintText: 'Maßnahmentitel', labelText: 'Maßnahmentitel'),
93         validator: (title) {
94           if (title == null || title.isEmpty) {
95             focusNode.requestFocus();
96             return 'Bitte Text eingeben';
97           }
98           var massnahmeTitleDoesAlreadyExists =
99             model.storage.value.massnahmen.any((m) =>
100               m.guid != vm.guid.value &&
101               m.identifikatoren.massnahmenTitel ==
102                 vm.massnahmenTitel.value);
103
104           if (massnahmeTitleDoesAlreadyExists) {
105             focusNode.requestFocus();
106             return 'Dieser Maßnahmentitel ist bereits vergeben';
107           }
108           return null;
109         },
110         onChanged: (value) {
111           vm.massnahmenTitel.value = value;
112         },
113       ),
114     ),
115   );
116 }

```

Listing 6.3.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Fall `true` ist, so evaluiert auch `any` mit `true`. Andernfalls ist die Rückgabe `false`. Die anonyme Funktion schließt zunächst den Vergleich mit derselben Maßnahme aus, welche sich gerade in Bearbeitung befindet. Der Vergleich der `guid` ist dafür ausreichend. Sollte es eine andere Maßnahme geben, welche den gleichen Titel hat (Z. 101-102), so wird Die lokale Variable `massnahmeTitleDoesAlreadyExists` auf `true` gesetzt. Der Benutzer bekommt die entsprechende Fehlermeldung `'Dieser Maßnahmentitel ist bereits vergeben'` zu lesen (Z. 106). Wenn keine der beiden Fallunterscheidungen das `return`-Statement (Z. 96, 106) auslöst, so erfolgt schließlich die Rückgabe von `null`. In dem Kontext der `validator`-Funktion bedeutet die Rückgabe von `null` (Z. 108), dass die Validierung erfolgreich war.

Das `Form`-Widget validiert lediglich Kindelemente vom Typ `FormField`. Dementsprechend wird das Widget `SelectionCard` nicht in die Validierung miteinbezogen. Es erbt nicht von `FormField`. Es wäre möglich, eine weitere Klasse zu erstellen, die von `FormField` erbt und alle Parameter für die Erstellung einer Selektions-Karte wiederverwendet. Doch das würde bedeuten, dass für alle folgenden Schritte jeder weitere Parameter in beiden Konstruktoren der Klassen gepflegt werden müsste. Um der Arbeit leichter folgen zu können, wurde

sich für einen anderen, simpleren Weg entschieden: Die Selektionskarte kann ebenso von einem `FormField` umgeben werden (Listing 6.4, Z. 121-148), welches die Selektionskarte in der `builder`-Funktion erstellt und an den Parametern nichts ändert, außer einen weiteren hinzuzufügen: der Text für die Fehlermeldung (Z. 147). Der erste Parameter der `builder`-Funktion ist das `State`-Objekt des `FormField`. Es enthält die Getter-Methode `errorText`, die bei gegebenenfalls fehlgeschlagener Validierung die zurückgegebene Fehlermeldung enthält.

```

118 Widget buildSelectionCard<ChoiceType extends Choice>(
119   {required Choices<ChoiceType> allChoices,
120     required BehaviorSubject<ChoiceType?> selectionViewModel}) {
121   return FormField(
122     validator: (_) {
123       Iterable<Choice> choices = {
124         if (selectionViewModel.value != null) selectionViewModel.value!
125       };
126
127       if (choices.isEmpty) {
128         return "Feld ${allChoices.name} enthält keinen Wert!";
129       }
130
131       return null;
132     },
133     builder: (field) => SelectionCard<ChoiceType>(
134       title: allChoices.name,
135       allChoices: allChoices,
136       initialValue: {
137         if (selectionViewModel.value != null)
138           selectionViewModel.value!
139       },
140       onSelect: (selectedChoice) =>
141         selectionViewModel.value = selectedChoice,
142       onDeselect: (selectedChoice) => selectionViewModel.value = null,
143       errorText: field.errorText,
144     ));
145 }

```

Listing 6.4.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die anonyme Funktion, die als Argument dem Parameter `validator` übergeben wird (Z. 122-132), erstellt eine temporäre Menge, die den Wert des `selectionViewModel` enthält, wenn dieser nicht `null` ist, andernfalls ist sie eine leere Menge (Z. 123-125). Die `validator`-Funktion gibt eine Fehlermeldung zurück, sollte die Menge leer sein (Z. 127-129). Ist die Menge dagegen gefüllt, so gibt sie `null` zurück, um mitzuteilen, dass die Validierung erfolgreich war (Z. 131).

Der `errorText` wird im Konstruktor der Klasse `SelectionCard` übergeben (Listing 6.5, Z. 29). Da er `null` sein darf, ist er mit dem Suffix `?` als Typ mit Null-Zulässigkeit gekennzeichnet (Z. 21).

Durch Einfügen einer `Column` zwischen der `Card` (Listing 6.6, Z. 53) und dem `ListTile` (Z. 57) kann die visuelle Repräsentation der Selektionskarte in der Höhe erweitert werden.

```

19 final OnSelect<ChoiceType> onSelect;
20 final OnDeselect<ChoiceType> onDeselect;
21 final String? errorText;
22
23 SelectionCard(
24   {required this.title,
25     required Iterable<ChoiceType> initialValue,
26     required this.allChoices,
27     required this.onSelect,
28     required this.onDeselect,
29     this.errorText,
30     Key? key})

```

Listing 6.5.: errorText wird der SelectionCard hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/widgets/selection_card.dart](#)

Sollte der errorText gesetzt sein (Z. 65), so erscheint unter dem Titel und dem Untertitel eine entsprechende Fehlermeldung (Z. 66-71).

```

53 return Card(
54   child: Column(
55     crossAxisAlignment: CrossAxisAlignment.start,
56     children: [
57       ListTile(
58         focusNode: focusNode,
59         title: Text(title),
60         subtitle: Text(
61           selectedChoices.map((c) => c.description).join(", "),
62           trailing: const Icon(Icons.edit),
63           onTap: navigateToSelectionScreen,
64         ),
65         if (errorText != null)
66           Padding(
67             padding: const EdgeInsets.all(8.0),
68             child: Text(errorText!,
69               style:
70                 const TextStyle(fontSize: 12.0, color: Colors.red)),
71           )
72       ],
73     ),
74 );

```

Listing 6.6.: errorText wird ausgegeben, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/widgets/selection_card.dart](#)

Oberhalb des vorhandenen `FloatingActionButton` wird nun ein weiterer eingefügt, der zum Speichern des Entwurfs mit der Funktion `saveDraftAndGoBackToOverviewScreen` genutzt werden soll (Listing 6.7, Z. 207-216). Der ursprüngliche `FloatingActionButton` speichert nun ausschließlich dann, wenn die Maßnahme als „in Bearbeitung“ markiert ist oder alle Eingabefelder valide sind. Dazu nutzt er die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` (Z. 222). Ist das der Fall, so folgt die Speicherung der Maßnahme mithilfe der bereits implementierten Funktion `saveRecord` (Z. 223). Diese funktioniert wie in den letzten Schritten, nur dass sie keinen Rückgabewert mehr hat (siehe Listing ?? in Anhang ?? auf Seite ??). Anschließend wird der Navigator erneut aufgefordert, zum Übersichtsbildschirm zurückzukehren (Z. 224). Sollte es allerdings zur Ausführung des `else`-Blocks führen (Z. 225-227), da die Maßnahme doch als „abgeschlossen“ markiert und nicht alle Eingabefelder valide

waren, so erhält der Benutzer eine Fehlermeldung. Die neu implementierte Hilfsfunktion `showValidationError` wird dafür verwendet (Z. 226).

```

206 children: [
207   FloatingActionButton(
208     mini: true,
209     heroTag: 'save_draft_floating_action_button',
210     child: const Icon(Icons.paste, color: Colors.white),
211     backgroundColor: Colors.orange,
212     onPressed: saveDraftAndGoBackToOverviewScreen,
213   ),
214   const SizedBox(
215     height: 10,
216   ),
217   FloatingActionButton(
218     tooltip: saveMassnahmeTooltip,
219     heroTag: 'save_floating_action_button',
220     child: const Icon(Icons.check, color: Colors.white),
221     onPressed: () {
222       if (inputsAreValidOrNotMarkedFinal()) {
223         saveRecord();
224         Navigator.of(context).pop();
225       } else {
226         showValidationError();
227       }
228     },
229   )
230 ],

```

Listing 6.7.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Auch der `WillPopScope` erhält die gleiche Fehlerbehandlung (Listing 6.8). Hier wird ebenfalls überprüft, ob die Maßnahme als „abgeschlossen“ markiert wurde und ob alle Eingabefelder valide sind (Z. 153). Falls ja, wird die Maßnahme direkt gespeichert Und ein Objekt des asynchronen Types `Future` zurückgegeben, welches direkt zu `true` evaluiert (Z. 155). Das führt dazu, dass dem Zurücknavigieren zum Übersichtsbildschirm zugestimmt wird. Sollte allerdings der `else`-Block ausgeführt werden, so erscheint erneut die entsprechende Fehlermeldung (Z. 157) und dieses Mal evaluiert das `Future`-Objekt zu `false`, um die Navigation zu unterbinden 158.

```

151 body: WillPopScope(
152   onWillPop: () {
153     if (inputsAreValidOrNotMarkedFinal()) {
154       saveRecord();
155       return Future.value(true);
156     } else {
157       showValidationError();
158       return Future.value(false);
159     }
160   },

```

Listing 6.8.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Funktion `saveDraftAndGoBackToOverviewScreen` funktioniert ähnlich wie die nun ausge-

tauschte Funktion `saveRecord`. Sie zeigt dem Benutzer an, dass die Maßnahme im Entwurfsmodus gespeichert wird (Z. 23-26), speichert sie im Model ab (Z. 31), und navigiert zur letzten Route zurück (Z. 32), welcher der Übersichtsbildschirm ist. Einer der beiden Unterschiede ist, dass die Maßnahme zuvor umgebaut wird. Unerheblich dessen, welchen letzten Status sie aktuell besitzt, erhält sie den letzten Status `"in Bearbeitung"` (Z. 28-29). Der zweite der beiden Unterschiede ist, dass die Funktion nun keinen Rückgabewert hat, während `saveRecord` einen Wert vom Typ `Future<bool>` zurückgeben musste. Der Grund dafür ist, dass die Funktion nur noch über den Aktionsbutton zum Speichern der Maßnahme im Entwurfsmodus ausgelöst wird. Der `FloatingActionButton` setzt keinen Rückgabewert der ausgelösten Funktion voraus.

```

22 void saveDraftAndGoBackToOverviewScreen() {
23   ScaffoldMessenger.of(context)
24     ..hideCurrentSnackBar()
25     ..showSnackBar(
26       const SnackBar(content: Text('Entwurf wird gespeichert ...')));
27
28   var draft = vm.model.rebuild((b) =>
29     b.letzteBearbeitung.letzterStatus = LetzterStatus.bearb.abbreviation);
30
31   model.putMassnahmeIfAbsent(draft);
32   Navigator.of(context).pop();
33 }

```

Listing 6.9.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` überprüft zunächst, ob der letzte Status ein anderer ist als „abgeschlossen“ (Listing 6.10, Z. 71). Da in diesem Fall keine weiteren Überprüfungen notwendig sind, gibt die Funktion direkt `true` zurück (Z. 73). Andernfalls validiert das Formular die Eingabefelder (Z. 76). Dazu muss das Element vom Typ `Form` in den Vaterelementen gefunden werden. Genauer gesagt wird dessen `State`-Objekt benötigt. Der Zugriff auf das Element ist einfach, da es über einen `GlobalKey` registriert wurde. Über `formKey.currentState` kann das `State`-Objekt des Elements abgerufen werden (Z. 76). Die Funktion `validate()` führt dann alle Funktionen aus, die jeweils als Argument dem Parameter `validator` aller Kindelemente des Typs `FormField` übergeben wurden. Sollten alle `validator`-Funktionen `null` zurückgegeben haben – was bedeutet, dass keine Fehler bei der Validierung geschehen sind – so erfolgt die Rückgabe von `true` (Z. 77). Anderenfalls bleibt nur die Rückgabe von `false` übrig (Z. 80).

Sollte es zu einem Fehler kommen, so zeigt die Hilfsfunktion `showValidationError` dem Benutzer die entsprechende Fehlermeldung an (Listing 6.11). Sie bietet ihm darüber hinaus an, über einen Button die Maßnahme direkt als Entwurf zu speichern. Das ist möglich, da die `SnackBar` (Z. 45) nicht nur die Anzeige von gewöhnlichem Text erlaubt, sondern von jedem beliebigen Widget. Zunächst kommt dazu das Widget `Row` zum Einsatz (Z. 46). Ähnlich wie das Widget `Column` erlaubt es Kindelemente in einer Reihe aufzulisten. Im Gegensatz zur `Column` allerdings nun horizontal statt vertikal. Als letztes Element der `Row`

```

71 bool inputsAreValidOrNotMarkedFinal() {
72   if (vm.letzterStatus.value != LetzterStatus.fertig) {
73     return true;
74   }
75
76   if (formKey.currentState!.validate()) {
77     return true;
78   }
79
80   return false;
81 }

```

Listing 6.10.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing,
 Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

wird der `ElevatedButton` verwendet. Genauso wie bereits der `FloatingActionButton` zum Speichern der Maßnahme im Entwurfsmodus verwendet nun auch dieser `ElevatedButton` die Funktion `saveDraftAndGoBackToOverviewScreen` (Z. 52).

