

Introduction

INVOX Medical must know how to perform write operations on an editor or any target element.

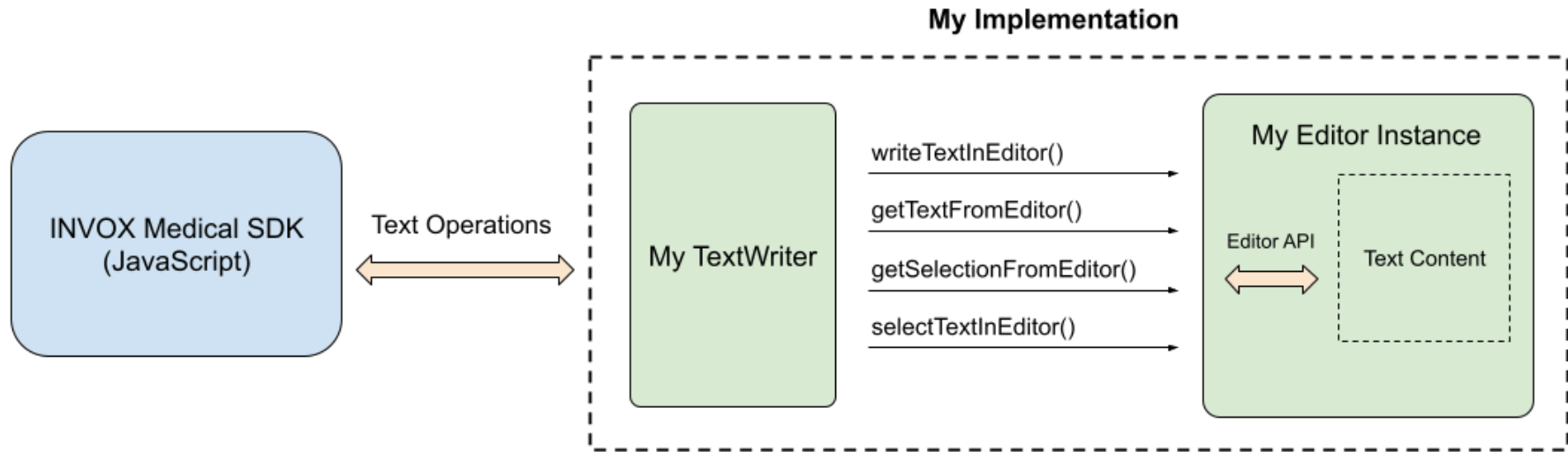


Figure 1: “Writer Schema”

The Writer or TextWriter **is responsible for performing the basic operations in a text editor**. It works as a proxy that references the editor instance and allows INVOX Medical to type the recognized text, move through the text using voice commands, select words or entire lines, delete text, redo, undo, etc.

Available TextWriters

Editor	TextWriter to use
TextArea	INVOX.TextAreaTextWriter
Others	Not implemented

If you use another type of editor, you must implement your own TextWriter.

Remember that the TextWriter is the entity that knows how the basic write/read operations work in a given editor, because each type of editor implements its own API. For example, the way of writing in a CKEditor V4 editor is different from writing in a TinyMCE editor.

Asynchronous Editors

INVOX Medical can adapt to the asynchronicity of a text editor based on the type of Writer implementation. If the writer's basic operations are synchronous, INVOX Medical will perform writing operations synchronously. However, if the implemented writer uses asynchronous techniques like Promises or Async/Await, the API for performing operations on the editor will follow the same behavior.

Asynchronous text editors use asynchronous programming techniques, such as Promises or Async/Await, to achieve this functionality. Instead of blocking the main browser thread to save changes to the server, AJAX requests are used to send the changes to the server and are then handled asynchronously in the background.

Some examples of asynchronous web text editors include CKEditor, TinyMCE, Quill, and Froala Editor. These libraries offer advanced text editing, customization, and the ability to save changes asynchronously.

Quick Start

To expedite your integration we provide you a template to implement your own TextWriter. Follow instructions below to set up the environment:

1. Choose the correct template and copy the full code in a JavaScript file.
2. Implement the essential methods and the optional extra methods.
3. Set up the writer via the INVOX Medical API.
4. Test functionality.

Choose a template

If you are unsure about the type of editor you are using, refer to [How to determine if my editor is synchronous or asynchronous](#).

Choose a template and implement its methods.

