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 anzubeten, 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 **??** 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 **??** mit den Grundlagen des Frameworks und der zugrunde liegenden Programmiersprache Dart.

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

In Kapitel **??** erhält die Anwendung die grundlegende Funktion, Felder zu validieren. Kapitel **??** 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 **??** hinzugefügt.

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

Kapitel 2 setzt sich mit den Erkenntnissen auseinander, die während der Entwicklung der Anwendung gesammelt wurden. Kapitel 4 bewertet die Erkenntnisse, ergänzt sie um einen Ausblick und vergleicht die Ergebnisse der Entwicklung mit den Anforderungen.

Teil II

Vorbereitung

Teil III

Fazit

2. Diskussion

2.1. 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 verwenden[1]. 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 neuzuzeichen, die wirklich eine Aktualisierung benötigen.

Zu diesem Zweck empfiehlt Google das Nutzen des Pakets *„provider“* der Flutter Community[2]. 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 **??**, 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

[1] Vgl. 1.
[2] Vgl. 3.

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

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 einzelnen 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 schweres Verständnis (Listing **??**)[3].

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 Sind 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.

[3] 2, TC: 27:37.

Der gesamte Quellcode kann stark vereinfacht werden (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.

3. 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.

4. 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/filterir`

Die Wahl von Flutter für die Entwicklung war trotzdessen 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 2.1 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] 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).

[2] Google LLC. *Build reactive mobile apps with Flutter (Google I/O '18)*. Juni 2021. URL: https://youtu.be/RS36gBEp8OI?t=1657 (besucht am 10. 05. 2018).

[3] 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).

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