*Human Factors in Accessibility and Assistive Technology,* Vol. 37, 2022, 45–52
https://doi.org/10.54941/ahfe1001641

# Verifying Screen Reader Accessibility of Apps Developed Using Google Flutter

**Alireza Darvishy**

Zurich University of Applied Sciences 8400 Winterthur, Switzerland

## ABSTRACT

This paper presents the results of a study to verify whether the Google UI framework Flutter can create accessible apps for iOS and Android platforms simultaneously. Flutter provides mechanisms such as semantic classes to optimise accessibility during app development. Optimising accessibility for mobile apps and especially for screen readers is a major challenge for many app developers. One key reason for this is that optimisations should ideally always be made for both Apple and Android. A possible solution is offered by the UI framework Flutter, which aims to enable development in only one codebase. This means that accessibility optimisations made in the Flutter codebase should take effect in both Apple and Android platforms simultaneously as well as with their respective screen readers, so that users are provided with a consistent and accessible user experience, regardless of the platform chosen. The purpose of this study was to test this hypothesis. To this end, a sample app was developed using Flutter and a usability test was conducted with six visually impaired screen reader users. Based on the initial test results, the app was then optimised in terms of accessibility using Flutter's semantics classes, and then tested and evaluated again with the same test group. The results showed that some user interface elements were still not accessible. In order to overcome these accessibility issues, workarounds such as writing additional code for each specific platform were implemented, before a final usability test showed that the sample app was fully accessible.

**Keywords:** Google flutter, Accessibility, Visual impairment, Screen reader, Accessible mobile

## INTRODUCTION

More than one billion people worldwide are affected by a disability. This number represents approximately 15% of all humanity, as stated by the World Bank on the International Day of Persons with Disabilities, December 3rd, 2019 (Fu et al., 2019). Nevertheless, accessibility is often lacking in many applications.

Among mobile apps in particular, accessibility via so-called "screen readers" is essential for all visually impaired and blind users, which according to statistics from the World Health Organization were estimated at 285 million in 2010 (Pascolini and Mariotti, 2012). A WebAIM's 2019 User Survey, which surveyed 1224 people with disabilities, revealed that 87.6% of respondents used a screen reader, of which 94.5% were either blind (76.0%) or visually impaired (18.5%).

 

The two most commonly used mobile screen readers are VoiceOver for Apple (71.2%) and TalkBack for Android (33.0%) (WebAIM, 2017). For app developers, optimising accessibility for both platforms often requires significant time and financial resources, which is one reason that accessibility on mobile apps is often insufficient. One possible solution is the open-source UI framework Flutter developed by Google, which is used as a so-called "hybrid" for Android and iOS app development. The goal in this case would be accessibility optimisations that simultaneously take effect on both platforms and their native screen readers, such that users are offered a consistent and user-friendly experience, regardless of the chosen platform. The feasibility of this goal was tested using a sample optimisation of an application developed in Flutter. Through this study, the possibilities and limitations of Flutter for accessible optimisation in app development could be examined and evaluated.

## METHODS

The study was conducted in three steps. The first step was the development of a sample application using Flutter for iOS and Android. The sample app was initially developed in such a way that it was not specially optimised for accessibility. This made it possible for the screen reader test group to identify and analyse potential accessibility issues in the standard app elements before optimisation. The second step was to optimise the app's accessibility as much as possible for both platforms. For this purpose, the Semantics class was used, which is an integrated class in Flutter that enables the developer to give a meaning to UI elements so that they can be interpreted by screen readers and read out in a meaningful way (Flutter Code 1, 2019). Finally, the app was tested again among the same test group after optimisation.

### Application

#### Initial Development

The sample application to be tested was created using the CodeLabs "Build a Cupertino app with Flutter" provided by Google Developers (Zakhour and Morgan, 2021). This made it possible to quickly and efficiently develop sample application that represents a realistic mock-up of an online shopping cart and has enough testable, different UI elements. The simple shop app consists of three tabs: Products, Search and Cart. The Products tab displays the store, which is equipped with a selection of available products including their name, price, image and a floating action button to add the product to the shopping cart (Figure 1, left). The Search Tab offers the user the possibility to search for a specific product in this list (Figure 1, centre). The Cart tab shows the final shopping cart. Here the user can enter his name, e-mail address, location and delivery time. Furthermore, the cart tab gives the user an overview of all products added to the shopping cart as well as the total price including taxes.