- Synchronous Editor Template.
- Asynchronous Editor Template.

Implement methods

To create your TextWriter, you must implement a series of methods that will wrap the editor, regardless of whether it is synchronous or asynchronous.

The following functions are the essential methods to be implemented.

Function	Purpose	Mandatory
writer.setEditor()	Handle one or several editor instances.	Yes
writer.getEditor()	Obtain the current editor instance.	Yes
writer.getText()	Obtain the full text from the editor.	Yes
writer.write()	Enable write operations in the editor.	Yes
writer.getSelection()	Obtain selections and the current caret position.	Yes
writer.setSelection()	Enable navigation commands through the text.	Yes

Function	Purpose	Mandatory
<code>writer.getTextContext()</code>	Enable write text with format.	Optional

In addition, you can complete the functionality with extra methods.

Function	Purpose	Mandatory
<code>writer.undo()</code>	Enable undo operation.	Optional
<code>writer.redo()</code>	Enable redo operation.	Optional
<code>writer.updateRedoUndoStack()</code>	Enable redo/undo operations management.	Optional
<code>writer.writeNewLine()</code>	Specify the editor new line format.	Optional
<code>writer.writeNewParagraph()</code>	Specify the editor new paragraph format.	Optional

Set up the Writer

The following API functions must be used after the user has logged in.

API Function	Description
<code>INVOX.SetTextWriter</code>	Allows you to set the specific TextWriter of an editor.
<code>INVOX.SetWriterTarget</code>	Allows you to set the target editor by its instance.

Specify the TextWriter and the editor instance to be used by INVOX Medical during the session. After establishing the writer and the editor to use, you will be able to perform write operations.

```
// Get editor instance
const yourEditorInstance = //...
// Import your Writer
const yourWriterImplementation = // ...

// Set the Writer
INVOX.SetTextWriter(yourWriterImplementation);

// Set the editor instance
INVOX.SetWriterTarget(yourEditorInstance);
```

How to determine if my editor is synchronous or asynchronous

To determine if a text editor library is synchronous or asynchronous, there are several ways to do it. The most common way is to review the library's documentation, since many times it is explicitly indicated if the library supports asynchronous programming. If the documentation doesn't mention it, you can also look in the documentation to see if there are any methods or properties that might indicate that asynchronous programming is being used.

Another way to determine if a library is synchronous or asynchronous is by reviewing the source code of the library. If the library uses asynchronous programming techniques, such as Promises or Async/Await, or even requests return a value using Callbacks, then it is most likely asynchronous. On the other hand, if these techniques are not used and write operations are performed sequentially, the library is likely to be synchronous.

If you still have doubts about whether the library is synchronous or asynchronous, you can always consult the library's technical support or the developer community for more information.

Example

If your editor returns a value from a callback, for example, to return the index of the cursor:

```
yourEditor.selection.getStart(function (currentSelectionStart) {  
    // This the callback function  
});  
  
yourEditor.selection.getEnd(function (currentSelectionEnd) {  
    // This the callback function  
});
```

Then you must use a the Asynchronous Writer to return a Promise with these values.

```
myWriter.getSelection = async function () {  
    const currentEditor = await this.getEditor()  
    // 1. Get editor selection range  
    const { start, end } = await getSelectionFromEditor(currentEditor)  
    // 2. Convert editor range into INVOX.Range  
    const range = new INVOX.Range(start, end)  
    return Promise.resolve(range)  
  
    async function getSelectionFromEditor(currentEditor) {  
        const start = await getSelectionStart(currentEditor)  
        const end = await getSelectionEnd(currentEditor)  
  
        return Promise.resolve({ start, end })  
    }  
}
```

```

function getSelectionStart(currentEditor) {
  return new Promise((resolve, reject) => {
    currentEditor.selection.getStart(function (currentSelectionStart) {
      resolve(currentSelectionStart)
    });
  })
}

function getSelectionEnd(currentEditor) {
  return new Promise((resolve, reject) => {
    currentEditor.selection.getEnd(function (currentSelectionEnd) {
      resolve(currentSelectionEnd)
    });
  })
}
}

