SmartOffice Documentation

Release 3.15.0

Artifex

Mar 04, 2024

CONTENTS

CHAPTER

ONE

QUICK START GUIDE

SmartOffice enables quick and easy document viewing and editing for *Android* and *iOS* platforms. This guide demonstrates how our *Drop-in UI* solution allows you to get up and running with minimal coding effort.

Note: The *Drop-in UI* connects with the native file browser on your *Android* and *iOS* platforms for ease of integration and to enable the best possible document management experience.

1.1 Android

After following the integration steps in *Getting Started for Android* use the *SmartOffice* drop-in activity, `DefaultOfficeUIActivity`, with data representing your file URI and start it:

Kotlin

Java

```
val intent = Intent(this, DefaultOfficeUIActivity::class.java)
intent.data = <your_file_uri>
startActivity(intent)
```

```
Intent intent = new Intent(this, DefaultOfficeUIActivity.class);
intent.setData(<your_file_uri>);
startActivity(intent);
```

1.2 iOS

After following the integration steps in *Getting Started for iOS* use the *SmartOffice* drop-in view controller, `SODKDefaultUIViewController`, with your file URL and push it from your navigation controller:

Swift

Objective-C

```
SODKDefaultUIViewController.viewController(url: <your_file_url>, whenReady:
↪{(docVC:ARDKDefaultUIViewController?) in
   if docVC != nil {
```

(continues on next page)

(continued from previous page)

```
        self.navigationController?.pushViewController(docVC!, animated: true)
    }
})
```

```
[SODKDefaultUIViewController viewControllerWithUrl:<your_file_url> whenReady:^
↪(ARDKDefaultUIViewController *docVC) {
    if (docVC) {
        [self.navigationController pushViewController:docVC animated:YES];
    }
}];
```

CHAPTER

TWO

ABOUT OUR SDKS

SmartOffice is a software framework for viewing, editing and converting *Word* (doc/x), *Presentation* (ppt/x), *Spreadsheet* (xls/x) and *PDF* (pdf) documents. The developer documentation outlines how to use the SDK for *Android*, *iOS*, *Mac* and *Windows* platforms.

All *Office* versions since 1997 are supported — *Word*, *PowerPoint* and *Excel* 97, 2000, XP, 2003, 2007, 2010, 2013, 2016 — doc, docx, xls, xlsx, ppt, pptx.

2.1 Licensing

Before using *SmartOffice*, please make sure that you have a valid commercial license to do so.

2.1.1 Commercial license

Please contact our sales team to discuss a commercial license. Each Artifex commercial license is crafted based on your individual use case.

CHAPTER

THREE

ANDROID SDK

3.1 Getting Started

3.1.1 System requirements

- Windows, Mac or Linux
- Android Studio

3.1.2 Adding the SDK to your project

In order to use the *SmartOffice* SDK in your app you need to use the Gradle build system to import the library dependencies.

SmartOffice libraries can be retrieved as pre-built artifacts from a local `app/libs` folder location relative to your Android project. Your Android project's module `build.gradle` should add this location.

You should have been supplied library items specific to *SmartOffice* within `libs` as follows:

- `editor.aar`
- `mupdf.aar`
- `phoenix.aar`
- `sodk_resources.aar`
- `solib.aar`
- `wheel.aar`

Your gradle file dependencies

To add the libraries to your project add the dependencies section in your Module `build.gradle` as follows:

```
implementation fileTree(dir: 'libs', include: ['*.aar'])
```

As well as these standaard *SmartOffice* libraries your gradle dependencies file will also need the following:

```
implementation 'androidx.navigation:navigation-fragment-ktx:2.5.1'
implementation 'androidx.navigation:navigation-ui-ktx:2.5.1'
```

Note: The exact version of the `androidx.navigation` modules may depend on your other dependency versions.

3.1.3 Verify your integration

To ensure you have correctly added the libraries project, synchronize your project to ensure that any changes to your `build.gradle` are applied and then ensure that your source code files can reference `import com.artifex.solib` without error.

Kotlin

Java

```
import com.artifex.solib.*
```

```
import com.artifex.solib.*;
```

Check the version

Once you have successfully integrated the libraries the build version details can be checked from an `Activity` as follows:

Kotlin

Java

```
val versionInfo:Array<String>? = SOLib.getLib(activity).versionInfo

if (versionInfo != null) {
    val releaseDate = versionInfo[0]
    val issue = versionInfo[1]
    val version = versionInfo[2]
    val customer = versionInfo[3]
}
```

```
String[] versionInfo = SOLib.getLib(activity).getVersionInfo();

if (versionInfo != null) {
   String releaseDate = versionInfo[0];
   String issue = versionInfo[1];
   String version = versionInfo[2];
   String customer = versionInfo[3];
}
```

3.1.4 Configuration Options

Before loading a document *SmartOffice* configuration options must be defined for your application. Without these being defined the application will throw an exception.

Early on in the lifecycle of your app instantiate configuration options as follows:

Kotlin

Java

```kotlin
import com.artifex.solib.ConfigOptions

val configOptions: ConfigOptions = ConfigOptions()
ArDkLib.setAppConfigOptions(configOptions)
```

```java
import com.artifex.solib.ConfigOptions;

ConfigOptions configOptions = new ConfigOptions();
ArDkLib.setAppConfigOptions(configOptions);
```

Configuration options can be tailored to suit your application requirements. To do so set the required `Boolean` value against the *Configuration option variable* you need.

Available configuration options

Note some of settings will *only* apply to the visibility of UI buttons in the *Default UI*.

Variable	Type	Only for Default UI	Notes
`isEditingEnabled`	`Boolean`	*no*	Enable or disable editing.
`isSaveAsEnabled`	`Boolean`	*yes*	Enable or disable save as.
`isSaveAsPdfEnabled`	`Boolean`	*yes*	Enable or disable save as PDF.
`isOpenInEnabled`	`Boolean`	*yes*	Enable or disable open in.
`isOpenPdfInEnabled`	`Boolean`	*yes*	Enable or disable open as a PDF.
`isShareEnabled`	`Boolean`	*yes*	Enable or disable sharing.
`isExtClipboardInEnabled`	`Boolean`	*no*	Enable or disable clipboard paste.
`isExtClipboardOutEnabled`	`Boolean`	*no*	Enable or disable clipboard copy.
`isPrintingEnabled`	`Boolean`	*yes*	Enable or disable printing.
`isLaunchUrlEnabled`	`Boolean`	*no*	Enable or disable external URL links from operating.
`isDigitalSignaturesEnabled`	`Boolean`	*yes*	Enable or disable digital signatures.
`isESignaturesEnabled`	`Boolean`	*yes*	Enable or disable e-signatures.
`isFormFillingEnabled`	`Boolean`	*no*	Enable or disable form filling.
`isFormSigningFeatureEnabled`	`Boolean`	*no*	Enable or disable form signing.
`isRedactionsEnabled`	`Boolean`	*no*	Enable or disable redactions.
`isFullscreenEnabled`	`Boolean`	*yes*	Enable or disable fullscreen mode.
`isInvertContentInDarkModeEnabled`	`Boolean`	*no*	Enable or disable dark mode content inversion.

3.2 Document Viewing

Ensuring that you have added the *SmartOffice* libraries to your project and have defined your *Configuration Options* then you should be ready to view a document.

The first consideration which should be taken into account before opening a document whether you are interested in building your own user interface (UI) to control the document view or not. If you do want to create your own UI then you should reference the *Custom UI* section, otherwise you should look at the *Default UI* implementation within this document.

3.2.1 Default UI

Implementing a *Default UI* is the simplist solution for using *SmartOffice* in your project. The UI controls for all the common document features will be provided within a pre-packaged user interface made by *Artifex*.

To create an `Activity` which employs the *Default UI* do the following:

1. Create a new blank `Activity` in your *Android* project.
2. Edit the `Activity` to extend the *SmartOffice* class `DefaultOfficeUIActivity`.
3. Instantiate the `Activity` as an `Intent` with the `data` object set to the file `Uri` of the document which you want to view.
4. Start the `Activity`.

This should then launch your `Activity` and display the chosen file along with the pre-packaged user interface.

Example code for your `Activity`:

Kotlin

Java

```
import com.artifex.sonui.phoenix.DefaultOfficeUIActivity

class MyDefaultUIActivity : DefaultOfficeUIActivity() {

}
```

```
import com.artifex.sonui.phoenix.DefaultOfficeUIActivity;

public class MyDefaultUIActivity extends DefaultOfficeUIActivity {

}
```

Example code for launching your `Activity`:

Kotlin

Java

```
val intent = Intent(this, MyDefaultUIActivity::class.java)
intent.data = <your_file_uri>
startActivity(intent)
```

```
Intent intent = new Intent(this, MyDefaultUIActivity.class);
intent.setData(<your_file_uri>);
startActivity(intent);
```

Default UI Customization

Adjusting Graphical Assets

It is possible to supply your own graphical assets for the icons in the *Default UI*. There are a few ways of doing this:

- Rebuild the `phoenix.aar` library for your project. (If you choose this route then you will need to update the gradle dependencies in the `phoenix` Android Studio project to ensure it can be built).
- By adding your own versions of the asset names to your project. In *Android Studio* do: **File > New > Vector Asset** and select the new `svg` asset you require. Simply rename it with the name of the asset you need to replace.

Note: To understand the names of the assets you might need to replace look at the `phoenix/src/main/res/drawable` location

---

- Take the provided library file `phoenix.aar` apart, then swap in the required resources, and then put it back together.

Unlimited customization of the *Default UI*

The *Default UI* can be rebuilt as a library for you project by republishing the related library `phoenix.aar` with any amends you require. In this way there is unlimited customization of the *Default UI* to suit your requirements.

Open the following project in *Android Studio*:

`phoenix`

Note: You will need to relink the library dependencies in the main project in order for your gradle build to synchronize without error.

3.2.2 Custom UI

If you choose to implement your own *Custom UI* then it is up to you to define as much (or as little!) UI as you wish.

You should create your own class with an associated layout XML area for the document view. A typical `AppCompatActivity` with an associated layout is a typical use case.

Defining the document view area

You need to define a dedicated holding area for your `DocumentView`. This "holding" view will attach a dedicated `DocumentView` which is the actual view where documents will be displayed and interacted with. For example if your *Custom UI* `Activity` is called `CustomUIActivity` then your associated layout XML may be the following:

```
<?xml version="1.0" encoding="utf-8"?>
    <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
↪android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".CustomUIActivity">

      <RelativeLayout
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:id="@+id/document_view_holder">
      </RelativeLayout>

</androidx.constraintlayout.widget.ConstraintLayout>
```

In the case above your *Custom UI* is simply a full screen view of a document.

Your Custom UI Activity

At a minimum an application developer requires to do the following in their `Activity`:

1. *Create the document* view with the correct file type.
2. *Add the document view* to the document view holder.
3. *Set default configuration options* for the document view.
4. *Start the document view* with the required file `Uri`.

1. Create the document view

Create the correct `DocumentView` for the file type:

Kotlin

Java

```
val filetype = FileUtils.getFileTypeExtension(this, <your_file_uri>)
documentView = DocumentView.create(this, filetype)
```

```
String filetype = FileUtils.getFileTypeExtension(this, <your_file_uri>);
documentView = DocumentView.create(this, filetype);
```

create(*activity*, *filetype*)

Returns the correct `DocumentView` for the document file type.

Note: This is a **static** method.

Parameters

- **activity** – Activity.
- **filetype** – String.

Returns

DocumentViewDoc | DocumentViewPpt | DocumentViewXls | DocumentViewPdf | DocumentView.

Note: There should only be one instance of DocumentView otherwise an error will be thrown.

2. Add the document view

Get your *document view holding area* and add the newly created DocumentView instance to it:

Kotlin

Java

```
val parent = findViewById<ViewGroup>(R.id.document_view_holder)
parent.addView(
    documentView,
    RelativeLayout.LayoutParams(
        ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT
    )
)
```

```
ViewGroup parent = findViewById(R.id.document_view_holder);
parent.addView(
    documentView,
    new RelativeLayout.LayoutParams(
        ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT
    )
);
```

3. Set default configuration options

Unlike with the *Default UI*, a *Custom UI* Activity needs to set default configuration options *against* the document view instance as follows:

Kotlin

Java

```
documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions())
```

```
documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions());
```

setDocConfigOptions(*options*)

Sets the configuration options for the `DocumentView`.

Parameters

options – `ConfigOptions`.

4. Start the document view

Finally, start the document view with the file `Uri` as follows:

Kotlin

Java

```
documentView.start(<your_file_uri>, 0, false)
```

```
documentView.start(<your_file_uri>, 0, false);
```

start(*file*, *page*, *showUI*)

Parameters

- **file** – `Uri` of file.
- **page** – `Int` referencing start page of document (zero-indexed, i.e. `0` will start the document on page 1).
- **showUI** – `Boolean` if set to true this will show the *Default UI*, note: this is *not* something we want to do if we are creating a *Custom UI*.

Putting it all together your *Custom UI* `Activity` may look like this:

Kotlin

Java

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.artifex.solib.ArDkLib
import com.artifex.sonui.editor.DocumentView
import com.artifex.solib.ConfigOptions
import com.artifex.solib.FileUtils

class CustomUIActivity : AppCompatActivity() {

    lateinit var documentView: DocumentView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_custom_ui)

        // 1. create the document view
        val filetype = FileUtils.getFileTypeExtension(this, <your_file_uri>)
```

(continues on next page)

(continued from previous page)

```
        documentView = DocumentView.create(this, filetype)

        // 2. add the document view
        val parent = findViewById<ViewGroup>(R.id.document_view_holder)
        parent.addView(
            documentView,
            RelativeLayout.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                ViewGroup.LayoutParams.MATCH_PARENT
            )
        )

        // 3. set the ConfigOptions
        documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions())

        // 4. start the document view, specifying showUI = false (otherwise it would␣
↪show the default ui)
        documentView.start(<your_file_uri>, 0, false)
    }

}
```

```
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import com.artifex.solib.ArDkLib;
import com.artifex.sonui.editor.DocumentView;
import com.artifex.solib.ConfigOptions;
import com.artifex.solib.FileUtils;

class CustomUIActivity extends AppCompatActivity {

    private DocumentView documentView = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_custom_ui);

        // 1. create the document view
        String filetype = FileUtils.getFileTypeExtension(this, <your_file_uri>);
        documentView = DocumentView.create(this, filetype);

        // 2. add the document view
        ViewGroup parent = findViewById(R.id.document_view_holder);
        parent.addView(
            documentView,
            new RelativeLayout.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                ViewGroup.LayoutParams.MATCH_PARENT
            )
        );
```

(continues on next page)

(continued from previous page)

```
        // 3. set the ConfigOptions
        documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions());

        // 4. start the document view, specifying showUI = false (otherwise it would␣
↪show the default ui)
        documentView.start(<your_file_uri>, 0, false);
    }

}
```

Now that you have a *Custom UI* displaying a document, the next stage is to look at *Document Listeners* to discover how to listen to document events.

3.3 Document Listeners

Document listeners should only be required when using the *Custom UI* as the application developer is responsible for providing their own UI to manage relevant document events.

3.3.1 DocumentListener

To set a *document listener* you should invoke the `setDocumentListener` method on the `DocumentView` instance with a defined `DocumentListener` object.

Kotlin

Java

```
documentView.setDocumentListener(object : DocumentListener {
   override fun onPageLoaded(pagesLoaded: Int) {

   }

   override fun onDocCompleted() {

   }

   override fun onPasswordRequired() {

   }

   override fun onViewChanged(scale: Float, scrollX: Int, scrollY: Int, selectionRect:␣
↪Rect?) {

   }
}
```

```
documentView.setDocumentListener(new DocumentListener() {
   @Override
   public void onPageLoaded(int pagesLoaded) {

   }

   @Override
   public void onDocCompleted() {

   }

   @Override
   public void onPasswordRequired() {

   }

   @Override
   public void onViewChanged(float scale, int scrollX, int scrollY, Rect selectionRect) {

   }
});
```

Available methods are as follows:

onPageLoaded(*pagesLoaded: Int*)

Called when pages are loaded.

Parameters

pagesLoaded – `Int` Represents number of pages loaded.

onDocCompleted()

Called when the document has completely loaded.

onPasswordRequired()

Called when a password is required by the document.

onViewChanged(*scale: Float*, *scrollX: Int*, *scrollY: Int*, *selectionRect: Rect?*)

Called when the scale, scroll, or selection in the document changes.

Parameters

- **scale** – `Float` Represents the zoom scale of the document.
- **scrollX** – `Int` Represents the x position of the document.
- **scrollY** – `Int` Represents the y position of the document.
- **selectionRect** – (optional) `android.graphics.Rect` Represents any current selection area on the document.

3.3.2 DocStateListener

To set a *document state listener* you should invoke the `setDocStateListener` method on the `DocumentView` instance with a defined `DocStateListener` object.

Kotlin

Java

```
documentView.setDocStateListener(object : DocStateListener {
    override fun docLoaded() {

    }

    override fun done() {

    }
})
```

```
documentView.setDocStateListener(new DocumentView.DocStateListener() {
    @Override
    public void docLoaded() {

    }

    @Override
    public void done() {

    }
});
```

Available methods are as follows:

docLoaded()
: Called when the document is fully loaded.

done()
: Called when the `DocumentView` is closed.

> **Note:** Typically you'll want to close your activity (i.e. `super.finish()`) on this event.

3.3.3 setOnUpdateUI

Use this method to set up a runnable to handle UI updates triggers by `DocumentView`.

Kotlin

Java

```
documentView.setOnUpdateUI(Runnable {

})
```

```
documentView.setOnUpdateUI(new Runnable() {
   @Override
   public void run() {

   }
});
```

Important: This method will only register once the document has fully loaded - therefore a developer should register for these events after the *docLoaded()* method is triggered.

3.3.4 setPageChangeListener

Called on a page change event - i.e. when the document has scrolled or jumped to another page.

Kotlin

Java

```
documentView.setPageChangeListener { pageNumber ->

}
```

```
documentView.setPageChangeListener(new DocumentView.ChangePageListener() {
    @Override
    public void onPage(int pageNumber) {

    }
});
```

Important: This method will only register once the document has fully loaded - therefore a developer should register for these events after the *docLoaded()* method is triggered.

3.4 Activity Interfaces

There are four *optional* interfaces that can be implemented via your own custom classes in order to define how *SmartOffice* manages data security.

- *SODataLeakHandlers*
- *SOPersistentStorage*
- *SOClipboardHandler*
- *SOSecureFS*

Note: These interfaces are optional and there are no defaults for these interfaces.

These interfaces are specifically important when considering a *Custom UI* approach.

With class implementations following these interfaces, a developer can implement security where a user's data might flow into, or out of, the application.

The following code example (which assumes your own implementations of these classes) could be invoked at the start of your main activity as part of your setup:

Kotlin

Java

```
import com.artifex.solib.*
import com.artifex.sonui.editor.Utilities

public override fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)

    Utilities.setDataLeakHandlers(MyOwnDataLeakHandlers())
    Utilities.setPersistentStorage(MyOwnPersistentStorage())
    ArDkLib.setClipboardHandler(MyOwnClipboardHandler())
    ArDkLib.setSecureFS(MyOwnSecureFS())

    ...
}
```

```
import com.artifex.solib.*;
import com.artifex.sonui.editor.Utilities;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Utilities.setDataLeakHandlers(new MyOwnDataLeakHandlers());
    Utilities.setPersistentStorage(new MyOwnPersistentStorage());
    ArDkLib.setClipboardHandler(new MyOwnClipboardHandler());
    ArDkLib.setSecureFS(new MyOwnSecureFS());

    ...
}
```

3.4.1 SODataLeakHandlers

An interface that specifies the basis for implementing a class to provide hooks for an app to control file saving and other functions.

initDataLeakHandlers(*activity*, *cfgOpts*)

This method initialises the `DataLeakHandlers` object.

Parameters

- **activity** – `Activity` The current activity.
- **cfgOpts** – `ConfigOptions` The *Configuration Options* to reference.

finaliseDataLeakHandlers()

This method finalizes the `DataLeakHandlers` object.

pauseHandler(*doc*, *hasModifications*, *whenDone*)

This method will be called when the application is backgrounded.

The implementor may save the document to a temporary location to allow restoration on a future run.

Parameters

- **doc** – ArDkDoc The document object.
- **hasModifications** – Boolean True if the document has unsaved modifications.
- **whenDone** – Runnable Run when everything is done. Can be null.

printHandler(*session*)

This method will be called when the application "Print" button is pressed.

The implementor should save the document, most likely as PDF, then use the available API's to print the PDF document.

Parameters

session – SODocSession The document session.

launchUrlHandler(*url*)

This method will be called when an external link has been activated. The implementor should launch an application to display the link target.

Parameters

url – String The Url to be launched.

customSaveHandler(*filename*, *doc*, *customDocData*, *completionCallback*)

This method will be called when the application custom save button is pressed.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.
- **customDocData** – String Custom data passed in the document open intent.
- **completionCallback** – SOCustomSaveComplete The method to be called once the operation is complete.

saveAsHandler(*filename*, *doc*, *completionCallback*)

This method will be called when the application "SaveAs" button is pressed. The implementor should allow the user to select the file save location, execute the save then inform application of the new location or failure/cancellation of the operation via completionCallback.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.
- **completionCallback** – SOSaveAsComplete The method to be called once the operation is complete.

saveAsSecureHandler(*filename*, *doc*, *resolution*, *language*, *completionCallback*)

This method will be called when the application "Save Secure" button is pressed.

The implementor should allow the user to select the file save location, execute the save then inform application of the new location or failure/cancellation of the operation via completionCallback.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.
- **resolution** – String The resolution to be used.
- **language** – String The language to be used.
- **completionCallback** – SOSaveAsComplete The method to be called once the operation is complete.

postSaveHandler(*completionCallback*)

This method will be called after the file is saved internally. The implementation can be empty.

Parameters

completionCallback – SOSaveAsComplete The method to be called once the operation is complete.

exportPdfAsHandler(*filename*, *exportFormat*, *doc*)

This method will be called when the application "ExportPdfAs" button is pressed.

Parameters

- **filename** – String The name of the file being edited.
- **exportFormat** – String The format of the exported document. Currently only "docx" and "txt" are available.
- **doc** – ArDkDoc The document object.

saveAsPdfHandler(*filename*, *doc*)

This method will be called when the application "SavePdfAs" button is pressed.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.

openInHandler(*filename*, *doc*)

This method will be called when the application "OpenIn" button is pressed.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.

openPdfInHandler(*filename*, *doc*)

This method will be called when the application "OpenPdfIn" button is pressed.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.

shareHandler(*filename*, *doc*)

This method will be called when the application "Share" button is pressed. The implementor should save the document to a suitable location suitable for sharing in a partner application.

Parameters

- **filename** – String The name of the file being edited.
- **doc** – ArDkDoc The document object.

getImageFromUser(*listener*)

This method will be called when the application "Insert Image" button is pressed.

The implementor may allow the user to select an image for insertion into the document. If so, then call `listener.done(true, Uri)` and pass a `Uri` identifying the selected file.

If not, call `listener.done(false, null)`, and a default image selection method will be used.

Parameters
listener – `GetImageListener` The listener instance.

getPhotoFromUser(*listener*)

This method will be called when the application "Insert Photo" button is pressed.

The implementor may allow the user to capture a photo for insertion into the document. If so, then call `listener.done(true, Uri)` and pass a `Uri` identifying the captured photo.

If not, call `listener.done(false, null)`, and a default photo capture method will be used.

Parameters
listener – `GetImageListener` The listener instance.

3.4.2 SOPersistentStorage

An interface that specifies the basis for implementing a class allowing for storage/retrieval of key/value pairs.

getStorageObject(*context*, *storeName*)

This method will be called to obtain a storage object for the named store.

Parameters

- **context** – `Context` The application context.
- **storeName** – `String` The data store identifier.

Returns
`Object` The object to be used to manage the store.

setStringPreference(*storageObject*, *key*, *value*)

This method will be called to set a key/value pair in the store.

Parameters

- **storageObject** – `Object` The object to be used to manage the store.
- **key** – `String` The data key.
- **value** – `String` The data value.

getStringPreference(*storageObject*, *key*, *defaultValue*)

This method will be called to retrieve data, identified by key, from the store. The default value will be returned if the key does not reside in the store.

Parameters

- **storageObject** – `Object` The object to be used to manage the store.
- **key** – `String` The data key.
- **defaultValue** – `String` The default data value.

Returns
`String` The retrieved string.

getAllStringPreferences(*storageObject*)

This method will be called to retrieve all entries from the persistent store.

Parameters
storageObject – Object The object to be used to manage the store.

Returns
Map<String,?> The retrieved key/value pairs.

removePreference(*storageObject*, *key*)

This method will be called to remove an entry from the persistent store.

Parameters

- **storageObject** – Object The object to be used to manage the store.
- **key** – String The data key.

3.4.3 SOClipboardHandler

An interface that specifies the basis for implementing a class to handle clipboard actions for the document editor.

putPlainTextToClipboard(*text*)

This method passes a string, cut or copied from the document, to be stored in the clipboard.

Parameters
text – String The text to be stored in the clipboard.

getPlainTextFromClipoard()

This method returns the contents of the clipboard.

Returns
String The text stored in the clipboard.

clipboardHasPlaintext()

This method ascertains whether the clipboard has any data.

Returns
Boolean True if the clipboard has data. False otherwise.

initClipboardHandler(*activity*)

This method initializes the clipboard handler

Parameters
activity – Activity The activity context for the clipboard.

releaseClipboardHandler()

This method releases the clipboard handler.

3.4.4 SOSecureFS

An interface that specifies the basis for implementing a class to allow proprietary encrypted files, stored in a secure container. A developer can use this opportunity to enforce security and role restrictions, or map the file operations onto another mechanism, such as a database.

`isSecurePath`(*path*)

This method determines whether the supplied file path resides within the secure container.

The path may be a pseudo path on which a mapping can be performed to access the actual file. For example "/SECURE/filename".

Parameters
`path` – `String` The file path to be analysed.

Returns
`Boolean` True if the file resides within the secure container. False otherwise.

`getTempPath`()

This method returns the directory to be used to store temporary files created during file translation/saving. This directory must reside within the secure container as identified in `isSecurePath(path)`.

Returns
The (pseudo) path to the temporary directory.

`getFileAttributes`(*path*)

This method returns the relevant attributes of the file located at the supplied path. The path will reference a file within the secure container.

The attributes should refer to the properties of the decrypted file.

The (pseudo) path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
`path` – `String` The path to the file to obtain attributes for.

Returns
`FileAttributes` A reference to a `FileAttributes` object, `null` on error.

`renameFile`(*src*, *dst*)

This method renames a file within the secure container

Both source and destination file paths will reside within the secure container and be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters

- `src` – `String` The path to the file to be renamed.
- `dst` – `String` The path to the destination file.

Returns
`Boolean` True on success. False on failure.

`copyFile`(*src*, *dst*)

This method copies a file to a new location within the secure container.

Both source and destination file paths will reside within the secure container and be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters

- `src` – `String` The path to the file to be copied.

- **dst** – String The path to the destination file.

Returns
Boolean True on success. False on failure.

deleteFile(*path*)

This method deletes a file from within the secure container.

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – String The path to the file to be deleted.

Returns
Boolean True on success. False on failure.

fileExists(*path*)

This method tests for the existence of a file within the secure container

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – String The path to the file to be checked.

Returns
Boolean True if the file exists. False otherwise.

recursivelyRemoveDirectory(*path*)

This method recursively deletes the supplied directory, and it's sub-directories, located within the secure container.

The path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – String The path to the directory to be deleted.

Returns
Boolean True on success. False on failure.

createDirectory(*path*)

This method creates a directory, and all non-existent directories in the supplied path, within the secure container.

The path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – String The path to the directory to be created.

Returns
Boolean True on success. False on failure.

createFile(*path*)

This method creates a file within the secure container.

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – String The path to the file to be created.

Returns
Boolean True on success. False on failure.

`getFileHandleForReading`(*path*)

This method opens an existing file, in the secure container, for reading.

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – `String` The path to the file to be opened.

Returns
`Object` The file handle object on success, `null` on error.

`getFileHandleForWriting`(*path*)

This method opens an existing file, in the secure container, for writing.

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – `String` The path to the file to be opened.

Returns
`Object` The file handle object on success, `null` on error.

`getFileHandleForUpdating`(*path*)

This method opens an existing file, in the secure container, for updating.

The file path will be constructed using the secure container identifier used in `isSecurePath(path)`.

Parameters
path – `String` The path to the file to be opened.

Returns
`Object` The file handle object on success, `null` on error.

`closeFile`(*handle*)

This method closes an open file within the secure container.

Parameters
handle – `Object` The file handle object to be closed.

Returns
`Boolean` True on success. False on failure.

`setFileLength`(*handle*, *length*)

This method sets the file length of a file located within the secure container.

Parameters

- **handle** – `Object` The file handle object to be used.
- **length** – `Long` The file length.

Returns
`Boolean` True on success. False on failure.

`readFromFile`(*handle*, *buf[]*)

This method reads data from a file located within the secure container.

Parameters

- **handle** – `Object` The file handle object to be used.
- **buf[]** – `byte` The buffer to put the read data in.

Returns
`Int` The amount of data placed in the buffer. 0 on EOF, -1 on error.

writeToFile(*handle*, *buf[]*)

This method writes data to a file located within the secure container.

Parameters

- **handle** – Object The file handle object to be used.
- **buf[]** – byte The buffer containing the data.

Returns
Int The amount of data written. -1 on error.

syncFile(*handle*)

This method forces buffered data to be written to the underlying device.

Parameters
handle – Object The file handle object to be used.

Returns
Boolean True on success. False on failure.

getFileLength(*handle*)

This method obtains the length of a file within the secure container.

The length returned is that of the decrypted file.

Parameters
handle – Object The file handle object to be used.

Returns
Int The file length. -1 on error.

getFileOffset(*handle*)

This method obtains the offset, from the start of the file.

The offset returned relates to the decrypted file.

Parameters
handle – Object The file handle object to be used.

Returns
Int The file pointer offset. -1 on error.

seekToFileOffset(*handle*, *offset*)

This method moves the file pointer to the requested offset from 0.

The offset relates to the decrypted file.

Parameters

- **handle** – Object The file handle object to be used.
- **length** – Long The offset length.

Returns
Int True on success. False on failure.

getSecurePath()

Return the physical root of the secure container.

Returns
String The physical path to the secure container.

`getSecurePrefix()`

Return the 'tag' to indicate the file is suitable for decryption.

Returns

`String` The secure 'tag' string.

3.5 Document API - All Main Document Types

Main document types

Word	
Presentation	
Spreadsheet	
PDF	

These methods apply to document types referenced above. Note: *SmartOffice* supports loading image files as well as some more obscure document types, however, for these kind of documents the methods outlined below will probably *not* be relevant.

The methods within the `DocumentView` class should be considered for the API as outlined below.

Note: *DocumentView* extends *NUIView*.

3.5.1 Starting the Document View

Assuming your `DocumentView` has been initialised then it requires to be started with a reference to the file which needs to be viewed.

There are a few `start` methods to consider as follows:

`start`(*uri*, *page*, *showUI*)

Starts the view with a given file path, page number and optional built in *Default UI* layer.

Parameters

- **`uri`** – `Uri` The file path
- **`page`** – `Int` Start at a given page number (zero-based).
- **`showUI`** – `Boolean` Controls whether the *Default UI* is used.

`start`(*uri*, *page*, *showUI*, *isTemplate)*()

Starts the view with a given file path, page number and optional built in *Default UI* layer.

Parameters

- **`uri`** – `Uri` The file path
- **`page`** – `Int` Start at a given page number (zero-based).
- **`showUI`** – `Boolean` Controls whether the *Default UI* is used.
- **`isTemplate`** – `Boolean` Defines whether the document is considered to be a template or not.

3.5.2 File Operations

There are a number of file operations available against a document, an application developer should only be required to implement these if using the *Custom UI*.

save()

Saving a document only needs to be invoked if there are changes made to a document. As such, an application developer can *verify if changes have been made or not* and act accordingly.

save(*listener*)

Save a document with an associated listener object registered for the result.

Parameters
listener – OnSaveListener The listener object for the save result.

On save listener interface

