



UWP



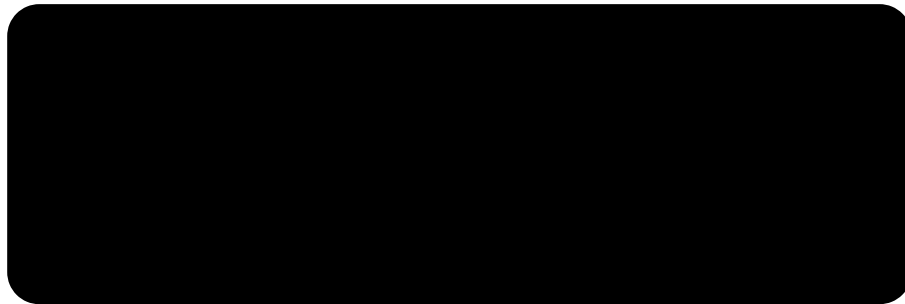
Create, edit, and remove



WINDOWS > GUIDES > ANNOTATIONS > CREATE, EDIT, AND REMOVE

Undo and redo annotations

Nutrient UWP SDK supports the undo and redo functionality when working with annotations. Users can undo and redo their changes using either the corresponding built-in toolbar items — undo and redo — or the standard shortcut key combinations `Control-Z` (undo), and `Control-Y` (redo).



The undo and redo toolbar buttons are hidden by default. To display them in the toolbar, use the `pdfView.SetToolbarItemsAsync()` API method.

To enable the undo and redo functionality, turn on the History API in one of the following ways:

- ❖ Set `pdfView.IsHistoryEnabled` to `true`.
- ❖ Call the `History.EnableAsync()` API method.

Once the History API is enabled, you can disable it in one of the following ways:

- ❖ Set `pdfView.IsHistoryEnabled` to `false`.
- ❖ Call the `History.DisableAsync()` API method.

This example enables the History API in XAML and adds the default undo and redo buttons in the toolbar. The `InitializationCompletedHandler` handler for



ASK AI

```

1 <ui:PdfView Name="PDFView"
2     Grid.Row="0" IsHistoryEnabled="True"
3     License="{StaticResource PSPDFKitLicense}"
4     PdfUriSource="ms-appx:///Assets/pdfs/PSPDFKit.pdf"
5     InitializationCompletedHandler="{x:Bind _viewModel.OnPDFViewInitiali

```

```

1 // In the `viewModel`.
2
3 internal async void OnPDFViewInitializationCompleted(PdfView sender, Document
4 {
5     IList<IToolbarItem> toolbarItems = sender.GetToolbarItems();
6     toolbarItems.Add(new UndoToolbarItem());
7     toolbarItems.Add(new RedoToolbarItem());
8
9     await sender.SetToolbarItemsAsync(toolbarItems);
10 }

```

Controlling undo and redo using the API

In addition to the dependency property mentioned above, the History API includes the following methods to perform different tasks related to this feature:

- ✧ `History.UndoAsync` — Rolls back the last annotation change — either a creation, an update, or a deletion — performed with the UI or the API. It returns a `Task<bool>` instance that resolves to `true` when the operation succeeds. Otherwise, it returns `false` — for example, when there are no operations left to be undone.

This example uses the API to create an annotation, moves it, and then undoes the move:

```

1 var annotation = new Note
2 {
3     Contents = "PSPDFKit is awesome!",
4     BoundingBox = new Rect(50,50,50,50)
5 };
6
7 await pdfView.Document.CreateAnnotationAsync(annotation);
8 Debug.WriteLine("Annotation created!");
9
10 createdAnnotation.BoundingBox = new Rect(100, 100, 50, 50);
11 await pdfView.Document.UpdateAnnotationAsync(createdAnnotation);
12 Debug.WriteLine("Annotation moved!");
13

```

```

14 await pdfView.Document.History.UndoAsync();
15 Debug.WriteLine("Annotation creation undone: annotation moved back to original

```

- ✦ `History.RedoAsync` — Repeats the annotation change rolled back by the last undo operation. It can be performed with either the UI or the API. It returns a `Task<bool>` instance that resolves to `true` when the operation succeeds. Otherwise, it returns `false` — for example, when there are no operations left to be done.