```

Synchronous Editor Template

```

// Initialize the Synchronous TextWriter
var myWriter = new INVOX.TextWriterBase()

/* ----- ESSENTIAL METHODS ----- */

// Allows you to specify the current editor instance.
myWriter.setEditor = function(editorInstance) {
  if (!editorInstance) {
    throw "Parameter editorInstance cannot be null or undefined"
  }
  this.editor = editorInstance
}

// Allows you to get the current editor instance.
myWriter.getEditor = function() {
  return this.editor
}

```

```

// Allows you to get the editor text content.
myWriter.getText = function () {
    const currentEditor = this.getEditor()
    return getTextFromEditor(currentEditor)

    function getTextFromEditor(currentEditor) {
        let text = ""

        // TODO: Here goes your code...

        return text
    }
}

// Allows you to get current selection range.
myWriter.getSelection = function () {
    // 1. Get editor selection range.
    const currentEditor = this.getEditor().
    const { start, end } = getSelectionFromEditor(currentEditor)
    // 2. Convert editor range into INVOX.Range.
    return new INVOX.Range(start, end)

    function getSelectionFromEditor(currentEditor) {
        let start = 0
        let end = 0

        // TODO: Here goes your code...

        return { start, end }
    }
}

// Allows you to set current selection range.
myWriter.setSelection = function (range) {
    // 1. Get INVOX.Range values.
    const { start, end } = range
    // 2. Convert INVOX.Range into editor range.
    const currentEditor = this.getEditor()
    setSelectionInEditor(currentEditor, start, end)
}

```

```

    function setSelectionInEditor(currentEditor, start, end) {
        // TODO: Here goes your code...
    }
}

// Allows you to write in the editor where the caret is.
myWriter.write = function (text) {
    const currentEditor = this.getEditor()
    writeTextInEditor(currentEditor, text)

    function writeTextInEditor(currentEditor, text) {
        // TODO: Here goes your code...
    }
}

/*

IMPORTANT:
Implement only if you cannot implement getText() or getSelection().
The following code is the current implementation.

// Allows you to get the context around the caret position.
myWriter.getTextContext = function() {

    // By default this function calculate the context with getText() and getSelection().
    // If any of these functions could not be implemented correctly,
    // you must modify this function to give it the surrounding text.

    const CONTEXT_RANGE = 10 // Characters around caret position
    const currentText = this.getText()
    const selection = this.getSelection()
    const leftText = currentText.substr(0, selection.start).slice(-CONTEXT_RANGE)
    const rightText = currentText.substr(selection.end, CONTEXT_RANGE)
    return [leftText, rightText]
}

*/

/* ----- EXTRA METHODS ----- */

```

```

// Allows you to apply redo in the editor.
myWriter.redo = function () {
    const currentEditor = this.getEditor()
    applyRedo(currentEditor)

    function applyRedo(currentEditor) {
        // TODO: Here goes your code...
    }
}

// Allows you to apply undo in the editor.
myWriter.undo = function () {
    const currentEditor = this.getEditor()
    applyUndo(currentEditor)

    function applyUndo(currentEditor) {
        // TODO: Here goes your code...
    }
}

// Allows you to manage the status of redo/undo operations.
myWriter.updateRedoUndoStack = function () {
    // This function is only necessary if you have to manage
    // the status of redo/undo operations.
    const currentEditor = this.getEditor()
    updateRedoUndoStack(currentEditor)

    function updateRedoUndoStack() {
        // TODO: Here goes your code...
    }
}

// Allows you to specify the line break format by the editor used.
myWriter.writeNewLine = function () {
    // By default this function writes "\n".
    // Perhaps your editor uses <br> to break line:
    // this.write("<br>")

    // TODO: Here goes your code...
}

```



```

}

// Allows you to specify the paragraph break format by the editor used.
myWriter.writeNewParagraph = function () {
  // By default this function writes two line breaks "\n\n".
  // Perhaps your editor uses two <br> to define a paragraph:
  // this.write("<br><br>")

  // TODO: Here goes your code...
}

```

Asynchronous Editor Template

All the following methods must return a Promise.

```

// Initialize the Asynchronous TextWriter
var myWriter = new INVOX.TextWriterBaseAsync()

/* ----- ESSENTIAL METHODS ----- */

// Allows you to specify the current editor instance.
myWriter.setEditor = function(editorInstance) {
  if (!editorInstance) {
    throw "Parameter editorInstance cannot be null or undefined"
  }
  this.editor = editorInstance
  return Promise.resolve()
}

// Allows you to get the current editor instance.
myWriter.getEditor = function() {
  return Promise.resolve(this.editor)
}

// Allows you to get the editor text content.
myWriter.getText = async function () {
  const currentEditor = await this.getEditor()
  return getTextFromEditor(currentEditor)
}