```
public interface OnSaveListener {
   void done(boolean val);
}
```

saveTo(*path*, *listener*)

When saving a document using this method an application developer must provide a valid path on the file system as well as a listener for the result.

Parameters

- **path** – String file path.
- **listener** – SODocSaveListener the listener which will be invoked at the result of the file save operation.

Kotlin

Java

```
documentView.saveTo(<file_path>, SODocSaveListener { result, err ->
   if (result == SODocSave_Succeeded) {
       //  success
   } else {
       //  error
   }
})
```

```
documentView.saveTo(<file_path>, new SODocSaveListener() {
   @Override
   public void onComplete(int result, int err) {
        if (result == SODocSave_Succeeded) {
            //  success
        } else {
            //  error
        }
   }
});
```

exportTo(*path*, *format*, *listener*)

Use this method to export the current file to another format.

Parameters

- **path** – `String` file path.
- **format** – `String` the file format to export to. Currently only `txt` and `pdf` is supported.
- **listener** – `SODocSaveListener` the listener which will be invoked at the result of the file save operation.

Kotlin

Java

```
documentView.exportTo(<file_path>, "txt", SODocSaveListener { result, err ->
   if (result == SODocSave_Succeeded) {
      //  success
   } else {
      //  error
   }
})
```

```
documentView.exportTo(<file_path>, "txt", new SODocSaveListener() {
   @Override
   public void onComplete(int result, int error) {

   }
});
```

`print`()

Application developers should call this method to open the native *Android* print dialog for the document.

`addDeleteOnClose`(*path*)

Adds the given file to the "delete on close" list. This file will be deleted when the application is closed.

Parameters
path – `String` The file path to be added to the list.

`reload`(*path*)

Reload the current view from a file at the given path.

Parameters
path – `String` the file path.

3.5.3 Pages

`getPageNumber`()

Gets the current page number of the document view.

Returns
`Int` Note: Zero-indexed.

`getPageCount`()

Gets the total page count of the document.

Returns
`Int`.

Once a document is loaded an application developer can view pages either by scrolling the document view or by using the *SmartOffice* API as follows:

goToPage(*pageNumber*)

Moves the document to the defined page.

Parameters
pageNumber – Int Note: this is zero-indexed, thus pageNumber = 0 would show page 1 of your document.

firstPage()

Jumps the document to the first page.

lastPage()

Jumps the document to the last page.

A handy way of showing or hiding the page navigator in your *Custom UI* can be utilized with the following methods:

showPageList()

Shows the page thumbnail list.

hidePageList()

Hides the page thumbnail list.

isPageListVisible()

Queries the presence of the page thumbnail list.

setPageChangeListener(*listener*)

Sets a listener for page changes on the document.

Parameters
listener – ChangePageListener the listener object.

Change page listener interface

```
public interface ChangePageListener {
   void onPage(int pageNumber);
}
```

Important: These methods will only reliably operate once the document has fully loaded - therefore a developer should register for these events after the *docLoaded()* method is triggered.

historyNext()

Goes to the next position in the page history.

historyPrevious()

Goes to the previous position in the page history.

hasNextHistory()

Returns whether the document has a next entry in the page history.

hasPreviousHistory()

Returns whether the document has a previous entry in the page history.

findPageContainingPoint(*point*)

Given a point in screen coordinates, find the page that contains that point. The result may be null if the page is not found.

Parameters
point – Point The lookup coordinate for the page.

Returns
Int Page pointer.

screenToPage(*pageNum*, *point*)

Given a point in screen coordinates and a page number, get the point in *MuPDF* page coordinates on that page.

Parameters

- **pageNum** – Int The lookup page number.
- **point** – Point The lookup coordinate for the page.

Returns
Point The return coordinates.

3.5.4 Viewing Full-screen

In order to view a document in full-screen, it is up to the application developer to hide any UI which has been presented, and set the frame of the document view to fill the screen. Once that's done, calling enterFullScreen and passing a Runnable to it will invoke the full-screen mode. When the user taps to exit full-screen mode, the Runnable will be invoked, at which time the UI and frame should be restored to their previous state.

enterFullScreen(*runnable*)

Parameters
runnable – Runnable The function to be triggered once the full-screen mode is exited.

Kotlin

Java

```
documentView?.enterFullScreen({
   // restore UI to non-full screen mode
})
```

```
documentView.enterFullScreen(new Runnable() {
      @Override
      public void run() {
         // restore UI to non-full screen mode
      }
});
```

3.5.5 Document Actions

isDocumentModified()

This method will return a Boolean value representing the modified status of the document (i.e. whether it has been edited or not).

Returns
Boolean *true* if the document has been modified, otherwise *false*.

Kotlin

Java

```
val modified: Boolean = documentView.isDocumentModified
```

```
Boolean modified = documentView.isDocumentModified();
```

undo()

Undo a previous action.

redo()

Redo a previous action.

canUndo()

Returns whether there is an action which can be undone.

Returns
Boolean.

canRedo()

Returns whether there is an action which can be redone.

Returns
Boolean.

providePassword(*password*)

Use this to provide a password to open the document.

Parameters
password – String.

searchForward(*text*)

Search forward for text.

Parameters
text – String The text to search for.

searchBackward(*text*)

Search backward for text.

Parameters
text – String The text to search for.

setGoBackHandler(*handler*)

Sets a handler for the *Android* back operation.

Parameters
handler – NUIDocView.GoBackHandler

finish()

Informs the document view to finish.

3.5.6 Document Selections

canSelect()

Return *true* if the view supports selection.

Returns
Boolean.

deleteSelection()

Deletes the current selection on the document.

canDeleteSelection()

Returns whether the selection can be deleted.

Returns
Boolean.

clearSelection()

Clears any selection on the document.

setSelectionText(*text*)

Sets the text on the document's current selection.

Parameters
text – String Text to set.

getSelectedText()

Gets the selected text (if there is a current selection) from the document.

Returns
String or NULL.

deleteSelectedText()

Deletes the currently selected text.

isSelectionInkAnnotation()

Returns whether the current selection is an ink annotation or not.

Returns
Boolean.

getSelectionHasAssociatedPopup()

Returns whether the current document selection has an associated popup or not.

Note: A "popup" in this case will be a graphical user interface of options for the selection.

Returns
Boolean.

select(*point*)

Select text based on one point, in view coordinates this will place the caret in the text near the given point.

Parameters
point – Point.

Returns
Boolean.

select(*topLeft*, *bottomRight*)

Select text based on two given points, in view coordinates.

Parameters

- **topLeft** – Point The top left point.
- **bottomRight** – Point The bottom right point.

Returns
Boolean.

getSelectionCanStyleBeChanged()

Returns whether the current selection style can be changed or not.

Returns
Boolean.

3.5.7 Document Positioning and Scaling

Additional to user interaction providing positioning and scaling of a document, the following API is available:

getScrollPositionX()

Return the view's current x-scroll position, a return value of -1 indicates an error.

Returns
Int.

getScrollPositionY()

Return the view's current y-scroll position, a return value of -1 indicates an error.

Returns
Int.

getScaleFactor()

Return the view's current scale factor, a return value of -1 indicates an error

Returns
Float.

setScrollX(*xPos*)

Sets the x position of the document.

Parameters
xPos – Int X position.

setScrollY(*yPos*)

Sets the y position of the document.

Parameters
yPos – Int Y position.

setScaleX(*xScale*)

Sets the x scale of the document.

Parameters
xScale – Float X scale.

setScaleY(*yScale*)

Sets the y scale of the document.

Parameters
yScale – Float Y scale.

setScaleAndScroll(*scale*, *xPos*, *yPos*)

Sets the scale and position of the document.

Parameters

- **scale** – Float scale.
- **xPos** – Int x position.
- **yPos** – Int y position.

gotoInternalLocation(*int page*, *RectF box*)

Given a page number and a rectangle, this method will scroll so that the given location in the document is visible.

Parameters

- **page** – Int The page number, zero-based.
- **box** – RectF The rectangle.

3.5.8 Media

setDrawModeOn()

Sets the draw mode to ink annotation drawing.

isDrawModeOn()

Check to see if ink annotation draw mode is enabled.

Returns
Boolean.

setDrawMode(*mode*)

Sets the draw mode for the document.

Parameters
mode – DocView.AnnotMode.

getDrawMode()

Gets the draw mode for the document.

Returns
DocView.AnnotMode.

isDrawModeOn(*mode*)

Check to see if specific draw mode is enabled.

Parameters
mode – DocView.AnnotMode.

Returns
Boolean.

DocView.AnnotMode enum

```
public enum AnnotMode { NONE, LINE, ARROW, RECTANGLE, OVAL, INK, POLYGON, POLYLINE }
```

setDrawModeOff()

Disables drawing mode.

setLineColor(*color*)

Sets the color for ink annotations.

Parameters
color – Int The color, in ARGB format.

getLineColor()

Gets the color for ink annotations.

Returns
Int The color, in ARGB format.

setOpacity(*opacity*)

Sets the opacity for ink annotations.

Parameters
opacity – Int The opacity (0-255).

getOpacity()

Gets the opacity for ink annotations.

Returns
Int The opacity (0-255).

setLineThickness(*thickness*)

Sets the line thickness in points.

Parameters
thickness – Float.

getLineThickness()

Gets the line thickness in points.

Returns
Float.

setSelectionInkColor(*color*)

Sets the ink color of the selection.

Parameters
color – Int Color in ARGB format.

setSelectionInkThickness(*width*)

Sets the ink thickness of the selection.

Parameters
width – Float Thickness in points.

setSelectionAnnotLineColor(*int color*)

Sets the line color of the selected annotation.

Parameters
color – Int Color in ARGB format.

setSelectionAnnotLineThickness(*thickness*)

Sets the line thickness of the selected annotation.

Parameters
thickness – `Float` Thickness in points.

3.5.9 Document Author

Word	
PDF	

For *Word* and *PDF* documents there is the concept of "Document Author".

This is used for *Word* track changes edits as well any *PDF* annotation comments.

setAuthor(*author*)

Sets the author.

Parameters
author – `String` The author name.

getAuthor()

Gets the author.

Returns
`String` The author name.

getPersistedAuthor()

Gets any persisted author which may have been saved between sessions.

Returns
`String` The author name.

3.5.10 Text Modes

toggleTextUnderlineMode()

Toggles the underline mode depending on the current text selection underline state.

isTextUnderlineModeOn()

Returns the current text underline mode.

Returns
`Boolean`.

toggleTextStrikeThroughMode()

Toggles the strike through mode depending on the current text selection strike through state.

isTextStrikeThroughModeOn()

Returns the current text strike through mode.

Returns
`Boolean`.

toggleTextSquigglyMode()

Toggles the squiggle mode depending on the current text selection squiggle state.

`isTextSquigglyModeOn()`

Returns the current text squiggle mode.

Returns
`Boolean`.

3.5.11 Other

`isKeyboardVisible()`

Returns *true* if the keyboard is visible.

Returns
`Boolean`.

`hasHistory()`

Returns *true* if the view supports history.

Returns
`Boolean`.

`hasSearch()`

Returns *true* if the document view supports searching.

Returns
`Boolean`.

`hasUndo()`

Returns *true* if the document view supports undo.

Returns
`Boolean`.

`hasRedo()`

Returns *true* if the document view supports undo.

Returns
`Boolean`.

`hasSelectionAlignment()`

Returns *true* if the view supports alignment setting.

Returns
`Boolean`.

`hasIndent()`

Returns *true* if the view supports indentation setting.

Returns
`Boolean`.

`shouldConfigureSaveAsPDFButton()`

Find out if the document can be saved as a PDF file.

Returns
`Boolean`.

`shouldConfigureExportPdfAsButton()`

Find out if the document can be exported as a PDF file.

Returns
Boolean.

3.6 Document API - *Office*

Office document types

Word		• **docx** • **doc** • **dotx** • **docm** • **dotx** • **odt**
Presentation		• **pptx** • **ppt** • **pps** • **pptm**
Spreadsheet		• **xlsx** • **xls** • **xlt** • **xlsm** • **xltm**

The methods within the `DocumentViewOffice` class should be considered for the API as outlined below.

Note: *DocumentViewOffice* extends *DocumentView*.

3.6.1 Selections and Styling

Standard styling

`setSelectionIsBold`(*active*)

Sets selected text to bold or not.

Parameters
active – Boolean.

`getSelectionIsBold`()

Returns whether the current text selection is bold or not.

Returns
Boolean.

`setSelectionIsItalic`(*active*)

Sets selected text to italic or not.

Parameters
active – Boolean.

`getSelectionIsItalic()`

Returns whether the current text selection is italic or not.

Returns
Boolean.

`setSelectionIsLinethrough(active)`

Sets selected text to strike-through or not.

Parameters
active – Boolean.

`getSelectionIsLinethrough()`

Returns whether the current text selection is struck-through or not.

Returns
Boolean.

`setSelectionIsUnderlined(active)`

Sets selected text to underline or not.

Parameters
active – Boolean.

`getSelectionIsUnderlined()`

Returns whether the current text selection is underlined or not.

Returns
Boolean.

Bullet lists

`setSelectionListStyleDisc()`

Sets a disc bullet list style on the selection.

`setSelectionListStyleDecimal()`

Sets a numeric bullet list style on the selection.

`setSelectionListStyleNone()`

Sets no bullet list style on the selection.

`getSelectionListStyleIsDisc()`

Returns
Boolean.

`getSelectionListStyleIsDecimal()`

Returns
Boolean.

`getSelectionListStyleIsNone()`

Returns
Boolean.

Alignment

setSelectionHorizontalAlignment(*alignment*)

Sets selected text horizontal alignment value.

Parameters
alignment – SODoc.HorizontalAlignment

enum class **HorizontalAlignment**

- HORIZONTAL_ALIGN_LEFT
- HORIZONTAL_ALIGN_RIGHT
- HORIZONTAL_ALIGN_CENTER
- HORIZONTAL_ALIGN_JUSTIFY
- HORIZONTAL_ALIGN_INVALID

setSelectionVerticalAlignment(*alignment*)

Sets selected text vertical alignment value.

Parameters
alignment – SODoc.VerticalAlignment

enum class **VerticalAlignment**

- VERTICAL_ALIGN_BOTTOM
- VERTICAL_ALIGN_TOP
- VERTICAL_ALIGN_CENTER
- VERTICAL_ALIGN_INVALID

Indentation

setIndentationLevel(*level*)

Sets the indentation level on the current text selection.

Parameters
level – Int Represents indentation level.

getIndentationLevel()

Returns
IntArray Represents available indentation levels.

Note: These indentation level methods only apply to text selections in *Word* documents.

Delete, Cut, Copy, Paste & Clipboard

`canCopySelection()`

Returns if the current selection can be copied. Note: if *true* then is can also be deleted.

Returns
`Boolean`.

`selectionDelete()`

Deletes the selection.

`selectionCutToClip()`

Cuts a selection to the clipboard.

`selectionCopyToClip()`

Copies a selection to the clipboard.

`selectionPaste()`

Pastes from the clipboard to/over the current selection.

`clipboardHasData()`

Returns whether the clipboard has data or not.

Returns
`Boolean`.

Other

`isSelectionActive()`

Returns whether there is an active selection or not.

Returns
`Boolean`.

`isSelectionCaret()`

Returns whether the selection is a caret or not.

Returns
`Boolean`.

`isSelectionAutoshapeOrImage()`

Returns *true* if the selection is an autoshape or image, otherwise *false*.

Returns
`Boolean`.

`isSelectionAlterableTextSelection()`

Returns *true* if the current selection is a text selection that can be extended or contracted.

Returns
`Boolean`.

`selectionPermitsInlineTextEntry()`

Returns *true* if the current selection can accept direct text entry (i.e. without requiring a virtual keyboard or other intermediary).

Returns
`Boolean`.

`selectionCanHavePictureInserted()`

Returns *true* if the current selection can have a picture/image inserted into it.

Returns
`Boolean`.

`getSelectionCanStyleBeChanged()`

Returns whether the text selection style can be modified or not.

Returns
`Boolean`.

`getSelectionHasAssociatedPopup()`

Returns whether the text selection has pop-up or not.

Returns
`Boolean`.

3.6.2 Media

Drawing

See the *media section for all document types* for how to set the document mode to draw.

`docSupportsDrawing()`

Returns whether the document supports the drawing feature or not.

Returns
`Boolean`.

`setSelectionLineWidth(`*width*`)`

Sets the line width for the current selection.

Parameters
width – `Float` The line width.

`getSelectionLineWidth()`

Gets the line width for the current selection.

Returns
`Float` The line width. `0` is returned on error.

`setSelectionLineType(`*type*`)`

Sets the line type for the current selection.

Parameters
type – `Int` The line type.

`getSelectionLineType()`

Gets the line type for the current selection.

Returns
`Int` The line type.

Line type enum

```
int lineTypes[] = {
        SOLineStyle_Solid,
        SOLineStyle_DotSys,
        SOLineStyle_DotGEL,
        SOLineStyle_DashSys,
        SOLineStyle_DashGEL,
        SOLineStyle_LongDashGEL,
        SOLineStyle_DashDotSys,
        SOLineStyle_DashDotGEL,
        SOLineStyle_LongDashDotGEL,
        SOLineStyle_DashDotDotSys,
        SOLineStyle_LongDashDotDotGEL
};
```

Note: See *color API* for color styling options.

Images

canBeInserted(*path*)

Checks whether the image at the supplied path is suitable for insertion into the document.

Parameters
path – `String` The file path of the image file to check.

Returns
`Boolean`.

doInsertImage(*path*)

Insert the image at the given path into the document at the current selection point.

Parameters
path – `String` The file path of the image file to insert.

3.6.3 Fonts

getFontList()

Returns
`Array<String>` An array of available fonts.

Kotlin

Java

