Chapter 4

Application Design and Implementation

This thesis does not use the **engineering method** (see Ertas and Jones 1996) as a scientific approach for evaluating the hypotheses (declared in Section 1.3). Thus application design and implementation will be explored from a perspective of a necessary requirement to conduct both the performance and UX comparison (Chapter 3) in this chapter.
Naturally, software implementation complexity may impede performance if unscalable algorithms or memory-inefficient datastructures are chosen. Therefore, the goal from a development perspective is to implement the Flutter clone as closely as possible to the baseline application in order to facilitate a fair comparison between both apps.
Further, it is important to note that the clone app was built with Flutter version 1.22.6 and Dart 2.10.5 whereas the baseline app is built with iOS 13.0 UIKit and Swift 5.0.
Firstly, general implementation differences resulting from Flutter's declarative nature are explained and contrasted to UIKit's imperative approach in Section 4.1. Secondly, **constant widget precompilation** is explained as a performance optimization of the Flutter clone (see Section 4.2). Finally, the usage of the Cupertino package and its UI components is inspected (Section 4.3).

4.1 Implications of Flutter's Declarative UI Paradigm

Flutter and UIKit are distinct technologies for building UI as they utilize the declarative and imperative programming paradigm respectively. These paradigms fundamentally determine how

UI code is written when using the corresponding framework.

4.1.1 UIKit's imperative UI Programming Paradigm

The imperative UI programming paradigm is characterized by explicit instructions to the framework to achieve a desired user interface.
For example, centering a piece of text on screen in UIKit takes a considerable amount of instructions as can be seen in Listing 19. Firstly a `UIViewController` is subclassed which manages the entire view hierarchy of the particular screen (Apple Inc. 2021c). Furthermore, a `UILabel` - *"[...] a view that displays one or more lines of informational text"* (Apple Inc. (2021b)) - is created via a closure (see Apple Inc. 2021a) for that parent view controller. Inside the closure the `text` property is explicitly set after initialization and the label's `translatesAutoresizingMaskIntoConstraints` property is set to `false` in order for the label to be positioned and sized programmatically. Once the view hierarchy for the view controller is loaded into memory, the `viewDidLoad` function is called by the system (see Apple Inc. 2021d). Subsequently, the label is added as a subview and centered horizontally and vertically in the view controller's view based on **Auto Layout** (Apple Inc. 2016) constraints.

```swift
import UIKit

class InitialViewController: UIViewController {
    private let customTextLabel: UILabel = {
        let label = UILabel()
        label.text = "Custom Text"
        label.translatesAutoresizingMaskIntoConstraints = false
        return label
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = UIColor.white
        view.addSubview(customTextLabel)
        customTextLabel.centerXAnchor.constraint(equalTo: view.
            centerXAnchor).isActive = true
        customTextLabel.centerYAnchor.constraint(equalTo: view.
            centerYAnchor).isActive = true
    }
}
```

Listing 4.1: iOS UIKit Example of Centering Text on a Screen

4.1.2 Flutter's Declarative UI Programming Paradigm

Contrarily, Flutter's code structure can be more directly mapped to actual user interface elements with its declarative widget composition paradigm.

The same example of centering a piece of text on screen can be written declaratively with Flutter as shown in Listing 8. A widget representing the screen is created by subclassing `StatelessWidget` (see Google Inc. 2021h). Unlike `StatefulWidgets` (see Google Inc. 2021g), `StatelessWidgets` do not hold any mutable state and can't change during runtime based on state changes. `StatelessWidgets` have a required `build` method which return any widget child relationships. The method is called once by the system to mount the widget's subtree structure into the global widget tree (explained in 2.2.2). Concretely, a `Center` with a `Text` is returned in order to display centered text in the UI.

4.1.3 Advantages and Disadvantages of the Declarative Paradigm

The advantages of using the declarative approach include the entire description of intent in one place and abstraction of details into a clean interface outsourcing implementation complexity to the framework (see Google Inc. 2021e).

However, while Flutter's declarative apprach may be more intuitive for the human mind, it might also introduce a slight performance overhead. This occurs due to the widget subtree diffing calcualation performed for each state change in `StatefulWidget` subtrees (explained in Subsection 2.2.2).

```
import "package:flutter/widgets.dart";
class InitialPage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Center(child: Text("Custom Text"));
    }
}
```

Listing 4.2: Flutter Example of Centering Text on a Screen

4.1.4 Partial View Rebuilding

In order to minimize the amount of computation coordinated to the framework, the Flutter clone makes use of **Partial View Rebuilds**. Instead diffing the entire view tree, the framework only recomputes parts of the UI that are actually affected by the state change. This can be achieved by pushing the state to the leaves of the tree (see Google Inc. 2021f). For example, each Posting has an associated countdown label UI element (see ...) which needs to calculate its remaining time based on an end date received from the *Kickdown* API every second. By associating a state object - holding the remaining time - with the **CountdownLabel** widget itself rather than the entire or unnecessarily large parts of the view hierarchy, Flutter's internal diffing algorithm runs on a substantially smaller tree data structure.

