Flutter
Text Editing Deltas

SUMMARY

Flutter should provide granular updates for changes made to the platform's editing state for better cross-platform support for rich text editors.

Author: Renzo Olivares @Renzo-Olivares
Go Link: flutter.dev/go/text-editing-deltas
Created: 08/2021 / **Last updated:** 09/2021

OBJECTIVE

Provide developers with granular text changes from the platform's text input control.

BACKGROUND

Rich text support in Flutter currently only supports non-editable text. Developers can use Text.rich or SelectableText.rich constructors to build their non-editable rich text document user interfaces. However, there is no equivalent convenience API to do the same for a TextField. What a developer has to do currently to achieve rich text in a TextField is to override the TextEditingController.buildTextSpan method and apply any desired rich text attributes in there. Even with this override it is not clear how a developer would contract/expand rich text attributes based on input that is typed into the TextField.

The way the Flutter's text input system currently works is that a TextEditingValue is synced between the platform's text editing plugin on the engine side and the TextInputClient on the framework side. When the framework receives this new TextEditingValue from the engine there is no auxiliary information on what happened to this TextEditingValue since it was last synced with the framework. The framework gets the entire new value and sets this new value in the

TextEditingController, to be displayed by the TextField. Because there is no information on what happened to the TextEditingValue the developer is unaware what attribute ranges to expand/contract in their overridden TextEditingController.buildTextSpan.

For example, let's take "**hello**| world", we have a bold attribute of range 0 to 4. If we insert an "o" at the cursor position the bold attribute range should expand to be 0 to 5. To expand this range we need the information on where and how long the insertion was, without this information we cannot expand the range.

If we take a step back and look at the overall problem we see that this issue not only affects the TextField, but also any developer that wishes to create their own rich text editor, whether that is using Flutter's provided TextField or creating their own. We can take SuperEditor as an example, it is a fairly powerful rich text editor built completely in Flutter. However this rich text editor only works on the web and desktop platforms. The reason for this is the main crux of our issue; not having enough information to expand/contract attribute ranges with the current TextInputClient. SuperEditor's solution for this was to not use the TextInputClient, and instead listen to RawKeyEvents through the RawKeyboardListener, and manage their own text input model in that way. RawKeyboardListener does not support listening to software keyboards so that immediately eliminates support for mobile platforms, and the only other way to access the mobile IME is through the TextInputClient but they eliminated that as an option because it did not provide the deltas they needed.

Similarly we can also take another third party rich text editor that chooses to only use Flutter's provided widgets, Boustro, and observe this same issue. Because they do not have the information necessary to expand/contract ranges, they implement their own custom string diff that makes the assumption there has only been a single character insertion/deletion. The issue with a diff is that there is a lot of room for error especially with the different features that mobile IME's provide. Another option here could be to use a third party diffing package such as diff-match-patch. However, a diff like this does not have enough information to come up with the exact diff 100% of the time. For example take "aa|aa", if we add another a where the cursor is located we will have "aaa|aa", how will the diff be able to decide which "a" was inserted. Here the diff will not get the exact delta and instead most likely report that an "a" had been appended to the end of the value. You can try this example out here.

Because Flutter's goal is to be a cross-platform UI toolkit, then we should provide the tools necessary to build these rich text editors not only on desktop/web platforms but also on mobile. This means providing the auxiliary information developers need to expand/contract their rich text attribute ranges.

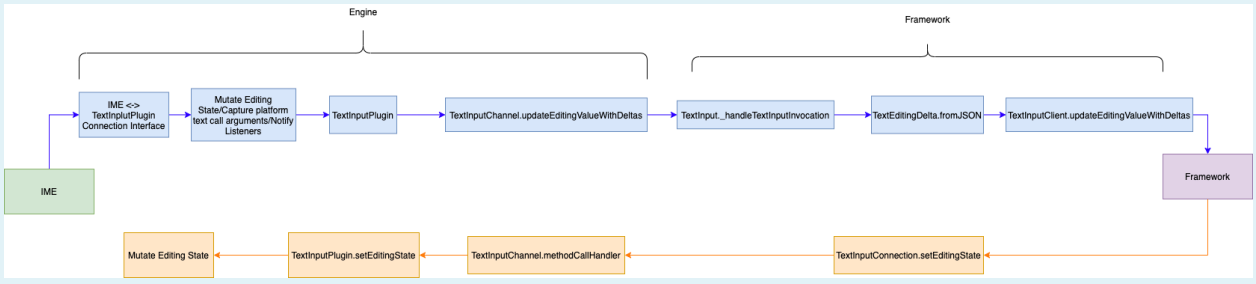
Glossary

- **Rich Text Attributes** - These are properties that a range of text can have. This can range from styling properties like bold and italics, or more complex constructs like bulleted lists.
- **TextEditingController** - A controller for an editable text field that can set the TextEditingValue and selection. A developer can also add listeners to the TextEditingController to do some action when the TextEditingValue changes.
- **TextEditingValue** - The value that is currently being edited by an editable text widget. It contains the plain text, the selection, and composing regions for the current editable state.
- **TextField** - A text field with material styling, builds upon EditableText which is the TextInputClient that interacts with the IME.
- **TextInputClient** - An interface on the framework side that receives and sends TextInput information to the engine. This information includes the current TextEditingValue.