```
val fontNames: Array<String> = documentView.fontList
```

```
final String[] fontNames = documentView.getFontList();
```

setSelectionFontName(*name*)

Sets a font name against the currently selected text.

Parameters
name – `String` The name of the font.

`getSelectionFontName()`

Returns
`String` The font name of the current text selection.

`setSelectionFontSize(size)`

Sets the font size for the selected text.

Parameters
size – `Double` The name of the font.

`getSelectionFontSize()`

Returns
`Double` The font size of the current text selection.

`getMaxFontSize()`

Gets the maximum font size.

Returns
`Double`.

`getMinFontSize()`

Gets the minimum font size.

Returns
`Double`.

3.6.4 Color

`setSelectionFontColor(hex)`

Sets the selected text color value.

Parameters
hex – `String` Hexidecimal string value in format "#rrggbb".

`getSelectionFontColor()`

Gets the selected text color value.

Returns
`String` Hexidecimal string value in format "#rrggbb".

`setSelectionBackgroundColor(hex)`

Sets the selected text background color value.

Parameters
hex – `String` Hexidecimal string value in format "#rrggbb".

`getSelectionBackgroundColor()`

Gets the selected text background color value.

Returns
`String` Hexidecimal string value in format "#rrggbb".

`setSelectionBackgroundTransparent()`

A handy method for setting no background color on the selected text.

`getBgColorList()`

Returns an array of the permitted background colors for the document.

Returns
`String[]` or `null`.

Note: Certain documents (e.g. `docx`) may have restrictions for background colors. This method will indicate what those might be.

`setSelectionLineColor(`*hex*`)`

Sets the line color for the current selection.

Parameters
`hex` – `String` Hexidecimal string value in format "#rrggbb".

`getSelectionLineColor()`

Returns the line color associated with the current selection.

Returns
`String` Hexidecimal string value in format "#rrggbb" or `null`.

`setSelectionFillColor(`*hex*`)`

Sets the fill color for the current selection.

Parameters
`hex` – `String` Hexidecimal string value in format "#rrggbb".

`getSelectionFillColor()`

Returns the fill color associated with the current selection.

Returns
`String` Hexidecimal string value in format "#rrggbb" or `null`.

3.7 Document API - *Word*

As well as *the API for all document types* & *the API for Office document types* *Word* documents have an API for functionality particular to their format.

The class which handles *Word* documents is the `DocumentViewDoc` class and the methods outlined below should be considered as the specific API for the following document file types:

- **docx**
- **doc**
- **dotx**
- **docm**
- **dotx**
- **odt**

Note: *DocumentViewDoc* extends *DocumentViewOffice*.

3.7.1 Reflow

hasReflow()

Return *true* if the view supports reflow.

Returns
Boolean.

setFlowMode(*mode*)

Set the document's flow mode.

Parameters
mode – Int The *flow mode enum*.

setFlowModeNormal()

Set the document's flow mode to "Normal".

setFlowModeReflow()

Set the document's flow mode to "Reflow".

Note: Reflowing a document ensures that as the document scale changes the page content will always fit the width of the document. This means that a user should never have to pan the document horizontally to read the content.

---

setFlowModeResize()

Set the document's flow mode to "Resize".

Note: This is effectively the same as "Reflow", however it will re-paginate the document to increase or decrease the total page numbers as the document scale changes.

---

getFlowMode()

Returns the document's flow mode.

Returns
Int.

Flow mode enum:

- 1: normal
- 2: reflow
- 3: resize

3.7.2 Track changes and reviewing

If your application requires track changes and reviewing functionality for documents you should consider the following API.

docSupportsReview()

Returns whether the document supports the review feature.

Returns
Boolean.

`setTrackingChanges`(*track*)

Switch on or off the tracking of changes for the document.

Parameters
`track` – `Boolean`

`getTrackingChanges`()

Returns whether changes are being tracked for the document.

Returns
`Boolean`.

`setShowingTrackedChanges`(*show*)

Sets whether tracked changes are to be shown or not.

Note: *Showing* the changes is a visual aspect of tracking changes and is independent of `setTrackingChanges()`.

Parameters
`show` – `Boolean`.

`getShowingTrackedChanges`()

Returns whether to enable or disable *showing* tracked changes.

Returns
`Boolean`.

`selectionIsReviewable`()

Returns whether the current selection can be tracked.

Returns
`Boolean`.

`acceptTrackedChange`()

Accepts a selected tracked change.

Only has effect if tracking of changes is enabled and if a tracked change is selected on the document.

`rejectTrackedChange`()

Rejects a selected tracked change.

Only has effect if tracking of changes is enabled and if a tracked change is selected on the document.

`addComment`()

This adds a highlight annotation to the selected text along with a review comment.

Only has effect if tracking of changes is enabled.

`saveComment`()

Save the current review comment.

Only has effect if tracking of changes is enabled.

`previousTrackedChange`()

Moves the selection to the previous tracked change, if any.

Only has effect if tracking of changes is currently *shown*.

`nextTrackedChange()`

Moves the selection to the next tracked change, if any.

Only has effect if tracking of changes is currently *shown*.

`deleteHighlightAnnotation()`

Deletes a highlight annotation at the current text selection.

Only has effect if tracking of changes is currently *shown*.

3.8 Document API - *Presentation*

As well as *the API for all document types* & *the API for Office document types* *Presentation* documents have an API for functionality particular to their format.

The class which handles *Presentation* documents is the `DocumentViewPpt` class and the methods outlined below should be considered as the specific API for the following document file types:

- **pptx**
- **ppt**
- **pps**
- **pptm**

Note: *DocumentViewPpt* extends *DocumentViewOffice*.

3.8.1 Autoshapes

Adding shapes

There are 13 available *Autoshapes* which are always inserted into the center of the current page when the `insertAutoShape` function is invoked. Additionally there are styling options which can be applied with the `properties` parameter.

`insertAutoShape(`*shape*, *properties*`)`

Insert a given *AutoShape* with the given properties in the centre of the current page.

Parameters

- **shape** – `String` shape to insert. Possible values are:
  - `TextBox`
  - `Line`
  - `RectRoundRect`
  - `Ellipse`
  - `Triangle`
  - `RtTriangle`
  - `Arrow`
  - `LeftRightArrow`

  - `Diamond`
  - `WedgeEllipseCallout`
  - `WedgeRectCallout`
  - `Pentagon`
  - `Star`
- **properties** – `String` A string of properties to apply. Can be `null`. A common case is to supply a *"TextBox"* with property *"fill-color:transparent"*, or a *"Line"* with property *"end-decoration:"arrow""* to add an arrow head to the end.

Arranging shapes

Shapes require to be selected before calling the following arrangement methods:

setSelectionArrangeForwards()

Arranges a shape forward by one in the stack.

setSelectionArrangeFront()

Arranges a shape to the front of the stack.

setSelectionArrangeBackwards()

Arranges a shape backward by one in the stack.

setSelectionArrangeBack()

Arranges a shape to the back of the stack.

3.8.2 Playing *Presentation* slide shows

Playing a slide show just requires the `doSlideShow` method to be invoked. Once in slide show mode the following controls should be considered:

- *Tap* or *Swipe left*: Next slide.
- *Swipe right*: Previous slide.
- *Long press*: Exit slide show.

Note: The slide show is an entirely new `Activity` which sits on top of the existing `Activity`.

---

doSlideShow()

Plays the *Presentation* slides.

3.9 Document API - *Spreadsheet*

As well as *the API for all document types* & *the API for Office document types* *Spreadsheet* documents have an API for functionality particular to their format.

The class which handles *Spreadsheet* documents is the `DocumentViewXls` class and the methods outlined below should be considered as the specific API for the following document file types:

- **xlsx**
- **xls**
- **xlt**
- **xlsm**
- **xltm**

Note: *DocumentViewXls* extends *DocumentViewOffice*.

3.9.1 Cell management

Adding and deleting rows & columns

The number of rows or columns to be added or deleted depends on the number of rows or columns which are currently selected. For example if 3 rows are currently selected and `addRowsAbove()` is invoked then 3 rows will be added above the 1st row of the current selection, similarly if 2 columns are selected and `addColumnsRight()` is invoked then 2 columns will be added to the right of the last column of the current selection.

`addRowsAbove()`

Adds rows above the 1st row of the selected area on the document.

`addRowsBelow()`

Adds rows below the last row of the selected area on the document.

`deleteSelectedRows()`

Deletes the rows bound by the selected area on the document.

`addColumnsLeft()`

Adds columns to the left of the 1st column of the selected area on the document.

`addColumnsRight()`

Adds columns to the right of the last column of the selected area on the document.

`deleteSelectedColumns()`

Deletes the columns bound by the selected area on the document.

Cell selections

`isCellSelected()`

Returns whether a cell is selected or not.

Returns
`Boolean`.

`isMergedCellSelected()`

Returns whether a merged cell is selected or not.

Returns
`Boolean`.

`isMultipleCellsSelected()`

Returns whether multiple cells are selected or not.

Returns
`Boolean`.

`toggleMerge()`

Toggles the merged status of any selected cells.

Cell metrics

Cell metrics values are always in centimetres.

`getSelectedColumnWidth()`

Returns the selected column width.

Returns
`Float`.

`getSelectedRowHeight()`

Returns the selected row height.

Returns
`Float`.

`increaseCellWidth()`

Increases the cell width by the default amount (0.50f).

`decreaseCellWidth()`

Decreases the cell width by the default amount (0.50f).

`increaseCellHeight()`

Increases the cell height by the default amount (0.50f).

`decreaseCellHeight()`

Decreases the cell height by the default amount (0.50f).

`setSelectedCellWidth(`*value*`)`

Sets the selected cell width.

Parameters
value – `Float`.

`setSelectedCellHeight`(*value*)

Sets the selected cell height.

Parameters
value – `Float`.

Freeze options

Freezing panes allows a *Spreadsheet* to fix rows & columns in place at the point of selection.

`onFreezeTopRow`()

Freeze panes at the first row in the document.

`onFreezeFirstColumn`()

Freeze panes at the first column in the document.

`isFrozen`()

Returns *true* if the current sheet has frozen panes.

Returns
`Boolean`.

`onFreezePanes`()

Freeze/unfreeze panes at the selected cell in the document.

`toggleFreezeShown`()

Toggles whether frozen panes are shown.

3.9.2 Cell formulas

The text value of a cell will define any formula. For example if the value of the cell is `=SUM(E3:E6)` then it will resolve as a basic addition formula for those cells.

3.10 Document API - *PDF*

SmartOffice also supports a few other *PDF* document types:

PDF document types

PDF		**pdf**	
SVG		**svg**	*Read Only*
ePub		**ePub**	*Read Only*
Comic book archive		**cbz**	*Read Only*
Fiction book		**fb2**	*Read Only*

As well as *the API for all document types* *PDF* documents have an API for functionality particular to their format.

The class which handles *PDF* documents is the `DocumentViewPdf` class and the methods outlined below should be considered as the specific API for this file type.

Note: *DocumentViewPdf* extends *DocumentView*.

3.10.1 Annotation Modes

A document can enter certain modes which define the user interaction mode with the document. When a mode is set, instead of the tapping to select an item, tapping will respect any current document annotation mode.

Drawing mode

As **Drawing mode** is also relevant to other document types the relevant API can be found in the sub-class *DocumentView setDrawMode()* method.

This mode deals with the following annotations:

- **Ink**
- **Line**
- **Square**
- **Circle**
- **Polygon**
- **Polyline**

Simply set the mode you require and then *touching the document area* will automatically start the *drawing* annotation.

Text mode

This mode deals with the following annotations:

- **Highlight**
- **Underline**
- **Strikethrough**
- **Squiggle**

When *text is selected* then this mode will automatically apply the annotation *text* style to the selection.

`setTextMode`(*mode*)

Sets the *text mode* of the document.

Parameters
mode – `TextMode` The mode to set.

`getTextMode`()

Returns the current text mode.

Returns
`TextMode`

Text mode enum:

```
TextMode { NONE, HIGHLIGHT, SQUIGGLE, UNDERLINE, STRIKETHROUGH }
```

`toggleTextHighlightMode`()

Toggles the highlight mode depending on the current text selection highlight state.

`isTextHighlightModeOn()`

Returns the current selection's highlight mode.

Returns
`Boolean`.

Placement mode

This mode deals with the following annotations:

- **Text**
- **Stamp**
- **Link**
- **Attachment**
- **Callout** (*roadmap*)
- **Caret** (*roadmap*)

When *the user taps on the document* then this mode will automatically *place* the annotation at the point of touch.

`setPlacementMode(`*mode*`)`

Set the *placement mode* of the document.

Parameters
mode – `PlacementMode` The mode to set.

`getPlacementMode()`

Returns the current placement mode.

Returns
`PlacementMode`.

`setPlacementModeOff()`

Sets the placement mode to `NONE`, this is a shortcut to `setPlacementMode(PlacementMode.NONE)`.

Placement mode enum:

```
PlacementMode { NONE, NOTE, TEXT, CALLOUT, CARET, STAMP, LINK, ATTACHMENT }
```

`isNoteModeOn()`

Check to see if placement note annotation mode is enabled.

Returns
`Boolean`.

`isTextModeOn()`

Check to see if placement text annotation mode is enabled.

Returns
`Boolean`.

`isStampModeOn()`

Check to see if placement stamp annotation mode is enabled.

Returns
`Boolean`.

`isLinkModeOn()`

Check to see if placement link annotation mode is enabled.

Returns
`Boolean`.

`isAttachmentModeOn()`

Check to see if placement attachment annotation mode is enabled.

Returns
`Boolean`.

`setNoteModeOn()`

Enable placement note annotation mode.

This is a shortcut for `setPlacementMode(PlacementMode.NOTE)`.

`setNoteModeOff()`

Disable placement note annotation mode.

`saveNoteData()`

Ensure that any outstanding note is saved to the document.

`setTextModeOn()`

Enable placement text annotation mode.

This is a shortcut for `setPlacementMode(PlacementMode.TEXT)`.

`setTextModeOff()`

Disable placement text annotation mode.

`setStampModeOn()`

Enable placement stamp annotation mode.

This is a shortcut for `setPlacementMode(PlacementMode.STAMP)`.

`setStampModeOff()`

Disable placement stamp annotation mode.

`setLinkModeOn()`

Enable placement link annotation mode.

This is a shortcut for `setPlacementMode(PlacementMode.LINK)`.

`setLinkModeOff()`

Disable placement link annotation mode.

`setAttachmentModeOn()`

Enable placement attachment annotation mode.

This is a shortcut for `setPlacementMode(PlacementMode.ATTACHMENT)`.

`setAttachmentModeOff()`

Disable placement attachment annotation mode.

3.10.2 Annotation Selections

isAlterableTextSelection()

Used to find out if selected text in a PDF file can have a highlight annotation created for it.

Returns
Boolean.

highlightSelection()

Create a highlight annotation from the currently selected text.

isSelectionLink()

Returns whether the current selection is a link or not.

Returns
Boolean

updateSelectedLinkDestination()

Updates the selected link destination.

Note: This will invoke a dialog to appear whereby a user can edit the link.

---

setTextHighlightColor(*color*)

Sets the color for text highlight annotations such as highlight, underline, squiggle, and strikethrough.

Parameters
color – Int The color, in ARGB format. Can be null.

setSelectedAnnotStyleAttributes(*strokeColor*, *fillColor*, *opacity*, *width*, *startStyle*, *endStyle*)

Set a collection of attributes at once to a selected annotation.

Parameters

- **strokeColor** – Int The stroke/line color, in ARGB format. Can be null.
- **fillColor** – Int The stroke/line color, in ARGB format. Can be null.
- **opacity** – Int The opacity (0-255).
- **width** – Float The thickness of the stroke/line in points.
- **startStyle** – Int The start style of an annotation.
- **endStyle** – Int The start style of an annotation.

getSelectedAnnotLineEndingStyles()

Returns the start and end styling for a selected annotation.

Returns
[Int] Array containing line style variables for the start and end of the line.

Line style variables:

```
LINE_ENDING_NONE = 0
LINE_ENDING_SQUARE = 1
LINE_ENDING_CIRCLE = 2
LINE_ENDING_DIAMOND = 3
LINE_ENDING_R_OPEN_ARROW = 4
LINE_ENDING_R_CLOSED_ARROW = 5
```

(continues on next page)

(continued from previous page)

```
LINE_ENDING_BUTT = 6
LINE_ENDING_OPEN_ARROW = 7
LINE_ENDING_CLOSED_ARROW = 8
LINE_ENDING_SLASH = 9
```

3.10.3 Annotation Styling

setLineEndStyles(*startStyle*, *endStyle*)

Sets the *line ending styles* for a drawing annotation.

Note: This only applies to "Line" annotations. This function will have no effect on other annotation types.

Parameters

- **startStyle** – Int
- **endStyle** – Int

setFillColor(*color*)

Set the fill color an annotation.

Parameters
color – Int The color, in ARGB format.

getFillColor()

Get the fill color for an annotation.

Returns
Int The color, in ARGB format.

3.10.4 Redactions

Redactions allow users to remove unwanted and/or sensitive information from documents. Essentially redacted areas are covered with black rectangles which obscure areas of the document. These areas will be permanently set on the document once applied.

redactMarkText()

Mark the selected text for redaction. Note, this does not *finalize* the redaction, rather it marks the text for redaction. This will apear as a red outlined area around the text on the document.

redactGetMarkTextMode()

Returns whether the document is in "mark text" redaction mode.

Returns
Boolean.

redactMarkArea()

Turn on "mark area" redaction mode. This allows the user to then draw a rectangle which is marked for redaction. Note, this does not *finalize* the redaction, rather it marks the area for redaction. This will apear as a red outlined area on the document.

`redactIsMarkingArea()`

Returns whether the document is in "mark area" redaction mode.

Returns
`Boolean`.

`redactRemove()`

Un-redact a marked and selected, *but not yet applied*, redaction.

`redactApply()`

Applies all marked and *not yet applied* redactions. This operation is not reversible.

Once applied any marked text and/or areas will turn solid black.

`canMarkTextRedaction()`

Returns whether currently selected text can be redacted.

Returns
`Boolean`.

`canRemoveRedaction()`

Returns whether a currently selected but *not yet applied* redaction can be removed (un-redacted).

Returns
`Boolean`.

`canApplyRedactions()`

Returns whether there are redactions that can be applied.

Returns
`Boolean`.

3.10.5 Secure Save

Secure Save is used to save a PDF document ensuring that any redacted information will be impossible to extract. Essentially the document is rasterized, so that all text data information is flattened into a bitmap and is thus impossible to be copied out again. Any previously redacted areas will therefore contain no text information.

`canSecureSave()`

Returns whether secure save can be used.

Returns
`Boolean`.

`secureSave()`

Invokes a dialog to choose a file name and then process the secure save operation for saving out the new (securely saved) file.

3.10.6 Signatures

`setDigitalSignatureModeOn()`

Enable digital signature creation mode.

`setDigitalSignatureModeOff()`

Disable digital signature creation mode.

`setESignatureModeOn(`*view*`)`

Enable e-signature creation mode.

Parameters
view – `View` Serves as an anchor for a popup that will appear. Must not be `null`.

`setESignatureModeOff()`

Disable e-signature creation mode.

`isDigitalSignatureMode()`

Returns *true* if digital signature mode is enabled.

Returns
`Boolean`.

`isESignatureMode()`

Returns *true* if e-signature mode is enabled.

Returns
`Boolean`.

`getSignatureCount()`

Returns the number of digital signatures found in the document.

Returns
`Int`.

`findNextSignature()`

Find and highlight the next digital signature.

`findPreviousSignature()`

Find and highlight the previous digital signature.

`setSigningHandler(`*handler*`)`

Specifies a handler for the file-saving step in document signing.

Parameters
handler – `NUIDocView.SigningHandler`.

3.10.7 Table of Contents

Also known as the "document outline" a table of contents may be available for the PDF document. The methods below can be used to get the listing and/or display the listing.

`isTOCEnabled()`

Returns whether the document has a table of contents.

Returns
`Boolean`.

`tableOfContents()`

Uses the built-in UI to display the table of contents.

`enumeratePdfToc(`*listener*`)`

Enumerates the document's table of contents. The listener's `nextTocEntry()` function will be called for each entry.

Parameters

`listener` – `EnumeratePdfTocListener` Dedicated listener object to handle enumeration.

The table of contents **enumeration listener object** is defined as follows:

```
public interface EnumeratePdfTocListener {
      /**
       * nextTocEntry is called once for each entry in the Table of Contents
       * handle and parentHandle can be used to navigate the TOC hierarchy
       * if page>=0 it's an internal link, so use x and y
       * if page<0 and url in not null, url is an external link
       *
       * @param handle        my ID number in the hierarchy
       * @param parentHandle  my parent's ID number in the hierarchy
       * @param page          target page number
       * @param label         user-friendly label
       * @param url           url for an external link
       * @param x             x - coordinate on the page.
       * @param y             y - coordinate on the page.
       */
      void nextTocEntry(int handle, int parentHandle, int page, String label, String url,
↪ float x, float y);

      /**
       * done is called once the enumeration is complete.
       */
      void done();
}
```

3.10.8 Metadata

Metadata allows a developer to set key/value pairs on PDF documents as well as retrieve metadata values from keys.

`setPDFMetaData(`*key*, *value*`)`

Sets the value of a metadata item in a PDF file from key/value pairs.

Parameters

- **`key`** – `String`.
- **`value`** – `String`.

`getPDFMetaData(`*key*`)`

Gets the value of a metadata item in a PDF file identified by a *key*.

Parameters

`key` – `String`.

Returns

`String` The value returned by the *key* or `null` if there is no match.

3.10.9 Links

Links in documents can be defined as *external* (a URL) or *internal* (to another page location within the document).

createPDFLink(*pageNum*, *hitRect*, *url*)

Creates an external link on a page.

Parameters

- **pageNum** – Int The page number, zero-based.
- **hitRect** – Rect A Rect on the page, in *MuPDF* page coordinates.
- **url** – String The external URL associated with the link.

createPDFLink(*pageNum*, *hitRect*, *destination*)

Creates an internal link on a page.

Parameters

- **pageNum** – Int The page number, zero-based.
- **hitRect** – Rect A Rect on the page, in *MuPDF* page coordinates.
- **destination** – LinkDestination A link destination within the document, created using makeLinkDestination()

makeLinkDestination(*page*)

Create a LinkDestination representing the top of the given page.

Parameters
page – Int The page number, zero-based.

Returns
LinkDestination.

makeLinkDestination(*page*, *left*, *top*, *width*, *height*)

Create a LinkDestination representing a rectangle on the given page.

Parameters

- **page** – Int The page number, zero-based.
- **left** – Float The left coordinate of the rectangle (x).
- **top** – Float The top coordinate of the rectangle (y).
- **width** – Float The width of the rectangle.
- **height** – Float The height of the rectangle.

Returns
LinkDestination.

3.10.10 Outlines

Outlines allow developers to create *Table of Contents* for PDF documents.

getOutlineIterator()

Get a new `OutlineIterator` for the document. The resulting iterator will be positioned at its beginning.

Returns
`OutlineIterator`.

createOutlineItem(*iterator*, *destination*, *title*)

Creates an `OutlineItem` from a `LinkDestination`.

Parameters

- **iterator** – `OutlineIterator`.
- **destination** – `LinkDestination`.
- **title** – `String`.

Returns
`OutlineIterator.OutlineItem`.

insertOutlineItem(*iterator*, *item*)

Insert an `OutlineItem` above the place where the iterator is positioned.

Parameters

- **iterator** – `OutlineIterator`.
- **item** – `OutlineIterator.OutlineItem`.

deleteOutlineItem(*item*)

Delete the `OutlineItem` at the iterator's current position.

Parameters
item – `OutlineIterator.OutlineItem`.

CHAPTER

FOUR

IOS SDK

4.1 Getting Started

4.1.1 System requirements

- Apple Mac
- Xcode minimum SDK: 9

4.1.2 Adding the SDK to your project

In order to use the *SmartOffice* SDK in your app you need to import the associated frameworks.

You should have been supplied the framework items specific to *SmartOffice* as follows:

- `mupdf_default_ui.xcframework`
- `mupdfdk.xcframework`
- `sodk_default_ui.xcframework`
- `sodk.xcframework`

4.1.3 Verify your integration

To ensure that the frameworks are correctly found check that the following import statements are valid:

Swift

Objective-C

```
import sodk
import sodk_default_ui
import mupdfdk
import mupdf_default_ui
```

```
#import "sodk/sodk.h"
#import "sodk_default_ui/sodk_default_ui.h"
#import "sodk_default_ui/sodk_default_ui-Swift.h"
#import "mupdfdk/mupdfdk.h"
#import "mupdf_default_ui/mupdf_default_ui.h"
#import "mupdf_default_ui/mupdf_default_ui-Swift.h"
```

(continues on next page)

(continued from previous page)

```
[super viewDidLoad];
```

Note: If the linked frameworks cannot be found ensure to validate your workspace in *Xcode* with:

Build Settings -> Build Options -> Validate Workspace = YES

Then build the project and re-check the import statements.

Check the version

Once you have successfully integrated the libraries the build version details can be checked as follows:

Swift

Objective-C

```
let versionInfo:[String:String?] = SODKLib.version()
print("versionInfo=\(versionInfo)")
```

```
NSDictionary *versionInfo = [SODKLib version];
NSLog(@"versionInfo=%@",versionInfo);
```

This returns the dictionaries of key/value pairs against your *SmartOffice SDK* supplied framework.

4.2 Document Viewing

Ensuring that you have added the *SmartOffice* frameworks to your project then you should be ready to view a document.

The first consideration which should be taken into account before opening a document is whether you are interested in building your own user interface (UI) to control the document view or not. If you do want to create your own UI then you should reference the *Custom UI* section, otherwise you should look at the *Default UI* implementation within this document.

4.2.1 Default UI

Implementing a *Default UI* is the simplist solution for using *SmartOffice* in your project. The UI controls for all the common document features will be provided within a pre-packaged user interface made by *Artifex*.

To use the *Default UI* in your project you need to instantiate it with a valid `URL` and on a closure method, push it from your `UINavigationController` as follows:

Swift

Objective-C

```
SODKDefaultUIViewController.viewController(url: <your_file_url>, whenReady:
↪{(docVC:ARDKDefaultUIViewController?) in
   if docVC != nil {
      self.navigationController?.pushViewController(docVC!, animated: true)
```

(continues on next page)

(continued from previous page)

```
    }
})
```

```
[SODKDefaultUIViewController viewControllerWithUrl:<your_file_url> whenReady:^
↪(ARDKDefaultUIViewController *docVC) {
    if (docVC) {
        [self.navigationController pushViewController:docVC animated:YES];
    }
}];
```

The Back Button

Your dedicated view controller, which pushes the instance of `ARDKDefaultUIViewController`, must include one bespoke method to enable a graceful exit back from the *Default UI*. This method is detailed as follows:

Swift

Objective-C

```
@IBAction func docCloseUnwindActionWithUnwindSegue(_ unwindSegue: UIStoryboardSegue) {}
```

```
- (IBAction)docCloseUnwindActionWithUnwindSegue:(UIStoryboardSegue *)unwindSegue {}
```

Important: The implementation can be left empty, but this function must be present in the view controller that should be unwound back to - otherwise nothing will happen when the user taps the back button.

Default UI Customization

Adjusting Document Settings

If you require to adjust the *document settings*, which will allow you to enable or disable features in the *Default UI*, then you will need to use the *session based implementation* to instantiate the *Default UI*.

Follow steps 1-5 as outlined in the *Custom UI implementation*, but instead of creating your own *Custom UI* just push the *Default UI* as you would normally.

See the *Full code example* for a quick reference.

Adjusting Graphical Assets

It is possible to supply your own graphical assets for the icons in the *Default UI*.

In order to do this you should rebuild the attached frameworks and edit the contents of the asset catalogs in *Xcode* as required. Then republish the frameworks for use in your main project.

- **sodk-default-ui**
  Contains the assets for Office document viewing (*Word*, *Presentation* & *Spreadsheet*).
- **mupdf-default-ui**
  Contains the assets for *PDF* viewing.

Unlimited customization of the *Default UI*

The *Default UI* can be rebuilt as a framework for you project by republishing the related *Xcode* projects with any amends you require. In this way there is unlimited customization of the *Default UI* to suit your requirements.

See:

`mupdf-default-ui.xcodeproj`

`sodk-default-ui.xcodeproj`

The simplest way to rebuild the *Default UI* in your project is to add one or both of the framework projects above as a module in your project. Here we use the `sodk-default-ui` as the example, but it's the same principle for `mupdf-default-ui`.

1. Remove the `sodk-default-ui.xcframework` from your project if it is currently included.
2. Copy the folder `sodk-default-ui` from the dev-kit into your project source code location.
3. Add `sodk-default-ui/sodk-default-ui.xcodeproj` to the same workspace as your application project. The framework project has been created, so all you have to do is drag it into your application project workspace. This video shows how to do this.
4. Change any import statements from `#import "sodk_default_ui/sodk_default_ui-Swift.h"``to ``#import "sodk_default_ui/sodk_default_ui.h"`
5. You should now be able to edit, rebuild and debug the *Default UI* code within your application workspace.

Alternatively, you can keep the *Default UI* frameworks as separate projects, change your code or assets, rebuild the projects and re-publish them using instructions from Apple's Developer documentation.

4.2.2 Custom UI

If you choose to implement your own *Custom UI* then it is up to you to define as much (or as little!) UI as you wish.

Firstly you should create a dedicated view controller which will be responsible for providing the *SmartOffice* document view controller.

Your dedicated view controller should at a minimum implement the following protocols `ARDKBasicDocViewDelegate` & `ARDKDocumentEventTarget`. For more on these protocols see *Document Delegates & Listeners*.

Your Custom UI View Controller

In order to successfully load a document from a `URL` and start a *SmartOffice* session an application developer requires to do the following in their custom `UIViewController`:

1. *Get the file type* for the document.
2. *Create an ARDKDefaultUrlFileState* instance with the file.
3. *Create the library instance* for *SmartOffice*.
4. *Define the document settings* for the *SmartOffice* session.
5. *Create a document session* with the required information.
6. *Create the correct type of document view controller* for the document type.
7. *Add the document view controller view* to your dedicated view.

See the *Full code example* for a quick reference.

1. Get the file type

Retrieving the file type as an enumeration of `ARDKDocType` is required in order for the file state object to correctly initialize.

Assuming you have a file URL the following code sample will get the file type as an *ARDKDocType* value.

Swift

Objective-C

```
guard url.startAccessingSecurityScopedResource() else {return} /// make the URL␣
↪accessible
let fileType:ARDKDocType = SODKDoc.docType(fromFileExtension: url.path)
url.stopAccessingSecurityScopedResource() /// close access to the URL
```

```
[url startAccessingSecurityScopedResource];
ARDKDocType fileType = [SODKDoc docTypeFromFileExtension:url.path];
[url stopAccessingSecurityScopedResource];
```

2. Create an `ARDKDefaultUrlFileState` instance

Now that you have the file type as an *ARDKDocType* value an `ARDKDefaultUrlFileState` should be created to allow for a temporary file state to handle file editing and saving.

The `ARDKDefaultUrlFileState` initializer supplies a closure method which delivers the file state object when ready as follows:

Swift

Objective-C

```
ARDKDefaultUrlFileState.fileState(for: url, of: fileType, whenReady:
↪{(fs:ARDKDefaultUrlFileState?) in
   if (fs != nil) {
      /// Create the SOLib with settings
   }
})
```

```
[ARDKDefaultUrlFileState fileStateForUrl:url ofType:fileType whenReady:^
↪(ARDKDefaultUrlFileState *fs) {
   if (fs) {
      /// Create the SOLib with settings
   }
}];
```

Note: If you require to create your own file state class, see the ARDKFileState interface.

3. Create the *SmartOffice* library instance

The library instance should have settings with a temporary folder for files defined before being instantiated.

Swift

Objective-C

