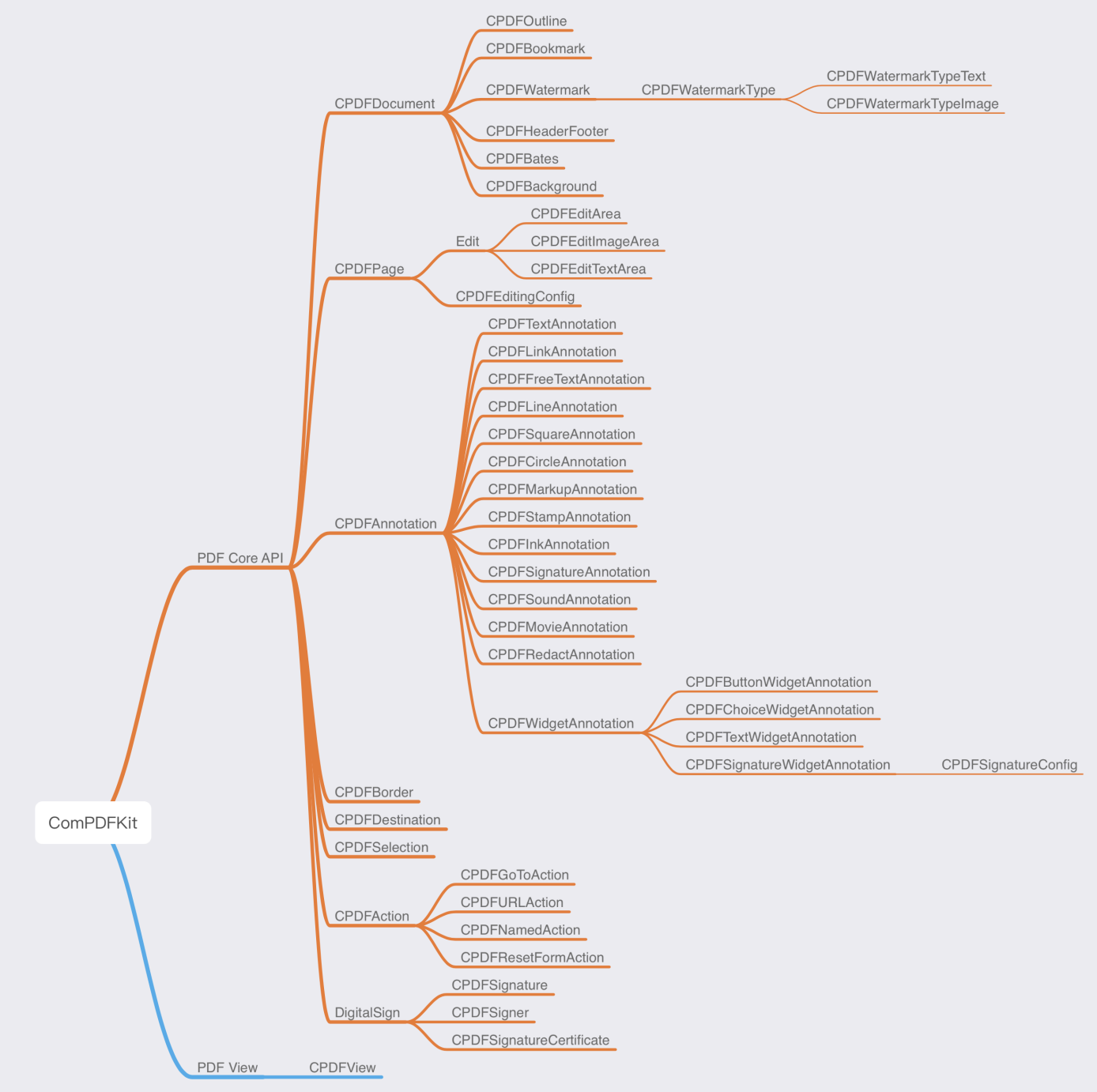
1 Overview

ComPDFKit PDF SDK for iOS is a robust PDF library for developers who need to develop applications on iOS, which offers powerful Swift or Objective-C APIs for quickly viewing, annotating, editing, and creating PDFs. It is feature-rich and battle-tested, making PDF files process and manipulation easier and faster for iOS devices.

1.1 ComPDFKit PDF SDK

ComPDFKit PDF SDK consists of two elements as shown in the following picture.



The two elements for ComPDFKit PDF SDK:

- **PDF Core API**

  The Core API can be used independently for document rendering, analysis, text extraction, text search, form filling, password security, annotation creation and manipulation, and much more.

- **PDF View**

  The PDF View is a utility class that provides the functionality for developers to interact with rendering PDF documents per their requirements. The View Control provides fast and high-quality rendering, zooming, scrolling, and page navigation features. The View Control is derived from platform-related viewer classes (e.g. `UIView` on iOS) and allows for extension to accommodate specific user needs.

1.2 Key Features

Viewer component offers:

- Standard page display modes, including Scrolling, Double Page, Crop Mode, and Cover Mode.
- Navigation with thumbnails, outlines, and bookmarks.
- Text search & selection.
- Zoom in and out & Fit-page.
- Switch between different themes, including Dark Mode, Sepia Mode, Reseda Mode, and Custom Color Mode.
- Text reflow.

Annotations component offers:

- Create, edit, and remove annotations, including Note, Link, Free Text, Line, Square, Circle, Highlight, Underline, Squiggly, Strikeout, Stamp, Ink, and Sound.
- Support for annotation appearances.
- Import and export annotations to/from XFDF.
- Support for annotation flattening.
- Predefine annotations.

Forms component offers:

- Create, edit and remove form fields, including Push Button, Check Box, Radio Button, Text Field, Combo Box, List Box, and Signature.
- Fill PDF Forms.
- Support for PDF form flattening.

Document Editor component offers:

- PDF manipulation, including Split pages, Extract pages, and Merge pages.
- Page edit, including Delete pages, Insert pages, Crop pages, Move pages, Rotate pages, Replace pages, and Exchange pages.
- Document information setting.

- Extract images.

Content Editor component offers:

- Programmatically add and remove text/images in PDFs and make it possible to edit PDFs like Word. Allow selecting text to copy, resize, change colors, text alignment, and the position of text boxes.
- Undo or redo any change.

Security component offers:

- Encrypt and decrypt PDFs, including Permission setting and Password protected.
- Create, edit, and remove watermark.
- Redact content including images, text, and vector graphics.
- Create, edit, and remove header & footer, including dates, page numbers, document name, author name, and chapter name.
- Create, edit, and remove bates numbers.
- Create, edit, and remove background that can be a solid color or an image.

Redaction component offers:

- Redact content including images, text, and vector graphics to remove sensitive information or private data, which cannot be restored once applied.
- Create redaction by selecting an area or searching for a specific text.
- Edit and save redaction annotations.

Watermark component offers:

- Add, remove, edit, update, and get the watermarks.
- Support text and image watermarks.

Conversion component offers:

- PDF to PDF/A.

Digital Signatures component offers:

- Sign PDF documents with digital signatures.
- Create and verify digital certificates.
- Create and verify digital digital signatures.
- Create self-sign digital ID and edit signature appearance.
- Support PKCS12 certificates.
- Trust certificates.

Measurement component offers:

- Measure the length of straight lines and polylines.
- Measure the perimeter and area of closed shapes.

Optimization component offers:

- Compress and Optimize PDF Files.

2 Get Started

It is easy to embed ComPDFKit in your iOS app with a few lines of Swift or Objective-C code. Take just a few minutes and get started.

The following sections introduce the structure of the installation package, how to run a demo, and how to make an iOS app in Swift or Objective-C with ComPDFKit PDF SDK.

2.1 Requirements

ComPDFKit requires the latest stable version of Xcode available at the time the release was made. This is a hard requirement, as each version of Xcode is bundled with a specific version of the iOS Base SDK, which often defines how UIKit and various other frameworks behave.

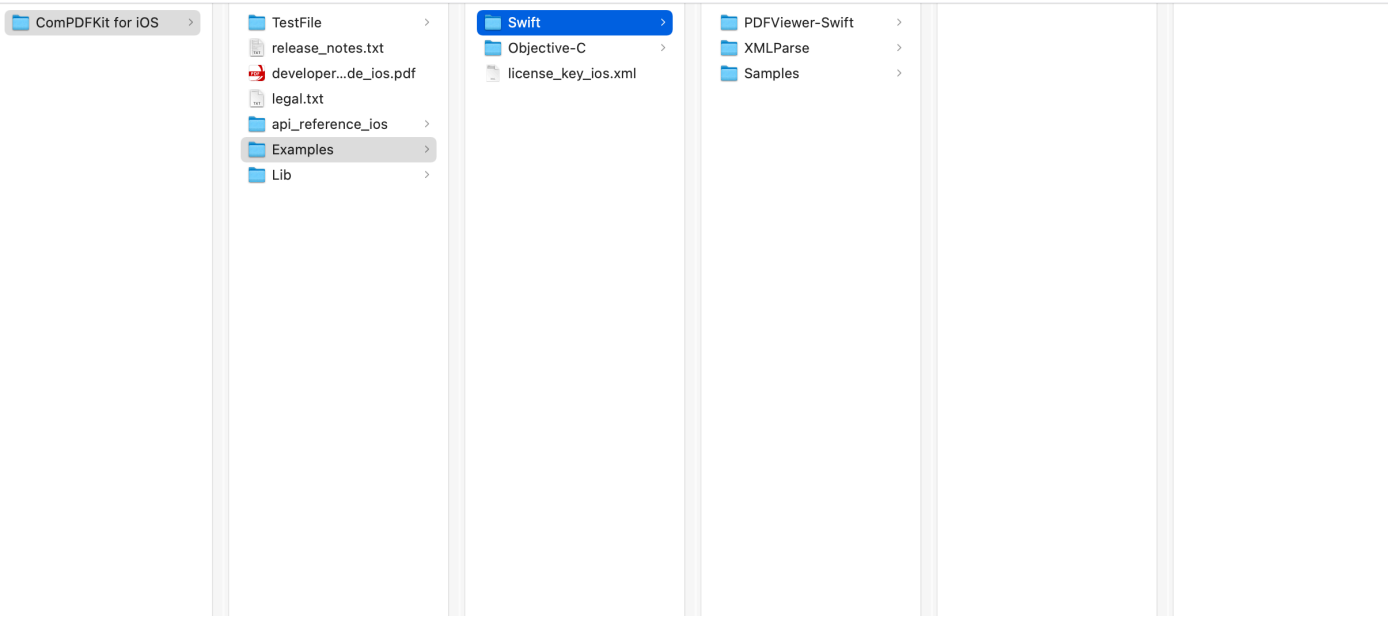
- iOS 10.0 or higher.
- Xcode 13.0 or newer for Swift or Objective-C .

2.2 iOS Package Structure

The package of ComPDFKit PDF SDK for iOS includes the following files:

- ***"Lib"*** - Include the ComPDFKit.xcframework dynamic library (arm64_armv7, x86_64-simulator,arm64_x86_64-maccatalyst) and associated header files.
    - ***"ComPDFKit_Tools"*** - A default control library for quickly building various function modules of PDF viewer.
    - ***"ComPDFKit.xcframework"*** - ComPDFKit.xcframework dynamic library (arm64_armv7, x86_64-simulator,arm64_x86_64-maccatalyst)
- ***"Examples"*** - A folder containing iOS sample projects.
    - ***Swift*** - Swift sample projects.
        - ***"PDFViewer-Swift"*** - A folder containing Swift iOS sample projects.
        - ***"Samples"*** - A folder containing console application.
    - ***Objective-C*** - Objective-C sample projects.
        - ***"Viewer"*** - A basic PDF viewer, including reading PDFs, changing themes, bookmarks, searching text, etc.
        - ***"Annotations"*** - A PDF viewer with full types of annotation editing, including adding annotations, modifying annotations, annotation lists, etc.
        - ***"ContentEditor"*** - A PDF viewer with text and image editing, including modifying text, replacing images, etc.

        - ***"Forms"*** - A PDF viewer with full types of forms editing, including radio button, combo box, etc.
        - ***"DocsEditor"*** - A PDF viewer with page editing, including inserting/deleting pages, extracting pages, reordering pages, etc.
        - ***"DigitalSignature"*** - A PDF viewer with digital signature functionality, including signing with a digital ID, creating a self-
          sign digital ID, signature authentication, viewing signature information, editing the signature appearance, etc.
        - ***"PDFViewer"*** - A multi-functional PDF program that integrates all of the above features.
        - ***"Samples"*** - A folder containing console application.
    - ***license_key_ios.xml*** - The license xml file needed to run the sample projects.
- ***"api_reference_ios"*** - API reference.
- ***"developer_guide_ios.pdf"*** - Developer guide.
- ***"release_notes.txt"*** - Release information.
- ***"legal.txt"*** - Legal and copyright information.
- ***"TestFile"*** - A folder containing test files.



2.3 Apply the License Key

ComPDFKit PDF SDK is a commercial SDK, which requires a license to grant developer permission to release their apps. Each license is only valid for one bundle ID in development mode. ComPDFKit supports flexible licensing options, please contact our marketing team to know more. However, any documents, sample code, or source code distribution from the released package of ComPDFKit to any third party is prohibited.

Online License: We have introduced an online license mechanism for enhanced license management. Through the online approach, you can manage and update your license more flexibly to meet the specific requirements of your project.

Offline License: In scenarios with high security requirements, no internet connectivity, or offline environments, we provide the option of an offline license. The offline license allows authorization and usage of the ComPDFKit PDF SDK when internet connectivity is not available.

2.3.1 Obtaining the License Key

If you intend to use the ComPDFKit license for your application, we offer two types of licenses: a trial license and a formal license. The formal license requires binding to the unique `Bundle ID` of your application. ComPDFKit provides two methods to obtain a license, and you can choose either based on your preferences.

Method One:

1. Initiate contact with our sales team by completing the requirements form on the Contact Sales page of the ComPDFKit official website.
2. After receiving your submission, our sales team will reach out to you within 24 hours to clarify your requirements.
3. Upon confirmation of your requirements, you will be issued a complimentary trial license valid for 30 days. Throughout this period, any issues you encounter will be supported with free technical assistance.
4. If you are satisfied with the product, you have the option to purchase the formal license. Once the transaction is completed, our sales team will send the official license to you via email.

Method Two (Mobile Platforms Only):

1. Log in to the ComPDFKit official website's Online Sales Interface to submit a trial application and receive an immediate free trial license for iOS platforms, valid for 30 days.
2. If content with the product, you can directly purchase the formal license on the Online Sales Interface of the official website.
3. When purchasing the formal license, ensure to bind the license to your application's `Bundle ID`. If uncertain about the `Bundle ID`, refer to How to Find the Bundle ID of Your App.
4. Following a successful payment, the system will automatically email the official license bound to your `Bundle ID`.
5. If any issues arise during the online transaction, you can submit your problems through the Technical Support page on the ComPDFKit official website. We will respond to your inquiries within 24 hours, providing free technical support services.

2.3.2 Copying the License Key

Accurately obtaining the license key is crucial for the application of the license.

1. In the email you received, locate the `XML` file containing the license key.
2. Open the XML file, and determine the license type based on the `<type>` field. If `<type>online</type>` is present, it indicates an online license. If `<type>offline</type>` is present or if the field is absent, it indicates an offline license.

   Online License:

```
xmlCopy code
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<license version="1">
    <platform>ios</platform>
    <starttime>xxxxxxxx</starttime>
    <endtime>xxxxxxxx</endtime>
    <type>online</type>
    <key>LICENSE_KEY</key>
</license>
```

Offline License:

```
xmlCopy code
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<license version="1">
    <platform>ios</platform>
    <starttime>xxxxxxxx</starttime>
    <endtime>xxxxxxxx</endtime>
    <key>LICENSE_KEY</key>
</license>
```

3. Copy the **LICENSE_KEY** from the `<key>LICENSE_KEY</key>` field.

2.3.3 Apply the License Key

You can contact the ComPDFKit team to obtain a trial license. Before using any ComPDFKit PDF SDK classes, you must perform the following steps to apply the license to your application:

1. In `AppDelegate.swift`, import the header file ***ComPDFKit***.
2. Depending on the type of authentication obtained in the previous step, whether online or offline, initialize the license using the respective method based on your requirements.
3. Initialize the license:

- **Online license:**

Follow the code below and call the method `CPDFKit.verify(withOnlineLicense: "LICENSE_KEY") { code, message in }` in `func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool`. You need to replace the **LICENSE_KEY** with the license you obtained.

```
  func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
  // Set your online license key here. ComPDFKit is commercial software.
  // Each ComPDFKit license is bound to a specific app bundle id.
  // com.compdfkit.pdfviewe

    CPDFKit.verify(withOnlineLicense: "YOUR_LICENSE_KEY_GOES_HERE") { code, message in
    }
}
```

```
#import <ComPDFKit/ComPDFKit.h>

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
  // Set your online license key here. ComPDFKit is commercial software.
  // Each ComPDFKit license is bound to a specific app bundle id.
  // com.compdfkit.pdfviewer

    [CPDFKit verifyLicense:@"YOUR_LICENSE_KEY_GOES_HERE"
completionHandler:^(CPDFKitOnlineLicenseCode code, NSString *message) {

    }];

    return YES;
}
```

- **Offline license:**

Follow the code below and call the method `CPDFKit.verifyWithKey:"LICENSE_SECRET"` in `func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool`. You need to replace the **LICENSE_KEY** with the license you obtained.

```
  func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
  // Set your license key here. ComPDFKit is commercial software.
  // Each ComPDFKit license is bound to a specific app bundle id.
  // com.compdfkit.pdfviewer

    CPDFKit.verify(withKey: "YOUR_LICENSE_KEY_GOES_HERE")
    return true
}
```

```
#import <ComPDFKit/ComPDFKit.h>

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
  // Set your local license key here. ComPDFKit is commercial software.
  // Each ComPDFKit license is bound to a specific app bundle id.
  // com.compdfkit.pdfviewer

    [CPDFKit verifyWithKey:@"YOUR_LICENSE_KEY_GOES_HERE"];

    return YES;
}
```

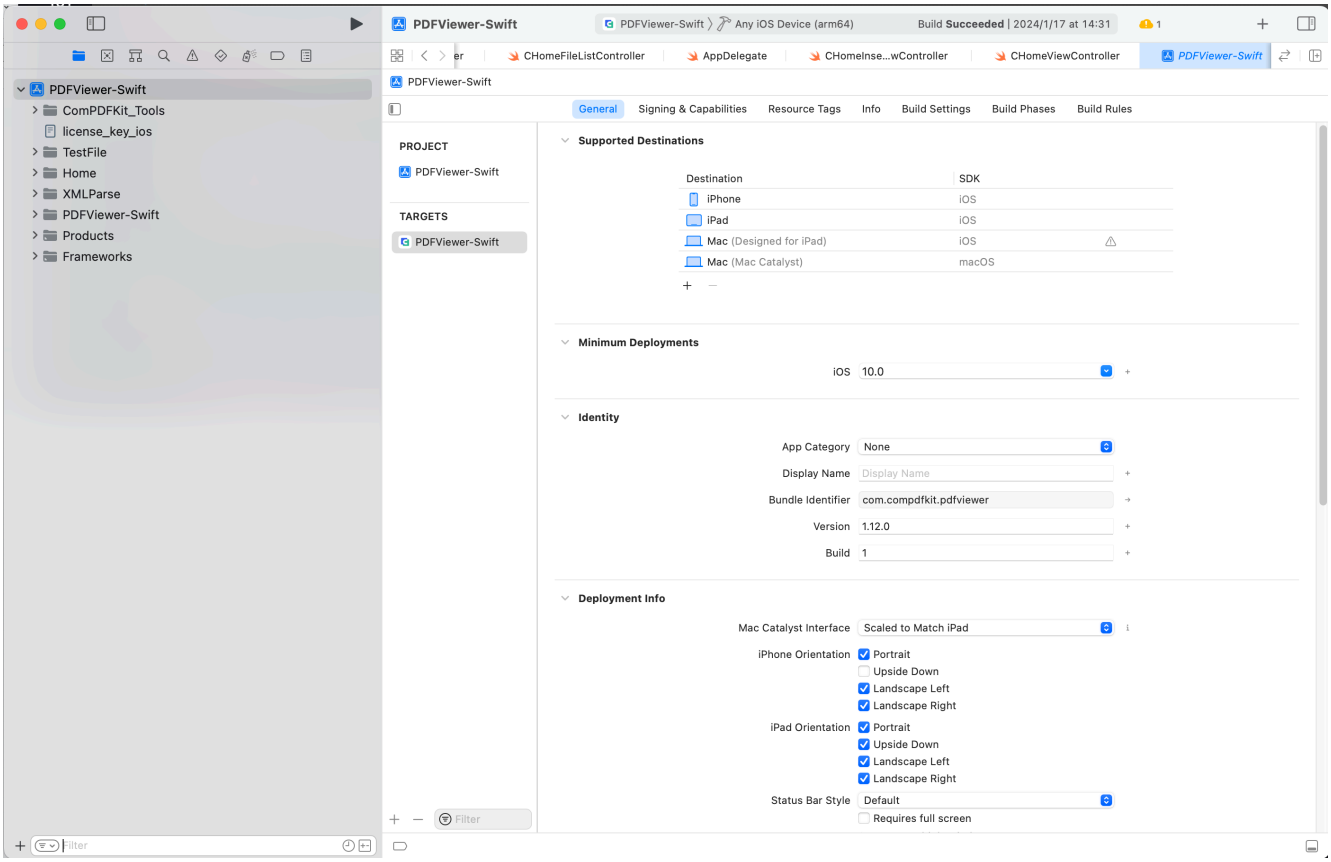
Compile and run the project. If the console outputs "version information", it means that the license has been set successfully. Otherwise, please check the "Troubleshooting" section at 2.5.6 or check error logs in the console to quickly identify and solve the issue.

2.4 How to Run a Demo

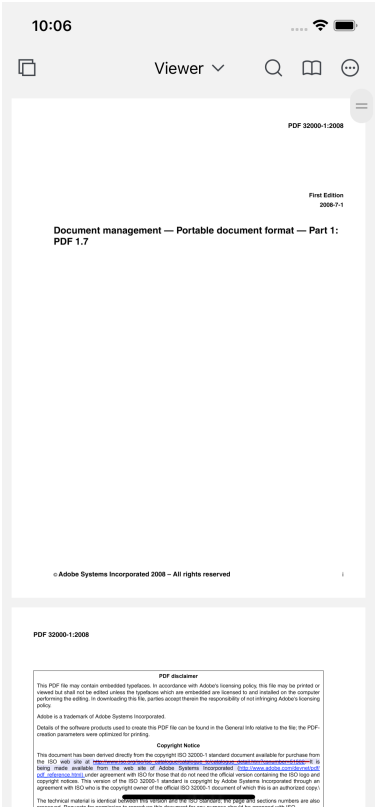
ComPDFKit PDF SDK for iOS provides multiple demos in Swift for developers to learn how to call the SDK on iOS. You can find them in the ***"Examples/Swift"*** folder.

In this guide, we take ***"PDFViewer-Swift"*** as an example to show how to run it in Xcode (The demo in Swift uses the ***"xcodeproj"*** method, so you can directly open ***"PDFViewer-Swift.xcodeproj"***).

1. Copy the ***"license_key_ios.xml"*** file, which will be used for local license or online license, into the ***"Examples"*** folder to replace (There is already a method to parse the xml file in demo, please do not modify the storage location and file name).
2. Find ***"PDFViewer-Swift.xcodeproj"*** in the ***"Examples/Swift"*** folder and double-click to open it, find the schemes of ***"PDFViewer-Swift"*** in Xcode, and select the corresponding simulator (ComPDFKit does not support the simulator to run M1 chip, but we have made it compatible in `Excluded Architectures`, you can see the processing method in 2.5.6 Troubleshooting).



3. Click **Product -> Run** to run the demo on an iOS device. In this guide, we use an iPhone 14 device as an example. After building the demo successfully, the ***"PDF32000_2008.pdf"*** file will be opened and displayed.



Note: *This is a demo project, presenting completed ComPDFKit PDF SDK functions. The functions might be different based on the license you have purchased. Please check that the functions you choose work fine in this demo project.*

2.5 How to Make an iOS App in Swift with ComPDFKit

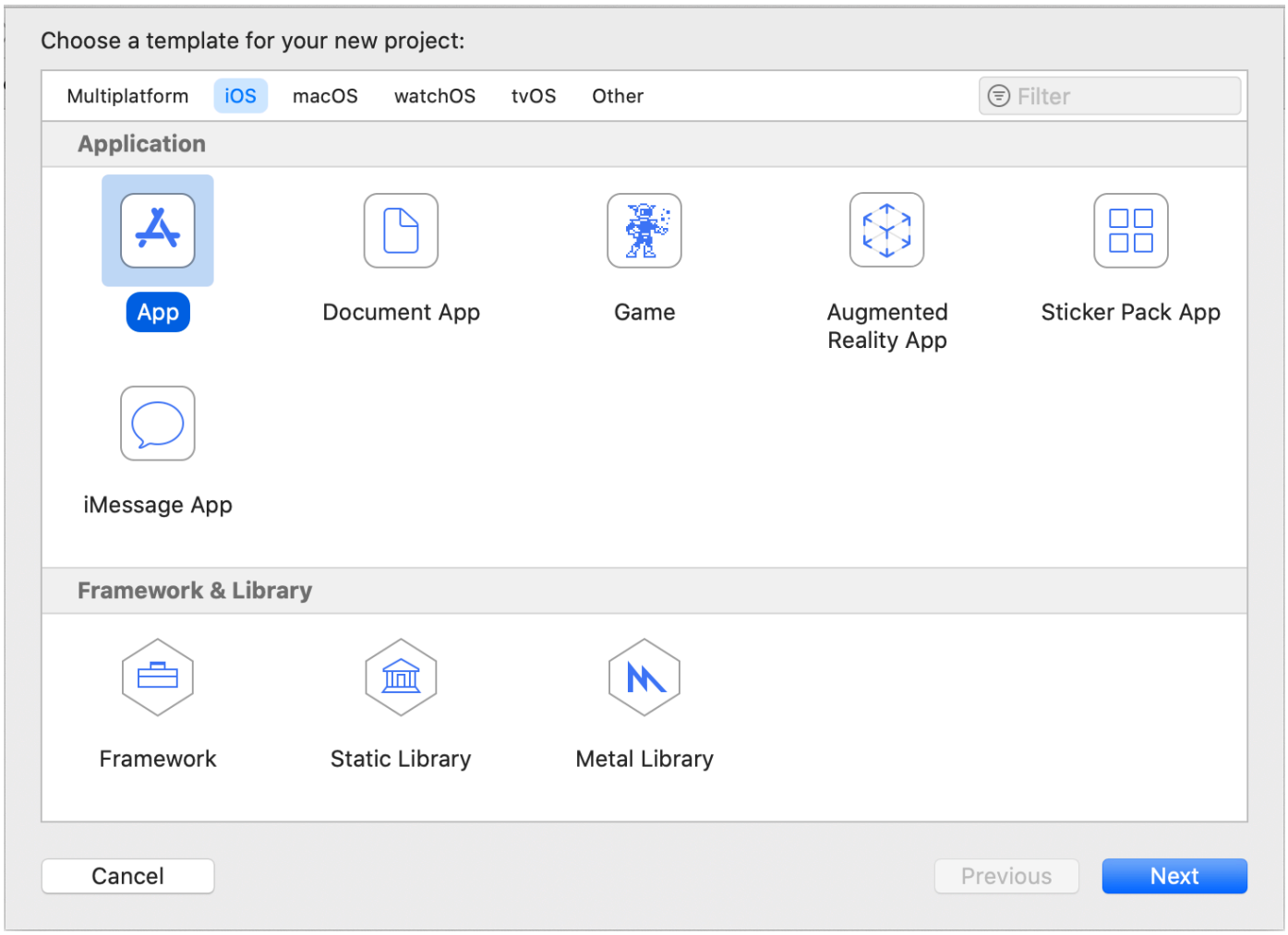
This section will help you to quickly get started with ComPDFKit PDF SDK to make an iOS app in Swift with step-by-step instructions, which include the following steps:

1. Create a new iOS project in Swift.
2. Integrate ComPDFKit into your apps.
3. Apply the license key.
4. Display a PDF document.

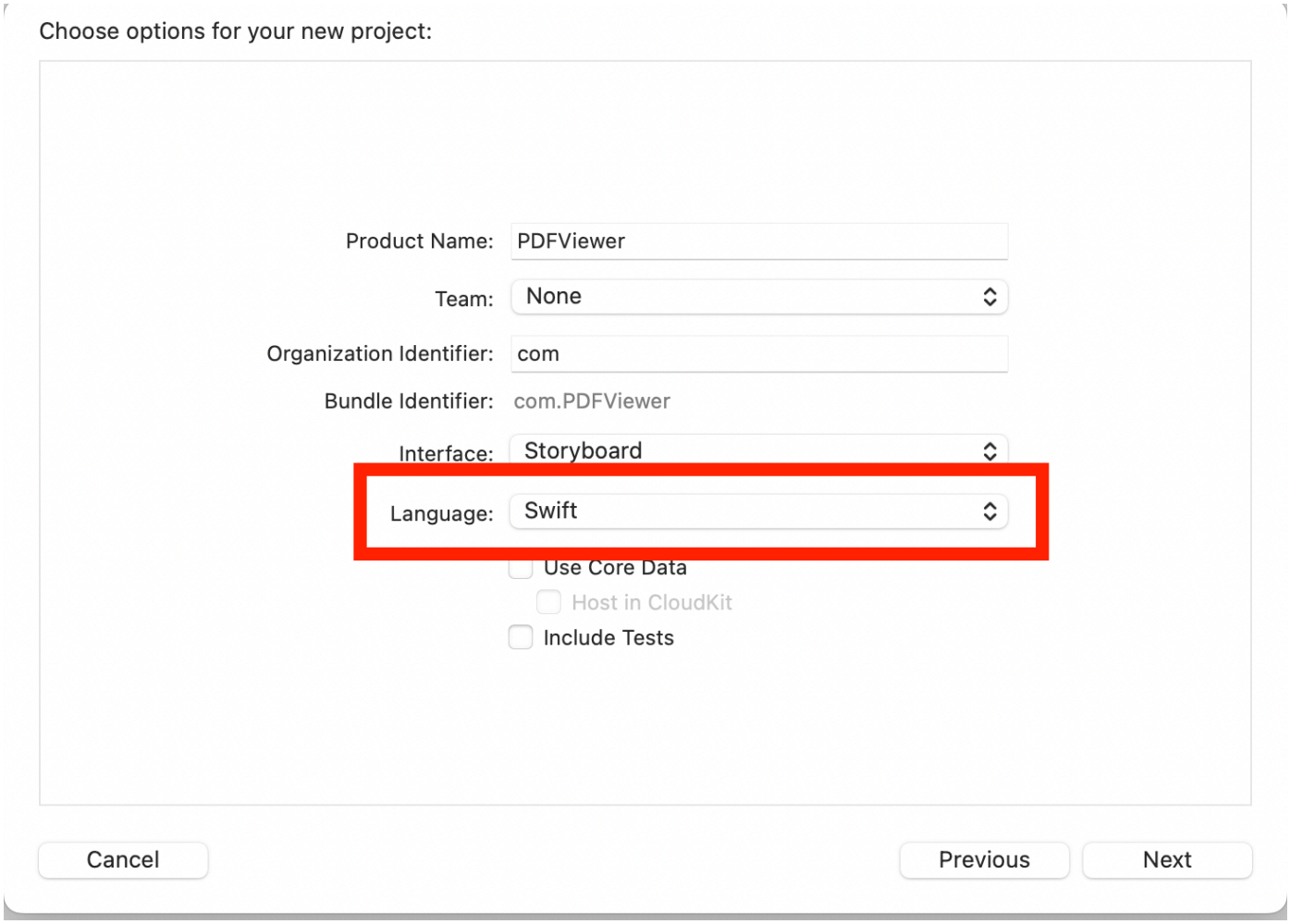
2.5.1 Create a New iOS Project in Swift

In this guide, we use Xcode 12.4 to create a new iOS project.

Fire up Xcode, choose **File** -> **New** -> **Project...**, and then select **iOS** -> **Single View Application**. Click **Next**.



Choose the options for your new project. Please make sure to choose Swift as the programming language. Then, click **Next**.



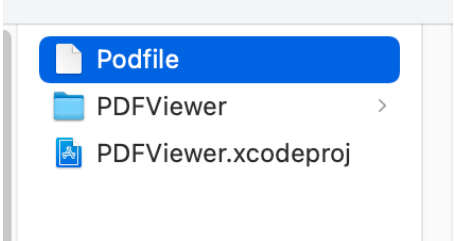
Place the project to the location as desired. Then, click **Create**.

2.5.2 Integrate ComPDFKit into Your Apps

There are two ways to integrate ComPDFKit PDF SDK for iOS into your apps. You can choose what works best for you based on your requirements.

Adding the ComPDFKit CocoaPods Dependency

1. Open the terminal and go to the directory containing your Xcode project: `cd .../PDFViewer`.
2. Run `pod init`. This will create a new Podfile next to your `.xcodeproj` file:



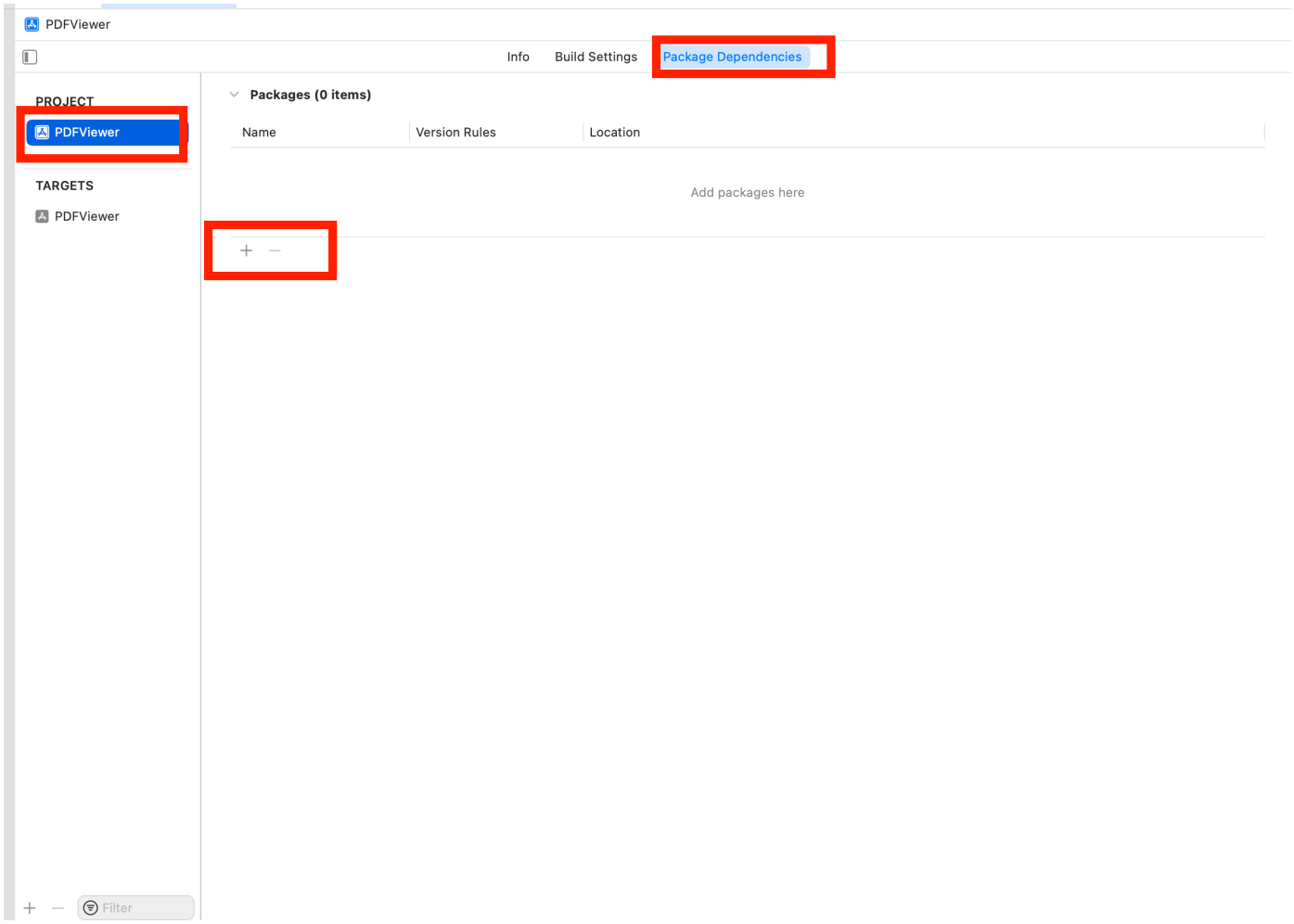
3. Open the newly created Podfile in a text editor and add the ComPDFKit pod URL:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'PDFViewer' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!

+  pod 'ComPDFKit', :git => 'https://github.com/ComPDFKit/compdfkit-pdf-sdk-ios-swift.git', :tag => '2.2.0'

  # Pods for PDFViewer

end
```

4. Run `pod install` and wait for CocoaPods to download ComPDFKit.
5. Open your application's newly created workspace (`PDFViewer.xcworkspace`) in Xcode.

Adding the ComPDFKit Swift Package Manager Dependency

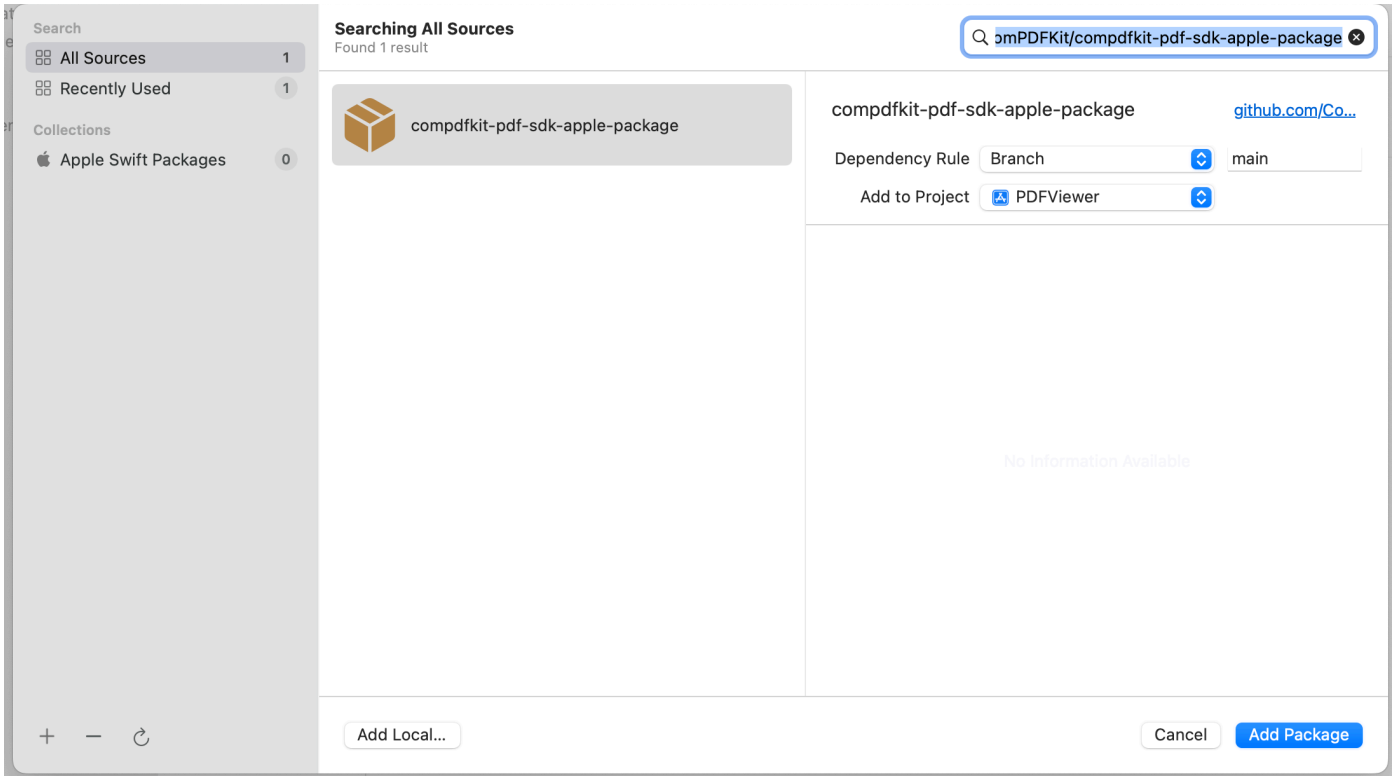
1.Open your application in Xcode and select your project's **Package Dependencies** tab.



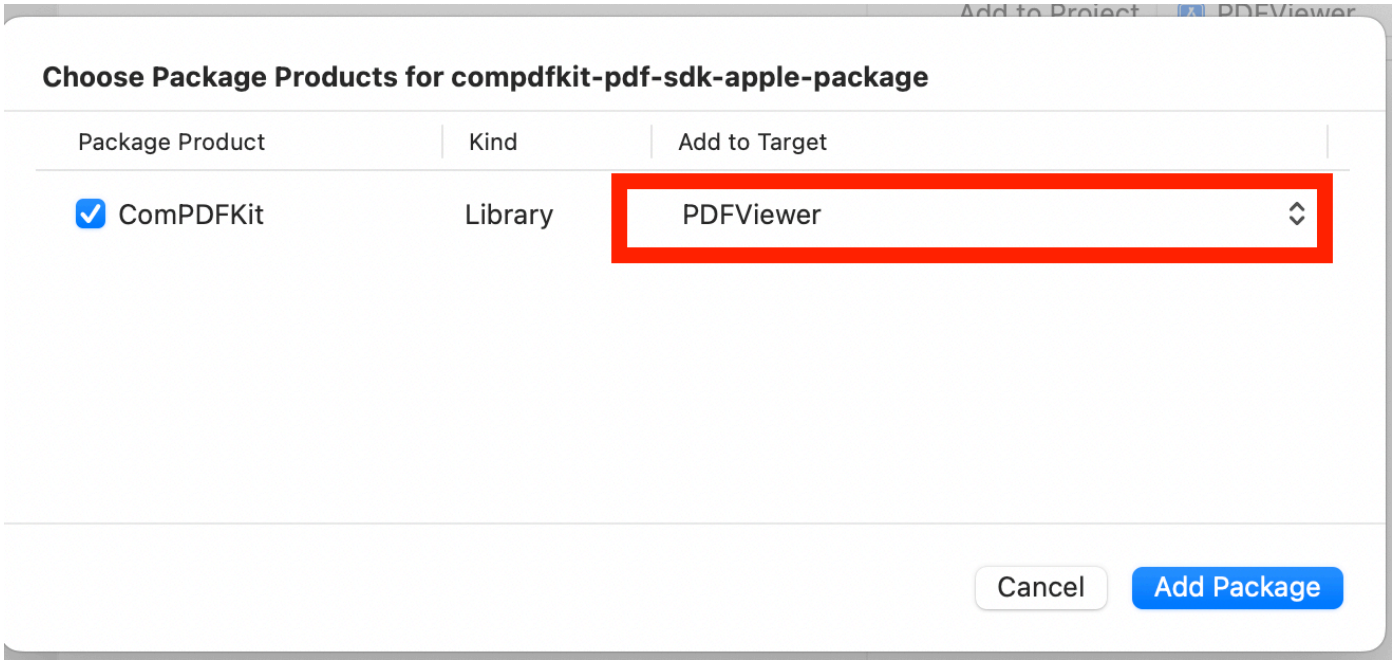
2.Copy the ComPDFKit Swift package repository URL into the search field:

```
https://github.com/ComPDFKit/compdfkit-pdf-sdk-apple-package
```

3.In the **Dependency Rule** fields, select **Branch** > **master**, and then click **Add Package**.



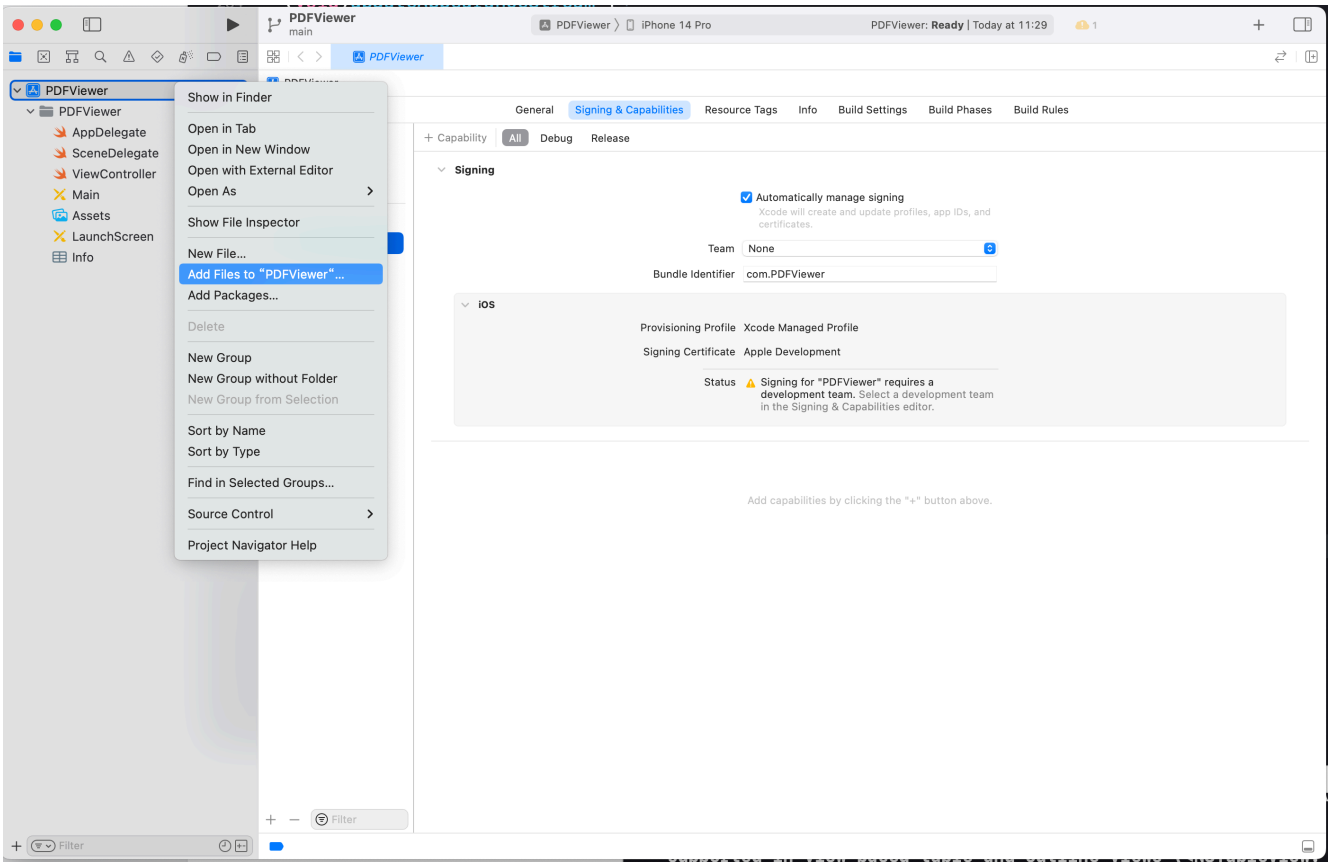
4.After the package download completes, select **Add Package**.



ComPDFKit should now be listed under Swift Package Dependencies in the Xcode Project navigator.

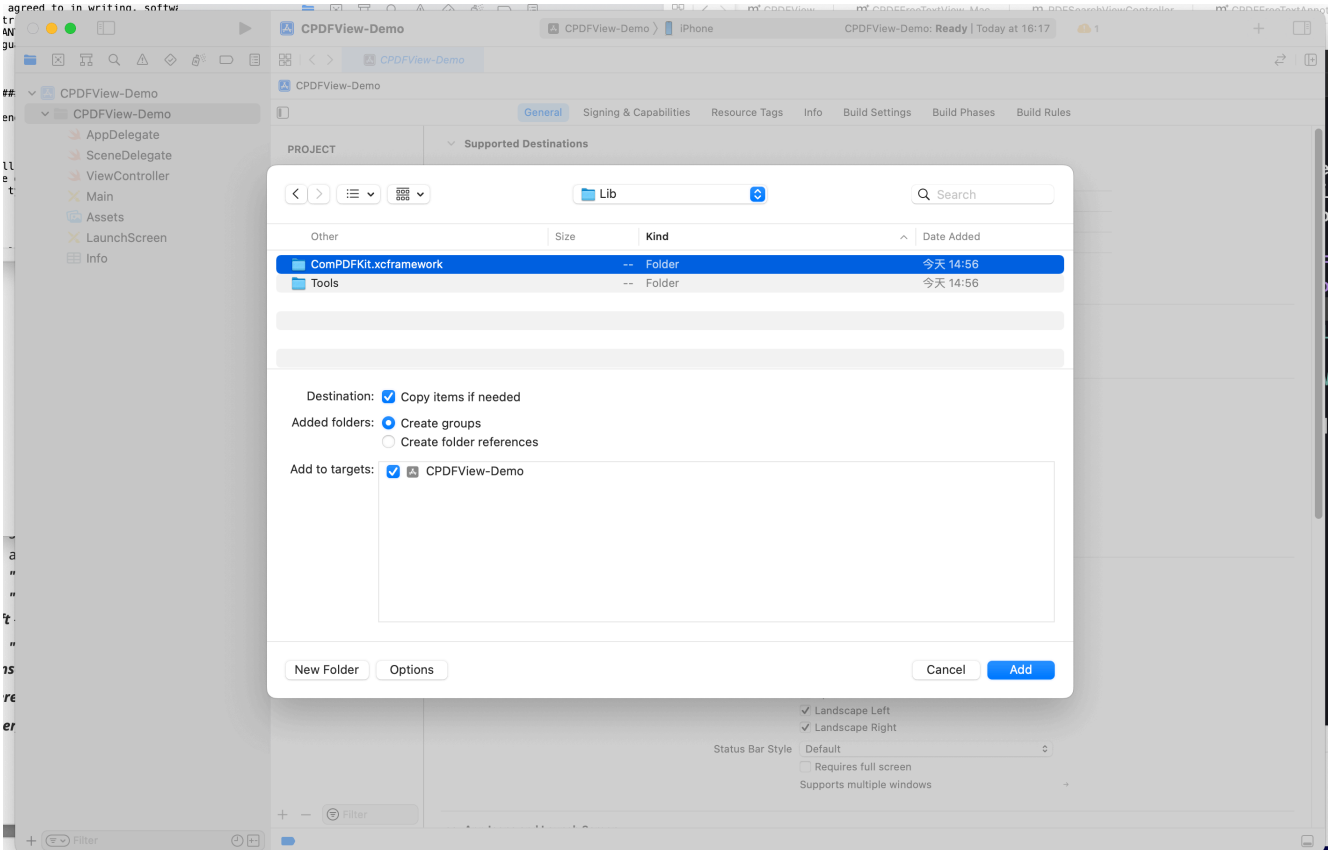
Adding the ComPDFKit XCFrameworks Manually

1. Right-click the ***"PDFViewer"*** project, select **Add Files to "PDFViewer"...**.

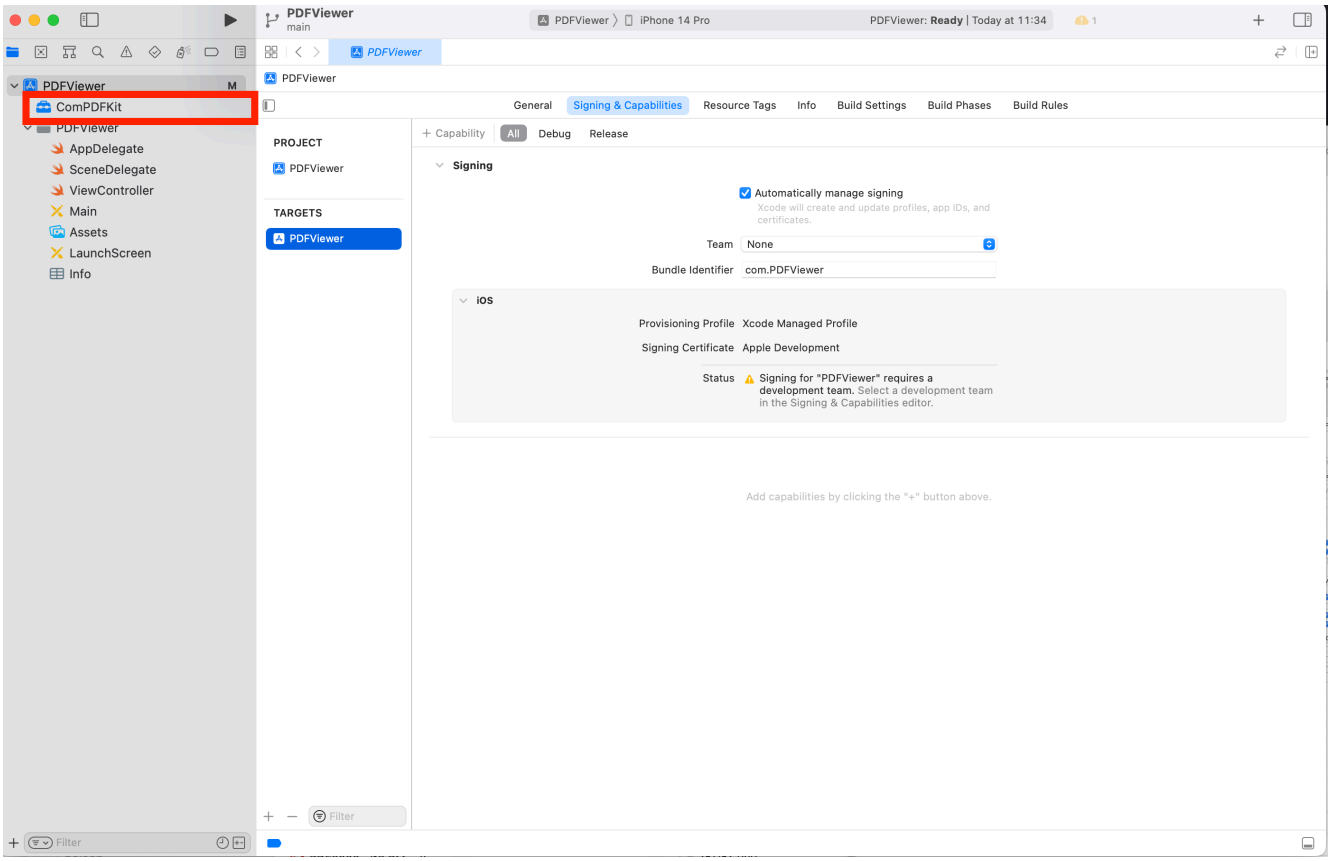


Find and choose ***"ComPDFKit.xcframework"*** in the download package, and then click **Add**.

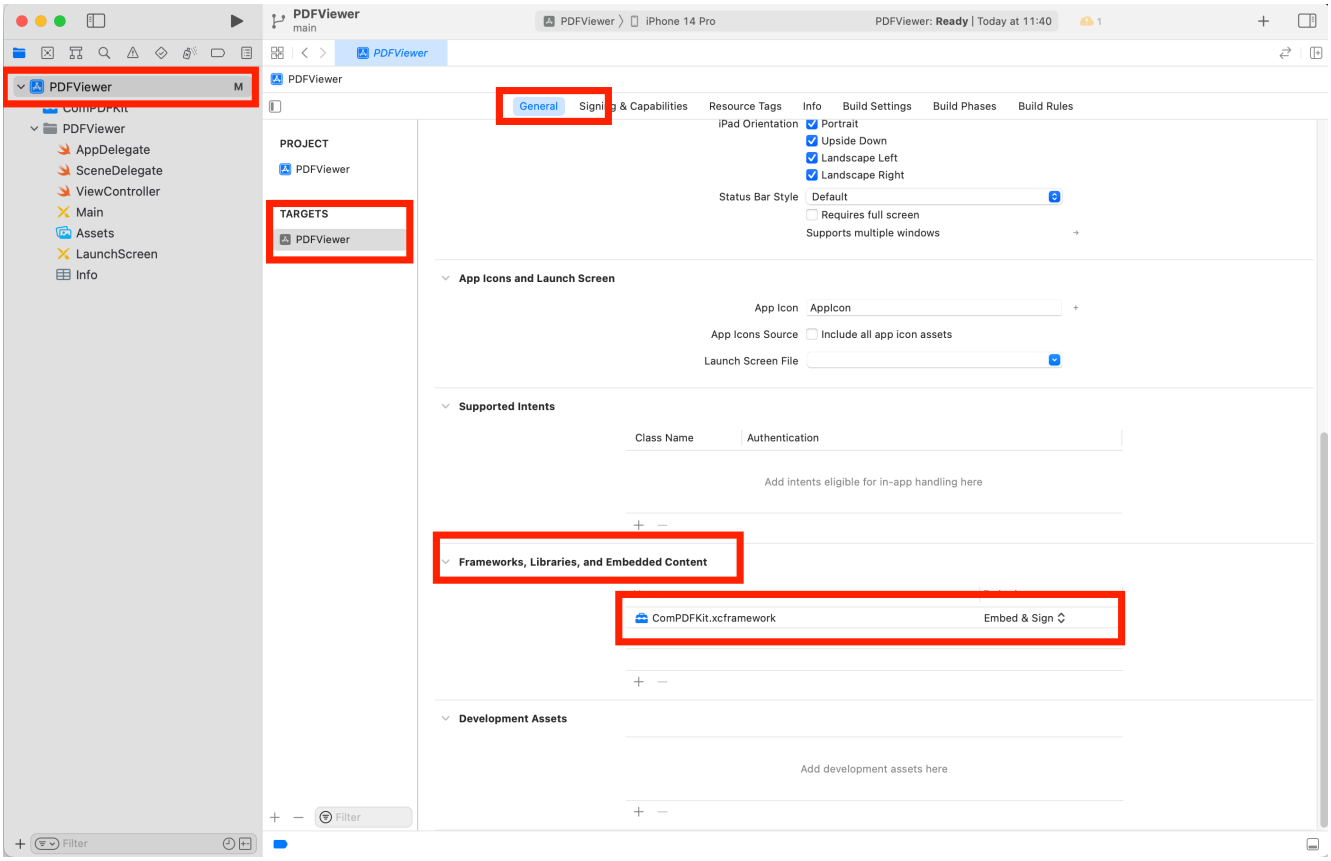
Note: *Make sure to check the* ***Copy items if needed*** *option.*



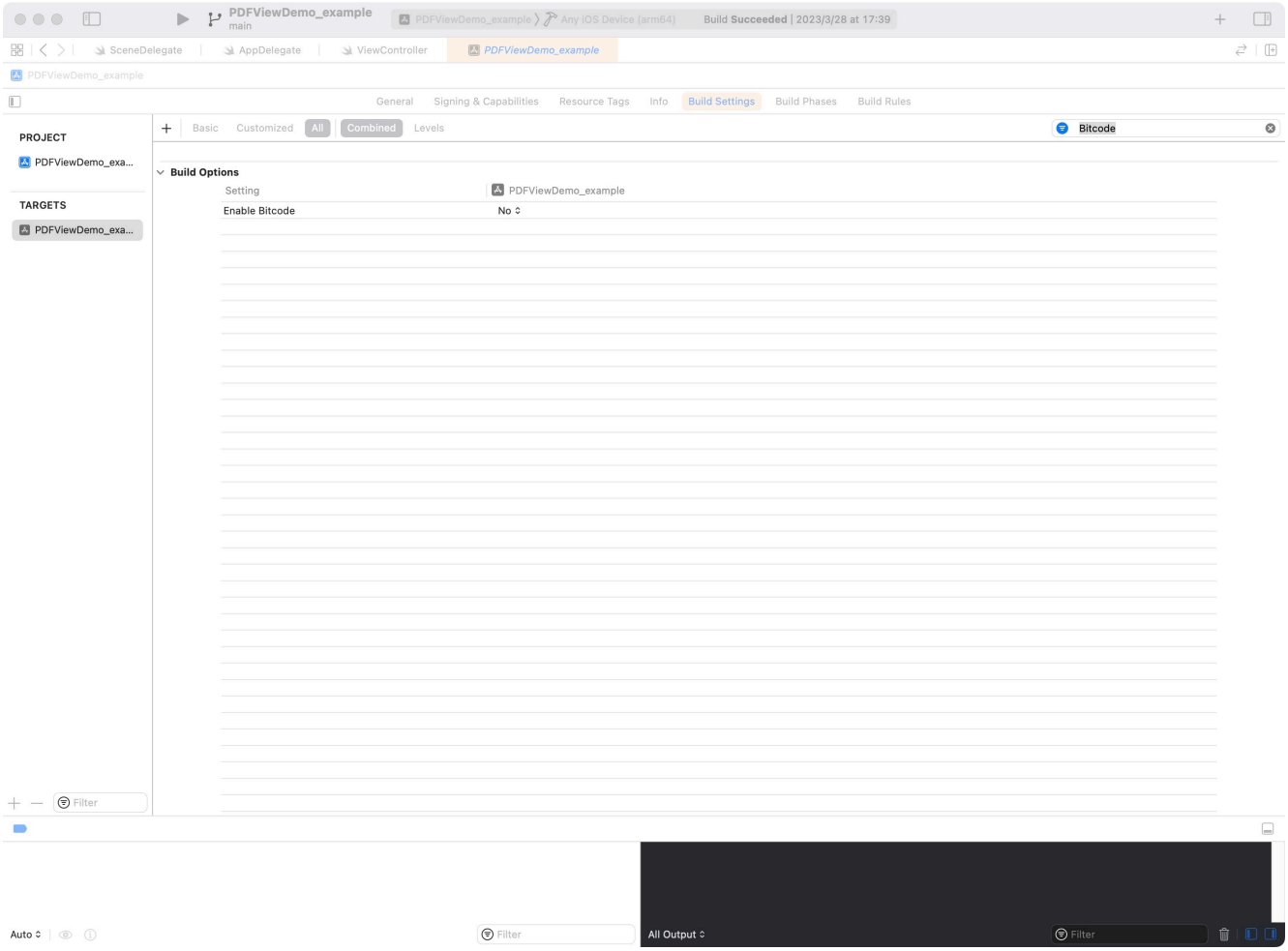
Then, the ***"PDFViewer"*** project will look like the following picture.



2. Add the dynamic xcframework ***"ComPDFKit.xcframework"*** to the Xcode's **Embedded Binaries**. Left-click the project, find **Embedded Binaries** in the **General** tab, and choose **Embed & Sign**.



For earlier versions of Xcode (like Xcode 13), the Bitcode option might be turned on by default, which requires it to be turned off to run. The precise step to do this are illustrated as shown in the picture below.



2.5.3 Swift Compatibility

To use the ComPDFKit Objective-C Framework in your Swift project, you have to create a Swift Bridging Header file in that project. The best way is to create the .h file manually.

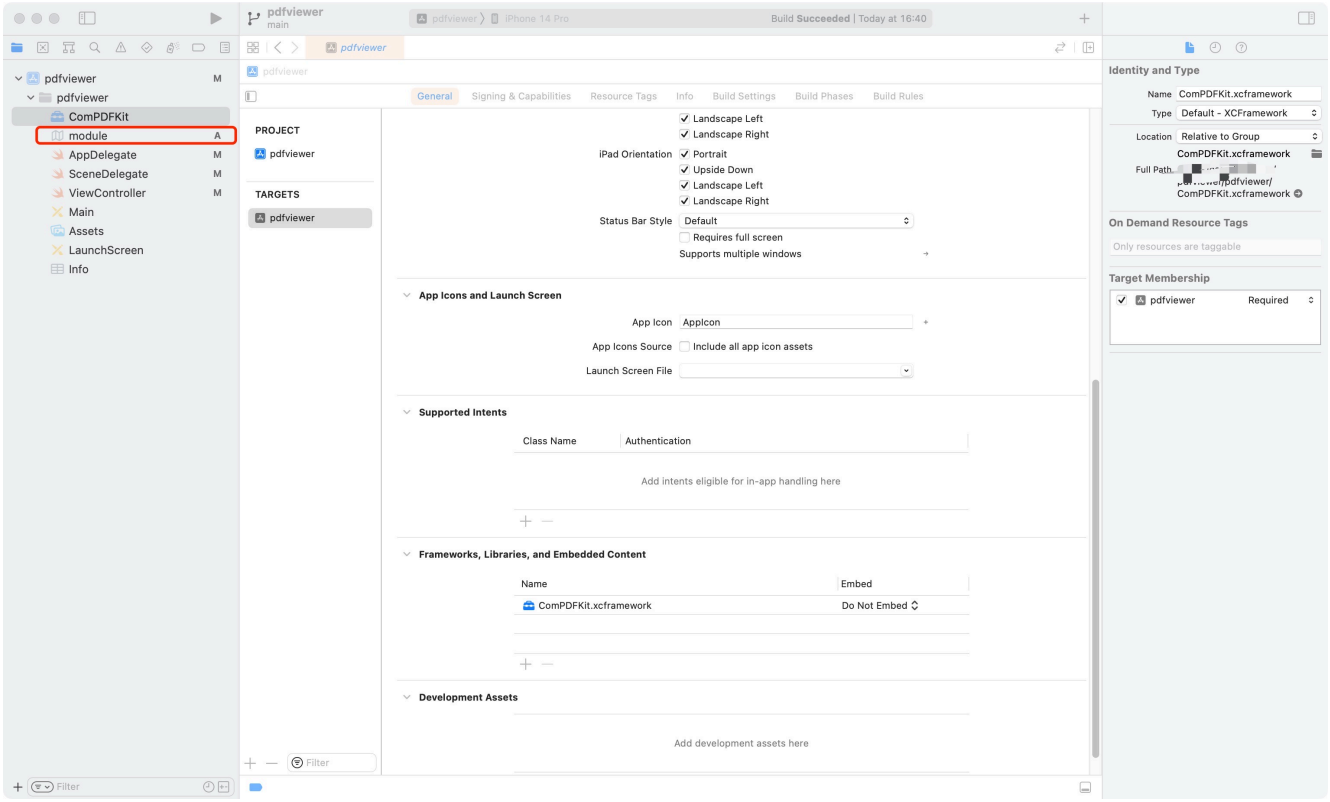
1. The First Method

   First, add a header file to your project with the name: ***"MyProjectName-Bridging-Header.h"***. This will be the single header file where you import any Objective-C code that you need your Swift code to access.

   Then, find Swift Compiler - Code Generation section in your project build settings. Add the path to your bridging header file next to Objective-C Bridging Header from the project root folder. It should be ***"MyProject/MyProject-Bridging-Header.h"***.

2. The Second Method:

   Import modules. Find the ***"ComPDFKit.xcframework"*** folder -> ***"ios-arm64_armv7"*** -> ***"ComPDFKit.framework"*** -> ***"Modules"***. Then, import the ***"Modules"*** folder entirely or just import the ***"Module"*** files within the ***"Modules"*** folder. See the picture below for the details.



The subsequent operations can be configured in the same way as the configuration method of Objective-C.

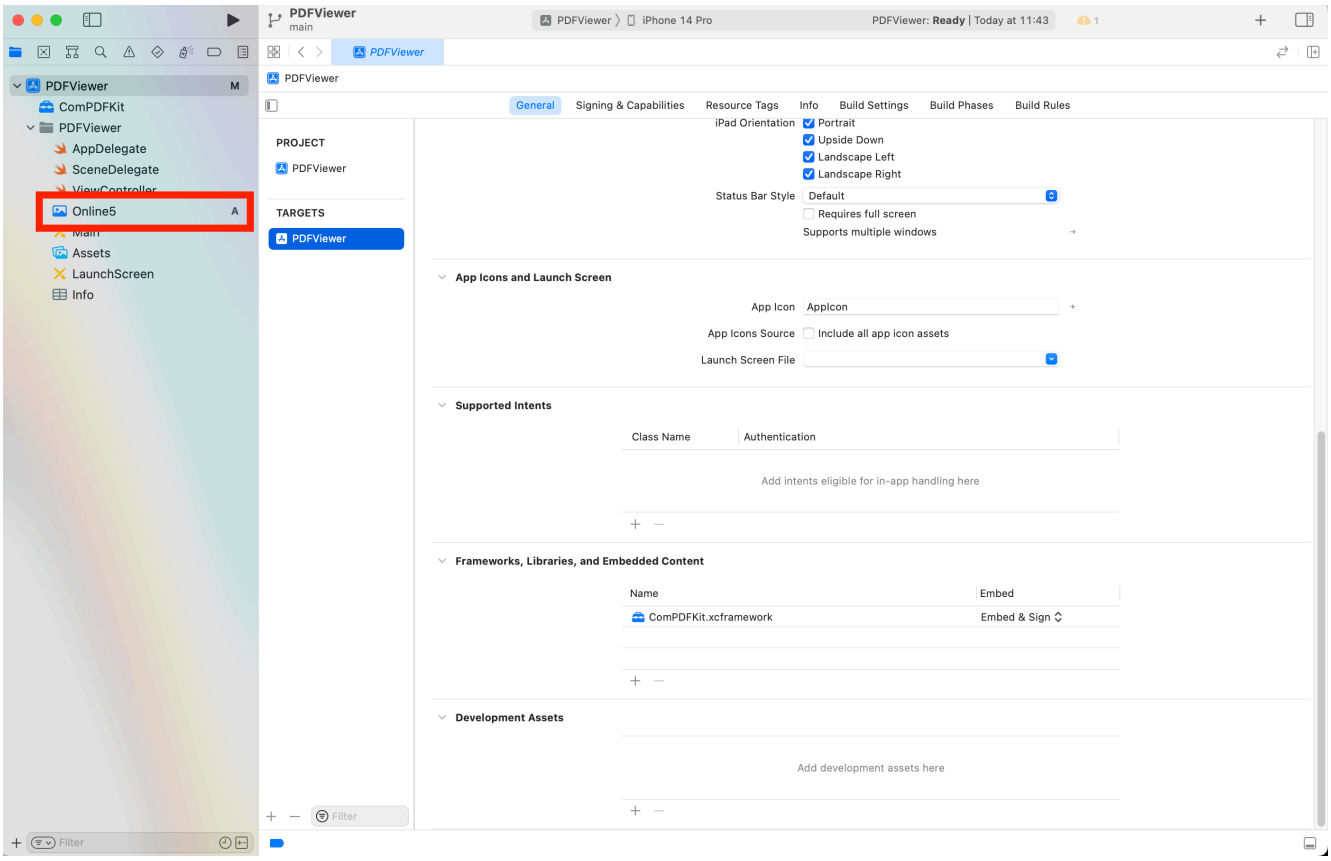
2.5.4 Apply the License Key

Please refer to section 2.3.3 Applying the License Key for detailed steps.

2.5.5 Display a PDF Document

So far, we have added ***"ComPDFKit.xcframework"*** to the ***"PDFViewer"*** project, and finished the initialization of the ComPDFKit PDF SDK. Now, let's start building a simple PDF viewer with just a few lines of code.

1. Prepare a test PDF file, drag and drop it into the newly created **PDFView** project. By this way, you can load and preview the local PDF document using `NSBundle`. The following image shows an example of importing a PDF document named "Online5" into the project.



2. Create a `CPDFDocument` object through **NSURL**, and create a `CPDFView` to display it. The following code shows how to load PDF data using a local PDF path and display it by `CPDFView`.

```
guard let filePath = Bundle.main.path(forResource: "Online5", ofType: "pdf") else {
return  }
let url = URL(fileURLWithPath: filePath)
let document = CPDFDocument(url: url)

let rect = self.view.bounds
let pdfView = CPDFView(frame: self.view.bounds)
pdfView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
pdfView.document = document
```

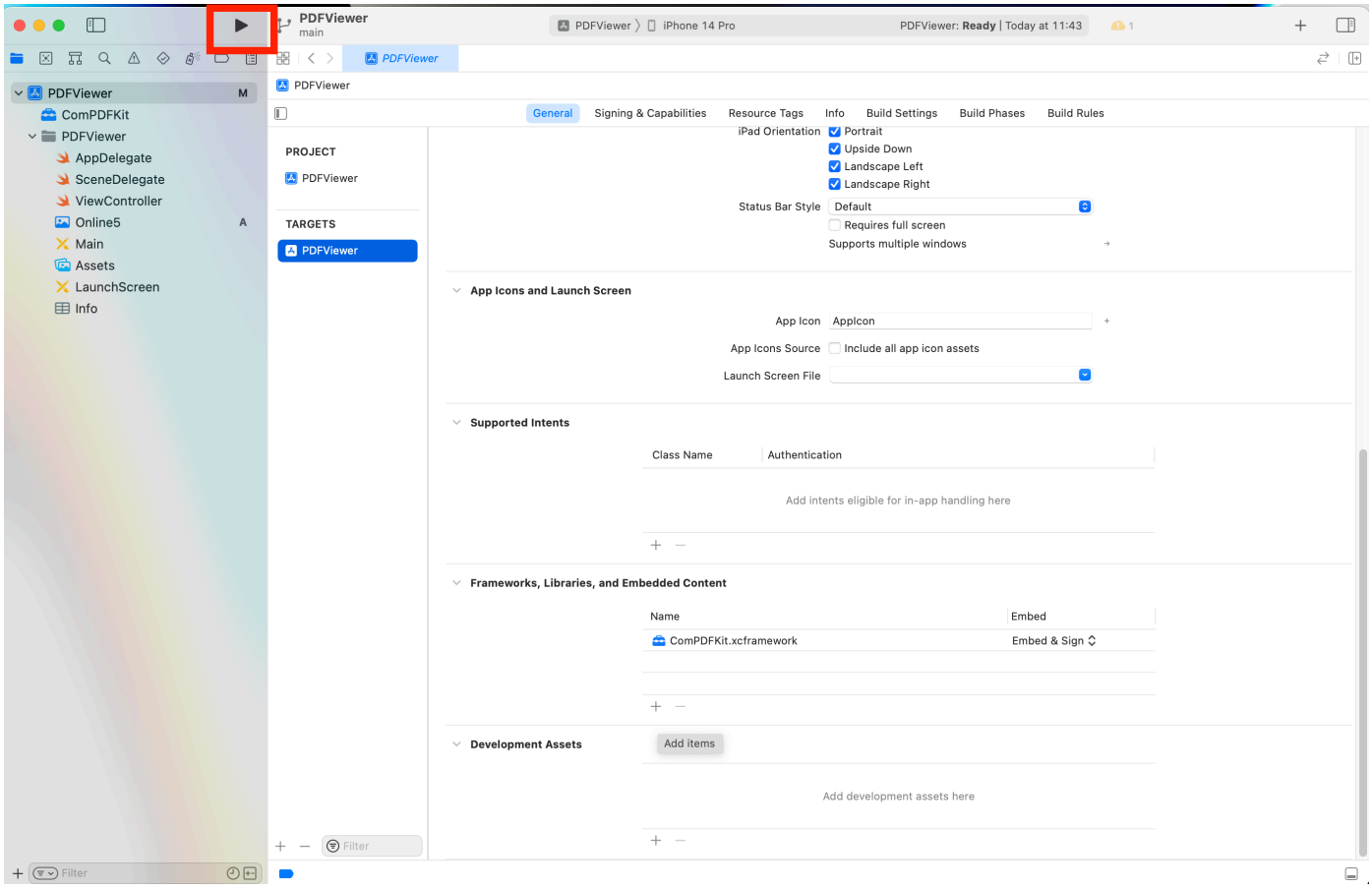
3. Add the created `CPDFView` to the view of the current controller. The sample code shows below.

```
self.view.addSubview(pdfView)
```

The code shown here is a collection of the steps mentioned above:

```
override func viewWillAppear(_ animated: Bool) {
      super.viewWillAppear(animated)

      guard let filePath = Bundle.main.path(forResource: "Online5", ofType: "pdf")
else { return  }
      let url = URL(fileURLWithPath: filePath)
      let document = CPDFDocument(url: url)

      let rect = self.view.bounds
      let pdfView = CPDFView(frame: self.view.bounds)
      pdfView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
      pdfView.document = document
      self.view.addSubview(pdfView)
}
```

4. Connect your device or simulator, and use shortcut **Command_R** to run the App. The PDF file will be opened and displayed.



2.5.6 Troubleshooting

1. Bitcode

   Even when all configurations are correct, there may still be compilation errors. First, check if bitcode is disabled. In earlier versions of Xcode (such as Xcode 13), the Bitcode option may be enabled by default. It needs to be set to **No** in order to run the app.

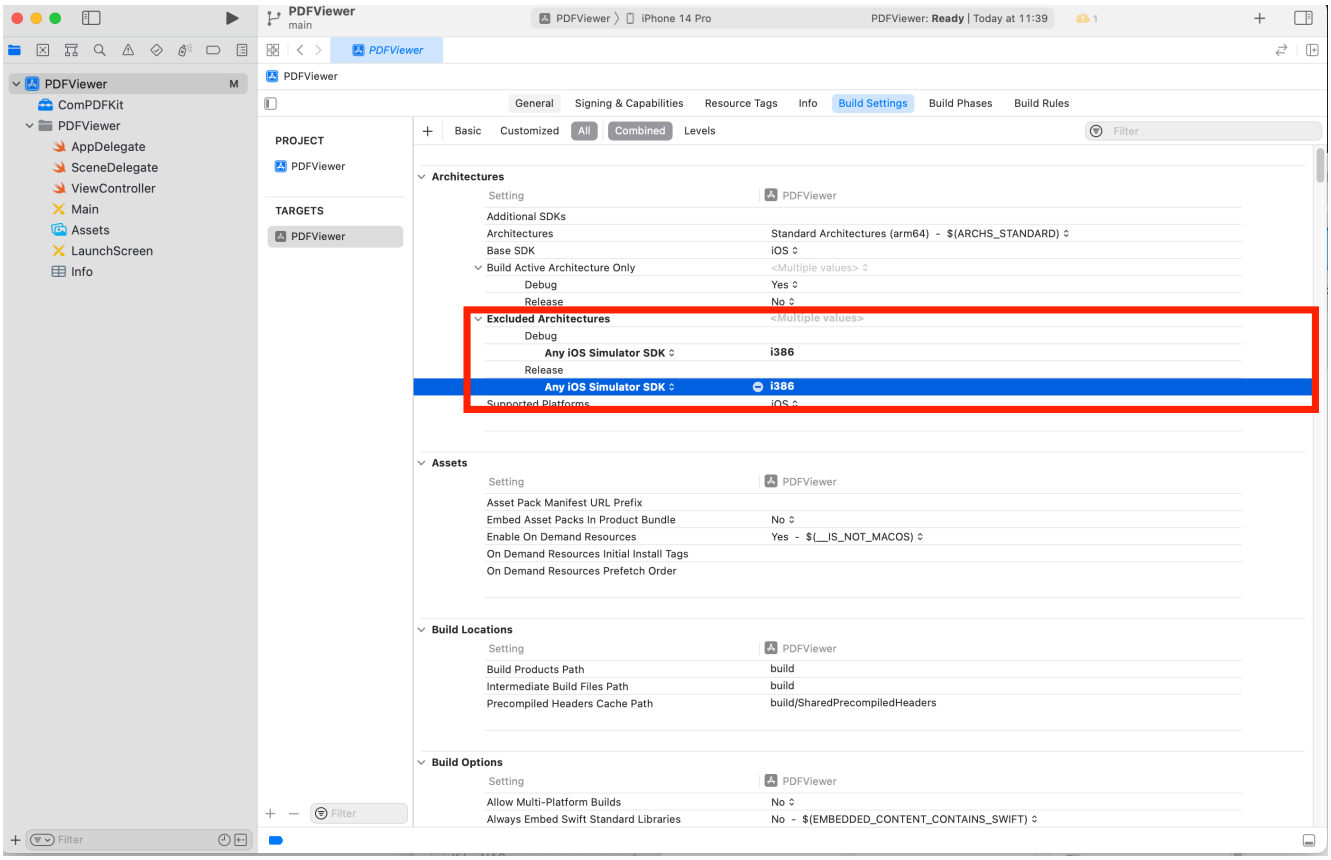
2. License

   If a License setting error occurs, ensure that the Identity (Bundle ID) setting in **General** matches the Bundle ID you provided when contacting us for the license. If an expired License message appears, please contact the ComPDFKit team to obtain the latest License and Key.

3. Cannot Run on i386 Architecture Simulator

   The version of Xcode 12.5 or newer, doesn't support i386 simulators. Apple dropped the i386 after switching to ARM processors and no longer maintains i386 architecture simulators. Please use ARM simulators or x86_64 architecture simulators to test and develop your program.

   So you need to search for **Excluded Architectures** in **Build Settings** in **TARGETS**, and then double-click it. A pop-up window will be popped up, click the plus sign (as shown below) to add **i386**.



4. No PDF Displayed

   Check if the special encoding is required in the path we passed in, or if the local path we passed in exists.

5. Other Problems

   If you meet some other problems when integrating our ComPDFKit PDF SDK for iOS, feel free to contact ComPDFKit team.

2.6 How to Make an iOS App in Objective-C with ComPDFKit

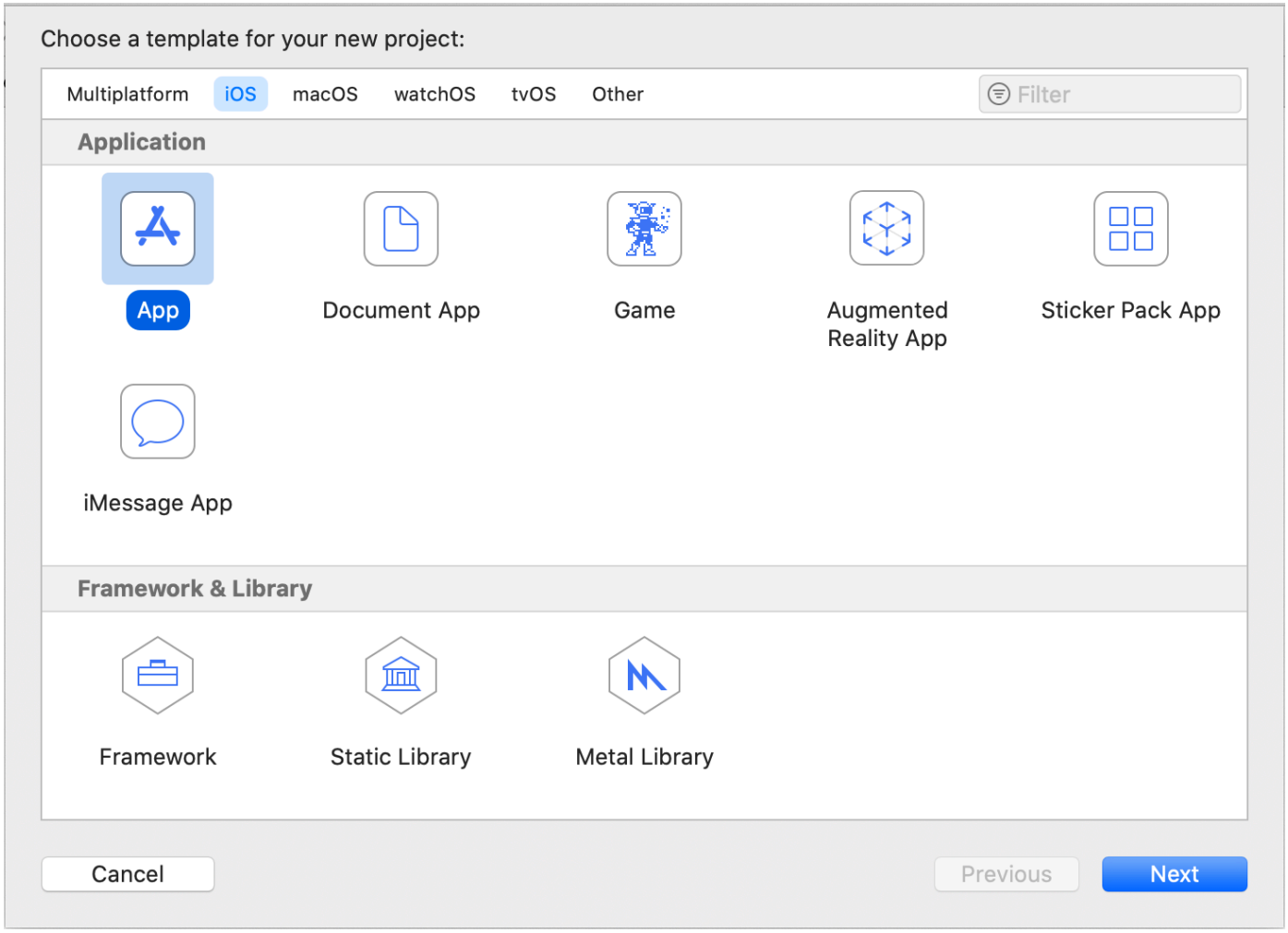
This section will help you to quickly get started with ComPDFKit PDF SDK to make an iOS app in Objective-C with step-by-step instructions, which include the following steps:

1. Create a new iOS project in Objective-C.
2. Integrate ComPDFKit into your apps.
3. Apply the license key.
4. Display a PDF document.

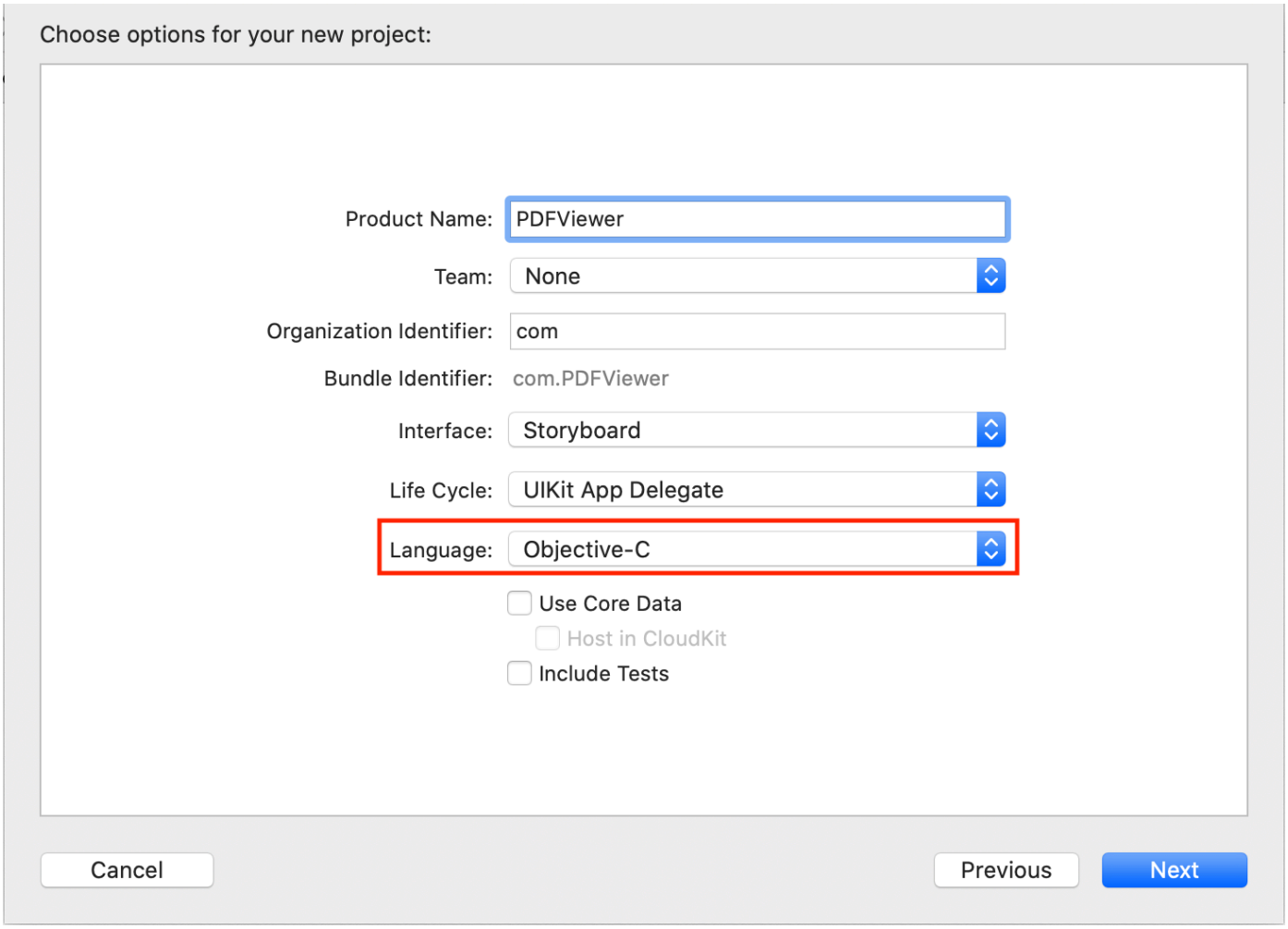
2.6.1 Create a New iOS Project in Objective-C

In this guide, we use Xcode 12.4 to create a new iOS project.

Fire up Xcode, choose **File** -> **New** -> **Project...**, and then select **iOS** -> **Single View Application**. Click **Next**.



Choose the options for your new project. Please make sure to choose Objective-C as the programming language. Then, click **Next**.



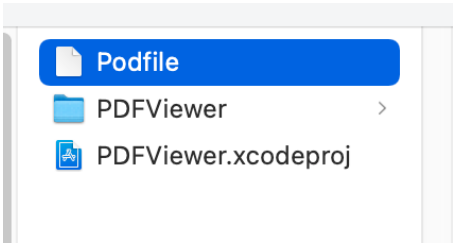
Place the project to the location as desired. Then, click **Create**.

2.6.2 Integrate ComPDFKit into Your Apps

There are two ways to integrate ComPDFKit PDF SDK for iOS into your apps. You can choose what works best for you based on your requirements.

Adding the ComPDFKit CocoaPods Dependency

1. Open the terminal and go to the directory containing your Xcode project: `cd .../PDFViewer`.
2. Run `pod init`. This will create a new Podfile next to your `.xcodeproj` file:



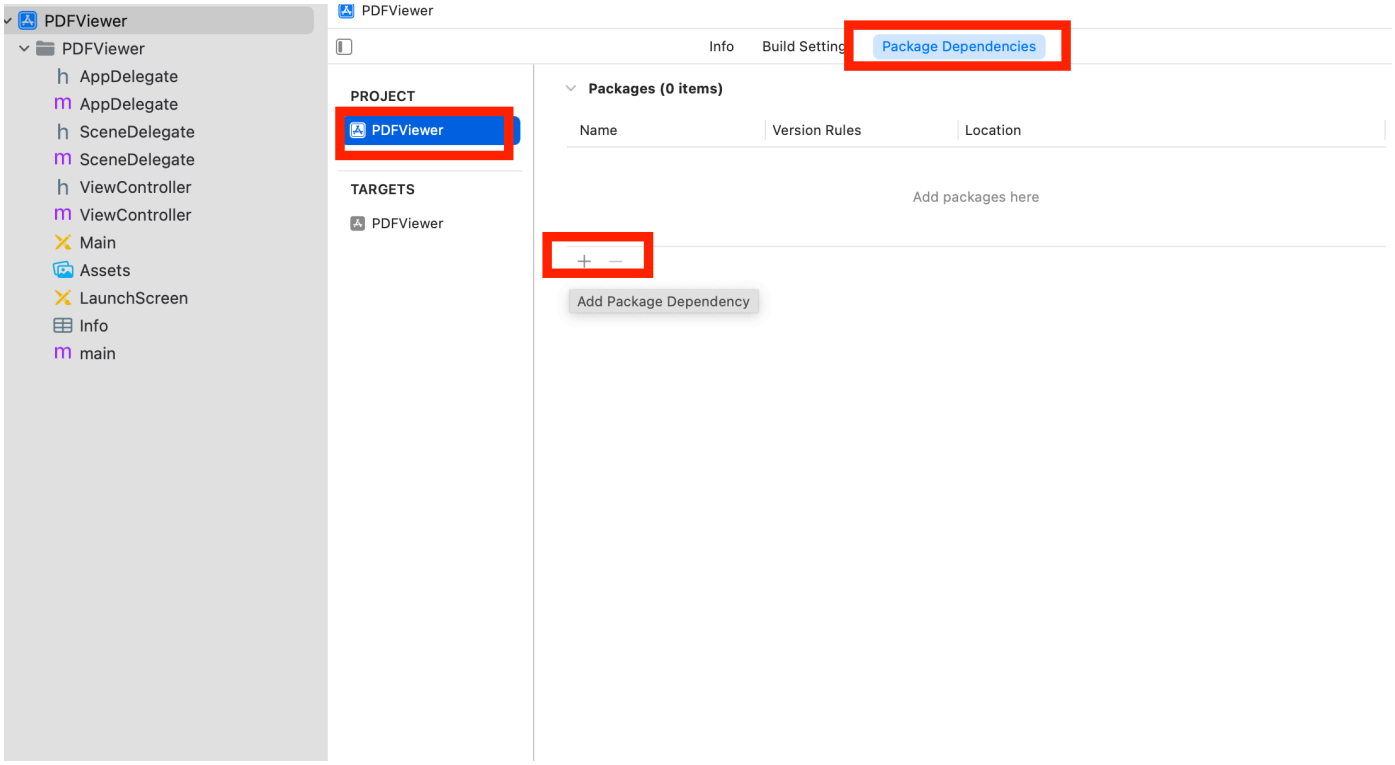
3. Open the newly created Podfile in a text editor and add the ComPDFKit pod URL:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'PDFViewer' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!

+  pod 'ComPDFKit', :git => 'https://github.com/ComPDFKit/compdfkit-pdf-sdk-ios-swift.git', :tag => '2.2.0'

  # Pods for PDFViewer

end
```

4. Run `pod install` and wait for CocoaPods to download ComPDFKit.
5. Open your application's newly created workspace (`PDFViewer.xcworkspace`) in Xcode.