#### Usability Testing

To test the app, several tasks were developed in which the test persons had to perform certain actions using a screen reader. This ensured that all existing UI

elements were tested for their accessibility. The execution of these tasks was documented in real time so that the test persons could give direct feedback on which UI elements and functions should be supplemented and adapted. Based on this feedback, the app was optimised in the second step with the help of the actions presented by Flutter in the Optimisation chapter, and then tested again with the same tasks and the same test group. The following tasks were defined for the following UI elements:

- Product Item: Add the fourth product item in the product list on the Products tab to the shopping cart twice. Then add the first product item to the basket once.
- Search Field: Search in Search Field in the Search Tab for the product "Vagabond sack" and add it to the shopping cart.
- Input Text Fields: Add the input text fields Name, Email and Location on the Cart tab.
- Date Picker: Use the Date Picker on the Cart tab to select the date, which is one week from today.
- Shopping Cart Item: Check in the Shopping Cart List on the Cart Tab if all previously selected products are present and find out what the total of all products is.

### Optimisation

Based on the initial usability tests, the optimisation of the UI elements with regards to accessibility included the following actions:

1. Adaptation of the hints
2. Addition of a feedback after the activation of an element
3. Replacing elements that do not fulfill essential basic functions of the screen reader.
4. Exclusion of elements that are merely decorative
5. Adding additional semantic texts for a better understanding of the app.
6. Improving reading flow by adapting the semantic text.

These optimisations are discussed in more detail in the subchapters below. The focus is on the problematic, complex optimisations for which workarounds had to be found. These are presented at the beginning in subchapters 2.2.1 - 2.2.3. This is followed in chapters 2.2.4 - 2.2.6 by the simpler optimisations that could be carried out without problems.

#### Adaptation of the Hints

By customising the hints, the screen reader should provide users with more detailed information about what will happen when an item is activated. For example, when selecting a product item, the screen reader should tell users that the item would be added to the shopping cart if activated. Using the Flutter callback functions onTapHint and onLongPressHint, for example, the standard sentence "Double tap to activate" could be replaced with the more user-friendly sentence "Double tap to add to cart". However, although these callback functions successfully override the default hints on Android, they are ignored on iOS. This is described in the Flutter documentation of the

SemanticsHintOverrides class (Flutter Code 2, 2019). To solve this problem, the hint property of the Semantics class was used (Flutter Code 3, 2019), which, like the onTapHint function, allows the default hint to be overridden. Here too, there are differences in the interpretation of the flutter code if the selected element is an activatable UI element, as in the given example. For example, if the string passed for the hint is "Double tap to add to cart", iOS only reads out the passed string as a hint. Android, on the other hand, adds the standard phrase "Double tap to activate" in addition to the string passed. Thus, instead of "Double tap to add to cart", the Android screen reader reads out the sentence "Double tap to add to cart, Double tap to activate". However, since this standard sentence should be overwritten with the more detailed sentence, the hint property for Android was unusable in the sample app. As a workaround, different behaviours had to be defined in the flutter code for Android and iOS. The onTapHint callback function was used for Android, while the hint property was defined as an alternative for iOS. Here, an additional if-else statement had to be used to distinguish which platform is used, so that the hint property can be deactivated when using Android and the onTapHint function when using iOS. When adapting the hints, it became clear that for such optimisations it is necessary to distinguish between the two platforms and to define separate behaviours in order to enable the best possible accessibility.

**Adding Feedback After Activating a UI Element**

Adding feedback after activating a UI element allows screen reader users to update their mental map of the screen so that they are aware of changes on the screen and in the app. Since the feedback on adding and removing a product item to or from the shopping cart was missing in the sample app, this had to be added according to the Accessibility Guidelines. For this purpose, the announce function of the SemanticsService class was used (Flutter Code 4, 2019), which returns a semantic announcement as feedback on a UI state change. Here, too, a similar problem as with the adaptation of the hints could be observed. The SemanticsService class, which accesses the platform-specific accessibility services, is interpreted differently by VoiceOver than by TalkBack. While iOS retains its own specific behaviour and reads out the entire semantic of the element, Android lets itself be overwritten with the passed string message as announcement and reads it out when the element is activated. As a result, both platforms react differently, despite the same codebase, so that a consistent user experience between the two platforms is impaired.