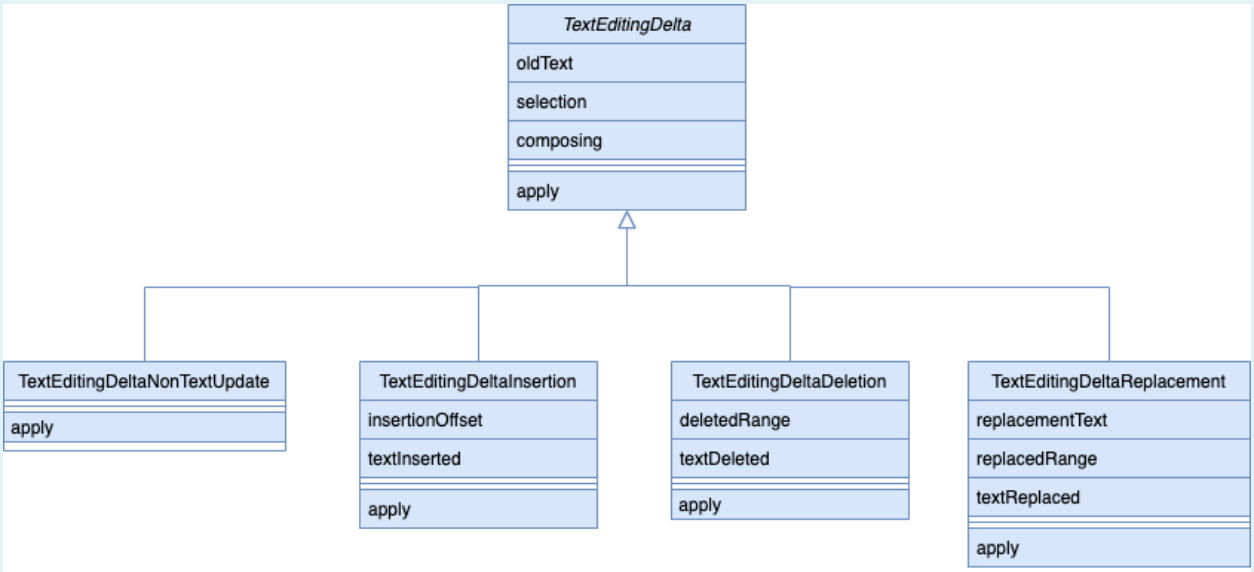
OVERVIEW

In order to allow developers to build powerful cross-platform rich text editors, I propose we provide a text editing delta for every change made by the platform's text input control.

We can do this by adding a new text input channel called updateEditingValueWithDeltas. Through this channel the platform's text input plugin can send it's platform text method call to the framework and that can be inferred into the appropriate delta.



Using Flutter's existing text editing architecture, this approach creates a new abstract class called TextEditingDelta where all the information for what happened to an editing state from time T1 to T2 will be stored. It will be accompanied by relevant subclasses: TextEditingDeltaInsertion, TextEditingDeltaDeletion, TextEditingDeltaReplacement, and TextEditingDeltaNonTextUpdate.



By sending the platform text method call to the framework first and having the delta logic be localized there we have a single place where this logic lives rather than having common logic across multiple platform text input plugins.

Non-goals

- Providing a user-friendly API to override InlineSpan's in a TextField.

