MuPDF App Kit Documentation

Release 2.2

Artifex

Jan 15, 2024

GETTING STARTED

MuPDF App Kit libraries enable quick and easy *PDF* document viewing and editing for *Android* and *iOS* platforms.

CHAPTER

ONE

QUICK START GUIDE

MuPDF App Kit libraries enable quick and easy *PDF* document viewing and editing for *Android* and *iOS* platforms.

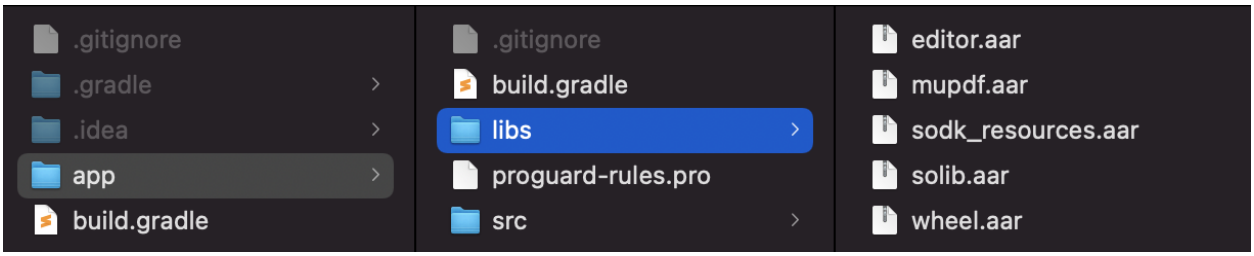
This guide demonstrates how our *Drop-in UI* solution allows you to get up and running with minimal coding effort.

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 Setup & integration

1.1.1 Android

1. Embed the *App Kit* libraries into your project folder:



2. Set the minimum *Android SDK* level in your *Gradle* file:

```
android {
    defaultConfig {
        minSdkVersion 23
        ...
    }
    ...
}
```

3. Reference the following dependencies in your *Gradle* file:

```
dependencies {
    ...
    // Dependency on view binding
```

(continues on next page)

(continued from previous page)

```
    implementation 'androidx.databinding:viewbinding:7.1.2'
    // Dependency on navigation fragments
    implementation 'androidx.navigation:navigation-fragment-ktx:2.2.2'
    implementation 'androidx.navigation:navigation-ui-ktx:2.2.2'

    // Dependency on local binaries
    implementation fileTree(dir: 'libs', include: ['*.aar'])
}
```

4. Include the default UI activity in your `AndroidManifest.xml` as follows:

```
<activity android:name="com.artifex.sonui.phoenix.DefaultUIActivity"
          android:exported="true"
          android:configChanges=
↪"orientation|keyboard|keyboardHidden|screenSize|smallestScreenSize|screenLayout|uiMode"
          android:screenOrientation="fullSensor"
          android:theme="@style/sodk_editor_mui_theme">
```

5. Import the required modules into your application code:

```
import com.artifex.sonui.phoenix.DefaultUIActivity
import com.artifex.solib.ArDkLib
import com.artifex.solib.ConfigOptions
```

6. Instantiate the App Kit `DefaultUIActivity` class with data representing your file `URI` and start the activity:

Kotlin

```
val defaultUI = Intent(this, DefaultUIActivity::class.java).apply {
    this.action = Intent.ACTION_VIEW
    this.data = uri
}

// Setup the document viewing configuration options
val appCfgOpts = ConfigOptions()
ArDkLib.setAppConfigOptions(appCfgOpts)

startActivity(defaultUI)
```

Java

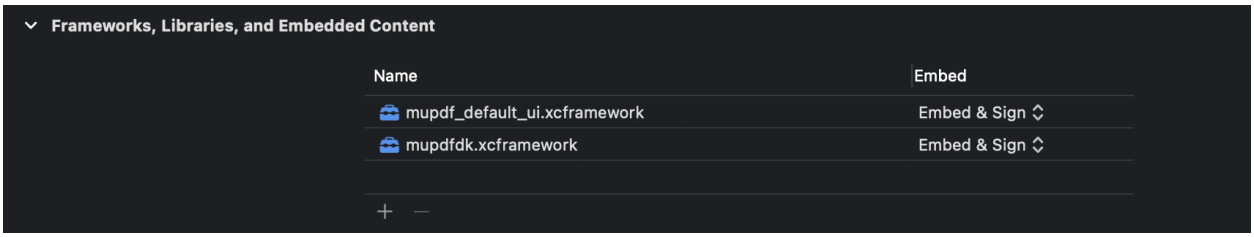
```
Intent defaultUI = new Intent(this, DefaultUIActivity.class);
defaultUI.setAction(Intent.ACTION_VIEW);
defaultUI.setData(uri);

// Setup the document viewing configuration options
ConfigOptions appCfgOpts = new ConfigOptions();
ArDkLib.setAppConfigOptions(appCfgOpts);

startActivity(defaultUI);
```

1.1.2 iOS

1. Embed the *App Kit* frameworks into your *Xcode* project:



2. Import the frameworks into your application code:

Swift

```
import mupdfdk
import mupdf_default_ui
```

Objective-C

```
#import "mupdfdk/mupdfdk.h"
#import "mupdf_default_ui/mupdf_default_ui.h"
```

3. Instantiate the *App Kit* `DefaultUIViewController` with your document `URL` and push it to your navigation controller:

Swift

```
DefaultUIViewController.viewController(url: url) { vc in
   if let vc = vc {
       vc.modalPresentationStyle = .fullScreen
       self.navigationController?.pushViewController(vc, animated: true)
   }
}
```

Objective-C

```
[DefaultUIViewController viewControllerWithUrl:url whenReady:^(DefaultUIViewController␣
↪*vc) {
   if (vc) {
      [self.navigationController pushViewController:vc animated:YES];
   }
}];
```

1.2 Customization

If you wish to build your own UI for file viewing then you should integrate your application code with the *App Kit API*.

CHAPTER

TWO

ABOUT

2.1 MuPDF SDK

MuPDF is a software framework for viewing and converting PDF, XPS, and E-book documents. It builds and runs on almost any OS you can imagine.

2.2 MuPDF App Kit

The MuPDF App Kits build upon the SDK to offer simple code samples to enable you to build mobile Apps based upon MuPDF. The developer documentation outlines how to use these App Kits for both Android and iOS platforms.

2.3 Licensing

You are free to use the MuPDF App Kit for evaluation purposes. However, if you require to publish an application, then a commercial license is required.

For more on how to use your license key see:

- *Android License Key*
- *iOS License Key*

2.3.1 Commercial license

A commercial license for MuPDF App Kit will be necessary before publishing your App project to an application store or other type of digital distribution platform for software applications. Please read the Software License Terms and Conditions document in its entirety.

CHAPTER

THREE

DOWNLOAD APP KIT

Artifex supplies projects, available to try for free, which utilize *App Kit* for both *Android* and *iOS*. Please use *Android Studio* or *Apple's Xcode* to try the projects out.

The sample projects contain the *App Kit* libraries, which are built using the corresponding version of *MuPDF*.

The demo version of *MuPDF App Kit* contains a watermark. To remove the watermark, you will need to purchase a commercial license from *Artifex*. Instructions on using the key to remove the watermark can be found in the "Getting Started" sections of this documentation.

CHAPTER

FOUR

ANDROID API

4.1 Getting Started

4.1.1 System requirements

Android minimum SDK: The MuPDF library needs Android version 6.0 or newer. Make sure that the `minSdkVersion` in your app's `build.gradle` is at least 23.

```
android {
    defaultConfig {
        minSdkVersion 23
        ...
    }
    ...
}
```

4.1.2 Adding the App Kit to your project

In order to include MuPDF in your app you need to use the Gradle build system.

The MuPDF App Kit 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.

To add the MuPDF App Kit to your project add the dependencies section in your `Module build.gradle`:

```
dependencies {
    ...
    // Dependency on view binding
    implementation 'androidx.databinding:viewbinding:7.1.2'
    // Dependency on navigation fragments
    implementation 'androidx.navigation:navigation-fragment-ktx:2.2.2'
    implementation 'androidx.navigation:navigation-ui-ktx:2.2.2'
    // Dependency on local binaries
    implementation fileTree(dir: 'libs', include: ['*.aar'])
}
```

Fetching the artifacts

Ensure to follow these steps:

- *Download* the latest MuPDF App Kit.
- Then copy all the aar files contained in `mupdf-test/app/libs` to your own app's `app/libs/` folder.
- Sync your `build.gradle` and then the App Kit libraries should be correctly configured and ready for use.

4.1.3 License Key

To remove the MuPDF document watermark from your App, you will need to use a license key to activate App Kit.

To acquire a license key for your app you should:

1. Go to artifex.com/appkit
2. Purchase your license(s)
3. In your application code add the API call to activate the license

---

Note: App Kit does not require network connection to validate a license key and there is no server tracking or cloud logging involved with key validation.

Using your License Key

If you have a license key for MuPDF App Kit, it will be bound to the App ID which you will have defined at the time of purchase. Therefore you should ensure that the App ID in your Android project is correctly set. This is defined in configuration within the Gradle file for the app (`build.gradle(:app)`).

```
defaultConfig {
      applicationId "your.license.key.app.id"
      ...
}
```

Once you have confirmed that the `applicationId` is correctly named, then call the following API early on in your application code:

Kotlin

```
import com.artifex.sonui.editor.DocumentView
...

val ctx: Context = this
val licenseKey:String = "put your license key here"
val ok:Boolean = DocumentView.unlockAppKit(ctx, licenseKey)
```

Java

```
import com.artifex.sonui.editor.DocumentView;
...

Context ctx = this;
```

(continues on next page)

(continued from previous page)

```
String licenseKey = "put your license key here";
Boolean ok = DocumentView.unlockAppKit(ctx, licenseKey);
```

Note: The `Context` parameter should reference your main application `Activity`.

The API call to set the license key should be made before you instantiate an App Kit `DocumentView`.

4.2 PDF Viewing

4.2.1 Presenting a Document

There are two fundamental ways of presenting a document to the screen. One way is to use the *Default UI* which includes an in-built user interface. The alternative is to load the document into a dedicated document view and provide your own *Custom UI* with listener methods for your document control.

4.2.2 Default UI

The *Default UI* is an App Kit UI created by Artifex which includes a user-interface for typical document features and actions. It is presented at the top of the document view and accommodates for both tablet and phone layout.

The *Default UI* aims to deliver a handy way of allowing for document viewing & manipulation without the need to provide your own *Custom UI*.

Instantiating the Default UI

An application developer should import the `DefaultUIActivity` and instantiate it as follows:

Kotlin

```
import com.artifex.sonui.phoenix.DefaultUIActivity
import com.artifex.solib.ArDkLib
import com.artifex.solib.ConfigOptions

...

val defaultUI = Intent(this, DefaultUIActivity::class.java).apply {
    this.action = Intent.ACTION_VIEW
    this.data = uri
}

// Setup the document viewing configuration options
val appCfgOpts = ConfigOptions()
ArDkLib.setAppConfigOptions(appCfgOpts)

startActivity(defaultUI)
```

Java

```
import com.artifex.sonui.phoenix.DefaultUIActivity;
import com.artifex.solib.ArDkLib;
import com.artifex.solib.ConfigOptions;

Intent defaultUI = new Intent(this, DefaultUIActivity.class);
defaultUI.setAction(Intent.ACTION_VIEW);
defaultUI.setData(uri);

// Setup the document viewing configuration options
ConfigOptions appCfgOpts = new ConfigOptions();
ArDkLib.setAppConfigOptions(appCfgOpts);

startActivity(defaultUI);
```

4.2.3 Custom UI

Providing a *Custom UI* means that the application developer is responsible for providing their own UI and functionality. This approach involves document presentation with an instance of `DocumentView`.

Considering that you have a valid document `Uri` from a Uri File instance, an application developer should initialize the document as explained in *Starting a Document View*.

Starting a Document View

Initializing the document view

On the `Activity` which you want to use to display the document you are required to setup the `DocumentView` class within the `onCreate` method as follows:

Kotlin

```
//  this needs to be done before calling super.onCreate
DocumentView.initialize(this)
```

Java

```
//  this needs to be done before calling super.onCreate
DocumentView.initialize(this);
```

Presenting the document

You should have an area of your UI dedicated to display the document defined in your corresponding activity's layout XML file (as an example this could be a constraint based layout called "document_view_holder"). Next we need to instantiate our `DocumentView`, add it to our layout and finally start the `DocumentView`.

The following code assumes have you have obtained a valid document `uri`:

Kotlin

```
// create a DocumentView for our document type
val filename = FileUtils.filenameFromUri(this, documentUri)
```

(continues on next page)

(continued from previous page)

```
val documentView:DocumentView = DocumentView.create(this, filename)

// add it to our layout
val parent = findViewById<ViewGroup>(R.id.document_view_holder)
parent.addView(documentView,
    RelativeLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
                                ViewGroup.LayoutParams.MATCH_PARENT))

documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions())
documentView.setDocDataLeakHandler(Utilities.getDataLeakHandlers())

// open it, specifying showUI = false;
documentView.start(documentUri, 0, false)
```

Java

```
// create a DocumentView for our document type
String filename = FileUtils.filenameFromUri(this, documentUri);
DocumentView documentView = DocumentView.create(this, filename);

// add it to our layout
ViewGroup parent = findViewById(R.id.document_view_holder);
parent.addView(documentView,
        new RelativeLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.
↪LayoutParams.MATCH_PARENT));

documentView.setDocConfigOptions(ArDkLib.getAppConfigOptions());
documentView.setDocDataLeakHandler(Utilities.getDataLeakHandlers());

documentView.start(documentUri, 0, false);
```

The `start()` method for `DocumentView` allows 3 parameters as follows:

- uri: `Uri` The document URI.
- page: `Int` The document page to start viewing from (Note: if this value is out of bounds then the document will render to the nearest available page).
- showUI: `Bool` Indicates whether to render and use the *Default UI* on top of the document view or not - for the *Custom UI* this should always be set to `false`.