```
let settings:ARDKSettings = ARDKSettings.init()
let tmp:URL = URL(fileURLWithPath: NSTemporaryDirectory()).appendingPathComponent(
↪"YourTempFolderName")
settings.temporaryPath = tmp.path

do {
   try FileManager.default.createDirectory(at: tmp, withIntermediateDirectories: true)
} catch {
   print("Failed to create tmp dir: \(error)")
}

let lib = SODKLib.init(settings: settings)
```

```
ARDKSettings *settings = [[ARDKSettings alloc] init];
NSURL *tmp = [[NSURL fileURLWithPath:NSTemporaryDirectory()]␣
↪URLByAppendingPathComponent:@"YourTempFolderName"];
settings.temporaryPath = tmp.path;

@try {
  NSError *error;

  [NSFileManager.defaultManager createDirectoryAtURL:tmp
        withIntermediateDirectories:YES
                         attributes:nil
                          error:&error];
}
@catch (id error) {
  NSLog(@"Failed to create tmp dir:%@",error);
}

SODKLib *lib = [[SODKLib alloc] initWithSettings:settings];
```

Note: There should only be one instance of `SODKLib` otherwise an error will be thrown.

4. Define the document settings

Session settings allow for enabling or disabling certain features for the document session. See *SODKDocumentSettings options* for a full listing of available settings.

Swift

Objective-C

```
let docSettings = SODKDocumentSettings.init()
docSettings.enableAll(true)
```

```
SODKDocumentSettings *docSettings = [SODKDocumentSettings init];
[docSettings enableAll:YES];
```

5. Create a document session

Now that we have the *SmartOffice* file state, library and document settings defined we are already to created the document session.

Swift

Objective-C

```
let session = SODKDocSession.init(fileState: fs!, ardkLib: lib, docSettings: docSettings)
```

```
SODKDocSession *session = [[SODKDocSession alloc] initWithFileState:fs ardkLib:lib␣
↪docSettings:docSettings];
```

6. Create the correct type of document view controller

Depending on the document type a developer should use either the *SODKBasicDocumentViewController* for *Office* type documents or the *MuPDFDKBasicDocumentViewController* for *PDF* type documents.

Therefore a developer should read the document type from the `ARDKDoc` instance on the session and then instantiate the correct type of view controller which can display the document accordingly:

Swift

Objective-C

```
switch session.doc.docType {

   /// Office documents
   case ARDKDocType_DOC,
        ARDKDocType_DOCX,
        ARDKDocType_PPT,
        ARDKDocType_PPTX,
        ARDKDocType_XLS,
        ARDKDocType_XLSX:

         let docVC = SODKBasicDocumentViewController(for:session)
         /// now we can reliably use this document view controller

      break

   /// PDF and others
   default:

         let docVC = MuPDFDKBasicDocumentViewController(for: session)
         /// now we can reliably use this document view controller
```

(continues on next page)

(continued from previous page)

```
        break

}
```

```
switch (session.doc.docType ) {

   /// Office documents
   case ARDKDocType_DOC:
   case ARDKDocType_DOCX:
   case ARDKDocType_PPT:
   case ARDKDocType_PPTX:
   case ARDKDocType_XLS:
   case ARDKDocType_XLSX:

      SODKBasicDocumentViewController *docVC = [[SODKBasicDocumentViewController alloc]␣
↪initForSession:session];
      /// now we can reliably use this document view controller

      break;

   /// PDF and others
   default:

      MuPDFDKBasicDocumentViewController *docVC = [[MuPDFDKBasicDocumentViewController␣
↪alloc] initForSession:session];
      /// now we can reliably use this document view controller

      break;

}
```

7. Add the document view controller

At this stage we should have the correct document view controller ready to be displayed with any defined settings and file information loaded. The final stage is to register delegates and to add the document view to our own custom `UIViewController`.

Within your instance of *Custom UI View Controller* use the following:

Swift

Objective-C

```
docVC.delegate = self
docVC.session.doc!.add(self)
self.addChild(docVC)
docVC.view.frame = self.view.bounds
self.view.addSubview(docVC.view)
docVC.didMove(toParent:self)
```

```
docVC.delegate = self;
[docVC.session.doc addTarget:self];
[self addChildViewController:docVC];
docVC.view.frame = self.view.bounds;
[self.view addSubview:docVC.view];
[docVC didMoveToParentViewController:self];
```

ARDKDocType values

ARDKDocType	View Controller
ARDKDocType_DOC ARDKDocType_DOCX ARDKDocType_PPT ARDKDocType_PPTX ARDKDocType_XLS ARDKDocType_XLSX	SODKBasicDocumentViewController
ARDKDocType_PDF ARDKDocType_CBZ ARDKDocType_FB2 ARDKDocType_SVG ARDKDocType_XPS ARDKDocType_EPUB	MuPDFDKBasicDocumentViewController

The ARDKBasicDocumentViewController base class

The document view controller will be an instance of SODKBasicDocumentViewController (for *Office* documents) or MuPDFDKBasicDocumentViewController (for *PDF* documents), both of these inherit from ARDKBasicDocumentViewController. The critical objects which a developer will need regular access to are:

- The session object (a read-only instance of ARDKDocSession)
- The document object (an instance of SODKDoc)

Throughout a typical *SmartOffice* an application developer will regularly have to access these objects to acheive certain tasks. This is as simple as referencing with dot-syntax as follows:

Swift

Objective-C

```
let docVC:ARDKBasicDocumentViewController = ARDKBasicDocumentViewController(for: session)
let session:ARDKDocSession = docVC.session
let doc:SODKDoc = docVC.doc
```

```
ARDKBasicDocumentViewController *docVC = [[ARDKBasicDocumentViewController alloc]␣
↪initForSession:session];
ARDKDocSession *session = docVC.session;
SODKDoc *doc = docVC.doc;
```

Document Page Selector

For a Custom UI context there is a handy drop-in UI available to enable a page viewer for your document. This drop-in UI understands how to display a column of page thumbnails from information obtained within the document session object. In order to use this UI, instantiate the ARDKPagesViewController class with the document session and add it to your UI with a frame layout as required.

ARDKPagesViewController(*session*)

Constructor method.

Parameters
session – SODKDocSession The session instance for the document.

selectPage(*pageNum*)

Selects the defined page from the page selector.

Parameters
pageNum – Int Note: this is zero-based, thus pageNum = 0 would select page 1 of your document.

Example code:

Swift

Objective-C

```
let pagesVC = ARDKPagesViewController(session: session)
pagesVC.view.frame = CGRect(x:0,y:0,width:80,height:self.view.frame.size.height)
pagesVC.didMove(toParent: self)
pagesVC.delegate = self
pagesVC.selectPage(0)
self.addChild(pagesVC)
view.addSubview(pagesVC.view)
```

```
ARDKPagesViewController *pagesVC = [ARDKPagesViewController␣
↪viewControllerWithSession:self.session];
pagesVC.view.frame = CGRectMake(0,0, 80, self.view.frame.size.height)
[pagesVC didMoveToParentViewController:self];
pagesVC.delegate = self;
[pagesVC selectPage:0];
[self addChildViewController:pagesVC];
[view addSubview:pagesVC.view];
```

The delegate protocol for the document page selector is the *ARDKPageSelectorDelegate* which will allow for user events on the drop-in UI to be intercepted.

4.2.3 SODKDocumentSettings options

Note some of settings will *only* apply to the visibility of UI buttons in the *Default UI*.

Variable	Type	Only for Default UI	Notes
editingEnabled	Bool	*no*	Enable or disable document editing.
saveButtonEnabled	Bool	*yes*	Enable or disable save.
saveAsEnabled	Bool	*yes*	Enable or disable save as.
openUrlEnabled	Bool	*no*	Enable or disable opening external URLs.
printingEnabled	Bool	*no*	Enable or disable document printing.
shareEnabled	Bool	*no*	Enable or disable document sharing.
fullScreenModeEnabled	Bool	*yes*	Enable or disable the full-screen view.
contentDarkModeEnabled	Bool	*no*	Enable or disable dark mode content inversion.
insertFromPhotosEnabled	Bool	*yes*	Enable or disable insertion of images from Photos.
insertFromCameraEnabled	Bool	*yes*	Enable or disable insertaion of image from the Camera.
trackChangesFeatureAvailable	Bool	*yes*	Enable or disable the track changes availability.
trackChangesFeatureEnabled	Bool	*no*	Enable or disable the track changes feature.
systemPasteboardEnabled	Bool	*no*	Enable or disable the system pasteboard for the document.
pdfAnnotationsEnabled	Bool	*yes*	Enable or disable PDF annotations.
pdfFormFillingAvailable	Bool	*no*	Enable or disable PDF form filling availability.
pdfFormFillingEnabled	Bool	*no*	Enable or disable PDF form filling.
pdfFormSigningEnabled	Bool	*no*	Enable or disable PDF form signing.
pdfRedactionAvailable	Bool	*yes*	Enable or disable PDF redaction availability.
pdfRedactionEnabled	Bool	*no*	Enable or disable PDF redactions.
pdfSignatureFieldCreationEnabled	Bool	*no*	Enable or disable PDF signature field creation.
pdfSecureRedactionAvailable	Bool	*yes*	Enable or disable PDF secure redaction availability.
pdfSecureRedactionEnabled	Bool	*no*	Enable or disable PDF secure redactions.
pdfFormESigningAvailable	Bool	*yes*	Enable or disable PDF electronic signing availability.
pdfFormESigningEnabled	Bool	*no*	Enable or disable PDF electronic signing.
pdfFormDigitalSigningAvailable	Bool	*yes*	Enable or disable PDF digital signing availability.
pdfFormDigitalSigningEnabled	Bool	*no*	Enable or disable PDF digital signing.

4.2.4 Code example

A full code sample for a session based *Default UI* or to get started with a *Custom UI* implementation is as follows:

Swift

Objective-C

```
/// 1. Get the file type
guard url.startAccessingSecurityScopedResource() else {return}
let type:ARDKDocType = SODKDoc.docType(fromFileExtension: url.path)
url.stopAccessingSecurityScopedResource()

/// 2. Create an ARDKDefaultUrlFileState instance
ARDKDefaultUrlFileState.fileState(for: url, of: type, whenReady:
↪{(fs:ARDKDefaultUrlFileState?) in
   if (fs != nil) {
```

(continues on next page)

(continued from previous page)

```
    /// 3. Create the SmartOffice library instance
    let settings:ARDKSettings = ARDKSettings.init()
    let tmp:URL = URL(fileURLWithPath: NSTemporaryDirectory()).appendingPathComponent(
↪"YourTempFolderName")
    settings.temporaryPath = tmp.path
    do {
       try FileManager.default.createDirectory(at: tmp, withIntermediateDirectories:␣
↪true)
    } catch {
       print("Failed to create tmp dir: \(error)")
    }

    let lib = SODKLib.init(settings: settings)

    /// 4. Define the document settings
    let docSettings = SODKDocumentSettings.init()
    docSettings.enableAll(true)

    /// 5. Create a document session
    let session = SODKDocSession.init(fileState: fs!, ardkLib: lib, docSettings:␣
↪docSettings)

    if <using_a_session_based_default_ui> {
       let docVC:ARDKDefaultUIViewController? = SODKDefaultUIViewController.
↪viewController(session: session)

       if docVC != nil {
          self.navigationController?.pushViewController(docVC!, animated: true)
       }
    } else { /// A Custom UI approach ...
       /// 6. Create the correct type of document view controller

       switch session.doc.docType {

          /// Office documents
          case ARDKDocType_DOC,
               ARDKDocType_DOCX,
               ARDKDocType_PPT,
               ARDKDocType_PPTX,
               ARDKDocType_XLS,
               ARDKDocType_XLSX:

                  let docVC:ARDKBasicDocumentViewController =␣
↪SODKBasicDocumentViewController(for:session)
                  /// now we can reliably use this document view controller
                  addDocVC(docVC:docVC)

                break

          /// PDF and others
          default:
```

(continues on next page)

(continued from previous page)

```
                    let docVC:ARDKBasicDocumentViewController =␣
↪MuPDFDKBasicDocumentViewController(for: session)
                    /// now we can reliably use this document view controller
                    addDocVC(docVC:docVC)

                break

            }
        }

        /// 7. Add the document view controller
        func addDocVC(docVC:ARDKBasicDocumentViewController) {
            docVC.delegate = self
            docVC.session.doc!.add(self)
            self.addChild(docVC)
            docVC.view.frame = self.view.bounds
            self.view.addSubview(docVC.view)
            docVC.didMove(toParent:self)
        }
    }
})
```

```
/// 1. Get the file type
[url startAccessingSecurityScopedResource];
ARDKDocType fileType = [SODKDoc docTypeFromFileExtension:url.path];
[url stopAccessingSecurityScopedResource];

/// 2. Create an ARDKDefaultUrlFileState instance
[ARDKDefaultUrlFileState fileStateForUrl:url ofType:fileType whenReady:^
↪(ARDKDefaultUrlFileState *fs) {
   if (fs) {
      /// 3. Create the SmartOffice library instance
      ARDKSettings *settings = [[ARDKSettings alloc] init];
      NSURL *tmp = [[NSURL fileURLWithPath:NSTemporaryDirectory()]␣
↪URLByAppendingPathComponent:@"YourTempFolderName"];
      settings.temporaryPath = tmp.path;

      @try {
        NSError *error;

        [NSFileManager.defaultManager createDirectoryAtURL:tmp
              withIntermediateDirectories:YES
                               attributes:nil
                                error:&error];
      }
      @catch (id error) {
        NSLog(@"Failed to create tmp dir:%@",error);
      }

      SODKLib *lib = [[SODKLib alloc] initWithSettings:settings];

      /// 4. Define the document settings
```

(continues on next page)

(continued from previous page)

```
    SODKDocumentSettings *docSettings = [SODKDocumentSettings init];
    [docSettings enableAll:YES];

    /// 5. Create a document session
    SODKDocSession *session = [[SODKDocSession alloc] initWithFileState:fs ardkLib:lib␣
↪docSettings:docSettings];

    if (<using_a_session_based_default_ui>) {
       SODKBasicDocumentViewController *docVC = [[SODKBasicDocumentViewController␣
↪alloc] initForSession:session];

       if (docVC != NULL) {
          [self.navigationController pushViewController:docVC animated:true];
       }

    } else { /// A Custom UI approach ...
       /// 6. Create the correct type of document view controller
       switch (session.doc.docType) {

          /// Office documents
          case ARDKDocType_DOC:
          case ARDKDocType_DOCX:
          case ARDKDocType_PPT:
          case ARDKDocType_PPTX:
          case ARDKDocType_XLS:
          case ARDKDocType_XLSX:
             {
                ARDKBasicDocumentViewController *docVC =␣
↪[SODKBasicDocumentViewController viewControllerForSession:session];
                /// now we can reliably use this document view controller
                /// 7. Add the document view controller
                docVC.delegate = self;
                [docVC.session.doc addTarget:self];
                [self addChildViewController:docVC];
                docVC.view.frame = self.view.bounds;
                [self.view addSubview:docVC.view];
                [docVC didMoveToParentViewController:self];
             }
             break;

          /// PDF and others
          default:
             {
                ARDKBasicDocumentViewController *docVC =␣
↪[MuPDFDKBasicDocumentViewController viewControllerForSession:session];
                /// now we can reliably use this document view controller
                /// 7. Add the document view controller
                docVC.delegate = self;
                [docVC.session.doc addTarget:self];
                [self addChildViewController:docVC];
                docVC.view.frame = self.view.bounds;
                [self.view addSubview:docVC.view];
```

(continues on next page)

(continued from previous page)

```
                    [docVC didMoveToParentViewController:self];
                }
                break;

            }
        }

    }

}];
```

4.3 Document Delegates & Listeners

Document listeners should only be required when using the *Custom UI* as the application developer is responsible for providing their own UI to manage relevant document events.

4.3.1 ARDKBasicDocViewDelegate

loadingAndFirstRenderComplete()

Called when the document has completely loaded and the first page has rendered.

updateUI()

Called when the document changes selection state and the UI should update appropriately.

viewDidMove()

The following method is called when the document is moved - i.e. on pan or zoom events.

viewDidScroll(*page*)

Called on a page change event - i.e. when the document has scrolled or jumped to another page.

Parameters
page – Int The page which has been scrolled to, zero-based.

Swallowing taps

An application developer has the ability to intercept tap events on the document to prevent a default selection from occurring. To do this the following delegate methods need to return *true* to "swallow" the events.

swallowSelectionTap()

Return *false* to allow the document view to interpret the event.

Return *true* to block the document view from interpreting the event.

Returns
Bool.

swallowSelectionDoubleTap()

Return *false* to allow the document view to interpret the event.

Return *true* to block the document view from interpreting the event.

Returns
Bool.

inhibitKeyboard()

Return *false* to allow the keyboard to appear.

Return *true* to inhibit the keyboard.

Returns
`Bool`.

callOpenUrlHandler(_ *url: URL, fromVC presentingView: UIViewController*)

This method is called when the document interaction invokes a URL to open.

Parameters

- **url** – `URL` The URL which has been requested to open.
- **presentingView** – `UIViewController` The presenting `UIViewController`.

4.3.2 ARDKDocumentEventTarget

This delegate is responsible for page & selection events.

updatePageCount(*pageCount*, *complete*)

Called as pages are registered as loaded from the document.

Parameters

- **pageCount** – `Int` The currently loaded page count.
- **complete** – `Bool` Returns *true* when the document has completely loaded.

Note: On small documents with little size or pages, the individual pages may not register and this listener may just trigger the final page with `complete` registered as *true*. *e.g.* A small 3 page document may just return as `pageCount = 3` , `complete = true`.

---

layoutHasCompleted()

Called when layout has completed.

pageSizeHasChanged()

Called when the page size changes.

selectionHasChanged()

Called when a selection is made within the document, moved or removed.

4.3.3 ARDKPageSelectorDelegate

This is used in conjunction with the *Document Page Selector* drop-in UI.

selectPage(*page*)

Parameters
page – `Int` The page number to select, zero-based.

deletePage(*page*)

Parameters
page – `Int` The page number to delete, zero-based.

duplicatePage(*page*)

Parameters
page – Int The page number to duplicate, zero-based.

movePage(*page*, *newPos*)

Parameters

- **page** – Int The page number to move, zero-based.
- **newPos** – Int The new position of the page number, zero-based.

4.3.4 Document Listeners

Document load

An application developer can listen for basic `success` or `error` for a document load.

When a document load is requested, the following function blocks can be defined for the document (i.e. the instance of `ARDKDoc`).

Swift

Objective-C

```
doc.successBlock = {

}

doc.errorBlock = {(error:ARDKDocErrorType?) in

}
```

```
doc.successBlock = ^() {

};

doc.errorBlock = ^(ARDKDocErrorType error) {

};
```

Request password

To listen for password requests, on the event that the selected document is password protected, a developer should set a method against the document session to be triggered as follows.

Swift

Objective-C

```
session.passwordRequestBlock = {[weak self] in

}
```

```
session.passwordRequestBlock = ^{

};
```

4.4 Document API - All Main Document Types

Main document types

Word	
Presentation	
Spreadsheet	
PDF	

These methods apply to document types referenced above. Note: *SmartOffice* supports loading image files as well as some more obscure document types, however, for these kind of documents the methods outlined below will probably *not* be relevant.

The methods and variables within the `ARDKBasicDocumentViewController` & `ARDKDoc` classes should be considered for the API as outlined below.

4.4.1 File Operations

There are a number of file operations available against a document, an application developer should only be required to implement these if using the *Custom UI*.

Saving a document only needs to be invoked if there are changes made to a document. As such an application developer can verify if changes have been made or not and act accordingly. The *ARDKBasicDocumentViewController* `session` can be used to save the document with a callback method for the completion.

Properties & methods which relate to file operations can be found on the `ARDKDoc` instance as follows:

`property docType`

Property on the `ARDKDoc` instance which stores the current document type.

Type
`ARDKDocType` See *ARDKDocType values*.

`property isBeingSaved`

Property on the `ARDKDoc` instance which holds whether the document is being saved.

Type
`Bool`.

`saveDocumentAnd`(*onCompletion*)

Saves the document with a completion callback.

Saving a document only needs to be invoked if there are changes made to a document. As such, an application developer can *verify if changes have been made or not* and act accordingly.

Parameters
`onCompletion` – `(ARDKSaveResult, ARError) -> Void` The completion callback function.

save(*to*, *completion*)

This is a "Save As" method which takes a file path and completion callback.

Parameters

- **to** – `String` The file path to save to.
- **completion** – `(ARDKSaveResult, ARError) -> Void` The completion callback function.

property hasBeenModified

Property which stores whether the document has been modified.

Type
`Bool`.

When opening a password protected file a developer should supply a method for a user to enter a password as follows:

providePassword(*password*)

Method which allows for document password to be provided.

Parameters
password – `String` The password string.

Note: Also see the *password request listener* which should be programmed to prompt the user to provide the password.

4.4.2 File State

The `ARDKFileState` interface should be adhered to if an application developer requires to create their own class to handle file operations and open documents using the session method.

This class interface methods are detailed as follows:

sessionDidLoadFirstPage(*session*)

Information method called when a session has loaded the first page of the document.

Parameters
session – `ARDKDocSession` The document session.

sessionDidShowDoc(*session*)

Information method called when the file is opened in the main document view ready for viewing and editing by the user.

Parameters
session – `ARDKDocSession` The document session.

sessionDidSaveDoc(*session*)

Information method called when a session saves document edits back to the supplied file.

Parameters
session – `ARDKDocSession` The document session.

sessionRequestedCopyBackOnCompletion(*block*)

In some use cases, an app may supply a copy of the file to be edited, which will require copying back after any edits have been saved. This method will be called when copying back may be necessary. For apps that supply the original file directly, and return *false* from the `requiresCopyBack` variable this method need do nothing.

Parameters
block – `@escaping (Bool) -> Void)` The completion block.

`sessionDidClose()`

Information method called when a session ends. In the case that an app supplies a copy of a file to be edited. This method might delete the copy, since the session is no longer using it. The file should NOT be copied back before removal.

4.4.3 Printing

Application developers should firstly instantiate a native `UIPrintInteractionController` and pass through a page renderer with `ARDKPrintPageRenderer` which has been initialized with the current document.

Swift

Objective-C

```
let printController:UIPrintInteractionController = UIPrintInteractionController.shared
let pageRenderer:ARDKPrintPageRenderer = ARDKPrintPageRenderer(document:session.doc)
printController.printPageRenderer = pageRenderer

printController.present(animated: true,
    completionHandler: { (printInteractionController:UIPrintInteractionController,
        completed:Bool,
        error:Error?) in
    if error != nil {
        print("Print failed due to error:\(error!)")
    }
})
```

```
UIPrintInteractionController *printController = [UIPrintInteractionController␣
↪sharedPrintController];
ARDKPrintPageRenderer *pageRenderer = [[ARDKPrintPageRenderer alloc]␣
↪initWithDocument:session.doc];
printController.printPageRenderer = pageRenderer;
[printController presentAnimated:YES
    completionHandler:^(
        UIPrintInteractionController * _Nonnull printInteractionController,
        BOOL completed,
        NSError * _Nullable error) {
    if (error) {
        NSLog(@"Print failed due to error %@", error);
    }
}];
```

4.4.4 Search

Searching can be made forward or backward from a defined point in the document. Successful searching automatically highlights the next instance of a found string and moves the document selection to that point.

Note: Search is case-insensitive.

Methods can be found on the `ARDKDoc` instance as follows:

setSearchStartPage(*page*, *offset*)

Sets the search start page with any offset point.

Parameters

- **page** – `Int` Page number, zero-based.
- **offset** – `CGPoint` An offset point to start the page search from, can be `CGPoint.zero` (which would equal the top of any given page).

search(*for*, *in*, *onEvent*)

Starts the search operation.

Parameters

- **for** – `String` The search string.
- **in** – `ARDKSearchDirection` The direction to search in.
- **onEvent** – `(ARDKSearchEvent, Int, CGRect) -> Void` The search event function block callback.

Sample code:

Swift

Objective-C

```
let doc:ARDKDoc = session.doc

// Starts searching the document from the first page
doc.setSearchStartPage(0, offset: .zero)

// search forward from this point for the word "hello" and
// highlight the found occurrence
doc.search(for: "hello",
      in:MuPDFDKSearch_Forwards,
      onEvent:{event, page, area in

        switch event {

            case MuPDFDKSearch_Progress:
                // If we had a progress indicator, we could set it here according
                // to where page is between 0 and self.doc.pageCount
            break

            case MuPDFDKSearch_Found:
                self.updateUI()
                // Pan to show the found occurrence
                self.docViewController.showArea(area, onPage: page)
            break

            case MuPDFDKSearch_NotFound:
                self.updateUI()
                // Could ask the user here whether to restart the search from
                // the start of the document
            break
```

(continues on next page)

(continued from previous page)

```
            case MuPDFDKSearch_Cancelled:
                self.updateUI()
            break

            case MuPDFDKSearch_Error:
                self.updateUI()
            break

            default:
                self.updateUI()
            break

        }
    }
)
```

```
ARDKDoc *doc = session.doc;

// Starts searching the document from the first page
[doc setSearchStartPage:0 offset:CGPointZero];

// search forward from this point for the word "hello" and
// highlight the found occurrence
[doc searchFor:@"hello"
        inDirection:MuPDFDKSearch_Forwards
          onEvent:^(MuPDFDKSearchEvent event, NSInteger page, CGRect area) {
    switch (event)
    {
        case MuPDFDKSearch_Progress:
            // If we had a progress indicator, we could set it here according
            // to where page is between 0 and self.doc.pageCount
            break;

        case MuPDFDKSearch_Found:
            [self updateUI];
            // Pan to show the found occurrence
            [self.docViewController showArea:area onPage:page];
            break;

        case MuPDFDKSearch_NotFound:
            [self updateUI];
            // Could ask the user here whether to restart the search from
            // the start of the
            break;

        case MuPDFDKSearch_Cancelled:
            [self updateUI];
            break;

        case MuPDFDKSearch_Error:
            [self updateUI];
            break;
```

(continues on next page)

(continued from previous page)