DETAILED DESIGN/DISCUSSION

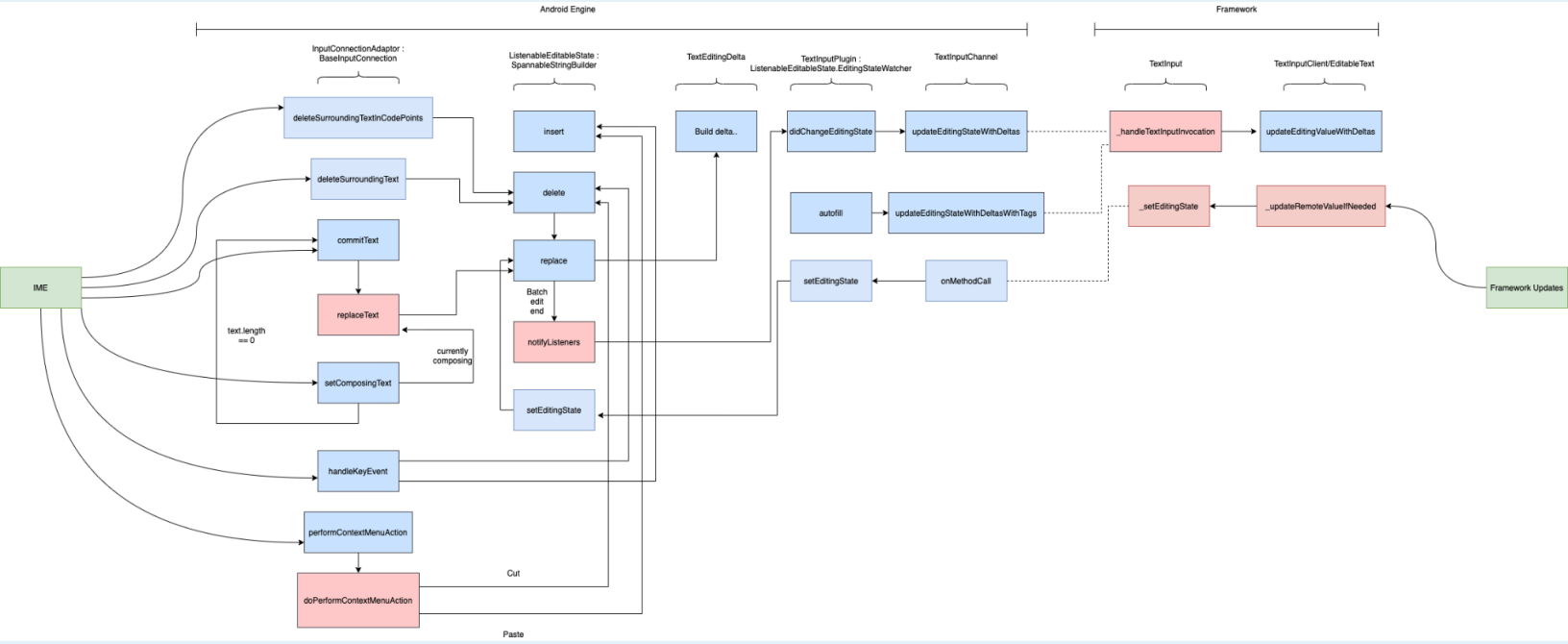
Some first steps here are identifying what methods in the framework and engine make modifications to the editing state. We need to hook into these methods and process the arguments into a normalized format that is sent to the framework by every platform text input plugin.

Framework

- Some components to keep in mind if we would like Flutter's own built in text editing widgets to support the delta model in the future.
    - InputFormatters
    - onChange Callbacks
    - TextEditingController
    - Clipboard
    - DefaultTextEditingActions
    - DefaultTextEditingShortcuts

Engine

Android TextInputPlugin



- ListenableEditingState extends SpannableStringBuilder and acts as our editable that the IME interacts with. To be clear the IME does not interact directly with the editable through methods in SpannableStringBuilder such as insert/append/delete. Instead the IME interacts indirectly with our editable through our BaseInputConnection implementation InputConnectionAdaptor.
- InputConnectionAdaptor
    - setComposingText()
        - calls BaseInputConnection.replaceText()
            - calls SpannableStringBuilder.replace()
                - Calls SpannableStringBuilder.setSpan()
                    - Called to update the composing or selection regions.
    - commitText()
        - BaseInputConnection.replaceText()
            - calls SpannableStringBuilder.replace()
    - deleteSurroundingText()
        - calls SpannableStringBuilder.delete()
    - deleteSurroundingTextInCodePoints()
        - calls SpannableStringBuilder.delete()
    - handleKeyEvent()
        - may call SpannableStringBuilder.delete()
        - may call SpannableStringBuilder.insert()
    - doPerformContextMenuAction()
        - Cut
            - calls SpannableStringBuilder.delete()
        - Paste