Note: Don't forget that your `Activity` should also be responsible for setting up the `Activity` lifecycle interfaces previously explained.

You should also adhere to the document listeners which allow for feedback against document events.

Going to a Page

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

Kotlin

```
// note: page number is zero-indexed,
// thus this would show page 8 of your document
documentView.goToPage(7)
```

Java

```
// note: page number is zero-indexed,
// thus this would show page 8 of your document
documentView.goToPage(7);
```

Jump to first page:

Kotlin

```
documentView.firstPage()
```

Java

```
documentView.firstPage();
```

Jump to last page:

Kotlin

```
documentView.lastPage()
```

Java

```
documentView.lastPage();
```

In the code sample above `documentView` refers to the instance of your `DocumentView`. Furthermore this API should only be called after the document has initially loaded and had it's first render (see *Document Listeners - Document completed*).

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.

Kotlin

```
documentView?.enterFullScreen({
    // restore our UI
})
```

Java

```
if (documentView != null) {
    documentView.enterFullScreen(new Runnable() {
        @Override
        public void run() {
            // restore our UI
        }
    });
}
```

Document Page Viewer

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

Kotlin

```
// show
documentView?.showPageList()

// hide
documentView?.hidePageList()
```

Java

```
// show
if (documentView != null) {
    documentView.showPageList();
}

// hide
if (documentView != null) {
    documentView.hidePageList();
}
```

You can also query the presence of the page list UI with:

Kotlin

```
val isVisible:Boolean? = documentView?.isPageListVisible
```

Java

```
boolean isVisible = documentView.isPageListVisible();
```

4.3 Document Setup

4.3.1 Setup

PDF documents in App Kit are always rendered inside a document view instance. Files which require to instantiate and access document views should reference the following:

Kotlin

```
import com.artifex.sonui.editor.DocumentView
```

Java

```
import com.artifex.sonui.editor.DocumentView;
```

Your Android `Activity` should handle the regular Android Activity Lifecycle events and inform any `DocumentView` instance of the corresponding *Activity Events*. Additionally *Activity Interfaces* require to be setup for full App Kit functionality.

For the *Custom UI*, there are also a set of common document events during the lifecycle of a document. An application developer can set up *Listeners* to respond to these events as required.

4.3.2 Activity Events

Other events should be passed through to your `DocumentView` instance as follows:

Kotlin

```
public override fun onPause() {
    super.onPause()
    documentView?.onPause()
}

override fun onResume() {
    super.onResume()
    documentView?.onResume()
}

override fun onDestroy() {
    super.onDestroy()
    documentView?.onDestroy()
}

override fun onBackPressed() {
    documentView?.onBackPressed()
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    documentView?.onActivityResult(requestCode, resultCode, data)
}

override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)
```

(continues on next page)

(continued from previous page)

```
    documentView?.onConfigurationChange(newConfig)
}
```

Java

```
@Override
public void onPause() {
    if (documentView != null)
        documentView.onPause();
    super.onPause();
}

@Override
protected void onResume() {
    super.onResume();
    if (documentView != null)
        documentView.onResume();
}

@Override
protected void onDestroy() {
    super.onDestroy();
    if (documentView != null)
        documentView.onDestroy();
}

@Override
public void onBackPressed() {
    if (documentView != null)
        documentView.onBackPressed();
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (documentView != null)
        documentView.onActivityResult(requestCode, resultCode, data);
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    if (documentView != null)
        documentView.onConfigurationChange(newConfig);
}
```

Note: The above events with their corresponding document view calls are critical for document text editing and selection.

4.3.3 Activity Interfaces

Security setup

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

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

Note: There are no defaults for these interfaces.

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

Taken together, your own class implementations following these interfaces can be used to implement security at all the points where a user's data might flow into, or out of, the application.

SODataLeakHandlers

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

SOPersistentStorage

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

SOClipboardHandler

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

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.

If required, the following code (with it's own implementations of these classes) should be invoked at the start of your app's main activity as part of your setup.

Kotlin

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

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

(continues on next page)

(continued from previous page)

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

    ...
}
```

Java

```
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());

    ...
}
```

4.4 Raster Image Export

An application developer can export a document's pages into raster image formats by invoking the MuPDF library directly, opening the document and processing the resulting pages into a raster image file sequence.

4.4.1 Loading MuPDF without a UI

To load the MuPDF library and use it directly an application developer should import `SODKLib` and request a document load as follows:

Kotlin

```
import com.artifex.solib.*

fun loadDocument(
        activity: Activity,
        path: String
    ) {
    // use the registered configuration options of the application as
    // the document configuration options
    val docCfg = ArDkLib.getAppConfigOptions()
```

(continues on next page)

(continued from previous page)

```
    val lib = ArDkUtils.getLibraryForPath(activity, path)

    // Load the document
    val doc:ArDkDoc = lib.openDocument(path, object : SODocLoadListener {
        override fun onPageLoad(pageNum: Int) {

        }

        override fun onDocComplete() {
            // see sample app
            val pngTask = GeneratePngsTask()
            pngTask.execute(null as Void?)
        }

        override fun onError(error: Int, errorNum: Int) {
            // Called when the document load fails
        }

        override fun onSelectionChanged(
            startPage: Int,
            endPage: Int
        ) {
            // Called when the selection changes
        }

        override fun onLayoutCompleted() {
            // Called when a core layout is done
        }
    }, activity, docCfg)
}


private class GeneratePngsTask : AsyncTask<Void, Void, Boolean>() {
    // see sample app
    override fun doInBackground(vararg p0: Void?): Boolean {

    }
}
```

Java

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

public void loadDocument(final Activity activity,
                         final String path)
{
    // use the registered configuration options of the application as
    // the document configuration options.
    ConfigOptions docCfg = ArDkLib.getAppConfigOptions();

    ArDkLib lib = ArDkUtils.getLibraryForPath(activity, path);
```

(continues on next page)

(continued from previous page)

```
    // Load the document.
    ArDkDoc doc = lib.openDocument(path, new SODocLoadListener()
    {
        @Override
        public void onPageLoad(int pageNum)
        {

        }

        @Override
        public void onDocComplete()
        {
            // see sample app
            GeneratePngsTask pngTask = new GeneratePngsTask();
            pngTask.execute((Void)null);

        }

        @Override
        public void onError(final int error, final int errorNum)
        {
            // Called when the document load fails
        }

        @Override
        public void onSelectionChanged(final int startPage,
                                       final int endPage)
        {
            // Called when the selection changes
        }

        @Override
        public void onLayoutCompleted()
        {
            // Called when a core layout is done
        }
    }, activity, docCfg);
}

private class GeneratePngsTask extends AsyncTask<Void, Void, Boolean> {
    @Override
    protected Boolean doInBackground(Void... voids) {
        return null;
    }
}
```

By using the document load listeners an application developer should be able use a background task to process document pages as required and use the `createBitmapForPath` method of the `SODKLib` to export pages in bitmap format.

See the *Android Sample app* and the `ExportFileAsPng` class to reference a full example.

4.5 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*.

4.5.1 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.

Kotlin

```
if (documentView.isDocumentModified {
    documentView.save()
}
```

Java

```
if (documentView.isDocumentModified()) {
    documentView.save();
}
```

4.5.2 Save As

When saving a document using this method an application developer must provide a valid path on the file system to save to.

Kotlin

```
val docPath:String = "<YOUR_DOCUMENT_PATH>"

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

Java

```
String docPath = "<YOUR_DOCUMENT_PATH>";

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

4.5.3 Export

It is possible to export the content of a PDF into an external text file for simple text extraction. To do so an application developer should define a valid file path to use with the `exportToAPI`.

Kotlin

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

Java

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

Note: At present the only valid format to export to is a text file, so the format parameter should always be set to "txt".

Further formats will become available later.

4.5.4 Print

Application developers should call the `print()` method against the `DocumentView` instance to open up the print dialog.

Kotlin

```
documentView.print()
```

Java

```
documentView.print();
```

4.5.5 Search

Searching is invoked from the current document selection or cursor position and can be made forward or backward from this point. Successful searching automatically highlights the next instance of a found String and moves the document selection to that point.

Note: Search is case-insensitive.

Kotlin

```
fun search(text:String, forward:Boolean) {
    if (forward) {
        documentView.searchForward(text)
    } else {
        documentView.searchBackward(text)
    }
}
```

Java

```
private void search(String text, Boolean forward) {
    if (forward) {
        documentView.searchForward(text);
    } else {
        documentView.searchBackward(text);
    }
}
```

4.6 Custom UI

4.6.1 Document API

Options

An application developer can register options for the following in the App Kit SDK:

To do so a developer should instantiate `ConfigOptions`, set the required variables within that object, and then register it against the `SODKLib` object.

The following code example disables editing on a document:

Kotlin

```
import com.artifex.solib.ConfigOptions
import com.artifex.solib.ArDkLib

fun setupConfigOptions() {
    var configOptions:ConfigOptions = ConfigOptions()
    configOptions.isEditingEnabled = false
    ArDkLib.setAppConfigOptions(configOptions)
}
```

Java

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

public void setupConfigOptions() {
    ConfigOptions configOptions = new ConfigOptions();
    configOptions.setEditingEnabled(false);
```

(continues on next page)

(continued from previous page)

```
    ArDkLib.setAppConfigOptions(configOptions);
}
```

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.

Available document listeners are as follows:

Page Loaded

Called when pages are loaded.

Document Completed

Called when the document has completely loaded.

Password Required

Called when a password is required by the document.

On View Changed

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

Kotlin

```
documentView.setDocumentListener(object : DocumentListener {
    override fun onPageLoaded(p0: Int) {

    }

    override fun onDocCompleted() {

    }

    override fun onPasswordRequired() {

    }

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

    }
}
```

Java

```
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) {

    }
});
```

Document State Listeners

To setup listeners for when the document has loaded and when document is exited (`done()`) use the following API:

Kotlin

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

Java

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

    @Override
    public void done() {}
});
```

Note: Listeners must be added before any invocation of the *DocumentView start method*.

On Update UI

Called when UI updates are invoked by `DocumentView`.

Kotlin

```
documentView.setOnUpdateUI { }
```

Java

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

    }
});
```

Page Change

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

Kotlin

```
documentView.setPageChangeListener { pageNumber ->

}
```

Java

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

    }
});
```

Full Screen Mode

A `DocumentView` document has the ability to fill the screen and enter a uneditable mode for an optimal reading experience. It is the application developer's responsibility to turn off the UI that they do not wish to see when this mode is invoked and to ensure that their `DocumentView` instance fills the device screen. To turn desired UI back on again the `DocumentView` instance will invoke the application developer's closure method upon exiting full screen mode (when the user taps the screen).

Kotlin

```
findViewById<View>(R.id.fullScreenButton).setOnClickListener {
    // hide this activity's UI

    // put DocumentView in full screen mode
    if (dv != null) {
        dv.enterFullScreen {
```

(continues on next page)

(continued from previous page)

```
            // closure method to restore our UI upon exit
        }
    }
}
```

Java

```
findViewById(R.id.fullScreenButton).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // hide this activity's UI

        // put DocumentView in full screen mode
        if (documentView != null) {
            documentView.enterFullScreen(new Runnable() {
                @Override
                public void run() {
                    // closure method to restore our UI upon exit
                }
            });
        }
    }
});
```

Document Actions

Providing a Password

An application developer should provide a way to enter a password for a document and then provide it to the document view as follows:

Kotlin

```
documentView.providePassword("my-password")
```

Java

```
documentView.providePassword("my-password");
```

Undo

To undo a previous action (such as adding an annotation) use the the following method against your document instance.

Kotlin

```
if (documentView.canUndo()) {
    documentView.undo()
}
```

Java

```
if (documentView.canUndo()) {
    documentView.undo();
}
```

Redo

To redo a previous action (such as adding an annotation) use the the following method against your document instance.

Kotlin

```
if (documentView.canRedo()) {
    documentView.redo()
}
```

Java

```
if (documentView.canRedo()) {
    documentView.redo();
}
```

Get Selected Text

To get the selected text from a document an application developer should request the `selectedText` property against the `DocumentView` instance.

Kotlin

```
val selectedText:String? = documentView.selectedText
```

Java

```
String selectedText = documentView.getSelectedText();
```

Note: If there is no selected text in the document then a `null` value will be returned.

Can Select

To query whether you are able to select objects in a document (i.e. if the document is in read-only mode or not) then use the following:

Kotlin

```
val canSelect:Boolean = documentView.canSelect()
```

Java

```
Boolean canSelect = documentView.canSelect();
```

4.6.2 PDF Annotations

Annotations

Annotations includes functionality for *drawing*, *text markup* and *placing objects* on a *PDF* document.

App Kit provides the following sets:

- ***Document Drawing***
  - Freehand ink drawing
  - Line
  - Rectangle
  - Ellipse
  - Polygon
  - Polyline
- ***Document Text***
  - Highlight
  - Underline
  - Squiggle
  - Strikethrough
- ***Document Placement***
  - Note
  - Textbox
  - Stamp
  - Attachment
  - Link

Note: You should not set annotation modes on a document until the document has loaded, therefore ensure to add any API calls for annotation methods after *listening for the document loaded* event.

Document Drawing Annotations

Document text annotations depend on a "Drawing Annotation Mode" being activated. The following API allows for control over setting this mode as well as finding out what mode is currently active.

Draw Mode

Turning on drawing is as simple as calling the `setDrawModeOn()` method against your `DocumentView` instance. To turn off drawing just call the `setDrawModeOff()` method against your `DocumentView` instance.