```
      }
   }
];
```

4.4.5 Undo/Redo

Properties & methods can be found on the `ARDKDoc` instance as follows:

property canUndo

Gets whether there is an action which can be undone.

Type
`Bool` *Read-only*.

property canRedo

Gets whether there is an action which can be redone.

Type
`Bool` *Read-only*.

undo()

Undo a previous action.

redo()

Redo a previous action.

4.4.6 Pages

Properties & methods can be found on the `ARDKBasicDocumentViewController` instance as follows:

property currentPage

Variable on the `ARDKBasicDocumentViewController` instance which holds the current page.

Type
`Int` The current page number, zero-based. *Read-only*.

Once a document is loaded an application developer can view pages either by scrolling the document view or by using the *SmartOffice* API as follows:

showPage(*pageNum*)

Moves the document to the defined page. Animates by default.

Parameters
pageNum – `Int` Note: this is zero-based, thus `pageNum = 0` would show page 1 of your document.

showPage(*pageNum*, *animate*)

Moves the document to the defined page.

Parameters

- **pageNum** – `Int` Page number, zero-based.
- **animate** – `Bool` Whether to animate the transition to the new page or not.

showPage(*pageNum*, *animate*, *onCompletion*)

Moves the document to the defined page.

Parameters

- **pageNum** – `Int` Page number, zero-based.
- **animate** – `Bool` Whether to animate the transition to the new page or not.
- **onCompletion** – `() -> Void` Completion method.

showPage(*pageNum*, *withOffset*)

Pan a specified position within a page to the top left of the screen.

Parameters

- **pageNum** – `Int` Page number, zero-based.
- **withOffset** – `CGPoint` The offset point.

showPage(*pageNum*, *withOffset*, *animated*)

Pan a specified position within a page to the top left of the screen.

Parameters

- **pageNum** – `Int` Page number, zero-based.
- **withOffset** – `CGPoint` The offset point.
- **animate** – `Bool` Whether to animate the transition or not.

showArea(*rect*, *page*)

Updates the view to make a specific area of the page visible.

Parameters

- **rect** – `CGRect` The area to show.
- **page** – `Int` The page number.

setupPageCell(*pageCell*, *forPage*)

Called to allow the delegate to create the page view and add it to the cell as a subview.

Parameters

- **pageCell** – `ARDKPageCell` the page cell instance.
- **forPage** – `Int` The page number.

iteratePages(*block*)

Iterates through each document page with the supplied function `block`.

Parameters
block – `(Int, (UIView & ARDKPageCellDelegate)?, CGRect) -> Void`

adjust(*size*, *toPage*)

Called to adjust the size of a cell to match the aspect ratio of the page.

Parameters

- **size** – `CGSize` The size of the cell.
- **toPage** – `Int` The page number.

page(*toScreen*)

Returns the transform matrix for a given page. An example usage may be to help with positioning views near a selection.

Parameters
toScreen – Int The page number.

Returns
CGAffineTransform The transform matrix.

showEnd(*ofPage*)

Pan the document view to show the bottom of the page.

Parameters
ofPage – Int The page number.

Properties & methods can be found on the ARDKDoc instance as follows:

property pageCount

Variable on the ARDKDoc instance which holds the page count.

Type
Int *Read-only*.

property docSupportsPageManipulation

Variable on the ARDKDoc instance which stores whether the document supports page manipulation or not.

Type
Bool *Read-only*.

These "page manipulation" methods will work if docSupportsPageManipulation is *true* (which is mostly used for *Spreadsheet* document contexts) and if they are requested after the document listener for *loadingAndFirstRenderComplete()* has triggered.

deletePage(*pageNumber*)

Deletes a page. Only supported for Powerpoint and Excel documents.

Parameters
pageNumber – Int The page number to delete, zero-based.

duplicatePage(*pageNumber*)

Duplicates a page. Only supported for Powerpoint documents.

Parameters
pageNumber – Int The page number to duplicate, zero-based.

movePage(*pageNumber*, *to*)

Moves a page from one location to another page location. Only supported for Powerpoint and Excel documents.

Parameters

- **pageNumber** – Int The page number to move, zero-based.
- **to** – Int The new page number to move to, zero-based.

addBlankPage(*pageNumber*)

Adds a new page if pageNumber is valid. Only supported for Excel documents.

Parameters
pageNumber – Int The page number used to add the new page to, zero-based.

`getPage`(*pageNumber*)

Returns a page if `pageNumber` is valid.

Parameters
pageNumber – `Int` The page number used to retrieve the page from the document, zero-based.

Returns
`ARDKPage` or `nil`.

`getPage`(*pageNumber*, *update*)

Returns a page if `pageNumber` is valid. Also includes an `update` callback method.

Parameters

- **pageNumber** – `Int` The page number used to retrieve the page from the document, zero-based.
- **update** – `((CGRect) -> Void)` The update callback method with a `CGRect` parameter.

Returns
`ARDKPage` or `nil`.

Note: See the *Document Page Selector* for the page thumbnailer control.

Page history

Properties & methods can be found on the `ARDKBasicDocumentViewController` instance as follows:

`property viewingStateNextAllowed`

Returns whether the document has a next entry in the page history.

Type
`Bool` *Read-only*.

`property viewingStatePreviousAllowed`

Returns whether the document has a previous entry in the page history.

Type
`Bool` *Read-only*.

`viewingStateNext()`

Goes to the next position in the page history.

`viewingStatePrevious()`

Goes to the previous position in the page history.

4.4.7 Viewing Full-screen

In order to view a document in full-screen, it is up to the application developer to hide any UI which they have present and set the frame of the document view to fill the screen. Once in full-screen, you can use *Tap selections* to exit and return from the full-screen frame to any previous UI.

4.4.8 Document Selections

Properties & methods can be found on the `ARDKBasicDocumentViewController` instance as follows:

`property selectionInfo`

Gets selection info on a selected object if applicable.

Type
`ARDKSelectionInfo` *Read-only*.

For more on document selection operations see the respective API for `SODKBasicDocumentViewController` & `SODKDoc` and `MuPDFDKBasicDocumentViewController` & `MUPDFDKDoc`.

4.4.9 Document Positioning and Scaling

Properties & methods can be found on the `ARDKBasicDocumentViewController` instance as follows:

`setZoomScale`(*scale*, *animated*)

Sets the zoom scale for the document.

Parameters

- **`scale`** – `Float` The zoom sacle.
- **`animated`** – `Bool` Whether to animate the zoom transition or not.

4.5 Document API - *Office*

Office document types

Word		• **docx** • **doc** • **dotx** • **docm** • **dotx** • **odt**
Presentation		• **pptx** • **ppt** • **pps** • **pptm**
Spreadsheet		• **xlsx** • **xls** • **xlt** • **xlsm** • **xltm**

The document types referenced above will be handled by the `SODKBasicDocumentViewController` & `SODKDoc` classes.

The properties & methods within these classes should be considered for the API as outlined below.

Note: *SmartOffice* supports loading image files as well as some more obscure document types, however, for these kind of documents the methods outlined below will probably *not* be relevant.

4.5.1 Selections and Styling

Standard styling

`SODKDoc` properties & methods:

`property selectionIsBold`

Getter and setter for whether the selected text is bold or not.

Type
`Bool`.

`property selectionIsItalic`

Getter and setter for whether the selected text is italic or not.

Type
`Bool`.

`property selectionIsUnderlined`

Getter and setter for whether the selected text is underlined or not.

Type
`Bool`.

`property selectionIsLinethrough`

Getter and setter for whether the selected text is strike-through or not.

Type
`Bool`.

Bullet lists

`SODKDoc` property:

`property selectionListStyle`

Getter and setter for the selection list style.

Type
`SODKListStyle` Available options are: `SODKListStyle_Disc` `SODKListStyle_Decimal` `SODKListStyle_None`.

Alignment

SODKDoc properties:

property selectionTextAlign

Getter and setter for the horizontal text alignment value.

Type
SODKTextAlign Available options are: SODKTextAlign_Left SODKTextAlign_Right SODKTextAlign_Center SODKTextAlign_Justify.

property selectionTextAlignV

Getter and setter for the vertical text alignment value.

Type
SODKTextAlignV Available options are: SODKTextAlignV_Top SODKTextAlignV_Bottom SODKTextAlignV_Center.

Indentation

SODKDoc properties:

property selectionIndentationLevel

Getter and setter for the indentation level on the current text selection.

Type
Int32.

property selectionMaxIndentationLevel

Gets the maximum indentation level for the selection.

Type
Int32 *Read-only*.

Note: Indentation levels only apply to text selections in *Word* documents.

Delete, Cut, Copy, Paste & Clipboard

SODKDoc properties:

property selectionCanBeCopied

Gets whether the current selection can be copied.

Type
Bool *Read-only*.

property selectionCanBeDeleted

Gets whether the current selection can be deleted.

Type
Bool *Read-only*.

property selectionCanBePasteTarget

Gets whether the current selection can be pasted onto.

Type
Bool *Read-only*.

property clipboardHasData

Gets whether the clipboard has data or not.

Type
`Bool` *Read-only*.

selectionDelete()

Deletes the selection.

selectionCutToClip()

Cuts a selection to the clipboard.

selectionCopyToClip()

Copies a selection to the clipboard.

pasteClipboard()

Pastes from the clipboard to/over the current selection.

Object positioning and metrics

Objects other than text (i.e. annotations and shapes) have the following properties:

`SODKDoc` properties:

property selectionCanBeRotated

Gets if the selection object can be rotated or not.

Type
`Bool` *Read-only*.

property selectionCanBeResized

Gets if the selection object can be resized or not.

Type
`Bool` *Read-only*.

property selectedObjectBounds

Getter and setter for object bounds.

Type
`CGRect`.

property selectedObjectPosition

Getter and setter for object position.

Type
`CGPoint`.

property selectedObjectRotation

Getter and setter for object rotation.

Type
`Float`.

property selectedObjectNaturalDimensions

Gets the natural dimensions of the selected object.

Type
`CGSize` *Read-only*.

Other

`SODKDoc` properties & methods:

property selectionIsAlterableTextSelection

Gets whether the text selection can be modified or not.

Type
`Bool` *Read-only*.

property selectionIsAutoshapeOrImage

Gets whether the selection is an autoshape or image.

Type
`Bool` *Read-only*.

property selectionIsAlterableTextSelection

Gets whether the current selection is a text selection that can be extended or contracted.

Type
`Bool` *Read-only*.

property selectionIsAlterableAnnotation

Gets whether the current selection is an annotation selection that can be altered.

Type
`Bool` *Read-only*.

property selectionPermitsInlineTextEntry

Gets whether the current selection can accept direct text entry (i.e. without requiring a virtual keyboard or other intermediary).

Type
`Bool` *Read-only*.

property selectionHasAssociatedPopup

Gets whether the text selection has pop-up or not.

Type
`Bool` *Read-only*.

property selectionCanHaveTextStyleApplied

Gets whether the text selection style can be modified or not.

Type
`Bool` *Read-only*.

property selectionCanHavePictureInserted

Gets whether the current selection can have a picture/image inserted into it.

Type
`Bool` *Read-only*.

selectionExists()

Returns whether there an active selection or not.

Returns
`Bool`.

`SODKBasicDocumentViewController` properties & methods:

property selectionIsActive

Gets whether there is a currently active selection.

Type
`Bool` *Read-only*.

property selectionIsCaret

Gets whether there is a caret selection.

Type
`Bool` *Read-only*.

showSelectionAfterLayout()

Pan to the selection after layout has completed.

showSelectionInfoViewFocussed()

When showing a selection with an associated info view, give the info view focus.

4.5.2 Media

Drawing

Properties & methods can be found on the `SODKBasicDocumentViewController` instance as follows:

property annotatingMode

Getter and setter for the document's current drawing mode.

Type
`SODKAnnotatingMode`.

Available types:

`SODKAnnotatingMode_None` `SODKAnnotatingMode_Draw` `SODKAnnotatingMode_Note` `SODKAnnotatingMode_Highlight`

A document can enter certain modes which define the user interaction mode with the document. When an annotating mode is set tapping and moving will respect any current document annotation mode.

`SODKDoc` properties & methods:

property selectionLineWidth

Getter and setter for the line width for the current selection.

Type
`CGFloat`.

property selectionLineStyle

Getter and setter for the line style for the current selection.

Type
`SODKLineStyle`.

Available types:

`SODKLineStyle_Solid` `SODKLineStyle_DotSys` `SODKLineStyle_DotGEL` `SODKLineStyle_DashSys` `SODKLineStyle_DashGEL` `SODKLineStyle_LongDashGEL` `SODKLineStyle_DashDotSys` `SODKLineStyle_DashDotGEL` `SODKLineStyle_LongDashDotGEL` `SODKLineStyle_DashDotDotSys` `SODKLineStyle_LongDashDotDotGEL`

Note: See *color API* for color styling options.

Images

SODKDoc properties & methods:

property selectionCanHavePictureInserted

Gets whether the selection can have an image inserted.

Type
Bool *Read-only*.

property selectionCanHaveShapeInserted

Gets whether the selection can have a shape inserted.

Type
Bool *Read-only*.

insertImage(*path*)

Insert an image at the caret or replace a current selection. Should not be called if there is no current selection.

Parameters
path – String The file path of the image file to insert.

insertImage(*path*, *pageNum*)

Inserts an image centrally on the page - for use with *Presentation* documents (PPT, PPTX).

Parameters

- **path** – String The file path of the image file to insert.
- **pageNum** – Int The page number to insert the image on, zero-based.

4.5.3 Fonts

SODKDoc properties & methods:

property fontList

Gets an array of available fonts or nil.

Type
[String]? *Read-only*.

property selectionFontName

Setter & getter for the font name against the currently selected text.

Type
String The name of the font.

property selectionFontSize

Setter & getter for the font size against the currently selected text.

Type
Int32 The font size.

4.5.4 Color

`SODKBasicDocumentViewController` properties & methods:

property inkAnnotationColor

Getter and setter for the drawing ink color.

Type
`UIColor`.

property inkAnnotationThickness

Getter and setter for the drawing ink thickness.

Type
`Float`.

`SODKDoc` properties & methods:

property selectionColor

Getter & setter for the selected text color.

Type
`UIColor?` Sets/gets a *UIColor* ` or `nil`.

property selectionBackgroundColor

Getter & setter for the selected text background color.

Type
`UIColor?` Sets/gets *UIColor* ` or `nil`.

property backgroundColorList

Gets the available list of colors for the permitted document background colors.

Type
`[UIColor]?` Returns an array of `UIColor` objects or `nil`. *Read-only*.

Note: Certain documents (e.g. `docx`) may have restrictions for background colors. This property getter will indicate what those might be.

---

property selectionCanHaveForegroundColourApplied

Gets whether the current selection can have its text color changed.

Type
`Bool` *Read-only*.

property selectionCanHaveBackgroundColourApplied

Gets whether the current selection can have its text background color changed.

Type
`Bool` *Read-only*.

These properties apply to drawing annotations and/or shapes:

property selectionLineColor

Getter & setter for the line color of the current selection.

Type
`UIColor?` Sets/gets a *UIColor* ` or `nil`.

property selectionFillColor

Getter & setter for the fill color of the current selection.

Type
`UIColor?` Sets/gets a *UIColor* ` or `nil`.

property selectionLineOpacity

Getter & setter for the line opacity of the current selection.

Type
`Int32` Sets/gets the opacity (0-100).

property selectionFillOpacity

Getter & setter for the fill opacity of the current selection.

Type
`Int32` Sets/gets the opacity (0-100).

4.5.5 Exporting a file as *PDF*

Office type documents can be exported to *PDF* using the following method:

The export method can be found on the `SODKDoc` instance as follows:

exportPDF(*to*, *flags*, *completion*)

Exports the file as a *PDF* to a specific location with a completion block.

Parameters

- **to** – `String` The file path to export to.
- **flags** – `SmartOfficeSaveFlags` Setting this reduces the need for a temporary file (and also avoids cases where the save fails because the destination file and *SmartOffice's* temporary file are the same file). Can be `nil` or `SmartOfficeSaveFlags_NoTemporary`.
- **completion** – `(ARDKSaveResult, SOError) -> Void` Completion block.

Sample code:

Swift

Objective-C

```
let doc:SODKDoc = session.doc as SODKDoc

doc.exportPDF(to: <file_path>, flags: SmartOfficeSaveFlags_NoTemporary, completion: {␣
↪(result:ARDKSaveResult, error:SOError) in
   switch result {

   case ARDKSave_Succeeded:

       break
```

(continues on next page)

(continued from previous page)

```
   case ARDKSave_Cancelled:

       break

   case ARDKSave_Error:

       break

   default:

       break
   }
})
```

```
SODKDoc *doc = (SODKDoc*)session.doc;

[doc exportTo:<file_path> flags: SmartOfficeSaveFlags_NoTemporary, completion:^
↪(ARDKSaveResult result, SOError error) {
   switch (result)
   {
        case ARDKSave_Succeeded:
            break;

        case ARDKSave_Cancelled:
        case ARDKSave_Error:

            break;
   }
}];
```

4.6 Document API - *Word*

The methods outlined below should be considered as the specific API for the following document file types:

- **docx**
- **doc**
- **dotx**
- **docm**
- **dotx**
- **odt**

4.6.1 Reflow

SODKDoc properties & methods:

property flowMode

Getter and setter for the document flow mode.

Type
SmartOfficeFlowMode.

Available types:

SmartOfficeFlowMode_Normal SmartOfficeFlowMode_Reflow
SmartOfficeFlowMode_ResizePage

property reflowWidth

Getter and setter for the document reflow width.

Type
Float.

property reflowHeight

Getter and setter for the document reflow height.

Type
Float.

4.6.2 Track changes and reviewing

If your application requires track changes and reviewing functionality for documents you should consider the following SODKDoc properties & methods:

property docSupportsReview

Gets whether the document supports the review feature.

Type
Bool *Read-only*.

property trackingChanges

Setter and getter for tracking changes for the document.

Type
Bool

property showingTrackedChanges

Setter and getter for showing tracking changes for the document or not.

Note: *Showing* the changes is a visual aspect of tracking changes and is independent of setting tracked changes.

Type
Bool

property selectionIsReviewable

Gets whether the current selection can be tracked.

Type
Bool *Read-only*.

property trackedChangeId

Gets the type of the currently selected track change.

Type
`Int` *Read-only*.

property trackedChangeDate

Gets a string representation of the date of the currently selected track change.

Type
`String` *Read-only* in format "dd/mm/yyyy hh:mm:ss".

property trackedChangeComment

Gets any comment against the currently selected track change.

Type
`String?` *Read-only*.

property trackedChangeAuthor

Gets the author of the currently selected track change.

Type
`String?` *Read-only*.

property trackedChangeType

Gets the type of the currently selected track change.

Type
`SmartOfficeTrackedChangeType` *Read-only*.

Available types:

`SmartOfficeTrackedChangeType_NoChange` `SmartOfficeTrackedChangeType_DeletedText`
`SmartOfficeTrackedChangeType_InsertedText` `SmartOfficeTrackedChangeType_InsertedParagraph`
`SmartOfficeTrackedChangeType_InsertedTableCell` `SmartOfficeTrackedChangeType_InsertedTableRow`
`SmartOfficeTrackedChangeType_ChangedParagraphProperties`
`SmartOfficeTrackedChangeType_ChangedRunProperties` `SmartOfficeTrackedChangeType_ChangedSectionPr`
`SmartOfficeTrackedChangeType_ChangedTableRowProperties`
`SmartOfficeTrackedChangeType_ChangedTableCellProperties`
`SmartOfficeTrackedChangeType_ChangedTableProperties` `SmartOfficeTrackedChangeType_ChangedTableGr`

acceptTrackedChange()

Accepts a selected tracked change.

Only has effect if tracking of changes is enabled and if a tracked change is selected on the document.

rejectTrackedChange()

Rejects a selected tracked change.

Only has effect if tracking of changes is enabled and if a tracked change is selected on the document.

addHighlightAnnotation()

This adds a highlight annotation to the seleccted text along with a review comment.

Only has effect if tracking of changes is enabled.

previousTrackedChange()

Moves the selection to the previous tracked change, if any.

Only has effect if tracking of changes is currently *shown*.

nextTrackedChange()

Moves the selection to the next tracked change, if any.

Only has effect if tracking of changes is currently *shown*.

4.7 Document API - *Presentation*

The methods outlined below should be considered as the specific API for the following document file types:

- **pptx**
- **ppt**
- **pps**
- **pptm**

4.7.1 Autoshapes

Adding shapes

There are 13 available *Autoshapes* which can be inserted onto a page when the `insertAutoshape` function is invoked. Additionally there are styling options which can be applied with the `properties` parameter as well as further placement options.

`SODKDoc` properties and methods:

property selectionCanHaveShapeInserted

Gets whether a current selection can have a shape inserted.

Type
`Bool` *Read-only*.

insertAutoshape(*shape*, *properties*, *onPage*, *atPosition*, *orCentrally*)

Insert an *AutoShape* with the given properties, on a given page, at a given position or in the centre of the page.

Parameters

- **shape** – `String` shape to insert. Possible values are:
  - `TextBox`
  - `Line`
  - `RectRoundRect`
  - `Ellipse`
  - `Triangle`
  - `RtTriangle`
  - `Arrow`
  - `LeftRightArrow`
  - `Diamond`
  - `WedgeEllipseCallout`
  - `WedgeRectCallout`

- Pentagon
- Star

- **properties** – String A string of properties to apply. Can be nil. A common case is to supply a *"TextBox"* with property *"fill-color:transparent"*, or a *"Line"* with property *"end-decoration:"arrow""* to add an arrow head to the end.
- **onPage** – Int Page number to insert autoshape into, zero-based.
- **atPosition** – CGPoint If orCentrally is *false* then this paoint will be used as the point to place the shape.
- **orCentrally** – Bool Set to *true* to place the shape in the centre of the supplied page, *false* to use atPosition.

Arranging shapes

Shapes require to be selected before calling the following SODKDoc arrangement methods:

selectedObjectArrangeForwards()

Arranges a shape forward by one in the stack.

selectedObjectArrangeFront()

Arranges a shape to the front of the stack.

selectedObjectArrangeBackwards()

Arranges a shape backward by one in the stack.

selectedObjectArrangeBack()

Arranges a shape to the back of the stack.

4.7.2 Playing *Presentation* slide shows

Playing a slide show just requires the openSlideShow method to be invoked. Once in slide show mode the following controls should be considered:

- *Tap* or *Swipe left*: Next slide.
- *Swipe right*: Previous slide.
- *Long press*: Exit slide show.

Note: The slide show is an entirely new UIViewController which is presented from the existing SODKBasicDocumentViewController instance.

SODKBasicDocumentViewController methods:

openSlideShow()

Plays the *Presentation* slides.

openSlideShow(*atPage*)

Plays the *Presentation* slides from a specific page.

Parameters

atPage – Int Page number, zero-based.

4.8 Document API - *Spreadsheet*

The methods outlined below should be considered as the specific API for the following document file types:

- **xlsx**
- **xls**
- **xlt**
- **xlsm**
- **xltm**

4.8.1 Cell management

Adding and deleting rows & columns

The number of rows or columns to be added or deleted depends on the number of rows or columns which are currently selected. For example if 3 rows are currently selected and `addRowAbove()` is invoked then 3 rows will be added above the 1st row of the current selection, similarly if 2 columns are selected and `addColumnRight()` is invoked then 2 columns will be added to the right of the last column of the current selection.

`SODKDoc` methods:

`addRowAbove()`
Adds rows above the 1st row of the selected area on the document.

`addRowBelow()`
Adds rows below the last row of the selected area on the document.

`deleteRows()`
Deletes the rows bound by the selected area on the document.

`addColumnLeft()`
Adds columns to the left of the 1st column of the selected area on the document.

`addColumnRight()`
Adds columns to the right of the last column of the selected area on the document.

`deleteColumns()`
Deletes the columns bound by the selected area on the document.

Cell selections

`SODKDoc` properties:

`property selectionIsTablePart`
Gets whether a selection is a cell selection or not.

Type
`Bool` *Read-only*.

`property tableCellsMerged`
Setter and getter for merging cells.

Type
`Bool`.

property cellFormat

Setter and getter for cell format.

Type
`String?` Default value is "General".

Cell metrics

Cell metrics values are always in centimetres.

`SODKDoc` properties:

property columnWidth

Setter and getter for selected column width.

Type
`Float`.

property rowHeight

Setter and getter for selected row height.

Type
`Float`.

Freeze options

Freezing panes allows a *Spreadsheet* to fix rows & columns in place at a designated point.

In order to set the row & cell numbers which the freeze points to on the page a developer should retrieve the `SODKPage` object for the *Spreadsheet* page which they are interested in. Once the `SODKPage` instance is acquired, then a cell address (`SODKTableCellAddress`) should be created and a row and column set on the object. Finally the `topLeftCellOfActivePane` property should be set on the `SODKPage` instance and this will then freeze the spreadsheet page from the supplied `row` and `col` variables.

Swift

Objective-C

```
guard let sodkVC:SODKBasicDocumentViewController = docVC as?␣
↪SODKBasicDocumentViewController else {
  return
}

let page:SODKPage = sodkVC.doc.getSODKPage(0)
var cellAddress = SODKTableCellAddress()
cellAddress.row = 1
cellAddress.col = 1
page.topLeftCellOfActivePane = cellAddress
```

```
SODKBasicDocumentViewController *sodocVC = (SODKBasicDocumentViewController*)docVC;

if (sodocVC) {
  SODKPage *page = [sodocVC.doc getSODKPage:0 update:nil];
  SODKTableCellAddress cellAddress;
  cellAddress.row = 1;
```

(continues on next page)

(continued from previous page)

```
  cellAddress.col = 1;
  page.topLeftCellOfActivePane = cellAddress;
}
```

Note: Setting a `SODKTableCellAddress` object with `0` for `row` and `0` for `col` will effectively unfreeze the *Spreadsheet* page.

4.8.2 Cell formulas

The text value of a cell will define any formula. For example if the value of the cell is `=SUM(E3:E6)` then it will resolve as a basic addition formula for those cells.

4.9 Document API - *PDF*

SmartOffice also supports a few other *PDF* document types:

PDF document types

PDF		**pdf**	
SVG		**svg**	*Read Only*
ePub		**ePub**	*Read Only*
Comic book archive		**cbz**	*Read Only*
Fiction book		**fb2**	*Read Only*

The document types referenced above will be handled by the `MuPDFDKBasicDocumentViewController` & `MuPDFDKDoc` classes.

The properties & methods within these classes should be considered for the API as outlined below.

Note: *SmartOffice* supports loading image files as well as some more obscure document types, however, for these kind of documents the methods outlined below will probably *not* be relevant.

4.9.1 Annotation Modes

A document can enter certain modes which define the user interaction mode with the document. When a mode is set, instead of the tapping to select an item, tapping will respect any current document annotation mode.

`MuPDFDKBasicDocumentViewController` properties & methods:

`property annotatingMode`

Getter and setter for annotation mode.

Type
`MuPDFDKAnnotatingMode`.

Available types:

MuPDFDKAnnotatingMode	Annotation mode
MuPDFDKAnnotatingMode_None	None
MuPDFDKAnnotatingMode_Ink	*Drawing mode* (Ink)
MuPDFDKAnnotatingMode_Line	*Drawing mode* (Line)
MuPDFDKAnnotatingMode_Square	*Drawing mode* (Square)
MuPDFDKAnnotatingMode_Circle	*Drawing mode* (Circle)
MuPDFDKAnnotatingMode_PolyLine	*Drawing mode* (Polyline)
MuPDFDKAnnotatingMode_Polygon	*Drawing mode* (Polygon)
MuPDFDKAnnotatingMode_HighlightTextSelect	*Text mode* (Highlight)
MuPDFDKAnnotatingMode_UnderlineTextSelect	*Text mode* (Underline)
MuPDFDKAnnotatingMode_SquiggleTextSelect	*Text mode* (Squiggle)
MuPDFDKAnnotatingMode_StrikethroughTextSelect	*Text mode* (Strike-through)
MuPDFDKAnnotatingMode_Note	*Placement mode* (Note)
MuPDFDKAnnotatingMode_FreeText	*Placement mode* (Free Text)
MuPDFDKAnnotatingMode_Link	*Placement mode* (Link)
MuPDFDKAnnotatingMode_FileAttachment	*Placement mode* (Attachment)
MuPDFDKAnnotatingMode_RedactionAreaSelect	*Redaction mode* (Area redaction)
MuPDFDKAnnotatingMode_RedactionTextSelect	*Redaction mode* (Text redaction)
MuPDFDKAnnotatingMode_ESignature	*Signature mode* (E-signature)
MuPDFDKAnnotatingMode_DigitalSignature	*Signature mode* (Digital signature)

Drawing mode

This mode deals with the following annotations:

- **Ink**
- **Line**
- **Square**
- **Circle**
- **Polygon**
- **Polyline**

Simply set the mode you require and then *touching the document area* will automatically start the *drawing* annotation.

For an example of how to set a typical drawing mode use the following:

Swift

Objective-C