```

```

function getTextFromEditor(currentEditor) {
    let text = ""

    // TODO: Here goes your code...

    return Promise.resolve(text)
}

// Allows you to get current selection range.
myWriter.getSelection = async function () {
    const currentEditor = await this.getEditor()
    // 1. Get editor selection range
    const { start, end } = await getTextFromEditor(currentEditor)
    // 2. Convert editor range into INVOX.Range
    const range = new INVOX.Range(start, end)
    return Promise.resolve(range)

    function getSelectionFromEditor(currentEditor) {
        let start = 0
        let end = 0

        // TODO: Here goes your code...

        return Promise.resolve({ start, end })
    }
}

// Allows you to set current selection range.
myWriter.setSelection = async function (range) {
    // 1. Get INVOX.Range values.
    const { start, end } = range
    // 2. Convert INVOX.Range into editor range.
    const currentEditor = await this.getEditor()
    return setSelectionInEditor(currentEditor, start, end)

    function setSelectionInEditor(currentEditor, start, end) {
        // TODO: Here goes your code...

        return Promise.resolve()
    }
}

```

```

    }
}

// Allows you to write in the editor where the caret is.
myWriter.write = async function (text) {
    const currentEditor = await this.getEditor()
    return writeTextInEditor(currentEditor, text)

    function writeTextInEditor(currentEditor, text) {
        // TODO: Here goes your code...

        return Promise.resolve()
    }
}

/*

IMPORTANT:
Implement only if you cannot implement getText() or getSelection().
The following code is the current implementation.

// Allows you to get the context around the caret position.
myWriter.getTextContext = async function() {

    // By default this function calculate the context with getText() and getSelection().
    // If any of these functions could not be implemented correctly,
    // you must modify this function to give it the surrounding text.

    const CONTEXT_RANGE = 10 // Characters around caret position
    const currentText = await this.getText()
    const selection = await this.getSelection()
    const leftText = currentText.substr(0, selection.start).slice(-CONTEXT_RANGE)
    const rightText = currentText.substr(selection.end, CONTEXT_RANGE)
    return Promise.resolve([leftText, rightText])
}
*/

```

```

/* ----- EXTRA METHODS ----- */

// Allows you to apply redo in the editor.
myWriter.redo = async function () {
  const currentEditor = await this.getEditor()
  return applyRedo(currentEditor)

  function applyRedo(currentEditor) {
    // TODO: Here goes your code...

    return Promise.resolve()
  }
}

// Allows you to apply undo in the editor.
myWriter.undo = async function () {
  const currentEditor = await this.getEditor()
  return applyUndo(currentEditor)

  function applyUndo(currentEditor) {
    // TODO: Here goes your code...

    return Promise.resolve()
  }
}

// Allows you to manage the status of redo/undo operations.
myWriter.updateRedoUndoStack = async function () {
  // This function is only necessary if you have to manage
  // the status of redo/undo operations.
  const currentEditor = await this.getEditor()
  return updateRedoUndoStack(currentEditor)

  function updateRedoUndoStack(currentEditor) {
    // TODO: Here goes your code...

    return Promise.resolve()
  }
}

```

```

// Allows you to specify the line break format by the editor used.
myWriter.writeNewLine = async function () {

    const currentEditor = await this.getEditor()
    const newLineInEditor = await getNewLineInEditor(currentEditor)
    return this.write(newLineInEditor)

    function getNewLineInEditor(currentEditor) {

        // By default this function writes "\n".
        // Perhaps your editor uses <br> to break line:
        // return this.write("<br>")

        const newLine = "\n"

        // TODO: Here goes your code...

        return Promise.resolve(newLine)
    }
}

// Allows you to specify the paragraph break format by the editor used.
myWriter.writeNewParagraph = async function () {

    const currentEditor = await this.getEditor()
    const newParagraphInEditor = await getNewParagraphInEditor(currentEditor)
    return this.write(newParagraphInEditor)

    function getNewParagraphInEditor(currentEditor) {

        // By default this function writes two line breaks "\n\n".
        // Perhaps your editor uses two <br> to define a paragraph:
        // this.write("<br><br>")

        const newParagraph = "\n\n"

        // TODO: Here goes your code...

        return Promise.resolve(newParagraph)
    }
}

```

```
}
```

Writer Methods

Use the API of INVOX Medical SDK to implement your own TextWriter.