Kotlin

```
import com.artifex.sonui.editor.DocumentView

fun setDocumentDrawMode(documentView:DocumentView, enable:Boolean) {
    if (enable) {
        documentView.setDrawModeOn()
    } else {
        documentView.setDrawModeOff()
    }
}
```

Java

```
import com.artifex.sonui.editor.DocumentView;

public void setDocumentDrawMode(documentView:DocumentView, enable:Boolean) {
    if (enable) {
        documentView.setDrawModeOn();
    } else {
        documentView.setDrawModeOff();
    }
}
```

To test if "Draw Mode" is active you can use `isDrawModeOn`

Kotlin

```
val dm:Boolean = documentView.isDrawModeOn
```

Java

```
Boolean dm = documentView.isDrawModeOn();
```

When *Draw Mode* is enabled, the user can draw an ink annotation with the selected *line thickness* and *color*. When *Draw Mode* is then disabled, the annotation is saved to the document.

Drawing Mode Annotation Types

The simplist drawing annotation is freehand ink drawing. As this was the first supported drawing annotation, in the Android App Kit SDK this is the default implementation. Therefore calling `setDrawModeOn()` without any parameter will enable ink annotations.

For other drawing types use the following with the `setDrawModeOn` method:

Drawing mode enumeration

Mode	Description
DocView.AnnotMode.NONE	None
DocView.AnnotMode.INK	Freehand ink
DocView.AnnotMode.LINE	Line
DocView.AnnotMode.RECTANGLE	Rectangle
DocView.AnnotMode.OVAL	Oval
DocView.AnnotMode.POLYGON	Polygon
DocView.AnnotMode.POLYLINE	Polyline
DocView.AnnotMode.ARROW	Arrow

Kotlin

```
documentView.drawMode = DocView.AnnotMode.LINE
```

Java

```
documentView.setDrawModeOn(DocView.AnnotMode.LINE);
```

When a "Drawing Annotation Mode" is disabled, the annotation is saved to the document.

Getting Draw Mode

To get the current draw mode use the `getDrawMode` method:

Kotlin

```
val dm:DocView.AnnotMode = documentView.drawMode
```

Java

```
DocView.AnnotMode dm = mDocumentView.getDrawMode();
```

Styling Drawing Annotations

Drawing annotations can be styled as follows:

Opacity

To get/set the opacity use the API as follows:

Kotlin

```
val color:Int = documentView.lineColor // getter
documentView.lineColor = 0xFF00FF00 // setter
```

Java

```
int color = documentView.getLineColor(); // getter
documentView.setLineColor(0xFF00FF00); // setter
```

Line Thickness

To get/set the line thickness use the API as follows:

Kotlin

```
val thickness:Float = documentView.lineThickness // getter
documentView.lineThickness = 2.0f // setter
```

Java

```
float thickness = documentView.getLineThickness(); // getter
documentView.setLineThickness(2.0f); // setter
```

Line Color

To get/set the line color use the API as follows:

Kotlin

```
val color:Int = documentView.lineColor // getter
documentView.lineColor = 0xFF00FF00 // setter
```

Java

```
int color = documentView.getLineColor(); // getter
documentView.setLineColor(0xFF00FF00); // setter
```

Line Endings

Line endings are only supported for the `DocView.AnnotMode.LINE` annotation. They can be set against the start & end of the line as follows:

Kotlin

```
documentView.setLineEndStyles(DocumentViewPdf.LINE_ENDING_BUTT, DocumentViewPdf.LINE_
↪ENDING_CIRCLE)
```

Java

```
documentView.setLineEndStyles(DocumentViewPdf.LINE_ENDING_BUTT, DocumentViewPdf.LINE_
↪ENDING_CIRCLE);
```

Line ending enumeration

Style	Description
DocumentViewPdf.LINE_ENDING_NONE	None
DocumentViewPdf.LINE_ENDING_BUTT	Butt
DocumentViewPdf.LINE_ENDING_SLASH	Slash
DocumentViewPdf.LINE_ENDING_CIRCLE	Cirle
DocumentViewPdf.LINE_ENDING_DIAMOND	Diamond
DocumentViewPdf.LINE_ENDING_OPEN_ARROW	Open arrow
DocumentViewPdf.LINE_ENDING_R_OPEN_ARROW	Right pointing open arrow
DocumentViewPdf.LINE_ENDING_SQUARE	Square
DocumentViewPdf.LINE_ENDING_CLOSED_ARROW	Closed arrow
DocumentViewPdf.LINE_ENDING_R_CLOSED_ARROW	Right pointing closed arrow

Document Text Annotations

Document text annotations depend on a "Text Annotation Mode" being activated. The following API allows for control over setting this mode as well as finding out what mode is currently active.

Find the Document Text Annotation Mode

Kotlin

```
val isHighlightMode:Boolean = documentView.isTextHighlightModeOn
val isTextSquigglyModeOn:Boolean = documentView.isTextSquigglyModeOn
val isTextStrikeThroughModeOn:Boolean = documentView.isTextStrikeThroughModeOn
val isTextUnderlineModeOn:Boolean = documentView.isTextUnderlineModeOn
```

Java

```
Boolean isHighlightMode = documentView.isTextHighlightModeOn();
Boolean isTextSquigglyModeOn = documentView.isTextSquigglyModeOn();
Boolean isTextStrikeThroughModeOn = documentView.isTextStrikeThroughModeOn();
Boolean isTextUnderlineModeOn = documentView.isTextUnderlineModeOn();
```

Toggle the text mode

Text Highlight Mode

Kotlin

```
documentView.toggleTextHighlightMode()
```

Java

```
documentView.toggleTextHighlightMode();
```

Text Squiggly Mode

Kotlin

```
documentView.toggleTextSquigglyMode()
```

Java

```
documentView.toggleTextSquigglyMode();
```

Text Strikethrough Mode

Kotlin

```
documentView.toggleTextStrikeThroughMode()
```

Java

```
documentView.toggleTextStrikeThroughMode();
```

Text Underline Mode

Kotlin

```
documentView.toggleTextUnderlineMode()
```

Java

```
documentView.toggleTextUnderlineMode();
```

Highlighting Selected Text

Another quick method to highlight *selected* text is to call the `highlightSelection()` method against your `DocumentView` instance.

Kotlin

```
documentView.highlightSelection()
```

Java

```
documentView.highlightSelection();
```

Note: Without a text selection being present in your document view calling `highlightSelection()` will have no effect.

Document Placement Annotations

Document placement annotations depend on a "Placement Annotation Mode" being activated. The following API allows for control over setting this mode as well as finding out what mode is currently active.

Find the Document Placement Annotation Mode

Kotlin

```
val isNoteModeOn:Boolean = documentView.isNoteModeOn
val isTextModeOn:Boolean = documentView.isTextModeOn
val isStampModeOn:Boolean = documentView.isStampModeOn
val isAttachmentModeOn:Boolean = documentView.isAttachmentModeOn
val isLinkModeOn:Boolean = documentView.isLinkModeOn
```

Java

```
Boolean isNoteModeOn = documentView.isNoteModeOn();
Boolean isTextModeOn = documentView.isTextModeOn();
Boolean isStampModeOn = documentView.isStampModeOn();
Boolean isAttachmentModeOn = documentView.isAttachmentModeOn();
Boolean isLinkModeOn = documentView.isLinkModeOn();
```

Setting Note Mode

Setting Note Mode On

Kotlin

```
documentView.setNoteModeOn()
```

Java

```
documentView.setNoteModeOn();
```

Setting Note Mode Off

Kotlin

```
documentView.setNoteModeOff()
```

Java

```
documentView.setNoteModeOff();
```

Setting Text Mode

Setting Text Mode On

Kotlin

```
documentView.setTextModeOn()
```

Java

```
documentView.setTextModeOn();
```

Setting Text Mode Off

Kotlin

```
documentView.setTextModeOff()
```

Java

```
documentView.setTextModeOff();
```

Setting Stamp Mode

Setting Stamp Mode On

Kotlin

```
documentView.setStampModeOn()
```

Java

```
documentView.setStampModeOn();
```

Setting Stamp Mode Off

Kotlin

```
documentView.setStampModeOff()
```

Java

```
documentView.setStampModeOff();
```

Setting Attachment Mode

Setting Attachment Mode On

Kotlin

```
documentView.setAttachmentModeOn()
```

Java

```
documentView.setAttachmentModeOn();
```

Setting Attachment Mode Off

Kotlin

```
documentView.setAttachmentModeOff()
```

Java

```
documentView.setAttachmentModeOff();
```

Setting Link Mode

Setting Link Mode On

Kotlin

```
documentView.setLinkModeOn()
```

Java

```
documentView.setLinkModeOn();
```

Setting Link Mode Off

Kotlin

```
documentView.setLinkModeOff()
```

Java

```
documentView.setLinkModeOff();
```

Deleting Annotations

Essentially all annotations are treated the same as simply selections once they have been selected by a user. This selection can then be removed by calling the `deleteSelection()` method against the `DocumentView` instance.

Kotlin

```
import com.artifex.sonui.editor.DocumentView

fun deleteDocumentSelection(documentView:DocumentView)  {
    documentView.deleteSelection()
}
```

Java

```
import com.artifex.sonui.editor.DocumentView;

public void deleteDocumentSelection(documentView:DocumentView) {
    documentView.deleteSelection();
}
```

Note: Without an annotation being currently selected in your document view calling `deleteSelection()` will have no effect.

Author

To get/set the annotation author use the API as follows:

Kotlin

```
val author:String = documentView.author // getter
documentView?.author = "John Doe" // setter
```

Java

```
String author = documentView.getAuthor();
documentView.setAuthor("Jane Doe");
```

4.6.3 PDF Redactions

Redactions

The redaction feature has been designed to work on text, images, and links. By selecting all or part of an image, text, or link, and applying redaction this then permanently redacts the selected information, making it impossible to retrieve the original data.

Redacted text and areas can be *marked*, and any marked redactions can be *removed*, however once the redactions are *applied* and the document is saved then the redacted information is blocked out with black to denote the area of redaction. At this point the original text and/or area is unretrievable.

Marking Text for Redaction

To mark text for redaction call the `redactMarkText()` method against the `DocumentView` instance. The document's currently selected text will then be marked with a red-outlined box to denote the redaction demarcation.

Kotlin

```
documentView.redactMarkText()
```

Java

```
documentView.redactMarkText();
```

Marking an Area for Redaction

To mark an area for redaction call the `redactMarkArea()` method against the `DocumentView` instance. The document will then enter a mode whereby the user can then draw a red-outlined boxed area to denote the redaction demarcation.

Kotlin

```
documentView.redactMarkArea()
```

Java

```
documentView.redactMarkArea();
```

Removing a Marked Redaction

To remove a marked redaction call the `redactRemove()` method against the `DocumentView` instance. The document's currently selected marked for redaction (i.e. a selection of text or an area with the red box outline) will then no longer be marked for redaction.

Kotlin

```
documentView.redactRemove()
```

Java

```
documentView.redactRemove();
```

Applying Redactions

Applying redactions means that all document selections which are marked for redaction will be redacted. To apply redactions call the `redactApply()` method against the `DocumentView` instance. Once redactions have been applied then the redacted text is blocked out with black and the red redaction demarcation areas are removed.

Kotlin

```
documentView.redactApply()
```

Java

```
documentView.redactApply();
```

Note: Redactions are not permanently set until the document is saved. If the document is exited without saving then the redactions will be lost.

4.6.4 PDF Signatures

Signatures

There are two distinct signature types which can be used with App Kit.

- *Digital Signatures*
- *E-signatures*

Digital signatures can be placed as signing areas in documents and also signed, whereas E-signatures can just be placed onto a document as part of a signing activity. E-signatures are the simplest use case and just involve a hand drawn signature without digital ID certification.

Digital Signatures

Placing a Digital Signature

Just like regular annotations, the mode within the document view needs to be set.

When the mode is set to "on" then when the user taps on the document a digital signature field will be placed. This field can be moved, repositioned or signed (given that the device has a valid signing certificate installed).

Once the document is saved then the position is set and the digital signature field becomes uneditable.

Kotlin

```
documentView.setDigitalSignatureModeOn() // turn on
documentView.setDigitalSignatureModeOff() // turn off
```

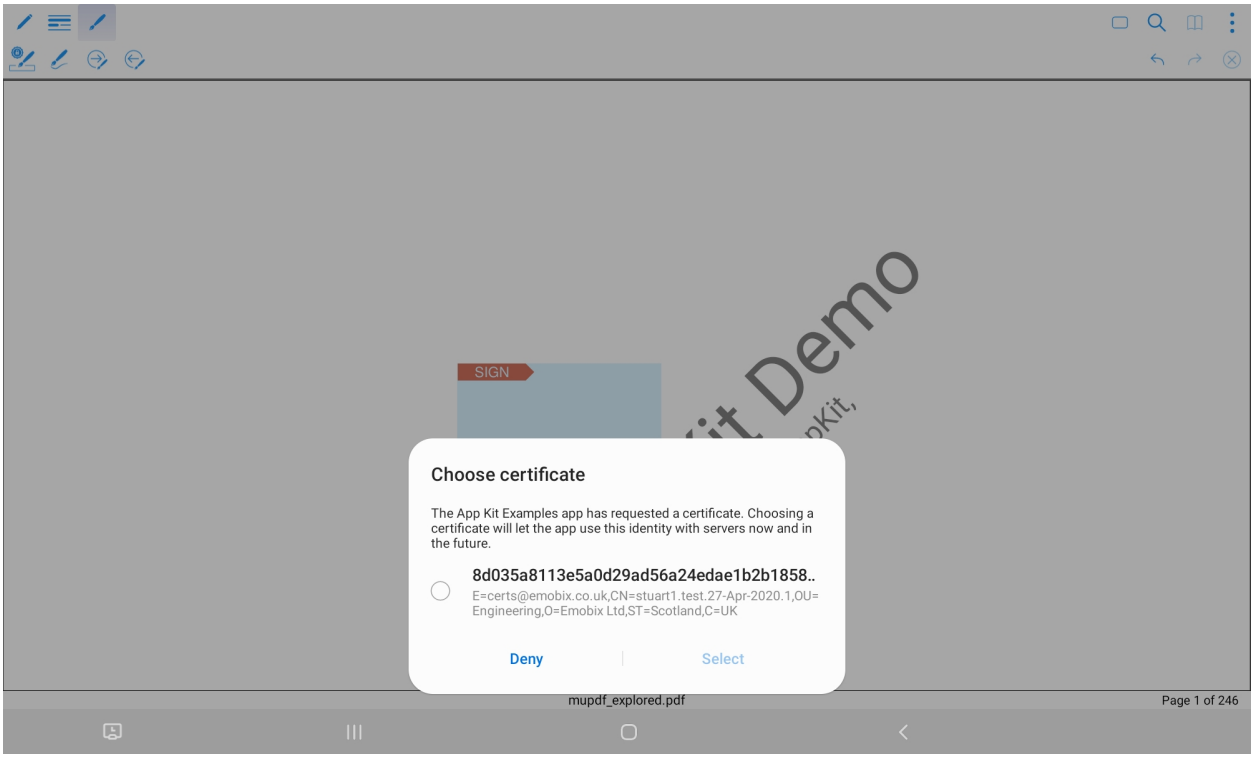
Java

```
documentView.setDigitalSignatureModeOn(); // turn on
documentView.setDigitalSignatureModeOff(); // turn off
```