    - may call SpannableStringBuilder.delete()
    - may call SpannableStringBuilder.insert()
- ListenableEditingState
    - The editable that the IME is currently editing. When updateEditingState is called to sync up the framework's TextEditingValue with the one from the engine, the values from this editable are the ones piped to the framework. For this reason I propose the member fields that represent a TextEditingDelta (the normalized format on the engine side) be directly part of this editable state so we can pipe them to the framework through updateEditingStateWithDeltas.
    - extends SpannableStringBuilder
    - overrides SpannableStringBuilder.replace()
        - insert/delete/append are all convenience methods wrapped around replace(). For this reason I propose we hook into this method and capture the method call arguments and pipe that information to the framework to be consumed and processed into the appropriate TextEditingDelta subclass.
        - We should also hook into SpannableStringBuilder.setSpan() to capture non text updates, updates to the selection/composing regions.
- Miscellaneous
    - It looks like some soft keyboards may trigger the DefaultTextEditingShortcuts/Actions. Moving the cursor outside the composing region on hackers keyboard and pressing backspace to delete triggers the DeleteTextAction.

iOS TextInputPlugin

- replaceRange withText
    - calls replaceRangeLocal withText
    - calls updateEditingState
- setMarkedText selectedRange
    - calls replaceRangeLocal withText
- insertText
    - calls replaceRange withText
- deleteBackwards
    - calls replaceRange withText
- Similar to Android there is a central method that all text editing value modifiers call to mutate the value. We can hook into replaceRangeLocal withText, and handle the logic for creating TextEditingDeltas in this method directly, or by creating a new method that is called by replaceRangeLocal withText.
- I also propose that we keep the member fields that represent the TextEditingDelta inside this FlutterTextInputPlugin because the values from this object are the ones piped to the framework when the editing state is

updated.

Common Desktop

- Each desktop platform has its own TextInputPlugin but they all share a common TextInputModel at text_input_model.cc. I propose we hook into this common model's method and keep the member fields representing the TextEditingDeltas in this model. When the engine pipes its new editing state to the framework, on each of these platforms they use values from this common model.
- Platforms:
    - macOS
    - Windows
    - Linux
    - GLFW
- Common TextInputModel methods
    - UpdateComposingText
        - MacOS TextInputPlugin
            - called in SetMarkedText which is called when composing something with a native language keyboard
                - Not called when emoji keyboard is launched
                - Not called with accent keyboard
        - Linux TextInputPlugin
            - called from Im_preedit_changed_cb
                - Signal handler for gtkIMContext::preedit-changed
        - Windows TextInputPlugin
            - called from ComposeChangeHook
    - Backspace
        - Glfw TextInputPlugin
            - KeyboardHook
                - On GLFW_KEY_BACKSPACE
    - Delete
        - Glfw TextInputPlugin
            - KeyboardHook
                - On GLFW_KEY_DELETE
    - DeleteSurrounding
        - Linux TextInputPlugin
            - Im_delete_surrounding_cb
                - Signal handler for GtkIMContext::Delete-surrounding
    - SetText
        - Looks like this is mostly called when receiving an updated text editing value from the framework. This replaces the entire value on the engine side with the framework's updated version.
        - Glfw & Windows call in HandleMethodCall

            - This looks like a setEditingState equivalent.
        - Linux & macOS called in setEditingState
            - Updates editing state from framework
    - AddCodePoint
        - Calls AddText()
        - Glfw TextInputPlugin
            - EnterPressed
            - CharHook
        - Linux TextInputPlugin
            - fl_text_input_filter_keypress_default
    - AddText is called by
        - macOS TextInputPlugin
            - insertText()
        - Linux TextInputPlugin
            - im_commit_cb
        - Windows TextInputPlugin
            - TextHook
            - ComposeChangeHook
            - EnterPressed
        - There is also an AddText called in lib/ui/text/paragraph_builder.cc but it seems unrelated
            - AddText() calls addText()
        - There are also calls to addText in third_party/txt/src/skia && txt/benchmarks but they also seem unrelated to this model.

Common Web

- BeforeInputEvent: This event is fired when the value of <input>, <select>, <textarea> or an element with contenteditable enabled is about to be modified. This allows us to override the action before the DOM is mutated. We can utilize this API to capture the character insertions.
    - Example of how this looks in dart: https://dartpad.dev/?id=80a4ac2f0d1a134b335e5286092a06f2&null_safety=true
- CompositionEvent:
    - compositionupdate: We can also utilize this API to capture the composing regions on different platforms.
- I propose we utilize BeforeInputEvent, and add a listener to the active DOM element that is being edited. In the callback we can handle the logic for normalizing the platform text change information into our common format, and these deltas can be piped back to the framework when the engine updates the editing state. We should also use compositionupdate for when we are composing text to capture the composing region.

