Read and write data to the active selection in a document or spreadsheet

Article • 04/11/2023

The Document object exposes methods that let you read and write to the user's current selection in a document or spreadsheet. To do that, the `Document` object provides the `getSelectedDataAsync` and `setSelectedDataAsync` methods. This topic also describes how to read, write, and create event handlers to detect changes to the user's selection.

The `getSelectedDataAsync` method only works against the user's current selection. If you need to persist the selection in the document, so that the same selection is available to read and write across sessions of running your add-in, you must add a binding using the Bindings.addFromSelectionAsync method (or create a binding with one of the other "addFrom" methods of the Bindings object). For information about creating a binding to a region of a document, and then reading and writing to a binding, see Bind to regions in a document or spreadsheet.

Read selected data

The following example shows how to get data from a selection in a document by using the getSelectedDataAsync method.

JavaScript

```
Office.context.document.getSelectedDataAsync(Office.CoercionType.Text,
function (asyncResult) {
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
        write('Action failed. Error: ' + asyncResult.error.message);
    }
    else {
        write('Selected data: ' + asyncResult.value);
    }
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

In this example, the first parameter, *coercionType*, is specified as `Office.CoercionType.Text` (you can also specify this parameter by using the literal string `"text"`). This means that the value property of the AsyncResult object that is available

from the *asyncResult* parameter in the callback function will return a **string** that contains the selected text in the document. Specifying different coercion types will result in different values. Office.CoercionType is an enumeration of available coercion type values. `Office.CoercionType.Text` evaluates to the string "text".

> **Tip**
>
> **When should you use the matrix versus table coercionType for data access?** If you need your selected tabular data to grow dynamically when rows and columns are added, and you must work with table headers, you should use the table data type (by specifying the *coercionType* parameter of the `getSelectedDataAsync` method as `"table"` or `Office.CoercionType.Table`). Adding rows and columns within the data structure is supported in both table and matrix data, but appending rows and columns is supported only for table data. If you aren't planning on adding rows and columns, and your data doesn't require header functionality, then you should use the matrix data type (by specifying the *coercionType* parameter of `getSelectedDataAsync` method as `"matrix"` or `Office.CoercionType.Matrix`), which provides a simpler model of interacting with the data.

The anonymous function that is passed into the method as the second parameter, *callback*, is executed when the `getSelectedDataAsync` operation is completed. The function is called with a single parameter, *asyncResult*, which contains the result and the status of the call. If the call fails, the error property of the `AsyncResult` object provides access to the Error object. You can check the value of the Error.name and Error.message properties to determine why the set operation failed. Otherwise, the selected text in the document is displayed.

The AsyncResult.status property is used in the **if** statement to test whether the call succeeded. Office.AsyncResultStatus is an enumeration of available `AsyncResult.status` property values. `Office.AsyncResultStatus.Failed` evaluates to the string "failed" (and, again, can also be specified as that literal string).

Write data to the selection

The following example shows how to set the selection to show "Hello World!".

JavaScript

```
Office.context.document.setSelectedDataAsync("Hello World!", function
(asyncResult) {
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
```

```
        write(asyncResult.error.message);
    }
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

Passing in different object types for the *data* parameter will have different results. The result depends on what is currently selected in the document, which Office client application is hosting your add-in, and whether the data passed in can be coerced to the current selection.

The anonymous function passed into the setSelectedDataAsync method as the *callback* parameter is executed when the asynchronous call is completed. When you write data to the selection by using the `setSelectedDataAsync` method, the *asyncResult* parameter of the callback provides access only to the status of the call, and to the Error object if the call fails.

Detect changes in the selection

The following example shows how to detect changes in the selection by using the Document.addHandlerAsync method to add an event handler for the SelectionChanged event on the document.