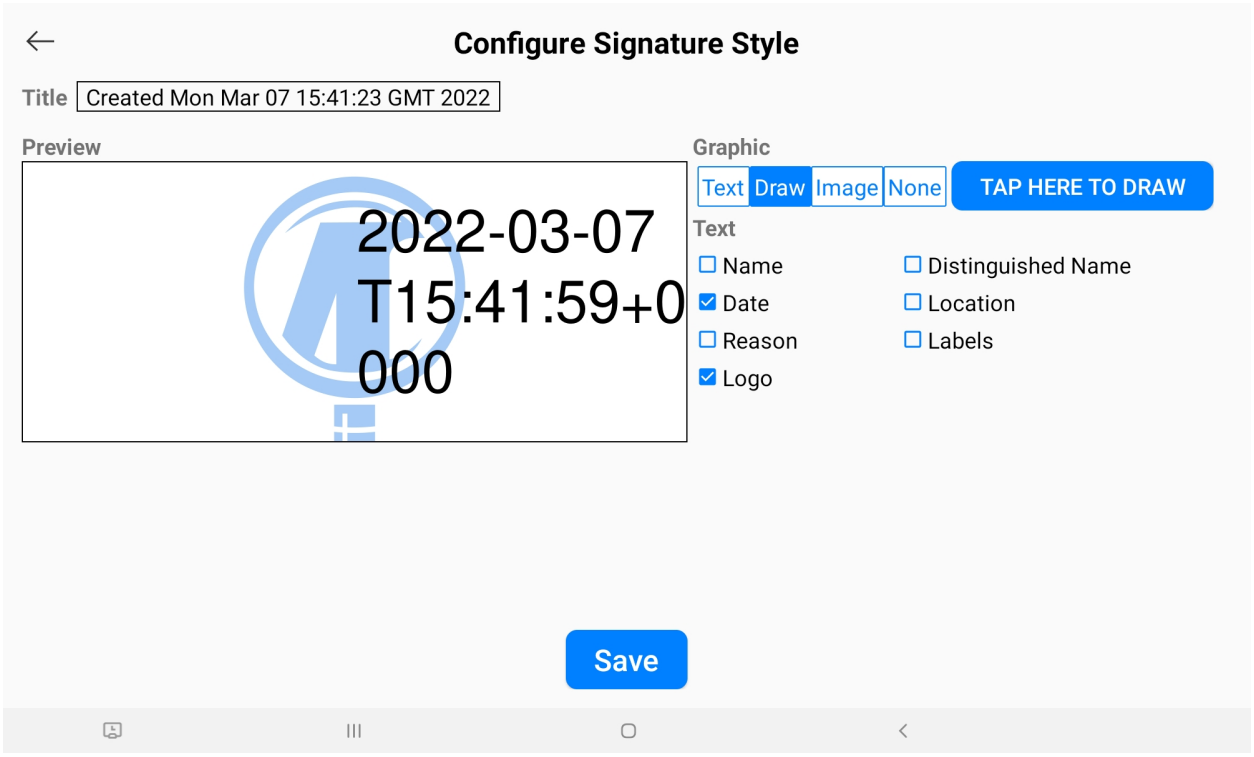
Signing a Digital Signature

Digital signatures use certificate-based digital IDs to authenticate signer identity and demonstrate proof of signing by binding each signature to the document with encryption.

If there is a set digital signature area on your document then tapping that will invoke App Kit to display a pre-built UI for the signing activity. This is a carefully considered UI and will not allow you to continue until you have imported a valid digital ID.



The UI for choosing a signing certificate



The UI for digital signing

E-signatures

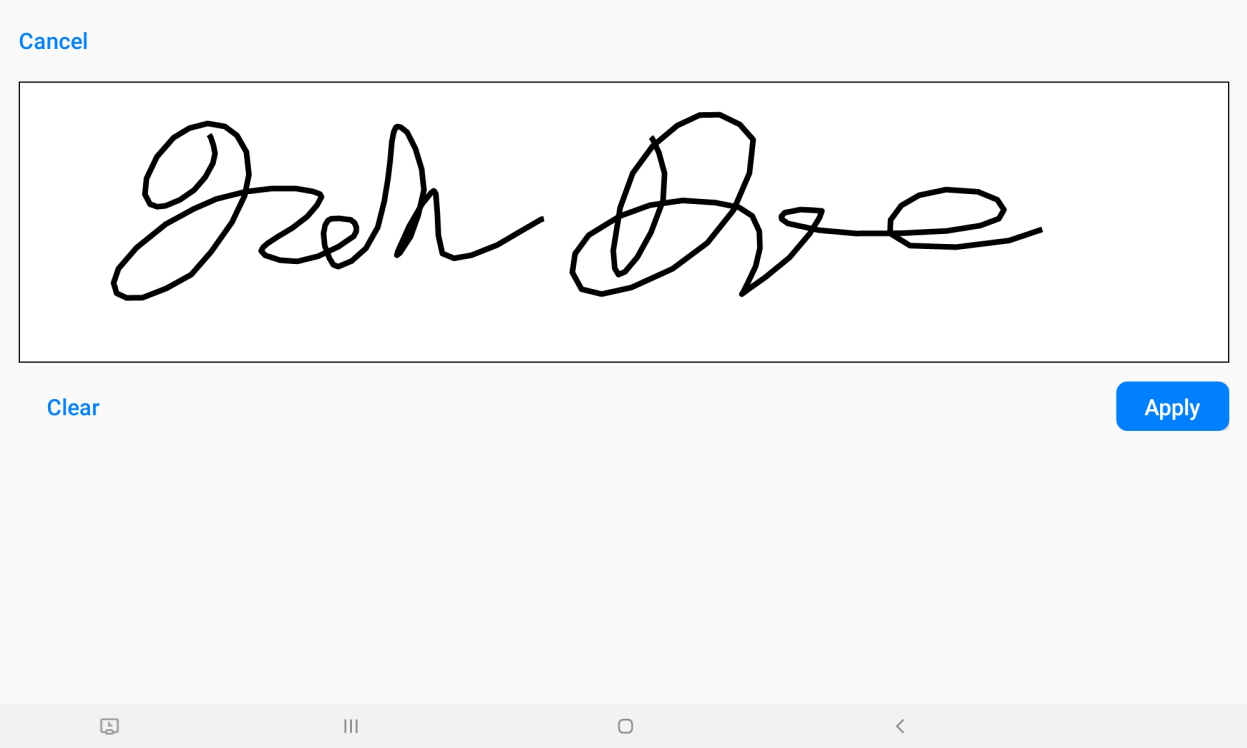
E-signatures should have been created (i.e. drawn by the user) before they are placed and there is some handy drop-in UI built into App Kit which allows for this. Calling the API to `setESignatureModeOn` (with a single parameter for the view to serve as an anchor for a popup that will appear) will invoke a UI to appear which allows the user to create or place an e-signature accordingly.

Kotlin

```
documentView.setESignatureModeOn(view) // turn on
documentView.setESignatureModeOff() // turn off
```

Java

```
documentView.setESignatureModeOn(view); // turn on
documentView.setESignatureModeOff(); // turn off
```



The UI for creating your e-signature

Querying Signature Modes

To query for signature mode use the following API:

Kotlin

```
val isDigitalSignatureMode:Boolean = documentView.isDigitalSignatureMode
val isESignatureMode:Boolean = documentView.isESignatureMode
```

Java

```
boolean isDigitalSignatureMode = documentView.isDigitalSignatureMode();
boolean isESignatureMode = documentView.isESignatureMode();
```

Searching for Signatures

If you have a document with multiple digital signatures you can query the document to search through and focus those signatures.

Next Signature

Kotlin

```
documentView.findNextSignature()
```

Java

```
documentView.findNextSignature();
```

Previous Signature

Kotlin

```
documentView.findPreviousSignature()
```

Java

```
documentView.findPreviousSignature();
```

Signature Count

To query a document to find out how many digital signature fields it may contain, use the following API:

Kotlin

```
val count:Int = documentView.signatureCount
```

Java

```
int count = documentView.getSignatureCount();
```

4.6.5 PDF Table of Contents

Overview

Not all PDF documents will have a Table of Contents, to check to see if the current document instance contains a Table of Contents an application developer should call the `isTOCEnabled` method once the document has fully loaded.

Kotlin

```
val hasTOC:Boolean = documentView.isTOCEnabled

if (hasTOC) {
    // enable UI (e.g. button/gesture) responsible for invoking TOC display
}
```

Java

```
boolean hasTOC = documentView.isTOCEnabled();

if (hasTOC) {
    // enable UI (e.g. button/gesture) responsible for invoking TOC display
}
```

In order to show the pre-built Table of Contents UI an application developer simply needs to call the tableOfContents method as follows:

Kotlin

```
documentView.tableOfContents()
```

Java

```
documentView.tableOfContents();
```

Note: Table of Contents is also known as “Bookmarks” or “Outline”.

Enumerating a Table of Contents

To get information about the table of contents the following API should be used to enumerate the listing:

Kotlin

```
documentView?.let { dv ->
    // get the TOC entries
    val entries = ArrayList<TocData>()

    dv.enumeratePdfToc(object : DocumentView.EnumeratePdfTocListener {
        override fun nextTocEntry(handle: Int, parentHandle: Int, page: Int,
                                  label: String, url: String, x: Float, y: Float) {
            val entry: TocData = TocData(handle, parentHandle, page, label, url, x, y)
            entries.add(entry)
        }
```

(continues on next page)

(continued from previous page)

```
        override fun done() {

        }
    })
}

private class TocData constructor(var handle: Int,
                                  var parentHandle: Int,
                                  var page: Int,
                                  var label: String,
                                  var url: String,
                                  var x: Float,
                                  var y: Float) {
    var level:Int = 0
    var tabIndent:String = ""
}
```

Java

```
final ArrayList<TocData> entries = new ArrayList<>();
documentView.enumeratePdfToc(new DocumentView.EnumeratePdfTocListener() {
    @Override
    public void nextTocEntry(int handle, int parentHandle, int page,
                             String label, String url, float x, float y) {
        TocData entry = new TocData(handle, parentHandle, page, label, url, x, y);
        entries.add(entry);
    }
    @Override
    public void done() {

    }
});

public class TocData {
    public int handle;
    public int parentHandle;
    public String label;
    private String url;
    private int page;
    private float x;
    private float y;

    TocData(int handle, int parentHandle, int page, String label, String url, float x,␣
↪float y) {
        this.handle = handle;
        this.parentHandle = parentHandle;
        this.page = page;
        this.label = label;
        this.url = url;
        this.x = x;
        this.y = y;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

CHAPTER

FIVE

IOS API

5.1 Getting Started

5.1.1 System requirements

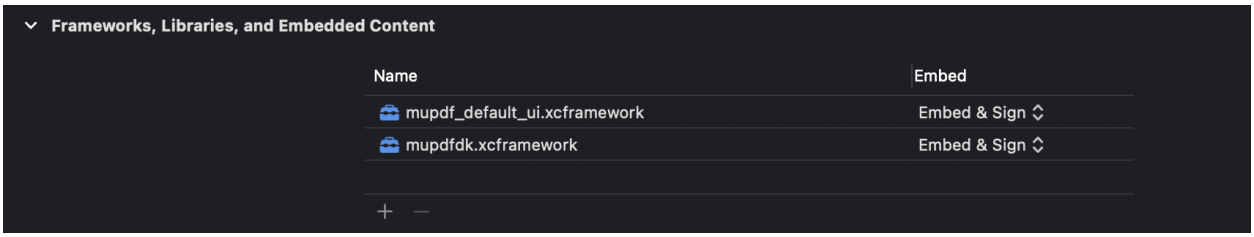
- Xcode minimum SDK: 9
- iOS minimum version: 11.0

5.1.2 Adding the App Kit to your project

You should have been provided with `mupdfdk.xcframework` & `mupdf_default_ui.xcframework` files for your project. These files are collectively the *MuPDF App Kit* and contain all the *App Kit* code required for your projects.

The linked frameworks 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:



Xcode embedding frameworks

Note: If there is an compile time error which complains that the linked framework cannot be found ensure to validate your workspace in Xcode with:

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

5.1.3 License Key

To remove the MuPDF document watermark from your App, you will need to use a license key to activate App Kit.

To acquire a license key for your app you should:

1. Go to artifex.com/appkit
2. Purchase your license(s)
3. In your application code add the API call to activate the license

---

Note: App Kit does not require network connection to validate a license key and there is no server tracking or cloud logging involved with key validation.

Using your License Key

If you have a license key for MuPDF App Kit, it will be bound to the App ID which you will have defined at the time of purchase. Therefore you should ensure that the App ID in your iOS project is correctly set. This is defined in the Bundle Identifier for the product `$(PRODUCT_BUNDLE_IDENTIFIER)`.

Once you have confirmed that your Bundle Identifier is correctly named, then call the following API early on in your application code:

Swift

```
import mupdfdk
...

let key:String = "put your license key here"
MuPDFDKDocumentViewController.unlockAppKit(key)
```

Objective-C

```
#import "mupdfdk/mupdfdk.h"
...

const char *key = "put your license key here";
[MuPDFDKDocumentViewController unlockAppKit:@(key)];
```

Note: The API call to set the license key should be made before you instantiate an App Kit `DocumentView`.

5.2 PDF Viewing

5.2.1 Present a Document View

There are two fundamental ways of presenting a document to the screen. One way is to use the *Default UI* which includes a user interface. The alternative is to load the document into a dedicated view controller and provide your own *Custom UI* with delegate methods available for your document control.

5.2.2 Default UI

The *Default UI* is an App Kit UI created by Artifex which includes a user-interface for typical document features and actions. It is presented at the top of the document view and accommodates for both tablet and phone layout.

The *Default UI* aims to deliver a handy way of allowing for document viewing & manipulation without the need to provide your own *Custom UI*.

Basic Usage

Assuming that your view controller has a navigation controller then local documents in the iOS main bundle can be presented by pushing an instance of `DefaultUIViewController` from your dedicated `UIViewController` as follows:

Swift

```
import mupdfdk
import mupdf_default_ui
...

let documentPath:String = "sample-document.pdf"

DefaultUIViewController.viewController(path: documentPath) { vc in
    if let vc = vc {
        vc.modalPresentationStyle = .fullScreen
        self.navigationController?.pushViewController(vc, animated: true)
    }
}
```

Objective-C

```
#import "mupdfdk/mupdfdk.h"
#import "mupdf_default_ui/mupdf_default_ui.h"
...