```
let vc:MuPDFDKBasicDocumentViewController = MuPDFDKBasicDocumentViewController(for:␣
↪session)
vc.annotatingMode = MuPDFDKAnnotatingMode_Ink
```

```
MuPDFDKBasicDocumentViewController *vc = [MuPDFDKBasicDocumentViewController␣
↪viewControllerForSession:session];
vc.annotatingMode = MuPDFDKAnnotatingMode_Ink;
```

Text mode

This mode deals with the following annotations:

- **Highlight**
- **Underline**
- **Strikethrough**
- **Squiggle**

When *text is selected* then this mode will automatically apply the annotation *text* style to the selection.

For an example of how to set a typical text mode use the following:

Swift

Objective-C

```
let vc:MuPDFDKBasicDocumentViewController = MuPDFDKBasicDocumentViewController(for:␣
↪session)
vc.annotatingMode = MuPDFDKAnnotatingMode_HighlightTextSelect
```

```
MuPDFDKBasicDocumentViewController *vc = [MuPDFDKBasicDocumentViewController␣
↪viewControllerForSession:session];
vc.annotatingMode = MuPDFDKAnnotatingMode_HighlightTextSelect;
```

Additionally, without setting a dedicated annotation mode, the following methods are available for text selections:

`MuPDFDKDoc` properties & methods:

addHighlightAnnotationLeaveSelected(*leaveSelected*)

Adds a highlight annotation to the selected text.

Parameters
leaveSelected – `Bool` Boolean switch to leave the text selected or not.

addUnderlineAnnotationLeaveSelected(*leaveSelected*)

Adds an underline annotation to the selected text.

Parameters
leaveSelected – `Bool` Boolean switch to leave the text selected or not.

addStrikethroughAnnotationLeaveSelected(*leaveSelected*)

Adds a strikethrough annotation to the selected text.

Parameters
leaveSelected – `Bool` Boolean switch to leave the text selected or not.

addSquiggleAnnotationLeaveSelected(*leaveSelected*)

Adds a squiggle annotation to the selected text.

Parameters
leaveSelected – `Bool` Boolean switch to leave the text selected or not.

Placement mode

This mode deals with the following annotations:

- **Note**
- **Free Text**
- **Stamp**

- **Link**
- **Attachment**

When *the user taps on the document* then this mode will automatically *place* the annotation at the point of touch.

For an example of how to set a typical placement mode use the following:

Swift

Objective-C

```
let vc:MuPDFDKBasicDocumentViewController = MuPDFDKBasicDocumentViewController(for:␣
↪session)
vc.annotatingMode = MuPDFDKAnnotatingMode_Note
```

```
MuPDFDKBasicDocumentViewController *vc = [MuPDFDKBasicDocumentViewController␣
↪viewControllerForSession:session];
vc.annotatingMode = MuPDFDKAnnotatingMode_Note;
```

4.9.2 Annotation Selections

`MuPDFDKBasicDocumentViewController` properties & methods:

`clearInkAnnotation`()
: Clears any currently drawn ink annotation.

`MuPDFDKDoc` properties & methods:

property `haveAnnotationSelection`
: Gets whether an annotation is selected.

  Type
  : `Bool` *Read-only*.

property `selectionIsAnnotationWithText`
: Gets whether an annotation with text is selected.

  Type
  : `Bool` *Read-only*.

property `selectionIsDraggable`
: Gets whether the annotation is draggable.

  Type
  : `Bool` *Read-only*.

property `selectionIsResizable`
: Gets whether the annotation is resizable.

  Type
  : `Bool` *Read-only*.

property `selectedAnnotationsAuthor`
: Gets the selected annotation author.

  Type
  : `String?` May be `nil` if not applicable. *Read-only*.

property selectedAnnotationsDate

Gets the selected annotation date.

Type
`Date` *Read-only*.

property selectedAnnotationsColor

Setter and getter for the selected annotation color.

Type
`UIColor`.

property selectedAnnotationsText

Setter and getter for the selected annotation text.

Type
`String?` May be `nil` if text is not applicable to the annotation type.

`MuPDFDKDoc` properties & methods:

outlineArea(*area*, *onPage*)

Outlines an area rectangle on a given page.

Parameters

- **area** – `CGRect`.
- **onPage** – `Int` page number, zero-based.

Drawing mode selections

`MuPDFDKDoc` properties & methods:

property selectionIsDrawing

Gets whether a drawing annotation is selected.

A drawing annotation includes **Ink**, **Line**, **Circle**, **Square**, **Polygon** and **Polyline** types.

Type
`Bool` *Read-only*.

property selectionIsInk

Gets whether an ink annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsLine

Gets whether a line annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsCircle

Gets whether a circle annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsSquare`

Gets whether a square annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsPolygon`

Gets whether a polygon annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsPolyLine`

Gets whether a polyline annotation is selected.

Type
`Bool` *Read-only*.

Text mode selections

`MuPDFDKDoc` properties & methods:

`property selectionIsTextHighlight`

Gets whether a text highlight annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsTextUnderline`

Gets whether a text underline annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsTextSquiggle`

Gets whether a text squiggle annotation is selected.

Type
`Bool` *Read-only*.

`property selectionIsTextStrikethrough`

Gets whether a text strikethrough annotation is selected.

Type
`Bool` *Read-only*.

Placement mode selections

`MuPDFDKDoc` properties & methods:

`property selectionIsNote`

Gets whether a note annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsFreeText

Gets whether a free text annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsStamp

Gets whether a stamp annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsLink

Gets whether a link annotation is selected.

Type
`Bool` *Read-only*.

property selectionIsFileAttachment

Gets whether a file attachment annotation is selected.

Type
`Bool` *Read-only*.

`MuPDFDKDoc` properties & methods:

property selectionIsRedaction

Gets whether a redaction is selected.

Type
`Bool` *Read-only*.

property selectionIsWidget

Gets whether a form widget is selected.

Type
`Bool` *Read-only*.

deleteSelectedAnnotation()

Deletes a selected annotation.

4.9.3 Annotation Styling

The API for annotation styling is available via the `MuPDFDKBasicDocumentViewController` instance.

`MuPDFDKBasicDocumentViewController` properties & methods:

For annotations with text:

property annotationFontFace

Getter and setter for the font face to use when creating text.

Type
String.

`property annotationFontSize`

Getter and setter for the font size to use when creating text.

Type
Float.

For Drawing annotation types:

`property annotationOpacity`

Getter and setter for the opacity to use when creating the annotation object.

Type
Float.

`property annotationStrokeColor`

Getter and setter for the stroke/line color of a drawing annotation.

Type
UIColor.

`property annotationStrokeThickness`

Getter and setter for the stroke/line thickness of a drawing annotation.

Type
Float.

`property annotationFillColor`

Getter and setter for the fill color of a drawing annotation.

Type
UIColor.

`property annotationLineHeadStyle`

Getter and setter for the line head style.

Type
ARDKLineEndStyle.

Available types:

```
ARDKLineEndStyle_None      ARDKLineEndStyle_Butt      ARDKLineEndStyle_Slash
ARDKLineEndStyle_Circle   ARDKLineEndStyle_Square   ARDKLineEndStyle_Diamond
ARDKLineEndStyle_OpenArrow                          ARDKLineEndStyle_ROpenArrow
ARDKLineEndStyle_RClosedArrow ARDKLineEndStyle_ClosedArrow
```

`property annotationLineTailStyle`

Getter and setter for the line tail style.

Type
ARDKLineEndStyle.

Available types:

```
ARDKLineEndStyle_None      ARDKLineEndStyle_Butt      ARDKLineEndStyle_Slash
ARDKLineEndStyle_Circle   ARDKLineEndStyle_Square   ARDKLineEndStyle_Diamond
ARDKLineEndStyle_OpenArrow                          ARDKLineEndStyle_ROpenArrow
ARDKLineEndStyle_RClosedArrow ARDKLineEndStyle_ClosedArrow
```

4.9.4 Redactions

Redactions allow users to remove unwanted and/or sensitive information from documents. Essentially redacted areas are covered with black rectangles which obscure areas of the document. These areas will be permanently set on the document once applied.

Redaction mode

This mode deals with the following annotations:

- **Redact Area**
- **Redact Text**

For an example of how to set redaction mode use the following:

Swift

Objective-C

```
let vc:MuPDFDKBasicDocumentViewController = MuPDFDKBasicDocumentViewController(for:␣
↪session)
vc.annotatingMode = MuPDFDKAnnotatingMode_RedactionTextSelect
```

```
MuPDFDKBasicDocumentViewController *vc = [MuPDFDKBasicDocumentViewController␣
↪viewControllerForSession:session];
vc.annotatingMode = MuPDFDKAnnotatingMode_RedactionTextSelect;
```

`MuPDFDKDoc` properties & methods:

`addRedactAnnotation()`

Adds a redaction based on currently selected text or area. Note, this does not *finalize* the redaction, rather it marks the text or area for redaction. This will apear as a red outlined area on the document.

`hasRedactions()`

Returns whether the document contains redactions or not.

Return Bool*Bool*

`finalizeRedactAnnotations{code}`

Applies all marked and *not yet applied* redactions. This operation is not reversible.

Once applied any marked text and/or areas will turn solid black.

Parameters

`code` – {} Any bespoke code which will be triggered on this action.

Note: To delete a marked, *but not yet applied*, redaction use the `deleteSelectedAnnotation()` method. Finalized redactions can not be edited or deleted.

4.9.5 Secure Save

Secure Save is used to save a PDF document ensuring that any redacted information will be impossible to extract. Essentially the document is rasterized, so that all text data information is flattened into a bitmap and is thus impossible to be copied out again. Any previously redacted areas will therefore contain no text information.

`MuPDFDKDoc` properties & methods:

`exportHighSecurityRedaction`(*to*, *resolution*, *ocrLanguage*, *completion*)

Saves the document securely with a completion method.

Parameters

- **`to`** – `String` The file path to save to.
- **`resolution`** – `Int` The image resolution in DPI for the pages
- **`ocrLanguage`** – `String` The language to use to detect text in the document.
- **`completion`** – `(ARDKSaveResult) -> Void` The file path to save to.

`exportHighSecurityRedaction(to, resolution, ocrLanguage) async`

An asynchronous method for saving the document securely.

Parameters

- **`to`** – `String` The file path to save to.
- **`resolution`** – `Int` The image resolution in DPI for the pages
- **`ocrLanguage`** – `String` The language to use to detect text in the document.

4.9.6 Signatures

Signature mode

This mode deals with the following annotations:

- **E-signatures**
- **Digital signatures**

For an example of how to set signature mode use the following:

Swift

Objective-C

```
let vc:MuPDFDKBasicDocumentViewController = MuPDFDKBasicDocumentViewController(for:␣
↪session)
vc.annotatingMode = MuPDFDKAnnotatingMode_ESignature
```

```
MuPDFDKBasicDocumentViewController *vc = [MuPDFDKBasicDocumentViewController␣
↪viewControllerForSession:session];
vc.annotatingMode = MuPDFDKAnnotatingMode_ESignature;
```

`MuPDFDKBasicDocumentViewController` properties & methods:

property eSigImage

Setter and getter for the image to be used as an e-signature.

Type
`UIImage`.

`MuPDFDKDoc` properties & methods:

findSigStart(*atPage*, *andAnnot*, *in*, *cancelFlag*, *onEvent*)

Searches the document for unsigned digital signature fields with a callback.

Parameters

- **atPage** – `Int` The page number to search, zero-based.
- **andAnnot** – `Int` The index number for tracking any found annotation.
- **in** – `ARDKSearchDirection` The direction for searching the document in (`ARDKSearch_Forwards` or `ARDKSearch_Backwards`).
- **cancelFlag** – `ObjCBool` The objective-C cancel variable.
- **onEvent** – `(ARDKSearchEvent, Int, Int, CGRect) -> Void` Event callback.

Sample code for searching a document for unsigned digital signature fields as follows:

Swift

Objective-C

```
var cancelled:ObjCBool = false
var sigSearchPage:Int = 0
var sigSearchAnnot:Int = -1
doc.findSigStart(atPage: sigSearchPage,
                 andAnnot: sigSearchAnnot,
                 in: ARDKSearch_Forwards,
                 cancelFlag: &cancelled,
                 onEvent: { [weak self] event, page, annot, rect in
    switch event {

    case ARDKSearch_Progress:

        break

    case ARDKSearch_Found:
        self?.sigSearchPage = page
        self?.sigSearchAnnot = annot
        self?.doc.outlineArea(rect, onPage: page)
        self?.documentViewController.showArea(rect, onPage: page)

        break

    case ARDKSearch_Cancelled:
    default:
        ()
    }
})
```

```
BOOL cancelled;
NSInteger sigSearchPage = 0;
NSInteger sigSearchAnnot = -1;
__weak typeof(self) weakSelf = self;

[_doc findSigStartAtPage:sigSearchPage
             andAnnot:sigSearchAnnot
          inDirection:ARDKSearch_Forwards
           cancelFlag:&cancelled
              onEvent:^(ARDKSearchEvent event, NSInteger page, NSInteger index, CGRect␣
↪rect) {
    switch (event)
    {
        case ARDKSearch_Progress:

            break;

        case ARDKSearch_Found:
            weakSelf.sigSearchPage = page;
            weakSelf.sigSearchAnnot = index;
            [weakSelf.doc outlineArea:rect onPage:page];
            [weakSelf.documentViewController showArea:rect onPage:page];

            break;

        case ARDKSearch_Cancelled:
        default:

            break;
    }
}];
```

4.9.7 Table of Contents

Also known as the "document outline" a table of contents may be available for the PDF document.

`MuPDFDKDoc` properties & methods:

`property toc`

Gets an array of `ARDKTocEntry` items or `nil` if there is no table of contents.

Type
`[ARDKTocEntry]?` *Read-only*.

`ARDKTocEntry` objects contain the following:

Property	Type
`label`	`String`
`depth`	`UInt`
`open`	`Bool`
`children`	`NSMutableArray?`

4.9.8 Links

Links in documents are handled by an in-built UI which will allow the user to edit or delete them as required.

CHAPTER

FIVE

MACOS SDK

5.1 Getting Started

5.1.1 System requirements

- Xcode minimum SDK: 11.3
- Minimum macOS version: 10.11

5.1.2 Adding the SDK to your project

You should have been provided with a `sodk.framework` file for your project. This file is effectively the SmartOffice Developer Kit and contains all the code required for your projects.

The linked framework should be added to your project file system and then referenced in the Frameworks section in Xcode.

See the Xcode screenshot below for an example:

5.1.3 Verify your integration

To ensure you have correctly added the framework to your project ensure that your source code files can reference `import sodk` without error.

5.1.4 Your application UI

If you have started a standard macOS application then you will likely have a default Storyboard with UI and application hooks for opening and saving files as shown below:

Assuming that this is the type of UI for your application then your application's first responder (in your delegate or view controller classes) can easily pick up user actions as follows:

Swift

Objective-C

```
@IBAction func newDocument(_ btn:NSButton) {

}

@IBAction func openDocument(_ btn:NSButton) {
```

(continues on next page)

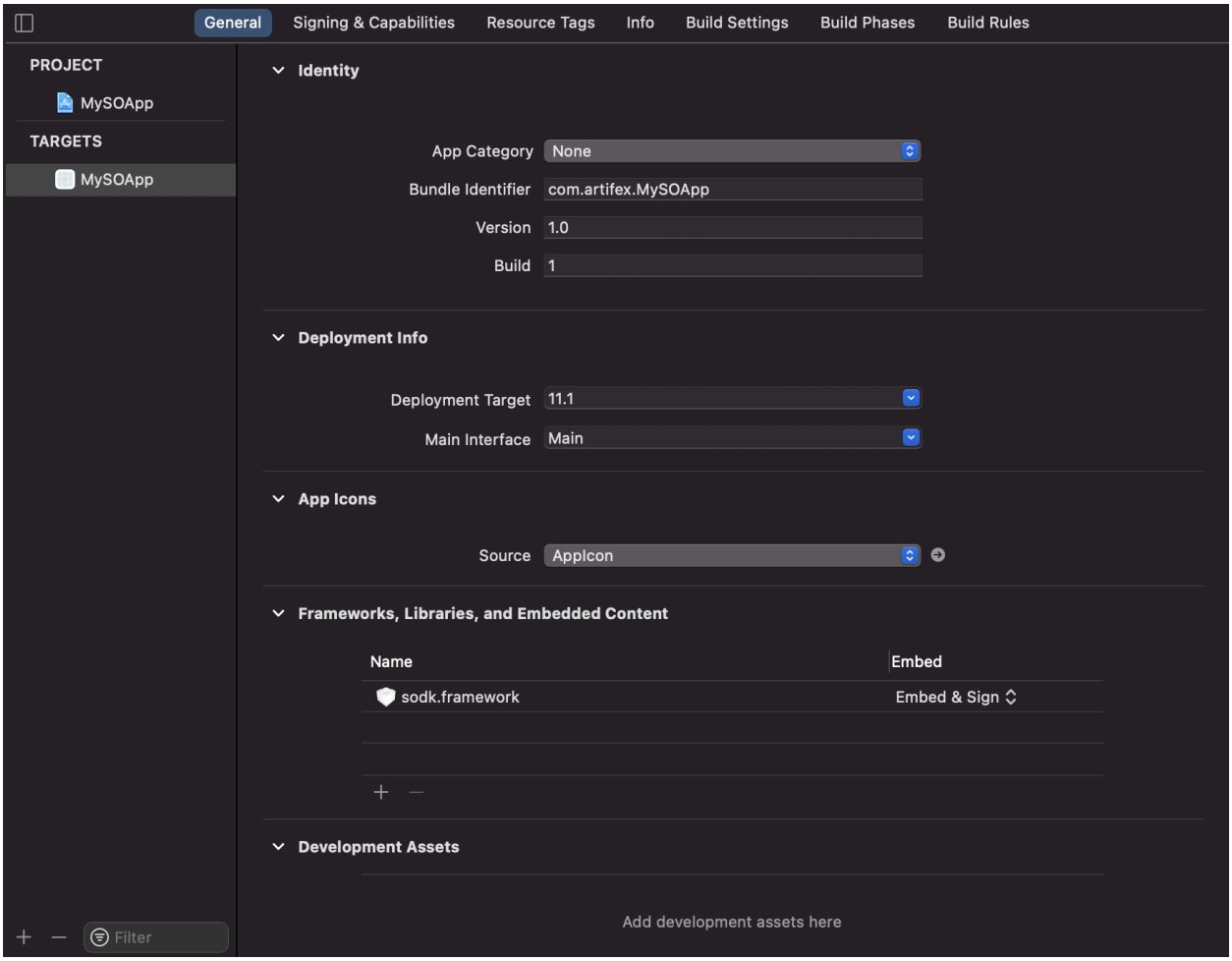

Fig. 1: Embedding the `sodk.framework` framework in Xcode

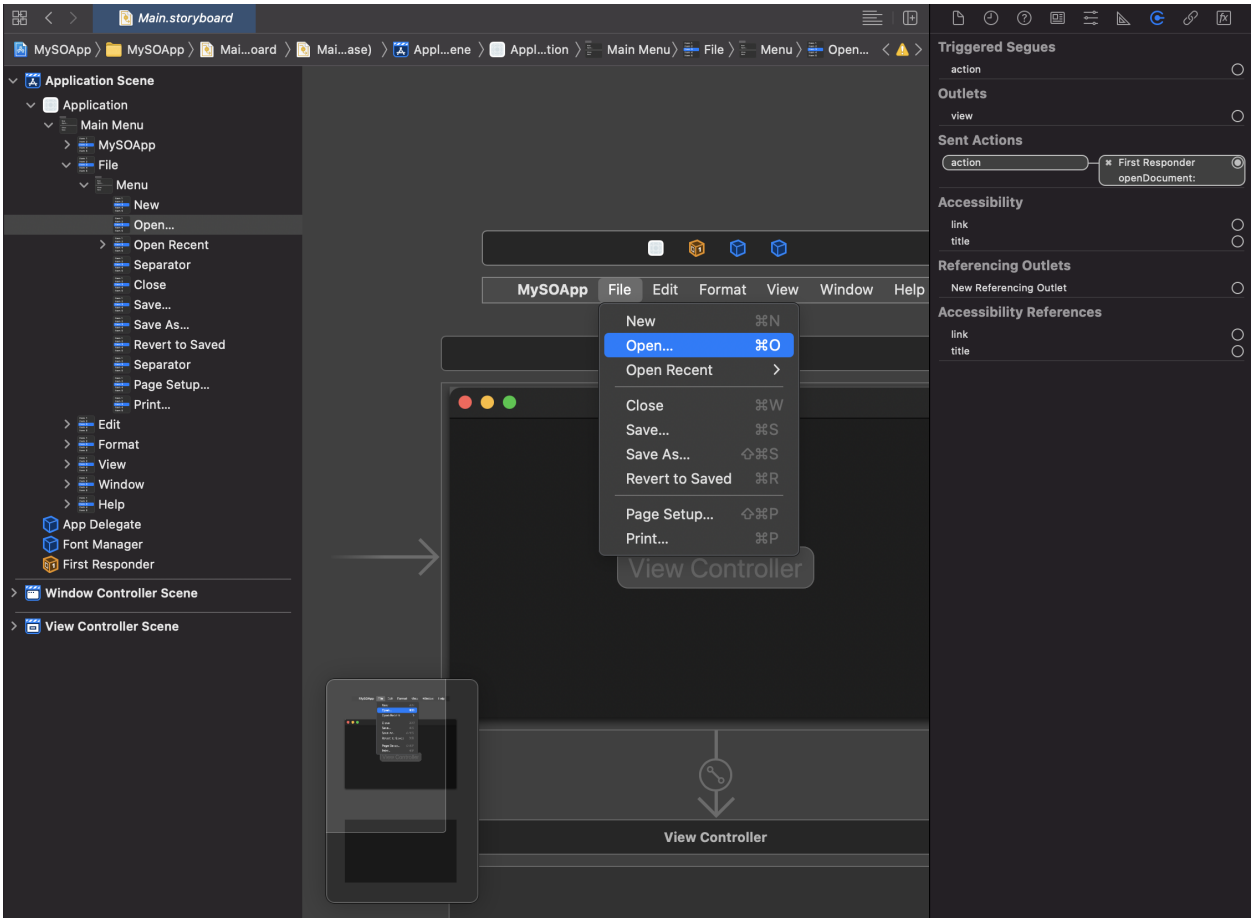

Fig. 2: Default macOS Storyboard in Xcode

(continued from previous page)

```
}

@IBAction func save(_ btn:NSButton) {

}

@IBAction func saveAs(_ btn:NSButton) {

}
```

```
- (IBAction)newDocument:(id)sender {

}

- (IBAction)openDocument:(id)sender {

}

- (IBAction)saveDocument:(id)sender {

}

- (IBAction)saveDocumentAs:(id)sender {

}
```

5.1.5 App Capabilities

Ensure to remove "App Sandbox" from under Xcode's "Signing & Capabilities". If included this prevents the "File Save" operation from successfully completing.

Note: Many Xcode applications will have included this capability by default.

5.1.6 Opting out of Dark Mode

For visual UI display the SmartOffice framework only supports *Light Mode* at present, therefore your application should opt out of Dark Mode by amending the application's plist as follows:

```
<key>NSRequiresAquaSystemAppearance</key>
<string>true</string>
```

5.2 Document based apps

Document based apps require to have the following components defined before being able to manage external files:

- A custom subclass of `NSDocument`.
- Declaration of document type.

Note: It is highly recommended to read Developing a document based app from Apple for further details and understanding.

5.2.1 Subclassing NSDocument

You should create a custom subclass to provide most of the behavior for managing your document. This custom subclass is not the same as the file but is an object in memory that owns and manages the document data.

Once created in Xcode your subclass stub should look something like the following:

Swift

Objective-C

```
import Cocoa

class Document: NSDocument {

    /*
    override var windowNibName: String? {
        // Override returning the nib file name of the document
        // If you need to use a subclass of NSWindowController or if your document␣
↪supports multiple NSWindowControllers, you should remove this method and␣
↪override -makeWindowControllers instead.
        return "Document"
    }
    */

    override func windowControllerDidLoadNib(_ aController: NSWindowController) {
        super.windowControllerDidLoadNib(aController)
        // Add any code here that needs to be executed once the windowController␣
↪has loaded the document's window.
    }

    override func data(ofType typeName: String) throws -> Data {
        // Insert code here to write your document to data of the specified type,␣
↪throwing an error in case of failure.
        // Alternatively, you could remove this method and override␣
↪fileWrapper(ofType:), write(to:ofType:), or␣
↪write(to:ofType:for:originalContentsURL:) instead.
        throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
    }

    override func read(from data: Data, ofType typeName: String) throws {
        // Insert code here to read your document from the given data of the␣
```

(continues on next page)

(continued from previous page)

```
↪specified type, throwing an error in case of failure.
        // Alternatively, you could remove this method and override␣
↪read(from:ofType:) instead.  If you do, you should also override␣
↪isEntireFileLoaded to return false if the contents are lazily loaded.

        throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
    }

    override class var autosavesInPlace: Bool {
        return true
    }

}
```

```
#import "Document.h"

@implementation Document

    /*
    - (NSString *)windowNibName {
        // Override to return the nib file name of the document.
        // If you need to use a subclass of NSWindowController or if your document␣
↪supports multiple NSWindowControllers, you should remove this method and override -
↪makeWindowControllers instead.
        return <#nibName#>;
    }
    */

    - (void)windowControllerDidLoadNib:(NSWindowController *)aController {
        [super windowControllerDidLoadNib:aController];
        // Add any code here that needs to be executed once the windowController has␣
↪loaded the document's window.
    }

    - (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError {
        // Insert code here to write your document to data of the specified type. If␣
↪outError != NULL, ensure that you create and set an appropriate error if you return␣
↪nil.
        // Alternatively, you could remove this method and override -
↪fileWrapperOfType:error:, -writeToURL:ofType:error:, or -
↪writeToURL:ofType:forSaveOperation:originalContentsURL:error: instead.
        if (outError) {
            *outError = [NSError errorWithDomain:NSOSStatusErrorDomain code:unimpErr␣
↪userInfo:nil];
        }
        return nil;
    }

    - (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError␣
↪**)outError {
        // Insert code here to read your document from the given data of the specified␣
↪type. If outError != NULL, ensure that you create and set an appropriate error if you␣
```

(continues on next page)

(continued from previous page)

```
↪return NO.
        // Alternatively, you could remove this method and override -
↪readFromFileWrapper:ofType:error: or -readFromURL:ofType:error: instead.
        // If you do, you should also override -isEntireFileLoaded to return NO if the␣
↪contents are lazily loaded.
        if (outError) {
            *outError = [NSError errorWithDomain:NSOSStatusErrorDomain code:unimpErr␣
↪userInfo:nil];
        }
        return NO;
    }

    + (BOOL)autosavesInPlace {
        return YES;
    }

@end
```

Note: This *custom subclass of NSDocument* - is the main class responsible for *file operations* and most of your SmartOffice application logic will need to be included in there.

5.2.2 Declaring document type

Your app declares its document type as a uniform type identifier (UTI), which is a name that identifies a data format. You should provide user-defined UTIs, which are custom identifiers for your custom data formats.

In the case of a SmartOffice app we will need to provide support for an exhaustive set of document types: `doc`, `docx`, `docm`, `dotx`, `dotm`, `xls`, `xlsx`, `xlsm`, `xltx`, `xltm`, `ppt`, `pps`, `pptx`, `pptm`, `potx`, `potm`, `ppsx`, `ppsm`, `pdf`, `txt`, `csv`, `bmp`, `jpg`, `jpeg`, `gif`, `png`, `tif`, `tiff`, `wmf`, `emf`, `hwp`.

To enable our macOS app to be able to recognize these file types, and be able to open them, we need to update our `plist` with the file type information.

It is easiest to open your application's `plist` as source code (as opposed to Xcode's property list) and then add the information for the key `CFBundleDocumentTypes`.