JavaScript

```
Office.context.document.addHandlerAsync("documentSelectionChanged",
myHandler, function(result){}
);

// Event handler function.
function myHandler(eventArgs){
    write('Document Selection Changed');
}

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

The first parameter, *eventType*, specifies the name of the event to subscribe to. Passing the string `"documentSelectionChanged"` for this parameter is equivalent to passing the `Office.EventType.DocumentSelectionChanged` event type of the Office.EventType enumeration.

The `myHandler()` function that is passed into the method as the second parameter, *handler*, is an event handler that is executed when the selection is changed on the document. The function is called with a single parameter, *eventArgs*, which will contain a reference to a DocumentSelectionChangedEventArgs object when the asynchronous operation completes. You can use the DocumentSelectionChangedEventArgs.document property to access the document that raised the event.

> **Note**
>
> You can add multiple event handlers for a given event by calling the `addHandlerAsync` method again and passing in an additional event handler function for the *handler* parameter. This will work correctly as long as the name of each event handler function is unique.

Stop detecting changes in the selection

The following example shows how to stop listening to the Document.SelectionChanged event by calling the document.removeHandlerAsync method.

JavaScript

```
Office.context.document.removeHandlerAsync("documentSelectionChanged",
{handler:myHandler}, function(result){});
```

The `myHandler` function name that is passed as the second parameter, *handler*, specifies the event handler that will be removed from the `SelectionChanged` event.

> **Important**
>
> If the optional *handler* parameter is omitted when the `removeHandlerAsync` method is called, all event handlers for the specified *eventType* will be removed.

Use annotations in your Word add-in

Article • 04/30/2025

Includes community contributions from: Abdulhadi Jarad

You can use annotations to provide feedback about grammar or other aspects of content in a Word document. The user may see colorful underlining that indicates there's an issue or other information. If the user hovers over the affected content, a popup dialog is displayed that shows them what the issue is and possible actions they can take.

APIs for working with annotations were introduced in the PowerPointApi 1.7 requirement set and expanded in the PowerPointApi 1.8 requirement set as part of supporting writing assistance scenarios like checking spelling and grammar or providing suggestions to improve writing.

In this article, we show how your add-in can insert feedback and critiques using annotations in a document and allow the user to react to them.

> **Important**
>
> These annotations aren't persisted in the document. This means that when the document is reopened, the annotations need to be regenerated. However, if the user accepts suggested changes, the changes will persist as long as the user saves them before closing the document.

Prerequisites

The annotation APIs rely on a service that requires a Microsoft 365 subscription. As such, using this feature in Word with a one-time purchase license won't work. The user must be running Word connected to a Microsoft 365 subscription so that your add-in can successfully run the annotation APIs.

Key annotation APIs

The following are the key annotation APIs.

- Paragraph.insertAnnotations
- Paragraph.getAnnotations
- Objects:
  - Annotation: Represents an annotation.
  - AnnotationCollection: Represents the collection of annotations.

    - AnnotationSet: Represents the set of annotations produced by your add-in in this session.
    - CritiqueAnnotation: Represents the critique type of annotation.
    - Critique: Represents feedback about an affected area of a paragraph, indicated by a colored underline.
- Annotation events on Document:
    - onAnnotationClicked
    - onAnnotationHovered
    - onAnnotationInserted
    - onAnnotationPopupAction
    - onAnnotationRemoved

Use annotation APIs

The following sections show how to work with annotation APIs. These examples are based on the Manage annotations code sample.

Your add-in code should use the feedback or critique results from your service run against the user's content in the document to manage annotations more dynamically.

Register annotation events

The following code shows how to register event handlers. To learn more about working with events in Word, see Work with events using the Word JavaScript API. For examples of annotation event handlers, see the following sections.

TypeScript

```
let eventContexts = [];

async function registerEventHandlers() {
  // Registers event handlers.
  await Word.run(async (context) => {
    eventContexts[0] = context.document.onParagraphAdded.add(paragraphChanged);
    eventContexts[1] = context.document.onParagraphChanged.add(paragraphChanged);

    eventContexts[2] = context.document.onAnnotationClicked.add(onClickedHandler);
    eventContexts[3] = context.document.onAnnotationHovered.add(onHoveredHandler);
    eventContexts[4] =
context.document.onAnnotationInserted.add(onInsertedHandler);
    eventContexts[5] = context.document.onAnnotationRemoved.add(onRemovedHandler);
    eventContexts[6] =
context.document.onAnnotationPopupAction.add(onPopupActionHandler);

    await context.sync();
```

```
    console.log("Event handlers registered.");
  });
}
```

onClickedHandler event handler

The following code runs when the registered `onAnnotationClicked` event occurs.

TypeScript