```

44 void showValidationError() {
45   ScaffoldMessenger.of(context).showSnackBar(SnackBar(
46     content: Row(
47       children: [
48         Text(
49           'Fehler im Formular trotz Status "${letzterStatus.fertig.description}"',
50           const SizedBox(width: 4),
51         ElevatedButton(
52           onPressed: saveDraftAndGoBackToOverviewScreen,
53           child: Padding(
54             padding: const EdgeInsets.fromLTRB(4, 4, 8, 4),
55             child: Row(
56               children: const [
57                 Icon(Icons.paste, color: Colors.white),
58                 SizedBox(width: 4),
59                 Text(
60                   "Entwurf speichern?",
61                   style: TextStyle(fontSize: 18.0, color: Colors.white),
62                 ),
63               ],
64             ),
65           ),
66         ],
67       )),
68   ));
69 }

```

Listing 6.11.: Die Maßnahmencharakteristika Selektionskarten werden ergänzt, Quelle: Eigenes Listing,
 Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

7. Schritt 4

← Zielfläche

☐ HFF

☐ bitte um Unterstützung

× Landschaftselement/Biotop o.Ä.

× Wald/Forst

Abbildung 7.1.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

← Maßnahmen Detail

Förderklasse
Ökolandbau

Kategorie
Extensivierung

Zielsetzung

Zielfläche
Wald/Forst

Wenigstens ein Wert im Feld Zielfläche enthält ist fehlerhaft!

Abbildung 7.2.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

← Zielfläche

☐ HFF

☒ Wald/Forst

☐ bitte um Unterstützung

× Landschaftselement/Biotop o.Ä.

Abbildung 7.3.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

Im Folgenden werden der Validierung die Bedingungen hinzugefügt, welche die Auswahloptionen untereinander haben.

7.1. Hinzufügen der Bedingungen zu den Auswahloptionen

Es gibt einfache Bedingungen wie beispielsweise die der Zielfläche „AL“. Dessen Auswahl kann nur dann erfolgen, wenn nicht die Kategorie „Anbau Zwischenfrucht/Untersaat“ ausgewählt ist (Listing 7.1).

```
79 static final al = ZielflaecheChoice("al", "AL",
80   condition: (choices) => !choices.contains(KategorieChoice.zf_us));
```

Listing 7.1.: XXXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

Doch es tauchen auch komplexe Bedingungen auf, wie etwa die Abhängigkeit der Zielfläche „Wald/Forst“ (Listing 7.2). Um sie auszuwählen, muss die Förderklasse eine von 3 Werten beinhalten: „Erschwerenausgleich“ (Z. 97), „Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz“ (Z. 98) oder „Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten , aber OHNE Vertragsnaturschutz“ (Z. 99).

Gleichzeitig darf für die „Kategorie“ weder „Anbau Zwischenfrucht/Untersaat“ (Z. 100) noch „Förderung bestimmter Rassen / Sorten / Kulturen“ (Z. 101) gewählt sein.

Äußerst wichtig ist hier die Auswahl der richtigen logischen Operatoren. Innerhalb des gleichen Typs – wie etwa der „Förderklasse“ – muss das logische Oder `||` verwendet werden (Z. 97, 98, 100). Das logische Und würde hier keinen Sinn ergeben, da es unmöglich ist, in einem Einfachauswahlfeld gleichzeitig zwei Optionen ausgewählt zu haben. Um Bedingungen unterschiedlichen Typs miteinander zu verknüpfen, ist dagegen das logische und `&&` zu benutzen (Z. 99), denn die Bedingungen der „Förderklasse“ und der „Kategorie“ müssen gleichzeitig erfüllt sein. Hier ist wiederum das Nutzen des logischen Oders nicht angemessen, denn es wäre nicht ausreichend, wenn nur die Bedingungen eines der beiden Typen erfüllt wäre. Wäre also beispielsweise für die „Förderklasse“ die Option „Erschwerenausgleich“ gewählt, so wäre es völlig unerheblich, welche Auswahl für die „Kategorie“ selektiert wurde. Die Bedingung wäre trotzdem erfüllt, auch wenn für die „Kategorie“ die nicht erlaubte Option „Anbau Zwischenfrucht/Untersaat“ gewählt ist.

```
95 static final wald = ZielflaecheChoice("wald", "Wald/Forst",
96   condition: (choices) =>
97     (choices.contains(FoerderklasseChoice.ea) ||
98     choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
99     choices.contains(FoerderklasseChoice.aukm_ohne_vns)) &&
100    (!choices.contains(KategorieChoice.zf_us) ||
101    !choices.contains(KategorieChoice.bes_kult_rass)));
```

Listing 7.2.: XXXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

Für die Liste aller hinzugefügten Bedingungen siehe Anhang ?? auf den Seiten ?? bis ??.

Bei der Bedingungen handelt sich um eine Funktion, die einen Wahrheitswert `bool` zurück-

gibt und als Parameter die Menge aller bisher ausgewählten Auswahloptionen `Set<Choice>` übergeben bekommt. Die Signatur dieser Funktion wird als Typdefinition mit dem Namen `Condition` deklariert (Listing 7.3, Z. 3). Über diese Typdefinition kann sie als Instanzvariable in der Klasse `Choice` deklariert werden (Z. 8). Der Konstruktor erhält einen weiteren Parameter für die Bedingung (Z. 12).

Er ist optional, da es Auswahloption gibt, die keine Bedingung haben. Deshalb wird mit der Notation `Condition?` erreicht, dass die Bedingung auch ausgelassen werden kann und in diesem Fall `null` ist. Sollte das der Fall sein, so soll eine Standardfunktion verwendet werden. Diese Standardfunktion ist `_conditionIsAlwaysMet` (Z. 15). Unerheblich davon welche Auswahloptionen in Vergangenheit gewählt wurden, gibt diese Funktion immer `true` zurück. Denn eine Auswahloption, die keine Bedingung hat, ist immer auswählbar. Sollte die übergebene Bedingungen ausgelassen worden und damit `null` sein, so wählt die „If-null Expression“ den Ausdruck rechts von dem `??` und damit die Standardfunktion `_conditionIsAlwaysMet` aus, welche der Instanzvariablen `condition` zugewiesen wird (Z. 13). Ansonsten speichert der Konstruktor die übergebene Funktion. Aus diesem Grund ist es nicht möglich, dass die `condition` in der Instanzvariablen nur sein kann, weshalb sie ohne den Suffix `?` als variable ohne zu null Zulässigkeit deklariert werden kann. Da der Ausdruck rechts von dem `??` nicht `null` sein kann, so kann auch der gesamte Ausdruck der vorliegenden „If-null Expression“ nicht `null` sein. Damit ist es möglich, die Instanzvariable `condition` ohne den Suffix `?` als Variable ohne Null-Zulässigkeit zu deklarieren. Die Instanzmethode `conditionMatches` ruft die übergebender Funktion für die Bedingung über die Methode `call` auf (Z. 10). Das erlaubt den Ausdruck durch vereinfacht darzustellen. Der Ausdruck `wald.condition(priorChoices)` kann dadurch durch die explizitere Schreibweise `wald.conditionMatches(priorChoices)` ersetzt werden.

```

3  typedef Condition = bool Function(Set<Choice> choices);
4
5  class Choice {
6    final String description;
7    final String abbreviation;
8    final Condition condition;
9
10   bool conditionMatches(Set<Choice> choices) => condition.call(choices);
11
12   const Choice(this.abbreviation, this.description, {Condition? condition})
13     : condition = condition ?? _conditionIsAlwaysMet;
14
15   static bool _conditionIsAlwaysMet(Set<Choice> choices) => true;
16 }

```

Listing 7.3.: XXXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/base/choice.dart](#)

7.2. Hinzufügen der Momentaufnahme aller ausgewählten Optionen im gesamten Formular

Die Menge der bisherigen Ausfülloptionen setzt sich aus den aktuellen Inhalten der Auswahlfelder zusammen. Sie ist also die Momentaufnahme aller Werte, die jeweils über die Getter-Methode `value` von allen `BehaviorSubject`-Objekten im ViewModel abgerufen werden kann. Doch genau diese Momentaufnahme muss immer dann neu erstellt werden, wenn sich auch nur ein Auswahlfeld ändert. Genau darum kümmert sich das `BehaviorSubject priorChoices` im ViewModel (Listing 7.4).

```

20 BehaviorSubject<Set<Choice>> priorChoices =
21     BehaviorSubject<Set<Choice>>.seeded({});
22
23 MassnahmenFormViewModel() {
24     Stream<Set<Choice>> choicesStream = Rx.combineLatest([
25         foerderklasse,
26         kategorie,
27         zielflaeche,
28         zieleinheit,
29         hauptzielsetzungLand,
30     ], (_) {
31         return {
32             if (foerderklasse.value != null) foerderklasse.value!,
33             if (kategorie.value != null) kategorie.value!,
34             if (zielflaeche.value != null) zielflaeche.value!,
35             if (zieleinheit.value != null) zieleinheit.value!,
36             if (hauptzielsetzungLand.value != null) hauptzielsetzungLand.value!,
37         };
38     });
39
40     choicesStream.listen((event) => priorChoices.add(event));
41 }

```

Listing 7.4.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

Es wird mit dem Typparameter `Set<Choice>` deklariert (Z. 20) und mit einer Momentaufnahme initialisiert: einer leeren Menge) (Z. 21). Im Konstruktor des ViewModels wird dann auf Änderung aller `BehaviorSubject`-Objekte im ViewModel gehorcht. Dies wird durch die Funktion `combineLatest` des Pakets `rx.dart` ermöglicht 24. Sie erlaubt die Übergabe einer Kollektion von Streams. In diesem Fall alle `BehaviorSubject`-Objekte des ViewModels (Z. 25-29). Wenn auch nur einer dieser Streams ein neues Ereignis sendet, so emittiert auch der kombinierte Stream ein neues Ereignis. Dem zweiten Parameter der Funktion `combineLatest` kann als Argument eine Funktion übergeben werden, die das zu emittierende Ereignis konstruiert (Z. 30-37). Der erste Parameter dieser Funktion enthält alle letzten Ereignisse der übergebenen Streams. Doch der vorliegende Aufruf hat keine Verwendung für den Parameter. Statt eines Variablennamens wird hier ein Unterstrich `_` verwendet (Z. 30).

In Sprachen wie etwa „JavaScript“ und „Python“ ist dies gängige Praxis für die Benennung von Parametern, die nicht genutzt werden. In Kotlin und Dart wurde diese Praxis

zur Konvention gemacht^{1,2}. Die anonyme Funktion gibt eine Menge zurück, in welcher alle Werte der `BehaviorSubject`-Objekte integriert werden (Z. 31-37). Das „*Collection if*“ Statement schließt dabei jeweils den Wert `null` aus (Z. 32-36). Somit taucht niemals der Wert `null` in der Menge auf und damit kann die Menge mit dem Typparameter `Choice` ohne Null-Zulässigkeit deklariert werden. Sollte ein Auswahlfeld nicht gewählt und damit der Wert des `BehaviorSubject` `null` sein, so taucht diese Option einfach nicht in der Menge auf. Sind alle Auswahlfelder nicht belegt und damit `null`, so ist die Menge leer. Doch der kombinierte Stream `choicesStream` liefert immer nur die neuen Ereignisse und speichert nicht den zuletzt übermittelten Wert. Deshalb wird das `BehaviorSubject` `priorChoices` verwendet. Die Methode `listen` horcht auf Änderungen des `choicesStream`-Objekts und fügt das übertragene Ereignis immer `priorChoices` hinzu. Damit existiert immer ein Wert für die Momentaufnahme der aktuell ausgewählten Auswahloptionen. Sie ist ursprünglich die leere Menge `{}` und nachfolgend immer das zuletzt übermittelte Ereignis des `choicesStream`.

7.3. Reagieren der Selektionskarte auf die ausgewählten Optionen

Dadurch, dass `priorChoices` nun im ViewModel verfügbar ist, kann es im Eingabeformular bei der Konstruktion der `SelectionCard` als Argument übergeben werden (Listing 7.5, Z. 143).

```

140 builder: (field) => SelectionCard<ChoiceType>(
141     title: allChoices.name,
142     allChoices: allChoices,
143     priorChoices: vm.priorChoices,

```

Listing 7.5.: Die Ausgabe der Formularfelder, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Klasse `SelectionCard` deklariert die `priorChoices` als Instanzvariable (Listing 7.6, Z. 19) und initialisiert sie direkt bei der Übergabe im Konstruktor, ohne sie zu modifizieren (Z. 28).

Dadurch, dass das `BehaviorSubject` ein Stream ist, kann die Selektionskarte auf Änderungen reagieren, die sich an `priorChoices` vollziehen, obwohl diese Änderungen außerhalb der Klasse geschehen. Würde stattdessen eine Liste der bisherigen Auswahloption übergeben werden, so wäre diese eine Kopie. Diese Kopie hätte den Zustand einer Momentaufnahme aller bisherigen Auswahloptionen zum Zeitpunkt der Konstruktion des `SelectionCard`-Elementes. Alle Änderungen, die nach diesem Zeitpunkt an den Auswahloptionen geschehen sind, würden sich nicht darin widerspiegeln. Eine Selektionskarte würde daher auch keinen Fehler anzeigen, wenn ihre ausgewählten Optionen durch Änderungen von außen

¹Vgl. 15.

²Vgl. 28.