4.2 Constant Widget Precompilation

As part of general performance optimization of the Flutter clone, Dart's `const` constructor is used where possible throughout the codebase.
Marking instances with this keyword makes them into immutable compile-time constants. Thereby, `const` widgets are only built once if part of a `StatefulWidget` reducing build cycle times leading to higher frame rates (FPS).
Furthermore, instances are **canonicalized** making multiple instances the same underlying instance in order to save memory (see **DartConstDocumentation**).
Additionally, this reduces instance tracking and deallocation work for the **Dart garbage collector** (see Google Inc. 2021d).
The process of marking appropriate instances as `const` was facilitated by using the `prefer_const_constructor` (see Google Inc. 2021a) linter rule for the Dart Static Analyzer (Google Inc. 2021c).

4.3 Flutter UI Component Usage

As mentioned in Section 2.2, Flutter uses the Cupertino package (Google Inc. 2021b) to resemble native iOS UI components from the HIG. These out-of-the-box components are used where possible as a best practice to map elements of the original app to the Flutter clone. Application elements such as the tab and navigation bar as well as switch controls and page transitions are implemented using Cupertino widgets.
The **More Screen**'s table view of the application could not be implemented as easily as in the baseline application.

4.3.1 More Section Custom Widget Composition Implementation

The baseline iOS application uses a **UICollectionView** component of the UIKit framework in order to build out the More screen. The collection view component offers the ability to easily create rows and sections backed by data source. Furthermore, insets, dividers, detail icons as well as tap interaction UI feedback are easily implemented. Contrarily, using Flutter there is no equivalent widget for creating the desired UI outcome and a custom implementation is written instead. A simplified version of the code can be seen in Listing 33. A **SliverList** which is a scrollable UI area that places children in a linear array is used to layout the individual cells. Each cell is represented by a **CupertinoListTile** which is responsible for adding text and a trailing widget to each cell as well as adding appropriate dividers and section insets. Furthermore, it implements a custom `GestureDetector` which highlights the cell when the user taps down on the cell and triggers an anonymous callback.

```
        SliverList(
            delegate: SliverChildBuilderDelegate(
                (BuildContext context, int index) {
                    switch (index) {
                    case 0:
                        return CupertinoListTile(
                            title: 'Mein Account',
                            trailing: Button03(
                                text: 'Anmelden',
                                onPressed: () {
                                    model.onTapLogin();
                                },
                            ),
                            onTap: null,
                            isStartOfSection: true,
                            isEndOfSection: true,
                        );
                    case 1:
                        return CupertinoListTile(
                            title: 'About Kickdown',
                            trailing: DetailIcon(),
                            onTap: () async => model.
                               onTapAboutKickdownTile(),
                            isStartOfSection: true,
                            isEndOfSection: false,
                        );
                    case 2:
                        ...
                    }
                },
                childCount: 6,
            ),
        );
```

Listing 4.3: Simplified Flutter More View Implementation

@Jan: Ist das so ausreichend vom Umfang des Implementation chapters? Ich könnte hier gefühlt unendlich viel schreiben. Aber eigentlich will ich nur dem Leser zeigen, dass die Apps in ihrer Implementierung vergleichbar sind für die Vergleichsstudie.

Bibliography

Apple Inc. 2016. *Understanding Auto Layout.* https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html.

———. 2021a. *Swift Language Guide Closures.* https://docs.swift.org/swift-book/LanguageGuide/Closures.html.

———. 2021b. *UILabel Documentation.* https://developer.apple.com/documentation/uikit/uilabel.

———. 2021c. *UIViewController Documentation Website.* https://developer.apple.com/documentation/uikit/uiviewcontroller?language=objc.

———. 2021d. *viewDidLoad Documentation.* https://developer.apple.com/documentation/uikit/uiviewcontroller/1621495-viewdidload.

Ertas, Atila, and Jesse Jones. 1996. *The Engineering Design Process.* John Wiley / Sons Inc.

Google Inc. 2021a. *const Constructor Dart Linter Rule Documentation.* (https://dart-lang.github.io/linter/lints/prefer_const_constructors.html.

———. 2021b. *Cupertino Package Documentation.* https://flutter.dev/docs/development/ui/widgets/cupertino.

———. 2021c. *Dart Static Code Analysis.* https://dart.dev/guides/language/analysis-options#enabling-linter-rules.

Google Inc. 2021d. *Dart VM Terms Glossary Documentation.* https://flutter.dev/docs/development/tools/devtools/memory#:~:text=In%20DevTools,%20you%20can%20perform,by%20clicking%20the%20GC%20button.&text=Dart%20objects%20that%20are%20dynamically,or%20when%20the%20application%20terminates..

———. 2021e. *Introduction to Declarative UI Programming Website.* https://flutter.dev/docs/get-started/flutter-for/declarative.

———. 2021f. *Stateful Widget Performance Considerations.* https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html#performance-considerations.

———. 2021g. *StatefulWidget Documentation.* https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html.

———. 2021h. *Stateless Widget Documentation.* https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html.