Android Web

- On Android web, the IME acts similarly to native Android where it replaces the entire composing region when you are composing text, if you are outside of the composing region we do get character insertions. Because of this BeforeInputEvent always reports the replacement of the entire composing region as the insertion versus a single character. This is the same problem we face on the Android TextInputPlugin, so we can definitely use the same logic here. This is the advantage of having the delta logic localized on the framework side; to add support for a new platform you must capture the platform's text change method call, normalize it, and send that over to the framework.

iOS Web

- On iOS web, character insertions are reported by BeforeInputEvent, however we will still need to have some logic to handle "Replacement" cases such as autofill, autocorrect, and suggestions. The logic for that will all be localized on the framework.

API

DeltaTextInputClient:

```
/// An interface to receive granular information from [TextInput].
///
/// See also:
///
///  * [TextInput.attach]
///  * [TextInputConfiguration], to opt-in to receive [TextEditingDelta]'s from
///    the platforms [TextInput] you must set
[TextInputConfiguration.enableDeltaModel]
///    to true.
abstract class DeltaTextInputClient extends TextInputClient {
 /// Requests that this client update its editing state by applying the deltas
 /// received from the engine.
 ///
 /// The list of [TextEditingDelta]'s are treated as changes that will be applied
 /// to the client's editing state. A change is any mutation to the raw text
 /// value, or any updates to the selection and/or composing region.
 void updateEditingValueWithDeltas(List<TextEditingDelta> textEditingDeltas);
}
```

TextEditingDelta:

```
/// A structure representing a granular change that has occurred to the editing
/// state because of text editing.
///
/// See also:
///
///  * [TextEditingDeltaInsertion], a delta representing an insertion.
```

```
///  * [TextEditingDeltaDeletion], a delta representing a deletion.
///  * [TextEditingDeltaReplacement], a delta representing a replacement.
///  * [TextEditingDeltaNonTextUpdate], a delta representing an update to the
///    selection and/or composing region.
///  * [TextInputConfiguration], to opt-in your [DeltaTextInputClient] to receive
///    [TextEditingDelta]'s you must set [TextInputConfiguration.enableDeltaModel]
///    to true.
abstract class TextEditingDelta {
 /// Creates a delta for a given change to the editing state.
 ///
 /// The [oldText], [selection], and [composing] arguments must not be null.
 const TextEditingDelta({
   required this.oldText,
   required this.selection,
   required this.composing,
 }) : assert(oldText != null),
      assert(selection != null),
      assert(composing != null);

 /// Creates an instance of this class from a JSON object by inferring the
 /// type of delta based on values sent from the engine.
 factory TextEditingDelta.fromJSON(Map<String, dynamic> encoded) {}

 /// The old text state before the delta has occurred.
 final String oldText;

 /// The range of text that is currently selected after the delta has been
 /// applied.
 final TextSelection selection;

 /// The range of text that is still being composed after the delta has been
 /// applied.
 final TextRange composing;

 /// This method will take the given [TextEditingValue] and return a new
 /// [TextEditingValue] with that instance of [TextEditingDelta] applied to it.
 TextEditingValue apply(TextEditingValue value);
}
```

TextEditingDeltaInsertion:

```
/// A structure representing an insertion of a single/or contiguous sequence of
/// characters at some offset of an editing state.
class TextEditingDeltaInsertion extends TextEditingDelta {
 /// Creates an insertion delta for a given change to the editing state.
 const TextEditingDeltaInsertion({
   required String oldText,
   required this.textInserted,
   required this.insertionOffset,
   required TextSelection selection,
   required TextRange composing,
 }) : super(
```

```
    oldText: oldText,
    selection: selection,
    composing: composing,
 );

 /// The text that is being inserted into [oldText].
 final String textInserted;

 /// The offset in the [oldText] where the insertion begins.
 final int insertionOffset;

 @override
 TextEditingValue apply(TextEditingValue value) {}
}
```

TextEditingDeltaDeletion:

```
/// A structure representing the deletion of a single/or contiguous sequence of
/// characters in an editing state.
class TextEditingDeltaDeletion extends TextEditingDelta {
 /// Creates a deletion delta for a given change to the editing state.
 const TextEditingDeltaDeletion({
   required String oldText,
   required this.deletedRange,
   required TextSelection selection,
   required TextRange composing,
 }) : super(
   oldText: oldText,
   selection: selection,
   composing: composing,
 );

 /// The range in [oldText] that is being deleted.
 final TextRange deletedRange;

 /// The text from [oldText] that is being deleted.
 String get textDeleted => oldText.substring(deletedRange.start, deletedRange.end);

 @override
 TextEditingValue apply(TextEditingValue value) {}
}
```

