Article

Adding Writing Tools support to a custom AppKit view

Integrate Writing Tools support, including support for inline replacement animations, to your custom text views on macOS.

Overview

Even if you don’t use a `NSTextView` or `NSTextField` object in your app, you can still add Writing Tools support to other views that contain text. You might use this support when you prefer not to use the standard text views. For example, you might use it if you build your own text view using TextKit or your own proprietary text engine. The AppKit API for this support gives you access to the same Writing Tools features available in the system views, including the ability to animate changes to the text in your view.

When a person triggers the Writing Tools UI from your custom view, AppKit works with your view to evaluate the relevant text and incorporate changes. You specify the type of Writing Tools experience you want for your view. The complete experience animates changes to your view’s content directly, working with your view to create those animations. The limited experience displays changes in the Writing Tools UI, and only incorporates the final changes back into your view’s text storage. All of these interactions happen with the help of the `NSWritingToolsCoordinator` object you assign to your view.

Add a Writing Tools coordinator to your view

An `NSWritingToolsCoordinator` object manages interactions between your view and the Writing Tools feature. You attach this object to your view by assigning it to the `writingToolsCoordinator` property of `NSView`. To manage your view-specific behavior, you provide a delegate object when setting up the coordinator. Your delegate provides Writing Tools with initial text to evaluate, incorporates changes, provides proofreading marks, and provides preview objects to use during animations.

Attach a coordinator object to your view, typically when creating and configuring that view. Supply a delegate object that adopts the `NSWritingToolsCoordinator.Delegate` protocol and has access to your view's text content and layout information. The following example shows an extension to a custom view that creates the coordinator and initializes it with the view itself. The custom method checks to see if Writing Tools is available before creating a coordinator object and adding it to the view.

```
class MyTextView : NSView {
    var coordinator: NSWritingToolsCoordinator?

   //…
}

extension MyTextView : NSWritingToolsCoordinator.Delegate {

    func configureWritingTools() {
        guard NSWritingToolsCoordinator.isWritingToolsAvailable else { return }
        guard coordinator == nil else { return }

        let coordinator = NSWritingToolsCoordinator(delegate:self)
        writingToolsCoordinator =  coordinator
    }
   //…
}
```

The presence of a coordinator object on your view causes AppKit to enable Writing Tools support for that view. When someone interacts with your view's contextual menu, AppKit automatically adds a command to that menu to launch Writing Tools. Similarly, the text-input system displays a visual affordance in response to relevant interactions with your view's text.

For a complete list of delegate methods you must implement to support Writing Tools, see `NSWritingToolsCoordinator.Delegate`.

Configure your view's Writing Tools behavior

When setting up Writing Tools for your view, configure your coordinator's `preferredBehavior` property with the types of interactions you support. Writing Tools offers different levels of integration with your view's content, which also affects how much work you need to do to support the feature. With the complete experience, you provide AppKit with more information so that it can make changes directly in your view. The limited experience pushes more interactions into the Writing Tools UI, giving you a less integrated experience, but one that's easier to implement.

At configuration time, you also specify the types of text content your view supports using your coordinator's `preferredResultOptions` property. The models that Writing Tools uses to evaluate your text can generate plain or formatted text. Writing Tools supports all types of output by default, but you can limit it to specific types as needed.

The following code updates the previous example method, and adds some preferred behaviors for the coordinator object. In addition to wanting the inline experience for Writing Tools interactions, the method requests that the system generate rich text and optional list-based content. Providing a specific list of result options tells Writing Tools to generate only that type of content. You might provide this information if your view doesn't support specific types of content, like tables.

```
func configureWritingTools() {
    guard NSWritingToolsCoordinator.isWritingToolsAvailable else { return }
    guard coordinator == nil else { return }

    let coordinator = NSWritingToolsCoordinator(delegate:self)
    coordinator.preferredBehavior = .complete
    coordinator.preferredResultOptions = [.richText, .list]
    writingToolsCoordinator =  coordinator
}
```

The availability of Writing Tools depends on the current device, its operating system, and its readiness to execute requests. On devices with appropriate hardware, some Writing Tools features operate locally after the system downloads the required models. However, more complex requests require a network connection so the system can use Private Cloud Compute to evaluate the provided text.