Adding the ComPDFKit Swift Package Manager Dependency

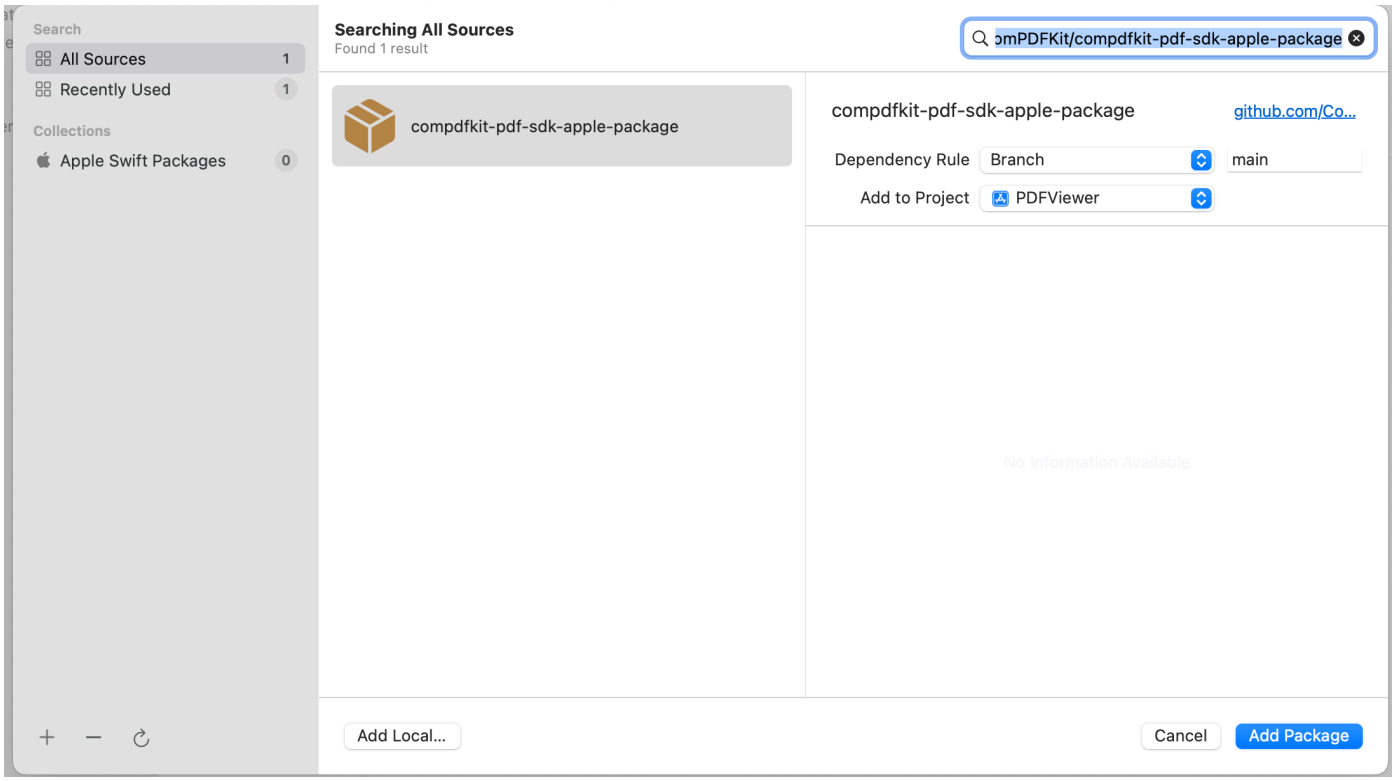
1.Open your application in Xcode and select your project's **Package Dependencies** tab.



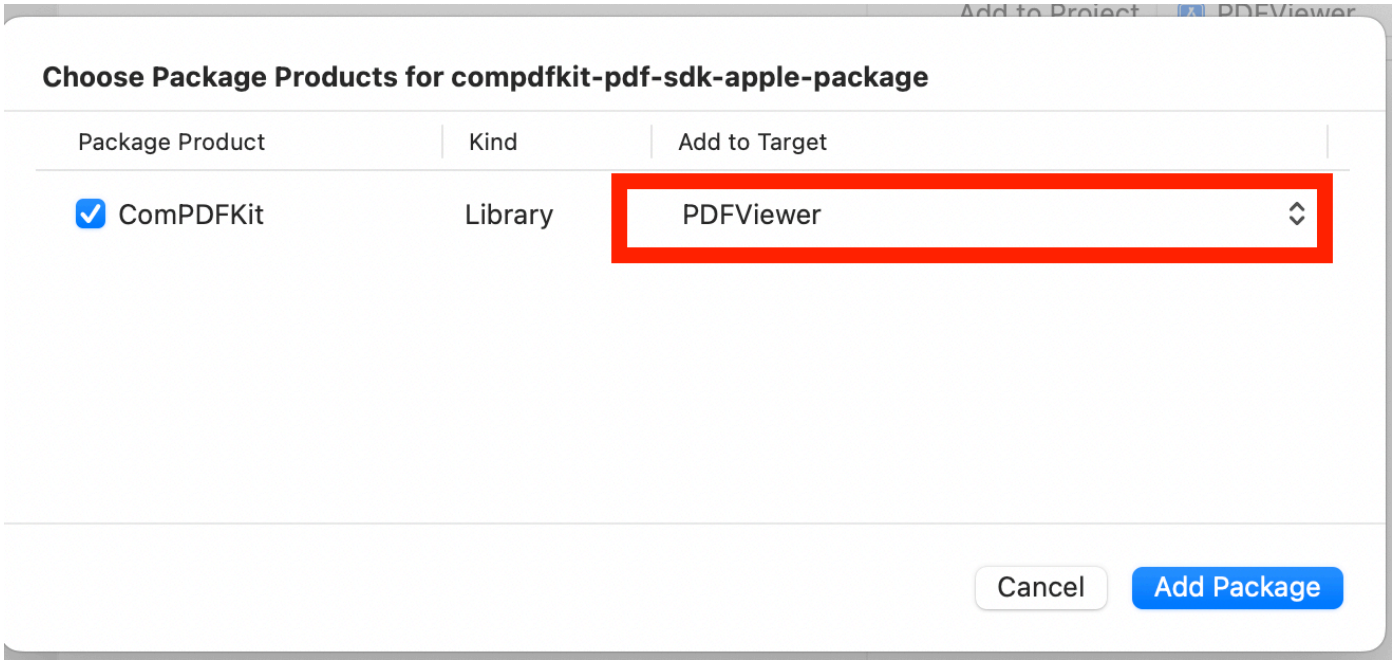
2.Copy the ComPDFKit Swift package repository URL into the search field:

```
https://github.com/ComPDFKit/compdfkit-pdf-sdk-apple-package
```

3.In the **Dependency Rule** fields, select **Branch** > **master**, and then click **Add Package**.



4.After the package download completes, select **Add Package**.

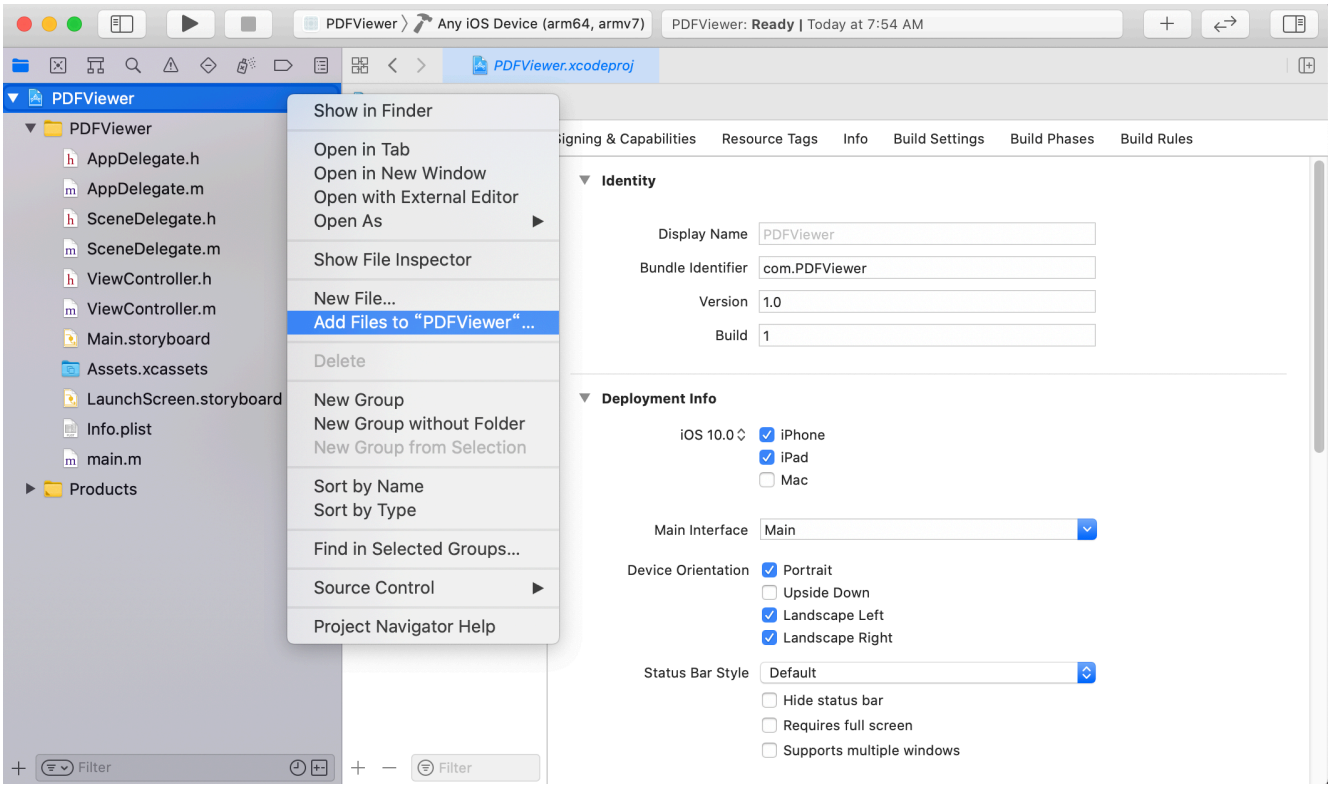


ComPDFKit should now be listed under Swift Package Dependencies in the Xcode Project navigator.

Adding the ComPDFKit XCFrameworks Manually

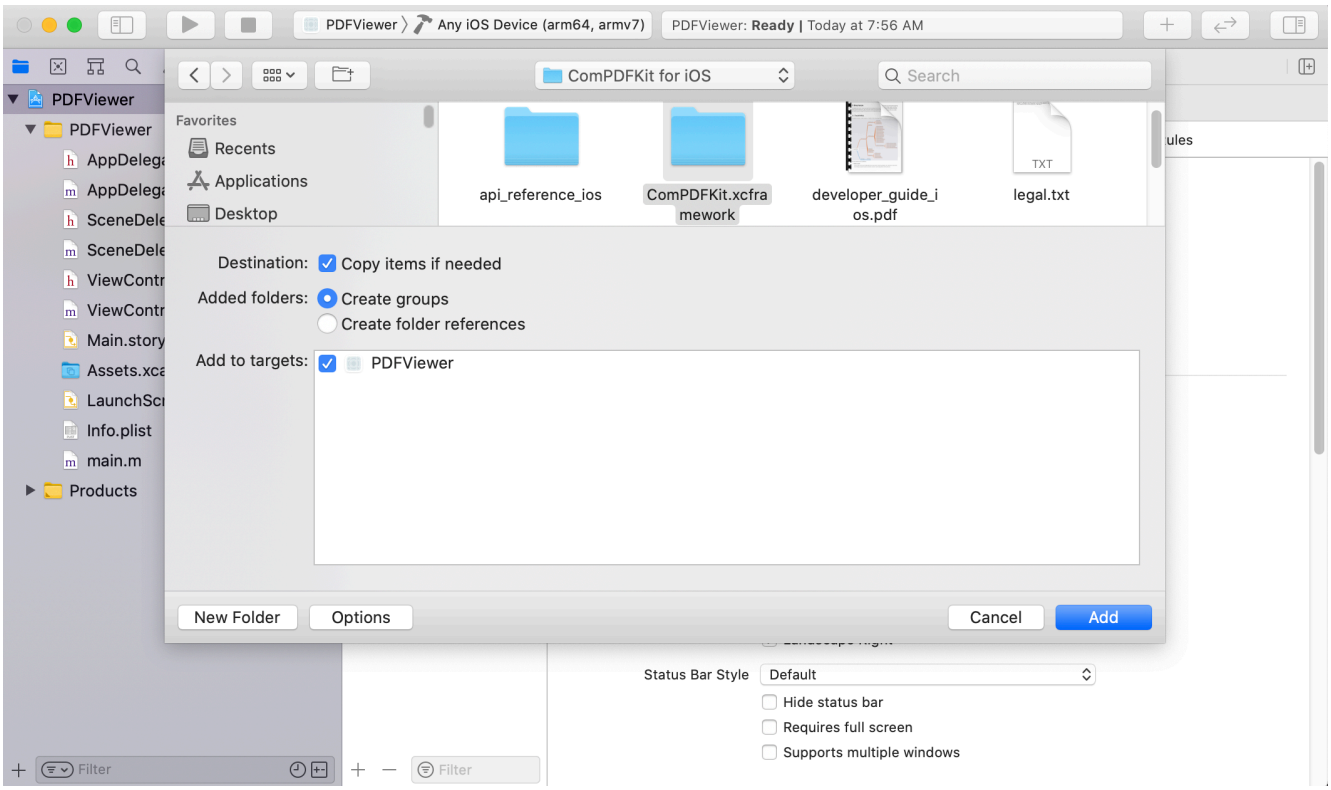
To add the dynamic xcframework ***"ComPDFKit.xcframework"*** into the ***"PDFViewer"*** project, please follows the steps below:

1. Right-click the ***"PDFViewer"*** project, select **Add Files to "PDFViewer"...**.

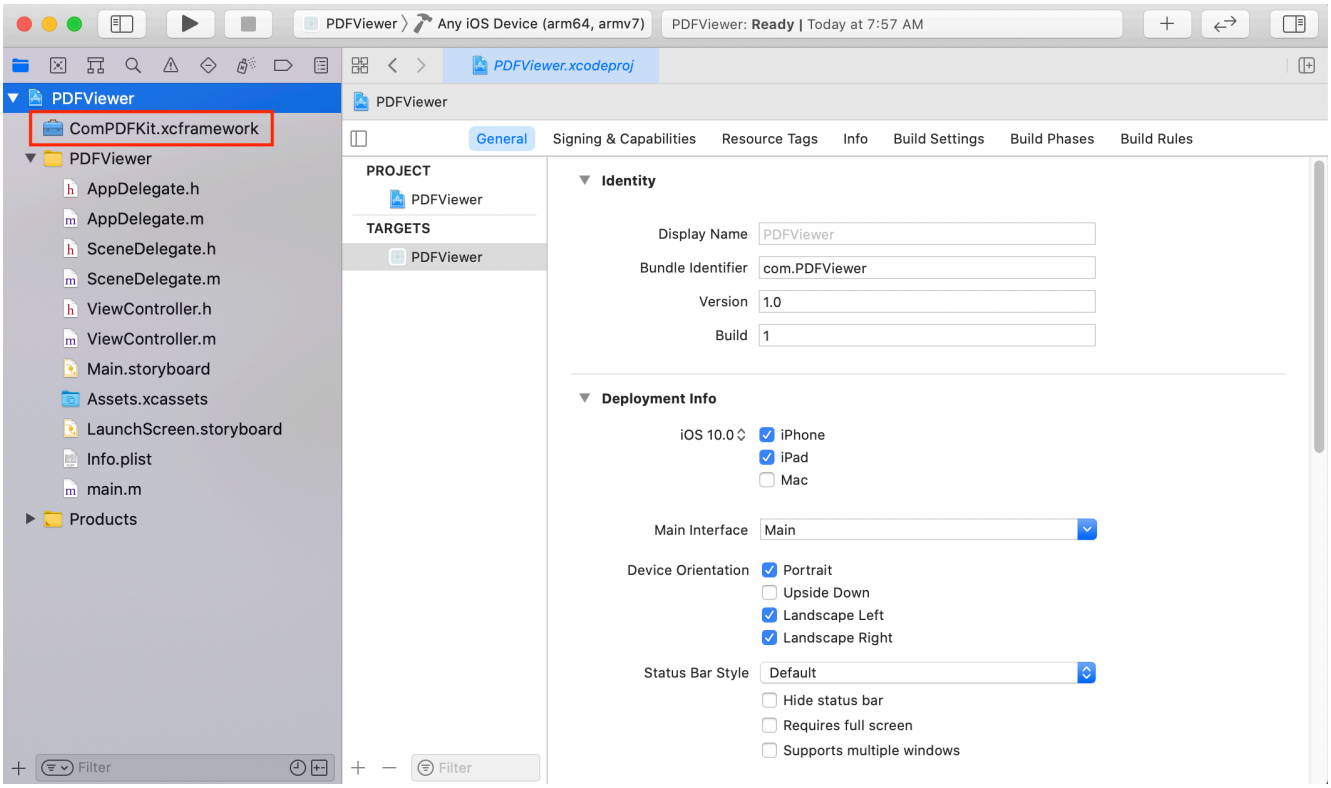


Find and choose ***"ComPDFKit.xcframework"*** in the download package, and then click **Add**.

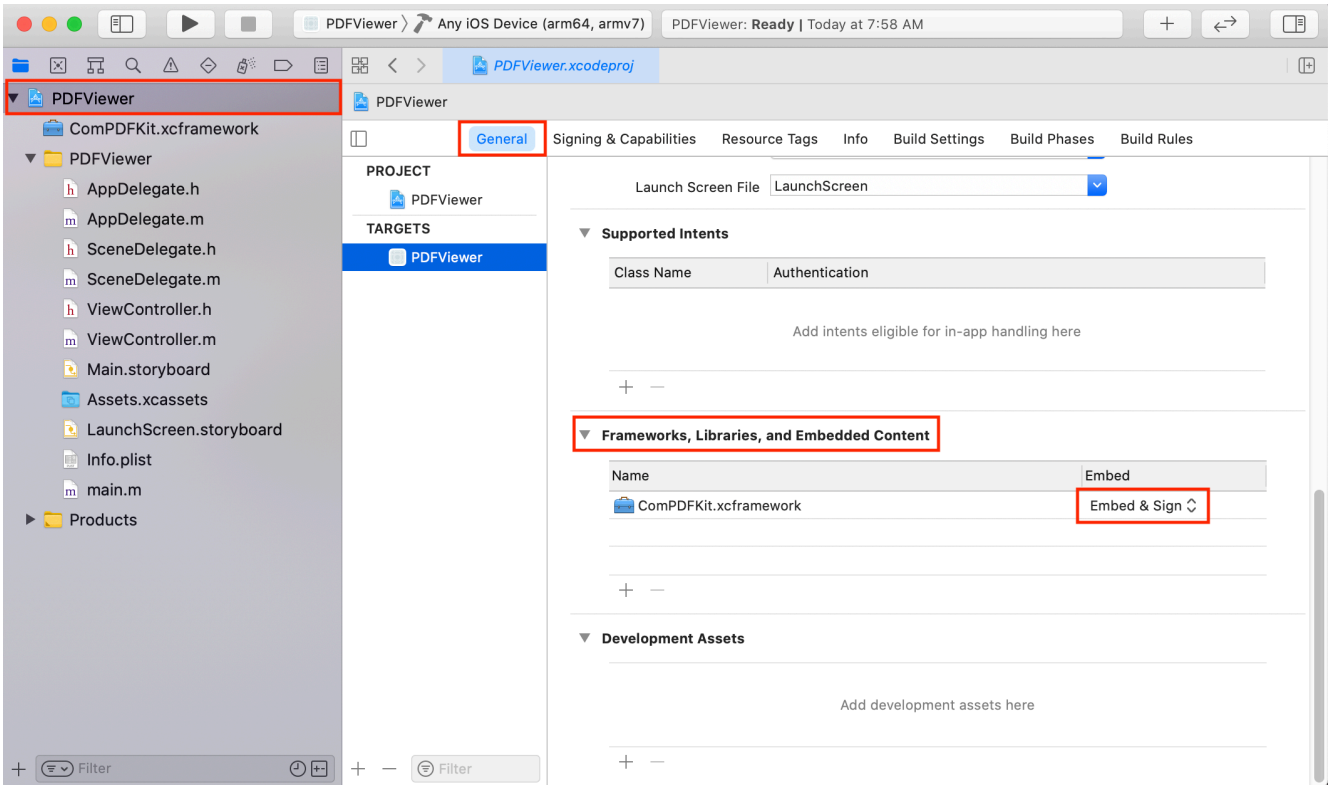
Note: *Make sure to check the* ***Copy items if needed*** *option.*



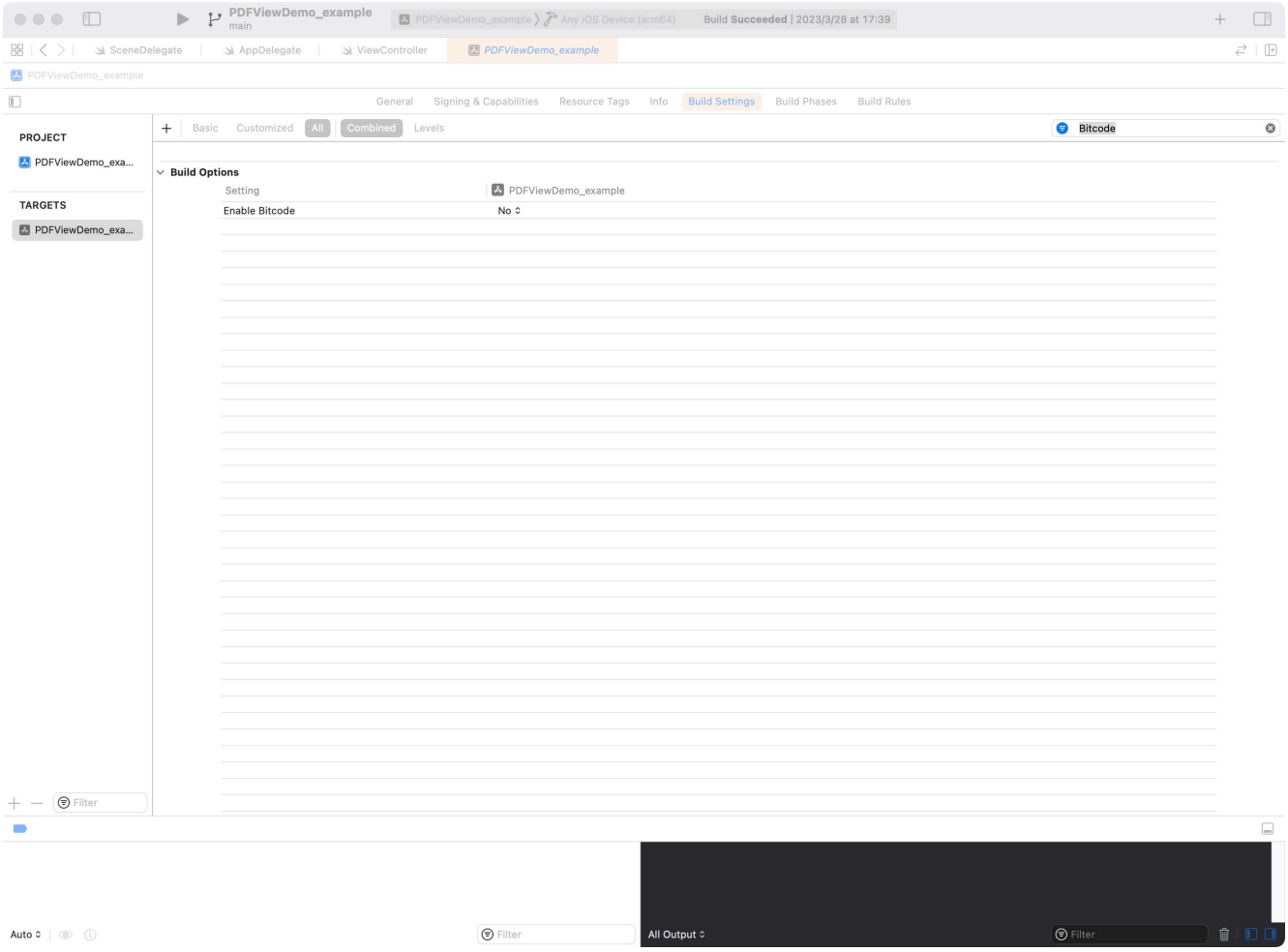
Then, the ***"PDFViewer"*** project will look like the following picture.



2. Add the dynamic xcframework ***"ComPDFKit.xcframework"*** to the Xcode's **Embedded Binaries**. Left-click the project, find **Embedded Binaries** in the **General** tab, and choose **Embed & Sign**.



For earlier versions of Xcode (like Xcode 13), the Bitcode option might be turned on by default, which requires it to be turned off to run. The precise step to do this are illustrated as shown in the picture below.



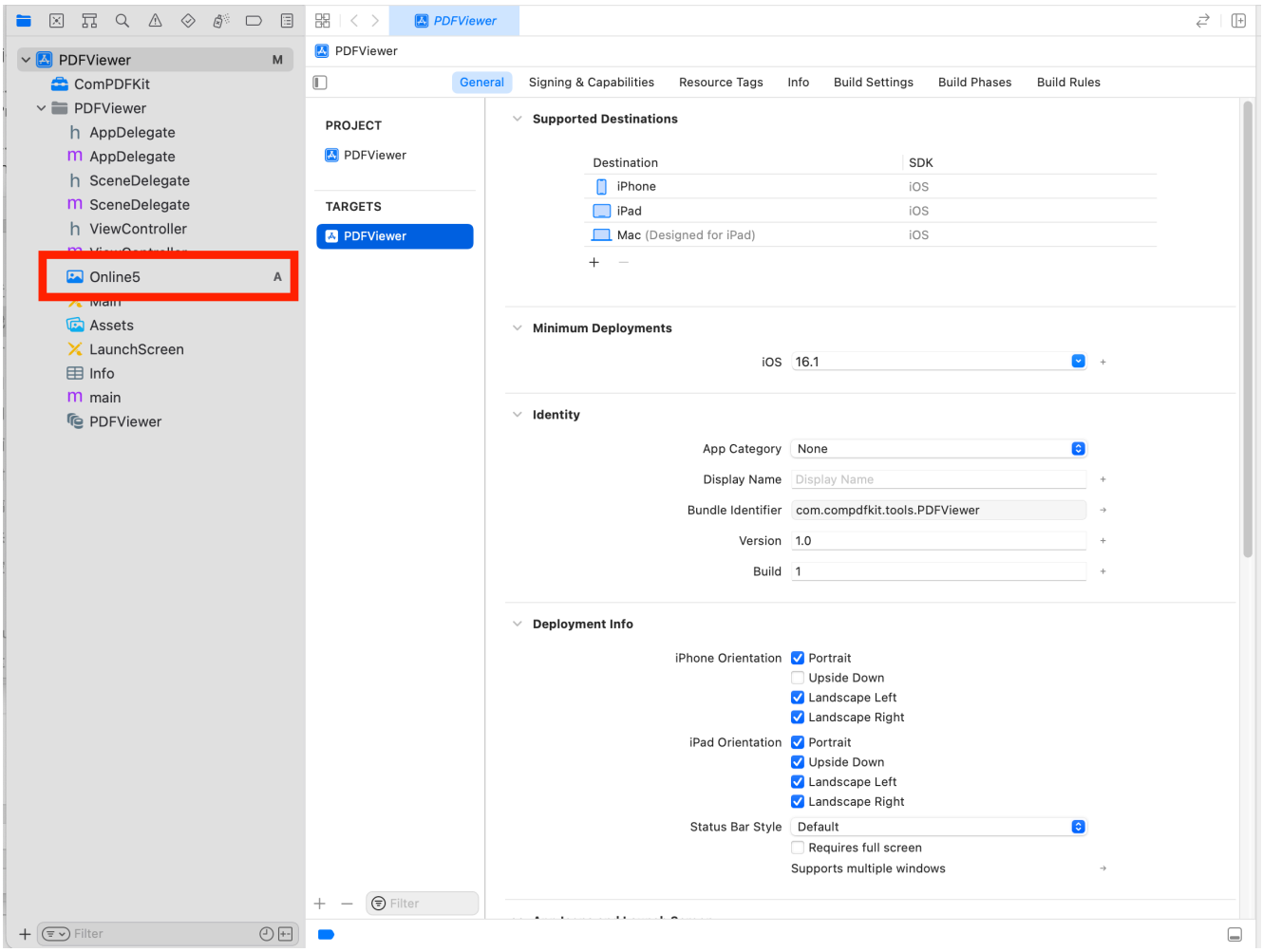
2.6.3 Apply the License Key

Please refer to section 2.3.3 Applying the License Key for detailed steps.

2.6.4 Display a PDF Document

So far, we have added ***"ComPDFKit.xcframework"*** to the ***"PDFViewer"*** project, and finished the initialization of the ComPDFKit PDF SDK. Now, let's start building a simple PDF viewer with just a few lines of code.

1. Prepare a test PDF file, drag and drop it into the newly created **PDFView** project. By this way, you can load and preview the local PDF document using `NSBundle`. The following image shows an example of importing a PDF document named "Online5" into the project.



2. Import `<ComPDFKit/ComPDFKit.h>` at the top of your `UIViewController.m` subclass implementation:

```
#import <ComPDFKit/ComPDFKit.h>
```

3. Create a `CPDFDocument` object through **NSURL**, and create a `CPDFView` to display it. The following code shows how to load PDF data using a local PDF path and display it by `CPDFView`.

```
NSBundle *bundle  = [NSBundle mainBundle];
NSString *pdfPath= [bundle pathForResource:@"Online5" ofType:@"pdf"];
NSURL *url = [NSURL fileURLWithPath:pdfPath];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

CGRect rect = self.view.bounds;
CPDFView *pdfView = [[CPDFView alloc] initWithFrame:rect];
pdfView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
pdfView.document = document;
```

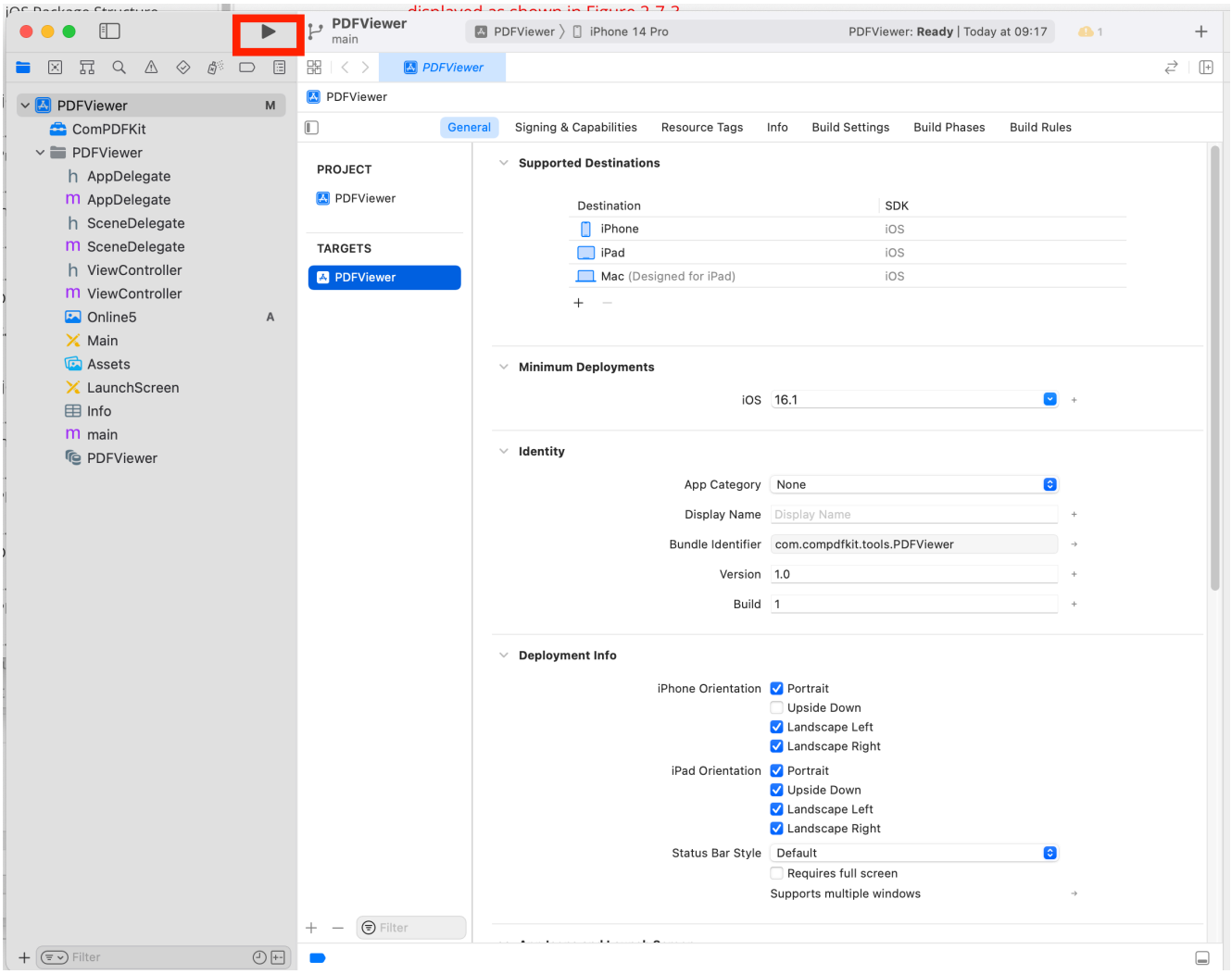
4. Add the created `CPDFView` to the view of the current controller. The sample code shows below.

```
[self.view addSubview:pdfView];
```

The code shown here is a collection of the steps mentioned above:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    NSBundle *bundle  = [NSBundle mainBundle];
    NSString *pdfPath= [bundle pathForResource:@"Online5" ofType:@"pdf"];
    NSURL *url = [NSURL fileURLWithPath:pdfPath];
    CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

    CGRect rect = self.view.bounds;
    CPDFView *pdfView = [[CPDFView alloc] initWithFrame:rect];
    pdfView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
    pdfView.document = document;
    [self.view addSubview:pdfView];
}
```

5. Connect your device or simulator, and use shortcut **Command_R** to run the App. The PDF file will be opened and displayed.



2.6.5 Troubleshooting

1. Bitcode

   Even when all configurations are correct, there may still be compilation errors. First, check if bitcode is disabled. In earlier versions of Xcode (such as Xcode 13), the Bitcode option may be enabled by default. It needs to be set to **No** in order to run the app.

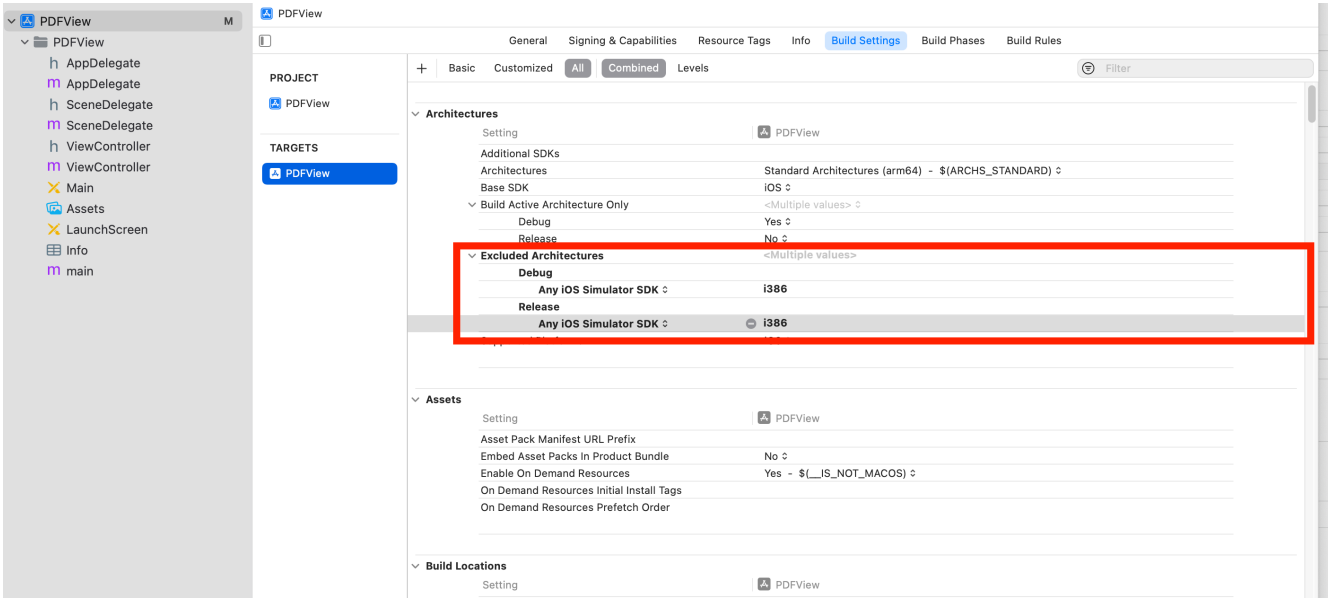
2. License

   If a License setting error occurs, ensure that the Identity (Bundle ID) setting in **General** matches the Bundle ID you provided when contacting us for the license. If an expired License message appears, please contact the ComPDFKit team to obtain the latest License and Key.

3. Cannot Run on i386 Architecture Simulator

   The version of Xcode 12.5 or newer, doesn't support i386 simulators. Apple dropped the i386 after switching to ARM processors and no longer maintains i386 architecture simulators. Please use ARM simulators or x86_64 architecture simulators to test and develop your program.

   So you need to search for **Excluded Architectures** in **Build Settings** in **TARGETS**, and then double-click it. A pop-up window will be popped up, click the plus sign (as shown below) to add **i386**.



4. No PDF Displayed

   Check if the special encoding is required in the path we passed in, or if the local path we passed in exists.

5. Other Problems

   If you meet some other problems when integrating our ComPDFKit PDF SDK for iOS, feel free to contact ComPDFKit team.

2.7 UI Customization

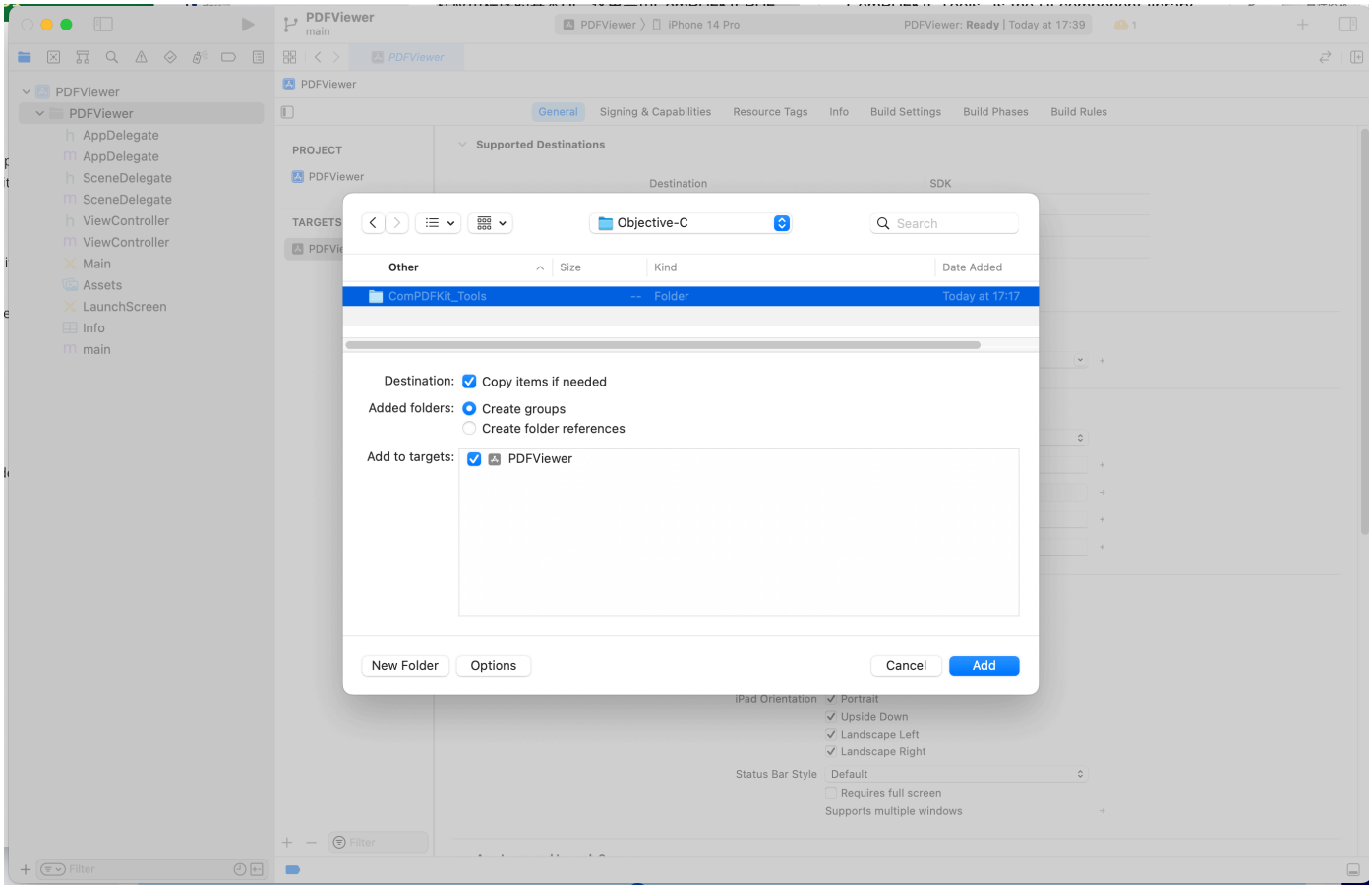
The folder of ***"ComPDFKit_Tools"*** includes the UI components to help conveniently integrate ComPDFKit PDF SDK. We have also built five standalone function programs, namely **Viewer**, **Annotations**, **ContentEditor**, **Forms**, **DocsEditor**, and **Digital Signatures**, using this UI component library. Additionally, we have developed a program called **PDFViewer** that integrates all the above-mentioned example features for reference.

In this section, we will introduce how to use it from the following parts:

1. Overview of ***"ComPDFKit_Tools"*** Folder: Show the folder structure and the main features included in the corresponding component.
2. UI Components: Introduce the UI components and how to use them easily and fast.
3. How to Use the `CPDFListView` Class: Contain the details to use `CPDFListView`.
4. Learn More: Show the special code structure and features for programming platforms.

How to Use the Default UI of ComPDFKit PDF SDK to Build iOS Applications:

The ***"ComPDFKit_Tools"*** in ComPDFKit PDF SDK is a UI component library of the sample project, which includes the basic UI of the application. To make a custom iOS application UI with the default UI of ComPDFKit PDF SDK, you can find the corresponding ***"ComPDFKit_Tools"*** file in the ***"Lib/Tools/Objective-C/"*** or ***"Lib/Tools/Swift/"*** (Choose the right one according to path according to your development language), and import it into your project.



2.7.1 Overview of *"ComPDFKit_Tools"* Folder

There are nine modules in ***"ComPDFKit_Tools"***: ***"Common"***, ***"Viewer"***, ***"Annotations"***, ***"ContentEditor"***, ***"Forms"***, and ***"DocsEditor"***, ***"Digital Signatures"***, ***"Security"***, and ***"Watermark"***. Each includes the code and UI components for corresponding PDF features, like the following table, to process PDFs.

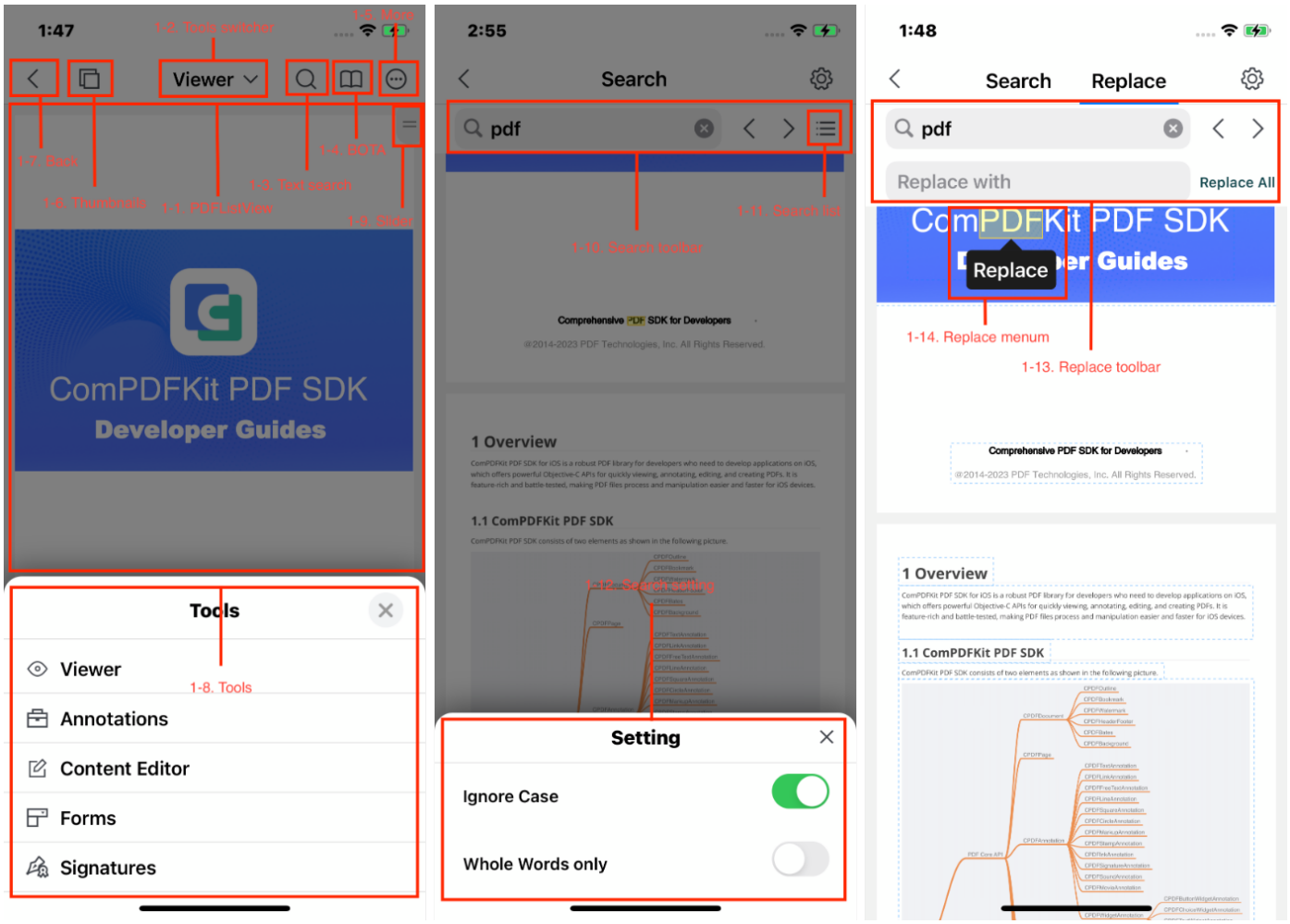
Folder	Subfolder	Description
Common	-	Include the reusable UI components and interaction of Viewer, Annotations, ContentEditor, Forms, and DocsEditor.
Viewer	PDFBookmark	Include the UI components and interaction of editing bookmarks and jumping pages.
	PDFOutline	Include the UI components and interaction of jumping and displaying the PDF outline.
	PDFSearch	Include the UI component and interaction for searching PDFs and generating the search list.
	PDFMoreMenu	Include the UI components and interaction of view setting, getting document information, sharing PDFs, switching PDFs, and editing PDF pages.
	PDFThumbnail	Include the UI component and interaction of PDF thumbnails.
Annotations	PDFAnnotationBar	Include the toolbar for choosing the annotation types, setting the properties of annotations, and undo/redo annotation property settings.
	PDFAnnatationList	Include the UI component and interaction of getting the annotation list.
	PDFAnnotationProperties	Include the property panel and UI component of setting annotation properties.
ContentEditor	PDFEditBar	Include the toolbar to edit PDF text/images and undo/redo the processing of editing PDF text/images.
	PDFEditProperties	Include the UI component and interaction of PDF text and image editing.
Forms	PDFFormBar	Include the toolbar to set the properties of tables and undo/redo the processing of forms.
	PDFFormProperties	Include the property panel and interaction to

		set the properties of forms.
DocsEditor	PDFPageEdit	Include the UI component and interaction of the document editor.
	PDFPageEditBar	Include the toolbar for creating, replacing, rotating, extracting, and deleting PDF pages.
	PDFPageEditInsert	Include the UI component and interaction for creating blank pages and inserting other PDF pages.
DigitalSignature	PDFDigitalSignatureBar	Include the toolbar for creating form fields, filling out digital signatures, and verifying signatures.
	PPDFDigitalSignatureProperties	Include the property panel and interaction to set the signature appearance and edit the properties of digital signatures.
Watermark	WatermarkSetting	Include the property panel and interaction to set and tile the text and image watermarks.
Security	-	Include the property panel to add and remove the passwords to open documents or set file permissions.

2.7.2 UI Components

This section mainly introduces the connection between the UI components and API configuration of ***"ComPDFKit_Tools"***, which can not only help you quickly get started with the default UI but also help you view the associated API configuration. These UI components could be used and modified to create your customize UI.

Part 1:



The left image above shows the main UI components associated with the Viewer module API, which are also the basic UI components of ***"ComPDFKit_Tools"***. The right image above shows the main UI components associated with the API for Text search. The following shows the details of the connection between UI components and APIs.

Number	Name	Functionality	Description
1-1	PDFView	`CPDFListView`	See the 2.5.3 part of this guide — How to Use the `CPDFListView` Class.
1-2	Tools switcher	`CPDFToolsViewController`	Switch the feature modules.
1-3	Text search	`CSearchToolbar`	Enter the searching mode.
1-4	BOTA	`CPDFBOTAViewController`	Enter the list of outlines, bookmarks, and annotations.
1-5	More	`CPDFPopMenuView`	Enter the manu of **More**.
1-6	Thumbnails	`CPDFThumbnailViewController`	Enter the thumbnails of PDF pages.
1-7	Back	`CNavBarButtonItem`	Exit the preview interface.
1-8	Tools	`CToolModel`	Enum types for mode switching.
1-9	Slider	`CPDFSlider`	Allow users to jump to other specific pages quickly.
1-10	Search toolbar	`CSearchToolbar`	Allow searching keywords.
1-11	Search list	`CPDFSearchResultsViewController`	Searching result list.
1-12	Search setting	`CSearchSettingViewController`	Allow users to configure search properties.
1-13	Replace toolbar	`CSearchToolbar`	Allow keyword search and replace.
1-14	Replace menu	`CSearchContentView`	Allow replacing a single keyword from the context menu.

Here is the method to initialize the **Tools switcher** (Number 2) below:

```
// CPDFToolsViewController: Initialize an array of CToolModel type: toolArrays
// toolArrays: Save the type and number of Tools display mode
let toolsVc = CPDFToolsViewController(customizeWithToolArrays: [NSNumber(value:
CPDFToolFunctionTypeState.viewer.rawValue),
                                                                NSNumber(value:
CPDFToolFunctionTypeState.edit.rawValue),
                                                                NSNumber(value:
CPDFToolFunctionTypeState.annotation.rawValue),
                                                                NSNumber(value:
CPDFToolFunctionTypeState.form.rawValue)])
```

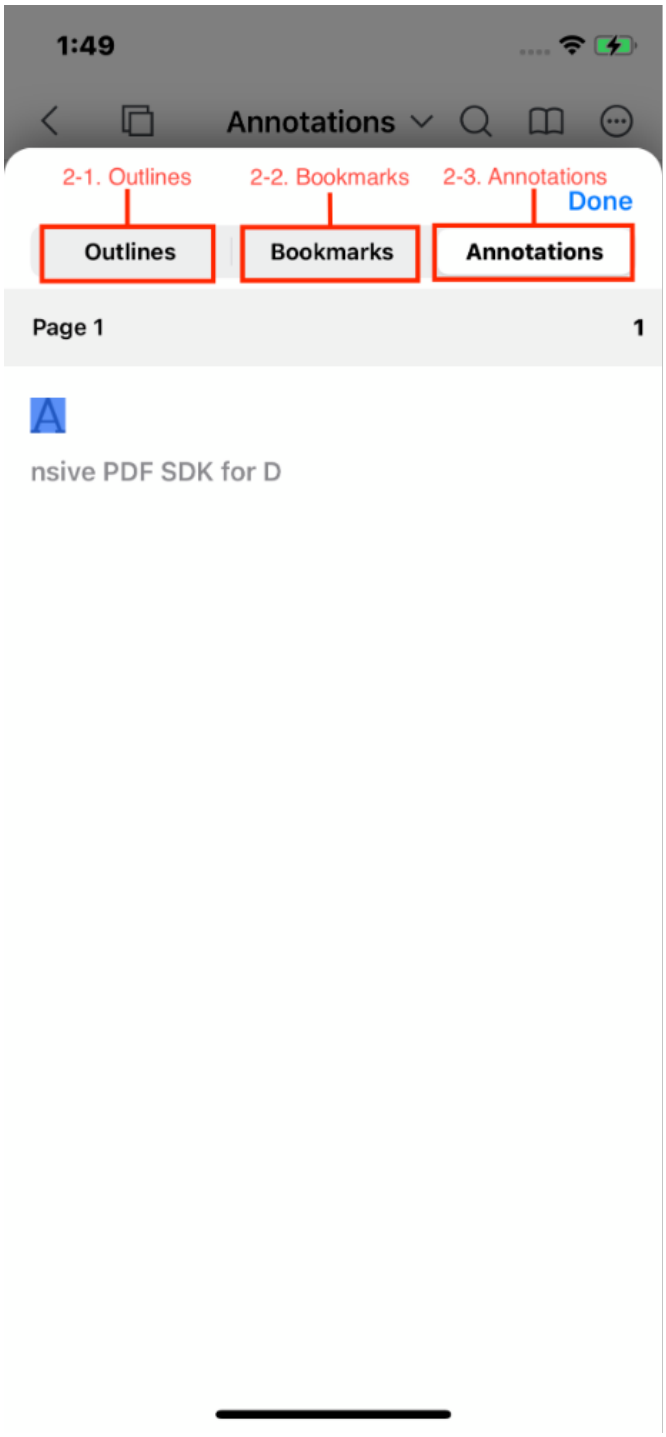
```
// CPDFToolsViewController: Initialize an array of CToolModel type: toolArrays
// toolArrays: Save the type and number of Tools display mode
CPDFToolsViewController * toolsVc = [[CPDFToolsViewController alloc]
initCustomizeWithToolArrays:@[@(CToolModelViewer),@(CToolModelEdit),@(CToolModelAnnotatio
n),@(CToolModelForm)]];
```

Here is the method to initialize the **BOTA** (Number 4) below:

```
// CPDFBOTAViewController: Initialize a CPDFListview object pdfView and an array of
CPDFBOTATypeState type navArrays
// navArrays: Save the displaying list types and number of BOTA
  let navArrays: [CPDFBOTATypeState] = [.CPDFBOTATypeStateOutline,
                                        .CPDFBOTATypeStateBookmark,
                                        .CPDFBOTATypeStateAnnotation]
        if(self.pdfListView != nil) {
            let botaViewController = CPDFBOTAViewController(customizeWith:
self.pdfListView!, navArrays: navArrays)
        }
```

```
// CPDFBOTAViewController: Initialize a CPDFListview object pdfView and an array of
CPDFBOTATypeState type navArrays
// navArrays: Save the displaying list types and number of BOTA
CPDFBOTAViewController *botaVC = [[CPDFBOTAViewController alloc]
initCustomizeWithPDFView:pdfListView
navArrays:@[@(CPDFBOTATypeStateOutline),@(CPDFBOTATypeStateBookmark),@(CPDFBOTATypeStateA
nnotation)]];
```

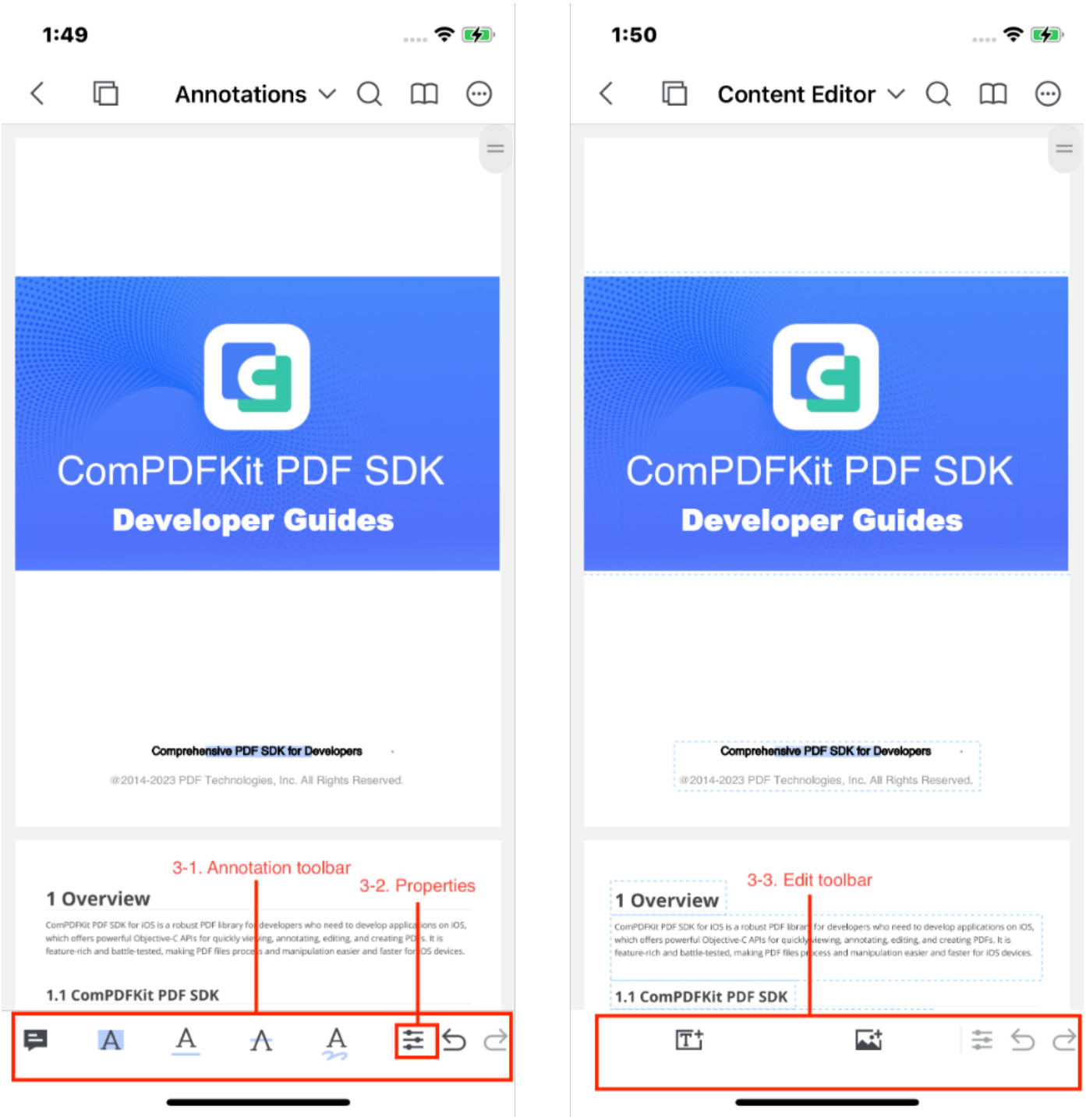
Part 2:



The left image above shows the main UI components associated with the `CPDFBOTAViewController` API. The right image above shows the main UI components associated with the `CPDFPopMenuView` API. See the following details of the connection between UI components and APIs.

Number	Name	Functionality	Description
2-1	Outlines	`CPDFOutlineViewController`	Enter the outlines of PDFs.
2-2	Bookmarks	`CPDFBookmarkViewController`	Enter the bookmark list of PDFs.
2-3	Annotations	`CPDFAnnotationViewController`	Enter the annotation list of PDFs.
2-4	View Setting	`CPDFDisplayViewController`	Enter the view setting of PDFs to set displaying mode, reading themes, splitting view, etc.
2-5	Page Edit	`CPDFPageEditViewController`	Enter the document pages editing of PDFs.
2-6	Document Info	`CPDFInfoViewController`	Enter the information list of PDF files.
2-7	Save	-	Document saved.
2-8	Save as Flattened	-	Save the document as a flattened copy.
2-9	Share	-	Share PDFs.
2-10	Open	-	Select PDFs.

Part 3:



The left image above shows the main UI components associated with the API of Annotations. The right image above shows the main UI components associated with the `ContentEditor` API. See the following details of the connection between UI components and APIs.

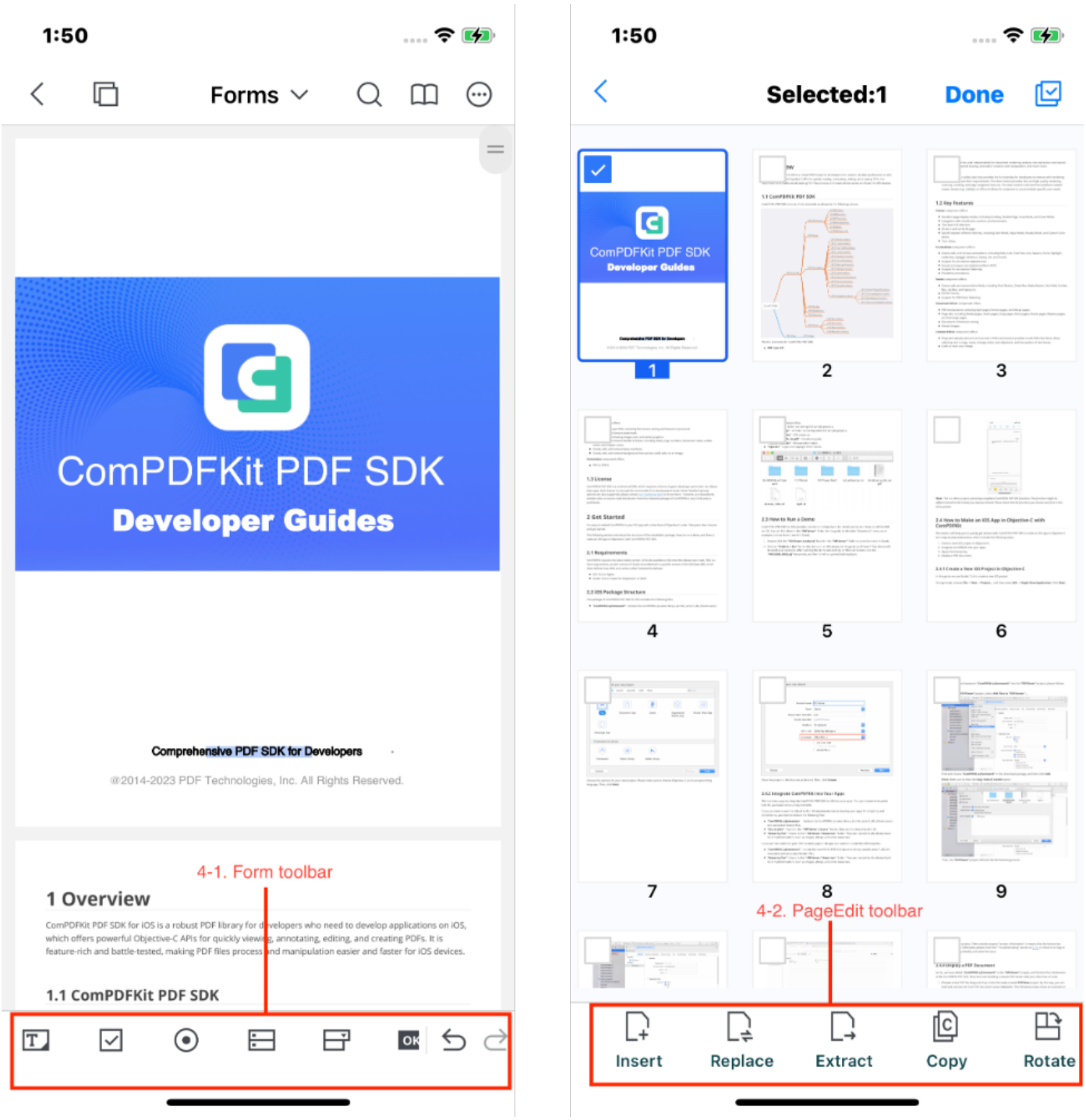
Number	Name	Functionality	Description
3-1	Annotation Toolbar	`CPDFAnnotationToolbar`	Set the properties of annotations and undo/redo the processing of annotations.
3-2	Properties	-	Open the property panel.
3-3	Edit Toolbar	`CPDFEditToolBar`	Set the properties of PDF text/images and undo/redo the processing of editing PDF content.

Here is the method to initialize the `CAnnotationManage` (Number 19) below:

```
// Initialize the CAnnotationManage object
  var annotationManage = CAnnotationManage.init(pdfListView: self.pdfListView!)

// Set the CAnnotationManage object to manage the currently selected annotations
annotationManage.setAnnotStyle(from: pdfListView.activeAnnotations)

// Set the CAnnotationManage object to manage the default annotations
annotationManage.setAnnotStyle(from: pdfListView.annotationMode)

// Set the CAnnotationManage object to adjust the properties of selected annotations
annotationManage.refreshPage(with: annotStyle.annotations)

// CPDFAnnotationToolbar: Initialize a CAnnotationManage object - annotationManage
// annotationManage: Manage the model data of annotation properties
let annotationBar = CPDFAnnotationToolBar.init(annotationManage: annotationManage)
```

```
// Initialize the CAnnotationManage object
CAnnotationManage *annotationManage = [[CAnnotationManage alloc]
initWithPDFView:pdfListView];

// Set the CAnnotationManage object to manage the currently selected annotations
[annotManage setAnnotStyleFromAnnotations:pdfListView.activeAnnotations];

// Set the CAnnotationManage object to manage the default annotations
[annotManage setAnnotStyleFromMode:pdfListView.annotationMode];

// Set the CAnnotationManage object to adjust the properties of selected annotations
[annotManage refreshPageWithAnnotations:annotStyle.annotations];

// CPDFAnnotationToolbar: Initialize a CAnnotationManage object - annotationManage
// annotationManage: Manage the model data of annotation properties
CPDFAnnotationToolbar *annotationToolbar = [[CPDFAnnotationToolbar alloc]
initAnnotationManage:annotationManage];
```

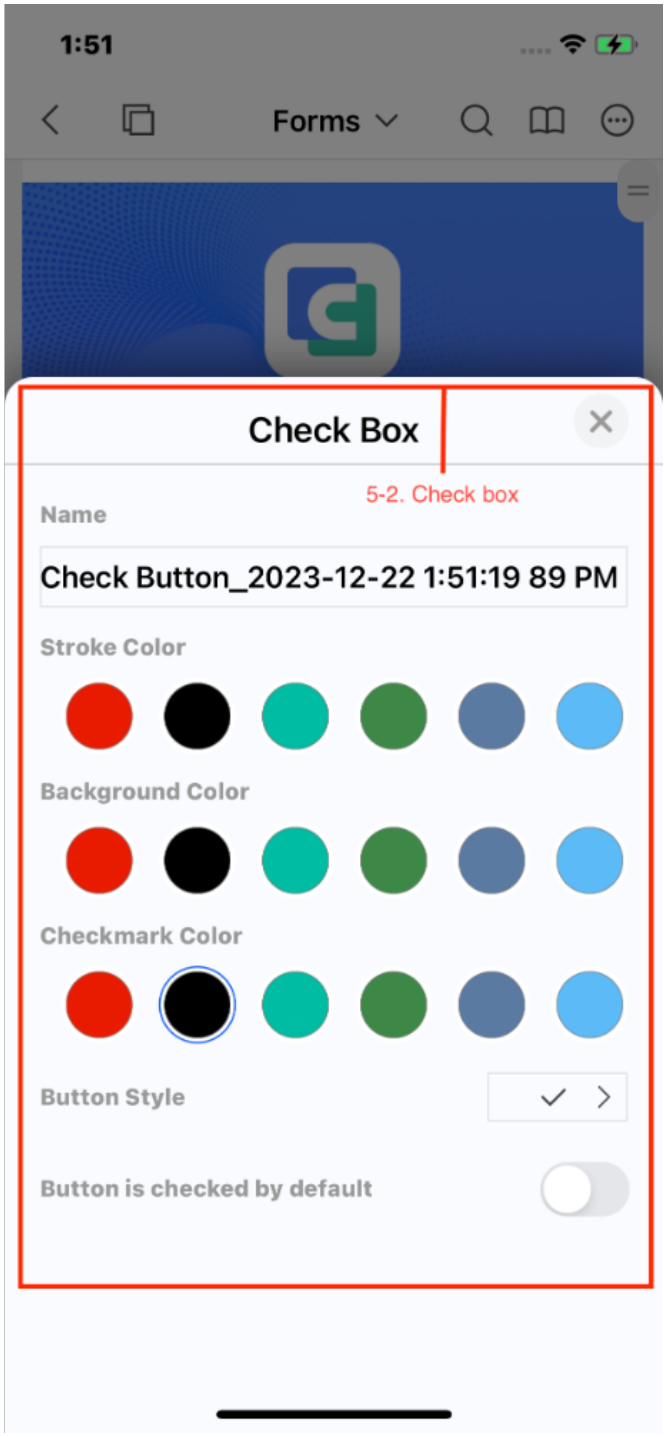
Part 4:



The left image above shows the main UI components associated with the API of Forms. The right image above shows the main UI components associated with the `DocsEditor` API. See the following details of the connection between UI components and APIs.

Number	Name	Functionality	Description
4-1	Form toolbar	`CPDFFormToolbar`	Set the properties of forms, and undo/redo the processing of form properties.
4-2	PageEdit toolbar	`CPageEditToolBar`	Edit PDF pages.

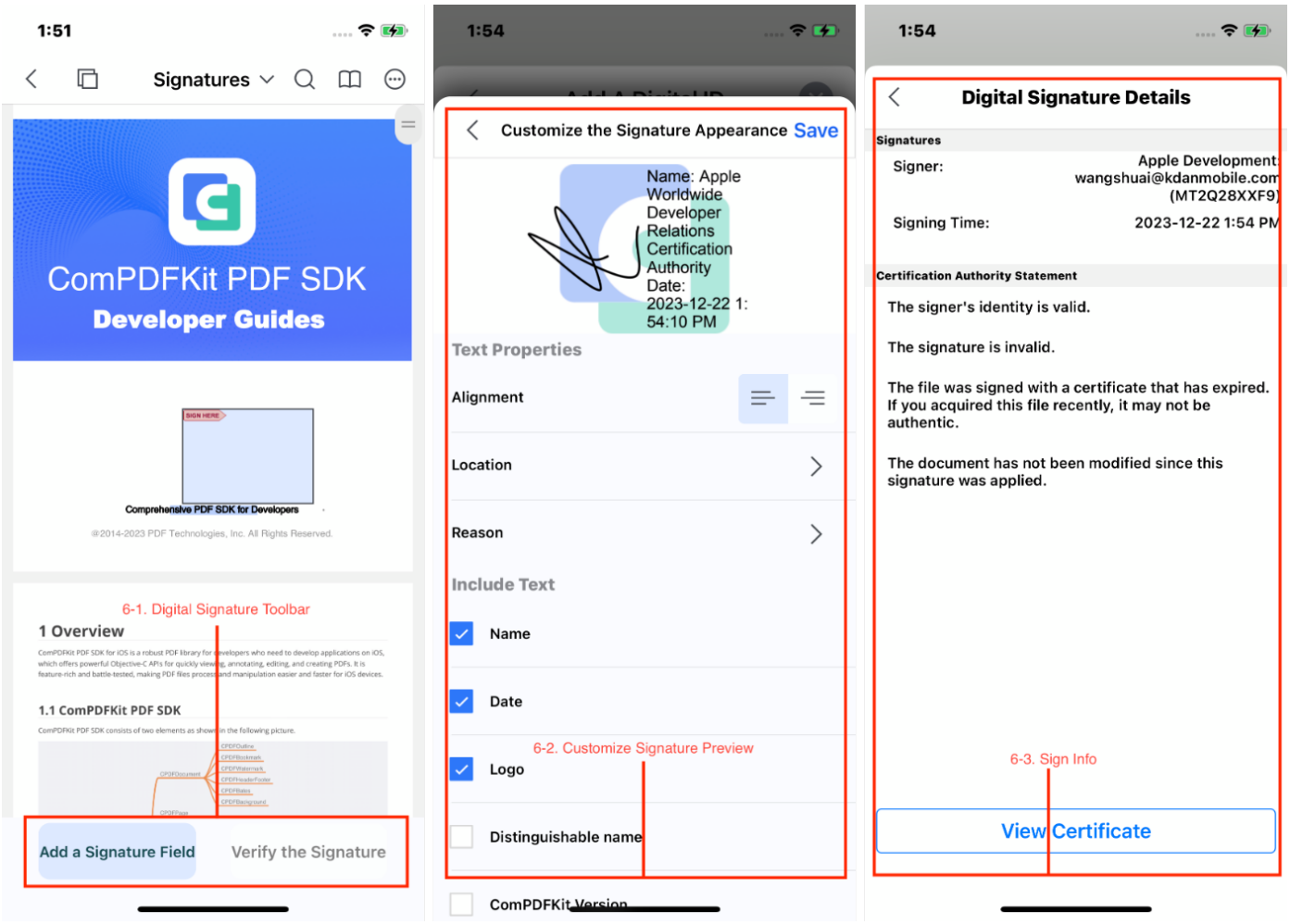
Part 5:



The left image above shows the main UI components associated with the `CPDFNoteViewController` API. The right image above shows the main UI components associated with the `CPDFFormCheckBoxViewController` API. See the following details of the connection between UI components and APIs.

Number	Name	Functionality	Description
5-1	Note	`CPDFNoteViewController`	Set the properties of Note annotations.
5-2	Check box	`CPDFFormCheckBoxViewController`	Set the properties of Check box form fields.

Part 6:



The picture above shows the UI components associated with the API of `Digital Signature`. The following table shows the details of the connection between UI components and APIs.

Number	Name	Functionality	Description
6-1	Digital Signature Toolbar	`CPDFDigitalSignatureToolBar`	The Digital Signature toolbar.
6-2	Customize Signature Preview	`CAddSignatureViewController`	Customize the digital signature appearance.
6-3	Sign Info	`CAddSignatureViewController`	Present the details of digital signatures.

2.7.3 How to Use the `CPDFListView` Class

`CPDFListView` is a subclass of `CPDFView` that includes operations for previewing, annotating, form filling, and content editing.

1. Feature Mode Switching: ToolModel

   To switch to a different feature mode, simply set the corresponding feature mode by `CPDFListView::toolModel`. Then, the `CPDFListView` will a display different UI. The table below shows the processing you can do with annotations and forms in the corresponding feature mode.

CToolModel	Description
CToolModelViewer	- Support selecting and copying text. - Support filling out and signing forms. - Do not support adding, selecting, moving, deleting annotations, and editing annotation properties. - Do not support adding, selecting, moving, deleting form fields, and editing form properties.
CToolModelEdit	- Do not support adding, selecting, moving, deleting annotations, and editing annotation properties. - Do not support form filling. - Do not support adding, selecting, moving, deleting form fields, and editing form properties.
CToolModelAnnotation	- Support selecting and copying text. - Support adding, selecting, moving, deleting annotations, and editing annotation properties. - Do not support form filling. - Do not support adding, selecting, moving, deleting form fields, and editing form properties.
CToolModelForm	- Support adding, selecting, moving, deleting form fields, and editing form properties. - Do not support selecting and copying text. - Do not support adding, selecting, moving, deleting annotations, and editing annotation properties. - Do not support form filling.
CToolModelPageEdit	- Do not support editing annotations and forms.

2. Adding Annotation or Form Mode: `CPDFViewAnnotationMode`

When entering annotation mode or form mode, simply set the corresponding annotation mode (The form is a kind of annotation) in `CPDFListView::annotationMode` to create specified annotations and forms in `CPDFListView`. The following table shows what the value is represented.

CPDFViewAnnotationMode	Description
CPDFViewAnnotationModeNone	Enter the annotation mode but do not specify an annotation type
CPDFViewAnnotationModeNote	Note Annotations
CPDFViewAnnotationModeHighlight	Markup Annotations: Highlight
CPDFViewAnnotationModeUnderline	Markup Annotations: Underline
CPDFViewAnnotationModeStrikeout	Markup Annotations: Strikeout
CPDFViewAnnotationModeSquiggly	Markup Annotations: Squiggly
CPDFViewAnnotationModeCircle	Shape Annotations: Circle
CPDFViewAnnotationModeSquare	Shape Annotations: Square
CPDFViewAnnotationModeArrow	Shape Annotations: Arrow
CPDFViewAnnotationModeLine	Shape Annotations: Line
CPDFViewAnnotationModeInk	Ink Annotations
CPDFViewAnnotationModePencilDrawing	Pencil Drawing
CPDFViewAnnotationModeFreeText	Free Text Annotations
CPDFViewAnnotationModeSignature	Signatures
CPDFViewAnnotationModeStamp	Stamp Annotations
CPDFViewAnnotationModeImage	Image Annotations
CPDFViewAnnotationModeLink	Link Annotations
CPDFViewAnnotationModeSound	Sound Annotations
CPDFViewFormModeText	Form Fields: Text
CPDFViewFormModeCheckBox	Form Fields: Check Box
CPDFViewFormModeRadioButton	Form Fields: Radio Button
CPDFViewFormModeCombox	Form Fields: Combo Boxes
CPDFViewFormModeList	Form Fields: List Box
CPDFViewFormModeButton	Form Fields: Push Button
CPDFViewFormModeSign	Form Fields: Signatures

Note:

- *Do not support scrolling the PDF `CPDFListView` after selecting an annotation type or selecting an annotation.*
- *`CPDFViewAnnotationModePencilDrawing` mode is only supported on iOS 13.0 or higher devices.*

3. Undo and Redo the Processing of Annotations and Forms

The undo and redo operations of forms and `CPDFListView`'s annotations, these operations actually involve observing the annotation properties using **KVO**. When initializing `CPDFListView`, simply call `CPDFListView::registerAsObserver`, and the `NSUndoManager` object will use the `keysForValuesToObserveForUndo` method to observe the relevant properties of the existing annotations and forms, as well as newly added annotations and forms.

```
func startObservingNotes(newNotes: [CPDFAnnotation]) {
        if self.notes == nil {
            self.notes = []
        }
        for note in newNotes {
            if self.notes?.contains(note) == false {
                self.notes?.append(note)
            }
            let keys: Set<String> = note.keysForValuesToObserveForUndo()
            for key in keys {
                note.addObserver(self, forKeyPath: key, options: [.new, .old], context:
&CPDFAnnotationPropertiesObservationContext)
            }
 }
```

```
- (void)startObservingNotes:(NSArray *)newNotes {
    if (!self.notes) {
        self.notes = [NSMutableArray array];
    }
    for (CPDFAnnotation *note in newNotes) {
        if (![self.notes containsObject:note]) {
            [self.notes addObject:note];
        }
        for (NSString *key in [note keysForValuesToObserveForUndo]) {
            [note addObserver:self forKeyPath:key options:(NSKeyValueObservingOptionNew |
NSKeyValueObservingOptionOld) context:&CPDFAnnotationPropertiesObservationContext];
        }
    }
}
```

Note: *When switching documents and removing comments or forms, be sure to remove the KVO observers for the comment's property.*

The following code shows how to undo and redo edits of Annotations and forms

```
@objc func buttonItemClicked_undo(_ button: UIButton) {
        if self.annotManage?.pdfListView?.undoPDFManager != nil &&
((self.annotManage?.pdfListView?.canUndo()) == true) {
            self.annotManage?.pdfListView?.undoPDFManager?.undo()
        }
    }

    @objc func buttonItemClicked_redo(_ button: UIButton) {
        if self.annotManage?.pdfListView?.undoPDFManager != nil &&
(self.annotManage?.pdfListView?.canRedo()) == true {
            self.annotManage?.pdfListView?.undoPDFManager?.redo()
        }
    }
```

```
- (void)buttonItemClicked_undo:(UIButton *)button {
    if(self.annotManage.pdfListView.undoPDFManager && [self.annotManage.pdfListView
canUndo]) {
        [self.annotManage.pdfListView.undoPDFManager undo];
    }
}

- (void)buttonItemClicked_redo:(UIButton *)button {
    if(self.annotManage.pdfListView.undoPDFManager && [self.annotManage.pdfListView
canRedo]) {
        [self.annotManage.pdfListView.undoPDFManager redo];
    }
}
```

4. Implementing the Delegate Method

About Implementing the delegate method, refer to the following method in the ***"CPDFViewController"*** file.

```
@optional

// Options of the context menu
    @objc optional func PDFListView(_ pdfListView: CPDFListView, customizeMenuItems
menuItems: [UIMenuItem], forPage page: CPDFPage, forPagePoint pagePoint: CGPoint) ->
[UIMenuItem]

// Click on a blank area
    @objc optional func PDFListViewPerformTouchEnded(_ pdfListView: CPDFListView)

// Switch feature modes
    @objc optional func PDFListViewChangedToolMode(_ pdfListView: CPDFListView,
forToolMode toolMode: Int)

// Switch annotation types
    @objc optional func PDFListViewChangedToolMode(_ pdfListView: CPDFListView,
forToolMode toolMode: Int)
```

```
// Annotations: Changes occur when selecting an annotation
    @objc optional func PDFListViewChangeatioActiveAnnotations(_ pdfListView:
CPDFListView, forActiveAnnotations annotations: [CPDFAnnotation])

// Annotations: Changes occur
    @objc optional func PDFListViewAnnotationsOperationChange(_ pdfListView:
CPDFListView)

// Editing annotations
    @objc optional func PDFListViewEditNote(_ pdfListView: CPDFListView, forAnnotation
annotation: CPDFAnnotation)

// Set the properties of annotations
    @objc optional func PDFListViewEditProperties(_ pdfListView: CPDFListView,
forAnnotation annotation: CPDFAnnotation)

// Play sound annotation
    @objc optional func PDFListViewPerformPlay(_ pdfView: CPDFListView, forAnnotation
annotation: CPDFSoundAnnotation)

// Cancle the adding of sound annotations
    @objc optional func PDFListViewPerformCancelMedia(_ pdfView: CPDFListView, atPoint
point: CGPoint, forPage page: CPDFPage)

// Sound annotations: Start to record
    @objc optional func PDFListViewPerformRecordMedia(_ pdfView: CPDFListView, atPoint
point: CGPoint, forPage page: CPDFPage)

// Sound annotations: If it's recording
    @objc optional func PDFListViewerTouchEndedIsAudioRecordMedia(_ pdfListView:
CPDFListView) -> Bool

// Add Stamp annotation
    @objc optional func PDFListViewPerformAddStamp(_ pdfView: CPDFListView, atPoint
point: CGPoint, forPage page: CPDFPage)

// Add Image annotation
    @objc optional func PDFListViewPerformAddImage(_ pdfView: CPDFListView, atPoint
point: CGPoint, forPage page: CPDFPage)

// Signature widget add signature
    @objc optional func PDFListViewPerformSignatureWidget(_ pdfView: CPDFListView,
forAnnotation annotation: CPDFSignatureWidgetAnnotation)

// Set properties for text area
    @objc optional func PDFListViewContentEditProperty(_ pdfListView: CPDFListView,
point: CGPoint)
```

```
@optional
```

```
// Options of the context menu
- (NSArray<UIMenuItem *> *)PDFListView:(CPDFListView *)pdfListView customizeMenuItems:
(NSArray *)MenuItems forPage:(CPDFPage *)page forPagePoint:(CGPoint)pagePoint;

// Click on a blank area
- (void)PDFListViewPerformTouchEnded:(CPDFListView *)pdfListView;

// Switch feature modes
- (void)PDFListViewChangedToolMode:(CPDFListView *)pdfListView forToolMode:
(CToolModel)toolMode;

// Switch annotation types
- (void)PDFListViewChangedAnnotationType:(CPDFListView *)pdfListView forAnnotationMode:
(CPDFViewAnnotationMode)annotationMode;

// Annotations: Changes occur when selecting an annotation
- (void)PDFListViewChangeatioActiveAnnotations:(CPDFListView *)pdfListView
forActiveAnnotations:(NSArray<CPDFAnnotation *> *)annotations;

// Annotations: Changes occur
- (void)PDFListViewAnnotationsOperationChange:(CPDFListView *)pdfListView;

// Editing annotations
- (void)PDFListViewEditNote:(CPDFListView *)pdfListView forAnnotation:(CPDFAnnotation
*)annotation;

// Set the properties of annotations
- (void)PDFListViewEditProperties:(CPDFListView *)pdfListView forAnnotation:
(CPDFAnnotation *)annotation;

// Play sound annotation
- (void)PDFListViewPerformPlay:(CPDFListView *)pdfView forAnnotation:(CPDFSoundAnnotation
*)annotation;

// Cancle the adding of sound annotations
- (void)PDFListViewPerformCancelMedia:(CPDFListView *)pdfView atPoint:(CGPoint)point
forPage:(CPDFPage *)page;

// Sound annotations: Start to record
- (void)PDFListViewPerformRecordMedia:(CPDFListView *)pdfView atPoint:(CGPoint)point
forPage:(CPDFPage *)page;

// Sound annotations: If it's recording
- (BOOL)PDFListViewerTouchEndedIsAudioRecordMedia:(CPDFListView *)pdfListView;

// Add Stamp annotation
- (void)PDFListViewPerformAddStamp:(CPDFListView *)pdfView atPoint:(CGPoint)point
forPage:(CPDFPage *)page;

// Add Image annotation
- (void)PDFListViewPerformAddImage:(CPDFListView *)pdfView atPoint:(CGPoint)point
forPage:(CPDFPage *)page;
```

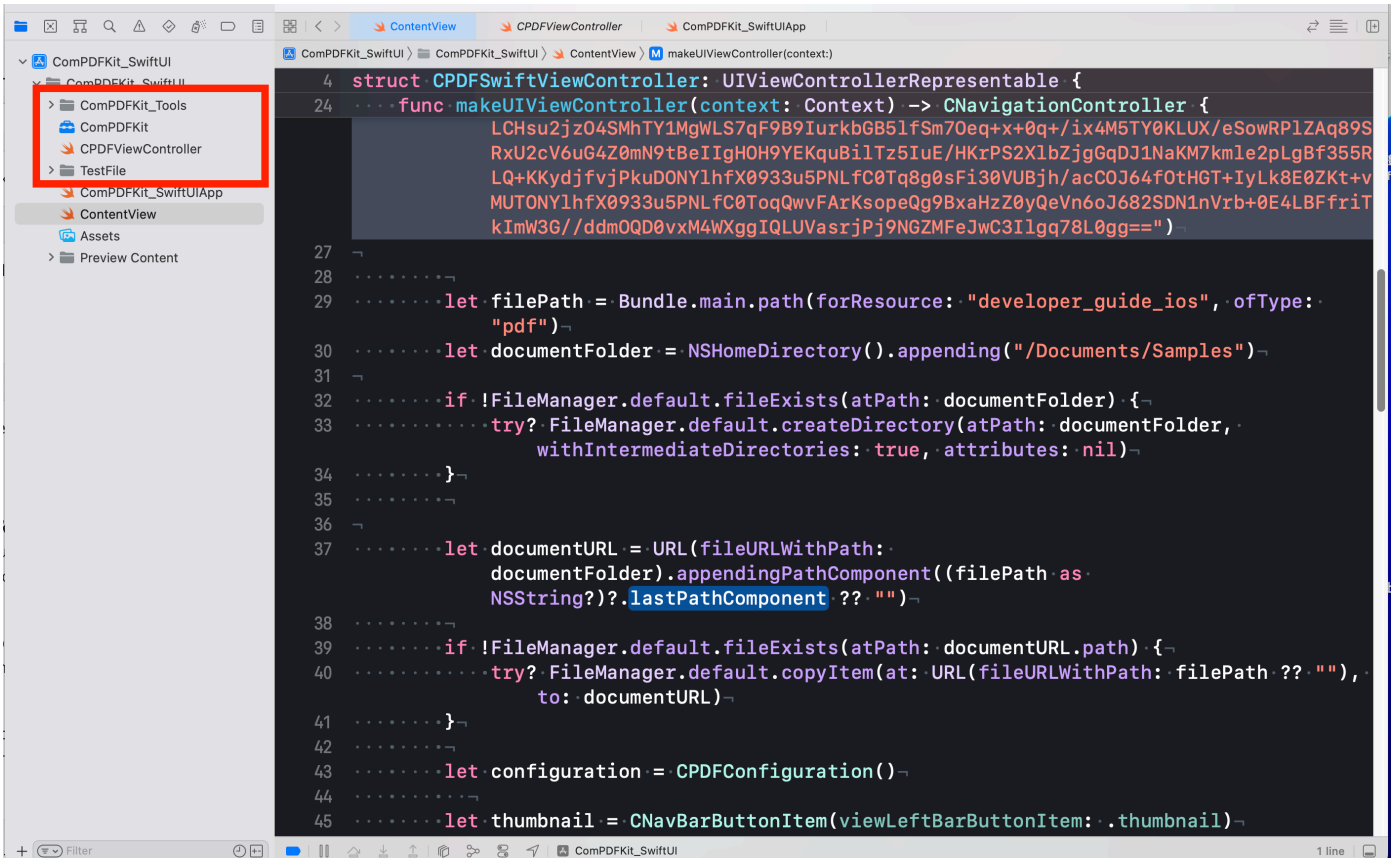
```
// Signature widget add signature
- (void)PDFListViewPerformSignatureWidget:(CPDFListView *)pdfView forAnnotation:
(CPDFSignatureWidgetAnnotation *)annotation;

// Set properties for text area
- (void)PDFListViewContentEditProperty:(CPDFListView *)pdfListView point:(CGPoint)point;
```

Note: *If you need to toggle or deselect annotations, you need to call* `CPDFListView::updateActiveAnnotations` *in order for* `CPDFListView::PDFListViewChangeatioActiveAnnotations: forActiveAnnotations:` *to respond.*

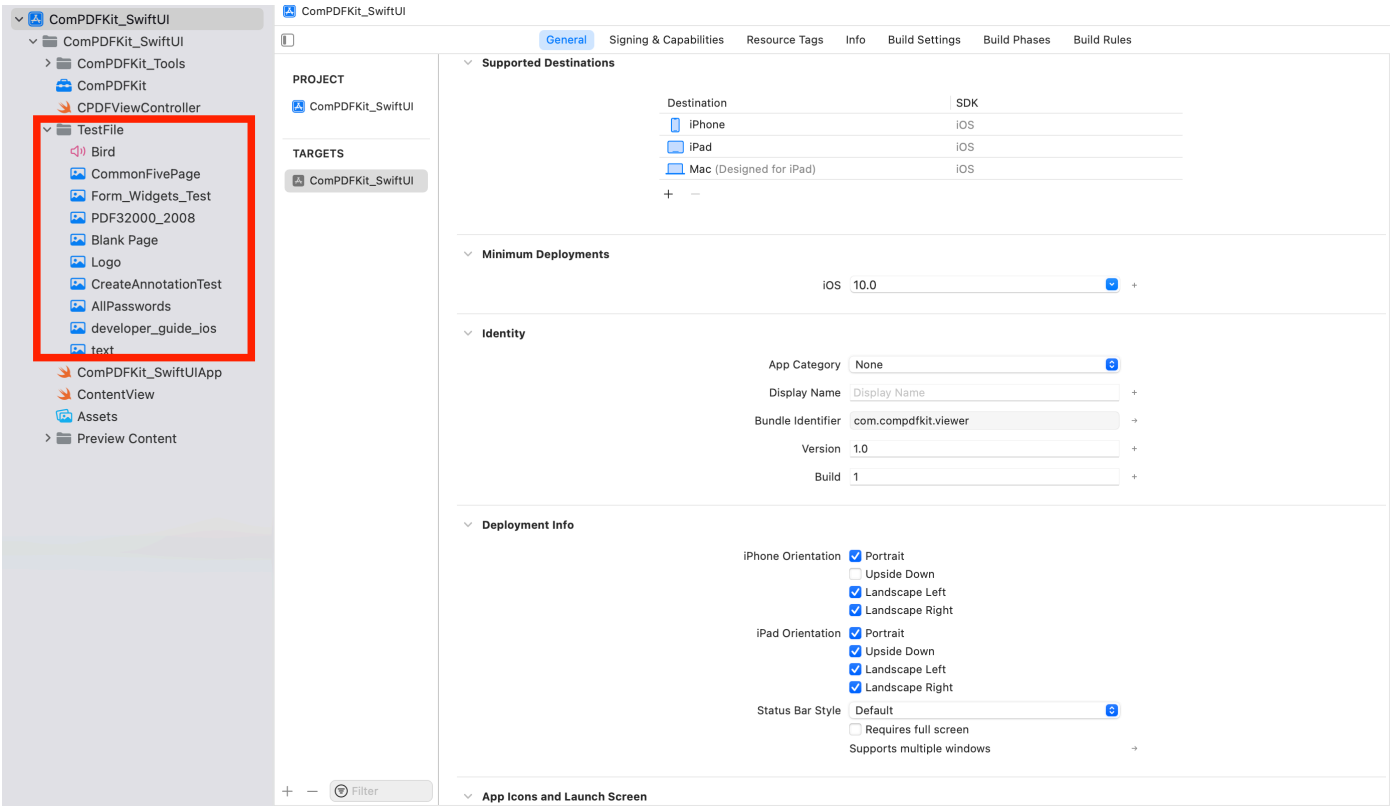
2.7.4 SwiftUI

Since version 10.1 for iOS, ComPDFKit exposes specific APIs to use ComPDFKit with SwiftUI out of the box. This makes dealing with ComPDFKit in a SwiftUI app easier, and it doesn't require you to wrap `CPDFViewController.swift` yourself if you want to go the SwiftUI route.

1. Import resource file, ***"ComPDFKit_Tools"*** for iOS sample projects comes with a default UI design, including the basic UI for the app and the feature modules UI, which are implemented using ComPDFKit PDF SDK and are shipped. Also included is a `CPDFViewController` view controller that contains ready-to-use UI module implementations.



2. Add PDF documents to your application by dragging and dropping them into your project. In the dialog that displays, select Finish to accept the default integration options. You can use the documents in the "TestFile" folder as an example.



3. Bridging `CPDFSwiftViewController` to SwiftUI

First, declare the `CPDFSwiftViewController` struct — which conforms to the `UIViewControllerRepresentable` protocol — so that you can bridge from UIKit to SwiftUI.