NSString *documentPath = @"sample-document.pdf";
[DefaultUIViewController viewControlllerWithPath:documentPath whenReady:^
↪(DefaultUIViewController *vc) {
    if (vc) {
        [self.navigationController pushViewController:vc animated:YES];
    }
}];
```

To load files via a defined *URL* use the following:

Swift

```
DefaultUIViewController.viewController(url: url) { vc in
    if let vc = vc {
        vc.modalPresentationStyle = .fullScreen
        self.navigationController?.pushViewController(vc, animated: true)
    }
}
```

Objective-C

```
[DefaultUIViewController viewControllerWithUrl:url whenReady:^(DefaultUIViewController␣
↪*vc) {
    if (vc) {
       [self.navigationController pushViewController:vc animated:YES];
    }
}];
```

Note: Files outside of your App bundle will need to be opened using the *advanced method*.

Advanced usage

Alternatively, an application developer can open a document with a session. This enables more control with regard to the file operations and settings for the document.

Opening a document within a session involves the following:

1) Initialize the MuPDF library:

Swift

```
let mupdfdkLib:MuPDFDKLib = MuPDFDKLib.init(settings: ARDKSettings())
```

Objective-C

```
MuPDFDKLib *mupdfdkLib = [[MuPDFDKLib alloc] initWithSettings:[[ARDKSettings alloc]␣
↪init]];
```

2) Initialize your own *FileState* class (ensuring to set the correct file path within it):

Swift

```
let fileState:MyFileState = MyFileState()
```

Objective-C

```
MyFileState *fileState = [[MyFileState alloc] init];
```

3) Initialize document settings with your required *Configuration Options*:

Swift

```
let docSettings:ARDKDocumentSettings = ARDKDocumentSettings()
docSettings.enableAll(true)
```

Objective-C

```
ARDKDocumentSettings *docSettings = [[ARDKDocumentSettings alloc] init];
[docSettings enableAll:YES];
```

4) Initialize the document session, `ARDKDocSession`, with your file state, MuPDF library and document settings instances, also set your signing delegate as required:

Swift

```
let session:ARDKDocSession = ARDKDocSession(fileState:fileState,
                                            ardkLib:self.mupdfdkLib,
                                        docSettings:docSettings)
session.signingDelegate = ARDKOpenSSLSigningDelegate()
```

Objective-C

```
ARDKDocSession *session = [ARDKDocSession sessionForFileState:fileState
                                                      ardkLib:self.mupdfdkLib
                                                  docSettings:docSettings];
session.signingDelegate = [[ARDKOpenSSLSigningDelegate alloc] init];
```

5) Instantiate the `DefaultUIViewController` with the session:

Swift

```
let vc:DefaultUIViewController = DefaultUIViewController.viewController(session: session)
```

Objective-C

```
DefaultUIViewController *vc = [DefaultUIViewController␣
↪viewControllerWithSession:session];
```

The Back Button

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

Swift

```
@IBAction func docCloseUnwindAction(_ sender: UIStoryboardSegue) {}
```

Objective-C

```
- (IBAction)docCloseUnwindAction:(UIStoryboardSegue *)sender {}
```

This method is triggered by the user pressing the 'back' button in the MuPDF UI.

Note: 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.

Configuration Options

When using the *Default UI* an application developer can optionally set certain configurable features.

The available settings conform to the `ARDKDocumentSettings` protocol which contains the following key/value pairs:

Note: The configuration options should be set before a document is opened

To use MuPDF with configuration options an application developer should use load documents using the *document session* method.

5.2.3 Custom UI

Your dedicated view controller should implement the following protocols *ARDKBasicDocViewDelegate* & *ARDKDocumentEventTarget*. For more on these protocols see the *Document API* page.

Swift

```
import mupdfdk

class DocumentViewController: UIViewController,
                              ARDKBasicDocViewDelegate,
                              ARDKDocumentEventTarget
```

Objective-C

```
#import "mupdfdk/mupdfdk.h"

@interface DocumentViewController() <ARDKBasicDocViewDelegate,
                                     ARDKDocumentEventTarget>

@end
```

These protocols will allow your `UIViewController` subclass to respond to events whilst presenting & interacting with the document.

There are several ways your `UIViewController` subclass can act as a container for an instance of `MuPDFDKBasicDocumentViewController`, along with UI related views. A layout described via a storyboard is one option, or a programmatic approach, as follows:

Swift

```
let documentViewController:MuPDFDKBasicDocumentViewController =
MuPDFDKBasicDocumentViewController.init(forPath:documentPath)
documentViewController.delegate = self
documentViewController.session.doc.add(self)
self.addChild(documentViewController)
documentViewController.view.frame = self.view.bounds
self.view.addSubview(documentViewController.view)
documentViewController.didMove(toParent:self)
```

Objective-C

```
MuPDFDKBasicDocumentViewController *documentViewController =
[MuPDFDKBasicDocumentViewController viewControllerForPath:documentPath];
documentViewController.delegate = self;
[documentViewController.session.doc addTarget:self];
[self addChildViewController:documentViewController];
documentViewController.view.frame = self.view.bounds;
[self.view addSubview:documentViewController.view];
[documentViewController didMoveToParentViewController:self];
```

Going to a Page

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

Swift

```
// note: page number is zero-indexed, thus this would show page 8 of your document
documentViewController.showPage(7)
```

Objective-C

```
// note: page number is zero-indexed, thus this would show page 8 of your document
[documentViewController showPage:7];
```

In the code sample above `documentViewController` refers to the instance of your `MuPDFDKBasicDocumentViewController`. Furthermore this API should only be called after the document has initially loaded and had it's first render (see *Document Listeners - Document completed*).

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.

Document Page Viewer

There is a drop-in UI available to enable a page viewer for your document which understands how to display a column of page thumbnails from information obtained from 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 as required.

Swift

```
let pagesVC = ARDKPagesViewController(session: session)
pagesVC.view.frame = CGRect
pagesVC.didMove(toParent: self)
pagesVC.delegate = self
pagesVC.selectPage(0)
self.addChild(pagesVC)
view.addSubview(pagesVC.view)
```

Objective-C

```
ARDKPagesViewController *pagesVC = [ARDKPagesViewController␣
↪viewControllerWithSession:self.session];
pagesVC.view.frame = CGRect;
[pagesVC didMoveToParentViewController:self];
pagesVC.delegate = self;
[pagesVC selectPage:0];
[self addChildViewController:pagesVC];
[view addSubview:pagesVC.view];
```

Note: The delegate protocol for `ARDKPagesViewController` is *ARDKPageSelectorDelegate*.

5.3 Document Setup

5.3.1 Setup

PDF documents in App Kit are always rendered inside a document view controller instance.

Depending on your integration your application should use one of the following document view controllers:

- `DefaultUIViewController` (for the *Default UI*)
- `MuPDFDKBasicDocumentViewController` (for a *Custom UI*)

The *Default UI* involves the least effort and should handle all your requirements unless you need nontrivial customization, more than altering colors and icons within the UI. Simply create and present a `DefaultUIViewController` instance. By using the *Default UI*, all user editing features are handled internally within the instance, without you as application programmer having to interact with the process.

For the *Custom UI*, a `MuPDFDKBasicDocumentViewController` instance provides a view on the document, but with minimal UI. It is the application developer's responsibility to create additional views and menus with UI elements via which the user can perform operations on the document. An application developer would typically create a `UIViewController` subclass that acts as a container for a `MuPDFDKBasicDocumentViewController` instance, along with additional views for UI. The `UIViewController` subclass would respond to the regular view-related `UIViewController` events in the usual way, and additionally implement some further protocols to make the UI responsive to events relating to the document and its view:

- *ARDKBasicDocViewDelegate* - this allows the document view to inform us of events and request information, regarding scrolling, user taps, etc.
- *ARDKDocumentEventTarget* - this allows the document itself to inform us of events, regarding the loading of the document and selection changes etc.
- *ARDKPageSelectorDelegate* - this allows to listen for page events such as selecting, deleting, duplicating or moving pages.

5.3.2 View Controller Interfaces

Security setup

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

- *ARDKPasteboard*
- *ARDKSettings*
- *ARDKFileState*
- *ARDKSecureFS*

Note: There are no defaults for these interfaces.

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

ARDKPasteboard

An interface that specifies the basis for implementing a class to control pasteboard copy and paste information.

ARDKSettings

An interface that specifies the basis for implementing a class to set the location of where temporary files are stored.

ARDKFileState

An interface that specifies the basis for implementing a class to provide information about the file being opened.

ARDKSecureFS

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.

5.4 Raster Image Export

An application developer can export a document's pages into raster image format by invoking the MuPDF library directly, opening the document and processing the resulting pages into a raster image file sequence.

5.4.1 Loading MuPDF without a UI

To load the MuPDF library and use it directly an application developer should import mupdfdk and request a document load as follows:

Swift

```
import mupdfdk

var mupdfdkLib:MuPDFDKLib?

func processDocument(_ docPath:String) {
    let settings:ARDKSettings = ARDKSettings()
    mupdfdkLib = MuPDFDKLib.init(settings: settings)

    let fullPath:String = "\(FileManager.default.urls(for: .documentDirectory,
                          in: .userDomainMask).last?.path ?? "")/\(docPath)"

    let doc:MuPDFDKDoc = mupdfdkLib.doc(forPath: fullPath,
                    of: MuPDFDKDoc.docType(fromFileExtension: docPath)) as! MuPDFDKDoc

    doc.successBlock = {
        /// See iOS Sample App
    }

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

    }

    doc.loadDocument()

}
```

Objective-C

```
#import "mupdfdk/mupdfdk.h"

@property (strong, nonatomic) MuPDFDKLib *mupdfdkLib;

- (void)processDocument:(NSString *)docPath {
    ARDKSettings *settings = [[ARDKSettings alloc] init];
    self.mupdfdkLib = [[MuPDFDKLib alloc] initWithSettings:settings];

    MuPDFDKDoc *doc = [_mupdfdkLib docForPath:[[self nsDocumentsDirectory].path
        stringByAppendingPathComponent:docPath]
    ofType:[MuPDFDKDoc docTypeFromFileExtension:docPath]];

    doc.successBlock = ^() {
        /// See iOS Sample App
    };

    doc.errorBlock = ^(ARDKDocErrorType error) {
        NSLog(@"Failed to load document: %x", error);
    };
```

(continues on next page)

(continued from previous page)

```
    [doc loadDocument];
}

- (NSURL *)nsDocumentsDirectory {
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
                inDomains:NSUserDomainMask] lastObject];
}
```

Upon document load completion an application developer should create an asynchronous task in the `successBlock` to render pages as bitmaps using the `bitmapAtSize` method of `MuPDFDKDoc` (the document instance) against the document's `MuPDFDKPage` page instances. See the *iOS Sample app* and the `ConvertToPNGSample` folder reference for a full example.

5.5 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*.

5.5.1 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. The `MuPDFDKBasicDocumentViewController` session can be used to save the document with a callback method for the completion.

Swift

```
self.session.saveDocumentAnd(onCompletion: { (result:ARDKSaveResult,
                                    error:ARError) in
    self.updateUI() // refresh the UI on completion

    switch result {

        case ARDKSave_Succeeded:

        break

        case ARDKSave_Cancelled:

        break

        case ARDKSave_Error:

        break

        default:
```

(continues on next page)

(continued from previous page)

```
        break
    }
})
```

Objective-C

```
[self.session saveDocumentAndOnCompletion:^(ARDKSaveResult result,
                                            ARError err) {
    [self updateUI]; // refresh the UI on completion

    switch (result)
    {
        case ARDKSave_Succeeded:

        break;

        case ARDKSave_Cancelled:

        break;

        case ARDKSave_Error:

        break;

        default:

        break;
    }
}];
```

5.5.2 Save As

An application developer should provide the new filename and use the saveTo method of the `MuPDFDKBasicDocumentViewController` session to save documents with a new name.

Swift