Supply your view's text for evaluation

When someone starts Writing Tools in your view, the system asks your delegate to provide the text to evaluate. Depending on what the person wants to do, Writing Tools might ask for only the selected text or it might ask for all of your view's text. For example, they might want to proof only the currently selected text, or they might want to proofread all of the text in your view. You provide that text from your delegate inside a context object.

A `NSWritingToolsCoordinator.Context` object is a data object that you fill with the requested text. Context objects provide the common ground that you and Writing Tools use to communicate throughout a single operation. For each request, you create one context object for each text-storage object that has text for Writing Tools to consider. Most views have only one text-storage object and therefore create only one context object. However, a view that uses multiple subviews to manage different parts of its content might assign separate text-storage objects to each subview. In that scenario, you create one context object for each subview that contains the requested text.

To request your view's context object, the coordinator calls your delegate's `writingToolsCoordinator(_:requestsContextsFor:completion:)` method. Use the parameters of that method to determine if Writing Tools wants all of your view's text or only some of it. The following example creates a single context object with either the currently selected text or the view's full text. After creating the context, the custom `storeContexts` method saves a reference to the context for subsequent tasks.

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        requestsContextsFor scope: NSWritingToolsCoordinator.ContextScope,
        completion: @escaping ([NSWritingToolsCoordinator.Context]) -> Void) {

    // Store the created contexts for the completion handler.
    var contexts = [NSWritingToolsCoordinator.Context]()

    switch scope {
    case .userSelection:
        let context = getContextObjectForSelection()
        contexts.append(context)
        break

    case .fullDocument:
        let context = getContextObjectForFullDocument()
        contexts.append(context)
        break

    case .visibleArea:
        let context = getContextObjectForVisibleArea()
        contexts.append(context)
        break

    default:
        break
    }

    // Save references to the contexts for later delegate calls.
    storeContexts(contexts)

    // Deliver the contexts to Writing Tools.
    completion(contexts)
}
```

When specifying only a subset of your view's text, provide some of the surrounding text to give Writing Tools additional content for its evaluation. As a general principle, include whole paragraphs of text in the context object, not just the currently selected text. Use the context object's range value to identify the portion of the overall text that Writing Tools requested. For example, if you fill a context object with the current text selection and some of the surrounding text, use the context object's `range` property to specify the location of the selected text in that object.

After you create a context object, cache any additional information that you need to map the text in your context object to the text in your view's text storage. You need to know where the content for each context object starts, in order to update your text storage later. One option is to create a dictionary that maps the context object's identifier to the starting location of its text in your text storage. You can use that value to adjust any context-specific ranges that Writing provides you later.

The following example shows a method that the delegate uses to create a context object for the current text selection. The method calls the custom `getSelectedTextToEvaluate` function, which returns an expanded version of the text that includes both the selection and some of the surrounding text. The method also returns the starting location of that expanded text in the view's text storage. Because the text selection is now in the middle of the text, the method initializes the context object with a range that provides the location of only the selected text, relative to the start of `textToEvaluate`. The method saves the actual starting location in a dictionary variable, mapping the value to the context's unique identifier.

```swift
var startingLocationsForContexts = [UUID : Int]()

func getContextObjectForSelection() -> NSWritingToolsCoordinator.Context {
    // Get the text to evaluate, which includes the text selection and some
    // of the surrounding text. The method returns (NSAttributedString, Int),
    // which represents the text itself, and the starting location of that
    // text in the view's text storage.
    let (textToEvaluate, startLocation) = getSelectedTextToEvaluate()

    // Get the NSRange for the text selection, relative to the text storage.
    let textSelectionRange = getTextSelectionRange()
    var contextRange = textSelectionRange

    // The view guarantees that startLocation is less than or equal to the
    // selection range. Adjust contextRange so that location 0 corresponds
    // to the first character in textToEvaluate. Keep the length the same.
    contextRange.location = textSelectionRange.location - startLocation