**Replacing Elements That Do Not Fulfill Essential Basic Functions of the Screen Reader**

In addition to the problems with hints and adding feedback, the interviews revealed that basic UI elements such as TextFields have various problems for iOS. For example, the function to switch between letters by swiping up and down, as known from the native Apple UITextViews, is missing. In addition, the focus does not return to the TextField after the keyboard has ended but remains at the point where the keyboard ended. These inconsistencies with the usual behaviour caused problems for the interviewees in operating and

orienting themselves in the app. In order to solve this problem, an external package was installed that addresses exactly this issue and recreates a native UITextView in Flutter (Leung et al., 2020). By using this package and the UITextView, the missing, essential basic functions of the TextFields on iOS could be added. Nevertheless, a distinction had to be made between the platforms and the desired behaviours, as the package only supports iOS and the Flutter TextField had to be implemented for Android.

**Exclude Exclusively Decorative Elements**

Excluding exclusively decorative elements from screen readers is important for both navigation and usability. For example, elements without meaning and functionality should be ignored by the screen reader and thus not be selectable. In addition, only content should be read aloud that helps screen reader users to operate and find their way around the app. This requirement is often not met by default and must be optimised. This could also be observed in the sample app, as the purely decorative product images are read out by the screen reader as "Image": "Vagabond sack, $240, 2 x $120, Image". To prevent this, the screen reader had to be explicitly told in the code that these elements should be excluded from the semantics. The ExcludeSemantics class could be used for this (Flutter Code 5, 2019). This is recognised and correctly interpreted by both screen readers without any problems.

**Adding Additional Semantic Texts for a Better Understanding of the App**

In certain cases, given information that is understandable in the user interface due to its grouping, colour or typography may be too imprecise for screen reader users. The same applies to special characters and abbreviations, which can be read out incomprehensibly by screen readers. Therefore, in these cases, an alternative, additional semantic must be defined for better understanding of the app. In the previous example with the product item, the character "x" is used as a multiplication sign.

Even if the special character is interpreted correctly in the user interface, the two screen readers do not recognise it as "times" but read it out as the normal character "x". Therefore, an alternative text must be defined for the screen reader, which communicates the information in a user-friendly, understandable way. To solve this problem, the semanticsLabel property of the Flutter Text element was used (Flutter Code 6, 2019). Using this, the original semantic "Vagabond sack, $240, 2 x $120" could be replaced with the semantic "Vagabond sack, $240, two times added to cart for 120$." As a result, the original text could be retained in the user interface, while an alternative text optimised for screen reader users was defined.

**Improving the Flow of Reading by Adapting the Semantic Text**

In addition to the problem that special characters and abbreviations are read out unintelligibly, there is also the problem that coherent text elements, which are, for example, only separated by paragraphs, are read out as one coherent text without reading pauses. This could also be observed in the sample app in Cart Tab with the overall price text. Despite the visually clear breaks, the entire text is interpreted and read out as a coherent text: "Shipping $21.00

Tax $10.32 Total $203.32". However, it is common in the user interface that individual words and information, such as prices, functions or categories, are only separated by breaks and not by dots or commas. Thus, the user interface was not changed here either, but only supplemented by an alternative semantic for the screen reader, which was defined by means of the semanticLabel property. In the alternative semantic, punctuation marks such as dots could thus be built in without any problems, allowing the desired reading pauses between the individual text breaks: "Shipping $21.00. [reading pause] Tax $10.32. [reading pause] Total $203.32. [reading pause]".

## RESULTS

### Initial Useability Tests

During the execution of the tasks in the non-optimised app, several accessibility issues were identified. For example, it was found that inactive, exclusively visual UI elements such as the product image are read out as "image". Also, essential elements like the Flutter TextField work differently than the native EditText elements on Android or the UITextField elements on iOS. Although all of these TextFields fulfill the main function of recording the text in the form of a string and displaying it in a text box or input field, important basic functionalities such as switching between letters in the TextField implemented by the Flutter are not recognised by VoiceOver. Other problems such as missing or too imprecise information about what should happen before or after activating a UI element were also described by the test group. The results of the testing showed that the sample app and its non-optimised, standard UI elements must be optimised in order to comply with the Accessibility Guidelines of Android (Android Developers, 2021) and iOS (Apple Developers, 2021) to be met.

### Useability Tests After Optimisation

After accessibility optimisation was carried out, the app was tested again by the same test group and with the same tasks. All test persons noticed a clear improvement in accessibility on both platforms and confirmed on both platforms that the implemented optimisation steps were successful.