```
struct CPDFSwiftViewController: UIViewControllerRepresentable {
    @Environment(\.presentationMode) var presentationMode: Binding<PresentationMode>

    class Coordinator: NSObject,CPDFViewBaseControllerDelete {
        var myview: CPDFSwiftViewController
        init(_ myview: CPDFSwiftViewController) {
            self.myview = myview
        }

        // MARK: - CPDFViewBaseControllerDelete
        func PDFViewBaseControllerDissmiss(_ baseControllerDelete:
CPDFViewBaseController) {
            baseControllerDelete.dismiss(animated: true)
        }

    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }

    func makeUIViewController(context: Context) -> CNavigationController {

        CPDFKit.setLicenseKey("", secret: "")
```

```
        let filePath = Bundle.main.path(forResource: "developer_guide_ios", ofType:
"pdf")
        let documentFolder = NSHomeDirectory().appending("/Documents/Samples")

        if !FileManager.default.fileExists(atPath: documentFolder) {
            try? FileManager.default.createDirectory(atPath: documentFolder,
withIntermediateDirectories: true, attributes: nil)
        }


        let documentURL = URL(fileURLWithPath:
documentFolder).appendingPathComponent((filePath as NSString?)?.lastPathComponent ?? "")

        if !FileManager.default.fileExists(atPath: documentURL.path) {
            try? FileManager.default.copyItem(at: URL(fileURLWithPath: filePath ?? ""),
to: documentURL)
        }

        let configuration = CPDFConfiguration()

        let thumbnail = CNavBarButtonItem(viewLeftBarButtonItem: .thumbnail)
        let back = CNavBarButtonItem(viewLeftBarButtonItem: .back)
        let search = CNavBarButtonItem(viewRightBarButtonItem: .search)
        let bota = CNavBarButtonItem(viewRightBarButtonItem: .bota)
        let more = CNavBarButtonItem(viewRightBarButtonItem: .more)

        configuration.showleftItems = [back, thumbnail]
        configuration.showRightItems = [search, bota, more]


        let vc = CPDFViewController(filePath: documentURL.path, password: nil,
configuration: configuration)
        vc.delegate = context.coordinator;
        let navController = CNavigationController(rootViewController: vc)

        return navController
    }

    func updateUIViewController(_ uiViewController: CNavigationController, context:
UIViewControllerRepresentableContext<CPDFSwiftViewController>) {
        // Update the view controller.
    }
}
```

4. Using `CPDFSwiftViewController` in SwiftUI

And finally, use `CPDFSwiftViewController` in your SwiftUI content view:

```
struct ContentView: View {
    @State var isPresented: Bool = false

    var body: some View {
```

```
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")

                .toolbar {

                    ToolbarItem(placement: .bottomBar) {
                        Button("Click to open the sample PDF") {
                            isPresented = true
                        }
                    }
                }
                .sheet(isPresented: $isPresented) {
                    CPDFSwiftViewController()
                }

        }
        .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

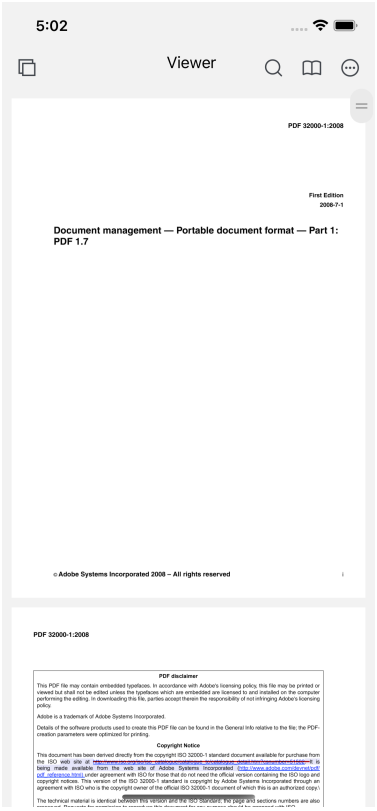
2.7.5 Learn More

1. How to Handle PDF Document Loading

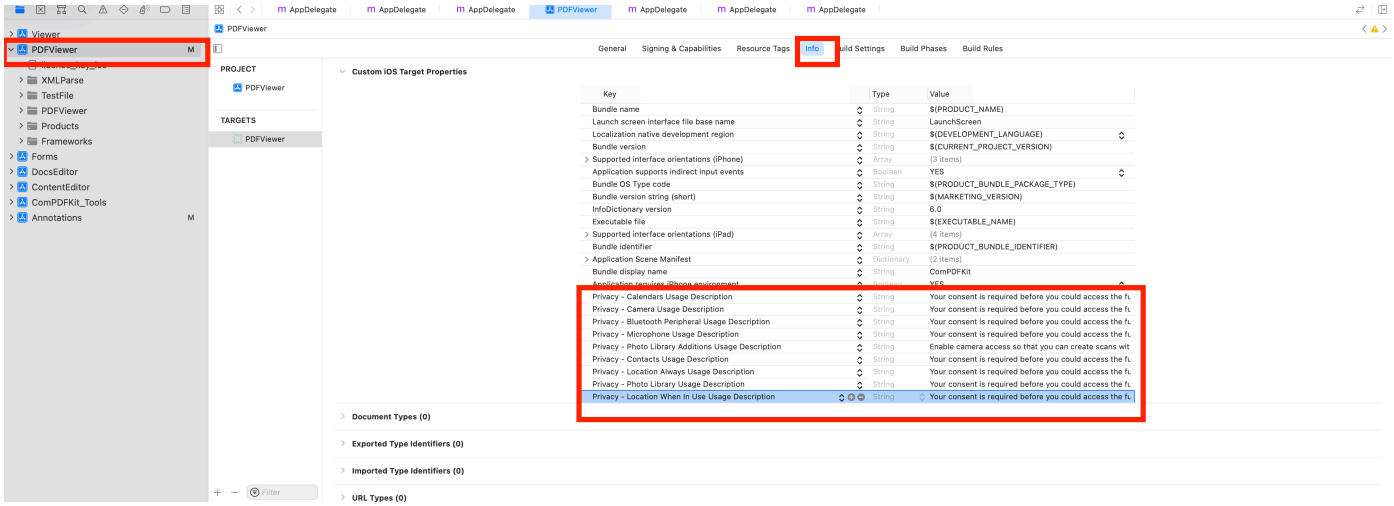
   To handle PDF document loading, refer to the following method in the ***"CPDFViewBaseController.m"*** file.

   ```
       func reloadDocument(withFilePath filePath: String, password: String?, completion: @escaping (Bool) -> Void)
   ```

   ```
   - (void)reloadDocumentWithFilePath:(NSString *)filePath password:(nullable NSString *)password completion:(void (^)(BOOL result))completion;
   ```



2. To protect user privacy, before accessing the sensitive privacy data, you need to find the "***Info***" configuration in your iOS 10.0 or higher iOS project and configure the relevant privacy terms as shown in the following picture.




```
<key>NSCameraUsageDescription</key>
<string>Your consent is required before you could access the function.</string>

<key>NSMicrophoneUsageDescription</key>
<string>Your consent is required before you could access the function.</string>

<key>NSPhotoLibraryAddUsageDescription</key>
<string>Your consent is required before you could access the function.</string>

<key>NSPhotoLibraryUsageDescription</key>
<string>Your consent is required before you could access the function.</string>
```

2.8 Samples

The Samples use preset parameters, documentation, and modular code examples to call the API of ComPDFKit PDF SDK for each function without UI interaction or parameter settings. The functions include creating, getting, and deleting various types of annotations and forms, extracting text and images, encrypting and decrypting documents, adding watermarks and bates numbers, and more.

These projects not only demonstrate the best practices for each function but also provide detailed introductions. The impact of each function on PDF documents can be observed in the output directory. With the help of the Samples, you can quickly learn how to use the functions you need and apply them to your projects.

Name	Description
Bookmark	Create a new bookmark, access the existing bookmark, and jump to a specific bookmark.
Outline	Create a new outline, and get existing outline information.
PDFToImage	Convert PDF pages to PNG.
TextSearch	Perform full-text search and highlight keywords.
Annotation	Print the annotation list information, set the annotations (including markup, note, ink, free text, circle, square, line, stamp, and sound annotations), and delete the annotations.
AnnotationImportExport	Export and import annotations with an xfdf file.
InteractiveForms	Print form list information, set up interactive forms (including text, checkbox, radio button, button, list, combo boxes, signing and deleting forms), and fill out form information.
PDFPage	Manipulate PDF pages, including inserting, splitting, merging, rotating, and replacing, etc.
ImageExtract	Extract images from a PDF document.
TextExtract	Extract the text of a PDF.
DocumentInfo	Extract the information of PDF files like the author, created time, etc.
Watermark	Create text/image watermarks and delete watermarks.
Background	Create a color/image background and delete the background.
HeaderFooter	This sample shows how to add and remove headers and footers.
Bates	Create and remove bates numbers.
PDFRedact	Create redaction to remove sensitive information or private data, which cannot be viewed and searched once applied.
Encrypt	Set passwords to encrypt PDFs and set document permissions. Allow decrypting PDFs.
PDFA	Convert PDF to PDF/A-1a and PDF/A-1b.
Flatten	Flatten PDF annotations and forms, and merge all layouts as one layout.
DigitalSignatures	Create, fill, and verify the signatures and certificates. Read the details of signatures and certificates. Remove digital signatures.

2.9 ARC Compatibility

ComPDFKit PDF SDK requires non-ARC. If you wish to use ComPDFKit PDF SDK in an ARC project, just add the -fno-objc-arc compiler flag. To do this, go to the Build Phases tab in your target settings, open the Compile Sources group, and double-click and type -fno-objc-arc into the popover.

3 Guides

If you're interested in all of the features mentioned in Overview section, please go through our guides to quickly add PDF viewing, annotating, and editing to your application. The following sections list some examples to show you how to add document functionalities to iOS apps using our Swift and Objective-C APIs.

3.1 Basic Operations

3.1.1 Overview

There are some common basic operations when handling documents.

Guides for Basic Operations

- **Open a Document**

  Open a local PDF document or create a new PDF document from scratch.

- **Save a Document**

  Save the document to the original path or save it as a new document.

- **Document Information**

  View information such as the file creator, creation time, modification time, and more.

3.1.2 Open a Document

ComPDFKit supports opening local PDF documents or creating new ones.

Open a Local PDF Document

The steps to open a local PDF document using a file path are as follows:

1. Obtain the local file path.
2. Initialize a `CDFDocument` object using the file path.

This example shows how to open a local PDF document:

```swift
let url = URL(fileURLWithPath: "File Path")
// Initialize a `CPDFDocument` object using the PDF file path.
let document = CPDFDocument(url: url)

if let error = document?.error, error._code != CPDFDocumentPasswordError {

}
// For encrypted documents, it is necessary to use a password to decrypt them.
if document?.isLocked == true {
    document?.unlock(withPassword: "password")
}
```

```objc
NSURL *url = [NSURL fileURLWithPath:@""];
// Initialize a `CPDFDocument` object using the PDF file path.
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

if (document.error &&
    document.error.code == CPDFDocumentPasswordError) {

}
// For encrypted documents, it is necessary to use a password to decrypt them.
if (document.isLocked) {
    [document unlockWithPassword:@"password"];
}
```

Create a New PDF Document

This example shows how to create a new PDF document:

```swift
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)
```

```objc
NSURL *url = [NSURL fileURLWithPath:@"File Path"];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
```

By default, a newly created document doesn't contain any pages. Please refer to the "Document Editing" functionality to learn how to create new pages and add existing pages to the document.

Explanation of Open Document Status

This is the explanation of opening document status:

Error Code	Description
CPDFDocumentUnknownError	Unknown error.
CPDFDocumentFileError	File not found or cannot be opened.
CPDFDocumentFormatError	Format Error: not a PDF or corrupted.
CPDFDocumentPasswordError	Password required or incorrect password.
CPDFDocumentSecurityError	Unsupported security scheme.
CPDFDocumentPageError	Page not found or content error.

3.1.3 Save a Document

ComPDFKit supports incremental saving and full saving.

When the document is saved to the original path, the PDF document will be saved incrementally, meaning all changes will be appended to the file. This can significantly speed up the saving process for large files. However, it results in an increase in document size with each save.

When the document is saved to a new path, the PDF document will undergo non-incremental saving. This entails overwriting the entire document instead of appending changes at the end.

This example shows how to save a document by incremental saving and full saving :

```
// Incrementally save the document object to the current path.
document?.write(to: newFilePath)

// Save the document object to the current path in a non-incremental manner.
document?.write(to: surl, withOptions: options)
```

```
// Incrementally save the document object to the current path.
[document writeToURL:newFilePath];

// Save the document object to the current path in a non-incremental manner.
[document writeToURL:surl withOptions:options];
```

3.1.4 Document Information

This example shows how to get document information:

```swift
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

if let documentAttributes = document?.documentAttributes() {
    let title = documentAttributes[.titleAttribute] as? String // Document title.
    let author = documentAttributes[.authorAttribute] as? String // Document author.
    let subject = documentAttributes[.subjectAttribute] as? String // Document subject
    let creator = documentAttributes[.creatorAttribute] as? String // Application name
that created the document
    let keywords = documentAttributes[.keywordsAttribute] as? String // Document
keywords.
    let creationDate = documentAttributes[.creationDateAttribute] as? String // Document
creation date
    let modificationDate = documentAttributes[.modificationDateAttribute] as? String //
Document last modified date
}
```

```objc
NSURL *url = [NSURL fileURLWithPath:@"File Path"];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
NSDictionary *documentAttributes = [document documentAttributes];
NSString *title = documentAttributes[CPDFDocumentTitleAttribute];           // Document
title.
NSString *author = documentAttributes[CPDFDocumentAuthorAttribute];         // Document
author.
NSString *subject = documentAttributes[CPDFDocumentSubjectAttribute];       // Document
subject
NSString *creator = documentAttributes[CPDFDocumentCreatorAttribute];      // Application
name that created the document
NSString *keywords = documentAttributes[CPDFDocumentKeywordsAttribute];     // Document
keywords.
NSString *creationDate = documentAttributes[CPDFDocumentCreationDateAttribute]; //
Document creation date
NSString *modificationDate = documentAttributes[CPDFDocumentModificationDateAttribute];
// Document last modified date
```

3.1.5 Font Management

ComPDFKit PDF SDK supports reading the existing font families and their styles on your device, and setting them as fonts for annotations, forms, watermarks, headers, footers, bates stamps, and many other functionalities. This will help you design aesthetically pleasing PDF documents or adjust and refine your PDF documents with fonts that comply with certain specifications.

When setting fonts using font management, you need to:

1. Retrieve all font family names available in the system.
2. Choose the font you need and obtain the style names of the font family.
3. After selecting the style name, obtain the `CPDFFont` object based on the font family name and style name.
4. The `CPDFFont` object can then be used to set the font.

5. Set the application-side font UIFont through the `CPDFFont` object.

Here is a code example:

```
// Retrieve all font family names available in the system
let fonts: [String] = CPDFFont.familyNames

// Obtain the list of font styles corresponding to the font family name and choose a font style
let fontStyles = CPDFFont.fontNames(forFamilyName: "familyName")

// Obtain the CPDFFont object based on the font family and font style
let font =  CPDFFont(familyName: "familyName", fontStyle: "fontStyle")

// Apply the name of the CPDFFont object to the functionality that needs to set the font.
// For specific setting property methods, refer to the documentation of the corresponding functionality.
let anotation = CPDFLineAnnotation(document: document)
anotation.cFont = font

// Set the UIFont on the application side using the CPDFFont object
let appleFont = UIFont.init(name: CPDFFont.convertAppleFont(cFont ?? CPDFFont(familyName: "Helvetica", fontStyle: "")) ?? "Helvetica",
```

```
// Retrieve all font family names available in the system
NSArray *fonts = [CPDFFont familyNames];

// Obtain the list of font styles corresponding to the font family name and choose a font style
NSArray *fontStyles = [CPDFFont fontNamesForFamilyName:@"familyName"];

// Obtain the CPDFFont object based on the font family and font style
CPDFFont *font = [[CPDFFont alloc] initWithFamilyName:@"familyName"
fontStyle:@"fontStyle"];

// Apply the name of the CPDFFont object to the functionality that needs to set the font.
// For specific setting property methods, refer to the documentation of the corresponding functionality.
CPDFLineAnnotation *anotation = [[CPDFLineAnnotation alloc] initWithDocument:document];
[anotation setCFont:font];

// Set the UIFont on the application side using the CPDFFont object
 NSString *fontName = [CPDFFont convertAppleFont:cFont];
 UIFont *appleFont = [[UIFont fontWithName:fontName size:18];
```

About Font Family and Style Names

1. Font Family:

   Font Family refers to the group or series name of a font, typically representing a set of fonts that share a common design style.

For example, the Helvetica font family includes various styles such as Helvetica Regular, Helvetica Bold, and Helvetica Italic, all of which belong to the Helvetica font family.

2. Font Style:

   Font Style refers to the specific style or variant name of a font. It is commonly used to differentiate between different font styles within the same font family, such as bold, italic, regular, etc.

   Taking the Helvetica font family as an example, Regular, Bold, Italic, etc., are all different style names.

Import Font

The fonts currently in use are obtained from the device's system, and the font collections available may vary across devices in different regions. You can enrich the available font styles by using the font import interface.

To import fonts, follow the steps below:

1. Copy the fonts you need to a designated folder path.
2. Use `CPDFFont.setImportDir(path, isContainSysFont: true)` to set the directory where your fonts are stored.
3. Initialize the SDK.

```
// Copy fonts from the Bundle directory to sandbox.
let dir = Bundle.main.path(forResource: "Font", ofType: nil) ?? ""
let fileManager = FileManager.default
let documentDirectory = fileManager.urls(for: .documentDirectory, in:
.userDomainMask).first!
let destinationPath = documentDirectory.appendingPathComponent("Font")

do {
    if fileManager.fileExists(atPath: destinationPath.path) {
        try fileManager.removeItem(at: destinationPath)
    }

    try fileManager.copyItem(atPath: dir, toPath: destinationPath.path)

    // Set the specified font folder
    // isContainSysFont determine whether to include system fonts
    CPDFFont.setImportDir(destinationPath.path, isContainSysFont: true)

    // Initialize the SDK
    CPDFKit.verify(withKey: "Your license key")
} catch {
    print("Error copying Font directory: \(error)")
}
```

```
// Copy fonts from the Bundle directory to sandbox.
NSString *dir = [[NSBundle mainBundle] pathForResource:@"Font" ofType:nil] ?: @"";
```

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentDirectory = [[fileManager URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] firstObject];
NSURL *destinationPath = [documentDirectory URLByAppendingPathComponent:@"Font"];

NSError *error = nil;
if ([fileManager fileExistsAtPath:[destinationPath path]]) {
    if (![fileManager removeItemAtURL:destinationPath error:&error]) {
        NSLog(@"Error removing existing directory: %@", error);
    }
}

if (![fileManager copyItemAtPath:dir toPath:[destinationPath path] error:&error]) {
    NSLog(@"Error copying Font directory: %@", error);
} else {

    // Set the specified font folder
    // isContainSysFont determine whether to include system fonts
    [CPDFFont setImportFontDir:[destinationPath path] isContainSysFont:YES];

    // Initialize the SDK
    [CPDFKit verifyWithKey:@"Your license key"];
}
```

3.2 Viewer

3.2.1 Overview

ComPDFKit for iOS includes a high-quality PDF viewer that's fast, precise, and feature-rich. It offers developers a way to quickly embed a highly configurable PDF viewer in any iOS application.

Benefits of ComPDFKit PDF Viewer

- **Display Modes**: Freely switch between single-page or double-page view, page flipping or scrolling modes, adapting to different reading scenarios.
- **Multiple Themes:** Choose themes suitable for work, night reading, prolonged screen time, or creating custom themes.
- **PDF Navigation:** Navigate directly to specific locations, or use bookmarks and outlines.
- **Extensibility:** Easily add features such as annotations, forms, signatures, etc., to the PDF viewer.
- **Fast UI Integration:** Achieve rapid integration and customization through extensible UI components.

Guides for Viewer

- Display Modes

  Choose between single-page, double-page, book, flip, or scroll modes, or set cropping modes.
- Page Navigation

  Efficiently navigate to different locations in the document through simple code.

- **Outlines**

  The outline provides a hierarchical structure of the document, allowing users to efficiently locate and navigate content.

- **Bookmarks**

  Bookmarks enable users to create personalized markers in the document and efficiently navigate to those bookmarked locations.

- **Text Search and Selection**

  Locate the position of keywords in the document, and copy or mark the text of interest.

- **Text Reflow**

  Reorganize the text to fit the device screen size.

- **Zooming**

  Adjust the degree of zoom in or zoom out of a page in a PDF document to fit the user's visual preferences or device screen size.

- **Themes**

  Meet various reading scenarios with a range of preset themes and customizable themes.

- **Render**

  Perform custom drawing on the PDF page.

- **Custom Menu**

  Implement quick actions for various scenarios through customizable context menus.

- **Highlight Form Fields and Hyperlinks**

  Efficiently locate forms and hyperlinks to enhance interaction efficiency and prevent information oversight.

- **Get the Selected Content**

  Real-time retrieval of selected text in the viewer.

3.2.2 Display Modes

ComPDFKit supports single-page, double-page, and book mode arrangements for documents, facilitating reading through both page flipping and scrolling methods. Additionally, it offers options to set cropping mode.

Set the Display Mode

This example shows how to set the display mode to single continuous:

```
var pdfview = CPDFView()
pdfview.displayTwoUp = false
pdfview.displaysAsBook = false
pdfview.layoutDocumentView()
```

```
CPDFView pdfview;
pdfview.displayTwoUp = NO;
pdfview.displaysAsBook = NO;
[pdfview layoutDocumentView];
```

Explanation of Display Mode Types

The table below displays Reading Mode types, descriptions, and their corresponding parameter names.

Type	Description	Parameter Name
Page continuous	Set whether to add spacing between adjacent pages.	pdfview.displaysPageBreaks
Double	Set whether to display two pages of the document side by side.	pdfview.displayTwoUp
Book	Set whether to have the cover occupy a separate line in double-page mode.	pdfview.displaysAsBook
Scrolling Direction	Set whether to use vertical scrolling or horizontal scrolling.	pdfView.displayDirection

Set the Crop Mode

Crop mode refers to a feature in PDF documents that allows the cropping of pages to alter their visible area or dimensions. Crop mode enables users to define the display range of a page, making it visually more aligned with specific requirements.

This example shows how to set the crop mode:

```
var pdfview = CPDFView()
pdfview.displayCrop = true
pdfview.layoutDocumentView()
```

```
CPDFView pdfview;
pdfview.displayCrop = YES;
[pdfview layoutDocumentView];
```

3.2.3 Page Navigation

After loading and displaying a PDF document in the `CPDFViewer`, users can navigate to different pages or positions within the document by adjusting the display area.

This example shows how to navigate to specific pages and positions within pages:

```
// Navigate to the first page.
pdfview.go(toPageIndex: 0, animated: false)

// Navigate to a specific position on the first page.
if let page = pdfView.document.page(at: 0) {
    pdfview.go(to: CGRect(x: 100, y: 100, width: 100, height: 100), on: page, animated:
true)
}
```

```
// Navigate to the first page.
[pdfview goToPageIndex:0 animated:NO];

// Navigate to a specific position on the first page.
[self.pdfView goToRect:CGRectMake(100, 100, 100, 100) onPage:[pdfView.document
pageAtIndex:0] animated:YES];
```

About Navigation Position Coordinates

In PDF, positions are typically described using coordinates. PDF uses points as the unit of measurement for positions and dimensions. One point is equal to 1/72 inch, so coordinates and dimensions in a PDF document are based on points.

In the page navigation functionality, with the origin at the top-left corner (0,0), the positive X direction extends horizontally to the right, and the positive Y direction extends vertically downward. Therefore, the coordinates for a specific position can be represented as (X, Y), where X is the horizontal coordinate and Y is the vertical coordinate.

For example, a point located at the bottom-left corner of a PDF page might have coordinates (0, 792), where the page height is 792 points (if the page size is 8.5 x 11 inches, then 792 points correspond to 11 inches).

3.2.4 Outlines

The outline is a structured navigation tool in PDF documents, typically displayed in the sidebar or panel of a document reader. It is often automatically generated based on the document's headings and chapter information, but can also be manually edited and adjusted.

The outline provides a hierarchical structure of the document, enabling users to locate and navigate content more easily. Additionally, users can use the outline to quickly navigate to different sections of the document.

Display the Outlines

Each heading or subheading in a PDF document is represented as a node in the outline tree, with branches connecting the nodes. The main title serves as the root node, and subheadings act as branches stemming from the root, forming a tree-like structure.

The outline is formed by recursively nesting nodes and subnodes, presenting the organizational framework of the document in a hierarchical manner. Nodes typically represent major sections, while subnodes represent subsections or chapters.

This example shows how to display the outline of a PDF recursively:

```swift
// Print document outline information.
func printOutline(document: CPDFDocument) {
    // Retrieve the root file from the PDF document.
    if let outline = document.outlineRoot() {
        // Get subdirectories from the root directory.
        loadOutline(outline: outline, level: 0)
    }
}

// Retrieve subdirectories from the root directory.
func loadOutline(outline: CPDFOutline, level: Int) {
    for i in 0..<outline.numberOfChildren {
        if let data = outline.child(at: i) {
            if let destination = data.destination {

            } else {
                if let action = data.action as? CPDFGoToAction {
                    if let destination = action.destination() {

                    }
                }
            }

            loadOutline(outline: data, level: level + 1)
        }
    }
}
```

```objc
// Print document outline information.
- (void)printOutline:(CPDFDocument *)document {
    // Retrieve the root file from the PDF document.
    CPDFOutline *outline = [document outlineRoot];

    // Get subdirectories from the root directory.
    [self loadOutline:outline level:0];
}

// Retrieve subdirectories from the root directory.
- (void)loadOutline:(CPDFOutline *)outline level:(NSInteger)level {

    for (int i=0; i<[outline numberOfChildren]; i++) {
        CPDFOutline *data = [outline childAtIndex:i];
        CPDFDestination *destination = [data destination];
        if (!destination) {
            CPDFAction *action = [data action];
            if (action && [action isKindOfClass:[CPDFGoToAction class]]) {
                destination = [(CPDFGoToAction *)action destination];
            }

        [self loadOutline:data level:level+1];
    }
```

```
}
```

Add a New Outline

The following are the steps for adding a new outline:

1. Locate the parent outline where the new outline needs to be added.
2. Create a new outline.
3. Add actions to the outline, such as jumping to a page.
4. Set properties.

This example shows how to add a new outline:

```
// Identify the parent outline where the new outline entry needs to be added.
if let outlineRoot = document?.outlineRoot() {
    // Create the outline entry to be added.
    if let outlinePage = outlineRoot.insertChild(at: 0) {
        // Add outline action, in this case, navigate to the first page.
        if let destination = CPDFDestination(document: document, pageIndex: 0, at:
CGPoint(x: 100, y: 100), zoom: 0) {
            let gotoAction = CPDFGoToAction(destination: destination)
            outlinePage.action = gotoAction
        }

        // Set the attributes.
        outlinePage.label = "1. page1"
    }
}
```

```
 // Identify the parent outline where the new outline entry needs to be added.
 CPDFOutline *outline = [document outlineRoot];
 // Create the outline entry to be added.
 CPDFOutline *outlinePage = [outline insertChildAtIndex:0];
 // Add outline action, in this case, navigate to the first page.
CPDFDestination *destination = [[CPDFDestination alloc] initWithDocument:document
pageIndex:0 atPoint:CGPointMake(100, 100) zoom:0];
CPDFGoToAction *gotoAction = [[CPDFGoToAction alloc] initWithDestination:destination];
outlinePage.action = gotoAction;
 // Set the attributes.
outlinePage1.label = @"1. page1";
```

Adjust the Order of the Outlines

Adjust the outline order using the function `CPDFOutline::insertChild:atIndex:`. When moving an item in the outline hierarchy, ensure to retain the item and first call `CPDFOutline::removeFromParent`.

Delete the Outline

Delete the target outline, after deletion, the child outlines of the target outline will also be removed.

This example shows how to delete the outlines:

```
var data = outline.child(at: i)
data?.removeFromParent()
```

```
CPDFOutline *data = [outline childAtIndex:i];
[data removeFromParent]
```

3.2.5 Bookmarks

Bookmarks are user-created markers, which is used to identify and quickly navigate to specific locations in a document. Unlike outlines, bookmarks are manually added by users, typically reflecting their personalized interests in the document content.

Bookmarks offer a user-customized navigation method, enabling users to create personalized markers within a document. Users can swiftly navigate to bookmarked locations without the need to browse through the entire document.

Retrieve the list of bookmarks

This example shows how to get the bookmarks list:

```
var bookmarks = document?.bookmarks()
```

```
NSArray *bookmarks = document.bookmarks;
```

Add a New Bookmark

The steps for adding a new bookmark are as follows:

1. Create a bookmark object.
2. Set bookmark properties.
3. Add the bookmark to the document.

This example shows how to add a new bookmark:

```
document?.addBookmark("new bookmark", forPageIndex:0)
```

```
[document addBookmark:@"new bookmark" forPageIndex:0];
```

Delete a Bookmark

Delete the bookmark for a specified page number.

This example shows how to delete a bookmark:

```
// Delete the bookmark for the first page.
document?.removeBookmark(forPageIndex:0)
```

```
// Delete the bookmark for the first page.
[document removeBookmarkForPageIndex:0];
```

3.2.6 Text Search and Selection

When users perform a text search in a PDF document, the search results are typically highlighted to indicate matching text segments, and links are provided for easy navigation to the corresponding locations. With the text search and selection functionality offered by the ComPDFKit SDK, you can effortlessly implement this feature.

Text Search

Text search enables users to input keywords throughout the entire PDF document to locate matching text.

The text search feature allows users to quickly pinpoint and retrieve information from large documents, enhancing document accessibility and search efficiency. This is particularly beneficial for workflows involving handling large documents, researching materials, or searching for specific information.

The steps for text search are as follows:

1. Create a container for the search results set.
2. Specify the `CPDFSelection` object to be searched, the keyword to be searched, search options, and the number of characters before and after the precise search results.
3. Record the content of the temporary variable in the container each time a result is found.

This example shows how to search specified text:

```
// Specify the container for storing the location and content of search results.
var rectArray = [CGRect]()
var stringArray = [String]()

// Set the search keyword and search options. The search options include case sensitivity
(by default, it is not case-sensitive, and it is not a full word match).
if let resultArray = document.document?.find("searchText", with: .caseSensitive) as?
[[CPDFSelection]] {
    // For loop to iterate through all pages and search for the keyword.
    for array in resultArray {
        for selection in array {
            // Save the search results to a collection.
            let rectValue = NSValue(cgRect: selection.bounds)
            rectArray.append(rectValue.cgRectValue)
            stringArray.append(selection.string())
        }
    }
}
```

```
// Specify the container for storing the location and content of search results.
NSMutableArray *rectAraay = [[NSMutableArray alloc] init];
NSMutableArray *stringAraay = [[NSMutableArray alloc] init];
```

```
// Set the search keyword and search options. The search options include case sensitivity
(by default, it is not case-sensitive, and it is not a full word match).
NSArray *resultArray = [document findString:@"searchText"
withOptions:CPDFSearchCaseInsensitive];

// For loop to iterate through all pages and search for the keyword.
for (NSArray *array in resultArray) {
        for (CPDFSelection *selection in array) {
          // Save the search results to a collection.
            NSValue *rectValue = [NSValue valueWithCGRect:selection.bounds];
            [rectAraay addObject:rectValue];
            [stringAraay addObject:selection.string];
        }
}
```

Explanation of Search Settings

Option	Description	Value
CPDFSearchCaseInsensitive	Case Insensitive	0
CPDFSearchCaseSensitive	Case Sensitive	1
CPDFSearchMatchWholeWord	Match Whole Word	2
CPDFSearchConsecutive	Will skip past the current match to look for the next match	4

Text Selection

The text content is stored in the `CPDFPage` object associated with the respective page. The `CPDFPage` object can be used to retrieve information about the text on a PDF page, such as individual characters, words, and text content within specified character ranges, or within specified boundaries.

By specifying a rectangular range (RectF), capture the text covered by this rectangle along with the position of the text field and store them in variables. This is done to simulate the action of selecting text by dragging the mouse or finger.

This example shows how to select specified text:

```
func selection(for page: CPDFPage, from fPoint: CGPoint, to tPoint: CGPoint) ->
CPDFSelection? {
    let fCharacterIndex = page.characterIndex(at: fPoint)
    let tCharacterIndex = page.characterIndex(at: tPoint)
    let range = NSRange(location: fCharacterIndex, length: tCharacterIndex -
fCharacterIndex + 1)
    let selection = page.selection(for: range)
    return selection
}
```

```
- (CPDFSelection *)selectionForPage:(CPDFPage *)page fromPoint:(CGPoint)fPoint toPoint:
(CGPoint)tPoint {
    NSInteger fCharacterIndex = [page characterIndexAtPoint:fPoint];
    NSInteger tCharacterIndex = [page characterIndexAtPoint:tPoint];
    NSRange range = NSMakeRange(fCharacterIndex, tCharacterIndex - fCharacterIndex + 1);
    CPDFSelection *selection = [page selectionForRange:range];
    return selection;
}
```

3.2.7 Text Reflow

Text reflow refers to reorganizing the text to fit the device screen size and display a layout suitable for reading on different devices.

This example shows how to set text reflow:

```
let url = URL(fileURLWithPath:pdfPath)
if let document = CPDFDocument(url: url),
   let page = document.page(at: 0) {

    let range = NSRange(location: 0, length: Int(page.numberOfCharacters))
    if let string = page.string(for: range) {

    }
}
```

```
NSURL *url = [NSURL fileURLWithPath:pdfPath];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
CPDFPage *page = [document pageAtIndex:0];

NSRange range = NSMakeRange(0, page.numberOfCharacters);
NSString *string = [page stringForRange:range];
```

3.2.8 Zooming

The PDF zoom feature allows you to adjust the page size while reading PDF documents for an enhanced reading experience. Specifically:

Manual Zoom: Use gestures to zoom in and out, adjusting the size of the PDF pages for a clearer view of the document content.

PDF zoom functionality caters to the diverse needs of users in various situations. Users can choose the most suitable zoom level based on their preferences while reading PDF documents, enhancing the clarity and readability of the content.

This example shows how to set custom zooming:

```
// Set the zoom ratio to five times.
pdfview.setScaleFactor(5.0, animated: true)
```

```
// Set the zoom ratio to five times.
[pdfview setScaleFactor:5.0];
```

3.2.9 Themes

Theme refers to using different background colors to render PDF document pages when displaying PDF files to adapt to user preferences and scene needs, enhancing the reading experience.

When modifying the theme, only the visual effects during document display are altered, and it does not modify the PDF document data on the disk. The theme settings are not saved within the PDF document data.

This example shows how to set the dark theme:

```
pdfview.displayMode = .night
```

```
pdfview.displayMode = CPDFDisplayModeNight;
```

Explanation of Theme

Mode	Description	Option Values
Light Color Mode	Uses a white background and black text, suitable for reading in well-lit environments.	`CPDFDisplayModeNormal`
Dark Mode	Uses a dark background and light text, suitable for reading in low-light environments.	`CPDFDisplayModeNight`
Soft Mode	Use a beige background for users who are used to reading on paper.	`CPDFDisplayModeSoft`
Eye Protection Mode	Soft light green background reduces discomfort from high brightness and strong contrast when reading, effectively relieving visual fatigue.	`CPDFDisplayModeGreen`
Custom Color Mode	Customizable color scheme.	`CPDFDisplayModeCustom`

3.2.10 Render

The `CPDFView` class calls `drawPage:toContext:` as necessary for each visible page that requires rendering. You can override this method to draw on top of a PDF page. In this case, invoke this method on `super` and then perform a custom drawing on top of the PDF page. Do not invoke this method, except by invoking it on `super` from a subclass.

3.2.11 Custom Menu

When viewing a PDF, selecting text and performing a long press in a blank area triggers the PDF context to enter the corresponding interactive state. In different interactive states, a context menu similar to `UIMenuController` will appear, allowing you to execute relevant operations through the context menu options.

This example shows how to set a custom menu:

```
func menuItems(at point: CGPoint, for page: CPDFPage) -> [UIMenuItem] {
    var items = super.menuItems(at: point, for: page) ?? []

    var menuItems = [UIMenuItem]()

    if let currentSelection = self.currentSelection {
        let textNoteItem = UIMenuItem(title: NSLocalizedString("Note", comment: ""),
                                      action: #selector(textNoteItemAction(_:)))
        let textShareItem = UIMenuItem(title: NSLocalizedString("Share", comment: ""),
                                       action: #selector(textShareItemAction(_:)))
        let defineItem = UIMenuItem(title: NSLocalizedString("Define", comment: ""),
                                    action: #selector(defineItemAction(_:)))
        let linkItem = UIMenuItem(title: NSLocalizedString("Link", comment: ""),
                                  action: #selector(linkItemAction(_:)))
        let searchItem = UIMenuItem(title: NSLocalizedString("Search", comment: ""),
                                    action: #selector(searchItemAction(_:)))

        menuItems.append(textNoteItem)
        menuItems += [textShareItem, defineItem, linkItem, searchItem]
    } else {
        let textNoteItem = UIMenuItem(title: NSLocalizedString("Note", comment: ""),
                                      action: #selector(textNoteItemAction(_:)))
        let textItem = UIMenuItem(title: NSLocalizedString("Text", comment: ""),
                                  action: #selector(textItemAction(_:)))
        let pasteItem = UIMenuItem(title: NSLocalizedString("Paste", comment: ""),
                                   action: #selector(pasteItemAction(_:)))

        menuItems += [textNoteItem, textItem]

        let textType = kUTTypeText as String
        let urlType = kUTTypeURL as String
        let urlFileType = kUTTypeFileURL as String
        let jpegImageType = kUTTypeJPEG as String
        let pngImageType = kUTTypePNG as String
        let rawImageType = "com.apple.uikit.image"
```

```
        let isPasteboardValid = UIPasteboard.general.contains(pasteboardTypes: [textType,
urlType, urlFileType, jpegImageType, pngImageType, rawImageType])
        if isPasteboardValid {
            menuItems.append(pasteItem)
        }
    }

    return menuItems + items
}
```

```
- (NSArray<UIMenuItem *> *)menuItemsAtPoint:(CGPoint)point forPage:(CPDFPage *)page {
    NSArray *items = [super menuItemsAtPoint:point forPage:page];

    NSMutableArray *menuItems = [NSMutableArray arrayWithArray:items];
    if (self.currentSelection) {
        UIMenuItem *textNoteItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Note", nil)

 action:@selector(textNoteItemAction:)];
        UIMenuItem *textShareItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Share", nil)

action:@selector(textShareItemAction:)];
        UIMenuItem *defineItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Define", nil)

 action:@selector(defineItemAction:)];
        UIMenuItem *linkItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Link", nil)

 action:@selector(linkItemAction:)];
        UIMenuItem *searchItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Search", nil)

 action:@selector(searchItemAction:)];
        [menuItems insertObject:textNoteItem atIndex:0];
        [menuItems addObject:textShareItem];
        [menuItems addObject:defineItem];
        [menuItems addObject:linkItem];
        [menuItems addObject:searchItem];
    } else {
        UIMenuItem *textNoteItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Note", nil)

 action:@selector(textNoteItemAction:)];
        UIMenuItem *textItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Text", nil)

 action:@selector(textItemAction:)];
```

```
        UIMenuItem *pasteItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Paste", nil)

action:@selector(pasteItemAction:)];
        [menuItems addObject:textNoteItem];
        [menuItems addObject:textItem];

        NSString *textType = (NSString *)kUTTypeText;
        NSString *urlType  = (NSString*)kUTTypeURL;
        NSString *urlFileType  = (NSString*)kUTTypeFileURL;
        NSString *jpegImageType = (NSString *)kUTTypeJPEG;
        NSString *pngImageType = (NSString *)kUTTypePNG;
        NSString *rawImageType = @"com.apple.uikit.image";
        BOOL isPasteboardValid = [[UIPasteboard generalPasteboard]
containsPasteboardTypes:[NSArray arrayWithObjects:textType, urlType, urlFileType,
jpegImageType, pngImageType, rawImageType, nil]];
        if (isPasteboardValid) {
            [menuItems addObject:pasteItem];
        }
    }
    return menuItems;
}
```

3.2.12 Highlight Form Fields and Hyperlinks

The highlight feature for PDF form fields helps users quickly locate and fill out forms, significantly enhancing efficiency in scenarios involving extensive form completion.

The highlight feature for hyperlinks allows users to add hyperlinks to crucial information within a PDF document, facilitating other users in swiftly locating and comprehending information. This not only improves the readability of the PDF document but also enhances its interactivity.

This example shows how to highlight form fields and hyperlinks:

```
// Set the form highlighting.
CPDFKitConfig.sharedInstance().setEnableFormFieldHighlight(true)
// Set hyperlink highlighting.
CPDFKitConfig.sharedInstance().setEnableLinkFieldHighlight(true)
```

```
// Set the form highlighting.
[[CPDFKitConfig sharedInstance] setEnableFormFieldHighlight: YES];
// Set hyperlink highlighting.
[[CPDFKitConfig sharedInstance] setEnableLinkFieldHighlight: YES];
```

3.2.13 Get the Selected Content

Users can drag the mouse or use their fingers to select text in the PDF document, obtaining the selected content in real-time.

This example shows how to Get the selected content:

```
// Extract the selected text
let currentSelection = pdfview.currentSelection
let currentText = currentSelection?.string()
```

```
// Extract the selected text
CPDFSelection *currentSelection = pdfView.currentSelection;
NSString *currentText = [currentSelection string];
```

3.3 Annotations

3.3.1 Overview

Annotations allow users to highlight paragraphs, add comments, markup, sign, or stamp PDF documents without modifying the original author's content. The annotated content, along with the original text, can then be shared together.

Benefits of ComPDFKit PDF Annotation SDK

- **Comprehensive Type Support:** Enables highlighting, text, freehand drawing, shapes, stamps, and more.
- **Create, Edit, Delete:** Perform creation, editing, and deletion operations either programmatically or directly through the UI.
- **Flattened Annotations:** Embed annotations permanently onto the document as images, ensuring document appearance stability and preventing further modifications.
- **Annotation Events:** Trigger specified workflows to achieve automation.
- **Annotation Import and Export:** Export annotations as XFDF templates and apply them to multiple documents.
- **Fast UI Integration:** Achieve rapid integration and customization through extensible UI components.

Guides for Annotations

- Access Annotations

Get a list of annotations and individual annotation objects within a document.

- Create Annotations

Generate annotations of various types.

- Edit Annotations

Modify annotation positions, appearances, and properties.

- **Update Annotation Appearances**

After edit the annotation properties, update the annotation appearance to apply the changes.

- **Delete Annotations**

Remove an annotation from the document.

- **Import and Export**

Export annotations as XFDF templates and apply them to multiple documents.

- **Flatten Annotations**

Permanently embed existing annotations within the document as images.

- **Annotation Replies**

Add replies and comments to facilitate written discussions directly within the document.

- **Annotation Rotation**

Allowing users to rotate annotations through a full range (-180~180 degrees).

- **Cloud Border Style for Annotations**

Allowing users to set cloud-shaped border styles for rectangular, circular, and other annotations.

- **Predefine Annotations**

Set default values for certain annotation properties, such as the color and line width of Ink annotations.

3.3.2 Supported Annotation Types

ComPDFKit supports annotation types that adhere to PDF standards, as defined in the PDF reference. These annotations can be read and written by compliant PDF processors, including Adobe Acrobat. The supported annotation types are as follows:

Type	Description	Class Name
Text Annotation	Text annotations appear as small icons or labels. Users can expand them to view related content, making them suitable for personal notes, reminders, or comments.	CPDFTextAnnotation
Link Annotation	Link annotations enable users to directly navigate to other locations within the document or external resources, providing a richer navigation experience.	CPDFLinkAnnotation
Free Text Annotation	Free text annotations allow users to insert free-form text into PDF documents, useful for adding annotations, comments, or explanations of document content.	CPDFFreetextAnnotation
Graphics: Rectangle, Circle, Line, Arrow	This category includes shapes like rectangles, circles, lines, and arrows, used to draw graphics in the document to highlight or mark specific areas.	CPDFSquareAnnotation, CPDFCircleAnnotation, CPDFLineAnnotation
Markup: Highlight, Underline, Strikethrough, Squiggly	Add markup annotations in the PDF document to emphasize, underline, strikethrough, or add squiggly lines to specific content, such as important paragraphs, lines, words, keywords, or tables.	CPDFHighlightAnnotation, CPDFUnderlineAnnotation, CPDFStrikeoutAnnotation, CPDFSquigglyAnnotation
Stamp	Stamp annotations allow the insertion of stamps or seals into PDF documents, containing text, images, or custom designs, resembling the process of stamping on physical documents.	CPDFStampAnnotation
Ink	Draw custom shapes, icons, or doodles using handwritten or mouse-drawn strokes, enabling users to freely create or add personalized graphic elements to the document.	CPDFInkAnnotation
Audio	Add audio files to the PDF document for a multimedia-rich presentation.	CPDFSoundAnnotation

3.3.3 Access Annotations

The steps to access a list of annotations and annotation objects are as follows:

1. Obtain the page object.
2. Access the list of annotations from the page object.
3. Iterate through each annotation object in the list.

This example shows how to access annotations:

```swift
var annotations: [CPDFAnnotation] = []

// Iterate over all pages
for i in 0..<document?.pageCount {
    if let page = document.page(at: i) {
        // terate through all the annotations on the page
        annotations.append(contentsOf: page.annotations)

        for annotation in annotations {
            // Process each annotation
        }
    }
}
```

```objc
NSMutableArray *annotations = [NSMutableArray array];
// Iterate over all pages
for (int i=0; i<document.pageCount; i++) {
    CPDFPage *page = [document pageAtIndex:i];
    // Iterate through all the annotations on the page
    [annotations addObjectsFromArray:[page annotations]];
    for (CPDFAnnotation *annotation in annotations) {

    }
}
```

3.3.4 Create Annotations

ComPDFKit supports a wide range of annotation types, including notes, links, shapes, highlights, stamps, freehand drawings, and audio annotations, catering to diverse annotation requirements.

Create Note Annotations

Note annotations appear as small icons or labels. When clicked by the user, they can expand to display relevant annotation content. This annotation type is used for adding personal notes, reminders, or comments, allowing users to add personalized additional information to the document without affecting the readability of the original text.

The steps to create a note are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a note annotation object on that page.
3. Set the annotation properties.
4. Update the annotation appearance to display it on the document.

This example shows how to create note annotations:

```swift
// Retrieve the page object for creating a note.
if let page = document?.page(at: 0) {
    // Configure the properties of the note annotation.
    let text = CPDFTextAnnotation(document: document)

    // Configure the properties of the note annotation.
    text?.contents = "test"
    text?.bounds = CGRect(x: 50, y: 200, width: 50, height: 50)
    text?.color = UIColor.yellow

    // Update the annotation appearance to make it visible on the document.
    page.addAnnotation(text!)
}
```

```objc
// Retrieve the page object for creating a note.
CPDFPage *page = [document pageAtIndex:0];
// Configure the properties of the note annotation.
CPDFTextAnnotation *text = [[CPDFTextAnnotation alloc] initWithDocument:document];
// Configure the properties of the note annotation.
text.contents = @"test";
text.bounds = CGRectMake(50, 200, 50, 50);
text.color = [UIColor yellowColor];

// Update the annotation appearance to make it visible on the document.
[page addAnnotation:text];
```

Create Link Annotations

Link annotations allow users to navigate directly to other locations within the document or external resources, enhancing the navigation experience.

The steps to create link annotations are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a link annotation object on that page.
3. Use `CPDFDestination` to specify the destination page, for example, jumping to page 2.
4. Set the annotation properties and attach the `CPDFDestination` object to the annotation.

This example shows how to create link annotations:

```swift
// Retrieve the page object for creating a note.
if let page = document?.page(at: 0) {
    // Create a link annotation on the page.
    // Use `CPDFDestination` to set the link to navigate to the second page.
    if let dest = CPDFDestination(document: document, pageIndex: 1),
       let link = CPDFLinkAnnotation(document: document) {
```

```
        // Configure annotation properties and attach the `CPDFDestination` object to the
annotation.
        link?.bounds = CGRect(x: 50, y: 100, width: 50, height: 50)
        link?.destination = dest
        // link.url = "https://www."

        page.addAnnotation(link!)
    }
}
```

```
// Retrieve the page object for creating a note.
CPDFPage *page = [document pageAtIndex:0];
// Create a link annotation on the page.
// Use `CPDFDestination` to set the link to navigate to the second page.
 CPDFDestination *dest = [[CPDFDestination alloc] initWithDocument:document pageIndex:1];
 CPDFLinkAnnotation *link = [[CPDFLinkAnnotation alloc] initWithDocument:document];

// Configure annotation properties and attach the `CPDFDestination` object to the
annotation.
link.bounds = CGRectMake(50, 100, 50, 50);
link.destination = dest; //    link.URL = @"https://www.";
[page addAnnotation:link];
```

Create Free Text Annotations

Free text annotations enable users to insert free-form text into PDF documents, serving the purpose of adding annotations, comments, or explanations to document content.

The steps to create a text annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a free text annotation object on that page.
3. Set the annotation properties.
4. Update the annotation appearance to display it on the document.

This example shows how to create free text annotations:

```
// Retrieve the page object for creating the annotation.
if let page = document?.page(at: 0) {
    // Create a text annotation on that page.
    let freeText = CPDFFreeTextAnnotation(document: document)

    // Configure the properties of the annotation.
    freeText?.contents = "\n\nSome swift brown fox snatched a gray hare out of the air by
freezing it with an angry glare."
    freeText?.bounds = CGRect(x: 10, y: 200, width: 160, height: 570)
    freeText?.font = UIFont(name: "Helvetica", size: 12)
    freeText?.fontColor = UIColor.red
```

```
    freeText?.alignment = .left

    // Add a free text annotation on the page.
    page.addAnnotation(freeText!)
}
```

```
// Retrieve the page object for creating the annotation.
CPDFPage *page = [document pageAtIndex:0];
// Create a free text annotation on that page.
CPDFFreeTextAnnotation *freeText = [[CPDFFreeTextAnnotation alloc]
initWithDocument:document];
// Configure the properties of the annotation.
freeText.contents = @"\n\nSome swift brown fox snatched a gray hare out of the air by
freezing it with an angry glare.";
freeText.bounds = CGRectMake(10, 200, 160, 570);
freeText.font = [UIFont fontWithName:@"Helvetica" size:12];
freeText.fontColor = [UIColor redColor];
freeText.alignment = NSTextAlignmentLeft;
//  add a free text annotation for page
[page1 addAnnotation:freeText1];
```

Create Shape Annotations

Graphic annotations encompass shapes such as rectangles, circles, lines, and arrows, used to draw attention to or highlight specific areas in a document, conveying information that may be challenging to describe with text alone.

The steps to create graphic annotations are as follows:

1. Obtain the page object where the annotations need to be created.
2. Sequentially create rectangle, circle, and line annotations on that page.
3. Set the annotation properties.
4. Sequentially update the annotation appearance to display them on the document.

This example shows how to create shape annotations:

```
// Retrieve the page object for creating the annotation.
let border1 = CPDFBorder(style: .dashed, lineWidth: 1, dashPattern: [2, 1])
let border2 = CPDFBorder(style: .dashed, lineWidth: 1, dashPattern: [2, 0])

if let page = document?.page(at: 0) {
    // Create a rectangle annotation object on that page.
    let square = CPDFSquareAnnotation(document: document)
    // Configure the properties of the annotation.
    square?.bounds = CGRect(x: 400, y: 200, width: 80, height: 300)
    square?.color = UIColor.green
    square?.interiorColor = UIColor.purple
    square?.opacity = 1.0
    square?.interiorOpacity = 1.0
    square?.border = border2
```

```
    page.addAnnotation(square!)