    // Create the new context object.
    let context = NSWritingToolsCoordinator.Context(attributedString: textToEvaluate
                       range: contextRange)

    // Save the starting location of the text, relative to the text storage.
    startingLocationsForContexts[context.identifier] = startLocation

    return context
}
```

After you create a context object, you can use its unique identifier as a key for saving other data. The context objects you create remain in existence for the duration of the current Writing Tools operation. If the person rejects the current changes and starts a new operation, Writing Tools asks you for a new set of context objects.

Incorporate changes back into your text storage

After evaluating your view's text, Writing Tools delivers any suggested changes to your delegate object. If your view adopts the limited experience, Writing Tools waits until the person accepts any changes before delivering them to your view. If you adopt the complete experience, Writing Tools delivers the changes before the person accepts them. If the person later rejects the changes, Writing Tools delivers a new set of changes that restore your view's original text.

In your delegate's `writingToolsCoordinator(_:replace:in:proposedText:reason:animationParameters:completion:)` method, incorporate the specified change into your view's text storage. Writing Tools calls this method for each distinct change it needs to make, and it might call the method multiple times with different range values for the same context object. The following simple example validates the provided range information and validates the view's own cached information for the context object. If everything is valid, the method then calculates the correct range of text to replace and creates a transaction to perform the replacement. Before returning, the method executes the provided completion handler with the text it incorporated, if any.

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        replace range: NSRange,
        in context: NSWritingToolsCoordinator.Context,
        proposedText replacementText: NSAttributedString,
        reason: NSWritingToolsCoordinator.TextReplacementReason,
        animationParameters: NSWritingToolsCoordinator.AnimationParameters?,
        completion: @escaping (NSAttributedString?) -> Void) {

    // Make sure there's a valid starting location in the text storage.
    guard let startingLocation = startingLocationsForContexts[context.identifier]
            else { completion(nil); return }
    guard let textStorage = textContentStorage.textStorage else { completion(nil); 
    // Determine the correct location in the text storage.
    let adjustedRange = NSRange(location: startingLocation + range.location, length:

    // Update the text storage using a transaction.
    textContentStorage.performEditingTransaction {
        textStorage.replaceCharacters(in: adjustedRange, with: replacementText)
    }

    completion(replacementText)
}
```

After each replacement operation, update other parts of your app as needed to account for the change. An `NSTextContentManager` object automatically generates the required layout updates when you make changes using a transaction. If you need to update other parts of your interface, initiate those changes from your `writingToolsCoordinator(_:replace:in:proposedText:reason:animationParameters:completion:)` method. For example, you might update a view that displays the current character count for your document. Check the `reason` parameter to determine when Writing Tools is making changes interactively, and use the provided animation parameters object to create the actual animations.

Update your view’s selected text

When working on your view’s selected text, Writing Tools updates the text selection to account for any text updates. Use your delegate’s `writingToolsCoordinator(_:select:in:completion:)` method to update the selected text in your view. The method delivers an array of range values to allow for views to create discontiguous selections. If your view supports only a continuous range of selected characters, update your view’s selection based on the first element in the `ranges` array.

When implementing your delegate method, remember to update range values to account for offset to the start of the text in your context object. The following example creates an adjusted set of ranges by adding the starting location recorded at the start of the operation for the context object. It then passes that information to the view’s text engine to highlight the appropriate ranges of text.

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        select ranges: [NSValue],
        in context: NSWritingToolsCoordinator.Context,
        completion: @escaping () -> Void) {

    guard let startingLocation = startingLocationsForContexts[context.identifier]
            else { completion(); return }
    var adjustedRanges = [NSRange]()

    for value in ranges {
        let range = value.rangeValue
        let newRange = NSRange(location: startingLocation + range.location, length:
        adjustedRanges.append(newRange)
    }