Essential methods

setEditor()

Function	Description	Parameters	Returns (sync)	Returns (async)
setEditor(editorInstance)	Allows you to specify the current editor instance.	Object	void	Promise<void>

Description: This function allows you to set the editor instance that you use in your web application.

Parameters:

- **editorInstance:** **Object**. Instance of the editor which contains methods to handle its content.

Returns: No value returned.

Test Implementation: Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
INVOX.SetWriterTarget()	Sets the current editor instance. You can change between several instances of your editor.

getEditor()

Function	Description	Parameters	Returns (sync)	Returns (async)
getEditor()	Allows you to get the current editor instance.	-	Object	Promise<Object>

Description: This function allows you to get the editor instance that you use in your web application to use it in the following methods.

Parameters: No parameters required.

Returns:

- Editor: **Object**. Instance of the editor.

Test Implementation: Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
INVOX.GetWriterTarget()	Get the current editor instance.

getText()

Function	Description	Parameters	Returns (sync)	Returns (async)
getText()	Allows you to get the editor text content.	-	String	Promise<String>

Description: This function allows you to get the text content from the current editor.

Parameters: No parameters required.

Returns:

IMPORTANT: the value returned must be plain text and cannot contain format strings.

- Text: **String**. Text in plain from the current editor instance.

Test Implementation: Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
INVOX.GetText()	Returns the full plain text from the editor content.

write()

Function	Description	Parameters	Returns (sync)	Returns (async)
write(text)	Allows you to write in the editor where the caret is.	String	void	Promise<void>

Description: This function allows you to write text in the current editor at the caret position.

Parameters:

- *text*: **String**. Text to write in the current editor.

Returns: No value returned.

Test Implementation: Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
INVOX.Write()	Writes the text at the caret position.
INVOX.AppendText()	Writes the text at the end of the content.
INVOX.PrependText()	Writes the text at the beginning of the content.
INVOX.ClearText()	Clears the content of the editor.

getSelection()

Function	Description	Parameters	Returns (sync)	Returns (async)
getSelection()	Allows you to get the range of the current selection.	-	Object	Promise<Object>

Description: This function allows you to get a range of type **INVOX.Range** of the mouse selection in the current editor.

e.g: Consider ^ as the caret position, and [] the selection.

- EDITOR CONTENT: "^this is all text in the editor"
CALL: `getSelection()`
OUTPUT: `new INVOX.Range(0,0)`
- EDITOR CONTENT: "this [is all text] in the editor"
CALL: `getSelection()`
OUTPUT: `new INVOX.Range(5,16)`

Parameters: No parameters required.

Returns:

If there is no selection in the text, the return value is the current position of the caret.

- Range: **Object**. The range **INVOX.Range** object of the current selection.
 - start: **Number**. Start position in the current text.
 - end: **Number**. End position in the current text.

Test Implementation:

1. Select text with the mouse in the editor.
2. Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
INVOX.Write()	Writes the text at the position of the current selection.

setSelection()

Function	Description	Parameters	Returns (sync)	Returns (async)
setSelection(range)	Allows you to select text in the current editor.	Object	void	Promise<void>

Description: This function allows you to select the text in the current editor from a range of type **INVOX.Range**.

e.g: Consider `^` as the caret position, and `[]` the selection.

1. EDITOR CONTENT: "this is all text in the editor^"
CALL: `setSelection(new INVOX.Range(5,16))`
EDITOR CONTENT: "this [is all text] in the editor"
2. EDITOR CONTENT: "this is all text in the editor^"
CALL: `setSelection(new INVOX.Range(0,0))`
EDITOR CONTENT: "^this is all text in the editor"

Parameters:

If the values of start and end are the same, then you are setting the position of the caret.

- **range**: Object. The range `INVOX.Range` object of the text you want to select.
 - start: **Number**. Start position in the current text.
 - end: **Number**. End position in the current text.

Returns: No value returned.

Test Implementation: Use the Writer Functions from the API of INVOX Medical SDK:

API Function	Expected behaviour
<code>INVOX.SetSelection()</code>	Select the range of text passed as a parameter.