    // Create a circular annotation object on that page.
    let circle = CPDFCircleAnnotation(document: document)
    // Configure the properties of the annotation.
    circle?.bounds = CGRect(x: 300, y: 300, width: 100, height: 100)
    circle?.color = UIColor.red
    circle?.interiorColor = UIColor.yellow
    circle?.opacity = 0.5
    circle?.interiorOpacity = 0.5
    circle?.border = border1
    page.addAnnotation(circle!)

    // Create a line segment annotation object on that page.
    let line = CPDFLineAnnotation(document: document)
    // Configure the properties of the annotation.
    line?.startPoint = CGPoint(x: 350, y: 270)
    line?.endPoint = CGPoint(x: 260, y: 370)
    line?.startLineStyle = .square
    line?.endLineStyle = .circle
    line?.color = UIColor.red
    line?.interiorColor = UIColor.yellow
    line?.opacity = 0.5
    line?.interiorOpacity = 0.5
    line?.border = border1
    line?.contents = "Dashed Captioned"
    page.addAnnotation(line!)
}
```

```
// Retrieve the page object for creating the annotation.
CPDFBorder *border1 = [[CPDFBorder alloc] initWithStyle:CPDFBorderStyleDashed
                                                lineWidth:1
                                              dashPattern:@[@(2), @(1)]];
CPDFBorder *border2 = [[CPDFBorder alloc] initWithStyle:CPDFBorderStyleDashed
                                                lineWidth:1
                                              dashPattern:@[@(2), @(0)]];
CPDFPage *page = [document pageAtIndex:0];

// Create a rectangle annotation object on that page.
CPDFSquareAnnotation *square = [[CPDFSquareAnnotation alloc] initWithDocument:document];
// Configure the properties of the annotation.
square.bounds = CGRectMake(400, 200, 80, 300);
square.color = [UIColor greenColor];
square.interiorColor = [UIColor purpleColor];
square.opacity = 1.0;
square.interiorOpacity = 1.0;
square.border = border2;
```

```
[page addAnnotation:square];

// Create a circular annotation object on that page.
CPDFCircleAnnotation *circle = [[CPDFCircleAnnotation alloc] initWithDocument:document];
// Configure the properties of the annotation.
circle.bounds = CGRectMake(300, 300, 100, 100);
circle.color = [UIColor redColor];
circle.interiorColor = [UIColor yellowColor];
circle.opacity = 0.5;
circle.interiorOpacity = 0.5;
circle.border = border1;
[page addAnnotation:circle];

// Create a line segment annotation object on that page.
CPDFLineAnnotation *line = [[CPDFLineAnnotation alloc] initWithDocument:document];
// Configure the properties of the annotation.
line.startPoint = CGPointMake(350, 270);
line.endPoint = CGPointMake(260, 370);
line.startLineStyle = CPDFLineStyleSquare;
line.endLineStyle = CPDFLineStyleCircle;
line.color = [UIColor redColor];
line.interiorColor = [UIColor yellowColor];
line.opacity = 0.5;
line.interiorOpacity = 0.5;
line1.border = border1;
[line setContents:@"Dashed Captioned"];
[page addAnnotation:line1];
```

Explanation of Line Types

Name	Description
CPDFLineStyleNone	No special line ending.
CPDFLineStyleOpenArrow	Two short lines meeting in an acute angle to form an open arrowhead.
CPDFLineStyleClosedArrow	Two short lines meeting in an acute angle as in the LINETYPE_ARROW style and connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color.
CPDFLineStyleSquare	A square filled with the annotation's interior color.
CPDFLineStyleCircle	A circle filled with the annotation's interior color.
CPDFLineStyleDiamond	A diamond shape filled with the annotation's interior color.

Create Markup Annotations

Incorporate annotations in a PDF document to highlight, emphasize, or explain specific content, such as important paragraphs, lines, words, keywords, tables, etc. ComPDFKit provides four types of markup annotations: highlight, underline, squiggly line, and strikethrough.

The steps to create a markup annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a text object through the page object.
3. Use the text object to identify the location of the text to be marked.
4. Create the corresponding markup object on that page.
5. Set the properties of the markup object.
6. Update the annotation appearance to display it on the document.

This example shows how to create markup annotations:

```
// Retrieve the page object for creating the annotation.
if let page = document?.page(at: 0) {
    // Get the array of search results.
    if let resultArray = document?.findString("Page", withOptions: .caseInsensitive) as?
[[CPDFSelection]] {

        // Get the first page of search results.
        if let selections = resultArray[safe: 3] {

            // Get the first search result on the first page.
            if let selection = selections.first {

                var quadrilateralPoints: [CGPoint] = []

                let bounds = selection. bounds
                quadrilateralPoints?.append(CGPoint(x: bounds.minX, y: bounds.maxY))
                quadrilateralPoints?.append(CGPoint(x: bounds.maxX, y: bounds.maxY))
                quadrilateralPoints?.append(CGPoint(x: bounds.minX, y: bounds.minY))
                quadrilateralPoints?.append(CGPoint(x: bounds.maxX, y: bounds.minY))

                // Create a highlight annotations.
                if let highlight = CPDFMarkupAnnotation(document: document, markupType:
.highlight) {
                    highlight.color = UIColor.yellow
                    highlight.quadrilateralPoints = quadrilateralPoints!
                    page.addAnnotation(highlight)
                }
            }
        }
    }
}
```

```
// Retrieve the page object for creating the annotation.
CPDFPage *page = [document pageAtIndex:0];
// Get the array of search results.
```

```
NSArray *resultArray = [document findString:@"Page"
withOptions:CPDFSearchCaseInsensitive];

// Get the first page of search results.
NSArray *selections = [resultArray objectAtIndex:3];

// Get the first search result on the first page.
CPDFSelection *selection = [selections objectAtIndex:0];

NSMutableArray *quadrilateralPoints = [NSMutableArray array];

CGRect bounds = selection.bounds;
[quadrilateralPoints addObject:[NSValue
valueWithCGPoint:CGPointMake(CGRectGetMinX(bounds), CGRectGetMaxY(bounds))]];
[quadrilateralPoints addObject:[NSValue
valueWithCGPoint:CGPointMake(CGRectGetMaxX(bounds), CGRectGetMaxY(bounds))]];
[quadrilateralPoints addObject:[NSValue
valueWithCGPoint:CGPointMake(CGRectGetMinX(bounds), CGRectGetMinY(bounds))]];
[quadrilateralPoints addObject:[NSValue
valueWithCGPoint:CGPointMake(CGRectGetMaxX(bounds), CGRectGetMinY(bounds))]];


// Create a highlight annotations.
CPDFMarkupAnnotation *highlight = [[CPDFMarkupAnnotation alloc] initWithDocument:document
markupType:CPDFMarkupTypeHighlight];
highlight.color = [UIColor yellowColor];
highlight.quadrilateralPoints = quadrilateralPoints;
[page4 addAnnotation: highlight];
```

Create Stamp Annotations

Stamp annotations are used to identify and validate the source and authenticity of a document. ComPDFKit supports standard stamps, text stamps, and image stamps.

The steps to create a stamp annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create the corresponding stamp on that page.
3. Set the properties of the stamp.
4. Update the annotation appearance to display it on the document.

This example shows how to create stamp annotations:

```
// Retrieve the page object for creating the annotation.
if let page = document?.page(at: 0) {
    // Create a standard stamp.
    let standard = CPDFStampAnnotation(document: document, type: i)
    standard?.bounds = CGRect(x: 50, y:30, width: 50, height: 30)
    page.addAnnotation(standard!)

    // Create a text stamp.
```

```
    let outputFormatter = DateFormatter()
    outputFormatter?.timeZone = TimeZone.current

    // Get date.
    let tDate: String
    outputFormatter.dateFormat = "yyyy-MM-dd HH:mm:SS"
    tDate = outputFormatter.string(from: Date())

    let text = CPDFStampAnnotation(document: document, text: "ComPDFKit", detailText:
tDate, style: .red,     shape: .arrowLeft)
    text?.bounds = CGRect(x: 150, y: 50, width: 80, height: 50)
    page.addAnnotation(text!)

    // Create an image stamp.
    if let logoImage = UIImage(named: "Logo") {
        let image = CPDFStampAnnotation(document: document, image: logoImage)
        image?.bounds = CGRect(x: 150, y: 120, width: 50, height: 50)
        page.addAnnotation(image!)
    }
}
```

```
// Retrieve the page object for creating the annotation.
CPDFPage *page = [document pageAtIndex:0];

// Create a standard stamp.
 CPDFStampAnnotation *standard = [[CPDFStampAnnotation alloc] initWithDocument:document
type:i];
 standard.bounds = CGRectMake(50, height5 - i*30, 50, 30);
 [page addAnnotation:standard];

// Create a text stamp.
NSTimeZone* timename = [NSTimeZone systemTimeZone];
NSDateFormatter *outputFormatter = [[NSDateFormatter alloc] init ];
[outputFormatter setTimeZone:timename ];

// Get date.
NSString *tDate = nil;
[outputFormatter setDateFormat:@"yyyy-MM-dd HH:mm:SS"];
tDate = [outputFormatter stringFromDate:[NSDate date]];
CPDFStampAnnotation *text = [[CPDFStampAnnotation alloc] initWithDocument:document
text:@"ComPDFKit" detailText:tDate style:CPDFStampStyleRed
shape:CPDFStampShapeArrowLeft];
text.bounds = CGRectMake(150, height5-50, 80, 50);
[page addAnnotation:text];

// Create an image stamp.
CPDFStampAnnotation *image = [[CPDFStampAnnotation alloc] initWithDocument:document
image:[UIImage imageNamed:@"Logo"]];
```

```
image.bounds = CGRectMake(150, height5-120, 50, 50);
[page addAnnotation:image];
```

Create Ink Annotations

Ink annotations provide a direct and convenient method for drawing annotations.

The steps to create a freehand annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a ink annotation on that page.
3. Set the path taken by the freehand annotation.
4. Set other properties of the annotation.
5. Update the annotation appearance to display it on the document.

This example shows how to create ink annotations:

```
// Retrieve the page object for creating the annotation.
if let page = document?.page(at: 0) {
    // Create a ink drawing annotation on that page.
    let ink = CPDFInkAnnotation(document: document)
    // Set the path traced by the freehand drawing annotation.
    let startPoint = CGPoint(x: 220, y: 505)
    let point1 = CGPoint(x: 100, y: 490)
    let point2 = CGPoint(x: 120, y: 410)
    let point3 = CGPoint(x: 100, y: 400)
    let point4 = CGPoint(x: 180, y: 490)
    let endPoint = CGPoint(x: 140, y: 440)

    ink?.color = UIColor.red
    ink?.opacity = 0.5
    ink?.borderWidth = 2.0
    ink?.paths = [
        [
            NSValue(cgPoint: startPoint),
            NSValue(cgPoint: point1),
            NSValue(cgPoint: point2),
            NSValue(cgPoint: point3),
            NSValue(cgPoint: point4),
            NSValue(cgPoint: endPoint)
        ]
    ]
    page.addAnnotation(ink!)
}
```

```
// Retrieve the page object for creating the annotation.
CPDFPage *page = [document pageAtIndex:0];
```

```objc
// Create a ink drawing annotation on that page.
CPDFInkAnnotation *ink = [[CPDFInkAnnotation alloc] initWithDocument:document];
// Set the path traced by the freehand drawing annotation.
CGPoint startPoint = CGPointMake(220, 505);
CGPoint point1 = CGPointMake(100, 490);
CGPoint point2 = CGPointMake(120, 410);
CGPoint point3 = CGPointMake(100, 400);
CGPoint point4 = CGPointMake(180, 490);
CGPoint endPoint = CGPointMake(140, 440);
ink.color = [UIColor redColor];
ink.opacity = 0.5;
ink.borderWidth = 2.0;
ink.paths = @[@[[NSValue valueWithCGPoint:startPoint],[NSValue valueWithCGPoint:point1],
[NSValue valueWithCGPoint:point2],[NSValue valueWithCGPoint:point3],[NSValue
valueWithCGPoint:point4],[NSValue valueWithCGPoint:endPoint]]];
[page addAnnotation:ink];
```

Create Audio Annotations

The steps to create an audio annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create an audio annotation on that page.
3. Set the audio file.
4. Set other properties.
5. Update the annotation appearance to display it on the document.

This example shows how to create audio annotations:

```swift
// Retrieve the page object for creating the annotation.
if let page = document.page(at: 0) {
    // Create an audio annotation on that page.
    if let soundAnnotation = CPDFSoundAnnotation(document: document) {
        // Set the audio file for the annotation.
        if let filePath = Bundle.main.path(forResource: "Bird", ofType: "wav") {
            soundAnnotation.setMediaPath(filePath)
            // Configure additional properties.
            soundAnnotation?.bounds = CGRect(x: 100, y: 200, width: 50, height: 50)
            page.addAnnotation(soundAnnotation!)
        }
    }
}
```

```
// Retrieve the page object for creating the annotation.
CPDFPage page = document.PageAtIndex(0);
// Create an audio annotation on that page.
CPDFSoundAnnotation *soundAnnotation = [[CPDFSoundAnnotation alloc]
initWithDocument:document];
// Set the audio file for the annotation.
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Bird" ofType:@"wav"];
// Configure additional properties.
if ([soundAnnotation setMediaPath:filePath]) {
   soundAnnotation.bounds = CGRectMake(100, 200, 50, 50);
   [page4 addAnnotation:soundAnnotation];
}
```

3.3.5 Edit Annotations

The steps to edit an annotation are as follows:

1. Obtain the page object where the annotation needs to be edited.
2. Access the list of annotations on that page.
3. Locate the desired annotation in the annotation list and convert it.
4. Set the properties to the annotation object.
5. Update the annotation appearance to display it on the document.

This example shows how to edit annotations:

```
var annotations: [CPDFAnnotation] = []

for i in 0..<document?.pageCount ?? 0 {
    // Iterate through all pages.
    if let page = document?.page(at: i) {
        // Iterate through all the annotations on the page.
        annotations += page.annotations
        for annotation in annotations {
            // Process every annotations here.
        }
    }
}
```

```
NSMutableArray *annotations = [NSMutableArray array];
for (int i=0; i<document.pageCount; i++) {
    // Loop through all anntation
    CPDFPage *page = [document pageAtIndex:i];
    [annotations addObjectsFromArray:[page annotations]];
    for (CPDFAnnotation *annotation in annotations) {

    }
}
```

3.3.6 Update Annotation Appearances

Annotations may contain properties that describe their appearance — such as annotation color or shape. However, these don't guarantee that the annotation will be displayed the same in different PDF viewers. To solve this problem, each annotation can define an appearance stream that should be used for rendering the annotation.

ComPDFKit PDF SDK will update the annotation appearance by default when you modify the annotation properties. You can also manually update the appearance by calling the `updateAppearanceStream` method, but you must call the `updateAppearanceStream` method manually when you modify the bounds of the FreeText, Stamp, Signature annotation, refer to the following method in the `CPDFAnnotation` class.

```
bool UpdateAp();
```

It's easy to set up an annotation to show a custom appearance stream. This is typically done with stamp annotations because they have few other properties. A stamp annotation used this way is usually called an image annotation.

The following part introduces how to set annotation appearance that does not match page rotation.

If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation bounds shall remain in a fixed location on the page.

```
CPDFKitShareConfig.enableAnnotationNoRotate = YES;
```

In addition, when it comes to the FreeText annotation, refer to the following method in the `PDFListView` class.

```
- (void)addAnnotationFreeTextAtPoint:(CGPoint)point forPage:(CPDFPage *)page;
- (void)drawPage:(CPDFPage *)page toContext:(CGContextRef)context;
- (void)moveAnnotation:(CPDFAnnotation *)annotation fromPoint:(CGPoint)fromPoint toPoint:
(CGPoint)toPoint forType:(PDFAnnotationDraggingType)draggingType;
```

3.3.7 Delete Annotations

The steps to delete an annotation are as follows:

1. Obtain the page object where the annotation needs to be deleted.
2. Access the list of annotations on that page.
3. Locate the desired annotation in the annotation list.
4. Delete the identified annotation.

This example shows how to delete annotations:

```
// Retrieve the page object for editing the annotation.
if let page = document?.page(at: 0) {
    // Retrieve the list of annotations for that page.
    if let annotation = page.annotations {
        // Delete the identified annotation.
        page.removeAnnotation(annotation[0])
    }
}
```

```
// Retrieve the page object for editing the annotation.
CPDFPage *page = [document pageAtIndex:0];
// Retrieve the list of annotations for that page.
CPDFAnnotation *annotation = [[page annotations] objectAtIndex:0];
// Delete the identified annotation.
[page removeAnnotation: annotation];
```

3.3.8 Import and Export

The methods for importing and exporting XFDF annotations allow users to save and restore annotations and form data without altering the original PDF document, facilitating the sharing and processing of documents across different editors or platforms.

Import Annotations

When importing annotations via XFDF, a temporary file directory is created. It is necessary to specify both the XFDF path and the temporary file path during the annotation import.

This example shows how to import annotations:

```
var document = CPDFDocument(url: URL(string: "filePath"))
document?.importAnnotation(fromXFDFPath: "importFilePath")
```

```
CPDFDocument *importDocument = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:"filePath"]];
[importDocument importAnnotationFromXFDFPath: self.exportFilePath];
```

Export Annotations

When exporting annotations via XFDF, a temporary file directory is generated. It is essential to specify both the XFDF path and the temporary file path during annotation export.

This example shows how to export annotations:

```
var document = CPDFDocument(url: URL(string: "filePath"))
document?.exportAnnotation(toXFDFPath: "exportFilePath")
```

```
CPDFDocument *importDocument = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:"filePath"]];
[importDocument importAnnotationFromXFDFPath: "exportFilePath"];
```

What is XFDF?

XFDF (XML Forms Data Format) is an XML format used to describe and transmit PDF form data. It is commonly used in conjunction with PDF files to store and pass values, states, and operations of form fields.

An XFDF file contains data corresponding to a PDF form, including the names, values, options, and formats of form fields.

XFDF serves as a format for describing form data and does not encompass the PDF file itself. It is employed for storing and transmitting form data, facilitating interaction and sharing between different systems and applications.

3.3.9 Flatten Annotations

Annotation flattening refers to converting editable annotations into non-editable, non-modifiable static images or plain text forms. When the annotations are flattened, all editable elements of the entire document (including comments and forms) will be flattened, so the annotation flattening is also known as document flattening.

This example shows how to flatten annotations:

```
var document = CPDFDocument(url: URL(string: "filePath"))
document?.writeFlatten(to: URL(string: "savePath"))
```

```
CPDFDocument *importDocument = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:"filePath"]];
document.WriteFlattenToFilePath("savePath");
```

What is Document Flattening?

Document flattening refers to the process of converting editable elements, such as annotations, form fields, or layers, in a PDF document into non-editable, static images, or pure text. The purpose of this process is to lock the final state of the document, eliminating editable elements.

Document flattening is typically applied in the following contexts:

1. **Content Protection:** Flattening can be used to protect document content, ensuring that the document remains unaltered during distribution or sharing. This is crucial for maintaining document integrity and confidentiality.
2. **Form Submission:** In form processing, flattening can convert user-filled form fields and annotations into static images for easy transmission, archiving, or printing, while preventing modifications to the form content in subsequent stages.
3. **Compatibility and Display:** Some PDF readers or browsers may encounter issues with displaying and interacting with PDF documents that contain numerous annotations or layers. Document flattening helps address these compatibility issues, enhancing the visual representation of documents in various environments.
4. **File Size Reduction:** Flattened documents typically have reduced file sizes since editable elements are converted into static images or text, eliminating the need to store additional data for editing information.

3.3.10 Annotation Replies

Annotation replies enable you and other users to engage in written discussions directly within the document. ComPDFKit provides convenient APIs for accessing replies in the document, as well as user interface (UI) components for viewing and editing replies.

Creating Text Replies

Text replies allow you to add written responses below annotations, commonly used for discussions related to the annotation.

The steps to create a text reply are as follows:

1. Retrieve the page object where you want to create the text reply from CPDFDocument.
2. Get the annotation on the page object.
3. Create a text reply annotation on the annotation and add the reply content.

Here's an example code for creating a text reply:

```
// Retrieve the page object where you want to create the note
if let page = document?.page(at: 0) {
  // Get the annotation on the page
  let annotation = page.annotations[0]

  // Create a reply annotation and add reply content
  let reply = annotation?.createReply()
  reply?.contents = "ComPDFKit"
}
```

```
// Retrieve the page object where you want to create the note
CPDFPage *page = [document pageAtIndex:0];

// Get the annotation on the page
CPDFAnnotation *annotation = page.annotations[0];

// Create a reply annotation and add reply content
CPDFAnnotation *reply = [annotation createReplyAnnotation];
reply.contents = @"ComPDFKit";
```

Creating State Replies

State replies allow you to mark the state of annotations, categorized into mark states and review states, which can be distinguished using the `CPDFAnnotationState` enumeration type.

The steps to create a state reply are as follows:

1. Retrieve the page object where you want to create the state reply from `CPDFDocument`.
2. Get the annotation on the page object.
3. Create mark state reply annotations and review state reply annotations on the annotation.

Here's an example code for creating state replies:

```
// Retrieve the page object where you want to create the note
if let page = document?.page(at: 0) {
  // Get the annotation on the page
  let annotation = page.annotations[0]

  // Create a mark state reply
  let checkReply = annotation?.createReplyStateAnnotation(.marked)
  // Create a review state reply
  let reviewReply = annotation?.createReplyStateAnnotation(.rejected)
}
```

```
// Retrieve the page object where you want to create the note
CPDFPage *page = [document pageAtIndex:0];

// Get the annotation on the page
CPDFAnnotation *annotation = page.annotations[0];

// Create a mark state reply
CPDFAnnotation *checkReply = [annotation
createReplyStateAnnotation:CPDFAnnotationStateMarked];
// Create a review state reply
CPDFAnnotation *reviewReply = [annotation
createReplyStateAnnotation:CPDFAnnotationStateRejected];
```

Enumeration Types for State Reply Annotations

Name	Description
`CPDFAnnotationStateMarked`	Check state, used for mark replies
`CPDFAnnotationStateUnMarked`	Uncheck state, used for mark replies
`CPDFAnnotationStateAccepted`	Accepted state, used for review replies
`CPDFAnnotationStateRejected`	Rejected state, used for review replies
`CPDFAnnotationStateCanceled`	Canceled state, used for review replies
`CPDFAnnotationStateCompleted`	Completed state, used for review replies
`CPDFAnnotationStateNone`	No state, used for review replies
`CPDFAnnotationStateError`	Error state

Getting All Replies

The feature to get all replies allows you to retrieve all replies under an annotation, including text replies and state replies. The `CPDFReplyAnnotationType` enumeration type can be used to distinguish the reply types.

The steps to get all replies are as follows:

1. Retrieve the page object you need from `CPDFDocument`.
2. Get the annotation on the page object.
3. Get all replies of the annotation and distinguish the reply types.

Here's an example code for getting all replies:

```
// Retrieve the page object where you want to get the replies
if let page = document?.page(at: 0) {
  // Get the annotation on the page
  let annotation = page.annotations[0]

  // Get all replies of the annotation
  let replies = annotation.replyAnnotations ?? []
  for reply in replies {
      if reply.replyAnnotationType == .reply {
          // Text reply annotation type
      } else if reply.replyAnnotationType == .mark {
          // Mark state reply annotation type
      } else if reply.replyAnnotationType == .review {
          // Review state reply annotation type
      }
  }
}
```

```
// Retrieve the page object where you want to get the replies
CPDFPage *page = [document pageAtIndex:0];

// Get the annotation on the page
CPDFAnnotation *annotation = page.annotations[0];

// Get all replies of the annotation
NSArray<CPDFAnnotation *> *replies = [annotation replyAnnotations];
for (CPDFAnnotation *reply in replies) {
    if (reply.replyAnnotationType == CPDFReplyAnnotationTypeReply) {
        // Text reply annotation type
    } else if (reply.replyAnnotationType == CPDFReplyAnnotationTypeMark) {
        // Mark state reply annotation type
    } else if (reply.replyAnnotationType == CPDFReplyAnnotationTypeReview) {
        // Review state reply annotation type
    }
}
```

Enumeration Types for Reply Annotations

Name	Description
`CPDFReplyAnnotationTypeNone`	Non-reply annotation type, used to distinguish between reply annotations and normal annotations
`CPDFReplyAnnotationTypeReply`	Text reply annotation type
`CPDFReplyAnnotationTypeMark`	Mark state reply annotation type
`CPDFReplyAnnotationTypeReview`	Review state reply annotation type

3.3.11 Annotation Rotation

Annotation rotation provides the functionality to rotate standard stamp annotations, custom stamp annotations, image annotations, and electronic signature annotations, allowing users to rotate annotations through a full range (-180~180 degrees). ComPDFKit offers convenient APIs for rotating annotations, along with demo demonstrations.

Rotating Annotations Programmatically

To rotate annotations programmatically, we need to set three properties of the annotation when initializing it: `annotationRotation`, `saveSourceRect`, and `saveRectRotationPoints`.

- **annotationRotation**

  Sets the rotation angle of the annotation in degrees, with a range from -180 to 180.

- **saveSourceRect**

  The rectangular Rect before rotation. It's worth noting that this only needs to be refreshed when the annotation is scaled or moved, not when it is rotated.

- **saveRectRotationPoints**

  All rectangular vertices after rotation. Below is an example of how to obtain the vertices after rotation.

Here is the example code for annotation rotation:

```
// Create standard Stamp annotations
let stampAnnotation = CPDFStampAnnotation(document: document, type: selectedIndex)
 annotation?.bounds = CGRect(x: 100, y: 100, width: 100, height: 100)

// Sets the rotation property of the annotations
stampAnnotation.annotationRotation = 0
// rotatePoints initialize the four vertices of the bounds
stampAnnotation.setSaveRectRotationPoints(rotatePoints)
stampAnnotation.updateRotationAppearanceStream()
stampAnnotation.setSaveSourceRect(stampAnnotation.bounds)
stampAnnotation.setSaveRectRotationPoints(rotatePoints)

// Annotation rotation
@objc func rotateStampAnnotation(_ stampAnnotation: CPDFStampAnnotation, Roate
angleDifference: CGFloat) {
    stampAnnotation.annotationRotation = Int(angleDifference * 180 / .pi)
  // Get stampAnnotation.saveSourceRect() four vertices
    var rotatedVertices = computeRotatedRectVerticesWithMidpoints(bounds:
stampAnnotation.saveSourceRect(), annotationRotationDegrees: 0, includeMidpoints: true)
  // rotatedVertices are sorted, with the lower left foot of saveSourceRect as the first
point, sorted counterclockwise
    rotatedVertices = sortPointsToFormRectangle(points: rotatedVertices)
    let center =  CGPoint(x: stampAnnotation.saveSourceRect().midX, y:
stampAnnotation.saveSourceRect().midY)
    var rotatePoints: [NSValue] = []
    var newRotatedVertices: [CGPoint] = []
    for point in rotatedVertices {
      // Gets the rotated vertex
        let rotatePoint = rotatePointAroundCenter(point: point, center: center,
angleRadians: angleDifference)
        let pointValue = NSValue(cgPoint: rotatePoint)
        rotatePoints.append(pointValue)
        newRotatedVertices.append(rotatePoint)
    }
    stampAnnotation.setSaveRectRotationPoints(rotatePoints)
    stampAnnotation.bounds = boundingRectForPoints(points: newRotatedVertices)
    stampAnnotation.updateRotationAppearanceStream()
    stampAnnotation.setSaveRectRotationPoints(rotatePoints)
}

// Gets the rotated vertex
func rotatePointAroundCenter(point: CGPoint, center: CGPoint, angleRadians: CGFloat) ->
CGPoint {
    let s = sin(angleRadians)
    let c = cos(angleRadians)
```

```
    // Translate point back to origin
    var translatedPoint = CGPoint(x: point.x - center.x, y: point.y - center.y)

    // Rotate point
    let xnew = translatedPoint.x * c - translatedPoint.y * s
    let ynew = translatedPoint.x * s + translatedPoint.y * c

    // Translate point back
    translatedPoint.x = xnew + center.x
    translatedPoint.y = ynew + center.y

    return translatedPoint
}
```

```
// Create standard Stamp annotations
CPDFStampAnnotation *stampAnnotation = [[CPDFStampAnnotation alloc]
initWithDocument:document type:selectedIndex];
stampAnnotation.bounds = CGRectMake(100, 100, 100, 100);

// Sets the rotation property of the annotations
stampAnnotation.annotationRotation = 0;
// rotatePoints initialize the four vertices of the bounds
[stampAnnotation setSaveRectRotationPoints:rotatedVertices];
[stampAnnotation updateAnnotationRotationAppearanceStream];
[stampAnnotation setSaveRectRotationPoints:rotatedVertices];
[stampAnnotation setSaveSourceRect:stampAnnotation.bounds];

// Annotation rotation
- (void)rotateStampAnnotation:(CPDFStampAnnotation *)stampRotation rotateAngle:
(CGFloat)currentAngle {
    CGPoint center = CGPointMake(CGRectGetMidX(stampRotation.saveSourceRect),
CGRectGetMidY(stampRotation.saveSourceRect));
   // Get stampAnnotation.saveSourceRect() four vertices
    NSMutableArray<NSValue *> *saveSourceRectPoint =
computeRotatedRectVerticesWithMidpoints(stampRotation.saveSourceRect, 0, YES);
  // rotatedVertices are sorted, with the lower left foot of saveSourceRect as the first
point, sorted counterclockwise
    NSMutableArray<NSValue *> *saveRectPoints =
sortPointsToFormRectangle(saveSourceRectPoint);

    [stampRotation setAnnotationRotation :currentAngle];
    NSMutableArray<NSValue *> *saveRectRotationPoints = [[NSMutableArray array]
mutableCopy];
    for (NSUInteger i = 0; i < saveRectPoints.count ; i ++) {
        NSValue *savePointValue = [saveRectPoints objectAtIndex:i];
        CGPoint savePoint = [savePointValue pointValue];
        CGFloat angleRadians = currentAngle * M_PI / 180.0;
      // Gets the rotated vertex
```

```
        CGPoint saveRotationPoint = rotatePointAroundCenter(savePoint, center,
angleRadians);
        [saveRectRotationPoints addObject:[NSValue valueWithPoint:saveRotationPoint]];
    }
    [stampRotation setSaveRectRotationPoints:saveRectRotationPoints];
    stampRotation.bounds = boundingRectForPoints(saveRectRotationPoints);
    [stampRotation updateAnnotationRotationAppearanceStream];
    [stampRotation setSaveRectRotationPoints:saveRectRotationPoints];

    [self setNeedsDisplayAnnotationViewForPage:stampRotation.page];
}

// Gets the rotated vertex
CGPoint rotatePointAroundCenter(CGPoint point, CGPoint center, CGFloat angleRadians) {
    CGFloat s = sin(angleRadians);
    CGFloat c = cos(angleRadians);

    // Translate point back to origin
    point.x -= center.x;
    point.y -= center.y;

    // Rotate point
    CGFloat xnew = point.x * c - point.y * s;
    CGFloat ynew = point.x * s + point.y * c;

    // Translate point back
    point.x = xnew + center.x;
    point.y = ynew + center.y;
    return point;
}
```

3.3.12 Cloud Border Style for Annotations

Users are allowed to set the border style of rectangular, circular, and polygonal annotations to a cloud-like pattern. The border style can also be changed between solid lines, dashed lines, or the cloud style. ComPDFKit provides convenient APIs for setting the cloud border style for annotations, along with demo demonstrations.

Here is an example code for adding a cloud-style border to a rectangular annotation:

```
// Dotted line style
let border = CPDFBorder(style: .dashed, lineWidth: 1, dashPattern: [2, 1])

let square = CPDFSquareAnnotation(document: document)
// Set the annotation properties.
square?.bounds = CGRect(x: 400, y: 200, width: 80, height: 300)
square?.color = UIColor.green
square?.interiorColor = UIColor.purple
square?.opacity = 1.0
```

```
square?.interiorOpacity = 1.0
// Set a solid or dashed border style
square?.border = border

// Set the cloud border style
let borderEffect = CPDFBorderEffect()
borderEffect?.borderEffectType = .cloudy
borderEffect?.intensityType = .two
square?.borderEffect = borderEffect

page.addAnnotation(square!)
```

```
// Dotted line style
CPDFBorder *border = [[CPDFBorder alloc] initWithStyle:CPDFBorderStyleDashed
                                             lineWidth:1
                                           dashPattern:@[@(2), @(1)]];

CPDFSquareAnnotation *square = [[CPDFSquareAnnotation alloc] initWithDocument:document];
// Set the annotation properties.
square.bounds = CGRectMake(400, 200, 80, 300);
square.color = [UIColor greenColor];
square.interiorColor = [UIColor purpleColor];
square.opacity = 1.0;
square.interiorOpacity = 1.0;
// Set a solid or dashed border style
square.border = border;

// Set the cloud border style
CPDFBorderEffect *borderEffect = [[CPDFBorderEffect alloc] init];
borderEffect.intensityType = CPDFIntensityTypeTwo;
borderEffect.borderEffectType = CPDFBorderEffectTypeCloudy;
square.borderEffect = borderEffect

[page addAnnotation:square];
```

Cloud Border style Type Enumeration

Name	Description
CPDFIntensityTypeZero	The influence intensity ranges from 0 to 2, and when it is 0, it is a solid line
CPDFIntensityTypeOne	
CPDFIntensityTypeTwo	

Name	Description
CPDFBorderEffectTypeSolid	The border style style is a solid line
CPDFBorderEffectTypeCloudy	The border style style is cloud style

3.3.13 Predefine Annotations

ComPDFKit PDF SDK has default values for some annotation properties, such as colors and line widths for ink annotations.

This is all handled in `CPDFKitConfig`, which is a global singleton. You can access it with `CPDFKitShareConfig`.

The current set of defaults is configured on the first run and saved in `NSUserDefaults`.

```
// Author.
CPDFKitConfig.sharedInstance().setAnnotationAuthor("")

// Color.
CPDFKitConfig.sharedInstance().setHighlightAnnotationColor(UIColor.yellow)
CPDFKitConfig.sharedInstance().setUnderlineAnnotationColor(UIColor.blue)
CPDFKitConfig.sharedInstance().setStrikeoutAnnotationColor(UIColor.red)
CPDFKitConfig.sharedInstance().setSquigglyAnnotationColor(UIColor.black)
CPDFKitConfig.sharedInstance().setShapeAnnotationColor(UIColor.red)
CPDFKitConfig.sharedInstance().setShapeAnnotationInteriorColor(UIColor.clear)
CPDFKitConfig.sharedInstance().setFreehandAnnotationColor(UIColor.red)

// Opacity.
CPDFKitConfig.sharedInstance().setMarkupAnnotationOpacity(0.5)
CPDFKitConfig.sharedInstance().setShapeAnnotationOpacity(1.0)
CPDFKitConfig.sharedInstance().setShapeAnnotationInteriorOpacity(0.0)
CPDFKitConfig.sharedInstance().setFreehandAnnotationOpacity(1.0)

// Border Width.
CPDFKitConfig.sharedInstance().setShapeAnnotationBorderWidth(1.0)
CPDFKitConfig.sharedInstance().setFreehandAnnotationBorderWidth(1.0)
```

```
// Author.
CPDFKitShareConfig.annotationAuthor = @"";

// Color.
CPDFKitShareConfig.highlightAnnotationColor = [UIColor yellowColor];
CPDFKitShareConfig.underlineAnnotationColor = [UIColor blueColor];
CPDFKitShareConfig.strikeoutAnnotationColor = [UIColor redColor];
CPDFKitShareConfig.squigglyAnnotationColor = [UIColor blackColor];
CPDFKitShareConfig.shapeAnnotationColor = [UIColor redColor];
```

```
CPDFKitShareConfig.shapeAnnotationInteriorColor = nil;
CPDFKitShareConfig.freehandAnnotationColor = [UIColor redColor];

// Opacity.
CPDFKitShareConfig.markupAnnotationOpacity = 0.5;
CPDFKitShareConfig.shapeAnnotationOpacity = 1.0;
CPDFKitShareConfig.shapeAnnotationInteriorOpacity = 0.0;
CPDFKitShareConfig.freehandAnnotationOpacity = 1.0;

// Border Width.
CPDFKitShareConfig.shapeAnnotationBorderWidth = 1.0;
CPDFKitShareConfig.freehandAnnotationBorderWidth = 1.0;
```

3.4 Forms

3.4.1 Overview

The Form (or AcroForm) feature allows users to create interactive form fields in a PDF document, enabling other users to provide information by filling out these fields. Essentially, PDF form fields are a type of PDF annotation known as Widget annotations. They are utilized to implement interactive form elements such as buttons, checkboxes, combo boxes, and more.

As PDF is an electronic format, it provides advantages that traditional paper forms do not have. For instance, users can edit information that has already been entered. Additionally, document creators can distribute PDF forms over the internet, restrict the content and format entered by users, as well as programmatically extract and categorize the information filled in by users.

Benefits of ComPDFKit Forms

- **Full Types Supported:** Supports all form field types, properties, and appearance settings.
- **Create, Edit, Delete Form Fields:** Perform creation, editing, and deletion operations programmatically or directly through the UI.
- **Fill Form Fields:** Seamlessly fill form fields using the `CPDFView` or automatically fill them programmatically.
- **Form Events:** Trigger specified workflows, enabling automation.
- **Form Flattening:** Permanently adds forms to the document as images, ensuring document appearance stability and preventing further modifications.
- **Fast UI Integration:** Achieve rapid integration and customization through extendable UI components.

Guides for Forms

- Create Form Fields

  Create various interactive form fields.
- Fill Form Fields

  Add content to form fields.

- **Edit Form Fields**
  Edit the content and properties of form fields using code.
- **Delete Form Fields**

  Delete form fields.
- **Flatten Forms**

  Flatten the form to render it in a fixed and non-editable appearance.

3.4.2 Supported Form Field Types

ComPDFKit supports various form field types compliant with the PDF standard, which are readable and writable by various programs, including Adobe Acrobat and other PDF processors that adhere to the standard. The supported form field types are as follows:

Type	Description	Class Name
Check Box	Select one or more options from predefined choices.	`CPDFButtonWidgetAnnotation`
Radio Button	Select one option from predefined choices.	`CPDFButtonWidgetAnnotation`
Push Button	Create custom buttons on the PDF document that acts when pressed.	`CPDFButtonWidgetAnnotation`
List Box	Select one or more options from a predefined list.	`CPDFChoiceWidgetAnnotation`
Combo Box	Select one option from a drop-down list of available text options.	`CPDFChoiceWidgetAnnotation`
Text	Input text content such as name, address, email, etc.	`CPDFTextWidgetAnnotation`
Signature	Digitally sign or electronically sign the PDF document	`CPDFSignatureWidgetAnnotation`

3.4.3 Create Form Fields

Create Text Fields

Text fields allow users to input text in a designated area, commonly used for collecting user information or filling out forms.

The steps to create a text field are as follows:

1. Obtain the page object from CPDFDocument where the text field needs to be created.
2. Create a text field on the page object.
3. Set the position and other properties of the text field.

This example shows how to create a text field:

```swift
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let textWidget = CPDFTextWidgetAnnotation(document: document)
textWidget?.setFieldName("TextField")
textWidget?.isMultiline = false
textWidget?.stringValue = "Basic Text Field"
textWidget?.fontColor = UIColor.black
textWidget?.font = UIFont.systemFont(ofSize: 15)
page?.addAnnotation(textWidget!)
```

```objc
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

CPDFTextWidgetAnnotation *textWidget = [[CPDFTextWidgetAnnotation alloc]
initWithDocument:document];
textWidget.fieldName = @"TextField";
textWidget.isMultiline = NO;
textWidget.stringValue = @"Basic Text Field";
textWidget.fontColor = [UIColor blackColor];
textWidget.font = [UIFont systemFontOfSize:15];
[page addAnnotation:textWidget];
```

Create Buttons

Buttons allow users to perform actions on PDF pages, such as page navigation and hyperlink jumps.

The steps to create a button are as follows:

1. Obtain the page object from CPDFDocument.
2. Create a button on the page object.
3. Set the button's position and appearance properties.
4. Create and set the text properties of the button.
5. Create and set the actions of the button.
6. Update the appearance of the button.

This example shows how to create a button:

```swift
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

//Create button to jump to the second page.
let pushButton1 = CPDFButtonWidgetAnnotation(document: document, controlType:
.pushButtonControl)
pushButton1?.bounds = CGRect(x: 267, y: 300, width: 130, height: 80)
pushButton1?.setFieldName("PushButton1")
pushButton1?.setCaption("PushButton")
pushButton1?.fontColor = UIColor.black
```

```
pushButton1?.font = UIFont.systemFont(ofSize: 15)

let destination1 = CPDFDestination(document: document, pageIndex: 1)
let goToAction1 = CPDFGoToAction(destination: destination1)
pushButton1?.setAction(goToAction1)

page?.addAnnotation(pushButton1!)

//Create button to jump to the web page.
let pushButton2 = CPDFButtonWidgetAnnotation(document: document, controlType:
.pushButtonControl)
pushButton2?.bounds = CGRect(x: 367, y: 303, width: 150, height: 80)
pushButton2?.setFieldName("PushButton2")
pushButton2?.setCaption("PushButton")
pushButton2?.fontColor = UIColor.black
pushButton2?.font = UIFont.systemFont(ofSize: 15)

let urlAction = CPDFURLAction(url: "https://www.compdf.com/")
pushButton2?.setAction(urlAction)

page?.addAnnotation(pushButton2)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

//Create button to jump to the second page.
CPDFButtonWidgetAnnotation *pushButton1 = [[CPDFButtonWidgetAnnotation alloc]
initWithDocument:document controlType:CPDFWidgetPushButtonControl];
pushButton1.bounds = CGRectMake(267, 300, 130, 80);
pushButton1.fieldName = @"PushButton1";
pushButton1.caption = @"PushButton";
pushButton1.fontColor = [UIColor blackColor];
pushButton1.font = [UIFont systemFontOfSize:15];

CPDFDestination * destination = [[CPDFDestination alloc] initWithDocument:document
pageIndex:1];
CPDFGoToAction * goToAction = [[CPDFGoToAction alloc] initWithDestination:destination];
[pushButton1 setAction:goToAction];

[page addAnnotation:pushButton1];

//Create button to jump to the web page.
CPDFButtonWidgetAnnotation *pushButton2 = [[CPDFButtonWidgetAnnotation alloc]
initWithDocument:document controlType:CPDFWidgetPushButtonControl];
pushButton2.bounds = CGRectMake(367, 303, 150, 80);
pushButton2.fieldName = @"PushButton2";
pushButton2.caption = @"PushButton";
pushButton2.fontColor = [UIColor blackColor];
```

```
pushButton2.font = [UIFont systemFontOfSize:15];

CPDFURLAction *urlAction = [[CPDFURLAction alloc]
initWithURL:@"https://www.compdf.com/"];
[pushButton2 setAction:urlAction];

[page addAnnotation:pushButton2];
```

Create List Boxes

List boxes enable users to choose one or multiple items from a predefined list, providing a convenient data selection feature.

The steps to create a list box are as follows:

1. Obtain the page object from CPDFDocument where the list box needs to be created.
2. Create a list box on the page object.
3. Set the position of the list box.
4. Add list items to the list box.
5. Set the properties of the list box.

This example shows how to create a list box:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

var items = [CPDFChoiceWidgetItem]()
let item1 = CPDFChoiceWidgetItem()
item1.value = "List Box No.1"
item1.string = "List Box No.1"
items.append(item1)

let item2 = CPDFChoiceWidgetItem()
item2.value = "List Box No.2"
item2.string = "List Box No.2"
items.append(item2)

let item3 = CPDFChoiceWidgetItem()
item3.value = "List Box No.3"
item3.string = "List Box No.3"
items.append(item3)

let choiceWidget = CPDFChoiceWidgetAnnotation(document: document, listChoice: true)
choiceWidget?.setFieldName("ListBox1")
choiceWidget?.bounds = CGRect(x: 267, y: 100, width: 200, height: 100)
choiceWidget?.items = items
choiceWidget?.selectItemAtIndex = 2

page?.addAnnotation(choiceWidget!)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

NSMutableArray *items = [NSMutableArray array];
CPDFChoiceWidgetItem *item1 = [[CPDFChoiceWidgetItem alloc] init];
item1.value = @"List Box No.1";
item1.string = @"List Box No.1";
[items addObject:item1];
CPDFChoiceWidgetItem *item2 = [[CPDFChoiceWidgetItem alloc] init];
item2.value = @"List Box No.2";
item2.string = @"List Box No.2";
[items addObject:item2];
CPDFChoiceWidgetItem *item3 = [[CPDFChoiceWidgetItem alloc] init];
item3.value = @"List Box No.3";
item3.string = @"List Box No.3";
[items addObject:item3];

CPDFChoiceWidgetAnnotation *choiceWidget = [[CPDFChoiceWidgetAnnotation alloc]
initWithDocument:document listChoice:YES];
choiceWidget.fieldName = @"ListBox1";
choiceWidget.bounds = CGRectMake(267, 100, 200, 100);
choiceWidget.items = items;
choiceWidget.selectItemAtIndex = 2;
[page addAnnotation:choiceWidget];
```

Create Signature Fields

Signature fields allow users to insert digital or electronic signatures into a document, verifying the authenticity and integrity of the document.