In particular, optimisations of critical elements that did not function properly before, such as the TextFields in iOS, were perceived as extremely positive. Other optimisations, such as the addition of hints, feedback and more comprehensible semantic texts, also improved accessibility and were significant according to the interview survey.

## CONCLUSION

In summary, it was found that Flutter offers developers several possibilities to simultaneously optimise the app for both platforms in terms of screen reader accessibility. One of the most important possibilities is the semantics class. Using this, simple but effective optimisations, such as the exclusion of purely decorative elements or the addition of labels, reading pauses or alternative texts, could be implemented quickly and easily. For more complex

optimisations, such as overwriting hints or defining individual feedbacks for interactive UI elements, it was observed that the two platforms and their respective screen readers interpreted the codebase differently. Complex workarounds were necessary to deal with these differences. Among other things, each platform had to be queried in the Flutter codebase individually in order to fix the respective app behaviour with separate code snippets. It also turned out that basic UI elements, such as the Flutter TextFields, are interpreted differently by the two screen readers and do not offer essential basic functionalities that are crucial for screen reader accessibility. In this respect, it is up to the developers of Flutter to add these functionalities in the future. To conclude, on the one hand, the sample app could be partially optimised using only one codebase, which eliminates the need for two development teams for iOS and Android and saves on budget and resources. But on the other hand, several complex workarounds and platform-specific adaptations were necessary to achieve full accessibility. Thus, if the goal is to generate an accessible app using Flutter for both platforms without much effort, this is not possible at the time this study was conducted.

## REFERENCES

Android Developers (2021). "Android Accessibility Guidelines: Make apps more accessible" (Online). Available: https://developer.android.com/guide/topics/ui/accessibility/apps (accessed: Mar. 31 2021).

Apple Developers (2021). "Human Interface Guidelines: Accessibility" (Online). Available: https://developer.apple.com/design/human-interface-guidelines/accessibility/overview/introduction/ (accessed: Mar. 31 2021).

Flutter Code 1 (2019). Dart API - widgets library - Semantics class (Online). Available: https://api.flutter.dev/flutter/widgets/Semantics-class.html (accessed: Mar. 8 2021).

Flutter Code 2 (2019). Dart API - semantics library – SemanticsHintOverrides Class (Online). Available: https://api.flutter.dev/flutter/semantics/SemanticsHintOverrides-class.html (accessed: Mar. 31 2021).

Flutter Code 3 (2019). Dart API - semantics library - SemanticsConfiguration class - hint property (Online). Available: https://api.flutter.dev/flutter/semantics/SemanticsConfiguration/hint.html?web=1&wdLOR=c84BF2E97-DD26-4195-BA0A-BF1E64848AE6 (accessed: Mar. 31 2021).

Flutter Code 4 (2019). Dart API - semantics library - SemanticsService class (Online). Available: https://api.flutter.dev/flutter/semantics/SemanticsService-class.html?web=1&wdLOR=cD7EF0364-486D-4819-B5E6-EE4B38C388F5 (accessed: Mar. 31 2021).

Flutter Code 5 (2019). Dart API - widgets library - ExcludeSemantics class. [Online]. Available: https://api.flutter.dev/flutter/widgets/ExcludeSemantics-class.html (accessed: Mar. 31 2021).

Flutter Code 6 (2019). Dart API - widgets library - Text class - semanticsLabel property (Online). Available: https://api.flutter.dev/flutter/widgets/Text/semanticsLabel.html (accessed: Mar. 31 2021).

Fu, H., Cord, L., and McClain-Nhlapo, C. (2019). A billion people experience disabilities worldwide — so where's the data? World Bank Online: https://blogs.worldbank.org/opendata/billion-people-experience-disabilities-worldwide-so-wheres-data (accessed: Mar. 8 2021).

Leung, H. D'Amours, M., and Kolinko, F. (2020). "Native Text Input for Flutter" (Code Online). Available: https://pub.dev/packages/flutter_native_text_input (accessed: Mar. 31 2021).

Pascolini, D and Mariotti, S.P. (2012). "Global estimates of visual impairment: 2010," in: The British Journal of Ophthalmology, vol. 96, no. 5, pp. 614–618.

WebAIM (2017). WebAIM Screen Reader User Survey #7 Results (Online). Available: https://webaim.org/projects/screenreadersurvey7/ (accessed: Mar. 8 2021).

Zakhour, S and Morgan, B (2021). Building a Cupertino app with Flutter https://codelabs.developers.google.com/codelabs/flutter-cupertino#0 (accessed: Mar. 8 2021).