TextEditingDeltaReplacement:

```
/// A structure representing a replacement of a range of characters with a
/// new sequence of text.
class TextEditingDeltaReplacement extends TextEditingDelta {
 /// Creates a replacement delta for a given change to the editing state.
 ///
 /// The range that is being replaced can either grow or shrink based on the
```

```
 /// given replacement text.
 ///
 /// A replacement can occur in cases such as auto-correct, suggestions, and
 /// when a single character replaces a selection.
 const TextEditingDeltaReplacement({
   required String oldText,
   required this.replacementText,
   required this.replacedRange,
   required TextSelection selection,
   required TextRange composing,
 }) : super(
   oldText: oldText,
   selection: selection,
   composing: composing,
 );

 /// The new text that is replacing [replacedRange] in [oldText].
 final String replacementText;

 /// The range in [oldText] that is being replaced.
 final TextRange replacedRange;

 /// The original text that is being replaced in [oldText].
 String get textReplaced => oldText.substring(replacedRange.start,
replacedRange.end);

 @override
 TextEditingValue apply(TextEditingValue value) {}
}
```

TextEditingDeltaNonTextUpdate:

```
/// A structure representing changes to the selection and/or composing regions
/// of an editing state and no changes to the text value.
class TextEditingDeltaNonTextUpdate extends TextEditingDelta {
 /// Creates a delta representing no updates to the text value of the current
 /// editing state. This delta includes updates to the selection and/or composing
 /// regions.
 ///
 /// A situation where this delta would be created is when dragging the selection
 /// handles. There are no changes to the text, but there are updates to the
selection
 /// and potentially the composing region as well.
 const TextEditingDeltaNonTextUpdate({
   required String oldText,
   required TextSelection selection,
   required TextRange composing,
 }) : super(
   oldText: oldText,
   selection: selection,
   composing: composing,
 );
```

```
 @override
 TextEditingValue apply(TextEditingValue value) {}
}
```

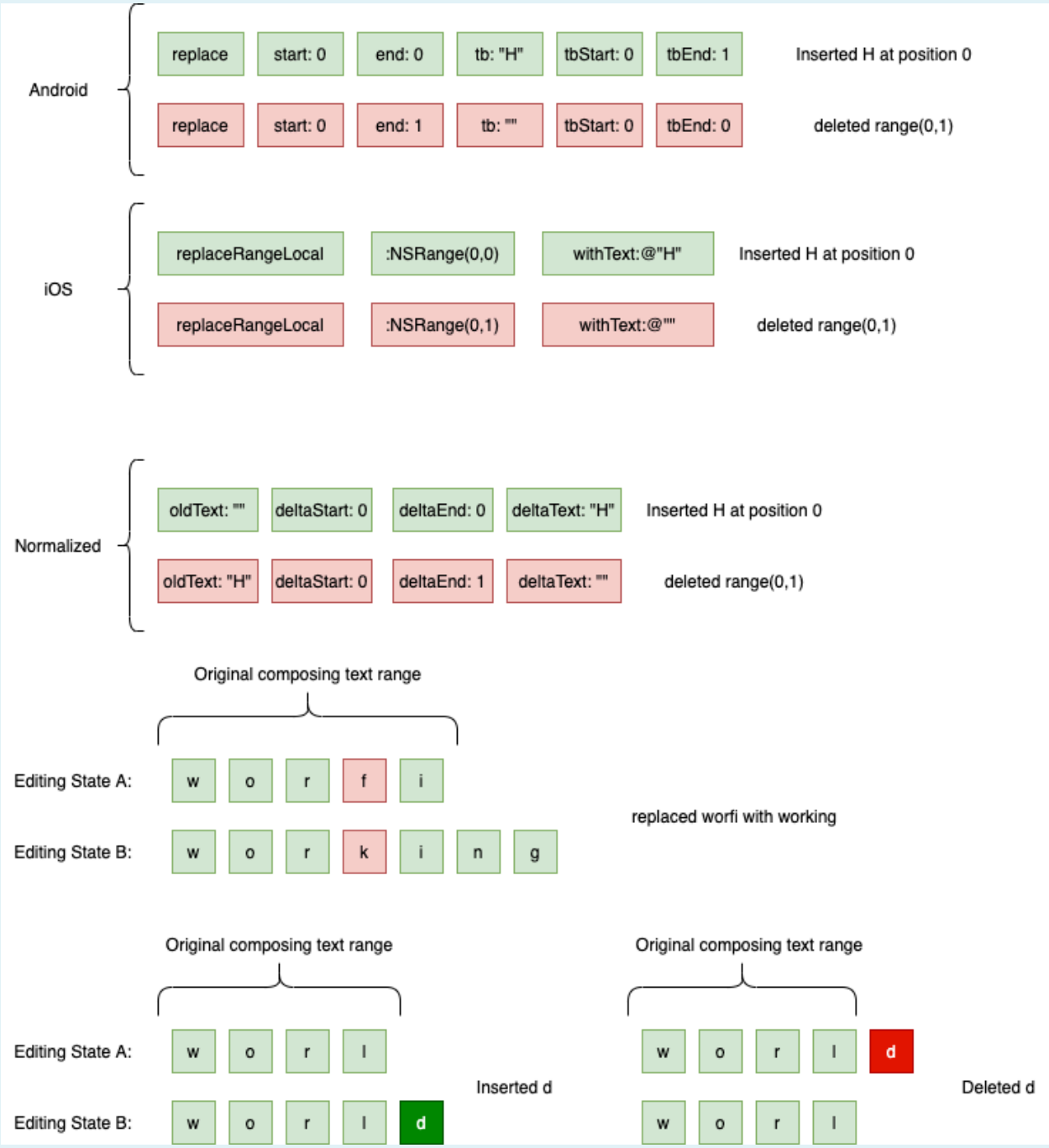
Updates to TextInputConfiguration

```
class TextInputConfiguration {
    /// Whether to enable that the engine sends text input updates to the
    /// framework as [TextEditingDelta]'s or as one [TextEditingValue].
    final bool enableDeltaModel;
}
```