```
let fileName:String = "new-filename.pdf"
let existingPath:NSString = session.fileState.absoluteInternalPath as NSString
let newPath:String = "\(existingPath.deletingLastPathComponent)/\(fileName)"

session.save(to: newPath, completion: { (result:ARDKSaveResult,
                                         error:ARError) in
    self.updateUI() // refresh the UI on completion

    switch result {

        case ARDKSave_Succeeded:

        break
```

(continues on next page)

(continued from previous page)

```
        case ARDKSave_Cancelled:

        break

        case ARDKSave_Error:

        break

        default:

        break
    }
})
```

Objective-C

```
NSString *fileName = @"new-filename.pdf";
NSString *newPath = [[self.session.fileState.absoluteInternalPath
    stringByDeletingLastPathComponent] stringByAppendingPathComponent:fileName];
[self.session saveTo:newPath completion:^(ARDKSaveResult result, ARError err) {
    switch (result)
    {
        case ARDKSave_Succeeded:
            break;

        case ARDKSave_Cancelled:
        case ARDKSave_Error:
            NSLog(@"Save failed with error: %d", err);
            break;
    }
}];
```

5.5.3 Export

It is possible to export the content of a PDF into an external text file for simple text extraction. To do so an application developer should define a valid file path to use with the `exportToAPI` against the instance of `MuPDFDKDoc` within the session.

Swift

```
let doc:MuPDFDKDoc = docVc.session.doc as! MuPDFDKDoc

doc.export(to: "docPath",
       format: "txt",
   completion: { (result:ARDKSaveResult) 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
    }
})
```

Objective-C

```
MuPDFDKDoc *doc = (MuPDFDKDoc*)docVc.session.doc;

[doc exportTo:@"docPath" format:@"txt" completion:^(ARDKSaveResult result) {
    switch (result)
    {
        case ARDKSave_Succeeded:
            break;

        case ARDKSave_Cancelled:
        case ARDKSave_Error:

            break;
    }
}];
```

Note: At present the only valid format to export to is a text file, so the format parameter should always be set to "txt". Further formats will become available later.

5.5.4 Print

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

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

printController.present(animated: true,
    completionHandler: { (printInteractionController:UIPrintInteractionController,
```

(continues on next page)

(continued from previous page)

```
        completed:Bool,
        error:Error?) in
    if error != nil {
        print("Print failed due to error:\(error!)")
    }
})
```

Objective-C

```
UIPrintInteractionController *printController =
[UIPrintInteractionController sharedPrintController];
ARDKPrintPageRenderer *pageRenderer =
[[ARDKPrintPageRenderer alloc] initWithDocument:self.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);
    }
}];
```

5.5.5 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.

Swift

```
let doc:MuPDFDKDoc = self.docViewController.session.doc as! MuPDFDKDoc

// 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
```

(continues on next page)

(continued from previous page)

```
            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

            case MuPDFDKSearch_Cancelled:
                self.updateUI()
            break

            case MuPDFDKSearch_Error:
                self.updateUI()
            break

            default:
                self.updateUI()
            break

        }
})
```

Objective-C

```
MuPDFDKDoc *doc = (MuPDFDKDoc*)docVc.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;
```

(continues on next page)

(continued from previous page)

```
        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;
    }
}];
```

5.5.6 FileState

The *FileState* protocol 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 protocol is detailed as follows:

```
@class ARDKDocSession;

/// Information about a file being opened by the SDK
///
/// The SDK user should provide an implementation of this interface.
/// An example minimal implementation is provided in the sample app.
@protocol ARDKFileState <NSObject>

/// The path to the document to display and edit
///
/// If ARDKSecureFS is in use (i.e. documents are not being stored directly to the device
/// filesystem unencrypted, this would normally be a path that for which ARDKSecureFS_
↪isSecure
/// will return 'true'.
///
/// If ARDKSecureFS is not in use, this should be a path to a file on the device␣
↪filesystem.
@property(readonly) NSString * _Nonnull absoluteInternalPath;

/// The file type
@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
```

(continues on next page)

(continued from previous page)

```
/// 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 * _Nonnull 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).
///
/// If a FileState with a non-null viewStateInfo is passed to
/// viewControllerForSessionRestoreLastViewingState then the
/// SDK will attempt to restore the file to show the same part
/// of the document that the user was viewing when they
/// previously opened the file.
///
/// The document view will write to this before sessionDidClose is called. A class
/// implementing the ARDKFileState interface can store this
/// value (using its NSCoding interface) against the file name
/// and then arrange to restore it should the same file be
/// reopened.
@property(nullable, retain) NSObject<NSCoding> *viewingStateInfo;

/// Information method called when a session has loaded the
/// first page of the document
- (void)sessionDidLoadFirstPage:(ARDKDocSession *_Nonnull)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 *_Nonnull)session;

/// Information method called when a session saves document
/// edits back to the supplied file
- (void)sessionDidSaveDoc:(ARDKDocSession *_Nonnull)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 (^_Nonnull)(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;

@end
```

5.6 Custom UI

5.6.1 Document API

Delegates

When developing your own custom UI, your application's `ViewController` should, at a minimum, implement the *ARDKBasicDocViewDelegate* and *ARDKDocumentEventTarget* protocols to intercept events.

ARDKBasicDocViewDelegate

Document Completed

Called when the document has completely loaded.

Swift

```
/// Inform the UI that both the loading of the document
/// and the initial rendering have completed. An app might
/// display a busy indicator while a document is initially
/// loading, and use this delegate method to dismiss the
/// indicator.
func loadingAndFirstRenderComplete() {

}
```

Objective-C

```
/// Inform the UI that both the loading of the document
/// and the initial rendering have completed. An app might
/// display a busy indicator while a document is initially
/// loading, and use this delegate method to dismiss the
/// indicator.
```

(continues on next page)

(continued from previous page)

```
- (void)loadingAndFirstRenderComplete {

}
```

UI Update

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

Swift

```
/// Tell the UI to update according to changes in the
/// selection state of the document. This allows an app
/// to refresh any currently displayed state information.
/// E.g., a button used to toggle whether the currently
/// selected text is bold, may show a highlight to indicate
/// bold or not. This call would be the appropriate place to
/// ensure that highlight reflects the current state.
func updateUI() {

}
```

Objective-C

```
/// Tell the UI to update according to changes in the
/// selection state of the document. This allows an app
/// to refresh any currently displayed state information.
/// E.g., a button used to toggle whether the currently
/// selected text is bold, may show a highlight to indicate
/// bold or not. This call would be the appropriate place to
/// ensure that highlight reflects the current state.
- (void)updateUI {

}
```

Detecting Page Movement

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

Swift

```
/// Tell the UI that the view of the document has moved. This
/// may be called very frequently if the document is scrolling,
/// and can be used to keep UI elements positioned next to items
/// within the document, such as the current selection.
func viewDidMove() {

}
```

Objective-C

```
/// Tell the UI that the view of the document has moved. This
/// may be called very frequently if the document is scrolling,
/// and can be used to keep UI elements positioned next to items
/// within the document, such as the current selection.
- (void)viewDidMove {

}
```

Page Change

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

Swift

```
/// Tell the UI that the document has scrolled to a new page.
/// An app may use this to update a label showing the current
/// displayed page number or to scroll a view of thumbnails
/// to the correct page.
func viewDidScroll(toPage page: Int) {

}
```

Objective-C

```
/// Tell the UI that the document has scrolled to a new page.
/// An app may use this to update a label showing the current
/// displayed page number or to scroll a view of thumbnails
/// to the correct page.
- (void)viewDidScrollToPage:(NSInteger)page {

}
```

Scrolling Completed

Called when a user scrolling event has concluded it's animation.

Swift

```
/// Tell the delegate when a scrolling animation concludes.
/// This can be used like viewDidScrollToPage, but for more
/// intensive tasks that one wouldn't want to run repeatedly
/// during scrolling.
func scrollViewDidEndScrollingAnimation() {

}
```

Objective-C

```
/// Tell the delegate when a scrolling animation concludes.
/// This can be used like viewDidScrollToPage, but for more
/// intensive tasks that one wouldn't want to run repeatedly
```

(continues on next page)

(continued from previous page)

```
/// during scrolling.
- (void)scrollViewDidEndScrollingAnimation {

}
```

Swallowing Tap Selections

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 false. Essentially a document is in a read only state in this mode.

Swift

```
/// Offer the UI the opportunity to swallow a tap that
/// may have otherwise caused selection. Return YES
/// to swallow the event. This is not called for taps
/// over links or form fields. An app might use this to
/// provide a way out of a special mode (full-screen for
/// example). In that case, if the app is using the tap to
/// provoke exit from full-screen mode, then it would return
/// YES from this method to avoid the tap being interpreted
/// also by the main document view.
func swallowSelectionTap() -> Bool {
    return false
}

/// Offer the UI the opportunity to swallow a double tap that
/// may have otherwise caused selection. Return YES to swallow
/// the event. This is not called for double taps over links
/// or form fields. An app might use this in a way similar to
/// that appropriate to swallowSelectionTap.
func swallowSelectionDoubleTap() -> Bool {
    return false
}
```

Objective-C

```
/// Offer the UI the opportunity to swallow a tap that
/// may have otherwise caused selection. Return YES
/// to swallow the event. This is not called for taps
/// over links or form fields. An app might use this to
/// provide a way out of a special mode (full-screen for
/// example). In that case, if the app is using the tap to
/// provoke exit from full-screen mode, then it would return
/// YES from this method to avoid the tap being interpreted
/// also by the main document view.
- (BOOL)swallowSelectionTap {
    return NO;
}

/// Offer the UI the opportunity to swallow a double tap that
```

(continues on next page)

(continued from previous page)

```
/// may have otherwise caused selection. Return YES to swallow
/// the event. This is not called for double taps over links
/// or form fields. An app might use this in a way similar to
/// that appropriate to swallowSelectionTap.
- (BOOL)swallowSelectionDoubleTap {
    return NO;
}
```

Note: It is important that these delegate methods return `true` by default for expected text and annotation selection behaviour to occur.

Inhibiting the Keyboard

If required an application developer can prevent the keyboard from appearing.

Swift

```
/// Called to allow the delegate to inhibit the keyboard. An app
/// might use this in special modes where there is limited
/// vertical space, so as to avoid the keyboard appearing.
func inhibitKeyBoard() -> Bool {
    return false
}
```

Objective-C

```
/// Called to allow the delegate to inhibit the keyboard. An app
/// might use this in special modes where there is limited
/// vertical space, so as to avoid the keyboard appearing.
- (BOOL)inhibitKeyBoard {
    return NO;
}
```

Opening a URL

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

Swift

```
/// The document view calls this when
/// a link to an external document is tapped.
func callOpenUrlHandler(_ url: URL!, fromVC presentingView: UIViewController!) {

}
```

Objective-C

```
/// The document view calls this when
/// a link to an external document is tapped.
```

(continues on next page)

(continued from previous page)

```
- (void)callOpenUrlHandler:(NSURL *)url fromVC:(UIViewController *)presentingView {

}
```

ARDKDocumentEventTarget

Page Load Events and Loading Complete

Called as pages are loaded from the document.

Swift

```
/// Called as pages are loaded from the document.
/// There may be further calls, e.g., if pages
/// are added or deleted from the document.
func updatePageCount(_ pageCount: Int, andLoadingComplete complete: Bool) {

}
```

Objective-C

```
/// Called as pages are loaded from the document.
/// There may be further calls, e.g., if pages
/// are added or deleted from the document.
- (void)updatePageCount:(NSInteger)pageCount andLoadingComplete:(BOOL)complete {

}
```

Called when layout has completed.

Swift

```
func layoutHasCompleted() {

}
```

Objective-C

```
- (void)layoutHasCompleted {

}
```

Called when the page size changes.

Swift

```
func pageSizeHasChanged() {

}
```

Objective-C

```
- (void)pageSizeHasChanged {

}
```

Selection Changes

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

Swift

```
func selectionHasChanged() {

}

// there is also a function method which can be
// associated against the instance of `MuPDFDKDoc` as follows
doc.onSelectionChanged = {

}
```

Objective-C

```
- (void)selectionHasChanged {

}

// there is also a function method which can be
// associated against the instance of `MuPDFDKDoc` as follows
doc.onSelectionChanged = ^() {

};
```

Selection Types

Once a selection has been made on a document it might be necessary to understand what type of selection it is and if there is any further data. For example, is the user's selection a redaction annotation or is it a note annotation? Does the selected annotation have a date associated with it?

To determine this information, the following type of query against the document instance (`MuPDFDKDoc`) can be made:

Swift

```
let selectionIsRedaction:Bool = doc.selectionIsRedaction
let selectionIsNote:Bool = doc.selectionIsAnnotationWithText
let selectedAnnotationsDate:Date? = doc.selectedAnnotationsDate
let haveTextSelection:Bool = doc.haveTextSelection
```

Objective-C

```
BOOL selectionIsRedaction = doc.selectionIsRedaction;
BOOL selectionIsNote = doc.selectionIsAnnotationWithText;
```

(continues on next page)

(continued from previous page)

```
NSDate* selectedAnnotationsDate = doc.selectedAnnotationsDate;
BOOL haveTextSelection = doc.haveTextSelection;
```

The following table defines the full set of available selections:

ARDKPageSelectorDelegate

Select Page

Called when the document page selector selects a page.

Swift

```
func selectPage(_ page: Int) {

}
```

Objective-C

```
-(void) selectPage:(NSInteger)page {

}
```

Delete Page

Called when the document page selector deletes a page.

Swift

```
func deletePage(_ page: Int) {

}
```

Objective-C

```
-(void) deletePage:(NSInteger)page {

}
```

Duplicate Page

Called when the document page selector duplicates a page.

Swift

```
func duplicatePage(_ page: Int) {

}
```

Objective-C

```
-(void) duplicatePage:(NSInteger)page {

}
```

Move Page

Called when the document page selector moves a page.

Swift

```
func movePage(_ page: Int, to newPos: Int) {

}
```

Objective-C

```
- (void)movePage:(NSInteger)pageNumber to:(NSInteger)newNumber {

}
```

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 `MuPDFDKDoc`).