This example uses the API to create an annotation, delete it, undo the deletion, and delete it again by using the redo functionality:

```

1 var annotation = new Note
2 {
3     Contents = "PSPDFKit is awesome!",
4     BoundingBox = new Rect(50,50,50,50)
5 };
6 var createdAnnotation = await pdfView.Document.CreateAnnotationAsync(annotation);
7 Debug.WriteLine("Annotation created!");
8
9 await pdfView.Document.DeleteAnnotationAsync(createdAnnotation.Id);
10 Debug.WriteLine("Annotation deleted!");
11
12 await pdfView.Document.History.UndoAsync();
13 Debug.WriteLine("Annotation deletion undone: annotation restored!");
14
15 await pdfView.Document.History.RedoAsync();
16 Debug.WriteLine("Annotation deletion redone: annotation deleted!");

```



To redo an operation, an undo operation is required first.

- ✦ `History.CanUndoAsync` — Returns `true` when there are operations that can be undone. Otherwise, it returns `false`:

```

1 if (pdfView.Document.History.CanUndoAsync()) {
2     pdfView.Document.History.UndoAsync();
3 }

```

- ✦ `History.CanRedoAsync` — Returns `true` when there are operations that can be redone. Otherwise, it returns `false`:

```

1 if (pdfView.Document.History.CanRedoAsync()) {
2     pdfView.Document.History.RedoAsync();

```

❖ `History.ClearAsync` — Resets the list of possible operations to be undone and redone:

```

1  var history = pdfView.Document.History;
2  Debug.WriteLine(await history.CanUndoAsync()); // `false`
3  Debug.WriteLine(await history.CanRedoAsync()); // `false`
4
5  var annotation = new Note
6  {
7      Contents = "PSPDFKit is awesome!",
8      BoundingBox = new Rect(50, 50, 50, 50)
9  };
10 var createdAnnotation = await pdfView.Document.CreateAnnotationAsync(annotation);
11 Debug.WriteLine("Annotation created!");
12
13 await pdfView.Document.DeleteAnnotationAsync(createdAnnotation.Id);
14 Debug.WriteLine("Annotation deleted!");
15
16 Debug.WriteLine(await history.CanUndoAsync()); // `true`
17 Debug.WriteLine(await history.CanRedoAsync()); // `false`
18
19 await history.UndoAsync();
20
21 Debug.WriteLine(await history.CanUndoAsync()); // `true`
22 Debug.WriteLine(await history.CanRedoAsync()); // `true`
23
24 await history.ClearAsync();
25
26 Debug.WriteLine(await history.CanUndoAsync()); // `false`
27 Debug.WriteLine(await history.CanRedoAsync()); // `false`

```



Any creation, update, or deletion performed either with the UI or the API that isn't a result of undoing or redoing will clear the list of possible operations.

Tracking history events

Undo and redo operations can be tracked by listening to the following events:

- ❖ `UndoEvent` occurs after performing an undo operation, either with the API or with the UI.
- ❖ `RedoEvent` occurs after performing a redo operation, either with the API or with the UI.

- ⌘ `HistoryChanged` occurs after performing an undo or a redo operation, either with the API or with the UI.

The event handlers for the above events will receive a `History` object that raised the event, along with the `HistoryEventArgs` object:

```
1 public sealed class HistoryEventArgs
2 {
3     /// <summary>
4     /// Previous state of the annotation, or `null` if it's being restored.
5     /// </summary>
6     public IAnnotation Before { get; }
7
8     /// <summary>
9     /// Resulting state of the annotation, or `null` if it's being deleted.
10    /// </summary>
11    public IAnnotation After { get; }
12
13    /// <summary>
14    /// Action that resulted in the event i.e. `HistoryAction.Undo` or `HistoryAction.Redo`.
15    /// </summary>
16    public HistoryAction Action { get; }
17 }
```

- ⌘ `Cleared` occurs after the history is cleared by the `History.ClearAsync` API call.

Unlike the event handlers of other events, event handlers of the `Cleared` event only receive the corresponding `History` object, and the `EventArgs` parameters are set to `null`.

You can subscribe or unsubscribe to these events at any moment — even when the History API is disabled.

Exceptions

Here's a description of how the behavior of undo and redo may affect different elements of the SDK:

- ⌘ **Annotation Z Order** — The original annotation Z order won't be restored for annotations that are deleted and later restored with undo. Those annotations will be recreated in the foreground.
- ⌘ **Annotation Presets** — Annotation presets aren't affected by undo and redo operations, and they won't be restored to a previous or following state as a result of undo or redo.

Was this helpful?

✓ YES

✗ NO

Questions? [Contact us](#)