The steps to create a signature field are as follows:

1. Obtain the page object from CPDFDocument where the signature field needs to be added.
2. Create a signature field on the page object.
3. Set the position of the signature field.
4. Set the properties of the signature field.

This example shows how to create a signature field:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let signatureWidget = CPDFSignatureWidgetAnnotation(document: document)
signatureWidget?.bounds = CGRect(x: 28, y: 206, width: 80, height: 101)
signatureWidget?.setFieldName("Signature1")
page?.addAnnotation(signatureWidget)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

PDFSignatureWidgetAnnotation *signatureWidget = [[CPDFSignatureWidgetAnnotation alloc]
initWithDocument:document];
signatureWidget.bounds = CGRectMake(28, 206, 80, 101);
signatureWidget.fieldName = @"Signature1";
[page addAnnotation:signatureWidget];
```

Create Check Boxes

Checkboxes allow users to indicate the state of an option by checking or unchecking it.

The steps to create a checkbox are as follows:

1. Obtain the page object from CPDFDocument where the checkbox needs to be added.
2. Create a checkbox on the page object.
3. Set the position of the checkbox.
4. Set the properties of the checkbox.

This example shows how to create a checkbox:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let checkBox = CPDFButtonWidgetAnnotation(document: document, controlType:
.checkBoxControl)
checkBox?.bounds = CGRect(x: 167, y: 351, width: 100, height: 90)
checkBox?.setFieldName("CheckBox2")
checkBox?.borderColor = UIColor.black
checkBox?.backgroundColor = UIColor.green
checkBox?.borderWidth = 2.0
checkBox?.setState(1)
checkBox?.font = UIFont.systemFont(ofSize: 15)
page?.addAnnotation(checkBox)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

CPDFButtonWidgetAnnotation *checkBox = [[CPDFButtonWidgetAnnotation alloc]
initWithDocument:document controlType:CPDFWidgetCheckBoxControl];
checkBox.bounds = CGRectMake(167, 351, 100, 90);
checkBox.fieldName = @"CheckBox2";
checkBox.borderColor = [UIColor blackColor];
checkBox.backgroundColor = [UIColor greenColor];
checkBox.borderWidth = 2.0;
checkBox.state = 1;
checkBox.font = [UIFont systemFontOfSize:15];
[page addAnnotation:checkBox];
```

Create Radio Buttons

Radio buttons allow users to select a unique option from a predefined group of options.

The steps to create a radio button are as follows:

1. Obtain the page object from CPDFDocument where the radio button needs to be added.
2. Create a radio button on the page object.
3. Set the position of the radio button.
4. Set the properties of the radio button.

This example shows how to create a radio button:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let radioButton = CPDFButtonWidgetAnnotation(document: document, controlType:
.radioButtonControl)
radioButton?.bounds = CGRect(x: 167, y: 451, width: 100, height: 90)
radioButton?.setFieldName("RadioButton1")
radioButton?.borderColor = UIColor.black
radioButton?.backgroundColor = UIColor.green
radioButton?.borderWidth = 2.0
radioButton?.setState(0)
radioButton?.font = UIFont.systemFont(ofSize: 15)
page?.addAnnotation(radioButton)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

CPDFButtonWidgetAnnotation *radioButton = [[CPDFButtonWidgetAnnotation alloc]
initWithDocument:document controlType:CPDFWidgetRadioButtonControl];
radioButton.bounds = CGRectMake(167, 451, 100, 90);
radioButton.fieldName = @"RadioButton1";
radioButton.borderColor = [UIColor blackColor];
radioButton.backgroundColor = [UIColor greenColor];
radioButton.borderWidth = 2.0;
radioButton.state = 0;
radioButton.font = [UIFont systemFontOfSize:15];
[page addAnnotation:radioButton];
```

Create Combo Boxes

A combo box is an area where a selected item from the dropdown will be displayed.

The steps to create a combo box are as follows:

1. Obtain the page object from CPDFDocument where the combo box needs to be added.
2. Create a combo box on the page object.
3. Add list items to the combo box.
4. Set the properties of the combo box.

This example shows how to create a combo box:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

var items = [CPDFChoiceWidgetItem]()
let item1 = CPDFChoiceWidgetItem()
item1.value = "Combo Box No.1"
item1.string = "Combo Box No.1"
items.append(item1)

let item2 = CPDFChoiceWidgetItem()
item2.value = "Combo Box No.2"
item2.string = "Combo Box No.2"
items.append(item2)

let item3 = CPDFChoiceWidgetItem()
item3.value = "Combo Box No.3"
item3.string = "Combo Box No.3"
items.append(item3)

let choiceWidget = CPDFChoiceWidgetAnnotation(document: document, listChoice: false)
choiceWidget?.setFieldName("ComboBox1")
choiceWidget?.bounds = CGRect(x: 267, y: 100, width: 200, height: 100)
choiceWidget?.items = items
```

```
choiceWidget?.selectItemAtIndex = 2

page?.addAnnotation(choiceWidget!)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

NSMutableArray *items = [NSMutableArray array];
CPDFChoiceWidgetItem *item1 = [[CPDFChoiceWidgetItem alloc] init];
item1.value = @"Combo Box No.1";
item1.string = @"Combo Box No.1";
[items addObject:item1];
CPDFChoiceWidgetItem *item2 = [[CPDFChoiceWidgetItem alloc] init];
item2.value = @"Combo Box No.2";
item2.string = @"Combo Box No.2";
[items addObject:item2];
CPDFChoiceWidgetItem *item3 = [[CPDFChoiceWidgetItem alloc] init];
item3.value = @"Combo Box No.3";
item3.string = @"Combo Box No.3";
[items addObject:item3];

CPDFChoiceWidgetAnnotation *choiceWidget = [[CPDFChoiceWidgetAnnotation alloc]
initWithDocument:document listChoice:NO];
choiceWidget.fieldName = @"ComboBox1";
choiceWidget.bounds = CGRectMake(267, 100, 200, 100);
choiceWidget.items = items;
choiceWidget.selectItemAtIndex = 2;
[page addAnnotation:choiceWidget];
```

3.4.4 Fill Form Fields

ComPDFKit supports programmatically filling form fields in a PDF document.

Since form fields are a special type of annotation, inheriting from the annotation class, the interface for annotations applies to form fields.

The steps to fill form fields using code are as follows:

1. Obtain the page object from CPDFDocument where you want to fill in the form.
2. Retrieve all annotations from the page object.
3. Iterate through all annotations to find the form to be filled.
4. Modify the form field content as needed.

This example shows how to fill form fields:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)
```

```swift
if let annotations = page?.annotations as? [CPDFAnnotation] {
    for annotation in annotations {
        if let widgetAnnotation = annotation as? CPDFWidgetAnnotation {
            if let textWidget = widgetAnnotation as? CPDFTextWidgetAnnotation {
                textWidget.stringValue = "text"
            } else if let buttonWidget = widgetAnnotation as? CPDFButtonWidgetAnnotation
{
                switch buttonWidget.controlType() {
                case .radioButtonControl:

                    break
                case .pushButtonControl:

                    break
                case .checkBoxControl:

                    break
                default:
                    break
                }
            } else if let choiceWidget = widgetAnnotation as? CPDFChoiceWidgetAnnotation
{

            }
        }
    }
}
```

```objc
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

NSArray *annotations = [page annotations];
for (CPDFAnnotation *annotationz in annotations) {
    if([annotationz isKindOfClass:[CPDFWidgetAnnotation class]]) {
        CPDFWidgetAnnotation * annotation = (CPDFWidgetAnnotation *)annotationz;

        ((CPDFTextWidgetAnnotation*)annotation).stringValue = "text"
      } else if ([annotation isKindOfClass:[CPDFButtonWidgetAnnotation class]]) {
          if (CPDFWidgetRadioButtonControl == [(CPDFButtonWidgetAnnotation *)annotation
controlType]{

          } else if (CPDFWidgetPushButtonControl == [(CPDFButtonWidgetAnnotation
*)annotation controlType]) {

              }
          } else if (CPDFWidgetCheckBoxControl == [(CPDFButtonWidgetAnnotation
*)annotation controlType]{
```

```
            }
        } else if ([annotation isKindOfClass:[CPDFChoiceWidgetAnnotation class]]) {

        }

    }
}
```

3.4.5 Edit Form Fields

Retrieve and edit the appearance and content of form fields.

Note that the properties of different form field types may not be entirely consistent.

The steps to edit a form field are as follows:

1. Obtain the form object to be edited.
2. Modify the form properties.
3. Update the form appearance.

This example shows how to edit form fields:

```
if let widgetAnnotation = annotation as? CPDFWidgetAnnotation {
    if let textWidget = widgetAnnotation as? CPDFTextWidgetAnnotation {
        textWidget.stringValue = "text"
        textWidget.updateAppearanceStream()
    }
}
```

```
CPDFWidgetAnnotation * annotation = (CPDFWidgetAnnotation *)myAnnotation;
((CPDFTextWidgetAnnotation*)annotation).stringValue = "text"
[annotation updateAppearanceStream];
```

3.4.6 Delete Form Fields

Since the form fields class inherits from the annotation class, the way to delete form fields is the same as delete annotations.

The steps to delete form fields are as follows:

1. Obtain the page object from CPDFDocument where you want to delete the form field.
2. Delete the form field.

This example shows how to delete form fields:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let annotation = page?.annotations[0]
page?.removeAnnotation(annotation)
```

```
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath"]];
CPDFPage *page = [document pageAtIndex:0];

CPDFAnnotation *annotation = [[page annotations] objectAtIndex:0];
[page removeAnnotation:annotation];
```

3.4.7 Flatten Forms

Form flattening refers to the process of converting editable form fields into non-editable, static images, or pure text. When flattening form fields, all editable elements in the entire document (including annotations and forms) undergo flattening. Therefore, form flattening is also referred to as document flattening.

This example shows how to flatten forms:

```
var document = CPDFDocument(url: URL(string: "filePath"))
document?.writeFlatten(to: URL(string: "savePath"))
```

```
CPDFDocument *importDocument = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:"filePath"]];
document.WriteFlattenToFilePath("savePath");
```

What is Document Flattening?

Document flattening refers to the process of converting editable elements, such as annotations, form fields, or layers, in a PDF document into non-editable, static images, or pure text. The purpose of this process is to lock the final state of the document, eliminating editable elements.

Document flattening is typically applied in the following contexts:

1. **Content Protection:** Flattening can be used to protect document content, ensuring that the document remains unaltered during distribution or sharing. This is crucial for maintaining document integrity and confidentiality.
2. **Form Submission:** In form processing, flattening can convert user-filled form fields and annotations into static images for easy transmission, archiving, or printing, while preventing modifications to the form content in subsequent stages.
3. **Compatibility and Display:** Some PDF readers or browsers may encounter issues with displaying and interacting with PDF documents that contain numerous annotations or layers. Document flattening helps address these compatibility issues, enhancing the visual representation of documents in various environments.

4. **File Size Reduction:** Flattened documents typically have reduced file sizes since editable elements are converted into static images or text, eliminating the need to store additional data for editing information.

3.5 Document Editor

3.5.1 Overview

The document editing functionality offers a range of capabilities to manipulate pages, allowing users to control the document structure and adjust the layout and formatting, ensuring that the document content is presented accurately and in a well-organized manner.

Benefits of ComPDFKit Document Editor

- **Insertion or Deletion of Pages:** Insert or delete pages within the document to meet specific layout requirements.
- **Document Merging and Splitting:** Combine multiple documents or pages into a new document, or split a large document into smaller ones.
- **Structural Adjustments:** Adjust the sequence or rotate the orientation of pages to meet specific display or printing needs.
- **Multi-Document Collaboration:** Extract pages from one document and insert them into another, facilitating collaboration and content integration.
- **Fast UI Integration:** Achieve rapid integration and customization through extensible UI components.

Guides for Document Editor

- **Insert Pages**

  Insert blank pages, images, or pages from another document into the target document.
- **Split Pages**

  Divide a portion of a multi-page document into independent documents.
- **Merge Pages**

  Combine pages from multiple documents into a single document.
- **Delete Pages**

  Remove pages from the document.
- **Rotate Pages**

  Rotate pages within a PDF document.
- **Replace Pages**

  Replace specified pages in the target document with pages from another document.
- **Extract Pages**

  Extract pages from the document.

3.5.2 Insert Pages

Insert a blank page or pages from other PDFs into the target document.

Insert Blank Pages

This example shows how to insert a blank page:

```
// The `InsertPage` method allows specifying an image path, and when the image path is empty, it inserts a blank page.
document?.insertPage(CGSize(width: 595, height: 852), withImage: "", at: 0)
```

```
// The `InsertPage` method allows specifying an image path, and when the image path is empty, it inserts a blank page.
[document insertPage:CGSizeMake(595, 852) withImage:"" atIndex:0]
```

Insert Pages from other PDFs

This example shows how to pages from other PDFs:

```
let insertDocument = CPDFDocument(url: URL(fileURLWithPath: "OtherPDF.pdf"))
var indexSet = IndexSet()
indexSet.insert(0)

document?.importPages(indexSet, from: insertDocument, at: 0)
```

```
CPDFDocument *insertDocument = [[CPDFDocument alloc] initWithURL:[NSURL fileURLWithPath:@"OtherPDF.pdf"]];
NSMutableIndexSet *indexSet = [NSMutableIndexSet indexSet];
[indexSet addIndex:0];

[document importPages:indexSet fromDocument:insertDocument atIndex:0];
```

3.5.3 Split Pages

The steps to split pages are as follows:

1. Create a new `CPDFDocument` object for each part to be split.
2. Add the portions of the target document that need to be split to the newly created `CPDFDocument` objects.

This example shows how to split pages:

```
// Split out all pages in a loop
for i in 0..<(document?.pageCount ?? 0) {
    let splitDocument = CPDFDocument()
    let index = IndexSet(integer: IndexSet.Element(i))

    splitDocument?.importPages(index, from: document, at: 0)

    if let writeURL = URL(string: "writeFilePath") {
        splitDocument?.write(to: writeURL)
    }
}
```

```
// Split out all pages in a loop
for (int i = 0; i < document.pageCount; i++) {
      CPDFDocument *splitDocument = [[CPDFDocument alloc] init];
      NSIndexSet *index = [[NSIndexSet alloc] initWithIndex:i];

      [splitDocument importPages:index fromDocument:document atIndex:0];
      [splitDocument writeToURL:[NSURL fileURLWithPath:@"writeFilePath"]];
 }
```

3.5.4 Merge Pages

The steps to merge pages are as follows:

1. Create a blank PDF document.
2. Open the PDF document containing the pages to be merged.
3. Select all the pages to be merged and combine them into the same document.

This example shows how to merge pages:

```
let mergeDocument = CPDFDocument()

if let document1 = CPDFDocument(url: URL(fileURLWithPath: "filePath1")),
   let document2 = CPDFDocument(url: URL(fileURLWithPath: "filePath2")) {

    let indexSet = IndexSet(integersIn: 0..<2)

    mergeDocument?.importPages(indexSet, from: document1, at: mergeDocument?.pageCount ??
0)//Insert the first two pages.
    mergeDocument?.importPages(indexSet, from: document2, at: mergeDocument?.pageCount ??
0)
}
```

```
CPDFDocument *mergeDocument = [[CPDFDocument alloc] init];

CPDFDocument *document1 = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath1"]];
CPDFDocument *document2 = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"filePath2"]];

NSMutableIndexSet *indexSet = [NSMutableIndexSet indexSet];
[indexSet addIndex:0];
[indexSet addIndex:1];
[mergeDocument importPages:indexSet fromDocument:document1
atIndex:mergeDocument.pageCount];//Insert the first two pages.
[mergeDocument importPages:indexSet fromDocument:document2
atIndex:mergeDocument.pageCount];
```

3.5.5 Delete Pages

This example shows how to delete a page:

```
var indexSet = IndexSet()
indexSet.insert(0)

document?.removePage(at: indexSet)// Delete the first page of the document.
```

```
NSMutableIndexSet *indexSet = [NSMutableIndexSet indexSet];
[indexSet addIndex:0];

[document removePageAtIndexSet:indexSet];// Delete the first page of the document.
```

3.5.6 Rotate Pages

This example shows how to rotate pages:

```
let page = document?.page(at: 0)
// Rotate the first page 90 degrees clockwise
page?.rotation += 90
```

```
CPDFPage *page = [document pageAtIndex:0];
// Rotate the first page 90 degrees clockwise
page.rotation += 90;
```

3.5.7 Replace Pages

The steps to replace pages are as follows:

1. Remove the pages in the target file that need to be replaced.
2. Insert the replacement pages into the location where the original document was deleted.

This example shows how to replace pages:

```
var indexSet = IndexSet()
indexSet.insert(0)
// Remove the first page from the document.
document?.removePage(at: indexSet)
let insertDocument = CPDFDocument(url: URL(fileURLWithPath: "OtherPDF.pdf"))
var indexSet = IndexSet()
indexSet.insert(0)
// Insert the first page of another document into the original document's first-page
position to complete the replacement.
document?.importPages(indexSet, from: insertDocument, at: 0)
```

```
NSMutableIndexSet *indexSet = [NSMutableIndexSet indexSet];
[indexSet addIndex:0];
// Remove the first page from the document.
[document removePageAtIndexSet:indexSet];
CPDFDocument *insertDocument = [[CPDFDocument alloc] initWithURL:[NSURL
fileURLWithPath:@"OtherPDF.pdf"]];
NSIndexSet *inserSet = [[NSIndexSet alloc]  initWithIndex:0];
// Insert the first page of another document into the original document's first-page
position to complete the replacement.
[document importPages:inserSet fromDocument:insertDocument atIndex:0];
```

3.5.8 Extract Pages

This example shows how to extract pages:

```
var indexSet = IndexSet()
indexSet.insert(0)

let extractDocument = CPDFDocument()
// Extract the first page of the original document and save it in a new document.
extractDocument?.importPages(index, from: document, at: 0)
```

```
NSMutableIndexSet *indexSet = [NSMutableIndexSet indexSet];
[indexSet addIndex:0];

CPDFDocument *extractDocument = [[CPDFDocument alloc] init];
// Extract the first page of the original document and save it in a new document.
[extractDocument importPages:indexSet fromDocument:document atIndex:0];
```

3.6 Security

3.6.1 Overview

The security module provides features including password protection, permission settings, Bates coding, background, and page header and footer functionalities. Document security is guaranteed by managing document passwords and permissions, and by adding logos and copyright information.

Benefits of ComPDFKit Security

- **Access Control:** Restrict sensitive permissions such as access, copying, or printing by configuring security permissions associated with the document.
- **Password Management:** Create, modify, or remove document passwords and permissions.
- **Encryption Standards:** Support for standard PDF security procedures (40 and 128-bit RC4 encryption) as well as 128 and 256-bit AES (Advanced Encryption Standard) encryption.
- **Copyright Identification:** Display document source and copyright information through document background and header/footer, preventing unauthorized screen captures or misuse.
- **Dynamic Identification:** Automatically add Bates coding and header/footer associated with document content through specific expressions.

Guides for Security

- **PDF Permissions**

  By managing document passwords and permission settings, unauthorized access is prevented, and control over user operational permissions for the document is maintained.

- **Background**

  Setting an image or color background not only enhances the document's visual appeal but also safeguards its privacy, preventing unauthorized copying or distribution.

- **Header and Footer**

  Enhance document readability and professionalism by adding marks in the header and footer. Incorporate information such as titles, page numbers, and brand identifiers to facilitate document navigation and recognition.

- **Bates Numbers**

  Insert Bates numbers in the document's header, footer, or other designated locations. Assign unique identification numbers to each page, facilitating document tracking, organization, and legal references.

3.6.2 PDF Permissions

PDF Permissions are employed to ensure the security of PDF documents, offering encryption, document permissions, decryption, and password removal features. This ensures users have secure control and effective management over the document.

Encrypt

Encrypt function consists of two parts: User Password and Owner Password.

The User Password is utilized to open the document, ensuring that only authorized users can access its content. When a user password is set, it typically restricts certain document permissions such as modification, copying, or printing. On the other hand, the Owner Password not only opens the document but also unlocks all restricted permissions, allowing users to modify, copy, or print the document. The dual-password system aims to provide a more flexible and secure approach to document access and management.

ComPDFKit offers a variety of encryption algorithms and permission settings. Depending on your requirements, you can use the appropriate algorithm and configure custom permissions to safeguard your document.

The steps to encrypt are as follows:

1. Set distinct user passwords and owner passwords.
2. Create a permissions information class.
3. Specify the encryption algorithm.
4. Encrypt the document using the user password, owner password, permission information, and the chosen algorithm.

This sample shows how to encrypt a document:

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let surl = URL(fileURLWithPath: "")
let options: [CPDFDocumentWriteOption: Any] = [
      .ownerPasswordOption: "userPassword",
      .userPasswordOption: "ownerPassword",
      .encryptionLevelOption: CPDFDocumentEncryptionLevel.AES128,
      .allowsPrintingOption: true,
      .allowsCopyingOption: false
  ]

document?.write(to: surl, withOptions: options)
```

```
NSURL *url = [NSURL fileURLWithPath:@"File Path"];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSURL *surl = [NSURL fileURLWithPath:@""];
NSDictionary *options = @{CPDFDocumentOwnerPasswordOption : @"userPassword",
                          CPDFDocumentUserPasswordOption : @"ownerPassword",
                          CPDFDocumentEncryptionLevelOption :
@(CPDFDocumentEncryptionLevelAES256),
                          CPDFDocumentAllowsPrintingOption : @(YES),
                          CPDFDocumentAllowsCopyingOption : @(NO),
                          };
[document writeToURL:surl withOptions:options];
```

Encryption Algorithm and its Description:

Algorithm	Description	Enumeration Values
None	No encryption	CPDFDocumentEncryptionLevel.CPDFDocumentNoEncryptAlgo
RC4	Encrypts plaintext using XOR with key	CPDFDocumentEncryptionLevel.CPDFDocumentRC4
AES-128	Encrypts using AES algorithm with 128-bit key	CPDFDocumentEncryptionLevel.CPDFDocumentAES128
AES-256	Encrypts using AES algorithm with 256-bit key	CPDFDocumentEncryptionLevel.CPDFDocumentAES256

PDF Permissions

In the PDF specification, there is support for configuring various permissions for a document. By configuring these permissions, it is possible to restrict users to perform only the expected actions.

The PDF specification defines the following permissions:

- Print: Allows printing of the document.
- High-Quality Print: Permits high-fidelity printing of the document.
- Copy: Enables copying of the document content.
- Modify Document: Allows modification of the document content, excluding document properties.
- Assemble Document: Permits insertion, deletion, and rotation of pages.
- Annotate: Enables the creation or modification of document annotations, including form field entries.
- Form Field Input: Allows modification of form field entries, even if document annotations are not editable.

The steps to view document permissions are as follows:

1. Retrieve document permission information.
2. Use the document permission information to view specific permissions.

This example shows how to view document permissions:

```swift
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

var commandLineStr = ""
commandLineStr.append("Printing: \(document!.allowsPrinting ? "true" : "false")\n")
commandLineStr.append("Content Copying: \(document!.allowsCopying ? "true" : "false")\n")
```

```objc
NSURL *url = [NSURL fileURLWithPath:@"File Path"];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSString *commandLineStr = "";
[commandLineStr stringByAppendingFormat:@"Printing: %@\n", document.allowsPrinting ?
@"true" : @"flase"];
commandLineStr = [commandLineStr stringByAppendingFormat:@"Content Copying:: %@\n",
document.allowsCopying ? @"true" : @"flase"];
```

Decrypt

Accessing a password-protected PDF document requires entering the password. Different levels of passwords provide varying levels of permission.

The steps for decryption are as follows:

1. When opening the document, check if it is encrypted.
2. For encrypted documents, entering either the user password or the owner password allows the document to be opened.

This example shows how to decrypt a document:

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

if document?.isLocked == true {
    document?.unlock(withPassword: "password")
}
```

```
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

if (document.isLocked) {
    [document unlockWithPassword:@"password"];
}
```

Remove passwords

Removing passwords means deleting the owner passwords and user passwords from a document and saving it as a new document, which will no longer require a password to open and will have all permissions available by default.

The steps to remove passwords are as follows:

1. Unlock the document to obtain all permissions.
2. Save the unlocked document.

This example shows how to remove passwords:

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

if document?.isLocked == true {
    let surl = URL(fileURLWithPath: "")
    document?.unlock(withPassword: "password")
    document?.writeDecrypt(to: surl)
}
```

```
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

if (document.isLocked) {
    NSURL *surl = [NSURL fileURLWithPath:@""];
    [document unlockWithPassword:@"password"];
    [document writeDecryptToURL:surl];
}
```

3.6.3 Background

The background refers to the underlying layer or pattern on the document pages, used to present the fundamental visual effect of the document. Adding a background can alter the document's appearance, making it more personalized or professional. It can be used to emphasize a brand, protect copyright, or enhance the reading experience of the document.

In a PDF document, only one background can exist, and adding a new background to pages containing an existing background will overwrite the old background.

Set Color Background

The steps to set a color background are as follows:

1. Obtain the document's background object.
2. Set the background type to color.
3. Configure the properties of the background.
4. Update the background of the document.

This example shows how to set the color background:

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let background = document?.background()
background?.type = .color
background?.color = .black // Background color (image background not applicable).
background?.opacity = 1.0 // Background transparency, with a range of values from 0 to 1, defaulting to 1.
background?.scale = 1.0 // Background tiling ratio.
background?.rotation = 0 // Background rotation angle, with a range of values from 0 to 360, defaulting to 0 (rotation around the page center).
background?.horizontalAlignment = 1 // Horizontal alignment of the background. `0` represents left alignment, `1` represents center alignment, and `2` represents right alignment.
background?.verticalAlignment = 1 // Vertical alignment of the background. `0` represents top alignment, `1` represents center alignment, and `2` represents bottom alignment.
background?.xOffset = 0 // Horizontal offset of the background. Positive values indicate a rightward offset, while negative values indicate a leftward offset.
background?.yOffset = 0 // Vertical offset of the background. Positive values indicate a downward offset, while negative values indicate an upward offset.
```

```
background?.pageString = "0,1,2" // Background page range, for example, "0,3,5-7".
background?.update()
```

```
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

CPDFBackground *background = document.background;
background.type = CPDFBackgroundTypeColor;
background.color = [UIColor blackColor]; // Background color (image background not applicable).
background.opacity = 1.0; // Background transparency, with a range of values from 0 to 1, defaulting to 1.
background.scale = 1.0; // Background tiling ratio.
background.rotation = 0; // Background rotation angle, with a range of values from 0 to 360, defaulting to 0 (rotation around the page center).
background.horizontalAlignment = 1; // Horizontal alignment of the background. `0` represents left alignment, `1` represents center alignment, and `2` represents right alignment.
background.verticalAlignment = 1; // Vertical alignment of the background. `0` represents top alignment, `1` represents center alignment, and `2` represents bottom alignment.。
background.xOffset = 0; // Horizontal offset of the background. Positive values indicate a rightward offset, while negative values indicate a leftward offset.
background.yOffset = 0; // Vertical offset of the background. Positive values indicate a downward offset, while negative values indicate an upward offset.
background.pageString = @"0,1,2"; // Background page range, for example, "0,3,5-7".
[background update];
```

Set Image Background

Setting the image background involves the following steps:

1. Obtain the document background object.
2. Set the background type to an image.
3. Specify the background properties.
4. Update the background on the document.

This example shows how to set the image background:

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let imagePath = ""
if let image = UIImage(contentsOfFile: imagePath) {
    let background = document?.background()
    background?.setImage(image)
    background?.type = .image
    background?.opacity = 1.0 // Background transparency, with a range of values from 0 to 1, defaulting to 1.
```

```
    background?.scale = 1.0 // Background image repetition.
    background?.rotation = 0 // Background rotation angle, with a range of values from 0 to 360, defaulting to 0 (rotation around the page center).
    background?.horizontalAlignment = 1 // Vertical alignment of the background. `0` represents top alignment, `1` represents center alignment, and `2` represents bottom alignment.
    background?.verticalAlignment = 1 // Horizontal alignment of the background. `0` represents left alignment, `1` represents center alignment, and `2` represents right alignment.
    background?.xOffset = 0 // Horizontal offset of the background. Positive values indicate a rightward offset, while negative values indicate a leftward offset.
    background?.yOffset = 0 // Vertical offset of the background. Positive values indicate a downward offset, while negative values indicate an upward offset.
    background?.pageString = "0,1,2" // Set the background page range through a string, for example, "0,3,5-7".
    background?.update()
}
```

```
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSString *imagePath = @"";
UIImage *image = [[UIImage alloc] initWithContentsOfFile:imagePath];

CPDFBackground *background = document.background;
[background setImage:image];
background.type = CPDFBackgroundTypeImage;
background.opacity = 1.0; // Background transparency, with a range of values from 0 to 1, defaulting to 1.
background.scale = 1.0; // Background image repetition.
background.rotation = 0; // Background rotation angle, with a range of values from 0 to 360, defaulting to 0 (rotation around the page center).
background.horizontalAlignment = 1; // Vertical alignment of the background. `0` represents top alignment, `1` represents center alignment, and `2` represents bottom alignment.
background.verticalAlignment = 1; // Horizontal alignment of the background. `0` represents left alignment, `1` represents center alignment, and `2` represents right alignment.
background.xOffset = 0; // Horizontal offset of the background. Positive values indicate a rightward offset, while negative values indicate a leftward offset.
background.yOffset = 0; // Vertical offset of the background. Positive values indicate a downward offset, while negative values indicate an upward offset.
background.pageString = @"0,1,2"; // Set the background page range through a string, for example, "0,3,5-7".
[background update];
```

Remove Background

Removing the background follows these steps:

1. Obtain the document background object.
2. remove the document background.

The steps to remove the background are as follows:

```swift
let background = document?.background()

background?.clear()
```

```objc
CPDFBackground *background = [document background];

[background clear];
```

3.6.4 Header and Footer

Header and Footer refer to annotations added at the top and bottom of a document, typically containing information such as titles, page numbers, and brand identification. By including headers and footers in a document, it becomes easier to navigate and identify, thereby enhancing the document's readability and professionalism.

In a PDF document, only one header and footer can exist, and adding a new header and footer will overwrite the old header and footer.

Add Header and Footer

The steps to add headers and footers are as follows:

1. Retrieve the header and footer objects within the document.
2. Specify the pages where headers and footers should be added.
3. Set attributes such as color and font size for the headers and footers.
4. Update the headers and footers on the pages.

This example shows how to add header and footer:

```swift
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let headerFooter = document?.headerFooter()
headerFooter?.setText("<<1,2>> page", at: 0)
headerFooter?.setTextColor(UIColor.red, at: 0)
headerFooter?.setFontSize(14.0, at: 0)
headerFooter?.update()
```

```
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

CPDFHeaderFooter *headerFooter = document.headerFooter;
[headerFooter setText:@"<<1,2>> page" atIndex:0];
[headerFooter setTextColor:[UIColor redColor] atIndex:0];
[headerFooter setFontSize:14.0 atIndex:0];
[headerFooter update];
```

Header and Footer Regular Expression Explanation

Headers and footers support format-specific regular expressions that take effect when `headerFooter.setRules(1)` is set, in the format: <<\d+,\d+>>|<<\d+>>|<\d+,>>

- <<i>> : `i` is the starting value of the page number.
- <<i,f>>: `i` is the starting value of the page number, and `f` is the number of digits in the page number, if the actual page number is not enough, it will be automatically filled with 0 in front.

eg: When text is set to "<<1,2>> page", the text displayed on the first page is "01 page".

Remove Header and Footer

Steps to remove headers and footers:

1. Retrieve the header and footer objects from the document.
2. Remove the headers and footers.

This example shows how to remove the header and footer:

```
let headerFooter = document?.headerFooter()
headerFooter?.clear()
```

```
CPDFHeaderFooter *headerFooter = document.headerFooter;
[headerFooter clear];
```

3.6.5 Bates Numbers

ComPDFKit provides a comprehensive API for adding, editing, and deleting Bates numbers in PDF documents, facilitating the management of user security information.

In a PDF document, only one Bates number can exist, and adding a new Bates number will overwrite the old Bates number.

Add Bates Numbers

The steps to add Bates numbers are as follows:

1. Retrieve the Bates numbers object from the document.
2. Set the properties for the Bates numbers to be added.
3. Update the Bates numbers in the document.

This example shows how to add Bates numbers:

```swift
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let bates = document?.bates()
bates?.setText("<<#3#5#Prefix-#-Suffix>>", at: 0)
bates?.setTextColor(UIColor.red, at: 0)
bates?.setFontSize(14.0, at: 0)
bates?.update()
```

```objc
NSURL *url = [NSURL fileURLWithPath:@""];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

CPDFBates *bates = document.bates;
[bates setText:@"<<#3#5#Prefix-#-Suffix>>" atIndex:0];
[bates setTextColor:[UIColor redColor] atIndex:0];
[bates setFontSize:14.0 atIndex:0];
[bates update];
```

Bates Numbers Regular Expression Explanation

The regular expression for Bates numbers supports a specific format: <<#\d+#\d+#\w+#\w+>>

- The first # is followed by the minimum number of digits to display for the page number. If the page number has fewer digits, leading zeros are added.
- The second # is followed by the starting value of the page number.
- The third # is followed by the prefix for the header/footer.
- The fourth # is followed by the suffix for the header/footer.

For example: When the text is set to "<<#3#1#ab#cd>>," the text displayed on the first page will be "ab001cd."

Remove Bates Numbers

The steps to delete Bates numbers are as follows:

1. Retrieve the Bates numbers object from the document.
2. Remove the Bates numbers.

This example shows how to remove the Bates numbers:

```swift
let bates = document?.bates()
bates?.clear()
```

```objc
CPDFBates *bates = document.bates;
[bates clear];
```

3.7 Redaction

3.7.1 Overview

Redaction is the process of removing visible text or images from a document, often permanently deleting or overwriting sensitive information and completely removing it from the PDF metadata. Revised content is often replaced with black bars or other visually prominent markings to indicate that the information has been redacted.

Redaction is commonly employed to safeguard privacy, comply with legal requirements, or ensure that sensitive content is not inadvertently disclosed. This process facilitates secure distribution to audiences such as courts, patent and government agencies, media, clients, suppliers, or any other restricted-access entities.

Benefits of ComPDFKit Redaction

- **Text Encryption:** Sensitive information is rendered unreadable by overlaying it with colored blocks or specific characters.
- **Image Encryption:** Protect confidential information by hiding or removing image areas with black boxes or special characters.
- **Permanent Alteration:** Redaction is a permanent process, ensuring that sensitive information cannot be recovered.
- **Metadata Purge:** In addition to removing visible content, any hidden metadata in the PDF that may contain sensitive information is also eliminated.

Guides for Redaction

- **Redact PDFs**

  Redact specific information PDF documents.

3.7.2 Redact PDFs

Redact content from a PDF document involves two steps:

1. **Create Redaction Annotations:** A user applies to redact annotations that specify the pieces or regions of content that should be removed. This step marks the regions where redaction changes will be applied. These redaction annotations can be set, moved, or deleted before applying redaction changes, without removing content from the document.
2. **Apply Redaction Changes:** After marking the areas for complete content removal, apply redaction changes. The content within the regions of the redaction annotations will be irreversibly deleted.

Through these two steps, you can thoroughly remove sensitive data from the document.

Create Redaction Annotations

You can use the `CPDFRedactAnnotation` class to create redaction annotations. Utilize the `quadrilateralPoints` or `bounds` methods to define the areas that should be covered by the redaction annotations.

In addition, the appearance of the ciphertext can be customized at this stage.
Note that once redaction annotations have been applied, their appearance cannot be changed. This is because the redaction annotations will become non-editable, non-modifiable, static content instead of an interactive ciphertext annotation after it has been applied.

This sample shows how to create a redaction annotation：

```
let url = URL(fileURLWithPath: "File Path")
let document = CPDFDocument(url: url)

let page = document?.page(at: 0)
let redact = CPDFRedactAnnotation(document: document)
redact?.bounds = CGRect(x: 0, y: 0, width: 50, height: 50)
redact?.setOverlayText("REDACTED")
redact?.setFont(UIFont.systemFont(ofSize: 12))
redact?.setFontColor(UIColor.red)
redact?.setAlignment(.left)
redact?.setInteriorColor(.black)
redact?.setBorderColor(.yellow)
page?.addAnnotation(redact)
```

```
NSURL *url = [NSURL fileURLWithPath:pdfPath];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
CPDFPage *page = [document pageAtIndex:0];

CPDFRedactAnnotation *redact = [[CPDFRedactAnnotation alloc] initWithDocument:document];
redact.bounds = CGRectMake(0, 0, 50, 50);
redact.overlayText = @"REDACTED";
redact.font = [UIFont systemFontOfSize:12];
redact.fontColor = [UIColor redColor];
redact.alignment = NSTextAlignmentLeft;
redact.interiorColor = [UIColor blackColor];
redact.borderColor = [UIColor yellowColor];
[page addAnnotation:redact];
```

Apply Redaction Changes

ComPDFKit SDK ensures that if text, images, or vector graphics are included in the region marked by a redaction annotation, the corresponding image or path data in that portion will be completely removed and cannot be restored.

This sample shows how to apply redaction changes：

```
redact?.applyRedaction()
```

```
[redact applyRedaction];
```

3.8 Watermark

3.8.1 Overview

A watermark is a mark placed on a PDF document, typically consisting of semi-transparent text or image elements added to the PDF. By adding a watermark, document source and copyright information can be displayed without disrupting readability.

Benefits of ComPDFKit PDF Watermark SDK

- **Copyright Notice:** Display document source and copyright information through a watermark to prevent screenshots or unauthorized use.
- **Branding:** Customize the appearance of the watermark to showcase a brand logo or other predefined content.
- **Fast UI Integration:** Achieve rapid integration and customization through extensible UI components.

Guides for Watermark

- Add Text Watermark

  Create a text watermark with custom content and appearance.
- Add Image Watermark

  Generate an image watermark using a custom image.
- Delete Watermark

  Remove the watermark attached to the PDF document.

3.8.2 Add Text Watermark

The steps to add a text watermark are as follows:

1. Initialize a `CPDFWatermark` object, specifying the watermark type as text.
2. Set the properties required for the text watermark, including content, font, color, and font size.
3. Set the general properties for the watermark.
4. Create the watermark in the document.

This example shows how to add a text watermark:

```
// Initialize a CPDFWatermark object, specifying the type as text.
let watermark = CPDFWatermark(document: document, type: .text)

// Set the text content, font, color, and font size properties.
watermark.text = "ComPDFKit"
watermark.textFont = UIFont(name: "Helvetica", size: 30)
watermark.textColor = UIColor.red
watermark.scale = 2.0
watermark.rotation = 45
watermark.opacity = 0.5
watermark.verticalPosition = .center
watermark.horizontalPosition = .center
```

```
watermark.tx = 0.0
watermark.ty = 0.0
watermark.isFront = true
watermark.isTilePage = false
watermark.pageString = "0-4"

document.addWatermark(watermark)
document.updateWatermark(watermark)

// Create a watermark in the document.
document.write(to: self.addTextWatermarkURL)
```

```
//Initialize a CPDFWatermark object, specifying the type as text.
CPDFWatermark *watermark = [[CPDFWatermark alloc] initWithDocument:document
type:CPDFWatermarkTypeText];
//Set the text content, font, color, and font size properties.
watermark.text = @"ComPDFKit";
watermark.textFont = [UIFont fontWithName:@"Helvetica" size:30];
watermark.textColor = [UIColor redColor];
watermark.scale = 2.0;

watermark.rotation = 45;
watermark.opacity = 0.5;
watermark.verticalPosition = CPDFWatermarkVerticalPositionCenter;
watermark.horizontalPosition = CPDFWatermarkHorizontalPositionCenter;
watermark.tx = 0.0;
watermark.ty = 0.0;
watermark.isFront = YES;
watermark.isTilePage = NO;
watermark.pageString = @"0-4";
[document addWatermark:watermark];
[document updateWatermark:watermark];
//Create a watermark in the document.
[document writeToURL:self.addTextWatermarkURL];
```

3.8.3 Add Image Watermark

The steps to add an image watermark are as follows:

1. Initialize a `CPDFWatermark` object, specifying the watermark type as an image.
2. Create a Bitmap based on the image file, setting the image source and scaling for the image watermark.
3. Set the general properties for the watermark.
4. Create the watermark in the document.

This example shows how to add an image watermark:

```
// Initialize a CPDFWatermark object, specifying the watermark type as an image.
```

```swift
let watermark = CPDFWatermark(document: document, type: .image)

watermark.image = UIImage(named: "Logo")
// Create a Bitmap based on the image file, and set the image source and scaling ratio
for the image watermark.