```

15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16   final String title;
17   final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
18   final Choices<ChoiceType> allChoices;
19   final BehaviorSubject<Set<Choice>> priorChoices;
20   final OnSelect<ChoiceType> onSelect;
21   final OnDeselect<ChoiceType> onDeselect;
22   final String? errorText;
23
24   SelectionCard(
25     {required this.title,
26     required Iterable<ChoiceType> initialValue,
27     required this.allChoices,
28     required this.priorChoices,
29     required this.onSelect,
30     required this.onDeselect,
31     this.errorText,
32     Key? key})

```

Listing 7.6.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

invalide werden würden. Der Grund dafür ist, dass sie noch eine alte Kopie der bisherigen Auswahloptionen verwendet.

Eine andere Möglichkeit wäre, eine Setter-Methode zu implementieren, die den Wert der bisherigen Auswahloptionen neu setzt. Doch das Programm verwaltet keine Referenzen auf alle gebauten Selektionskarten. Somit kann auch nicht über eine Referenz eine Setter-Methode aufgerufen werden, denn eine solche Referenz existiert nicht. Die übliche Vorgehensweise wäre in Flutter, das gesamte Widget neu zu zeichnen. Bei Einsatz eines „*Stateful Widgets*“ und Zustandsänderungen über die `setState`-Methode würde dies das Neuzeichnen des gesamten Formulars bedeuten.

Performante ist es dagegen, wenn nur die Inhalte der Selektionskarten ausgetauscht werden. Anstatt ausschließlich auf die Änderungen der eigenen Auswahloptionen zu reagieren, horcht der StreamBuilder nun auf den Stream `priorChoices` (Listing 7.7, Z. 52) und damit auf die Änderungen aller Auswahlfelder. Vor der Konstruktion der Karte wird nun überprüft, ob einer der ausgewählten Auswahloptionen in `selectedChoices` eine invalide Auswahl enthält (Z. 55-56). Das kann über die Funktion `any` herausgefunden werden, indem für jede ausgewählte Option die Methode `conditionMatches` mit der Menge aller ausgewählten Optionen im gesamten Formular aufgerufen wird (Z. 56). Die rote Farbe der Selektionskarte wurde bereits bei der Validierung im letzten Schritt verwendet, wenn der dem Konstruktor ein `errorText` übergeben wurde. Nun wird diese Bedingung erweitert. Sollte es auch nur eine falsche Selektion geben oder aber der `errorText` gesetzt sein, so ist die Karte rot. Anderenfalls wird dem Parameter `tileColor` `null` übergeben (Z. 70). `null` bedeutet, dass keine Farbe übergeben und damit die Standardfarbe verwendet wird.


```

51 return StreamBuilder(
52   stream: priorChoices,
53   builder: (context, snapshot) {
54     final selectedChoices = selectionViewModel.value;
55     final bool wrongSelection = selectedChoices
56       .any((c) => !c.conditionMatches(priorChoices.value));
57
58     return Card(
59       child: Column(
60         crossAxisAlignment: CrossAxisAlignment.start,
61         children: [
62           ListTile(
63             focusNode: focusNode,
64             title: Text(title),
65             subtitle: Text(
66               selectedChoices.map((c) => c.description).join(", "),
67             trailing: const Icon(Icons.edit),
68             onTap: navigateToSelectionScreen,
69             tileColor:
70               wrongSelection || errorText != null ? Colors.red : null,

```

Listing 7.7.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

7.4. Reagieren des Auswahlbildschirms auf die ausgewählten Optionen

Der Auswahlbildschirm wird im Folgenden um zwei weitere Funktionalitäten erweitert (Listing 7.8). Sollten durch neue Selektionen im Formular bereits selektierte Optionen im Auswahlbildschirm nun invalide sein, so werden diese rot gefärbt. Weiterhin erscheinen invalide Optionen, die nicht ausgewählt sind, am Ende der Liste ohne Checkbox zum Auswählen. Außerdem erhält die Option ein Kreuz-Icon als Indikator dafür, dass sie nicht ausgewählt werden kann.

Zu diesem Zweck konstruiert der `StreamBuilder` vor der Rückgabe des `ListViews` zwei Mengen. Die Menge `selectedAndSelectableChoices` (Z. 95) beinhaltet alle Auswahloptionen, die entweder selektiert oder selektierbar sind. Dies beinhaltet auch Optionen, die invalide und trotzdem selektiert sind. Die zweite Menge `unselectableChoices` (Z. 96) dagegen beinhaltet alle Optionen, die invalide sind, und nicht selektiert sind.

Eine Schleife iteriert über alle verfügbaren Optionen, welche der Auswahl Bildschirm anzeigt (Z. 90-105). Sollte die Option in den selektierten Optionen enthalten (Z. 99), oder aber mit den Selektionen aller anderen Auswahlfelder kompatibel sein, so wird sie der Menge `selectedAndSelectableChoices` hinzugefügt (Z. 101). In jedem anderen Fall wird die Option Teil der Menge `unselectableChoices` (Z. 103).

Für die Konstruktion der `CheckboxListTile`-Elemente wurde zuvor die Menge aller Auswahloptionen verwendet. Nun wird stattdessen nur die Menge der selektierbaren und selektierten Auswahloptionen genutzt (Z. 108). Neben dem Vergleich, ob die Option selektiert ist (Z. 109), erfolgt nur noch ein weiterer Vergleich, ob die Option inkompatibel mit den ausgewählten Optionen aller anderen Auswahlfelder ist (Z. 111). Das Ergebnis des Vergleiches wird in der lokalen Variable `selectedButDoesNotMatch` gespeichert.

Sollte diese Variable `true` sein, so erscheint das `CheckboxListTile`-Element mit einem rot eingefärbten im Hintergrund. Der Benutzer hat über die Checkbox dann die Möglichkeit, diese Auswahl zu deselektieren. Da das hinterlegte `ViewModel` durch diese Deselektion direkt aktualisiert wird (Z. 122-123), so baut der `StreamBuilder` auch den `ListView` neu. Die deselektierte Option wird dann Teil von der Menge `unselectableChoices` (Z. 103) sein. So erscheint sie dann – ganz genau wie alle anderen unselektierbaren Auswahloptionen – ohne roten Hintergrund aber auch ohne anklickbare Checkbox am Ende der Liste (Z. 134-142).

Solche unselektierbaren Optionen werden schlicht als `ListTile`-Element statt als `CheckCoxListTile` gezeichnet (Z. 135-139). Damit fehlt ihnen die Checkbox zum Selektieren. Über den Parameter `leading` kann jedoch anstelle der Checkbox ein beliebiges Widget – in diesem Fall ein Icon – eingefügt werden. `Icons.close` zeichnet ein Kreuz-Symbol, um zu signalisieren,

dass diese Option nicht anwählbar ist.

```

88   title: Text(title),
89 ),
90 body: StreamBuilder(
91   stream: selectionViewModel,
92   builder: (context, snapshot) {
93     final selectedChoices = selectionViewModel.value;
94
95     Set<ChoiceType> selectedAndSelectableChoices = {};
96     Set<ChoiceType> unselectableChoices = {};
97
98     for (ChoiceType c in allChoices) {
99       if (selectedChoices.contains(c) ||
100         c.conditionMatches(priorChoices.value)) {
101         selectedAndSelectableChoices.add(c);
102       } else {
103         unselectableChoices.add(c);
104       }
105     }
106
107     return ListView(children: [
108       ...selectedAndSelectableChoices.map((ChoiceType c) {
109         bool isSelected = selectedChoices.contains(c);
110         bool selectedButDoesNotMatch =
111           !c.conditionMatches(priorChoices.value);
112
113         return CheckboxListTile(
114           key: Key(
115             "valid choice ${allChoices.name} - ${c.abbreviation}"),
116           controlAffinity: ListTileControlAffinity.leading,
117           title: Text(c.description),
118           tileColor: selectedButDoesNotMatch ? Colors.red : null,
119           value: isSelected,
120           onChanged: (selected) {
121             if (selected != null) {
122               selectionViewModel.value =
123                 selectionViewModel.value.rebuild((b) {
124                   b.replace(isSelected ? [] : [c]);
125                 });
126             if (selected) {
127               onSelect(c);
128             } else {
129               onDeselect(c);
130             }
131           }
132         );
133       }).toList(),
134       ...unselectableChoices.map((Choice c) {
135         return ListTile(
136           key: Key(
137             "invalid choice ${allChoices.name} - ${c.abbreviation}"),
138           title: Text(c.description),
139           leading: const Icon(Icons.close));
140       }).toList()
141     ]);

```

Listing 7.8.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

7.4.1. Hinzufügen der Momentaufnahme zur Validierung

Alle bisher eingefügten Vergleiche hatten lediglich den Zweck, die invaliden Optionen einzufärben und von der Selektion durch den Benutzer auszuschließen. Doch noch sind sie nicht Teil der Validierung des Formulars. Sollte der Benutzer die aktuell eingetragene Maßnahmen im abgeschlossenen Status abspeichern wollen, so kann dies auch mit invaliden Optionen erfolgen. Um das zu verhindern, wird noch ein Vergleich zu der anonymen Funktion hinzugefügt, welche als Argument dem Parameter `validator` des `FormField` übergeben wird (Listing 7.9). Sollte auch nur eine der selektierten Optionen `choices` die ihr hinterlegte Bedingungen nicht erfüllen (Z. 132), so speichert die lokale Variable `atLeastOneValueInvalid` den Wert `true` ab (Z. 131).

In dem Fall gibt die Funktion die entsprechende Fehlermeldung an den Benutzer zurück (Z. 135). Somit ist es nun auch nicht mehr möglich, eine Maßnahme abzuspeichern, wenn sie invalide Auswahloptionen enthält. Erst wenn alle Auswahlfelder gefüllt sind und die gefüllten Optionen alle die jeweils hinterlegten Bedingungen erfüllen, so werden die `validator`-Funktionen `null` statt einer Fehlermeldung zurückgeben (Z. 138). Nur dann kann eine Maßnahme mit dem Status „abgeschlossen“ gespeichert werden.

```

121 return FormField(
122   validator: (_) {
123     Iterable<Choice> choices = {
124       if (selectionViewModel.value != null) selectionViewModel.value!
125     };
126
127     if (choices.isEmpty) {
128       return "Feld ${allChoices.name} enthält keinen Wert!";
129     }
130
131     bool atLeastOneValueInvalid =
132       choices.any((c) => !c.conditionMatches(vm.priorChoices.value));
133
134     if (atLeastOneValueInvalid) {
135       return "Wenigstens ein Wert im Feld ${allChoices.name} enthält ist fehlerhaft!";
136     }
137
138     return null;
139   },
140   builder: (field) => SelectionCard<ChoiceType>(

```

Listing 7.9.: Die Ausgabe der Formularfelder, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

8. Schritt 5

Im letzten Schritt wurde das primäre Problem der Formularanwendung gelöst: Auswahloptionen sollen nur dann anwählbar sein, wenn sie die ihr hinterlegte Bedingung erfüllen. Darüber hinaus können nur Maßnahmen gespeichert werden, deren Auswahloptionen untereinander kompatibel sind.

Durch das Lösen dieses Problems ist ein neues Problem entstanden: Alle Selektionskarten müssen bei einer Selektion neu gezeichnet werden. Bei einer geringen Anzahl von Auswahlfeldern sollte das noch keine gravierenden Auswirkungen auf das Laufzeitverhalten der Applikation haben. Doch je zahlreicher die Auswahlfelder werden, desto länger dauert die Aktualisierung der Oberfläche.

Das Problem kann folgendermaßen entschärft werden: Noch bevor das Widget `SelectionCard` den `StreamBuilder` in der `build`-Methode zurückgibt, wird ein neuer Stream namens `validityChanged` erstellt (Listing 8.1, Z. 51-54).

Es handelt sich um eine sogenannte Transformation des Streams `priorChoices`, welcher die Momentaufnahme aller ausgewählten Optionen im gesamten Formular übermittelt. Immer dann, wenn der Stream `priorChoices` ein neues Ereignis sendet, geschieht für die Abwandlung dieses Streams folgendes: Die Methode `map` wandelt jedes Ereignis in ein neues Objekt um (Z. 52). Die aktuelle Momentaufnahme der Auswahloptionen im Formular wird dazu im Parameter `choices` gespeichert. Bei der Umwandlung des Ereignisses werden die ausgewählten Optionen der aktuellen Selektionskarte über `selectionViewModel.value` abgerufen. Sollte es sich beispielsweise bei der aktuellen Selektionskarte um das Auswahlfeld der „Kategorie“ handeln, so könnte der ausgewählte Wert „Düngemanagement“ sein. Für den Wert oder die Werte wird nun überprüft, ob sie mit der neuen Momentaufnahme der Selektionen im Formular kompatibel sind. Wurde also beispielsweise bei der neuen Selektion in der „Förderklasse“ nun „Ökolandbau“ ausgewählt, so würde die Option „Düngemanagement“ nun invalide werden, da sie nur mit der Förderklasse „Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz“ bzw. „Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohl-Aspekten, aber OHNE Vertragsnaturschutz“ kompatibel ist. Die Methode `map` wandelt also das neue Ereignis der Momentaufnahme aller Selektionen im Formular in einen einzigen Wahrheitswert um. Ist der Wahrheitswert `true`, bedeutet dies, dass alle ausgewählten Optionen in der aktuellen Selektionskarte valide sind. Ist er dagegen `false`, so ist wenigstens eine der Auswahloption mit den restlichen Auswahloptionen der anderen Auswahlfelder im

Formularen nicht kompatibel.

Der resultierende Stream wird weiter transformiert: Durch die Funktion `distinct` (Z. 54) werden nur Ereignisse gesendet, sofern sie sich von dem letzten Ereignis unterscheiden. Ein Beispiel: Für die „Kategorie“ ist „Düngemanagement“ ausgewählt. Für die „Förderklasse“ ist „Erschwerungsungleich“ im letzten Ereignis ausgewählt worden. „Düngemanagement“ ist mit „Erschwerungsungleich“ nicht kompatibel, weshalb das letzte Ereignis des durch `map` transformierten Streams `false` war. Nun wird für die „Förderklasse“ eine weitere Selektion vorgenommen: „Ökologischer Landbau“ wird ausgewählt. Auch diese Option ist mit „Düngemanagement“ nicht kompatibel. Der durch `map` transformierte Stream wird also erneut ein Ereignis mit dem Wert `false` senden. Doch bereits das letzte Ereignis war `false`. Die Methode `distinct` verhindert, dass dieses redundante Ereignis weitergeleitet wird. Nun erfolgt noch eine weitere Selektion: Für die „Förderklasse“ wird „Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz“ selektiert. Nun ist die „Kategorie“ „Düngemanagement“ mit der neuen Selektion kompatibel. Der aus der Methode `map` resultierende Stream liefert dieses Mal den Wert `true`. Das letzte Ereignis hatte den Wert `false`. Die Werte der beiden letzten Ereignisse unterscheiden sich also, was dazu führt, dass die Methode `distinct` das veränderte Ereignis nicht filtert sondern weiterleitet.

Der Stream `validityChanged` sendet also immer genau dann Ereignisse, wenn sich etwas an der Validität der Auswahloptionen der aktuellen Selektionskarte ändert. Doch dieser Stream kann nicht für den `StreamBuilder` benutzt werden. Denn wenn sich die Auswahl in der aktuellen Selektionskarte ändert und die Validität dadurch unverändert bleibt, so erfolgt kein neues Zeichnen der Selektionskarte. Deshalb ist eine Kombination der Streams `validityChanged` und `selectionViewModel` erforderlich. Das `BehaviorSubject` `needsRepaint` soll als diese Kombination fungieren (Z. 56). Es wird mit dem Wert (Z. true) initialisiert. Es ist unerheblich, welcher Wert in dem Stream aktuell gespeichert ist. Lediglich dass ein neues Ereignis hinzugefügt wird, um die Aktualisierung der Oberfläche auszulösen, ist wesentlich. Mit der Methode `listen` wird nun sowohl auf den Stream `validityChanged` (Z. 57) als auch auf `selectionViewModel` (Z. 58) gehorcht. Jedes empfangene Ereignis wird dabei dem `BehaviorSubject` `needsRepaint` hinzugefügt.

Dadurch, dass `needsRepaint` für den `StreamBuilder` verwendet wird (Z. 61), zeichnet sich die Selektionskarte immer dann neu, wenn sich die beinhaltenden Auswahloptionen oder aber dessen Validität ändert.

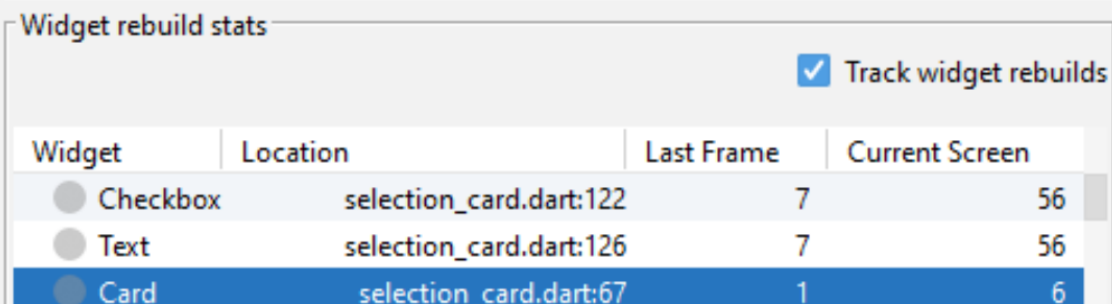
Dieses Verhalten kann auch bei Ausführung der Applikation im Debugmodus in Android Studio beobachtet werden. Der „Flutter Performance“-Tab gibt eine Übersicht über die Anzahl der im letzten Frame neu gezeichneten Widgets (Abb. 8.1). Angenommen für die „Förderklasse“ ist „Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz“ und für die „Kategorie“ ist „Düngemanagement“ ausgewählt. Wenn nun für die Förderklasse die Option „Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE Vertragsnaturschutz“ selektiert wird, so ist im „Flutter Performance“-Tab zu beobachten,