Swift

```
doc.successBlock = {

}

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

}
```

Objective-C

```
doc.successBlock = ^() {

};

doc.errorBlock = ^(ARDKDocErrorType error) {

};
```

Request Password

For documents which may be password protected a developer should set a method against the document session to be triggered on the event of a password protected file attempting to load.

Swift

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

}
```

Objective-C

```
session.passwordRequestBlock = ^{

};
```

Typically the `passwordRequestBlock` implementation should offer a way which allows the user to input a document password. When your implementation is ready the `providePassword` API call should be made. If the password is correct then the document will display, if incorrect then your application should handle the failure.

Swift

```
doc.providePassword("password")
```

Objective-C

```
[doc providePassword:@"password"];
```

Document actions

Undo

To undo a previous action (such as adding an annotation) use the the following method against your document instance.

Swift

```
if doc.canUndo {
    doc.undo()
}
```

Objective-C

```
if (doc.canUndo) {
    [doc undo];
}
```

Redo

To redo a previous action (such as adding an annotation) use the the following method against your document instance.

Swift

```
if doc.canRedo {
    doc.redo()
}
```

Objective-C

```
if (doc.canRedo) {
    [doc redo];
}
```

Get Selected Text

If the document has selected text (i.e. PDF document text which the user has selected) then this text can be read by your application with the selectedText method.

Swift

```
doc.selectedText({ text in
    print("text=\(text)")
})
```

Objective-C

```
[doc selectedText:^(NSString *text) {
    NSLog(@"text=%@",text);
}];
```

Note: Typically an application will want to copy the text value to the pasteboard.

5.6.2 PDF Annotations

Annotations

Annotations includes functionality for *drawing*, *text markup* and *placing objects* on a *PDF* document.

App Kit provides the following sets:

- ***Document Drawing***
  - Freehand ink drawing
  - Line
  - Rectangle
  - Ellipse

  - Polygon
  - Polyline
- *Document Text*
  - Highlight
  - Underline
  - Squiggle
  - Strikethrough
- *Document Placement*
  - Note
  - Textbox
  - Stamp
  - Attachment
  - Link

Note: You should not set annotation modes on a document until the document has loaded, therefore ensure to add any API calls for annotation methods after *listening for the document loaded* event.

Document Drawing Annotations

Document text annotations depend on a "Drawing Annotation Mode" being activated. The following API allows for control over setting this mode.

Setting the Drawing Annotation Mode

This is as simple as setting the annotating mode to a given "Drawing Annotation Mode" type on your `MuPDFDKBasicDocumentViewController` instance. To turn off the mode set the annotating mode to `MuPDFDKAnnotatingMode_None`.

Drawing mode enumeration

Mode	Description
`MuPDFDKAnnotatingMode_Ink`	Freehand ink
`MuPDFDKAnnotatingMode_Line`	Line
`MuPDFDKAnnotatingMode_Square`	Squares & Rectangles
`MuPDFDKAnnotatingMode_Circle`	Circles & Ovals
`MuPDFDKAnnotatingMode_Polygon`	Polygon
`MuPDFDKAnnotatingMode_PolyLine`	Polyline

The following code toggles the freehand ink drawing mode on and off and might serve as the handler for one of the buttons in your UI.

Swift

```
if basicDocVc.annotatingMode == MuPDFDKAnnotatingMode_Ink {
    basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_None
} else {
    basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_Ink
}
```

Objective-C

```
if (basicDocVc.annotatingMode == MuPDFDKAnnotatingMode_Ink) {
    basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_None;
}
else {
    basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_Ink;
}
```

When a "Drawing Annotation Mode" is disabled, the annotation is saved to the document.

Styling Drawing Annotations

Drawing annotations can be styled as follows:

Opacity

To set & get annotation opacity, access the `annotationOpacity` value as a `Float`.

Swift

```
basicDocVc.annotationOpacity = 1.0
```

Objective-C

```
basicDocVc.annotationOpacity = 1.0;
```

Line Thickness

To set & get line thickness, access the `annotationStrokeThickness` value as a `CGFloat`.

Swift

```
basicDocVc.annotationStrokeThickness = 10
```

Objective-C

```
basicDocVc.annotationStrokeThickness = 10;
```

Line Color

To set & get line color, access the `annotationStrokeColor` value as a `UIColor`.

Swift

```
basicDocVc.annotationStrokeColor = .green
```

Objective-C

```
basicDocVc.annotationStrokeColor = [UIColor green];
```

Line Endings

Line endings are only supported for the `MuPDFDKAnnotatingMode_Line` annotation. They can be set & get by accessing the `annotationLineHeadStyle` & `annotationLineTailStyle` properties.

Swift

```
basicDocVc.annotationLineHeadStyle = ARDKLineEndStyle_Circle
basicDocVc.annotationLineTailStyle = ARDKLineEndStyle_ClosedArrow
```

Objective-C

```
basicDocVc.annotationLineHeadStyle = ARDKLineEndStyle_Circle;
basicDocVc.annotationLineTailStyle = ARDKLineEndStyle_ClosedArrow;
```

Line ending enumeration

Style	Description
ARDKLineEndStyle_None	None
ARDKLineEndStyle_Butt	Butt
ARDKLineEndStyle_Slash	Slash
ARDKLineEndStyle_Circle	Cirle
ARDKLineEndStyle_Diamond	Diamond
ARDKLineEndStyle_OpenArrow	Open arrow
ARDKLineEndStyle_ROpenArrow	Right pointing open arrow
ARDKLineEndStyle_Square	Square
ARDKLineEndStyle_ClosedArrow	Closed arrow
ARDKLineEndStyle_RClosedArrow	Right pointing closed arrow

Document Text Annotations

Document text annotations depend on a "Text Annotation Mode" being activated. The following API allows for control over setting this mode.

Setting the Text Annotation Mode

This is as simple as setting the annotating mode to a given "Text Annotation Mode" type on your `MuPDFDKBasicDocumentViewController` instance. To turn off the mode set the annotating mode to `MuPDFDKAnnotatingMode_None`.

Text mode enumeration

mode	Description
`MuPDFDKAnnotatingMode_HighlightTextSelect`	Highlight text
`MuPDFDKAnnotatingMode_UnderlineTextSelect`	Underline text
`MuPDFDKAnnotatingMode_SquiggleTextSelect`	Squiggle text
`MuPDFDKAnnotatingMode_StrikethroughTextSelect`	Strikethrough text

When a "Text Annotation Mode" is active, the user can select text and apply the text annotations accordingly.

Document Placement Annotations

Document text annotations depend on a "Placement Annotation Mode" being activated. The following API allows for control over setting this mode.

Setting the Placement Annotation Mode

This is as simple as setting the annotating mode to a given "Placement Annotation Mode" type on your `MuPDFDKBasicDocumentViewController` instance. To turn off the mode set the annotating mode to `MuPDFDKAnnotatingMode_None`.

Placement mode enumeration

mode	Description
`MuPDFDKAnnotatingMode_Note`	Note
`MuPDFDKAnnotatingMode_FreeText`	Free text
`MuPDFDKAnnotatingMode_Stamp`	Stamp
`MuPDFDKAnnotatingMode_FileAttachment`	File attachment
`MuPDFDKAnnotatingMode_Link`	Link

When a "Placement Annotation Mode" is active, the user can tap on the document to create the placement annotation. In-built UI will then handle the anootation creation and manipulation.

Getting the Annotation Mode

Access the `annotatingMode` variable on your `MuPDFDKBasicDocumentViewController` instance to find out the current mode.

Swift

```
let mode:MuPDFDKAnnotatingMode = documentViewController.annotatingMode
```

Objective-C

```
MuPDFDKAnnotatingMode mode = self.docViewController.annotatingMode;
```

De-selecting Annotations

Essentially all annotations are treated the same as simply selections once they have been selected by a user. This selection can then be cleared (de-selected) by calling the `clearSelection()` method against the `MuPDFDKDoc` instance within `MuPDFDKBasicDocumentViewController`.

Swift

```
let myDoc:MuPDFDKDoc? = basicDocVc?.session.doc as! MuPDFDKDoc
myDoc?.clearSelection()
```

Objective-C

```
MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
[myDoc clearSelection];
```

Without an annotation being currently selected, calling `clearSelection()` will have no effect.

Deleting Annotations

Selections can be removed (deleted) by calling the `deleteSelectedAnnotation()` method against the `MuPDFDKDoc` instance within `MuPDFDKBasicDocumentViewController`.

Swift

```
func deleteDocumentSelection() {
    let myDoc:MuPDFDKDoc? = basicDocVc?.session.doc as! MuPDFDKDoc
    myDoc?.deleteSelectedAnnotation()
}
```

Objective-C

```
-(void)deleteDocumentSelection {
    MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
    [myDoc deleteSelectedAnnotation];
}
```

Without an annotation being currently selected, calling `deleteSelectedAnnotation()` will have no effect.

Author

To set the author name for subsequent annotation creations, an application developer should set the `documentAuthor` property to the required string value within the `MuPDFDKDoc` instance.

Swift

```
func setAuthor(name:String) {
    let myDoc:MuPDFDKDoc? = basicDocVc?.session.doc as! MuPDFDKDoc
    myDoc?.documentAuthor = name
}
```

Objective-C

```
-(void)setAuthor:(NSString *)name {
    MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
    myDoc.documentAuthor = name;
}
```

Targetting Document Areas

Sometimes in order to focus an annotation, or to suggest an area to annotate, a developer may wish to highlight areas in a document and/or jump to a specific area of interest in a document. There are two API calls available to assist with this.

Outlining an Area

To draw outlines on a document page use `outlineArea`.

Swift

```
basicDocVc.session.doc.outlineArea(<rect>, onPage: <page>)
```

Objective-C

```
[basicDocVc.session.doc outlineArea:<rect> onPage:<page>];
```

Showing an Area

To show an area on a document page use `showArea`.

Swift

```
basicDocVc.showArea(<rect>, onPage: <page>)
```

Objective-C

```
[basicDocVc.showArea:<rect> onPage:<page>];
```

5.6.3 PDF Redactions

Redactions

The redaction feature has been designed to work on text, images, and links. By selecting all or part of an image, text, or link, and applying redaction this then permanently redacts the selected information, making it impossible to retrieve the original data.

Redacted text and areas can be *marked*, and any marked redactions can be *removed*, however once the redactions are *applied* and the document is saved then the redacted information is blocked out with black to denote the area of redaction. At this point the original text and/or area is unretrievable.

Redaction Modes

Your custom UI can support redaction by employing three specific annotating modes.

Edit Redaction Mode

Firstly set the annotation mode for your document view to `MuPDFDKAnnotatingMode_EditRedaction`. This ensures that only redaction markings can be selected. When selected they can be adjusted or removed.

Swift

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_EditRedaction
```

Objective-C

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_EditRedaction;
```

After adding a redaction it is advised to return your `MuPDFDKBasicDocumentViewController` back to this edit mode as default.

Text Redaction Mode

Set the annotation mode for your document to `MuPDFDKAnnotatingMode_RedactionTextSelect`. This prepares the document view to mark text for redaction.

Swift

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_RedactionTextSelect
```

Objective-C

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_RedactionTextSelect;
```

In this mode, the user can create a redaction marking by dragging across text, whereupon the mode will revert to `MuPDFDKAnnotatingMode_EditRedaction`, leaving the newly created marking of the text selected. The selection will show drag handles via which the user can adjust it.

Area Redaction Mode

Set the annotation mode for your document to `MuPDFDKAnnotatingMode_RedactionAreaSelect`. This prepares the document view to mark an area for redaction.

Swift

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_RedactionAreaSelect
```

Objective-C

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_RedactionAreaSelect;
```

In this mode, the user can create a redaction marking by dragging out an area from one corner to the diagonally opposite one, whereupon the mode will revert to `MuPDFDKAnnotatingMode_EditRedaction`, leaving the newly created marking of an area selected. The selection will show drag handles via which the user can adjust it.

Marking Already Selected Text

There is an alternative way to mark text for redaction. If the user has already selected an area of text, it is possible to mark that text for redaction by calling the `addRedactAnnotation()` method against your `MuPDFDKDoc` instance. The document's currently selected text will then be marked with a red box outline to denote the redaction demarcation.

Swift

```
let myDoc:MuPDFDKDoc = basicDocVc.session.doc as! MuPDFDKDoc
myDoc.addRedactAnnotation()
```

Objective-C

```
MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
[myDoc addRedactAnnotation];
```

De-selecting a Redaction

A selected redaction can then be cleared (de-selected) by calling the `clearSelection()` method against the `MuPDFDKDoc` instance within `MuPDFDKBasicDocumentViewController`.

Swift

```
let myDoc:MuPDFDKDoc = basicDocVc.session.doc as! MuPDFDKDoc
myDoc.clearSelection()
```

Objective-C

```
MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
[myDoc clearSelection];
```

Removing a Marked Redaction

To remove a marked redaction call the `deleteSelectedAnnotation()` method against your `MuPDFDKDoc` instance. The document's currently selected marked for redaction (i.e. a selection of text with the red box outline) will then no longer be marked for redaction.

Swift

```
let myDoc:MuPDFDKDoc = basicDocVc.session.doc as! MuPDFDKDoc
myDoc.deleteSelectedAnnotation()
```

Objective-C

```
MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
[myDoc deleteSelectedAnnotation];
```

Applying Redactions

Applying redactions means that all document selections which are marked for redaction will be redacted. To apply redactions call the `finalizeRedactAnnotations()` method against your `MuPDFDKDoc`z instance. Once redactions have been applied then the redacted text is blocked out with black.

Swift

```
let myDoc:MuPDFDKDoc = basicDocVc.session.doc as! MuPDFDKDoc
myDoc.finalizeRedactAnnotations({
    /// onComplete
})
```

Objective-C