    // Highlight the specified text in the view.
    selectTextInDocument(adjustedRanges)
    completion()
}
```

Generate preview images for inline animated changes

When you choose the complete Writing Tools experience for your view, the system animates changes to your content directly in your view. The system creates animations when Writing Tools starts evaluating your text, when it removes old text, and when it inserts new text. Because the animations involve your content, you must help the system create them. At appropriate times, the coordinator asks your delegate object to perform the following tasks:

1. Create a preview image of a specific portion of your content.

2. Hide that content when the animation starts.

3. Show the content again when the animation finishes.

To create a preview image of your text, configure a graphics context for bitmap-based drawing in your your delegate's `writingToolsCoordinator(_:requestsPreviewFor:of:in:completion:)` method. Get the frame rectangle for the specified range of text using your layout manager, and use your graphics context to draw the text in that rectangle and generate an image.

The following example shows an implementation of the `writingToolsCoordinator(_:requestsPreviewFor:of:in:completion:)` method that creates a preview for the view's text. The method relies on the view's layout manager to get the text rectangles that surround each line of text in the specified range. It also uses the layout manager to draw the text using the provided graphics context. The method then uses that context to generate an image of the rendered text and create a `NSTextPreview` object with the required information. When creating animations, Writing Tools applies the required visual effects to the provided preview image instead of to the text itself.

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        requestsPreviewFor textAnimation: NSWritingToolsCoordinator.TextAnimation,
        of range: NSRange,
    in context: NSWritingToolsCoordinator.Context,
    completion: @escaping ([NSTextPreview]?) -> Void) {

let textRange = MyTextRange(range)         // Create the view's custom subclass
let textRects = getTextRectangles(range)    // Returns an array of NSValue<NSRec
var previews = [NSTextPreview]()

if !textRange.isEmpty {
    // Get the frame rectangle that encloses the text.
    let textFrame = unionRect(for: textRects.map({$0.rectValue}))

    // Configure a graphics context for drawing in the specified rectangle.
    let scaleFactor = self.window!.backingScaleFactor
    let bitsPerComponent = 8
    let numberOfComponents = 4
    let pixelWidth = Int(textFrame.size.width * scaleFactor)
    let pixelHeight = Int(textFrame.size.height * scaleFactor)
    let bytesPerRow = pixelWidth * numberOfComponents
    let bitmapInfo = CGImageAlphaInfo.premultipliedLast.rawValue
    let colorSpace = CGColorSpaceCreateDeviceRGB()

    if let context = CGContext(data: nil, width: pixelWidth, height: pixelHeight
                      bitsPerComponent: bitsPerComponent, bytesPerRow: bytesPerR
                      space: colorSpace, bitmapInfo: bitmapInfo) {
        // Scale / Translate
        context.scaleBy(x: scaleFactor, y: -scaleFactor)
        context.translateBy(x: 0, y: -textFrame.size.height)
        // Translate for origin
        context.translateBy(x: -textFrame.origin.x, y: -textFrame.origin.y)

        // Draw the specified text using the provided graphics context.
        drawText(textRange, in: context)

        // Generate the image and preview.
        if let image = context.makeImage() {
            let preview = NSTextPreview(snapshotImage: image,
                    presentationFrame: textFrame,
                    candidateRects: textRects)
            previews.append(preview)
        }
    }
}
```

```
    completion(previews)
```

In addition to providing the initial image, use the `writingToolsCoordinator(_:prepareFor:for:in:completion:)` method to hide the specified range of text in your view. UIKit places the image from your `NSTextPreview` object in your view at the location you specified. When the animations finish, Writing Tools calls the `writingToolsCoordinator(_:finish:for:in:completion:)` method so you can show the text in the specified range again.

Create proofreading marks for your content

If someone chooses a proofreading option, Writing Tools evaluates your view's text and asks you to provide proofreading marks to decorate the view. For each mark, Writing Tools asks you to provide a Bézier path that underlines the text in a particular range. The following example uses the view's layout manager to get the bounding rectangles for the specified range of text. It then flattens those rectangles to create a line shape underneath the text and passes the resulting paths to the completion handler. The code creates separate shapes for each rectangle to account for situations where a proofreading mark extends onto multiple lines of text.