```

51 final validityChanged = priorChoices
52   .map((choices) =>
53     selectionViewModel.value.any((c) => !c.conditionMatches(choices)))
54   .distinct();
55
56 final needsRepaint = BehaviorSubject.seeded(true);
57 validityChanged.listen((value) => needsRepaint.add(true));
58 selectionViewModel.listen((value) => needsRepaint.add(true));
59
60 return StreamBuilder(
61   stream: needsRepaint,
62   builder: (context, snapshot) {
63     final selectedChoices = selectionViewModel.value;
64     final bool wrongSelection = selectedChoices
65       .any((c) => !c.conditionMatches(priorChoices.value));
66
67     return Card(
68       child: Column(
69         crossAxisAlignment: CrossAxisAlignment.start,
70         children: [
71           ListTile(

```

Listing 8.1.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-5/conditional_form/lib/widgets/selection_card.dart](#)



Widget	Location	Last Frame	Current Screen
Checkbox	selection_card.dart:122	7	56
Text	selection_card.dart:126	7	56
Card	selection_card.dart:67	1	6

Abbildung 8.1.: XXXXX, Quelle: Eigene Abbildung

dass das Widget Card nur einmal neu gezeichnet wurde.

Das ergibt Sinn, denn es hat sich nichts an der Validität eines anderen Auswahlfeld geändert. Lediglich die Selektionskarte für die Förderklasse muss neu gezeichnet werden, da sich seine Selektion angepasst hat. Wird nun aber die „Förderklasse“ „Ökolandbau“ ausgewählt, so ist zu beobachten, dass das `card` Widget zweimal gebaut wurde: Einmal für die Selektionskarte der „Förderklasse“, da sich dessen ViewModel änderte; Ein weiteres Mal für die Selektionskarte der „Kategorie“, da die Auswahl „Düngemanagement“ nicht länger valide ist und die Karte deshalb mit einem roten Hintergrund eingefärbt werden muss (Abb. 8.2).

Ohne die Änderungen in diesem Schritt zeigt der „Flutter Performance“-Tab, dass sich bei jeder Auswahl einer Option sechs `card`-Elemente aktualisieren (Abb. 8.3). Das ist der Fall, weil es in Summe sechs Auswahlfelder gibt.

Widget rebuild stats ☒ Track widget rebuilds

Widget	Location	Last Frame	Current Screen
<input type="radio"/> Checkbox	selection_card.dart:122	7	42
<input type="radio"/> Text	selection_card.dart:126	7	42
<input checked="" type="radio"/> Card	selection_card.dart:67	2	7

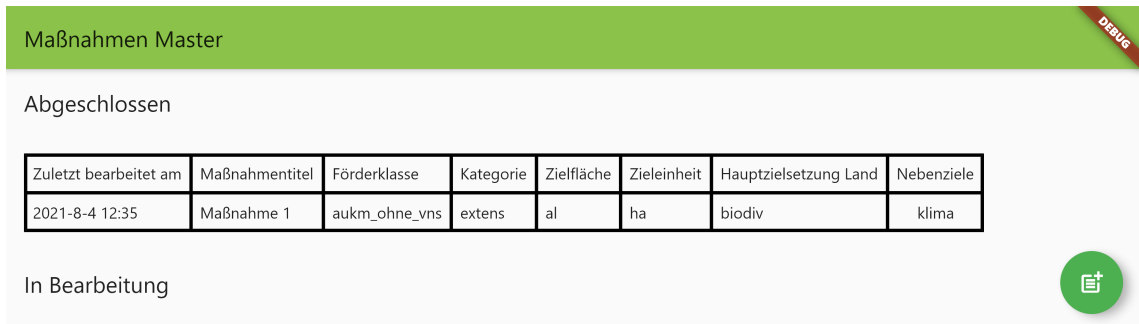
Abbildung 8.2.: XXXXX, Quelle: Eigene Abbildung

Widget rebuild stats ☒ Track widget rebuilds

Widget	Location	Last Frame	Current Screen
<input type="radio"/> Checkbox	selection_card.dart:122	7	28
<input type="radio"/> Text	selection_card.dart:126	7	28
<input checked="" type="radio"/> Card	selection_card.dart:67	6	12

Abbildung 8.3.: XXXXX, Quelle: Eigene Abbildung

9. Schritt 6



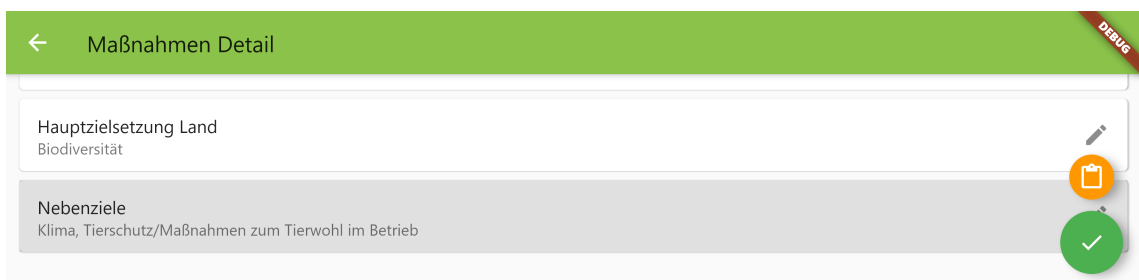
Maßnahmen Master DRAFT

Abgeschlossen

Zuletzt bearbeitet am	Maßnahmentitel	Förderklasse	Kategorie	Zielfläche	Zieleinheit	Hauptzielsetzung Land	Nebenziele
2021-8-4 12:35	Maßnahme 1	aukm_ohne_vns	extens	al	ha	biodiv	klima

In Bearbeitung +

Abbildung 9.1.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung



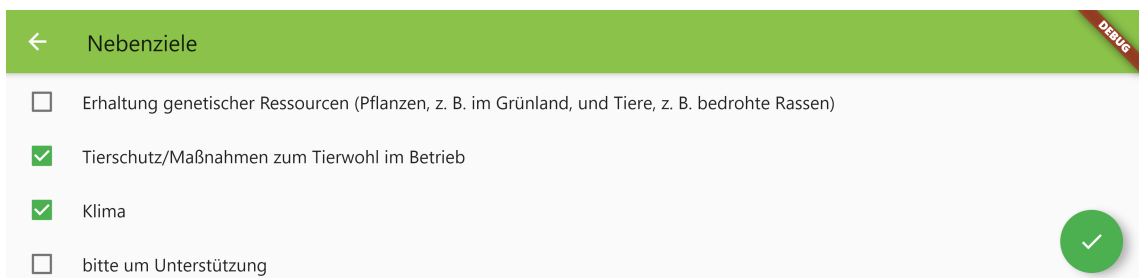
← **Maßnahmen Detail** DRAFT

Hauptzielsetzung Land
Biodiversität

Nebenziele
Klima, Tierschutz/Maßnahmen zum Tierwohl im Betrieb

+
✓

Abbildung 9.2.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung



← **Nebenziele** DRAFT

- ☐ Erhaltung genetischer Ressourcen (Pflanzen, z. B. im Grünland, und Tiere, z. B. bedrohte Rassen)
- ☒ Tierschutz/Maßnahmen zum Tierwohl im Betrieb
- ☒ Klima
- ☐ bitte um Unterstützung

✓

Abbildung 9.3.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

In diesem Schritt soll das Formular um Mehrfachauswahlfelder erweitert werden. Im Speziellen handelt es sich um das Auswahlfeld Nebenziele. Es beinhaltet die gleichen Auswahloptionen wie das Auswahlfeld Hauptzielsetzung.

9.1. Integrationstest erweitern

Zunächst wird der Integrationstest um die Auswahl der Nebenziele erweitert (Listing 9.1).

```

118 await tabSelectionCard(hauptzielsetzungLandChoices);
119 await tabOption(ZielsetzungLandChoice.biodiv, tabConfirm: true);
120
121 await tabSelectionCard(nebenzielsetzungLandChoices);
122 await tabOption(ZielsetzungLandChoice.bsch);
123 await tabOption(ZielsetzungLandChoice.klima, tabConfirm: true);
124
125 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
126 await tester.tap(saveMassnahmeButton);
127 await tester.pumpAndSettle(durationAfterEachStep);

```

Listing 9.1.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/integration_test/app_test.dart](#)

Zu diesem Zweck löst der Test nach der Auswahl der Hauptzielsetzung (Z. 118-119) nun einen Klick auf die Selektionskarte für die Nebenzzielsetzung aus (Z. 121). Dadurch öffnet sich der Auswahlbildschirm, in welchem die Option „Bodenschutz“ (Z. 122) und anschließend die Option „Klima“ (Z. 123) gewählt wird. Mit Auswahl der letzten Option und durch die damit verbundene Übergabe des Arguments `true` für den optionalen Parameter `tabConfirm` wird der Auswahlbildschirm umgehend wieder geschlossen. Anschließend erfolgt erneut das Speichern der Maßnahme (Z. 125-126).

Anders als bei den bisherigen Schlüssel-Werte-Paaren innerhalb des Objektes `'massnahmenCharakteristika'` kann der Wert der Nebenziele nicht als einzelner String gespeichert werden (Listing 9.2).

```

136 var expectedJson = {
137   'letzteBearbeitung': {'letzterStatus': 'fertig'},
138   'identifikatoren': {'massnahmenTitel': massnahmeTitle},
139   'massnahmenCharakteristika': {
140     'nebenziele': [
141       'bsch',
142       'klima',
143     ],
144     'foerderklasse': 'aukm_ohne_vns',
145     'kategorie': 'extens',
146     'zielflaeche': 'al',
147     'zieleinheit': 'ha',
148     'hauptzielsetzungLand': 'biodiv'
149   },
150 };

```

Listing 9.2.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/integration_test/app_test.dart](#)

Bei dem Inhalt der Mehrfachauswahlfelder handelt es sich schließlich um eine Auflistung mehrerer Werte. Sie wird im erwarteten JSON-Dokument als Array-Literal codiert (Z. 140-143).

9.2. Hinzufügen der Menge der Nebenziele

Für die Menge der Nebenziele müssen keine weiteren Auswahloptionen hinzugefügt werden. Es werden die gleichen Optionen verwendet, die auch bei der Menge mit dem Namen „Hauptzielsetzung Land“ zum Einsatz kommen (Listing 9.3, Z. 123-124).