Updates to TextInput

```
class TextInput {
    Future<dynamic> _handleTextInputInvocation(MethodCall methodCall) async {
        switch (method) {
          case 'TextInputClient.updateEditingStateWithDeltas':
             (_currentConnection!._client as
DeltaTextInputClient).updateEditingValueWithDeltas(batchDeltas);
             break;
        }
    }
}
```

How are deltas inferred?



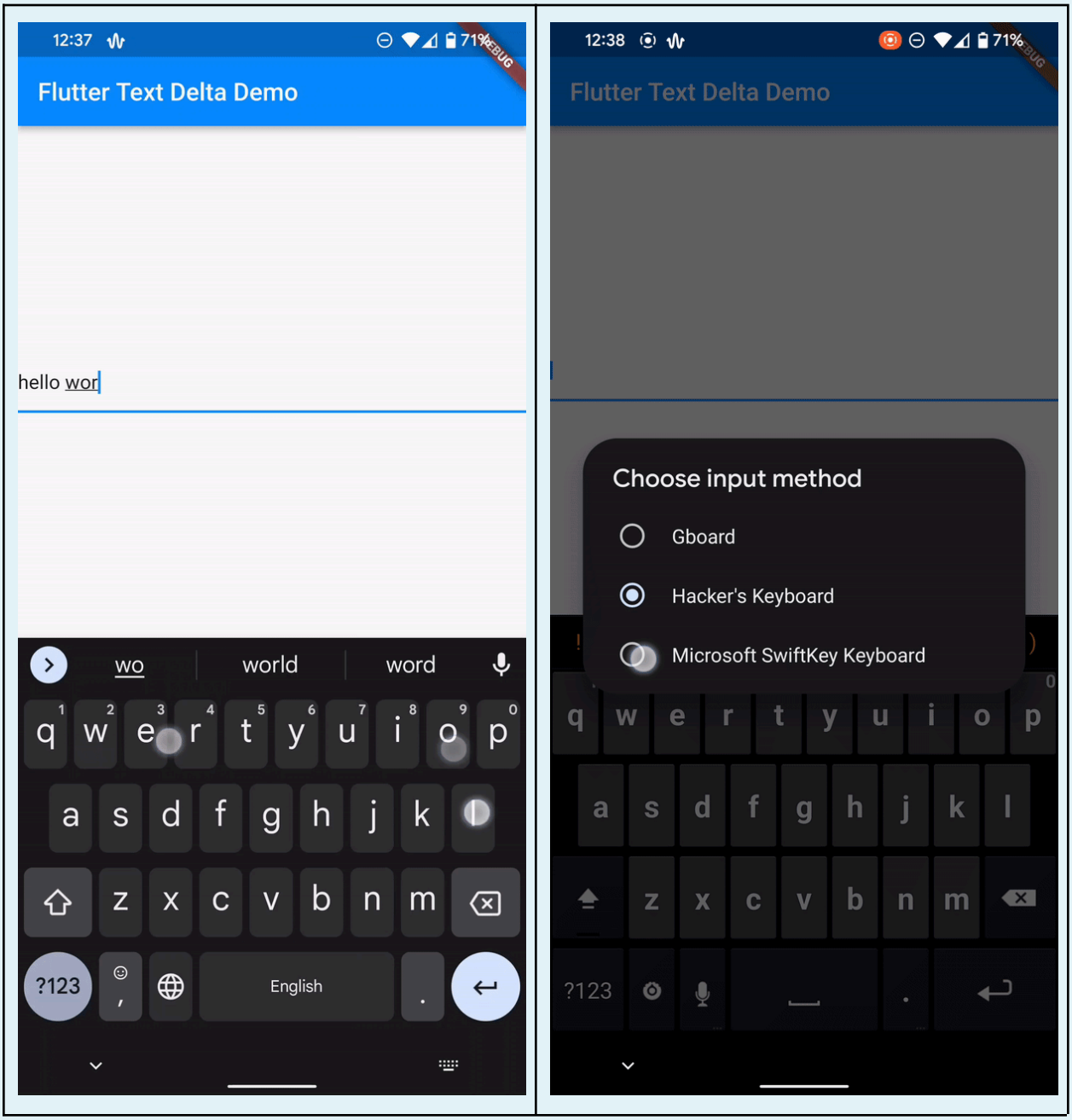
- In the diagram above we use Android and iOS as an example. We show the platform text input call for inserting an H and then deleting that same H. The main information that is being told by both these platforms variations of a replace call is that some range of text is being replaced by some given text. Given this information we can normalize these calls into a common format.
    - oldText holds the old editing state before the platform text method was called.
    - deltaText contains the source string being used for replacement.
    - deltaStart contains the beginning of the range being replaced.
    - deltaEnd contains the end of the range being replaced.
- We do not have to “infer” everything. The platform’s give concrete documentation for what an insertion and deletion is.