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        requestsUnderlinePathsFor range: NSRange,
        in context: NSWritingToolsCoordinator.Context,
        completion: @escaping ([NSBezierPath]) -> Void) {

    let textRects = getTextRectangles(range)   // Returns an array of NSValue<NSRect

    var paths = [NSBezierPath]()
    for rect in textRects.map({$0.rectValue}) {
        let underlineHeight: CGFloat = 2
        let newRect = CGRect(x: rect.origin.x, y: rect.origin.y + rect.height - (unc
                             width: rect.width, height: underlineHeight)
        paths.append(NSBezierPath(rect: newRect))
    }

    completion(paths)
}
```

In addition to providing the proofreading mark shapes, Writing Tools also asks you to provide the bounding rectangles for the text itself. Writing Tools uses these bounding rectangles to draw highlights around your text. The following example retrieves the bounding rectangles for the specified range of text and passes a Bézier path for each rectangle to the completion handler:

```
func writingToolsCoordinator(_ writingToolsCoordinator: NSWritingToolsCoordinator,
        requestsBoundingBezierPathsFor range: NSRange,
        in context: NSWritingToolsCoordinator.Context,
        completion: @escaping ([NSBezierPath]) -> Void) {

    let textRects = getTextRectangles(range)   // Returns an array of NSValue<NSRect

    var paths = [NSBezierPath]()
    for rect in textRects.map({$0.rectValue}) {
        paths.append(NSBezierPath(rect: rect))
    }

    completion(paths)
}
```

Respond to state changes

During the course of changing content, the state of the Writing Tools system changes based on what's happening. Writing Tools starts in the inactive state, but quickly moves to other states based on the type of experience it creates. When a person accepts or rejects the changes for the current operation, Writing Tools moves back to inactive state. You might use your delegate's `writingToolsCoordinator(_:willChangeTo:completion:)` method to respond to the following types of changes:

- Use a transition to the `NSWritingToolsCoordinator.State.inactive` state to clear any cached data from the previous operation.
- Use transitions to the `NSWritingToolsCoordinator.State.noninteractive` or `NSWritingToolsCoordinator.State.interactiveResting` state to determine the level of interactivity with your view.
- Use a transition to the `NSWritingToolsCoordinator.State.interactiveStreaming` state to start progress controls or otherwise indicate that Writing Tools is working on the request. In addition, use transitions to and from this state to implement undo coalescing. Specifically, start a new undo group on a transition to this state, and end that undo group on a transition away from this state. Notify Writing Tools of any undo stack changes by calling the `updateRange(_:with:reason:forContextWithIdentifier:)` method.

For more information about how to handle individual states, see the `NSWritingToolsCoordinator.State` type.

Inform the coordinator of external changes to your content

When Writing Tools is active, it tracks changes to the text in your view. If you make changes to your view's text storage while Writing Tools is active, let the system know immediately. Writing Tools tracks the changes it makes internally, so it needs to know about any external changes to make sure it delivers accurate information to your delegate object. To notify Writing Tools of any changes, use one of the following methods:

- Call the `updateRange(_:with:reason:forContextWithIdentifier:)` method if you change the text that corresponds to text in one of your context objects. Depending on the scope of your changes, Writing Tools might incorporate your changes or abort the current operation altogether.
- Call the `updateForReflowedTextInContextWithIdentifier(_:)` method to report any changes that affect your view's layout. For example, call this method if the size of your view changes, or if you change the text that precedes what's in one of your context objects. When you call this method, Writing Tools requests new previews, proofreading marks, and other layout-dependent information.

See Also

Writing Tools for custom views

Supporting Writing Tools via the pasteboard

Adopt a simplified version of the Writing Tools experience in a custom view using the pasteboard and macOS services.

`class NSWritingToolsCoordinator`

An object that manages interactions between Writing Tools and your custom text view.

`protocol Delegate`

An interface that you use to manage interactions between Writing Tools and your custom text view.

`class Context`

A data object that you use to share your custom view's text with Writing Tools.

`class AnimationParameters`

An object you use to configure additional tasks or animations to run alongside the Writing Tools animations.

{} Enhancing your custom text engine with Writing Tools

Add Writing Tools support to your custom text engine to enhance the text editing experience.