```

219 final hauptzielsetzungLandChoices = Choices<ZielsetzungLandChoice>(
220     _zielsetzungLandChoices,
221     name: "Hauptzielsetzung Land");
222
223 final nebenzielsetzungLandChoices =
224     Choices<ZielsetzungLandChoice>(_zielsetzungLandChoices, name: "Nebenziele");

```

Listing 9.3.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/choices/choices.dart](#)

9.3. Aktualisierung des Models

Um die Liste der Nebenziele im Wertetyp `MassnahmenCharakteristika` einzufügen, kann der Datentyp `BuiltSet` verwendet werden (Listing 9.4, Z. 77). Die Getter-Methode `nebenziele`

```

68 abstract class MassnahmenCharakteristika
69     implements
70         Built<MassnahmenCharakteristika, MassnahmenCharakteristikaBuilder> {
71     String? get foerderklasse;
72     String? get kategorie;
73     String? get zielflaeche;
74     String? get zieleinheit;
75     String? get hauptzielsetzungLand;
76
77     BuiltSet<String> get nebenziele;

```

Listing 9.4.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/data_model/massnahme.dart](#)

bedarf keiner Null-Zulässigkeit, da das Nicht-Vorhandensein von Werten darüber erreicht werden kann, dass die Menge leer ist.

9.4. Aktualisierung der Übersichtstabelle

Für das Einfügen der Überschrift in der Übersichtstabelle gibt es keine Unterschiede zum bisherigen Vorgehen. Die Überschrift wird nach der Spaltenüberschrift für die „Hauptzielsetzung“ eingefügt (Listing 9.5, Z. 28).

Die Anzeige der Werte in den Tabellenzellen ist dagegen unterschiedlich (Listing 9.6). Dieses Mal handelt es sich um die Aufzählung von mehreren Werten, weshalb ein `Column`-

```

27 _buildColumnHeader(const Text("Hauptzielsetzung Land")),
28 _buildColumnHeader(const Text("Nebenziele")),

```

Listing 9.5.: XXXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/massnahmen_table.dart](#)

Widget die einzelnen Einträge untereinander auflistet (Z. 46-49). Jedes Element des `BuiltSet` `nebenziele` (Z. 47) wird über die Methode `map` jeweils in ein Element des Widgets `Text` konvertiert (Z. 48).

```

42 _buildSelectableCell(m,
43   Text(m.massnahmenCharakteristika.hauptzielsetzungLand ?? "")),
44 _buildSelectableCell(
45   m,
46   Column(
47     children: m.massnahmenCharakteristika.nebenziele
48       .map((n) => Text(n))
49       .toList(),
50   )),

```

Listing 9.6.: XXXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/massnahmen_table.dart](#)

9.5. Aktualisierung des ViewModels

Die Nebenziele werden – erneut Mit dem Datentyp `BuiltSet` – im ViewModel hinzugefügt (Listing 9.7).

```

18 final hauptzielsetzungLand =
19   BehaviorSubject<ZielsetzungLandChoice?>.seeded(null);
20 final nebenziele = BehaviorSubject<BuiltSet<ZielsetzungLandChoice>>.seeded(
21   BuiltSet<ZielsetzungLandChoice>());

```

Listing 9.7.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

Der benannte Konstruktor `seeded` initialisiert die Instanzvariable mit einer leeren Menge (Z. 20). Dafür wird der parameterlose Konstruktor von `BuiltSet` aufgerufen (Z. 21). Dadurch unterscheidet sich das `BehaviorSubject` von den anderen im ViewModel und muss dementsprechend bei der Konvertierung zwischen Model in ViewModel gesondert behandelt werden.

Bei Konvertierung von Model in ViewModel sind für alle Auswahloptionen – genau wie in den Schritten zuvor – jeweils nur die Abkürzungen verfügbar. Die Liste der gespeicherten Abkürzungen der Nebenziele muss dementsprechend zuerst in eine Menge von Auswahloptionen konvertiert werden, bevor sie dem `BuiltSet` übergeben werden kann (Listing 9.8). Die Methode `map` löst das Problem, indem sie die ihr als Argument übergebene Funktion für jede Abkürzung in der Menge „Nebenziele“ aufruft (Z. 65). Die übergebene anonyme

Funktion konvertiert die Abkürzung in die zugehörige Auswahloption. Die resultierende Menge kann dem Konstruktor von `BuiltSet` übergeben werden (Z. 64-65).

```

46 set model(Massnahme model) {
47   guid.value = model.guid;
48
49   letzterStatus.value = letzterStatusChoices
50     .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
51   massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
52
53   {
54     final mc = model.massnahmenCharakteristika;
55
56     foerderklasse.value =
57       foerderklasseChoices.fromAbbreviation(mc.foerderklasse);
58     kategorie.value = kategorieChoices.fromAbbreviation(mc.kategorie);
59
60     zielflaeche.value = zielflaecheChoices.fromAbbreviation(mc.zielflaeche);
61     zieleinheit.value = zieleinheitChoices.fromAbbreviation(mc.zieleinheit);
62     hauptzielsetzungLand.value =
63       hauptzielsetzungLandChoices.fromAbbreviation(mc.hauptzielsetzungLand);
64     nebenziele.value = BuiltSet(mc.nebenziele
65       .map((n) => hauptzielsetzungLandChoices.fromAbbreviation(n)));
66   }
67 }

```

Listing 9.8.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

Ähnlich verhält es sich bei der Umwandlung des ViewModels in das Model (Listing 9.9).

```

69 Massnahme get model => Massnahme((b) => b
70   ..guid = guid.value
71   ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
72   ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
73   ..identifikatoren.update((b) => b..massnahmenTitel = massnahmenTitel.value)
74   ..massnahmenCharakteristika.update((b) => b
75     ..foerderklasse = foerderklasse.value?.abbreviation
76     ..kategorie = kategorie.value?.abbreviation
77     ..zielflaeche = zielflaeche.value?.abbreviation
78     ..zieleinheit = zieleinheit.value?.abbreviation
79     ..hauptzielsetzungLand = hauptzielsetzungLand.value?.abbreviation
80     ..nebenziele =
81       SetBuilder(nebenziele.value.map((n) => n.abbreviation).toList()));

```

Listing 9.9.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

In diesem Fall muss die Menge der Auswahloptionen der „Nebenziele“ in die entsprechenden Abkürzungen umgewandelt werden, bevor sie im „Model“ gespeichert wird. Die Methode `map` erhält zu diesem Zweck erneut eine anonyme Funktion, welche die Abkürzung der Auswahloptionen abfragt (Z. 81). Die resultierende Menge wird als Parameter dem Konstruktor `SetBuilder` übergeben (Z. 80-81). Der `SetBuilder` wiederum kümmert sich um das Bauen des `BuiltSet`, sobald ein Objekt des Typs `Massnahme` gebaut wird.

9.6. Aktualisierung der Eingabemaske

Unterhalb des Auswahlfeldes für das Hauptziel wird die Selektionkarte für die Nebenziele eingefügt (Listing 9.10).

```

212 buildSelectionCard<ZielsetzungLandChoice>(
213     allChoices: hauptzielsetzungLandChoices,
214     selectionViewModel: vm.hauptzielsetzungLand),
215 buildMultiSelectionCard(
216     allChoices: nebenzielsetzungLandChoices,
217     selectionViewModel: vm.nebenziele),

```

Listing 9.10.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Allerdings handelt es sich dieses Mal um ein Mehrfachauswahlfeld, weshalb eine neue Methode namens `buildMultiSelectionCard` aufgerufen wird (Z. 215-217).

Da nun zwei Methoden zum Erstellen von Elementen des Widgets `SelectionCard` existieren, ist es sinnvoll, den Quellcode zu refaktorisieren, um redundanten Code zu vermeiden.

Innerhalb der bereits vorhandenen Methode `buildSelectionCard` wird die Routine, welche für die Validierung des Formulars genutzt wird, in eine neue Methode namens `validateChoices` (Listing 9.11, Z. 123-128) ausgelagert.

```

119 Widget buildSelectionCard<ChoiceType extends Choice>(
120     {required Choices<ChoiceType> allChoices,
121     required BehaviorSubject<ChoiceType?> selectionViewModel}) {
122     return FormField(
123         validator: (_) => validateChoices(
124             name: allChoices.name,
125             choices: {
126                 if (selectionViewModel.value != null) selectionViewModel.value!
127             },
128             priorChoices: vm.priorChoices.value),
129     builder: (field) => SelectionCard<ChoiceType>(

```

Listing 9.11.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Sie bekommt die Attribute für den Namen der Menge (Z. 124), die zu validierenden Optionen (Z. 125-127) und schließlich die bisher ausgewählten Optionen aller Auswahlfelder (Z. 128) übergeben. Die ausgelagerte Funktion ist in Anhang ?? in Listing ?? auf Seite ?? zu finden.

Für die Erstellung der Mehrfachauswahlfelder ist die Methode `buildMultiSelectionCard` zuständig (Listing 9.12).

Das übergebene `selectionViewModel` unterstützt mit dem Typometer `BuiltSet` die Auswahl


```

144 Widget buildMultiSelectionCard<ChoiceType extends Choice>(
145   {required Choices<ChoiceType> allChoices,
146   required BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel}) {
147   return FormField(
148     validator: (_) => validateChoices(
149       name: allChoices.name,
150       choices: selectionViewModel.value,
151       priorChoices: vm.priorChoices.value),
152     builder: (field) => SelectionCard<ChoiceType>(
153       title: allChoices.name,
154       multiSelection: true,
155       allChoices: allChoices,
156       priorChoices: vm.priorChoices,
157       initialValue: selectionViewModel.value,
158       onSelect: (selectedChoice) => selectionViewModel.value =
159         selectionViewModel.value
160           .rebuild((b) => b.add(selectedChoice)),
161       onDeselect: (selectedChoice) => selectionViewModel.value =
162         selectionViewModel.value
163           .rebuild((b) => b.remove(selectedChoice)),
164       errorText: field.errorText,
165     ));
166 }

```

Listing 9.12.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

von mehreren Auswahloption (Z. 146). Bei `selectionViewModel` handelt es sich bereits um eine Menge. Für die Validierung 150 sowie für die Übergabe des initialen Wertes an den Konstruktor der `SelectionCard` (Z. 157) ist eine Umwandlung in eine Menge daher nicht mehr nötig. Dem Konstruktor `SelectionCard` wird weiterhin über den Parameter `multiSelection` mitgeteilt, dass mehr als eine Auswahl zum gewählt werden darf (Z. 154). Die Methoden `onSelect` und `onDeselect` ersetzen nun nicht mehr den aktuell gespeicherten Wert über eine einfache Zuweisung. Sie nutzen stattdessen die Methode `rebuild` des `BuiltSet` um ein Element mit Hilfe von `add` hinzuzufügen (Z. 160) bzw. mit `remove` Elemente zu entfernen (Z. 163). Der Methodenaufruf `rebuild` sorgt jedoch nicht für das Hinzufügen oder Löschen am Original-Objekt, sondern erstellt eine Kopie der Liste mit der gewünschten Änderungen. Deshalb erfolgt eine Zuweisung der Kopie zum Wert des `BehaviorSubject`-Objekts, was wiederum das Auslösen eines neuen Ereignisses bewirkt (Z. 158,161).

9.7. Aktualisierung der Selektionskarte

Diese Selektionskarte wird um die Instanzvariable `multiSelection` erweitert (Listing 9.13, Z. 17), dessen Wert im Konstruktor übergeben wird (Z. 27) aber auch ausgelassen werden kann, da der Standardwert `false` angegeben ist.

Die Rückruffunktion `onChanged` des `CheckboxListTile` unterscheidet schließlich zwischen Mehrfach- und Einzel-Selektion. Sollte `multiSelection` mit `true` gesetzt sein (Z. 133), so erstellt die Methode `rebuild` von `BuiltSet` eine Kopie des aktuellen ViewModels der Se-

```

15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16   final String title;
17   final bool multiSelection;
18   final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
19   final Choices<ChoiceType> allChoices;
20   final BehaviorSubject<Set<Choice>> priorChoices;
21   final OnSelect<ChoiceType> onSelect;
22   final OnDeselect<ChoiceType> onDeselect;
23   final String? errorText;
24
25   SelectionCard(
26     {required this.title,
27     this.multiSelection = false,

```

Listing 9.13.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/selection_card.dart](#)

lektionen. In der anonymen Funktion, welche für die Manipulationen an der Kopie genutzt wird, wird in einer Fallunterscheidung überprüft, ob das angewählte Element bereits selektiert ist (Z. 136). Sollte das der Fall sein, so wird diese bereits selektierte Option, die nun erneut angewählt wurde, mit der Methode `remove` des Builder-Objekts aus dem `BuiltSet` entfernt (Z. 137). Anderenfalls war die Option nicht selektiert, weshalb sie mit der Methode `add` hinzugefügt wird.