```
async function onClickedHandler(args: Word.AnnotationClickedEventArgs) {
  // Runs when the registered Document.onAnnotationClicked event occurs.
  await Word.run(async (context) => {
    const annotation: Word.Annotation =
context.document.getAnnotationById(args.id);
    annotation.load("critiqueAnnotation");

    await context.sync();

    console.log(`AnnotationClicked: ID ${args.id}:`,
annotation.critiqueAnnotation.critique);
  });
}
```

onHoveredHandler event handler

The following code runs when the registered `onAnnotationHovered` event occurs.

TypeScript

```
async function onHoveredHandler(args: Word.AnnotationHoveredEventArgs) {
  // Runs when the registered Document.onAnnotationHovered event occurs.
  await Word.run(async (context) => {
    const annotation: Word.Annotation =
context.document.getAnnotationById(args.id);
    annotation.load("critiqueAnnotation");

    await context.sync();

    console.log(`AnnotationHovered: ID ${args.id}:`,
annotation.critiqueAnnotation.critique);
  });
}
```

onInsertedHandler event handler

The following code runs when the registered `onAnnotationInserted` event occurs.

TypeScript

```
async function onInsertedHandler(args: Word.AnnotationInsertedEventArgs) {
  // Runs when the registered Document.onAnnotationInserted event occurs.
  await Word.run(async (context) => {
    const annotations = [];
    for (let i = 0; i < args.ids.length; i++) {
      let annotation: Word.Annotation =
context.document.getAnnotationById(args.ids[i]);
      annotation.load("id,critiqueAnnotation");
      annotations.push(annotation);
    }

    await context.sync();

    for (let annotation of annotations) {
      console.log(`AnnotationInserted: ID ${annotation.id}:`,
annotation.critiqueAnnotation.critique);
    }
  });
}
```

onRemovedHandler event handler

The following code runs when the registered `onAnnotationRemoved` event occurs.

TypeScript

```
async function onRemovedHandler(args: Word.AnnotationRemovedEventArgs) {
  // Runs when the registered Document.onAnnotationRemoved event occurs.
  await Word.run(async (context) => {
    for (let id of args.ids) {
      console.log(`AnnotationRemoved: ID ${id}`);
    }
  });
}
```

onPopupActionHandler event handler

The following code runs when the registered `onAnnotationPopupAction` event occurs.

TypeScript

```
async function onPopupActionHandler(args: Word.AnnotationPopupActionEventArgs) {
  // Runs when the registered Document.onAnnotationPopupAction event occurs.
  await Word.run(async (context) => {
```

```
    let message = `AnnotationPopupAction: ID ${args.id} = `;
    if (args.action === "Accept") {
      message += `Accepted: ${args.critiqueSuggestion}`;
    } else {
      message += "Rejected";
    }

    console.log(message);
  });
}
```

Insert annotations

The following code shows how to insert annotations into the selected paragraph.

TypeScript

```
async function insertAnnotations() {
  // Adds annotations to the selected paragraph.
  await Word.run(async (context) => {
    const paragraph: Word.Paragraph =
context.document.getSelection().paragraphs.getFirst();
    const options: Word.CritiquePopupOptions = {
      brandingTextResourceId: "PG.TabLabel",
      subtitleResourceId: "PG.HelpCommand.TipTitle",
      titleResourceId: "PG.HelpCommand.Label",
      suggestions: ["suggestion 1", "suggestion 2", "suggestion 3"]
    };
    const critique1: Word.Critique = {
      colorScheme: Word.CritiqueColorScheme.red,
      start: 1,
      length: 3,
      popupOptions: options
    };
    const critique2: Word.Critique = {
      colorScheme: Word.CritiqueColorScheme.green,
      start: 6,
      length: 1,
      popupOptions: options
    };
    const critique3: Word.Critique = {
      colorScheme: Word.CritiqueColorScheme.blue,
      start: 10,
      length: 3,
      popupOptions: options
    };
    const critique4: Word.Critique = {
      colorScheme: Word.CritiqueColorScheme.lavender,
      start: 14,
      length: 3,
      popupOptions: options
    };
```