```
<key>CFBundleDocumentTypes</key>
 <array>
     <dict>
         <key>CFBundleTypeExtensions</key>
         <array>
             <string>doc</string>
             <string>docx</string>
             <string>docm</string>
             <string>dotx</string>
             <string>dotm</string>
             <string>xls</string>
             <string>xlsx</string>
             <string>xlsm</string>
             <string>xltx</string>
             <string>xltm</string>
```

(continues on next page)

(continued from previous page)

```
            <string>ppt</string>
            <string>pps</string>
            <string>pptx</string>
            <string>pptm</string>
            <string>potx</string>
            <string>potm</string>
            <string>ppsx</string>
            <string>ppsm</string>
            <string>pdf</string>
            <string>txt</string>
            <string>csv</string>
            <string>bmp</string>
            <string>jpg</string>
            <string>jpeg</string>
            <string>gif</string>
            <string>png</string>
            <string>tif</string>
            <string>tiff</string>
            <string>wmf</string>
            <string>emf</string>
            <string>hwp</string>
        </array>
        <key>CFBundleTypeIconFile</key>
        <string></string>
        <key>CFBundleTypeName</key>
        <string>DocumentType</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>????</string>
        </array>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>NSDocumentClass</key>
        <string>$(PRODUCT_MODULE_NAME).Document</string>
    </dict>
</array>
```

Note: At the end of the the `plist` under the key `NSDocumentClass` we are referencing our *custom subclass of NSDocument* - this is critical for your application to correctly connect your custom class (in this case `Document` as in the prior subclassing example) against the associated file types.

5.2.3 Opening a File

Once you have your custom `NSDocument` class ready and your `plist` updated to recognize required document types your application should be capable of being able to open a file.

As outlined previously you should have an `IBAction` hooked up in your Storyboard UI to your view controller or application delegate as follows:

Swift

Objective-C

```
@IBAction func openDocument(_ btn:NSButton) {

}
```

```
- (IBAction)openDocument:(id)sender {

}
```

When the user presses invokes this action via a button then our application should invoke a modal window to request a file chooser with an instance of NSOpenPanel as follows:

Swift

Objective-C

```
let panel:NSOpenPanel = NSOpenPanel()
panel.allowsMultipleSelection = false
panel.canChooseDirectories = false
panel.isFloatingPanel = true

let result = panel.runModal()

if result == .OK {
    let documentURL:URL? = panel.url
    let docController:NSDocumentController = NSDocumentController.shared
    var doc:NSDocument? = docController.currentDocument

    if doc == nil {

        do {
            doc = try docController.openUntitledDocumentAndDisplay(true)
            doc?.fileURL = documentURL

            guard let document:Document = doc as? Document else {
                print("cannot cast NSDocument to Document")
                return
            }

            document.loadDocument()

        } catch let error {
            print("error:\(error.localizedDescription)")
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
NSOpenPanel *panel = [NSOpenPanel openPanel];
[panel setAllowsMultipleSelection: NO];
[panel setCanChooseDirectories: NO];
[panel setFloatingPanel: YES];

NSInteger result = [paenl runModal];

if (result == NSModalResponseOK) {

    NSURL *url = [panel URL];

    if (url != nil && url.isFileURL) {

        NSDocumentController *dc;
        Document             *doc;

        dc  = [NSDocumentController sharedDocumentController];
        doc = dc.currentDocument;

        if (doc == nil) {
            NSError *err = nil;

            doc = [dc openUntitledDocumentAndDisplay:YES error:&err];

            if (err == nil) {
                doc.fileURL = url
                [doc loadDocument];
            }
            else {
                NSLog(@"Failed to create a document: %@", err.description);
            }
        }
    }
}
```

The above code fetches the user-selected document as a file URL and associates this URL with our bespoke `NSDocument` class (in this case `Document`), our bespoke class then has a *loadDocument* method which should handle the file URL.

Note: The *loadDocument* method exemplified above is the responsibility of the developer and can obviously be renamed as required.

5.3 File Operations

In order to create, open and save files an application developer needs to work with the *document session instance*, `SODKDocSession`, the session requires a single instance of the *SmartOffice library* `SODKLib`, alongside a *custom file state class* conforming to the `SODKFileState` protocol and any *document settings* defined with `SODKDocumentSettings`.

Once these are in place a session can be instantiated and passed into an instance of a *SmartOffice document view controller* for your application window.

5.3.1 Your NSDocument class

Assuming that documents are being supplied from a bespoke instance of `NSDocument` then we might want to have a method ready within the class to handle this, as in the following example:

Swift

Objective-C

```
func loadDocument() {
    /// 1: Ensure there is an instance of the SmartOffice library Singleton in␣
↪memory
    /// 2: Ensure there is an instance of a custom file state class with it's file␣
↪path variables correctly set
    /// 3: Ensure there is an instance of document settings
    /// 4: Start the document session
    /// 5: Create a SmartOffice document view controller for the session in a␣
↪window
}
```

```
- (void)loadDocument {
    /// 1: Ensure there is an instance of the SmartOffice library Singleton in memory
    /// 2: Ensure there is an instance of a custom file state class with it's file path␣
↪variables correctly set
    /// 3: Ensure there is an instance of document settings
    /// 4: Start the document session
    /// 5: Create a SmartOffice document view controller for the session in a window
}
```

Note: The following steps in the above code stub are outlined below.

Instantiating the SmartOffice library

The SmartOffice library, `SODKLib`, should be instantiated as a "Singleton" class usually this would be done in your application delegate as a static class variable as follows:

Swift

Objective-C

```
class AppDelegate: NSObject, NSApplicationDelegate {

   /// Return the singleton library object with minimum settings
   static let SODK_LIB: SODKLib? = {
        let settings:SODKSettings = SODKSettings()
        settings.temporaryPath = "\(NSSearchPathForDirectoriesInDomains(.
↪applicationSupportDirectory,.userDomainMask, true)[0])/tmp"
        let sodklib = SODKLib(settings:settings)
        return sodklib
   }()

...
```

```
@implementation AppDelegate

static SODKLib *SODK_LIB = nil;

/// Return the singleton library object with minimum settings
+ (SODKLib *)getSODKLibrary
{
      dispatch_once(&onceToken, ^{
         SODKSettings *settings = [[SODKSettings alloc] init];
         NSString *tempPath =␣
↪NSSearchPathForDirectoriesInDomains(NSApplicationSupportDirectory, NSUserDomainMask,␣
↪YES)[0];
         settings.temporaryPath = [tempPath stringByAppendingPathComponent:@"/tmp"];
         SODK_LIB = [[SODKLib alloc] initWithSettings:settings];
      });

      return SODK_LIB;
}
```

...

Note: Passing through the setting for the `temporaryPath` in the `SODKSettings` instance is a minimum requirement for temporary file read & write access. Unless you are using `SecureFS`, this temporary location should be a folder located in the application's support directory.

Custom file state class

An application developer should create their own custom class conforming to the `SODKFileState` protocol. This file state class contains logic for file path, document type, read-only state as well as session callback methods for session events. The full protocol is as follows:

```
@protocol ARDKFileState <NSObject>

/// The path to the document to display and edit
@property(readonly) NSString *absoluteInternalPath;

/// The file type
```

(continues on next page)

(continued from previous page)

```
@property(readonly) ARDKDocType docType;

/// The path which will be displayed within the UI to
/// denote the file being edited. This may be different
/// to absoluteInternalPath for two reasons.
/// The app may be supplying the path to a copy of the
/// file in absoluteInternalPath, and wish to give a
/// displayPath that better represents the location of
/// the original file. Secondly, displayPath may use
/// a more readable start of path (e.g., "Storage/")
/// in place of the true location of the file
@property(readonly) NSString *displayPath;

/// Whether this file can be overwritten. If YES, saving
/// back of edits to the file are not permitted, and
/// the user will need to save to another location. An
/// app might return YES here for document templates.
@property(readonly) BOOL isReadonly;

/// In some use cases, an app may supply a copy of the file
/// to be edited, which will require copying back after any
/// edits have been saved. This property keeps track of
/// whether the original file is out of date with the
/// supplied one, and hence whether copying back may be needed.
/// For apps that supply the original file directly, this
/// property can simply return NO.
@property(readonly) BOOL requiresCopyBack;

/// Information regarding the viewing state of the file (e.g.,
/// which page is being viewed). The document view will ensure
/// this is valid when sessionDidClose is called. A class
/// implementing the ARDKFileState interface can record this
/// value (using its NSCoding interface) against the file name
/// and then arrange to restore it should the same file be
/// reopened.
@property(retain) NSObject<NSCoding> *viewingStateInfo;

/// Information method called when a session has loaded the
/// first page of the document
- (void)sessionDidLoadFirstPage:(ARDKDocSession *)session;

/// Information method called when the file is opened in the
/// main document view ready for viewing and editing by the user.
- (void)sessionDidShowDoc:(ARDKDocSession *)session;

/// Information method called when a session saves document
/// edits back to the supplied file
- (void)sessionDidSaveDoc:(ARDKDocSession *)session;

/// In some use cases, an app may supply a copy of the file
/// to be edited, which will require copying back after any
/// edits have been saved. This method will be called when
```

(continues on next page)

(continued from previous page)

```
/// copying back may be necessary. For apps that supply the
/// original file directly, and return NO from requiresCopyBack
/// this method need do nothing.
- (void)sessionRequestedCopyBackOnCompletion:(void (^)(BOOL succeeded))block;

/// Information method called when a session ends. In the case
/// that an app supplies a copy of a file to be edited. This
/// method might delete the copy, since the session is no
/// longer using it. The file should NOT be copied back before
/// removal.
- (void)sessionDidClose;
```

An application developer should adhere to the above protocol in their own file states class (e.g. "MyFileState") and then instantiate this class, at a minimum, ensuring to correctly set the variables `absoluteInternalPath`, `displayPath` and `docType`.

Swift

Objective-C

```
let fileState:MyFileState = MyFileState()
```

```
MyFileState *fileState = [[MyFileState alloc] init];
```

Document settings

Document settings should be defined in an instance of `SODKDocumentSettings`, the available settings are as follows:

Key	Value type	Description
enableAll	`bool`	Use this to enable or disable all settings.
saveButtonEnabled	`bool`	Whether the save button appears on the document File ribbon.
editingEnabled	`bool`	Whether document editing is enabled.
printingEnabled	`bool`	Whether the user is allowed to print documents.
trackChangesFeatureEnabled	`bool`	Whether track changes for Word documents is enabled.
insertFromCameraEnabled	`bool`	Whether the user is allowed to insert photos from the camera into documents.
insertFromGalleryEnabled	`bool`	Whether the user is allowed to insert images from the gallery into documents.

Sample code:

Swift

Objective-C

```
let settings:SODKDocumentSettings = SODKDocumentSettings()
settings.enableAll(true)
```

```
Settings *settings = [[SODKDocumentSettings alloc] init];
[settings enableAll:true];
```

Document session

Once you have:

- *created your SmartOffice library singleton*.
- *instantiated your custom file* class and set its required variables.
- *instantiated your document settings*.

then you are ready to start a document session as follows:

Swift

Objective-C

```
let session:SODKDocSession = SODKDocSession(fileState:fileState,
                                            ardkLib:SODK_LIB,
                                        docSettings:settings)
```

```
SODKDocSession *session = [SODKDocSession sessionForFileState:fileState
                                                      ardkLib:SODK_LIB
                                                  docSettings:settings];
```

The document session should then be passed into a SmartOffice document view controller which is associated with the current window.

Create a SmartOffice document view controller

An instance of `ARDKDocumentViewController` should be created via the `SODKDocumentViewController` constructor. This requires the document session, start page number and window passed into it. There is also a completion block for additional handler settings as required.

Constructor:

Swift

Objective-C

```
class func viewController(for session: SODKDocSession?,
             openOnPage page: Int32,
                  for window: NSWindow?,
               settingsBlock: ((ARDKDocumentViewController?) -> Void)? = nil) ->␣
↪ARDKDocumentViewController?
```

```
+ (ARDKDocumentViewController * _Nullable)
      viewControllerForSession:(SODKDocSession * _Nullable )session
                   openOnPage:(int)page
                    forWindow:(NSWindow * _Nullable )window
                 settingsBlock:(void (^ _Nullable )(ARDKDocumentViewController * _
↪Nullable))settingsBlock;
```

Sample code:

Swift

Objective-C

```
// Find the window for the view controller
guard let window:NSWindow = self.windowForSheet else {
    Swift.print("No window for sheet")
    return
}

// Create the view controller with the document session
let ardkDocVC = SODKDocumentViewController.viewController(for:session,
                                                          openOnPage:0,
                                                                 for:window,
                                                       settingsBlock:
↪{(docVC:ARDKDocumentViewController?) in

                func saveResultHandler(destDocPath:String?, saveResult:SODKSaveResult) {
                    if saveResult == ARDKSave_Succeeded {

                    } else if saveResult == ARDKSave_Cancelled {

                    } else if saveResult == ARDKSave_Error {

                    }
                }

                // This handler will be called when the document save action has␣
↪been completed.
                docVC?.saveResultHandler = saveResultHandler
            }
        )
```

```
// Find the window for the view controller
NSWindow *window = self.windowForSheet;

if (window == nil) {
    NSLog(@"No window for sheet");
    return;
}

// Create the view controller with the document session
ARDKDocumentViewController *ardkDocVC = [SODKDocumentViewController
                                      viewControllerForSession:session
                                                    openOnPage:0
                                                     forWindow:window
                                                 settingsBlock:^
↪(ARDKDocumentViewController *docVC) {

            // This handler will be called when the document save action has been␣
↪completed.
            docVC.saveResultHandler = ^(NSString *destDocPath, SODKSaveResult␣
```

(continues on next page)

(continued from previous page)

```
↪saveResult) {
                if (saveResult == ARDKSave_Succeeded) {

                } else if (saveResult == ARDKSave_Cancelled) {

                } else if (saveResult == ARDKSave_Error) {

                }
            };
        }
    ];
```

At this point you should then see the SmartOffice UI with your chosen document displayed!

Example:

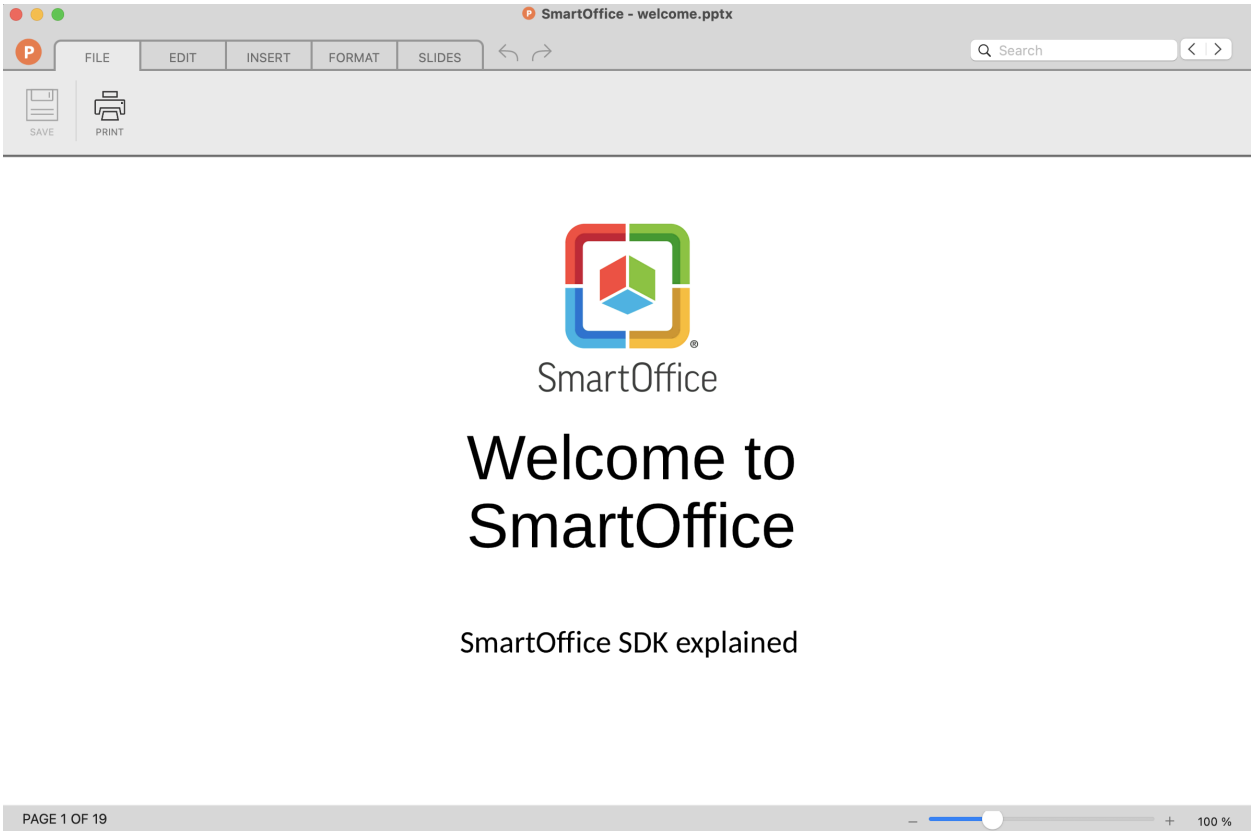

Fig. 3: SmartOffice UI

SmartOffice document view controller handlers:

SmartOffice document view controller handlers

In order to handle certain user events, and for associated button icons to appear in the SmartOffice UI, handlers should be defined at the point of the view controller's instantiation. Without defining associated handler methods certain SmartOffice functionality will be unavailable.

Additionally other handler types can be defined to listen to other user events, for example to handle the event of the user clicking on a URL within a document.

Handlers are `nil` by default - it is the responsibility of the application developer to create them as they require once the instance of `ARDKDocumentViewController` is ready.

Handler	Handler type	Description	UI
saveResultHandler	`ARDKSaveResultHandler`	Called when a 'save' action has been completed.	n/a
saveAsHandler	`ARDKSaveAsHandler`	Called when a 'save as' process is required.	If the handler is nil, the 'Save As' button will not be present in the UI.
savePdfHandler	`ARDKSavePdfHandler`	Called when the 'save pdf' button is pressed.	If the handler is nil, the 'Save PDF' button will not be present in the UI.
shareHandler	`ARDKButtonHandler`	Called when the 'Share' button is pressed.	If the handler is nil, the 'Share' button will not be present in the UI.
openPdfInHandler	`ARDKButtonHandler`	Called when 'Open PDF In' button is pressed.	If the handler is nil, the 'Open PDF In' button will not be present in the UI.
openUrlHandler	`ARDKOpenURLHandler`	Called when user taps an external link in a document.	If the handler is nil, nothing will happen when document links are tapped.
openInHandler	`ARDKButtonHandler`	n/a	n/a
insertFromCameraHandler	`ARDKInsertFromCameraHandler`	Called when 'Insert From Camera' process is initiated.	If the handler is nil, the 'Insert Photo' button will not be present in the UI.
insertFromGalleryHandler	`ARDKInsertFromGalleryHandler`	Called when 'Insert From Photos/Gallery' process is initiated.	If the handler is nil, the 'Insert Image' button will not be present in the UI.

Save result

Add this handler to listen to document "Save" events.

```
typedef void (^ARDKSaveResultHandler)(NSString * _Nullable destDocPath,
                                      ARDKSaveResult       saveResult);
```

Swift

Objective-C

```
ardkDocVC.saveResultHandler = {(destDocPath:String?, saveResult:SODKSaveResult) in

                    }
```

```
ardkDocVC.saveResultHandler = ^(NSString *destDocPath, SODKSaveResult saveResult) {

                       };
```

Note: This handler only listens to standard document "Save" events, it does not capture all types of saves for e.g. "Save As" or "Save PDF".

Save As

Add this handler to add the "Save As" button to the UI and to listen to "Save As" events.

```
typedef void (^ARDKSaveAsHandler)(NSViewController * _Nullable presentingVc,
                                  ARDKDocSession   * _Nullable session,
                                  NSString         * _Nullable currentFilename);
```

Swift

Objective-C

```
ardkDocVC.saveAsHandler = {(presentingVc:NSViewController?,
                        session:ARDKDocSession?,
                currentFilename:String?) in

                 /*
                 Open an `NSSavePanel`,
                 listen to the user input result,
                 use the `save` method on the session,
                 update your file state as required
                 */

                 }
```

```
ardkDocVC.saveAsHandler = ^(NSViewController *presentingVc,
                        ARDKDocSession *session,
                        NSString *currentFilename) {

                   /*
                   Open an `NSSavePanel`,
                   listen to the user input result,
                   use the `save` method on the session,
                   update your file state as required
                   */

                   }
```

Save PDF

Add this handler to add the "Save PDF" button to the UI and to listen to "Save PDF" events.

```
typedef void (^ARDKSavePdfHandler)(NSViewController * _Nullable presentingVc,
                                   ARDKDocSession   * _Nullable session,
                                   NSString         * _Nullable currentFilename);
```

Swift

Objective-C

```
ardkDocVC.savePdfHandler = {(presentingVc:NSViewController?,
                             session:ARDKDocSession?,
                    currentFilename:String?) in

                      /*
                      Open an `NSSavePanel`,
                      set the filename to default with .pdf extension,
                      listen to the user input result,
                      use the `exportPDF` method on the session document
                      */

                     }
```

```
ardkDocVC.savePdfHandler = ^(NSViewController *presentingVc,
                             ARDKDocSession *session,
                             NSString *currentFilename) {

                        /*
                        Open an `NSSavePanel`,
                        set the filename to default with .pdf extension,
                        listen to the user input result,
                        use the `exportPDF` method on the session document
                        */

                       }
```

Note: Given a valid pdf filename, then this handler will require using the `exportPDF` API on the document session's `SODKDoc` document.

Share

Add this handler to add the "Share" button to the UI and to listen to "Share" events.

```
typedef void (^ARDKButtonHandler)(ARDKHandlerInfo * _Nullable hInfo,
                                  void (^ _Nullable completion)(void));
```

Swift

Objective-C

```
ardkDocVC.shareHandler = {(hInfo:ARDKHandlerInfo?, completion:(() -> Void)?) in

                }
```

```
ardkDocVC.shareHandler = ^(ARDKHandlerInfo *hInfo, void (^completion)(void)) {

                 }
```

Open PDF In

Add this handler to add the "Open PDF In" button to the UI and to listen to "Open PDF In" events.

```
typedef void (^ARDKButtonHandler)(ARDKHandlerInfo * _Nullable hInfo,
                                  void (^ _Nullable completion)(void));
```

Swift

Objective-C

```
ardkDocVC.openPdfInHandler = {(hInfo:ARDKHandlerInfo?, completion:(() -> Void)?) in

                    }
```

```
ardkDocVC.openPdfInHandler = ^(ARDKHandlerInfo *hInfo, void (^completion)(void)) {

                     }
```

Open URL

Add this handler to listen to "Open URL" events. The application could pass the url to the OS, or an internal browser, depending on the security requirements of the app.

```
typedef void (^ARDKOpenURLHandler)(NSViewController * _Nullable presentingVc,
                                   NSURL            * _Nullable url);
```

Swift

Objective-C

```
ardkDocVC.openUrlHandler = {(presentingVc:NSViewController?, url:URL?) in

                  }
```

```
ardkDocVC.openUrlHandler = ^(NSViewController *presentingVc, NSURL *url) {

                  }
```

Insert camera image

Add this handler to add the "Insert photo" button to the UI and to listen to "Insert photo" events. For more on accessing the camera on macOS please refer to the guide cameras and media capture.

```
typedef void (^ARDKInsertFromCameraHandler)(NSViewController * _Nullable presentingVc,
                                            ARDKInsertImageResultHandler _Nullable completion);
```

Swift

Objective-C

```
ardkDocVC.insertFromCameraHandler = {(presentingVc:NSViewController?,
                                      completion:ARDKInsertImageResultHandler?) in

                                    }
```

```
ardkDocVC.insertFromCameraHandler = ^(NSViewController *presentingVc,
                                      ARDKInsertImageResultHandler  completion) {

                                      }
```

Note: Only applicable in the context of Word and Powerpoint documents.

Insert gallery image

Add this handler to add the "Insert image" button to the UI and to listen to "Insert image" events.