```

124 return CheckboxListTile(
125   key: Key(
126     "valid choice ${allChoices.name} - ${c.abbreviation}"),
127   controlAffinity: ListTileControlAffinity.leading,
128   title: Text(c.description),
129   tileColor: selectedButDoesNotMatch ? Colors.red : null,
130   value: isSelected,
131   onChanged: (selected) {
132     if (selected != null) {
133       if (multiSelection) {
134         selectionViewModel.value =
135           selectionViewModel.value.rebuild((b) {
136             if (selectionViewModel.value.contains(c)) {
137               b.remove(c);
138             } else {
139               b.add(c);
140             }
141           });
142       } else {
143         selectionViewModel.value =
144           selectionViewModel.value.rebuild((b) {
145             b.replace(isSelected ? [] : [c]);
146           });
147       }
148       if (selected) {
149         onSelect(c);
150       } else {
151         onDeselect(c);
152       }
153     }
154   });

```

Listing 9.14.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/selection_card.dart](#)

10. Schritt 7

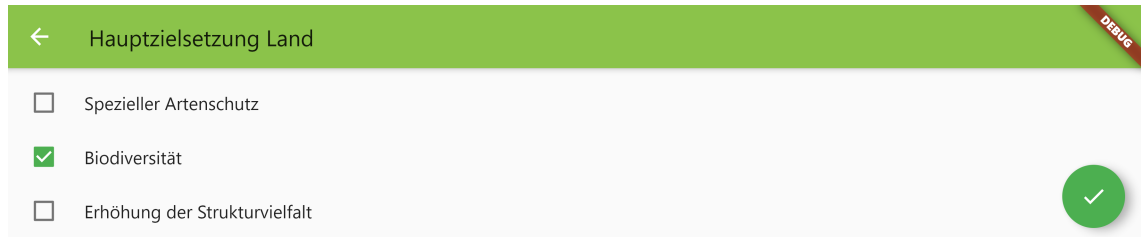


Abbildung 10.1.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung



Abbildung 10.2.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

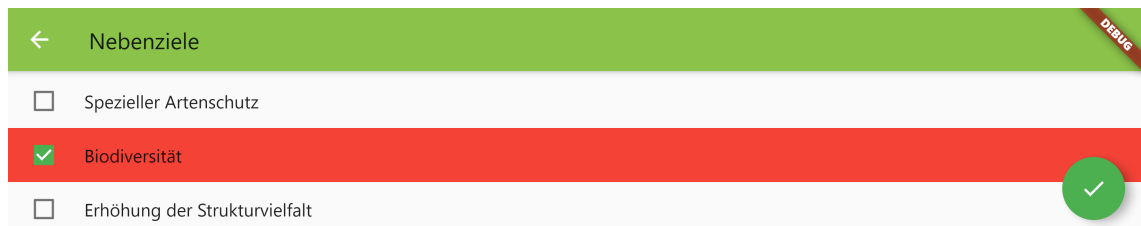


Abbildung 10.3.: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel, Quelle: Eigene Abbildung

Nachdem im letzten Schritt nun die Mehrfachauswahl für die Nebenziel hinzugefügt wurde, soll in diesem Schritt die Möglichkeit geschaffen werden, benutzerdefinierte Abhängigkeiten für Auswahloptionen anzugeben. Denn die Nebenziele haben mehrere besondere Voraussetzungen:

Sollte das Hauptziel nicht gesetzt sein oder die Option „keine Angabe/Vorgabe“ oder „bitte um Unterstützung“ enthalten, so ist es nicht sinnvoll, dass ein tatsächliches Nebenziel gewählt wird. In diesem Fall kommen wiederum nur die Werte „keine Angabe/Vorgabe“ oder „bitte um Unterstützung“ infragen.

Sollte dagegen ein Hauptziel gesetzt sein, so darf das Nebenziel nicht die gleiche Option enthalten. Diese Bedingungen lassen sich nicht mit Funktion `condition` der Basisklasse `Choice` lösen.

Denn das Argument `priorChoices`, welches der Funktion `condition` übergeben wird, enthält zwar alle Auswahloptionen, die im gesamten Formular gewählt worden, gibt aber keine Auskunft darüber, von welchem Auswahlfeld sie stammen. Sollte also die Auswahloptionen „Biodiversität“ in der Menge der `priorChoices` auftauchen, so ist unklar, ob sie im Auswahlfeld für das Hauptziel oder dem der Nebenziele gewählt wurde.

Wenn der Selektionskarte aber eine benutzerdefinierte Funktion übergeben werden könnte, welche im aufrufenden Kontext auch Zugriff auf das ViewModel hat, so könnte direkt auf die Auswahlfelder zugegriffen werden.

Zu diesem Zweck wird der Klasse `SelectionCard` die Instanzvariable `choiceMatcher` hinzugefügt (Listing 10.1, Z. 27). Ein Parameter des gleichen Namens wird den Hilfsmethoden `buildSelectionCard` und `buildMultiSelectionCard` welche ihn unverändert an den Konstruktor der Klasse `SelectionCard` weitergeleitet. Die entsprechenden Listing sind in Anhang ?? auf den Seiten 144 und ?? zu finden.

```

13 typedef ChoiceMatcher<ChoiceType extends Choice> = bool Function(
14     ChoiceType choice, Set<Choice> priorChoices);
15
16 bool defaultChoiceMatcherStrategy(Choice choice, Set<Choice> priorChoices) {
17     return choice.conditionMatches(priorChoices);
18 }
19
20 const confirmButtonTooltip = 'Auswahl übernehmen';
21
22 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
23     final String title;
24     final bool multiSelection;
25     final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
26     final Choices<ChoiceType> allChoices;
27     final BehaviorSubject<Set<Choice>> priorChoices;
28     final OnSelect<ChoiceType> onSelect;
29     final OnDeselect<ChoiceType> onDeselect;
30     final String? errorText;
31     final ChoiceMatcher<ChoiceType> choiceMatcher;
32
33     SelectionCard(
34         {required this.title,
35         this.multiSelection = false,
36         required Iterable<ChoiceType> initialValue,
37         required this.allChoices,
38         required this.priorChoices,
39         required this.onSelect,
40         required this.onDeselect,
41         ChoiceMatcher<ChoiceType>? choiceMatcher,
42         this.errorText,
43         Key? key})
44         : selectionViewModel = BehaviorSubject<BuiltSet<ChoiceType>>.seeded(
45             BuiltSet.from(initialValue)),
46           this.choiceMatcher = choiceMatcher ?? defaultChoiceMatcherStrategy,
47           super(key: key);

```

Listing 10.1.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

Der initialisierende Wert kann im Konstruktor gesetzt (Z. 41), aber auch ausgelassen wer-

den, da er nicht mit dem `required`-Schlüsselwort gekennzeichnet und damit nicht verpflichtend ist. Doch aus diesem Grund kann der Parameter den Wert `null` annehmen, weshalb er mit dem Suffix `?` gekennzeichnet werden muss. In der Initialisierungsliste erfolgt die Initialisierung der Instanzvariable `choiceMatcher` (Z. 46). Sollte der im Konstruktor übergebene Parameter nicht `null` sein, so wird er der Instanzvariable zugewiesen. Ist der aber `null`, so sorgt die „*If-null Expression*“ dafür, dass der Standardwert rechts von dem `??` zugewiesen wird: die Funktion `defaultChoiceMatcherStrategy` 46. Diese Funktion kapselt die Überprüfung der Abhängigkeiten – welche die Auswahloption und untereinander haben – so wie sie in den letzten Schritten durchgeführt wurde (Z. 16-18). Ihr wird die zu überprüfende Auswahloption `choice`, sowie die Menge `priorChoices` – die mit allen bisher ausgewählten Auswahloptionen im Formular gefüllt ist – übergeben (Z. 16). Die Auswahloption `choice` ruft – wie zuvor auch – die Methode `conditionMatches` auf und übergibt ihr das Objekt `priorChoices` (Z. 17). Diese Implementierung soll immer dann verwendet werden, wenn kein benutzerdefinierter `choiceMatcher` übergeben wurde. An dem Namen `defaultChoiceMatcherStrategy` wird offensichtlich, um welches Entwurfsmuster es sich hierbei handelt: das „*Strategie-Entwurfsmuster*“.

Strategie-Entwurfsmuster Das Strategie-Entwurfsmuster ist ein Verhaltensmuster der Gang of Four. Es erlaubt Algorithmen zu kapseln und auszutauschen¹. Abbildung 10.4 zeigt das UML-Diagramm des „*Strategie-Entwurfsmusters*“.

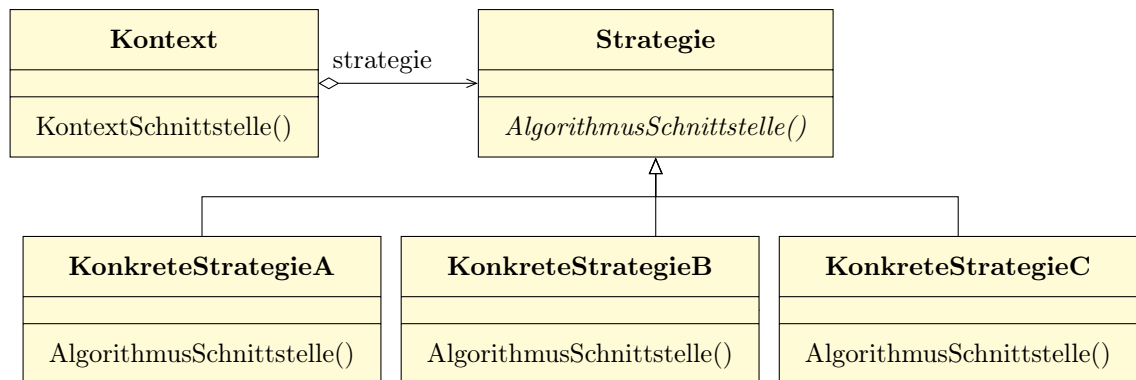


Abbildung 10.4.: UML Diagramme, Quelle: Eigene Abbildung

Die Typdefinition `ChoiceMatcher` (Z. 13) kann nach dem Strategie-Entwurfsmuster als die Schnittstelle namens „*Strategie*“ interpretiert werden. Sie definiert, welche Voraussetzung an die Schnittstelle gegeben ist. In diesem Fall ist die Voraussetzung, dass es sich um eine Funktion mit dem Rückgabewert `bool` handelt, der als erstes Argument eine Auswahloption – der Parameterbezeichner lautet `choice` – und als zweites Argument eine Menge von Auswahloptionen – der Parameterbezeichner ist `priorChoices` – übergeben wird. Sollte der Parameter `choiceMatcher` gesetzt sein, so tauscht er die standardmäßig genutzte Strategie `defaultChoiceMatcherStrategy` durch die benutzerdefinierte Strategie aus (Z. 46). Beide werden nach dem Strategie-Entwurfsmuster als „*konkrete Strategien*“ bezeichnet. Im Entwurfsmuster gibt es noch den Akteur „*Kontext*“, wobei es sich um die aufrufende Klasse

¹Vgl. 11, S. 373.

handelt, welche die Strategien verwendet. In diesem Fall ist das die „Klasse“ `SelectionCard`. Abbildung 10.5 zeigt das UML-Diagramm der konkreten Implementierung des „Strategie-Entwurfsmusters“ für die „Strategie“ `ChoiceMatcher`. Da sich bei der konkreten Strategie für das Auswahlfeld der „Nebenziele“ um eine anonyme Funktion handelt, wurde sie zum besseren Verständnis im UML-Diagramm „*nebenzieleChoiceMatcherStrategy*“ genannt.

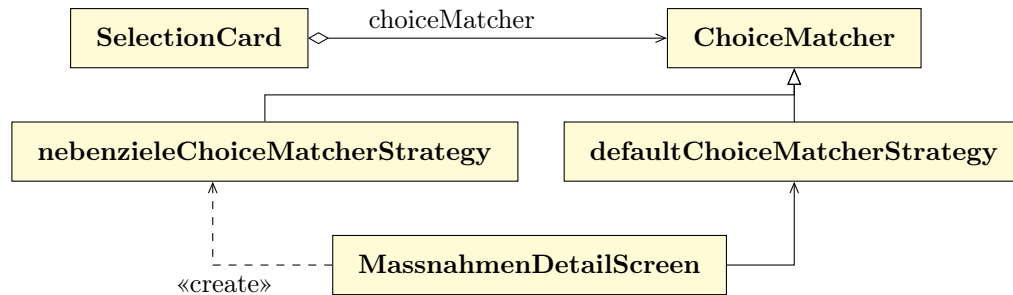


Abbildung 10.5.: UML Diagramme, Quelle: Eigene Abbildung

Im Diagramm ist ebenfalls der „View“ `MassnahmenDetailScreen` enthalten, denn er verwendet die konkrete Strategie `defaultChoiceMatcherStrategy` für die Validierung (Listing 10.2).