```
MuPDFDKDoc *myDoc = (MuPDFDKDoc *)basicDocVc.session.doc;
[myDoc finalizeRedactAnnotations:^{
    /// onComplete
}];
```

Note: Redactions are not permanently set until the document is saved. If the document is exited without saving then the redactions will be lost.

5.6.4 PDF Signatures

Signatures

There are two distinct signature types which can be used with App Kit.

- *Digital Signatures*
- *E-signatures*

Digital signatures can be placed as signing areas in documents and also signed, whereas E-signatures can just be placed onto a document as part of a signing activity. E-signatures are the simplest use case and just involve a hand drawn signature without digital ID certification.

Digital Signatures

Placing a Digital Signature

Just like regular annotations, the annotating mode within the document's view controller needs to be set.

To turn on digital signature mode set the annotating mode to `MuPDFDKAnnotatingMode_DigitalSignature` within your `MuPDFDKBasicDocumentViewController` instance. To exit this mode then set the annotating mode to `MuPDFDKAnnotatingMode_None`.

When in digital signature mode if the user taps the document then a digital signature area will be placed at that point, it can be resized and moved to the desired position, but once the document is saved then the position is set and the digital signature field becomes uneditable.

Swift

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_DigitalSignature
```

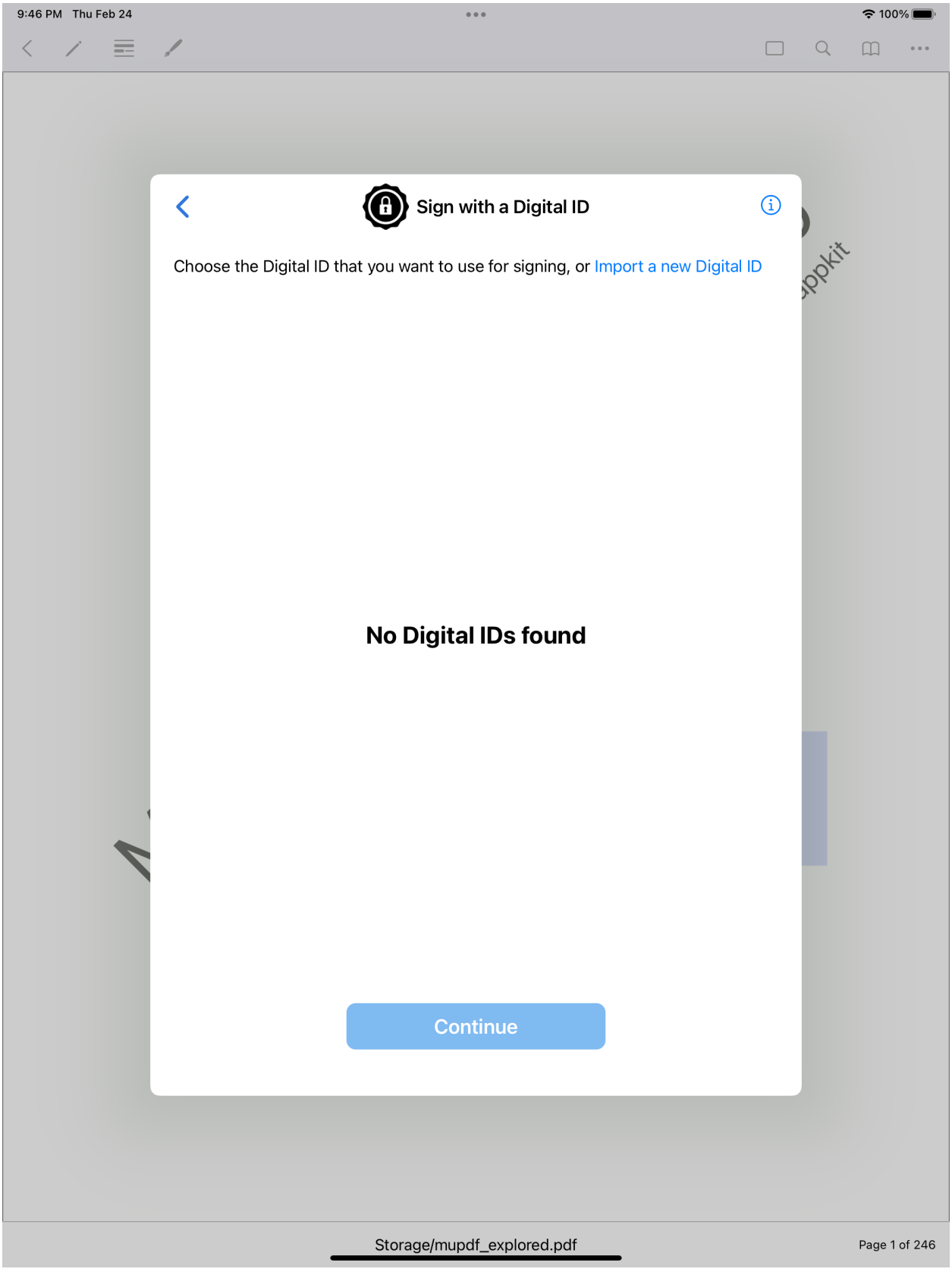
Objective-C

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_DigitalSignature;
```

Signing a Digital Signature

Digital signatures use certificate-based digital IDs to authenticate signer identity and demonstrate proof of signing by binding each signature to the document with encryption.

If there is a set digital signature area on your document then tapping that will invoke App Kit to display a pre-built UI for the signing activity. This is a carefully considered UI and will not allow you to continue until you have imported a valid digital ID.



The UI for digital signing

E-signatures

E-signatures should have been created (i.e. drawn by the user) before they are placed and there is some handy drop-in UI built into App Kit which allows for this. On your document view controller instance you can invoke the following segue to make a dialog appear:

Swift

```
self.performSegue(withIdentifier: "esignature-place-edit", sender: nil)
```

Objective-C

```
[self performSegueWithIdentifier:@"esignature-place-edit" sender:nil];
```

The UI for creating your e-signature

The E-signatures Drop-In UI

To pick up key events from the e-signatures drop-in UI, your document view controller code should listen to the unwind seques from the e-signature storyboard and act accordingly.

Swift

```
@IBAction func unwindForEditESignature(_ unwindSegue:UIStoryboardSegue) {
    /// User asked to edit(draw) the e-signature
    let segue = unwindSegue as! ARDKSegueWithCompletion
    segue.completion = {[weak self] in
        self?.performSegue(withIdentifier:"signature-draw", sender: nil)
    }
}

@IBAction func unwindForPlaceESignature(_ unwindSegue:UIStoryboardSegue) {
    /// User asked to place an e-signature
    if documentViewController.eSigImage != nil {
        documentViewController.annotatingMode = MuPDFDKAnnotatingMode_ESignature
    }
}

@IBAction func unwindForCancel(_ unwindSegue:UIStoryboardSegue) {
    /// User chose cancel
}

@IBAction func signatureDrawnUnwindAction(_ unwindSegue:UIStoryboardSegue) {
    /// set the e-signature image on the document view controller
    let vc = unwindSegue.source as! ARDKSignaturesDrawViewController
    documentViewController.eSigImage = vc.image
}

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

Objective-C

```
- (IBAction)unwindForEditESignature:(UIStoryboardSegue *)unwindSegue
{
    ARDKSegueWithCompletion *segue = (ARDKSegueWithCompletion *)unwindSegue;
    __weak typeof(self) weakSelf = self;
    segue.completion = ^{
        [weakSelf performSegueWithIdentifier:@"signature-draw" sender:nil];
    };
}

- (IBAction)unwindForPlaceESignature:(UIStoryboardSegue *)unwindSegue
{
    if (documentViewController.eSigImage != nil) {
        documentViewController.annotatingMode = MuPDFDKAnnotatingMode_ESignature;
    }
}

- (IBAction)unwindForCancel:(UIStoryboardSegue *)unwindSegue
```

(continues on next page)

(continued from previous page)

```
{
}

- (IBAction)signatureDrawnUnwindAction:(UIStoryboardSegue *)unwindSegue
{
    ARDKSignaturesDrawViewController *vc = (ARDKSignaturesDrawViewController␣
↪*)unwindSegue.sourceViewController;
    documentViewController.eSigImage = vc.image;
}

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

Setting an E-signature Image

To set the image used for an e-signature then access the `eSigImage` property within your `MuPDFDKBasicDocumentViewController` instance to a valid `UIImage`.

Swift

```
documentViewController.eSigImage = UIImage
```

Objective-C

```
documentViewController.eSigImage = UIImage;
```

Placing an E-signature

To turn on e-signature mode set the annotating mode to `MuPDFDKAnnotatingMode_ESignature` within your `MuPDFDKBasicDocumentViewController` instance. To exit this mode then set the annotating mode to `MuPDFDKAnnotatingMode_None`.

When in e-signature mode if the user taps the document then an e-signature area will be placed at that point, it can be resized and moved to the desired position. Unlike digital signatures, an e-signature will always be able to be edited even after a document has been saved.

Swift

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_ESignature
```

Objective-C

```
basicDocVc.annotatingMode = MuPDFDKAnnotatingMode_ESignature;
```

Querying Signature Modes

To query for signature mode an application developer should simply query the annotation mode of the document view controller. If the mode returns `MuPDFDKAnnotatingMode_DigitalSignature` or `MuPDFDKAnnotatingMode_ESignature` then there is currently a signature mode in progress.

Searching for signatures

If you have a document with multiple digital signatures you can query the document to search through and focus those signatures.

Searching for signatures involves invoking the `findSigStart` method with a designated callback which is responsible for keeping track of the page number and annotation identifier as results come through from the document. Thus, specific class member variables which keep track of these variables must be designated before the signature searching starts. Once a signature is found the callback mechanism should be responsible for *focusing the outline* of the signature as well as moving the document view controller to the location of the signature.

To start the search use the following code with `MuPDFDKSearchDirection` being set to either `MuPDFDKSearch_Forwards` or `MuPDFDKSearch_Backwards`:

Swift

```
var sigSearchPage:Int = -1
var sigSearchAnnot:Int = -1
...

func searchForSignature(direction:MuPDFDKSearchDirection) {
    var cancelled:ObjCBool = false
    doc.findSigStart(atPage: sigSearchPage,
                     andAnnot: sigSearchAnnot,
                     in: direction,
                     cancelFlag: &amp;cancelled,
                     onEvent: { event, page, annot, rect in
                         switch event {
                         case MuPDFDKSearch_Found:
                             self.sigSearchPage = page
                             self.sigSearchAnnot = annot
                             self.documentViewController.session.doc.outlineArea(rect,␣
↪onPage: page)
                             self.documentViewController.showArea(rect, onPage: page)

                         case MuPDFDKSearch_Progress:
                         case MuPDFDKSearch_Cancelled:
                         default:
                             ()
                         }
                     })
}
```

Objective-C

```
NSInteger sigSearchPage = -1;
NSInteger sigSearchAnnot = -1;
...
```

(continues on next page)

(continued from previous page)

```
- (void)searchForSignature:(MuPDFDKSearchDirection)direction
{
    BOOL cancelled;

    [doc findSigStartAtPage:sigSearchPage
                  andAnnot:sigSearchAnnot
               inDirection:direction
                cancelFlag:&amp;cancelled
                   onEvent:^(MuPDFDKSearchEvent event, NSInteger page, NSInteger annot,
↪ CGRect rect) {
            switch (event)
            {
                case MuPDFDKSearch_Found:
                    self.sigSearchPage = page;
                    self.sigSearchAnnot = annot;
                    [self.documentViewController.session.doc outlineArea:rect␣
↪onPage:page];
                    [self.documentViewController showArea:rect onPage:page];

                case MuPDFDKSearch_Progress:
                case MuPDFDKSearch_Cancelled:
                default:
                    break;
            }
        }
    ];
}
```

Note: Searching for signatures only looks for digital signature fields. If there are e-signatures placed on the document these will be ignored.

5.6.5 PDF Table of Contents

Overview

Not all PDF documents will have a Table of Contents, but for those that do there will be a non-nil array associated against the `MuPDFDKDoc` document instance as follows:

Swift

```
let toc:[ARDKTocEntry]? = session.doc.toc
```

Objective-C

```
NSArray<id<ARDKTocEntry>> *toc = session.doc.toc;
```

Note: If there is no Table of Contents for the PDF then the `toc` array will simply be `nil`.

Table of Contents is also known as "Bookmarks" or "Outline".

To understand each `ARDKTocEntry` entry object inside the delivered array, an application developer can loop the array to strip out the entries into a flat array for listing purposes. This is because `ARDKTocEntry` items can contain sub-nested `ARDKTocEntry` items (items at lower depths representing a table of contents sub-listing from a parent item). An example of how to traverse the `toc` array is as follows:

Swift

```swift
func traverse(toc:[ARDKTocEntry]) {
    var tocEntries:[ARDKTocEntry] = []

    func add(entries:[ARDKTocEntry]) {
        for entry:ARDKTocEntry in entries {
            tocEntries.append(entry)
            if entry.children != nil {
                add(entries:entry.children! as! [ARDKTocEntry])
            }
        }
    }

    add(entries:toc)

    /// Items should now be in a flat array, check their label and depth
    for item:ARDKTocEntry in tocEntries {
        print("item.label=\(item.label)")
        print("item.depth=\(item.depth)")
    }
}
```

Objective-C

```objc
NSMutableArray<id<ARDKTocEntry>> *tocEntries = [[NSMutableArray alloc] init];

-(void)traverse:(NSArray<id<ARDKTocEntry>> *)toc {
    [self addEntries:toc];

    // Items should now be in a flat array, check their label and depth
    for (id<ARDKTocEntry> item in self.tocEntries) {
        NSLog(@"item.label=%@",item.label);
        NSLog(@"item.depth=%lu",item.depth);
    }
}

- (void)addEntries:(NSArray<id<ARDKTocEntry>> *)entries {
    for (id<ARDKTocEntry> entry in entries) {
        [self.tocEntries addObject:entry];
        if (entry.children)
            [self addEntries:entry.children];
    }
}
```