```
typedef void (^ARDKInsertFromGalleryHandler)(NSViewController * _Nullable presentingVc,
                                             ARDKInsertImageResultHandler _Nullable completion);
```

Swift

Objective-C

```
ardkDocVC.insertFromGalleryHandler = {(presentingVc:NSViewController?,
                                       completion:ARDKInsertImageResultHandler?) in

/// open up a chooser for image files
let panel:NSOpenPanel = NSOpenPanel()
panel.allowsMultipleSelection = false
panel.canChooseDirectories = false
panel.canChooseFiles = true

panel.allowedFileTypes = ["jpg","jpeg","png"]

panel.beginSheetModal(for: (presentingVc?.view.window)!,
         completionHandler:{(result:NSApplication.ModalResponse) in

         if result == .OK {
            let url:URL? = panel.urls[0]
```

(continues on next page)

(continued from previous page)

```
            if url != nil {
                completion?(url!.path)
            }
        }
    }
)
```

}

```
ardkDocVC.insertFromGalleryHandler = ^(NSViewController *presentingVc,
                  ARDKInsertImageResultHandler  completion) {

   /// open up a chooser for image files
   NSOpenPanel *panel = [NSOpenPanel panel];
   panel.canChooseFiles          = YES;
   panel.canChooseDirectories    = NO;
   panel.allowsMultipleSelection = NO;

   NSArray* fileTypes = [NSArray arrayWithObjects:@"jpg", @"jpeg", @"png", nil];
   [panel setAllowedFileTypes:fileTypes];

   [panel beginSheetModalForWindow:presentingVc.view.window
                  completionHandler:^(NSInteger result)
   {
        if (result == NSFileHandlingPanelOKButton)
        {
            NSURL* url = [[panel URLs] objectAtIndex:0];

            if (url)
            {
               completion(url.path);
            }
        }
   }];
}
```

Images will be inserted at the point of the cursor position in the document, if there is no cursor position then image insertion is unavailable.

Note: Only applicable in the context of Word and Powerpoint documents.

5.3.2 Document Pasteboard

To connect your document view controller with the system paste board, to allow for copying and pasting text between apps, ensure to instantiate your own `Pasteboard` class which conforms to the `SODKPasteboard` protocol. Your `Pasteboard` class should typically connect with the general paste board as follows:

Swift

Objective-C

```
class Pasteboard: NSObject, SODKPasteboard {

   var ardkPasteboard_hasStrings: Bool  = {
        let pb:NSPasteboard = .general
        let str:String? = pb.string(forType: .string)
        return str != nil
   }()

   var ardkPasteboard_string: String! = {
        let pb:NSPasteboard = .general
        return pb.string(forType: .string)
   }()

   var ardkPasteboard_changeCount: Int = {
        let pb:NSPasteboard = .general
        return pb.changeCount
   }()

}
```

```
@implementation Pasteboard

   -(BOOL)ARDKPasteboard_hasStrings {
        NSPasteboard *pb  = [NSPasteboard generalPasteboard];
        NSString     *str = [pb stringForType:NSPasteboardTypeString];
        return (str != nil);
   }

   -(NSInteger)ARDKPasteboard_changeCount {
        NSPasteboard *pb = [NSPasteboard generalPasteboard];
        return pb.changeCount;
   }

   -(NSString *)ARDKPasteboard_string {
        NSPasteboard *pb = [NSPasteboard generalPasteboard];
        return [pb stringForType:NSPasteboardTypeString];
   }

@end
```

When ready set an instance of your `Pasteboard` class against the `pasteboard` variable on your instance of `ARDKDocumentViewController`.

Swift

Objective-C

```
ardkDocVC.pasteboard = Pasteboard()
```

```
ardkDocVC.pasteboard = [[Pasteboard alloc] init];
```

5.3.3 Export file as PDF

An application developer can export a Word, Powerpoint or Excel file as a PDF by using the SODKDoc instance method exportPDF as outlined below:

Instance method:

Swift

Objective-C

```
func exportPDF(to path: String,
         flags: SmartOfficeSaveFlags,
         completion block: @escaping (ARDKSaveResult, SOError) -> Void)
```

```
- (void)exportPDFTo:(nonnull NSString *)path
      flags:(SmartOfficeSaveFlags)flags
      completion:(nonnull void (^)(ARDKSaveResult, SOError))block;
```

Sample code:

Swift

Objective-C

```
let doc:SODKDoc = session!.doc as! SODKDoc

doc.exportPDF(to:filePath,
           flags:SmartOfficeSaveFlags_NoTemporary,
      completion:{(result:ARDKSaveResult, error:SOError) in

       switch result {

              case ARDKSave_Succeeded:
                  Swift.print("save to PDF success")
                  break

              case ARDKSave_Error:
                  Swift.print("save to PDF error")
                  break

              case ARDKSave_Cancelled:
                  Swift.print("save to PDF cancelled")
                  break

              default:

                  break
          }
```

(continues on next page)

(continued from previous page)

```
        }
    )
```

```
SODKDoc *doc = ((SODKDocSession *)session).doc;

[doc exportPDFTo:savePdfURL.path
           flags:SmartOfficeSaveFlags_NoTemporary
      completion:^(ARDKSaveResult result, SOError error) {

        switch (result) {

            case ARDKSave_Succeeded:
                NSLog(@"save to PDF success");
                break;

            case ARDKSave_Error:
                NSLog(@"save to PDF success");
                break;

            case ARDKSave_Cancelled:
                NSLog(@"save to PDF cancelled");
                break;

            default:
                break;
        }

    }
];
```

5.3.4 New documents

To create a new document in SmartOffice an application developer should include the blank documents as template documents within the application bundle and then open copies of them for the user to work with.

5.3.5 Closing a document

In the event of a user closing the document window then your bespoke `NSDocument` class should listen for this by overriding the method `shouldCloseWindowController` and insert logic to see if the file has changed. If there has been a change then the application logic should prompt the user whether they wish to save the file or not.

The close document method is associated on the SmartOffice View Controller as follows:

Sample code:

Swift

Objective-C

```
docVC.closeDocument()
```

```
[docVC closeDocument];
```

To detect for document changes use the following read-only variables which are available in your instance of `ARDKDocumentViewController`.

variable name	variable type
documentHasBeenModified	`bool`
documentIsBeingSaved	`bool`

Note: When you are ready to exit the document and close the window it is good practice to call the `closeDocument` method on your instance of `ARDKDocumentViewController` beforehand to free up memory.

CHAPTER

SIX

WINDOWS SDK

6.1 Getting Started

6.1.1 System requirements

- Visual Studio 2019 (or above)

See: recommended system requirements for Visual Studio 2019 from Microsoft.

Note: Visual Studio 2019 Community is not licensed for use with developing commercial products.

6.1.2 Starting a project

The SmartOffice SDK is built for MFC applications written in *C++*. Your Visual Studio project should therefore predominately be a *C++* project. If starting a new project it is recommended to use the "Windows Desktop Application" or "MFC App" project template.

6.1.3 Adding the SDK to your project

You should have been supplied with a zip bundle containing libraries and sample projects for your development work. The `lib` folder in the bundle effectively contains the SDK. Project properties for your application need to be correctly configured in order to use the SDK.

Project properties

Additional Include Directories

Your Visual Studio project should link to this location as defined in the project's properties *C/C++* setting for "Additional Include Directories".

See the Visual Studio 2019 screenshot below for an example:

Note: Ensure to select "All Platforms" in your configuration when including the library.

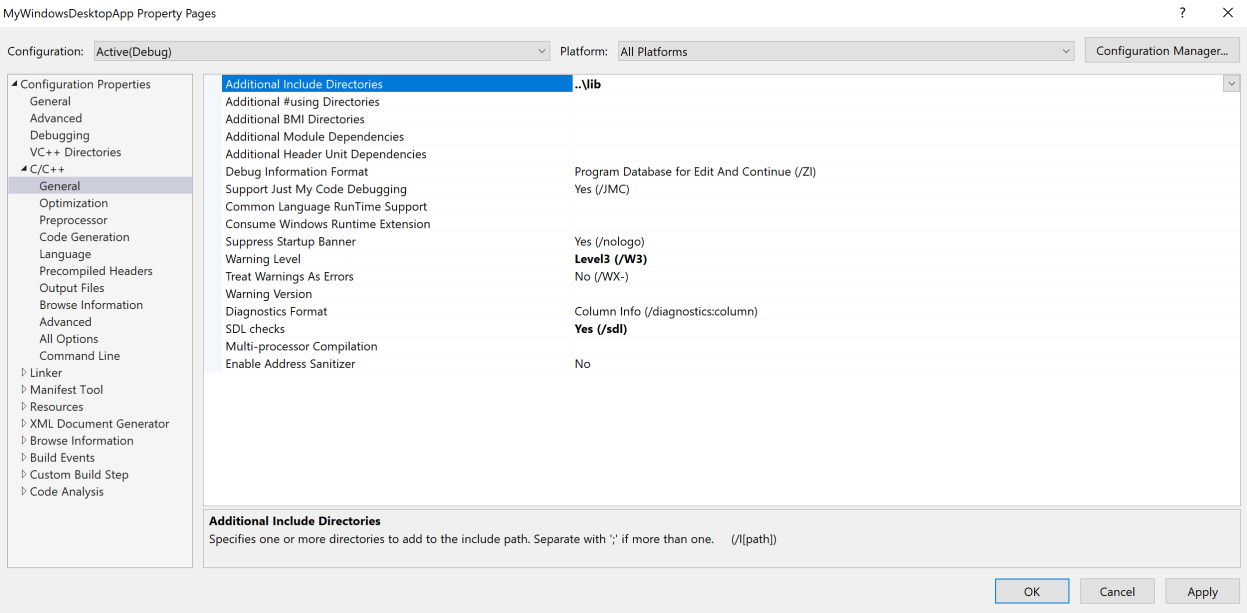

Fig. 1: Visual Studio project properties

Linker Dependencies

Your Visual Studio project should have “Additional Dependencies” & “Additional Library Directories” setup to read the `SOControl` libs as well as the main `lib` directory.

See the Visual Studio 2019 screenshot below for an example:

Required DLLs

Ensure to copy the following DLLs to your project folder:

- `SOControlMD.dll`
- `SOControlMDd.dll`
- `SOControlUIBridge.dll`

6.1.4 Verify your integration

1. To ensure you have correctly added the `lib` folder to your project ensure that your source code files can reference the following without error:

```
#include "SOControl.h"
```

2. Compile, build and run your project to ensure that the DLLs can be found and that there are no errors.

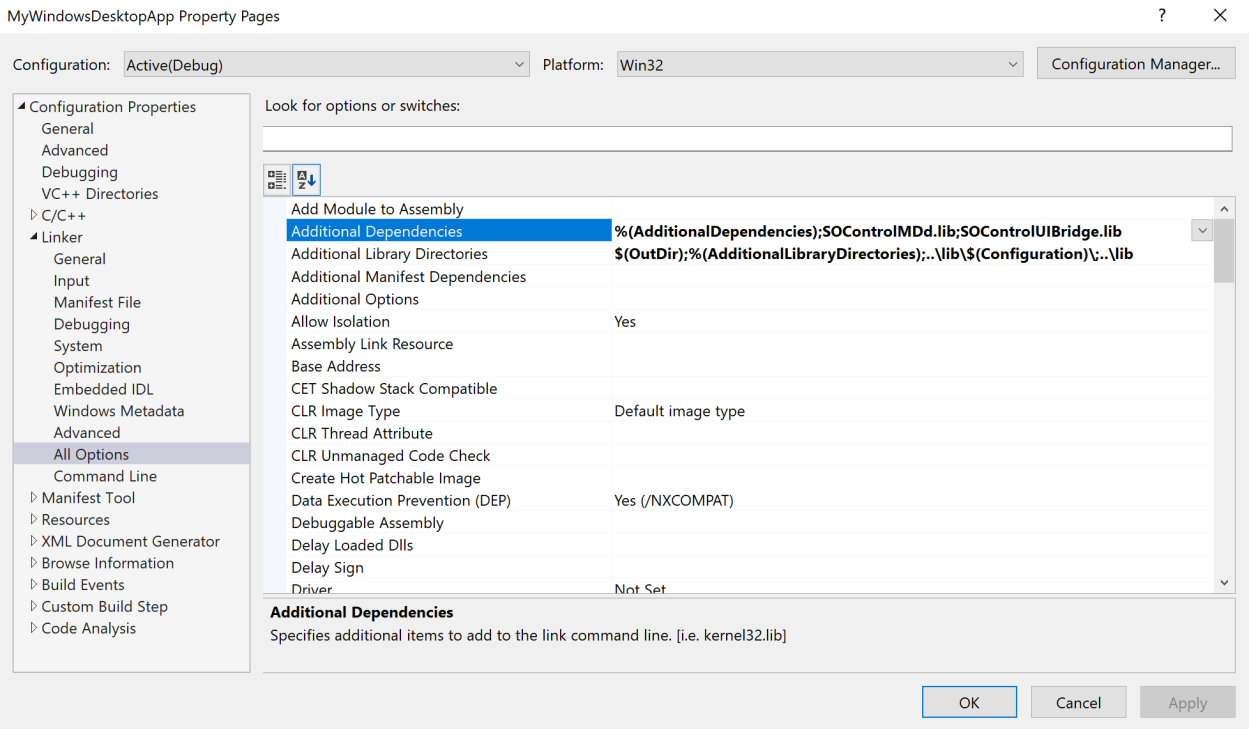

Fig. 2: Linker properties

6.2 SOControl

The SDK requires to have a subclassed instance of the SmartOffice Control `SOControl` ready and initialized in order to open, close and manage documents.

`SOControl` is also responsible for presenting a default UI around the opened document which allows for document editing and other file operations.

It is advisable to have a single instance of this subclassed object and to have it globally availabe throughout your application until it is no longer required.

The `SOControl` constructor is defined as follows:

```
SOControl();
```

6.2.1 Initialize

Call this method to initialize the `SOControl` with the required windows and optional settings.

The `SOControl` initialization method is defined as follows:

```
/**
 * @param parent[in]     HWND for the parent window
 * @param socWnd[out]    HWND of the created control window
 * @param settings[in]   User defined settings.
 *
 * @return 0 on success, else error
```

(continues on next page)

(continued from previous page)

```
 */
SOCtlError Initialize(HWND parent, HWND *socWnd, SOSettings *settings=nullptr);
```

Note: To define any required Settings for your document session see *SOSettings*.

The `SOCtlError` enum is defined as follows:

```
enum
{
   SOCtlAppError_NoError = 0,
   SOCtlAppError_NotInitialized = 0x01000000,
   SOCtlAppError_FailedToInitialize,
   SOCtlAppError_FailedToLoadResources,
   SOCtlAppError_InUse,
   SOCtlAppError_FileIOInProgress,  /* Cannot continue the requested operation because␣
↪of unfinished File I/O */
   SOCtlAppError_OperationFailed
};
```

Example: "MySmartOfficeControl"

An example of creating and initializing a subclassed instance of `SOControl`, in this case named "MySmartOfficeControl", is as follows:

Header file example:

```
#pragma once
#include "SOControl.h"

class MySmartOfficeControl : public SOControl {

public:

    // Constructor
    MySmartOfficeControl() { }

};
```

Class file example:

```
#include "MySmartOfficeControl.h"

MySmartOfficeControl *mySOC = new MySmartOfficeControl();
HWND hChild;
result = mySOC->Initialize(hWnd, &hChild, nullptr);

if (result != SOCtlAppError_NoError)
{
   DoDebugOutput("Failed to initialize!!\r\n");
   // Failed to init. closing.
```

(continues on next page)

(continued from previous page)

```
   SendMessage(hWnd, WM_CLOSE, 0, 0);
}
```

6.2.2 Finalize

Call this method on your subclassed instance of `SOControl` to free up resources allocated to the control.

Instance method:

```
/**
 * This method deletes the internal resources allocated for the control.
 *
 * @return 0 on success, else error
 */
SOCtlError Finalize();
```

6.2.3 Open

Use this method to open a file.

Instance method:

```
/**
 * @param filepath   path to the file to load (in utf8)
 * @param newdoc     true if creating new doc from template
 *
 * @return 0 on success, else error
 */
SOCtlError Open(const char *filepath, bool newdoc = false);
```

6.2.4 Close

Use this method to close a file.

Instance method:

```
/**
 * @return 0 on success, else error
 *
 * Note. SOCtlAppError_FileIOInProgress can be returned if Close() cannot be continued,
 *       because there's an on-going file operation(such as file save). in this case, it
 *       is recommended to call Close() after the operation completes.
 */
SOCtlError Close();
```

6.2.5 Save

This method saves a file. Call the `Save` method on your subclassed instance of `SOControl` for "Save" and "Save As" operations.

Instance method:

```
/**
 * @param filepath           path to the file to save (in utf8)
 *                           null for overwriting the file
 * @param completionCallback The method to be called once the operation is
 *                           complete. if this is null, then saving operation
 *                           is synchronous.
 *
 * @return 0 if save operation gets started, or saving has completed successfully.
 *         else error.
 *
 * Note. SOCtlAppError_FileIOInProgress can be returned if Save() cannot be continued,
 *       because there's an on-going file operation (such as file save).
 */
SOCtlError Save(const char *filepath, SODocSaveListener *completionCallback);
```

Note: If the filepath parameter is *not null* then this is considered to be a "Save As" operation.

6.2.6 Save As PDF

This method saves a file as pdf.

Instance method:

```
/**
 * @param filepath           path to the file to save as pdf (in utf8)
 *
 * @param completionCallback The method to be called once the operation is complete.
 *
 * @return 0 if save operation gets started successfully.
 *
 * Note. SOCtlAppError_FileIOInProgress can be returned if SaveAsPdf() cannot be␣
↪continued,
 *       because there's an on-going file operation (such as file save).
 */
SOCtlError SaveAsPdf(const char *filepath, SODocSaveListener *completionCallback);
```

6.2.7 Document has changes

A simple boolean check to test if the document has unsaved changes.

Instance method:

```
/**
 * @return true if changes exist. false otherwise
 */
bool HasChanges();
```

6.2.8 Document can finish

This checks to see what operations there may be on a document and whether it can therefore be considered to be able to be "finished" or not.

Instance method:

```
/**
 * @return false if the control cannot be finished, because an internal unbreakable
 *         operation (such as file saving/exporting..) is currently in progress.
 */
bool CanFinish();
```

6.2.9 Get supported file extensions

Call this method to retrieve a list of supported file extensions.

Instance method:

```
/**
 * @return  comma-delimited list of extensions, without the leading ".".
 *          The caller should free the returned pointer..
 */
static char *GetSupportedFileExtensions(void);
```

6.2.10 Set SecureFS

This method sets the `SecureFS` handler to be used by the `SOLib`.

Instance method:

```
/**
 * The passed handler is picked up ONLY ONCE by the SOLib
 * when the first SOControl gets initialized and it is shared
 * with other SOControls.
 */
static void SetSecureFS(SOSecureFs *securefs);
```

6.2.11 Set the Clipboard

This method sets Clipboard handler to be used by the SOLib.

Instance method:

```
/**
 * The passed handler is picked up ONLY ONCE by the SOLib
 * when the first SOControl gets initialized and it is shared
 * with other SOControls.
 */
static void SetClipboardHandler(SOClipboardHandler *clipboard);
```

6.3 SOSettings

To define specific settings the SDK requires to have a subclassed instance of SOSettings initialized and passed through to your SOControl initialization.

Your subclassed implementation should override the following SOSettings methods as required in order to define your available UI features:

Method Return	
isExtClipboardInEnabled	boolean true if clipboard input is enabled.
isExtClipboardOutEnabled	boolean true if clipboard output is enabled.
isEditingEnabled	boolean true if editing is enabled.
isPrintingEnabled	boolean true if the printing feature is enabled.
isSaveEnabled	boolean true if the “save” feature is enabled.
isSaveAsEnabled	boolean true if the “save as” feature is enabled.
isSaveAsPdfEnabled	boolean true if the “save as PDF” feature is enabled.
isShareEnabled	boolean true if the “share” feature is enabled.
isImageInsertEnabled	boolean true if the “insert image” feature is enabled.
isPhotoInsertEnabled	boolean true if the “insert photo” feature is enabled.
isTrackChangesFeatureEnabled	boolean true if the track changes feature is enabled.
isOpenPdfInEnabled	boolean true if the “open PDF in” feature is enabled.

6.4 File operations

Assuming you have instantiated and initialized and subclassed `SOControl` correctly you are able to use the control's API to manage documents.

6.4.1 Opening a document

```
/**
 * This method opens a file.
 *
 * @param filepath   path to the file to load (in utf8)
 * @param newdoc     true if creating new doc from template
 *
 * @return 0 on success, else error
 */
SOCtlError Open(const char *filepath, bool newdoc = false);
```

An example of this, with an subclassed instance of `SOControl` named as "mySOC", is as follows:

Sample code:

```
char *documentPath = "C:/documents/my-document.docx";
SOCtlError result = mySOC->Open(documentPath);
```

Note: If the document is considered to be a template then the `newdoc` parameter should be set to true. Template documents cannot be saved over and therefore the Default UI will only allow for the "Save As" function as the available saving option.

6.4.2 Document handlers

Your bespoke instance of `SOControl` (as previously exemplified with *"MySmartOfficeControl"*) should override the following `SOControl` methods as required in order to listen and respond to document handler events:

Method	Parameters	Return	Description
onDocumentLoadingError	`int` - error number.	`boolean` true if the error is processed	This method is used to monitor errors in the process of loading a document.
saveAsHandler	`const char *` -the filename, `SOSaveAsComplete` - the method to be called once the operation is complete		This method will be called when the "Save As" button is pressed. The implementor should allow the user to select the file save location, execute the save then inform application of the new location or failure/cancellation of the operation via completion callback.
saveAsPdfHandler	`const char *` - the filename.		This method will be called when the application "Save As PDF" button is pressed. The implementor should allow the user to select the file save location, then execute the `saveAsPdf` operation.
postSaveHandler	`SOSaveAsComplete` - the method to be called once the operation is complete.		This method will be called after the file is saved internally. The implementation can be minimal.
openPdfInHandler	`const char *` - the filename.		This method will be called when the application "Open PDF In" button is pressed. The implementor should export the document to a suitable location suitable for opening in a partner application.
photoInsertHandler	`SOFileSelectionComplete` - the method to be called once the operation is complete.		This method gets called when the photo insert button is pressed.
imageInsertHandler	`SOFileSelectionComplete` - the method to be called once the operation is complete.		This method gets called when the image insert button is pressed.
openUrlHandler	`const char *` - the URL of the external link.		This method gets called when an external link has been activated. The implementor should decide how to handle the URL as required.
shareHandler			This method gets called when the share button is pressed. The implementor should export the document to a suitable location in preparation for sharing.

Your subclassed `SOControl` header might therefore be as follows:

Sample code:

```
#pragma once
#include "SOControl.h"

class MySmartOfficeControl : public SOControl {

public:

    // Constructor
    MySmartOfficeControl() { }
```

(continues on next page)

(continued from previous page)

```
    virtual bool onDocumentLoadingError(int error) override;
    virtual void postSaveHandler(SOSaveAsComplete *completionCallback) override;
    virtual void saveAsHandler(const char *filename, SOSaveAsComplete *listener)␣
↪override;
    virtual void saveAsPdfHandler(const char *filename) override;
    virtual void openPdfInHandler(const char *filename) override;
    virtual void imageInsertHandler(SOFileSelectionComplete *listener) override;
    virtual void photoInsertHandler(SOFileSelectionComplete *listener) override;
    virtual void openUrlHandler(const char *url) override;
    virtual void shareHandler() override;

};
```

Document Load Error

To listen for load errors your corresponding class implementation might implement this method as follows:

Sample code:

```
// Subtype MySmartOfficeControl to receive errors in the process of loading a document.
bool MySmartOfficeControl::onDocumentLoadingError(int error)
{
    char *str;
    switch(error)
    {
        case SOCtlDocErrorType_NoError:
            str = "No error\r\n";
            break;
        case SOCtlDocErrorType_UnsupportedDocumentType:
            str = "Unsupported document type\r\n";
            break;
        case SOCtlDocErrorType_EmptyDocument:
            str = "Document is empty\r\n";
            break;
        case SOCtlDocErrorType_UnsupportedEncryption:
            str = "Document uses unsupported encryption\r\n";
            break;
        case SOCtlDocErrorType_Aborted:
            str = "Document loading aborted by caller\r\n";
            break;
        case SOCtlDocErrorType_OutOfMemory:
            str = "Document is too large\r\n";
            break;
        case SOCtlDocErrorType_UnableToLoadDocument:
        default:
            str = "Document could not be loaded\r\n";
            break;
    }

    wchar_t *wStr = Win32_convertUtf8ToWchar(str);
```

(continues on next page)

(continued from previous page)

```
    if (wStr != NULL)
    {
        MessageBox(mOwner->hWndSOC,
                   wStr,
                   _T("Document Load Error"),
                   MB_OK | MB_ICONERROR);

        free(wStr);
    }

    return true;
}
```

Save As

The "Save As" handler should be responsible for opening up a window to allow the user to choose the filename and location for their document save.

Sample code:

```
void MySmartOfficeControl::saveAsHandler(const char *filename, SOSaveAsComplete␣
↪*listener)
{
    // Open a dialog for the user to choose a filename and location,
    // listen to the user input result,
    // use the `Save` method supplied by SOControl,
    // update your file state as required
    // trigger the completion method
}
```

When the filename and location is chosen then the *Save* method which should be invoked.

Save PDF

Add this handler to listen to "Save PDF" events.

Sample code:

```
void MySmartOfficeControl::saveAsPdfHandler(const char *filename)
{
    // Open a dialog for the user to choose a filename and location,
    // listen to the user input result,
    // use the `SaveAsPdf` method supplied by SOControl,
    // update your file state as required
}
```

When the filename and location is chosen then the *Save As PDF* method which should be invoked.

Save result

Save result can be handled by a `postSaveHandler` implementation, exemplified as follows:

Sample code:

```
void MySmartOfficeControl::postSaveHandler(SOSaveAsComplete *completionCallback)
{
    // Document has saved.
    // Please implement a custom save handler.

    // Trigger the completion of the operation.
    if (completionCallback)
        completionCallback->onComplete(SOSaveAsComplete::Cancelled, nullptr);
}
```

Open PDF In

Add this handler to listen to "Open PDF In" events.

Sample code:

```
void MySmartOfficeControl::openPdfInHandler(const char *filename)
{
    // The implementor should export the document to a suitable location
    // suitable for opening in a partner application.
    // Refer to the saveAsPdfHandler() function for exporting the document.
}
```

Insert camera image

Add this handler to listen to "Insert Photo" events.

Sample code:

```
void MySmartOfficeControl::photoInsertHandler(SOFileSelectionComplete *listener)
{
    // Open a dialog for the user to take a photo with the system camera,
    // listen to the user input result,
    // trigger the completion method
}
```

Insert gallery image

Add this handler to listen to "Insert Image" events.

Sample code:

```
void MySmartOfficeControl::imageInsertHandler(SOFileSelectionComplete *listener)
{
    // Open a dialog for the user to choose an image file,
    // listen to the user input result,
```

(continues on next page)

(continued from previous page)

```
    // trigger the completion method
}
```

Open URL

Add this handler to listen to URL links pressed in a document.

Sample code:

```
void MySmartOfficeControl::openUrlHandler(const char *url)
{
    // Open the URL in an external browser or not
}
```

Share

Add this handler to intercept when the "Share" button is pressed.

Sample code:

```
void MySmartOfficeControl::shareHandler()
{
    // The implementor should export the document to a suitable location prior to sharing
}
```

6.4.3 Document Pasteboard

Integrators should subclass and implement their own version of `SOClipboardHandler` and set it against their subclassed `SOControl` instance with the *SetClipboardHandler* method.

Method	Parameters	Return	Description
`~ClipboardHandler`			Destructor
`registerListener`	`OnClipboardChanged*` - the listener object.		This method registers the clipboard change listener.
`unregisterListener`	`OnClipboardChanged*` - the listener object.		This method unregisters the clipboard change listener.
`putPlainTextToClipboard`	`char*` - the text to be stored in the clipboard.		This method passes a string, cut or copied from the document, to be stored in the clipboard.
`getPlainTextFromClipboard`		`char*` - the text read from the clipboard.	This method returns the contents of the clipboard.
`clipboardHasPlaintext`		`bool` - returns true if the clipboard has data.	This method ascertains whether the clipboard has any data.

6.4.4 New documents

To create a new document in SmartOffice an application developer should include the blank documents as template documents within the application bundle and then allow the user to open them as required.

Note: See : *Opening a document*, for more on opening a template document.

6.4.5 Closing a document

In the event of a user closing the document window then, before proceeding, your application should first intercept the WM_CLOSE message and verify if the *document has changes*, which the user may want to save, or if the *document is busy*.

Note: Typically if the document has changes then, then your application should present a UI for the user to allow them to close without saving, save and close, or, cancel the closing operation.

If the document is busy then the user should be informed and the closing operation delayed or cancelled.

Once *Close* is then ready to be called, your `SOControl` instance should then call *Finalize* to free up memory allocations.

INDEX

I

L

M

N

O

P

R

S

T

U

V

W