`getTextContext()`

If you already have the `getSelection()` and `getText()` methods done then **you can skip this implementation**.

Function	Description	Parameters	Returns (sync)	Returns (async)
<code>getTextContext()</code>	Allows INVOX Medical to write text with format.	-	<code>Array<String, String></code>	<code>Promise<Array<String, String>></code>

Description: This function allows INVOX Medical to decide whether to include spaces or capital letters when typing recognized text in the current editor.

e.g: consider `^` as the caret position, and `[]` the selection.

- EDITOR CONTENT: "this is all text in the editor^"
 CALL: `getTextContext()`
 OUTPUT: `["the editor", ""]`
- EDITOR CONTENT: "this [is all text] in the editor"
 CALL: `getTextContext()`
 OUTPUT: `["this ", " in the ed"]`

Parameters: No parameters required.

Returns:

- Array. **Array**. Context around the caret position.
 - leftText: **String**. Left text from caret position.
 - rightText: **String**. Right text from caret position.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Check that after dictating a dot character `.`, the next word is capitalized.

Extra methods**undo()**

Function	Description	Parameters	Returns (sync)	Returns (async)
undo()	Allows INVOX Medical to use undo voice command in the editor.	-	void	Promise<void>

Description: This function allows INVOX Medical to take advantage of the undo editor feature and use it in a voice command.

Parameters: No parameters required.

Returns: No value returned.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Dictate “undo” in your language and the command should work.

redo()

Function	Description	Parameters	Returns (sync)	Returns (async)
redo()	Allows INVOX Medical to use redo voice command in the editor.	-	void	Promise<void>

Description: This function allows INVOX Medical to take advantage of the redo editor feature and use it in a voice command.

Parameters: No parameters required.

Returns: No value returned.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Dictate “redo” in your language and the command should work.

updateRedoUndoStack()

This method should only be implemented if the editor you use does not implement a redo/redo operations. You can skip this implementation in this case.

Function	Description	Parameters	Returns (sync)	Returns (async)
updateRedoUndoStack()	Allows INVOX Medical to use redo/undo operations.	-	void	Promise<void>

Description: This function allows INVOX Medical to take advantage of the redo and undo editor features and use it in voice commands.

Parameters: No parameters required.

Returns: No value returned.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Dictate “redo” and “undo” in your language several times and the command should work.

writeNewLine()

Function	Description	Parameters	Returns (sync)	Returns (async)
writeNewLine()	Allows you to specify the line break format.	-	void	Promise<void>

Description: This function allows you to specify the line break format by the editor used.

Parameters: No parameters required.

Returns: No value returned.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Dictate “new line” in your language and the command should write a new line in the specified format.

`writeNewParagraph()`

Function	Description	Parameters	Returns (sync)	Returns (async)
<code>writeNewParagraph()</code>	Allows you to specify the new paragraph format.	-	<code>void</code>	<code>Promise<void></code>

Description: This function allows you to specify the paragraph break format by the editor used.

Parameters: No parameters required.

Returns: No value returned.

Test Implementation:

1. Start dictation with `INVOX.SetDictationRunning()`.
2. Dictate “new paragraph” in your language and the command should write a new paragraph in the specified format.

Usage Example

The **TextAreaTextWriter** implementation is provided to write in TextArea elements. The TextWriter must be established as follows **after session starts** in order to tell INVOX Medical which writer to use.

Use the API to call this writer: `INVOX.TextAreaTextWriter`.

1. Assign the TextWriter to be used.

```
INVOX.SetTextWriter(INVOX.TextAreaTextWriter);
```

2. Create the target or targets to be handled by INVOX Medical.

```
<textarea id="inbox-textarea-1" class="form-control" placeholder="Recognized text..."></textarea>
<textarea id="inbox-textarea-2" class="form-control" placeholder="Recognized text..."></textarea>
```

3. Indicate the TextArea element to become the current target. **The TextAreaTextWriter uses the element identifier to `setEditor()` and not the editor instance.**

```
const editorInstance = document.querySelector("#inbox-textarea-1");  
INVOX.SetWriterTarget(editorInstance);
```

4. To switch to the other WriterTarget, use the previous function with the new identifier. It would no longer be necessary to indicate the TextWriter as they are of the same type.

```
const editorInstance = document.querySelector("#inbox-textarea-2");  
INVOX.SetWriterTarget(editorInstance);
```

And that's it!