```

119 Widget buildSelectionCard<ChoiceType extends Choice>(
120   {required Choices<ChoiceType> allChoices,
121     required BehaviorSubject<ChoiceType?> selectionViewModel,
122     ChoiceMatcher<ChoiceType>? choiceMatcher}) {
123   return FormField(
124     validator: (_) => validateChoices(
125       name: allChoices.name,
126       choices: {
127         if (selectionViewModel.value != null) selectionViewModel.value!
128       },
129       priorChoices: vm.priorChoices.value,
130       choiceMatcher: choiceMatcher ?? defaultChoiceMatcherStrategy),
  
```

Listing 10.2.: XXXX, Quelle: Eigenes Listing, Datei: `Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart`

Sollte nämlich ein Argument für den Parameter `choiceMatcher` übergeben werden (Z. 122), so wird es auch für die Validierung verwendet (Z. 130). Ist das Argument aber nicht gesetzt und damit `null`, so sorgt die „*If-null Expression*“ dafür, dass die `defaultChoiceMatcherStrategy` für die Validierung verwendet wird.

Außerdem erstellt `MassnahmenDetailScreen` die konkrete Strategie „*nebenzieleChoiceMatcherStrategy*“, wie in Listing 10.3 zu sehen ist.

Der Aufruf `buildMultiSelectionCard` wird um die Übergabe einer anonymen Funktion für den Parameter `choiceMatcher` erweitert (Z. 224-239). In der ersten Fallunterscheidung wird überprüft, ob die gewählte Option ein tatsächliches Nebenziel ist (Z. 225). Dies kann über die Getter-Methode `hasRealValue` abgefragt werden. Ist dies nicht der Fall, so handelt es sich um die Auswahloptionen „keine Angabe/Vorgabe“ bzw. „bitte um Unterstützung“, weshalb `true` zurückgegeben werden kann (Z. 237), da diese Auswahloptionen immer erlaubt


```

221 buildMultiSelectionCard<ZielsetzungLandChoice>(
222   allChoices: nebenzielsetzungLandChoices,
223   selectionViewModel: vm.nebenziele,
224   choiceMatcher: (choice, priorChoices) {
225     if (choice.hasRealValue) {
226       if (vm.hauptzielsetzungLand.value == null ||
227         vm.hauptzielsetzungLand.value!
228           .hasNoRealValue) {
229         return false;
230       } else if (choice ==
231         vm.hauptzielsetzungLand.value) {
232         return false;
233       } else {
234         return true;
235       }
236     }
237     return true;
238   },
239 ),

```

Listing 10.3.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

sind. Sollte sich dagegen um ein tatsächliches Nebenziel handeln, so überprüft die nächste Fallunterscheidung, ob das Hauptziel entweder nicht gesetzt ist oder mit einem nicht tatsächlichen Hauptziel belegt ist (Z. 226-228)., Dazu wird die Getter-Methode `hasNoRealValue` benutzt, welche als Gegenteil zu `hasRealValue` fungiert, und dementsprechend `true` zurückgibt wenn die Auswahloption entweder „keine Angabe/Vorgabe“ oder „bitte um Unterstützung“ ist (Z. 226-228). Sollte das Hauptziel keinen tatsächlichen Wert einer Zielsetzung enthalten, dann ist die Wahl eines oder mehrerer Nebenziele nicht sinnvoll. Waren beide zuverigen Bedingungen nicht wahr, so steht bereits fest, dass sowohl das Hauptziel, als auch den Nebenziel gesetzt sind und weder die Option „keine Angabe/Vorgabe“ oder „bitte um Unterstützung“ enthalten. Nun soll eine letzte Fallunterscheidung überprüfen, ob das Nebenziel bereits im Hauptziel gesetzt ist (Z. 230-321). Das ist nicht erlaubt, weshalb `false` zurückgegeben werden soll (Z. 232). Anderenfalls sind alle Bedingungen erfüllt und `true` kann zurückgegeben werden.

An diesem Beispiel wird auch offensichtlich, welchen Nutzen die Generalisierung der Klasse `SelectionCard` hat. Der Typparameter `ZielsetzungLandChoice` wird beim Aufruf der Methode `buildMultiSelectionCard` übergeben (Z. 221). Die Methode übergibt den Typparameter wiederum der Klasse `SelectionCard` (Listing 10.4).

```

131 builder: (field) => SelectionCard<ChoiceType>(

```

Listing 10.4.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Schließlich übergibt die Klasse `SelectionCard` den Typparameter an die Instanzvariable `choiceMatcher` (Listing 10.5).

```
41 ChoiceMatcher<ChoiceType>? choiceMatcher,
```

Listing 10.5.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

Damit handelt es sich also auch bei dem ersten Parameter `choice` der anonymen Funktion, die dem Parameter `choiceMatcher` übergeben wird um den Typ `ZielsetzungLandChoice`. Aus diesem Grund können die Methoden `hasRealValue` (Z. 225) und `hasNoRealValue` (Z. 228) auf dem Objekt `choice` aufrufen werden, obwohl sie nur Teil der Klasse `ZielsetzungLandChoice` aber nicht der Basisklasse `Choice` sind. Ohne Parametrisierung über den Typ müsste das Objekt `choice` in einen anderen Typen umgewandelt werden. Doch nach dieser Typumwandlung könnte ein Laufzeitfehler geschehen, sollte es sich bei dem Objekt tatsächlich nicht um den gewünschten Typ handeln. Durch die Generalisierung der Klassen und die Angabe des Typparameters ist das Vorhandensein des richtigen Typs garantiert und keine Typumwandlung nötig.

Die beiden neuen Methoden sind in Listing 10.6 zu sehen.

```
185 class ZielsetzungLandChoice extends Choice {
186   static final ka = ZielsetzungLandChoice("ka", "keine Angabe/Vorgabe");
187   ...
198   static final contact =
199     ZielsetzungLandChoice("contact", "bitte um Unterstützung");
200
201   bool get hasRealValue => this != ka && this != contact;
202
203   bool get hasNoRealValue => !hasRealValue;
```

Listing 10.6.: XXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/choices/choices.dart](#)

`hasRealValue` vergleicht, ob der aktuelle Wert weder „keine Angabe/Vorgabe“ noch „bitte um Unterstützung“ ist (Z. 201). `hasNoRealValue` ruft dagegen intern `hasRealValue` auf und negiert Wert (Z. 203).

Überall dort, wo zuvor der Ausdruck `choice.conditionMatches(priorChoices)` verwendet wurde, muss nun der Aufruf des `choiceMatcher` erfolgen. So zum Beispiel der Stream, welcher die Validität der Auswahlfelder prüft (Listing 10.7).

```
63 final validityChanged = priorChoices
64   .map((choices) =>
65     selectionViewModel.value.any((c) => !choiceMatcher(c, choices)))
66   .distinct();
```

Listing 10.7.: XXXX, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

Alle Vorkommnisse, die durch den neuen Ausdruck ersetzt werden, sind im Anhang ?? auf den Seiten ?? bis ?? zu finden.

Teil IV

FAZIT

11. Diskussion

11.1. Reevaluation des Zustandsmanagements

Während der Implementierung wurde eine passende Vorgehensweise gesucht, um den Zustand der Applikation zu verwalten und damit die Aktualisierung der Oberfläche auszulösen. Für simple Applikationen empfiehlt Google den integrierten Mechanismus der „*StatefulWidgets*“ und deren Methode „*setState*“ zu verwenden¹. Doch durch die hohe Anzahl der Oberflächenelemente in der finalen Applikation ist diese Vorgehensweise nicht empfehlenswert. Sie setzt das Aktualisieren gesamter Widgets bei Anpassung des Zustandes voraus, was für die Laufzeitgeschwindigkeit die intensivste Belastung darstellt. Stattdessen wurde versucht, einem Mechanismus zu verwenden, der es erlaubt, nur Teile der Oberfläche neuzeichnen, die wirklich eine Aktualisierung benötigen.

Zu diesem Zweck empfiehlt Google das Nutzen des Pakets „*provider*“ der Flutter Community². Dieser Ansatz wurde in der Implementierung ursprünglich verwendet. Das Paket hat den Nachteil, dass für jeden Zustand, der die Aktualisierung eines Teils der Oberfläche bewirken soll, eine neue Klasse erstellt werden muss, die von `ChangeNotifier` erbt. Eine Möglichkeit ist, dass jede dieser Klassen den nötigen Boilerplate-Quellcode enthält, welcher die Oberfläche über die Methode `notifyListeners` benachrichtigt. Eine andere Möglichkeit ist es, für den gleichen Datentyp den benötigten BoilerplateCode in einer eigenen Basisklasse auszulagern und dann von dieser Klasse zu erben wie in Listing zu sehen. `ChoiceChangeNotifier` verwaltet den internen privaten Zustand `_choices` (Z. 3) über die öffentlichen Schnittstellen zum Lesen (Z. 4) und Schreiben (Z. 6-9). Bei Aktualisierung des Wertes erhalten alle Listener eine Benachrichtigung (Z. 8). `LetzterStatusViewModel` erbt dieses Verhalten, doch hat die Klasse darüber hinaus keine Implementierung.

Anschließend muss jeder `ChangeNotifier` als ein `ChangeNotifierProvider` registriert werden (Listing 11.2, Z. 7). Der `MultiProvider` kann genutzt werden, um mehrere Provider in einer Liste zu übergeben. Dort werden auch andere Services wie etwa `MassnahmenFormViewModel` (Z. 3) und `MassnahmenModel` (Z. 6) hinterlegt.

Dann ist der `ChangeNotifier` in dem Widget, welches den Parameter `child` übergeben

¹Vgl. 12.

²Vgl. 24.

```

1 class ChoiceChangeNotifier extends ChangeNotifier {
2     BuiltSet<Choice> _choices = BuiltSet<Choice>();
3
4     BuiltSet<Choice> get choices => _choices;
5
6     set choices(BuiltSet<Choice> choices) {
7         _choices = choices;
8         notifyListeners();
9     }
10 }
11 class LetzterStatusViewModel extends ChoiceChangeNotifier {}

```

Listing 11.1.: Live Template für die Erstellung von built_value Boilerplate-Code in Android Studio, Quelle: JetBrains Marketplace Built Value Snippets Plugin

```

1 MultiProvider(
2   providers: [
3     Provider<MassnahmenFormViewModel>(create: (_) => MassnahmenFormViewModel()),
4     Provider<MassnahmenJsonFile>(create: (_) => MassnahmenJsonFile()),
5     Provider(
6       create: (context) => MassnahmenModel(
7         Provider.of<MassnahmenJsonFile>(context, listen: false)),
8       ChangeNotifierProvider(create: (context) => LetzterStatusViewModel()),
9   ],
10  child: MaterialApp(),
11 )

```

Listing 11.2.: Live Template für die Erstellung von built_value Boilerplate-Code in Android Studio, Quelle: JetBrains Marketplace Built Value Snippets Plugin

wird und darüber hinaus allen Kindern-Elementen dieses Widgets verfügbar. Über einen `Consumer` kann in der Oberfläche auf Änderungen des `ChangeNotifier` reagiert werden (Listing 11.3).

```

1 Consumer<LetzterStatusViewModel>(
2   builder: (context, choiceChangeNotifier, child) {
3   },
4 )

```

Listing 11.3.: Live Template für die Erstellung von built_value Boilerplate-Code in Android Studio, Quelle: JetBrains Marketplace Built Value Snippets Plugin

Doch diese Vorgehensweise bietet im Vergleich zu den von Flutter mitgelieferten „Widgets“ keine Vorteile. Das Äquivalent zum `Consumer` ist das mitgelieferten Widget `StreamBuilder`, welcher mit jeder Art von „Stream“ verwendet werden kann.

Damit unterstützt er ein breiteres Spektrum von Einsatzmöglichkeiten. Beispielsweise kann ein transformierter „Stream“ übergeben werden, wie im Kapitel ?? gezeigt.

Die einzige fehlende Komponente dafür ist ein „Stream“, der den zuletzt übermittelten Wert speichert und den neuen StreamBuilder Elementen übermittelt. Deshalb wurde sich für das Package „rx.dart“ entschieden, welches genau dieses Verhalten mit dem „BehaviorSubject“ abdeckt. Durch dessen Verwendung kann sowohl auf das Registrieren des `ChangeNotifierProvider` verzichtet werden und es muss keine weitere Klasse für die ein-

zelenen beobachtbaren Objekte erstellt werden.

Auch der `MultiProvider` erscheint auf den ersten Blick als sehr nützlich. Doch das Anbieten der Services durch ein eigens implementiertes `InheritedWidget` erlaubt einen Zugriff, der kürzer und expliziter ist. Durch die Umstellung konnte der Zugriff auf das ViewModel mithilfe des Ausdrucks `Provider.of<MassnahmenFormViewModel>(context, listen: false)` durch `AppState.of(context).viewModel` ersetzt werden.

Eine ganz ähnliche, wenn auch deutlich kompliziertere Variante dieser Vorgehensweise, wurde auf der Google I/O 2018 von Filip Hracek und Matt Sulliivan vorgestellt. Doch anstatt lediglich das `BehaviorSubject` für das ViewModel zu verwenden, sorgte die Präsentation durch den zusätzlichen – jedoch überflüssigen – Einsatz zwei weiterer Stream-Klassen für schwereres Verständnis (Listing 11.4)³.

```

1  class CartBloc{
2      final _cart = Cart();
3
4      Sink<Product> get addition => _additionalController.sink;
5
6      final _additionController = StreamController<Product>();
7
8      Stream<int> get itemCount => _itemCountSubject.stream;
9
10     final _itemCountSubject = BehaviorSubject<int>();
11
12     CartBloc(){
13         _additionController.stream.listen(_handle);
14     }
15
16     void _handle(Product product){
17         _cart.add(product);
18         _itemCountSubject.add(_cart.itemCount);
19     }
20 }

```

Listing 11.4.: Die Klasse CartBloc, Quelle: [14] TC: 27:37 ⁴

Obwohl das `BehaviorSubject` die Funktionsweise des ViewModels bereits löst, wurde ein Objekt des Typs `Sink` verwendet, um Ereignisse von dem View an das ViewModel senden zu können (Z. 4). `StreamController` verwendet. Ein Sink implementiert jedoch ausschließlich Methoden zum Hinzufügen von Ereignissen Punkt um den Stream zu lesen, wird ein dazugehöriger `StreamController` erstellt (Z. 6). Er hat im Gegensatz zum `Sink` auch lesenden Zugriff auf die Ereignisse. Sobald ein Ereignis eintrifft, so wird es dem Model `_cart` hinzugefügt (Z. 17). Es existiert außerdem ein weiterer `Stream itemCount` (Z. 8) welcher lediglich die transitive Eigenschaft der Anzahl der hinzugefügten Elemente darstellt 18. Er nutzt das `BehaviorSubject` 10, verwendet allerdings keine der bedeutsamen Methoden. Es könnte genauso gut durch einen weiteren `StreamController` ersetzt werden.

Der gesamte Quellcode kann stark vereinfacht werden (Listing 11.5).

³14, TC: 27:37.