    - Let's take Android for example https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/text/Editable.java;l=29?q=editable .
        - An insertion is when the platform's text editing method arguments contain a collapsed target range and a non-empty source text.
        - A deletion is when the platform's text editing method arguments contain an empty source text.
- Where we do have to infer is the composing region.
    - When the composing region is active the platform reports the entire composing region as being replaced on every update. Take the example in the diagram above, where "worfi" is being auto-corrected to "working". We say this is a replacement because when we compare the original composing text "worfi" with the same range from the new source text "worki" we see that they are different.
    - Another example is the one where we go from "worl" to "world". The platform will tell us "worl" has been replaced with "world", but we would like instead to know that "d" has been inserted at the end of the word. We can do this by comparing the original composing text "worl" and the same range in source text which is also "worl". Because these two are the same we can say everything that came after the composing region is an insertion. Similarly you can follow similar logic for a deletion.

Proof of Concept

- In the proof of concept below the TextField is using a DeltaTextEditingController which is identical to a TextEditingController except with a function to apply TextEditingDelta's to its value. Below we are listening to input changes through the TextInputClient.updateEditingValueWithDeltas channel which gives us deltas instead of TextInputClient.updateEditingValue which only gives us a plain text value and not a delta.

Demo part 1	Demo part 2



Usage

To use the new TextEditingDelta API a developer will have to implement the new DeltaTextInputClient. They will also need a TextInputConfiguration with the enableDeltaModel flag set to true. Doing these two things will allow a developer to access granular updates from the platform's text input control through TextEditingDelta's. It will then be up to them on how they consume these deltas.

OPEN QUESTIONS

- Should TextEditingDeltas be sent to the engine as well?
    - Does the engine require this information for anything/ will it in the future?

- How does fuchsia handle text input? Does it?

IMPLEMENTATION PLAN

Framework

- PR: https://github.com/flutter/flutter/pull/88477 .
    - Follow up: https://github.com/flutter/flutter/pull/90205

Engine

- Separate PR's
    - Android
        - https://github.com/flutter/engine/pull/28175
    - iOS
        - https://github.com/flutter/engine/pull/28418
    - Windows
        - https://github.com/flutter/engine/pull/30329
    - Linux
        - https://github.com/flutter/engine/pull/29215
    - MacOS
        - https://github.com/flutter/engine/pull/29036
    - Web
        - https://github.com/flutter/engine/pull/28527

TESTING PLAN

Framework

- Tests added for testing the TextEditingDelta.fromJSON constructor and verifying the correct delta is inferred
- Tests for verifying the new updateEditingValueWithDeltas channel is called.

Engine

- Tests added to Android embedding to verify the creation of deltas.
- Tests to verify that updates are only coming through one channel: updateEditingValue or updateEditingValueWithDeltas
- Tests added to iOS embedding to verify the creation of deltas.
- Tests added to Web embedding to verify correct inference of deltas.

DOCUMENTATION PLAN

- Any special handling of the composing region, autocorrect, suggestions, and any other IME features should be well documented.
- TextEditingDelta and its subclasses should be well documented with explanations on how a delta is inferred.
- A simple demo of that includes a TextField and some text above it that shows the deltas as you type them into the TextField.

MIGRATION PLAN

- Update any broken tests