```
    const critique5: Word.Critique = {
      colorScheme: Word.CritiqueColorScheme.berry,
      start: 18,
      length: 10,
      popupOptions: options
    };
    const annotationSet: Word.AnnotationSet = {
      critiques: [critique1, critique2, critique3, critique4, critique5]
    };

    const annotationIds = paragraph.insertAnnotations(annotationSet);

    await context.sync();

    console.log("Annotations inserted:", annotationIds.value);
  });
}
```

Get annotations

The following code shows how to get annotations from the selected paragraph.

TypeScript

```
async function getAnnotations() {
  // Gets annotations found in the selected paragraph.
  await Word.run(async (context) => {
    const paragraph: Word.Paragraph =
context.document.getSelection().paragraphs.getFirst();
    const annotations: Word.AnnotationCollection = paragraph.getAnnotations();
    annotations.load("id,state,critiqueAnnotation");

    await context.sync();

    console.log("Annotations found:");

    for (let i = 0; i < annotations.items.length; i++) {
      const annotation: Word.Annotation = annotations.items[i];

      console.log(`ID ${annotation.id} - state '${annotation.state}':`,
annotation.critiqueAnnotation.critique);
    }
  });
}
```

Accept an annotation

The following code shows how to accept the first annotation found in the selected paragraph.

TypeScript

```
async function acceptFirst() {
  // Accepts the first annotation found in the selected paragraph.
  await Word.run(async (context) => {
    const paragraph: Word.Paragraph =
context.document.getSelection().paragraphs.getFirst();
    const annotations: Word.AnnotationCollection = paragraph.getAnnotations();
    annotations.load("id,state,critiqueAnnotation");

    await context.sync();

    for (let i = 0; i < annotations.items.length; i++) {
      const annotation: Word.Annotation = annotations.items[i];

      if (annotation.state === Word.AnnotationState.created) {
        console.log(`Accepting ID ${annotation.id}...`);
        annotation.critiqueAnnotation.accept();

        await context.sync();
        break;
      }
    }
  });
}
```

Reject an annotation

The following code shows how to reject the last annotation found in the selected paragraph.

TypeScript

```
async function rejectLast() {
  // Rejects the last annotation found in the selected paragraph.
  await Word.run(async (context) => {
    const paragraph: Word.Paragraph =
context.document.getSelection().paragraphs.getFirst();
    const annotations: Word.AnnotationCollection = paragraph.getAnnotations();
    annotations.load("id,state,critiqueAnnotation");

    await context.sync();

    for (let i = annotations.items.length - 1; i >= 0; i--) {
      const annotation: Word.Annotation = annotations.items[i];

      if (annotation.state === Word.AnnotationState.created) {
        console.log(`Rejecting ID ${annotation.id}...`);
        annotation.critiqueAnnotation.reject();

        await context.sync();
        break;
```

```
      }
    }
  });
}
```

Delete annotations

The following code shows how to delete all the annotations found in the selected paragraph.

TypeScript

```
async function deleteAnnotations() {
  // Deletes all annotations found in the selected paragraph.
  await Word.run(async (context) => {
    const paragraph: Word.Paragraph =
context.document.getSelection().paragraphs.getFirst();
    const annotations: Word.AnnotationCollection = paragraph.getAnnotations();
    annotations.load("id");

    await context.sync();

    const ids = [];
    for (let i = 0; i < annotations.items.length; i++) {
      const annotation: Word.Annotation = annotations.items[i];

      ids.push(annotation.id);
      annotation.delete();
    }

    await context.sync();

    console.log("Annotations deleted:", ids);
  });
}
```

Deregister annotation events

The following code shows how to deregister the event handlers using their event contexts you tracked in the `eventContext` variable.

TypeScript

```
async function deregisterEventHandlers() {
  // Deregisters event handlers.
  await Word.run(async (context) => {
    for (let i = 0; i < eventContexts.length; i++) {
      await Word.run(eventContexts[i].context, async (context) => {
        eventContexts[i].remove();
```

```
      });
    }

    await context.sync();

    eventContexts = [];
    console.log("Removed event handlers.");
  });
}
```

See also

- Annotation APIs need a Microsoft 365 subscription
- Manage annotations code sample
- Work with events using the Word JavaScript API