```

1 class CartBloc{
2   final _cart = BehaviorSubject<Cart>(seedValue: Cart());
3
4   addProduct(Product product) => _cart.value = _cart.value..add(product);
5
6   Stream<int> get itemCount => _cart.map((cart) => cart.itemCount);
7 }

```

Listing 11.5.: Die vereinfachte Klasse CartBloc, Quelle: Eigenes Listing

Durch Einsatz der für das `BehaviorSubject` einzigartigen Getter-Methode `value` kann dem „Stream“ ein neues Objekt hinzugefügt werden, wodurch er gleichzeitig ein neues Ereignis sendet (Z. 4). Die Zuweisung hat zwar ansonsten keinen Zweck, da das Objekt vor und nach der Zuweisung das gleiche ist, denn es handelt sich um einen Referenztyp und nicht um einen Werttyp. Die Erstellung weiterer `StreamController` zum Senden der transitiven Eigenschaft `itemCount` ist nicht nötig. Sendet das `BehaviorSubject` `_cart` ein neues Event (Z. 4), so wird auch die Methode `map` ausgelöst und ein transformiertes Eigenschaft gesendet (Z. 6).

Durch eine Anleitung mit diesem Ergebnis könnten gegebenenfalls weitere Entwickler das „*BloC-Pattern*“ dem Paket „*provider*“ vorziehen.

12. Anzeige von fehlerhaften Teilkomponenten der Bedingungen von deaktivierten Auswahloptionen

Einen Wunschkriterium für die Formularapplikation war es, bei der Auswahl von deaktivierten Optionen einen Hinweise zu erhalten, warum diese deaktiviert ist.

In Kapitel ?? ist die Umsetzung der Deaktivierung von Optionen beschrieben. Eine Funktion zur Überprüfung der Bedingung einer Optionen wird der Option bei dessen Erstellung im Konstruktor übergeben. Sie wird bei Überprüfung der Kompatibilität der Auswahloption mit den restlichen im Formular ausgewählten Optionen ausgeführt. Die Konjunktion, Disjunktion und Negation wird mit den Operatoren für das logische Und und das logische Oder sowie das logische Nicht umgesetzt. Doch auf diese Art und Weise ist es nicht möglich, herauszufinden, welche der einzelnen Abfragen zu einem Fehler führte. Auf den Inhalt der Funktion kann zur Laufzeit nicht zugegriffen werden. Die Einzelkomponenten der Bedingung sind damit also nicht bekannt. Es ist daher nur möglich, auf die Komponenten der Bedingung zuzugreifen, wenn die gesamte Bedingung als eine Datenstruktur abgelegt ist. Diese Datenstruktur muss die Konjunktion, Disjunktion und Negation unterstützen.

Die Konzeption und Implementierung einer solchen Datenstruktur und des dazugehörige Algorithmus zur Identifizierung der inkompatiblen Komponenten bedarf einer intensiven wissenschaftlichen Recherche und Ausarbeitung. Als Wunschkriterien steht diese Funktion somit nicht im Kosten-Nutzen-Verhältnis, weshalb sich gegen die Ausarbeitung in dieser wissenschaftlichen Arbeit entschieden wurde.

13. Schlussfolgerung-und-Ausblick

In dieser Arbeit wurde gezeigt, dass das Hauptproblem der Formular Anwendung mit Hilfe von Funktionsobjekten und logischen Operatoren gelöst werden konnte.

Auch die Aktualisierung der sich tatsächlich ändernden Elemente in der Oberfläche wurde umgesetzt. In jedem Fall war die deklarative und reaktive Programmierung der Oberfläche eine Erleichterung und Voraussetzung dafür. Die Implementierung hätte auch mit „*React Native*“ stattfinden können, da es ebenso einen deklaratives Oberflächen Framework ist. Die Stream Transformationen aus der Kern-Bibliothek von Dart und aus „*RxDart*“ haben ihre Äquivalente in der Bibliothek „*RxJS*“. <https://www.learnrxjs.io/learn-rxjs/operators/filtering/distinct>

Die Wahl von Flutter für die Entwicklung war trotzdem aus den folgenden Gründen eine gute Entscheidung:

Die gesichteten Anleitungen für die Einarbeitung in das automatisierte Testen ebneten eine vollumfängliche und zielgerichtete Einarbeitung. Keine weiteren Quellen von Drittanbietern mussten genutzt werden, um die im Rahmen dieser Masterarbeit entstandenen „*Unit-*“ und „*Integrationstests*“ zu entwickeln. Lediglich die initialen Probleme bei der Generierung von Mocks im Ordner für die Integrationstest stoppten die Entwicklung für einen Moment.

Hätte die Umsetzung in the React Native stattgefunden, so hätte die Einarbeitung in die Entwicklung von „*Unit-*“ und „*Integrationstest*“ eventuell einen höheren Aufwand bedeutet, da die Dokumentation auf den unterschiedlichen Web-Portal in der Drittanbieter verstreut ist.

Auch die Rezepte im Flutter-Kochbuch boten die benötigten Funktionalitäten wie die Formularvalidierung, die Navigation über Routen

Allerdings fällt die Wahl für das angemessene Zustandsmanagement für einen Anfänger in der deklarativen Programmierung nicht leicht. Die Empfehlung von Google das Paket „*Provider*“ zu nutzen führte zu Schwierigkeiten, wie in Sektion 11.1 beschrieben. Das ursprünglich von Google beworbener „*Bloc-pattern*“, welches bei der „*Flutter*“-Community weniger beliebt ist, war am Ende die angemessene Technologie. Es fehlte aber die Dokumentation darüber, wie es richtig eingesetzt wird. Die Erkenntnisse, die im Rahmen dieser Masterarbeit bezüglich der reibungslosen Implementierung des Zustandsmanagements mit

„*RxDart*“ gesammelt wurden, sollen in Zukunft mit der „*Flutter*“-Community geteilt werden.

Das Wunsch Kriterium, den Benutzer auch die fehlerhafte Auswahl anzuzeigen, die verhindert, eine spezielle Option zu wählen, konnte nicht umgesetzt werden. Vor dem Hintergrund der für diese Arbeit festgelegten Ziele und der Komplexität des Problems wurde sich gegen die Konzeption und Implementierung entschieden. An den bisherigen Erkenntnissen soll jedoch weiter gearbeitet werden. Nutzerumfragen sollen darüber hinaus zeigen, in welcher Art und Weise eine solche Fehlermeldung präsentiert werden könnte.

Literatur

- [1] Adobe Inc. *FAQ / PhoneGap Docs*. Aug. 2016. URL: <https://web.archive.org/web/20200806024626/http://docs.phonegap.com/phonegap-build/faq/>.
- [2] Adobe Inc. *Update for Customers Using PhoneGap and PhoneGap Build*. Aug. 2020. URL: <https://web.archive.org/web/20200811121213/https://blog.phonegap.com/update-for-customers-using-phonegap-and-phonegap-build-cc701c77502c?gi=df435eca31bb>.
- [3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003. URL: <https://archive.org/details/extremeprogrammi00beck/page/9>.
- [4] Matan Borenkraout. *Native Testing Library Introduction / Testing Library Docs*. Nov. 2020. URL: <https://web.archive.org/web/20210128142719/https://testing-library.com/docs/react-native-testing-library/intro/>.
- [5] Brandon Bray. *Async in 4.5: Worth the Await*. Apr. 2012. URL: <https://web.archive.org/web/20210702135551/https://devblogs.microsoft.com/dotnet/async-in-4-5-worth-the-await/> (besucht am 09.08.2021).
- [6] *Does redux-form work with React Native?* Juni 2021. URL: <https://web.archive.org/web/20210602234346/https://redux-form.com/7.3.0/docs/faq/reactnative.md/>.
- [7] Facebook Inc. *The React Native Ecosystem*. Juni 2021. URL: <https://web.archive.org/web/20210602191504/https://github.com/facebook/react-native/blob/d48f7ba748a905818e8c64fe70fe5b24aa098b05/ECOSYSTEM.md>.
- [8] *<Formik /> / Formik Docs API*. Juni 2021. URL: https://web.archive.org/web/20210409184616if_/https://formik.org/docs/api/formik.
- [9] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Jan. 2004. URL: <http://web.archive.org/web/20210707041912/https://martinfowler.com/articles/injection.html>.
- [10] Martin Fowler. *InversionOfControl*. Juni 2005. URL: <http://web.archive.org/web/20050628234825/https://martinfowler.com/bliki/InversionOfControl.html>.
- [11] Erich Gamma u. a. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, 2009.
- [12] Google LLC. *Adding interactivity to your Flutter app*. URL: <https://web.archive.org/web/20210603051020/https://flutter.dev/docs/development/ui/interactive> (besucht am 16.08.2021).

- [13] Google LLC. *Build a form with validation*. Juni 2021. URL: <https://web.archive.org/web/20210122020924/https://flutter.dev/docs/cookbook/forms/validation>.
- [14] Google LLC. *Build reactive mobile apps with Flutter (Google I/O '18)*. Juni 2021. URL: <https://youtu.be/RS36gBEp80I?t=1657> (besucht am 10.05.2018).
- [15] Google LLC. *Dart - Effective Dart - Style - PREFER using _, __, etc. for unused callback parameters*. URL: https://web.archive.org/web/20210728114518/https://dart.dev/guides/language/effective-dart/style#prefer-using-_-__-etc-for-unused-callback-parameters (besucht am 08.08.2021).
- [16] Google LLC. *Dart - Language tour - spread operator*. Juli 2021. URL: <https://web.archive.org/web/20210625070139/https://dart.dev/guides/language/language-tour#spread-operator> (besucht am 08.07.2021).
- [17] Google LLC. *Dart Programming Language Specification 5th edition*. Apr. 2021. URL: <https://web.archive.org/web/20210702071617/https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf>.
- [18] Google LLC. *Dart: The platforms*. URL: <https://web.archive.org/web/20210719180726/https://dart.dev/overview#platform> (besucht am 09.08.2021).
- [19] Google LLC. *Desktop support for Flutter*. Juni 2021. URL: <https://web.archive.org/web/20210531034514/http://flutter.dev/desktop/>.
- [20] Google LLC. *Flutter - Beautiful native apps in record time*. Juni 2021. URL: <https://web.archive.org/web/20210630233338/https://flutter.dev/>.
- [21] Google LLC. *Flutter - Introduction to widgets*. Juni 2021. URL: <http://web.archive.org/web/20210603081649/https://flutter.dev/docs/development/ui/widgets-intro>.
- [22] Google LLC. *Forms | Flutter Docs Cookbook*. Juni 2021. URL: <https://web.archive.org/web/20201102003629/https://flutter.dev/docs/cookbook/forms>.
- [23] Google LLC. *Häufig gestellte Fragen zu Google Trends-Daten - Google Trends-Hilfe*. Mai 2021. URL: <https://support.google.com/trends/answer/4365533>.
- [24] Google LLC. *Provider - A recommended approach*. URL: <https://web.archive.org/web/20210729143240/https://flutter.dev/docs/development/data-and-backend/state-mgmt/options#provider> (besucht am 16.08.2021).
- [25] Google LLC. *Web support for Flutter*. Mai 2021. URL: <http://web.archive.org/web/20210506012158/https://flutter.dev/web>.
- [26] James Gosling u. a. *The Java® Language Specification Java SE 16 Edition*. Feb. 2021. URL: <https://web.archive.org/web/20210514051033/https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf>.
- [27] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. Okt. 2005. URL: <https://web.archive.org/web/20101103111603/http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>.

- [28] JetBrains s.r.o. *Kotlin - High-order functions and lambdas - Underscore for unused variables*. URL: http://web.archive.org/web/20210331062820if_/https://kotlinlang.org/docs/lambdas.html#underscore-for-unused-variables (besucht am 08.08.2021).
- [29] Ralph E Johnson und Brian Foote. „*Designing reusable classes*“. In: *Journal of object-oriented programming* 1.2 (1988), S. 22–35.
- [30] Max Lynch. *The Last Word on Cordova and PhoneGap*. 2014. URL: <https://web.archive.org/web/20210413012559/https://blog.ionicframework.com/what-is-cordova-phonegap/>.
- [31] MDN contributors. *Promise - JavaScript | MDN*. Mai 2021. URL: https://web.archive.org/web/20210516053958/https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise (besucht am 06.08.2021).
- [32] MDN contributors. *async function - JavaScript | MDN*. Juni 2021. URL: https://web.archive.org/web/20210608034309/https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/async_function (besucht am 06.08.2021).
- [33] *React Native | Formik Docs*. Juni 2021. URL: https://web.archive.org/web/20210507005917if_/https://formik.org/docs/guides/react-native.
- [34] *React Native | React Hook Form - Get Started*. Juni 2021. URL: https://web.archive.org/web/20210523042601if_/https://react-hook-form.com/get-started/.
- [35] Joel Spolsky. „*How Hard Could It Be?: The Unproven Path*“. In: *inc.com* (Nov. 2008). URL: <http://web.archive.org/web/20081108094045/http://www.inc.com/magazine/20081101/how-hard-could-it-be-the-unproven-path.html>.
- [36] Stack Exchange, Inc. *Stack Overflow Insights - Developer Hiring, Marketing, and User Research*. Mai 2021. URL: <https://insights.stackoverflow.com/survey/>.
- [37] *reduxForm | Redux Form - API*. Juni 2021. URL: <https://web.archive.org/web/20210506221401/https://redux-form.com/7.4.2/docs/api/reduxform.md/#-code-validate-values-object-props-object-gt-errors-object-code-optional->.
- [38] *register | React Hook Form - API*. Juni 2021. URL: <https://web.archive.org/web/20210406032209/https://react-hook-form.com/api/useform/register>.

Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Masterarbeit „*Entwicklung einer Formularanwendung mit Kompatibilitätsvalidierung der Einfach- und Mehrfachauswahl-Eingabefelder*“ selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Wernigerode, den 01.09.2021

Alexander Johr