// Configure general properties for the watermark.
watermark.scale = 2.0
watermark.rotation = 45
watermark.opacity = 0.5
watermark.verticalPosition = .center
watermark.horizontalPosition = .center
watermark.tx = 0.0
watermark.ty = 0.0
watermark.isFront = true
watermark.pageString = "0-4"

document.addWatermark(watermark)
// Create a watermark in the document.
document.write(to: self.addImageWatermarkURL)
```

```objc
//Initialize a CPDFWatermark object, specifying the watermark type as an image.
CPDFWatermark *watermark = [[CPDFWatermark alloc] initWithDocument:document
type:CPDFWatermarkTypeImage];

watermark.image = [UIImage imageNamed:@"Logo"];
//Create a Bitmap based on the image file, and set the image source and scaling ratio for
the image watermark.
//Configure general properties for the watermark.
atermark.scale = 2.0;

watermark.rotation = 45;
watermark.opacity = 0.5;
watermark.verticalPosition = CPDFWatermarkVerticalPositionCenter;
watermark.horizontalPosition = CPDFWatermarkHorizontalPositionCenter;
watermark.tx = 0.0;
watermark.ty = 0.0;
watermark.isFront = YES;
watermark.pageString = @"0-4";
[document addWatermark:watermark];
//Create a watermark in the document.
[document writeToURL:self.addImageWatermarkURL];
```

3.8.4 Delete Watermark

To delete watermarks, follow these steps:

Call `DeleteWatermarks()` on the `CPDFDocument` object to remove all watermarks from the document.

This example shows how to delete the watermark:

```
//Remove the first watermark from the document.
let watermarks = document?.watermarks()
document?.removeWatermark(watermarks[0])
```

```
//Remove the first watermark from the document.
NSArray *watermarks = [document watermarks];
[document removeWatermark:watermarks[0]];
```

3.9 Conversion

3.9.1 PDF/A

ComPDFKit SDK supports analyzing the content of existing PDF files and making a series of modifications to generate documents compliant with the PDF/A standard.

During the process of converting a PDF file to one that complies with the PDF/A standard, features unsuitable for long-term archiving, such as encryption, outdated compression schemes, missing fonts, or device-dependent colors, will be replaced with equivalent elements that conform to the PDF/A standard. Since the conversion process applies only the necessary changes to the source file, minimal information loss occurs.

This example shows how to convert an existing PDF file into a document compliant with the PDF/A-1a standard:

```
let url = URL(string: pdfPath)
var document = CPDFDocument(url: url)

document?.writePDFA(to: url, with: .pdfa1a)
```

```
NSURL *url = [NSURL fileURLWithPath:pdfPath];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
[document writePDFAToURL:url withType:CPDFTypePDFA1a];
```

What is PDF/A?

PDF/A (Portable Document Format Archival) is a PDF file format standard specifically designed for the long-term preservation of electronic documents. Its purpose is to ensure that documents maintain readability and accessibility over time, meeting the requirements for archiving and long-term preservation. The PDF/A standard is defined by the International Organization for Standardization (ISO) and has become the international standard for electronic document archiving.

PDF/A Versions

Version	Release Date	Standard	Based On
PDF/A-1	2005-09-28	ISO 19005-1 certified	PDF 1.4
PDF/A-2	2011-06-20	ISO 19005-2 certified	PDF 1.7 (ISO 32000-1: 2008)
PDF/A-3	2012-10-15	ISO 19005-3 certified	PDF 1.7 (ISO 32000-1: 2008)
PDF/A-4	2020-11	ISO 19005-4 certified	PDF 2.0 (ISO 32000-2: 2020)

Features of PDF/A

Self-Containment: PDF/A documents should be self-contained, meaning they include all necessary elements and resources, such as fonts, images, and other embedded files. This ensures that the document can be rendered and displayed independently in different environments.

Font Embedding: PDF/A requires that fonts used in the document must be embedded to prevent issues with missing fonts on different systems, ensuring correct document display.

No Compression: PDF/A typically disallows the use of compression algorithms that are unstable for long-term preservation, such as JPEG2000. This helps ensure the reliability and stability of the document.

No Encryption: PDF/A mandates that documents cannot use encryption to guarantee future accessibility and readability. This also ensures that documents are not password-protected and inaccessible for decryption.

Metadata: PDF/A encourages or requires the inclusion of metadata, such as document information, author, title, etc., to provide basic description and management information for the document.

Color Management: The PDF/A standard provides support for color management to ensure a consistent display of colors across different devices.

3.10 Content Editor

3.10.1 Overview

Content editing provides the ability to edit text and images, allowing users to easily insert, adjust, and delete text or images. It makes PDF software similar to a rich text editor, enabling direct composition and modification.

Benefits of ComPDFKit Content Editor

- **Text Editing:** Freely insert or delete text, or adjust text properties and positions.
- **Image Editing:** Freely insert or delete images, or adjust image properties and positions.
- **Path Editing:** Edit existing path objects in a document, such as cut, copy, move, delete, and other operations.
- **Undo and Redo:** Undo or redo any changes.
- **Custom Menus:** Customize content editing context menus.
- **Fast UI Integration:** Achieve quick integration and customization through expandable UI components.

Guides for Content Editor

- **Initialize Editing Mode**

  Set the content editing mode and choose the target type for editing.

- **Create, Move, and Delete Text and Images**

  Create, move, and delete text and images.

- **Edit Text and Images Properties**

  Edit the existing text and image properties on the document.

- **Undo and Redo**

  Undo any modified content or restore content that has been undone.

- **End Content Editing and Save**

  Exit editing mode and save the document after completing the edits.

- **Find and Replace in Content Editing Mode**

  Perform text search and replacement in content editing mode..

3.10.2 Initialize Editing Mode

Before performing PDF content editing, you should initialize the content editing mode.

This example shows how to initialize editing mode:

```
var document = CPDFDocument(url: URL(string: "filePath"))
let page = document?.page(at: 0)

let pdfView = CPDFView(frame: self.view.bounds)

// Set the document to be displayed.
pdfView?.document = document
pdfView?.beginEditingLoadType([.text, .image, .path])
```

```
NSURL *url = [NSURL fileURLWithPath:pdfPath];
CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

CPDFView *pdfView = [[CPDFView alloc] initWithFrame:self.view.bounds];

// Set the document to be displayed.
pdfView.document = document;
[pdfView beginEditingLoadType:CEditingLoadTypeText | CEditingLoadTypeImage |
CEditingLoadTypePath];
```

Explanation of Editing Mode

Here is the explanation of the editing mode settings:

Editing Mode	Description	Parameters
Text Mode	In text mode, the text blocks surrounded by dotted lines will be displayed in the PDF document. Users can select text blocks and add, delete, copy, and paste text.	CEditingLoadTypeText
Image Mode	In image mode, the images surrounded by dotted lines will be displayed in the PDF document. Users can select images and then delete, crop, rotate, mirror, replace, save images, or set image properties.	CEditingLoadTypeImage
Path Mode	In path mode, the path in the PDF document can be selected, which can be cut, copied, moved, deleted, and so on.	CEditingLoadTypePath
Text & Image &Path Mode	In text , image and path mode, the text blocks and images surrounded by dotted lines will be displayed in the PDF document. Users can select and edit both text and images.	CEditingLoadTypeText|CEditingLoadTypeImage | CEditingLoadTypePath

3.10.3 Create, Move, and Delete Text , Images and Path

ComPDFKit provides comprehensive methods for creating, moving, and deleting text , images and path.

Performing Actions Through `CPDFView`.

`CPDFView` provides basic interactive capabilities by default, allowing the user to create and delete text , images and path, drag and drop to move the position of images and text blocks, resize images , text and path blocks, etc. by using the mouse and keyboard.

Configure the Context Menu.

If you need to copy, paste, cut, or delete text , images or path, you can add these methods to the context menu through the `menuItemsEditingAtPoint` event of `CPDFView`.

This example shows how to configure the context menu:

```
override func menuItemsEditing(at point: CGPoint, forPage page: CPDFPage) -> [UIMenuItem]
{
      var menuItems = super.menuItemsEditing(at: point, forPage: page)

      self.menuPoint = point
      self.menuPage = page

      if editStatus == CEditingSelectStateEmpty {
          let addTextItem = UIMenuItem(title: NSLocalizedString("Add Text", comment: ""),
                                       action: #selector(addTextEditingItemAction(_:)))
          let addImageItem = UIMenuItem(title: NSLocalizedString("Add Image", comment:
""),
                                        action: #selector(addImageEditingItemAction(_:)))

          menuItems.append(addImageItem)
          menuItems.append(addTextItem)
      }
```

```
        return menuItems
    }
```

```
- (NSArray<UIMenuItem *> *)menuItemsEditingAtPoint:(CGPoint)point forPage:(CPDFPage
*)page {
    NSArray * items = [super menuItemsEditingAtPoint:point forPage:page];
    NSMutableArray *menuItems = [NSMutableArray array];
    if (items)
        [menuItems addObjectsFromArray:items];

    self.menuPoint = point;
    self.menuPage = page;

    if(CEditingSelectStateEmpty == self.editStatus) {
        UIMenuItem *addTextItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Add Text", nil)

 action:@selector(addTextEditingItemAction:)];
        UIMenuItem *addImageItem = [[UIMenuItem alloc]
initWithTitle:NSLocalizedString(@"Add Image", nil)

action:@selector(addImageEditingItemAction:)];
        [menuItems addObject:addImageItem];
        [menuItems addObject:addTextItem];

    }

    return menuItems;
}
```

Inserting Text and Images

You can specify the ability to insert text and image blocks through the `SetPDFEditCreateType` method of `CPDFView`. The following code shows how to achieve this:

```
// Insert Image
pdfView.changeEditingLoadType(.image)
pdfView.setShouAddEdit(.image)
// Insert Text
pdfView.changeEditingLoadType(.text)
pdfView.setShouAddEdit(.text)
// Cancel
pdfView.changeEditingLoadType([.image, .text])
pdfView.setShouAddEdit([])
```

```
// Insert Image
[pdfView changeEditingLoadType:CEditingLoadTypeImage];
[pdfView setShouAddEditAreaType:CAddEditingAreaTypeImage];
// Insert Text
 [pdfView changeEditingLoadType:CEditingLoadTypeText];
[pdfView setShouAddEditAreaType:CAddEditingAreaTypeText];
// Cancel
[pdfView changeEditingLoadType:CEditingLoadTypeText | CEditingLoadTypeImage];
[pdfView setShouAddEditAreaType:CAddEditingAreaTypeNone];
```

3.10.4 Edit Text and Image Properties

ComPDFKit offers various methods to modify text and image attributes.

Edit Text Properties

ComPDFKit supports modifying text properties, such as text font size, name, color, alignment, italics, bold, transparency, etc.And

copying, and creating or removing underlines and strikethroughs for text.

This example shows how to set text to 12pt, red, and bold:

```
pdfView.setEditingSelectionFontSize(12.0)
pdfView.setEditingSelectionFontColor(.red)
pdfView.setCurrentSelectionIsBold(true)
```

```
[pdfView setEditingSelectionFontSize:12.0];
[pdfView setEditingSelectionFontColor:[CPDFKitPlatformColor redColor]];
[pdfView setCurrentSelectionIsBold:YES];
```

Below is the example code for creating and removing underlines and strikethroughs for text:

```
//  creating underlines and strikethroughs for text
pdfView?.setCurrentSelectionShowUnderline(true, with: editingArea as? CPDFEditTextArea)
pdfView?.setCurrentSelectionShowStrikeout(true, with: editingArea as? CPDFEditTextArea)

//  removing underlines and strikethroughs for text
pdfView?.setCurrentSelectionShowUnderline(false, with: editingArea as? CPDFEditTextArea)
pdfView?.setCurrentSelectionShowStrikeout(false, with: editingArea as? CPDFEditTextArea)
```

```
//  creating underlines and strikethroughs for text
[pdfView setCurrentSelectionShowUnderline:YES withTextArea:(CPDFEditTextArea
*)editingArea];
[pdfView setCurrentSelectionShowStrikeout:YES withTextArea:(CPDFEditTextArea
*)editingArea];

//  removing underlines and strikethroughs for text
[pdfView setCurrentSelectionShowUnderline:NO withTextArea:(CPDFEditTextArea
*)editingArea];
[pdfView setCurrentSelectionShowStrikeout:NO withTextArea:(CPDFEditTextArea
*)editingArea];
```

Edit Image Properties

ComPDFKit supports modifying image properties, such as rotating, cropping, mirroring, and setting transparency.

This example shows how to rotate an image and set it to semi-transparent:

```
pdfView.rotateEdit(pdfView.editingArea() as! CPDFEditImageArea, rotateAngle: 90)
/**
 * Set the image transparency.
 */
pdfView.setImageTransparencyEdit(pdfView.editingArea() as! CPDFEditImageArea,
transparency: 0.5)
```

```
[pdfView rotateEditArea:pdfView.editingArea rotateAngle:90];
/**
 * Set the image transparency.
 */
[pdfView setImageTransparencyEditArea:pdfView.editingArea transparency:0.5];
```

3.10.5 Undo and Redo

Undo refers to reversing previous actions and restoring the document to its previous state. After a user operates, if they find the result unsatisfactory, they can undo the step to revert to the previous state. This helps prevent data loss due to mistakes or dissatisfaction with the outcome.

Redo is a complementary feature to undo; it allows users to reapply previously undone actions, restoring the system state to its state before the undo. Redo is typically used when a user realizes that the initially undone actions were necessary after one or multiple undo steps.

This example shows how to undo and redo：

```
// undo.
if pdfView.canEditTextUndo() {
    pdfView.editTextUndo()
}
// redo.
if pdfView.canEditTextRedo() {
    pdfView.editTextRedo()
}
```

```
// undo.
if ([pdfView canEditTextUndo]) {
    [pdfView editTextUndo];
}
// redo.
if ([pdfView canEditTextRedo]) {
    [pdfView editTextRedo];
}
```

3.10.6 End Content Editing and Save

After completing edits, you can exit editing mode by using `endOfEditing` with `CPDFView` to switch out of edit mode. To save the changes, utilize `commitEditing` to save the modified document.

```
if pdfView.isEdited() == true {
    DispatchQueue.global().async {
        pdfView.commitEditing()
        DispatchQueue.main.async {
            pdfView.endOfEditing()
        }
    }
}
```

```
if (pdfView.isEdited) {
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
            [pdfView commitEditing];
            dispatch_async(dispatch_get_main_queue(), ^{
                [pdfView endOfEditing];
            });
        });
  }
```

3.10.7 Find and Replace in Content Editing Mode

In content editing mode, when users perform text searches and replacements in a PDF document, the search results are typically highlighted, providing links for navigation to the corresponding positions. Users can also replace individual keywords or all occurrences of a keyword with the provided text. By leveraging the search and replace features provided by the ComPDFKit SDK, you can easily implement these functionalities.

Text Search

In content editing mode, text search allows users to input keywords to search for matching text throughout the entire PDF document. It also enables text replacement for individual or all occurrences of keywords.

```
let results = document?.startFindEditText(from: page, with: "searchText",
options:.caseSensitive)
```

```
NSArray *results = [documet startFindEditTextFromPage:page withString:@"searchText"
options:CPDFSearchCaseSensitive];
```

Single Replacement

Replace the text for a single keyword.

```
let currentSelection: CPDFSelection
document?.replace(with: currentSelection, search: "searchText", toReplace: "replaceText")
```

```
CPDFSelection *currentSelection;
  [documet replaceWithSelection:currentSelection searchString:@"searchText"
 toReplaceString:@"replaceText" completionHandler:nil];
```

Replace All

Replace the text for all occurrences of a keyword.

```
document?.replaceAllEditText(with:"searchText", toReplace: "replaceText")
```

```
[documet replaceAllEditTextWithString:@"searchText" toReplaceString:@"replaceText"];
```

3.11 Digital Signatures

3.11.1 Overview

A digital signature is legally binding and can be equivalent to an ink pen signature on paper contracts and other documents.

Unlike electronic signatures, digital signatures have a unique digital ID that identifies the signer's identity. Digital signatures can get information about whether the signature is trustworthy and whether the document has been modified after the signature, thereby ensuring the legal validity of the document.

Benefits of ComPDFKit PDF Digital Signatures

- **Authentication:** Digital signatures can accurately identify the creator and the signer of a document.
- **Integrity:** Digital signatures allow users to easily verify whether the document's content has been altered after signing.

- **Non-Repudiation:** When the signature is valid, it can prove the signer's intent to sign, they can't deny that they have signed the document.
- **Built-in Certificate Support:** Full support for PFX and P12 certificates.
- **Custom Appearance:** Customize the appearance of signatures through drawn, image, or typed signatures.
- **Fast UI Integration:** Achieve rapid integration and customization through extensible UI components.

Guides for Digital Signatures

- **Create Digital Certificates**

  Create certificates in PFX or P12 formats, which can be used for digital signatures.

- **Create Digital Signatures**

  This function allows users to generate a digital signature using a digital certificate with a personal private key, and attach it to a specific document to ensure data integrity and origin verification.

- **Read Digital Signature Information**

  Extracting signature information refers to extracting the information of a digital signature so that other users can view or archive this information.

- **Verify Digital Certificates**

  Certificate information verification allows users to confirm the validity and authenticity of the digital certificates.

- **Verify Digital Signatures**

  By verifying signature information, users can determine whether specific data or documents have been authorized and remain unaltered.

- **Trusting Certificates**

  Trusting a certificate refers to the act of considering a specific certificate or a certificate authority as trustworthy. This is a key part of the digital signature system as it ensures the trustworthiness of the signature and the certificate, thereby building a secure digital communication and interaction environment.

- **Remove Digital Signatures**

  Deleting a digital signature refers to the action of revoking or invalidating a digital signature. This may occur due to the loss or compromise of the signer's private key or when the signature is deemed no longer valid. Removing a signature is part of digital security management to ensure the integrity and security of data.

3.11.2 Concepts of Digital Signatures

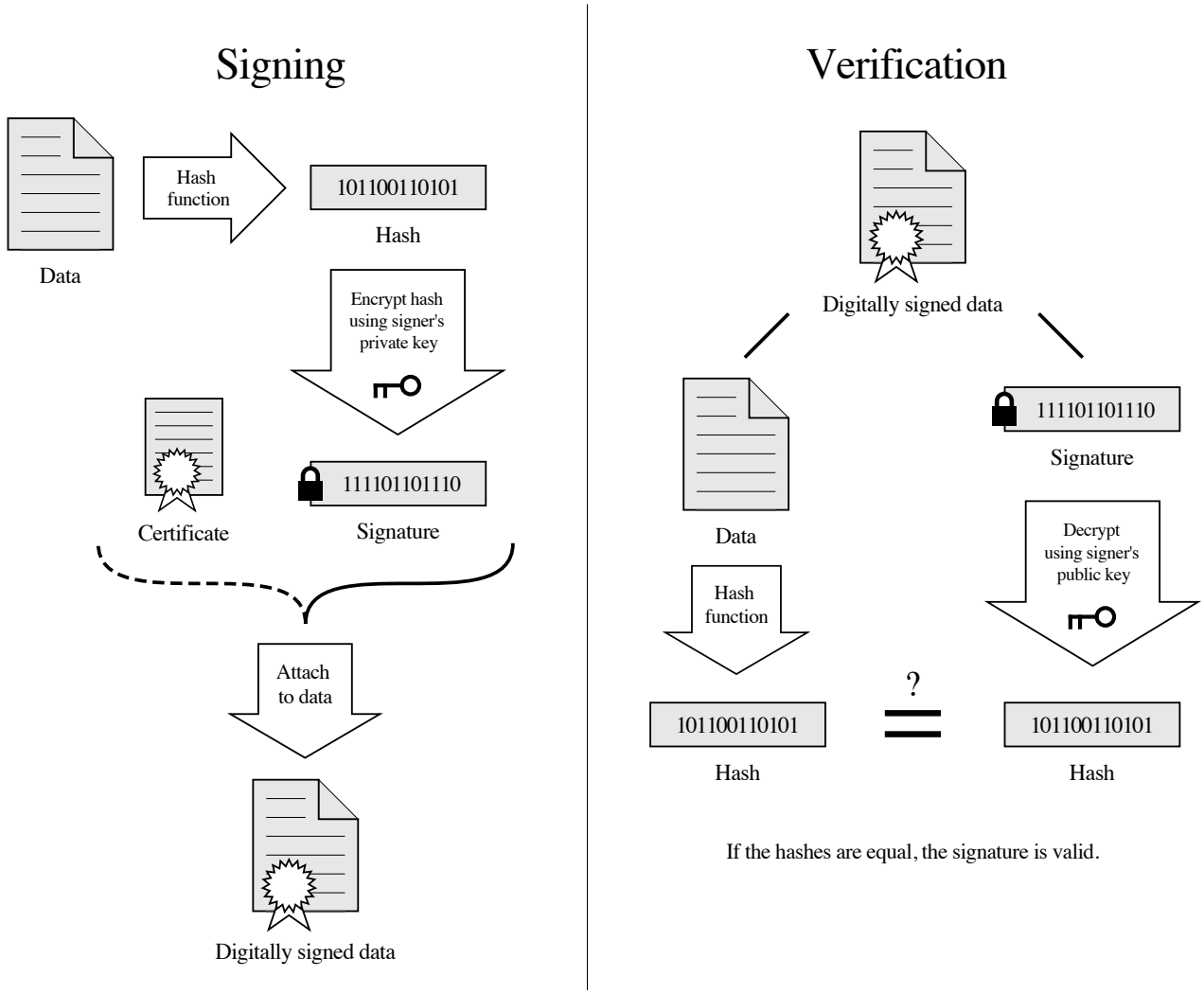
How Digital Signatures Work

Principle of Signature:

A hash value of the data to be encrypted is obtained through a hash function (a unique fingerprint of the data. Any tampering with the data content will result in a different hash). The hash value is encrypted using the signer's private key to obtain the digital signature. The signed data will be generated after attaching the digital signature to the data.

Verification Principle:

Separate the signature from the data, and obtain the hash value of the data through the same hash function used by the signer. Decrypt the hash value using the signer's public key to get the signer's hash value. By comparing the two values, we can confirm whether the file has been tampered with.



Digital Signatures vs Electronic Signatures

An electronic signature is essentially an annotation within a document. Apart from the customizable appearance of the signature, it lacks identifiable information about the creator and cannot verify whether the document has been altered.

However, a digital signature uses complex encryption algorithms to create a unique identifier that is linked to both the document's content and the creator's information. Any modification to the document's content results in a failed digital signature verification, ensuring the uniqueness and legitimacy of the signer's identity.

What Is a Digital Certificate

A digital certificate is a digital authentication that marks the identity information of the parties in Internet communication. It can be used online to identify the identity of the other party, hence, it is also known as a digital ID. The format of the digital certificate typically adopts the X.509 international standard and will generally include the certificate's public key, user information, the validity period of the public key, the name of the certificate authority, the serial number of the digital certificate, and the digital signature of the issuing organization.

Digital certificates provide the transmission of information and data in an encrypted or decrypted form during communication between network users, ensuring the integrity and security of information and data.

Support PKCS12 Certificate

PKCS12 (Or PKCS #12) is one of the family of standards called Public-Key Cryptography Standards (PKCS) published by RSA Laboratories. ComPDFKit supports signing PDFs with PKCS12 files which are with ".p12" or ".pfx" file extensions.

What Is Certificate Chain

A Certificate Chain (Chain of Trust), is an ordered collection of digital certificates used to verify the authenticity and trustworthiness of a digital certificate. Certificate chains are typically employed to establish trust, ensuring that both the public key and the identity of entities are legitimate and trustworthy.

Here are some key concepts within a certificate chain:

- **Root Certificate**

  The starting point of a certificate chain is the Root Certificate. Root certificates are top-level certificates issued by trusted Certificate Authorities (CAs) and are often built into operating systems or applications. These root certificates serve as the foundation of trust because they are considered inherently trustworthy.

- **Intermediate Certificates**

  Intermediate certificates, also known as issuer certificates or sub-certificates, are issued by root certificate authorities and are used to issue certificates for end entities. Intermediate certificates form an intermediate link within the certificate chain.

- **End Entity Certificate**

  An end entity certificate is the certificate of the subject of a digital signature (typically an individual, server, or device). These certificates are issued by intermediate certificate authorities and contain the public key and relevant identity information.

- **Trust Establishment**

  Trust is established through the certificate chain, passing trust from the root certificate to the end entity certificate. If the root certificate is trusted, then the end entity certificate is also trusted, as the trust chain between them is continuous.

Certificate Authority (CA)

A digital certificate issuing authority is an authoritative body responsible for issuing and managing digital certificates, and as a trusted third party in e-commerce transactions, it bears the responsibility for verifying the legality of public keys in the public key system.

The CA center issues a digital certificate to each user who uses a public key, the function of the digital certificate is to prove that the user listed in the certificate legally owns the public key listed in the certificate. The CA is responsible for issuing, certifying, and managing issued certificates. It needs to formulate policies and specific steps to verify and identify user identities and sign user certificates to ensure the identity of the certificate holder and the ownership of the public key.

Whether a Digital Signature Needs a CA

It is not necessary. When there is not a third-party notary, a CA is not needed, and we can use a self-signed certificate. With ComPDFKit, you can manually set to trust self-signed certificates, which is very useful for trusted parties to sign and check files. However, since there is no digital certificate issuing authority for certification, self-signed digital identity cards cannot guarantee the validity of identity information, and they may not be accepted in some use cases.

How to Confirm the Identity of the Digital Certificate Creator

Subject contains identity information about the certificate holder, commonly including fields such as C (Country), ST (Province), L (Locality), O (Organization), OU (Organizational Unit), CN (Common Name), and others. These details help identify who the certificate holder is.
DN (Distinguished Name) represents the complete and hierarchical representation of the "Subject" field. It includes all the information from the "Subject" field and organizes it in a structured manner.

The X.509 standard specifies a specific string format for describing DN, for example:

```
CN=Alan, OU=RD Department, O=ComPDFKit, C=SG, Email=xxxxxx@example.com
```

3.11.3 Create Digital Certificates

PKCS12 (Public Key Cryptography Standard #12) format digital certificates usually contain a public key, a private key, and other information related to the certificate. PKCS12 is a standard format used to store security certificates, private keys, and other related information. This format is commonly used to export, backup, and share digital certificates and private keys which are used in secure communications and identity verification.

When creating a PKCS12 standard certificate, in addition to the data confirming your identity, a password is typically required to protect your certificate. Only those who possess the password can access the private key contained within and perform actions such as signing documents through the certificate.

This example shows how to create digital certificates:

```
// Generate certificate.
//
// Password: ComPDFKit
//
// info: /C=SG/O=ComPDFKit/D=R&D Department/CN=Alan/emailAddress=xxxx@example.com
//
// C=SG: This represents the country code "SG," which typically stands for Singapore.
// O=ComPDFKit: This is the Organization (O) field, indicating the name of the
organization or entity, in this case, "ComPDFKit."
```

```
// D=R&D Department: This is the Department (D) field, indicating the specific department within the organization, in this case, "R&D Department."
// CN=Alan: This is the Common Name (CN) field, which usually represents the name of the individual or entity. In this case, it is "Alan."
// emailAddress=xxxx@example.com: Email is xxxx@example.com
//
// CPDFCertUsage.CPDFCertUsageAll: Used for both digital signing and data validation simultaneously
//
// is_2048 = true: Enhanced security encryption
//

 var certificateInfo = [String: Any]()
 certificateInfo["CN"] = "Alan"
 certificateInfo["emailAddress"] = emailTextField?.inputTextField?.text
 certificateInfo["C"] = coutryCode

 let path = NSHomeDirectory() + "/Documents"
 let writeDirectoryPath = "\(path)/Signature"
 let tempFilePath = "\(writeDirectoryPath)/\(URL(fileURLWithPath: filePath ?? "").lastPathComponent)"

 CPDFSignature.generatePKCS12Cert(withInfo: certificateInfo, password: "123", toPath: tempFilePath, certUsage: .digSig)
```

```
// Generate certificate.
//
// Password: ComPDFKit
//
// info: /C=SG/O=ComPDFKit/D=R&D Department/CN=Alan/emailAddress=xxxx@example.com
//
// C=SG: This represents the country code "SG," which typically stands for Singapore.
// O=ComPDFKit: This is the Organization (O) field, indicating the name of the organization or entity, in this case, "ComPDFKit."
// D=R&D Department: This is the Department (D) field, indicating the specific department within the organization, in this case, "R&D Department."
// CN=Alan: This is the Common Name (CN) field, which usually represents the name of the individual or entity. In this case, it is "Alan."
// emailAddress=xxxx@example.com: Email is xxxx@example.com
//
// CPDFCertUsage.CPDFCertUsageAll: Used for both digital signing and data validation simultaneously
//
// is_2048 = true: Enhanced security encryption
//

NSMutableDictionary * cer = [NSMutableDictionary dictionary];
[cer setValue:@"Alan" forKey:@"CN"];
[cer setValue:@"xxxx@example.com" forKey:@"emailAddress"];
[cer setValue:@"CN" forKey:@"C"];
```

```
NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"Documents"];
  NSString *writeDirectoryPath = [NSString stringWithFormat:@"%@/%@", path,
@"Signature"];

  if (![[NSFileManager defaultManager] fileExistsAtPath:writeDirectoryPath])
      [[NSFileManager defaultManager] createDirectoryAtURL:[NSURL
fileURLWithPath:writeDirectoryPath] withIntermediateDirectories:YES attributes:nil
error:nil];

  NSString *currentDateString = [self tagString];

  NSString *writeFilePath = [NSString stringWithFormat:@"%@/%@-
%@.pfx",writeDirectoryPath,@"CSignature",currentDateString];


BOOL save = [CPDFSignature generatePKCS12CertWithInfo:cer password:@"123"
writeFilePathtoPath:writeFilePath certUsage:CPDFCertUsageDigSig];
```

3.11.4 Create Digital Signatures

Creating a digital signature involves two steps:

1. Create a Signature Field
2. Sign within the Signature Field

By following these two steps, you can either self-sign a document or invite others to sign within the signature field you've created.

Create a Signature Field

ComPDFKit offers support for customizing the styles of the signature form field and allows you to customize the appearance of your signature using drawn, image, and typed signatures.

This example shows how to create a signature field：

```
// Create a signature field.
//
// Page Index: 0
// Rect: CRect(28, 420, 150, 370)
// Border RGB:{ 0, 0, 0 }
// Widget Background RGB: { 150, 180, 210 }
//

if let url = Bundle.main.url(forResource: "filename", withExtension: "pdf") {
    let document = CPDFDocument(url: url)
    if let page = document?.page(at: 0) {
        let widgetAnnotation = CPDFSignatureWidgetAnnotation(document: document)

        widgetAnnotation.fieldName = "Signature_\(self.tagString())"
        widgetAnnotation.borderWidth = 2.0
```

```
        widgetAnnotation.bounds = CGRect(x: 28, y: 420, width: 150, height: 370)
        widgetAnnotation.modificationDate = Date()

        page.addAnnotation(widgetAnnotation)
    }
}
```

```
// Create a signature field.
//
// Page Index: 0
// Rect: CRect(28, 420, 150, 370)
// Border RGB:{ 0, 0, 0 }
// Widget Background RGB: { 150, 180, 210 }
//

NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
CPDFPage *page = [document pageAtIndex:0];

CPDFSignatureWidgetAnnotation *widgetAnnotation = [[CPDFSignatureWidgetAnnotation alloc]
initWithDocument:self.document];

[widgetAnnotation setFieldName:[NSString stringWithFormat:@"%@%@",@"Signature_",[self
tagString]]];

widgetAnnotation.borderWidth = 2.0;
widgetAnnotation.bounds = CGRectMake(28, 420, 150, 370);
[widgetAnnotation setModificationDate:[NSDate date]];
[page addAnnotation:widgetAnnotation];
```

Sign Within the Signature Field

To sign within the signature field, you need to do three things:

- Possess a certificate that conforms to the PKCS12 standard (in PFX or P12 format) and ensure that you know its password. You can create a compliant digital certificate using the built-in methods within the ComPDFKit SDK.
- Set the appearance of the digital signature.
- Write the data into the signature field.

This example shows how to sign within the signature field：

```
//     Sign in the signature field.
//
//     Text: Grantor Name
//     Content:
//     Name: Get grantor name from certificate
//     Date: Now (yyyy.mm.dd)
//     Reason: I am the owner of the document
```

```
//     DN: Subject
//     IsContentAlginLeft: False
//     IsDrawLogo: True
//     LogoBitmap: logo.png
//     text color RGB: { 0, 0, 0 }
//     Output file name: document.FileName + "_Signed.pdf"
//

let writeDirectoryPath = NSHomeDirectory().appending("/Documents/Signatures")
let documentURL = self.pdfListView?.document.documentURL
let documentName = documentURL?.lastPathComponent.components(separatedBy: ".").first ??
""
let writeFilePath = writeDirectoryPath + "/" + "\(documentName)_Signed_\
(tagString()).pdf"

if FileManager.default.fileExists(atPath: writeFilePath) {
    try? FileManager.default.removeItem(at: URL(fileURLWithPath: writeFilePath))
}

let contents = CPDFSignatureConfig()
signatureConfig.text = nil
signatureConfig.isDrawOnlyContent = true
signatureConfig.image = image

signatureConfig.isContentAlginLeft = false
signatureConfig.isDrawLogo = true
signatureConfig.isDrawKey = true
signatureConfig.logo = UIImage(named: "ImageNameDigitalSignature", in: Bundle(for:
self.classForCoder), compatibleWith: nil)

var contents = signatureConfig.contents ?? []

let configItem1 = CPDFSignatureConfigItem()
configItem1.key = "Digitally signed by Apple Distribution"
configItem1.value = NSLocalizedString(self.signatureCertificate?.issuerDict["CN"] as?
String ?? "", comment: "")
contents.append(configItem1)

let configItem2 = CPDFSignatureConfigItem()
configItem2.key = "Date"
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
configItem2.value = dateFormatter.string(from: Date())
contents.append(configItem2)

let configItem3 = CPDFSignatureConfigItem()
configItem3?.key = "DN"
let dn = signatureCertificate?.issuerDict["C"]
configItem3?.value = NSLocalizedString(dn, comment: "")
contents3.append(configItem)

let configItem4 = CPDFSignatureConfigItem()
```

```
configItem4?.key = VERSION_KEY
configItem4?.value = infoDictionary["CFBundleShortVersionString"] as? String
contents.append(configItem4)

let configItem5 = CPDFSignatureConfigItem()
configItem5?.key = "Location"
configItem5?.value = NSLocalizedString("<your signing location here>", comment: "")
contents.append(configItem5)

let configItem6 = CPDFSignatureConfigItem()
configItem6?.key = "Reason"
configItem6?.value = NSLocalizedString("<your signing reason here>", comment: "")
contents.append(configItem6)

annotation?.signAppearanceConfig(signatureConfig)

let isSuccess = self.pdfListView?.document.writeSignature(to: URL(fileURLWithPath:
writeFilePath), withWidget: widget, pkcs12Cert: path, password: password, location:
locationStr, reason: reasonStr, permissions: .none) ?? false
```

```
//     Sign in the signature field.
//
//     Text: Grantor Name
//     Content:
//     Name: Get grantor name from certificate
//     Date: Now (yyyy.mm.dd)
//     Reason: I am the owner of the document
//     DN: Subject
//     IsContentAlginLeft: False
//     IsDrawLogo: True
//     LogoBitmap: logo.png
//     text color RGB: { 0, 0, 0 }
//     Output file name: document.FileName + "_Signed.pdf"
//

NSString *homePath = [NSHomeDirectory() stringByAppendingPathComponent:@"Documents"];
NSString *writeDirectoryPath = [NSString stringWithFormat:@"%@/%@", homePath,
@"Signature"];

if (![[NSFileManager defaultManager] fileExistsAtPath:writeDirectoryPath])
[[NSFileManager defaultManager] createDirectoryAtURL:[NSURL
fileURLWithPath:writeDirectoryPath] withIntermediateDirectories:YES attributes:nil
error:nil];

NSString *writeFilePath = [NSString
stringWithFormat:@"%@/%@_Signed_%@.pdf",writeDirectoryPath,self.pdfListView.document.docu
mentURL.lastPathComponent.stringByDeletingPathExtension,[self tagString]];
if([[NSFileManager defaultManager] fileExistsAtPath:writeFilePath]) {
[[NSFileManager defaultManager] removeItemAtPath:writeFilePath error:nil];
}
```

```objc
CPDFSignatureConfig *signatureConfig = [[CPDFSignatureConfig alloc] init];
signatureConfig.image = image;
signatureConfig.isContentAlginLeft = NO;
signatureConfig.isDrawLogo = YES;
signatureConfig.isDrawKey = YES;
signatureConfig.logo = [UIImage imageNamed:@"ImageNameDigitalSignature"];

NSMutableArray *contents = [NSMutableArray arrayWithArray:signatureConfig.contents];

CPDFSignatureConfigItem *configItem1 = [[CPDFSignatureConfigItem alloc]init];
configItem1.key = @"Digitally signed by Apple Distribution";
configItem1.value = NSLocalizedString([signatureCertificate.issuerDict
objectForKey:@"CN"], nil);
[contents addObject:configItem1];

CPDFSignatureConfigItem *configItem2 = [[CPDFSignatureConfigItem alloc]init];
configItem2.key = @"Date";
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
configItem2.value = [dateFormatter stringFromDate:[NSDate date]];
[contents addObject:configItem2];

CPDFSignatureConfigItem *configItem3 = [[CPDFSignatureConfigItem alloc]init];
configItem3.key = @"DN";
configItem3.value = NSLocalizedString([signatureCertificate.subjectDict
objectForKey:@"C"], nil);
[contents addObject:configItem3];

CPDFSignatureConfigItem *configItem4 = [[CPDFSignatureConfigItem alloc]init];
configItem4.key = @"ComPDFKit Version";
NSDictionary*infoDictionary = [[NSBundle mainBundle] infoDictionary];
NSString *app_Version = [infoDictionary objectForKey:@"CFBundleShortVersionString"];
configItem4.value = app_Version;
[configItem4 addObject:configItem4];

CPDFSignatureConfigItem *configItem5 = [[CPDFSignatureConfigItem alloc]init];
configItem5.key = @"Reason";
configItem5.value = NSLocalizedString(@"I am the owner of the document.", nil);
[contents addObject:configItem5];

CPDFSignatureConfigItem *configItem6 = [[CPDFSignatureConfigItem alloc]init];
configItem6.key = @"Location";
configItem6.value = NSLocalizedString(@"<your signing location here>", nil);
[contents addObject:configItem6];

signatureConfig.contents = contents;
[widgetAnnotation signWithSignatureConfig:signatureConfig];

BOOL isSuccess = [document writeSignatureToURL:[NSURL fileURLWithPath:writeFilePath]
withWidget:widgetAnnotation PKCS12Cert:@"Certificate path" password:@"123" location:@"
<your signing location here>" reason:@"I am the owner of the document."
permissions:CPDFSignaturePermissionsNone];
```

##

3.11.5 Read Digital Signature Information

You can read various pieces of information from a document's digital signature, including the signature itself, the signer of the signature, and certain details of the signer's digital certificate.

For a comprehensive list of retrievable information, please refer to the API Reference.

This example shows how to read digital signature information：

```swift
if let url = Bundle.main.url(forResource: "filename", withExtension: "pdf") {
    let document = CPDFDocument(url: url)

    var commandLineStr = ""

    if let signatures = document?.signatures, signatures.count > 0 {
        for signature in signatures {
            commandLineStr += "Name: \(signature.name)\n"
            commandLineStr += "Location: \(signature.location)\n"
            commandLineStr += "Reason: \(signature.reason)\n"
            commandLineStr += "Date: \(signature.date)\n"
            commandLineStr += "Subject: \(signature.subFilter)\n"
        }
    }
}
```

```objc
NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSString *commandLineStr = @""

NSArray *signatures = [document signatures];
if (signatures.count > 0) {
  for (CPDFSignature *signature in signatures) {
      commandLineStr = [commandLineStr stringByAppendingFormat:@"Name: %@\n",
signature.name];
      commandLineStr = [commandLineStr stringByAppendingFormat:@"Location: %@\n",
signature.location];
      commandLineStr = [commandLineStr stringByAppendingFormat:@"Reason: %@\n",
signature.reason];
      commandLineStr = [commandLineStr stringByAppendingFormat:@"Date: %@\n",
signature.date];
      commandLineStr = [commandLineStr stringByAppendingFormat:@"Subject: %@\n",
signature.subFilter];
  }
}
```

The Connection Between Digital Signatures, Signers, and Digital Certificates

A digital signature is generated by encrypting a document using the private key of the signer and then verifying the validity of the signature using the public key from the signer's certificate. The signature, signer, and digital certificate constitute a crucial part of digital signatures in a PDF document.

In most cases, one signature corresponds to one signer. However, in some situations, a digital signature can include multiple signers, each with their own certificate chain. This multi-signer mechanism can be very useful in certain application scenarios because it allows multiple entities to digitally sign the same document, each using their certificate and private key.

3.11.6 Verify Digital Certificates

When verifying digital certificates, the system automatically checks the trustworthiness of all certificates in the certificate chain and also verifies whether the certificates have expired. Only certificates that are both not expired and considered trustworthy in the entire certificate chain are considered trusted digital certificates.

Key Code

```
// Verify digital certificates
//
// To verify the trustworthiness of a certificate, you need to verify that all
certificates in the certificate chain are trustworthy.
//
// In ComPDFKit, this progress is automatic.
//
// You should first call "signature.verifySignature(with: document)"。
// Then you can check the properties. "CPDFSignatureCertificate.IsTrusted"
//

if let url = URL(string: pdfPath) {
    let document = CPDFDocument(url: url)

    if let signatures = document?.signatures {
        for signature in signatures {
            signature.verifySignature(with: document)
            if let signer = signature.signers.first, let certificate =
signer.certificates.first {
                var isCertTrusted = true

                if !signer.isCertTrusted {
                    isCertTrusted = false
                }

            }
        }
    }
}
```

```
// Verify digital certificates
//
```

```
// To verify the trustworthiness of a certificate, you need to verify that all
certificates in the certificate chain are trustworthy.
//
// In ComPDFKit, this progress is automatic.
//
// You should first call "signature.verifySignature(with: document)"。
// Then you can check the properties "isCertTrusted"
//

NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSArray *signatures = [document signatures];

for (CPDFSignature *signature in signatures) {
    [sign verifySignatureWithDocument:document];
    CPDFSigner *signer = signature.signers.firstObject;

    BOOL isCertTrusted = YES;

   if (!signer.isCertTrusted) {
        isCertTrusted = NO;
    }

}
```

3.11.7 Verify Digital Signatures

When verifying digital certificates, the system automatically checks the trustworthiness of all certificates in the certificate chain and also verifies whether the certificates have expired. Only certificates that are both not expired and considered trustworthy in the entire certificate chain are considered trusted digital certificates.

This example shows how to verify digital certificates：

```
if let url = URL(string: pdfPath) {
    let document = CPDFDocument(url: url)

    if let signatures = document?.signatures {
        // Iterate through all digital signatures.
for
        for signature in signatures {
            signature.verifySignature(with: document)
            if let signer = signature.signers.first, let certificate =
signer.certificates.first {
                var isSignVerified = true
                var isCertTrusted = true
```

```
                if !signer.isCertTrusted {
                    isCertTrusted = false
                }

                if !signer.isSignVerified {
                    isSignVerified = false
                }

                if isSignVerified && isCertTrusted {
                    //Signature is valid and the certificate is trusted.
                    //Perform the corresponding actions.
                } else if isSignVerified && !isCertTrusted {
                    //Signature is valid but the certificate is not trusted.
                    //Perform the corresponding actions.
                } else if !isSignVerified && !isCertTrusted {
                    // signature is invalid
                    // Perform the corresponding actions.
                } else {
                    // signature is invalid
                    // Perform the corresponding actions.
                }
            }
        }
    }
}
```

```
NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSArray *signatures = [document signatures];

// Iterate through all digital signatures.
for (CPDFSignature *signature in signatures) {
    [sign verifySignatureWithDocument:document];
    CPDFSigner *signer = signature.signers.firstObject;
    CPDFSignatureCertificate * cer = signer.certificates.firstObject;

    BOOL isSignVerified = YES;
    BOOL isCertTrusted = YES;

   if (!signer.isCertTrusted) {
        isCertTrusted = NO;
    }

    if (!signer.isSignVerified) {
        isSignVerified = NO;
    }

    if (isSignVerified && isCertTrusted) {
```

```
        // Signature is valid and the certificate is trusted.
      // Perform the corresponding actions.

    } else if(isSignVerified && !isCertTrusted) {
        // Signature is valid but the certificate is not trusted.
      // Perform the corresponding actions.
    } else if(!isSignVerified && !isCertTrusted){

    } else {
       // Signature is invalid.
      // Perform the corresponding actions.
    }
}
```

3.11.8 Trust Certificate

Verifying a digital signature consists of signature validity and certificate trustworthiness.

- Signature validity indicates that the document has not been tampered with.
- Certificate trustworthiness confirms that the signer is trustworthy.

Generally, a signature is verified only when both the signature is valid and the certificate is trustworthy.

This example shows how to verify digital signatures：

```
if let url = Bundle.main.url(forResource: "filename", withExtension: "pdf") {
    let document = CPDFDocument(url: url)

    if let signatures = document?.signatures, let signature = signatures.first,
       let signer = signature.signers.first, let certificate = signer.certificates.first
{

        certificate.checkCertificateIsTrusted()
        let success = certificate.addToTrustedCertificates()
        certificate.checkCertificateIsTrusted()
    }
}
```

```
NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];

NSArray *signatures = [document signatures];

CPDFSignature *signature = signatures[0];
CPDFSigner *signer = signature.signers.firstObject;
CPDFSignatureCertificate * certificate = signer.certificates.firstObject;

[certificate checkCertificateIsTrusted];
BOOL success = [certificate addToTrustedCertificates];
[certificate checkCertificateIsTrusted];
```

3.11.9 Remove Digital Signatures

You can easily remove a digital signature, and when you do so, both the appearance and data associated with the signature will be deleted.

It's important to note that removing a signature does not remove the signature field.

This example shows how to remove digital signatures：

```
// Get the digital signatures object.
if let url = Bundle.main.url(forResource: "filename", withExtension: "pdf") {
    let document = CPDFDocument(url: url)
    if let signatures = document?.signatures, let signature = signatures.first {

        // Delete the digital signatures.
        document?.removeSignature(signature)
    }
}
```

```
// Get the digital signatures object.
NSURL *url = [NSURL fileURLWithPath:@"file path"];

CPDFDocument *document = [[CPDFDocument alloc] initWithURL:url];
NSArray *signatures = [document signatures];

CPDFSignature *signature = signatures[0];

// Delete the digital signatures.
[document removeSignature:signature];
```

3.12 Measurement

3.12.1 Introduction

The ComPDFKit PDF SDK supports drawing annotations of lines, polylines, and polygons with measurement information in PDF documents to measure distances, perimeters, and areas of specified parts in the document. By setting a scale, such as specifying that one inch on the PDF document corresponds to one foot in actual distance, you can measure the actual dimensions of objects in architectural design drawings.

Advantages of ComPDFKit Measurement

- **Complete Tools:** Use line, polyline, rectangle, and polygon measurement tools to accurately draw and measure complex graphic dimensions.
- **Comprehensive Data:** Measure distances, perimeters, and areas, suitable for various scenarios.
- **Configurable Properties:** Freely set global scales, unit conversions, and precision settings.
- **Adjust Measurement Annotations:** Adjust the scale, unit conversions, and precision of existing measurement annotations.
- **Set Appearance:** Customize the line style, color, font, etc., of measurement annotations.
- **Rapid Integration of UI:** Achieve rapid integration and customization through extensible UI components.

Features Supported by ComPDFKit Measurement

Configure Measurement Properties

Before using the measurement feature, configure the scale, units, and precision.

Measure Distance

The distance measurement tool allows your users to measure the distance between two points in the PDF.

Measure Perimeter and Area

Use the perimeter and area measurement tools to measure the perimeter and area of enclosed shapes in the PDF.

Adjust Existing Measurement Annotations

Adjust properties related to measurement in existing measurement annotations.

3.12.2 Configure Measurement Properties

Setting the measurement scale and precision by configuring the `measureInfo` property in `CPDFLineAnnotation`, `CPDFPolylineAnnotation`, and `CPDFPolygonAnnotation` annotations.

Below is an example code for configuring measurement properties:

```
// Get the page object where the annotation needs to be created
if let page = document?.page(at: 0) {
 // Create a line annotation
  var anotation = CPDFLineAnnotation(document: document)
```

```
  // Set the start and end points
  anotation?.startPoint = CGPoint(x: 350, y: 270)
  anotation?.endPoint = CGPoint(x: 260, y: 370)

  // Create measurement properties, set measurement scale and precision (default scale 1
cm = 1 cm)
  var measureInfo = CPDFDistanceMeasureInfo()
  measureInfo.rulerBase = 1.0
  measureInfo.rulerBaseUnit = CPDFMeasureConstants.sharedInstance().cpdfCm
  measureInfo.rulerTranslate = 1.0;
  measureInfo.rulerTranslateUnit = CPDFMeasureConstants.sharedInstance().cpdfKm

  // Set precision to 0.01
  measureInfo.precision = CPDFMeasureConstants.sharedInstance().precisionValueTwo

  // Set the length of the leader line at both ends (this property is unique to
CPDFDistanceMeasureInfo)
  measureInfo.leadLength = 5.0

  // Set the measurement properties for the line annotation
  // Set MeasureInfo: This information will not be effective until it is added to the
page
  anotation?.measureInfo = measureInfo;

  // Update the appearance of the annotation to display it on the document
  page.addAnnotation(anotation!)
}
```

```
// Get the page object where the annotation needs to be created
CPDFPage *page = [document pageAtIndex:0];

// Create a line annotation
CPDFLineAnnotation *anotation = [[CPDFLineAnnotation alloc] initWithDocument:document];

// Set the start and end points
anotation.startPoint = CGPointMake(350, 270);
anotation.endPoint = CGPointMake(260, 370);

// Create measurement properties, set measurement scale and precision (default scale 1 cm
= 1 cm)
CPDFDistanceMeasureInfo *measureInfo = [[CPDFDistanceMeasureInfo alloc] init];
measureInfo.rulerBase = 1.0;
measureInfo.rulerBaseUnit = [CPDFMeasureConstants sharedInstance].cpdfCm;
measureInfo.rulerTranslate = 1.0;
measureInfo.rulerTranslateUnit = [CPDFMeasureConstants sharedInstance].cpdfKm;

 // Set precision to 0.01
measureInfo.precision = [CPDFMeasureConstants sharedInstance].precisionValueTwo;

// Set the length of the leader line at both ends (this property is unique to
CPDFDistanceMeasureInfo)
```

```
measureInfo.leadLength = 5.0;

// Set the measurement properties for the line annotation
// Set MeasureInfo: This information will not be effective until it is added to the page
[anotation setMeasureInfo:measureInfo];

// Update the appearance of the annotation to display it on the document
[page addAnnotation:anotation];
```

Supported Measurement Units

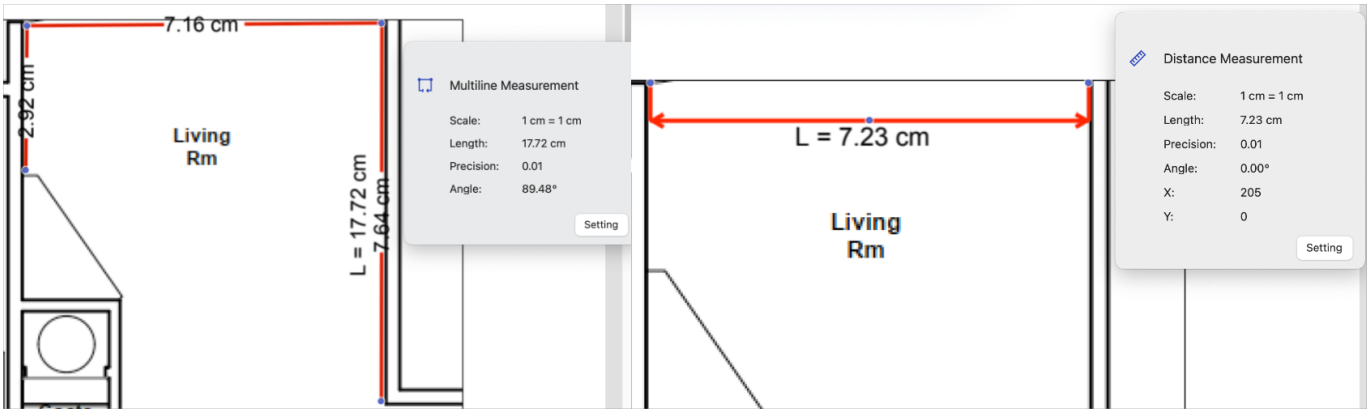
You can set the units for measuring distances on the PDF file and the units for real-world objects using `measureInfo.rulerBaseUnit` and `measureInfo.rulerTranslateUnit`. The supported units and their corresponding constant values are listed in the table below:

Unit	Constant	Value
Point	[CPDFMeasureConstants sharedInstance].cpdfPt	pt
Inch	[CPDFMeasureConstants sharedInstance].cpdfIn	in
Millimeter	[CPDFMeasureConstants sharedInstance].cpdfMm	mm
Centimeter	[CPDFMeasureConstants sharedInstance].cpdfCm	cm
Meter	[CPDFMeasureConstants sharedInstance].cpdfM	m
Kilometer	[CPDFMeasureConstants sharedInstance].cpdfKm	km
Foot	[CPDFMeasureConstants sharedInstance].cpdfFt	ft
Yard	[CPDFMeasureConstants sharedInstance].cpdfYd	yd
Mile	[CPDFMeasureConstants sharedInstance].cpdfMi	mi

3.12.3 Measure Distance

Distance measurement annotations allow your users to measure the distance between two points representing objects on a plane, such as buildings, streets, or walls. With this annotation, users can simply provide the starting and ending points to obtain the distance between the two points.

There are two types of distance measurements: line segment measurement annotations and polyline measurement annotations. Line segment measurement annotations measure the distance between the starting and ending points, while polyline measurement annotations measure the distance between all adjacent points along the polyline and calculate the total length.



Here is an example code for creating a polyline distance measurement annotation:

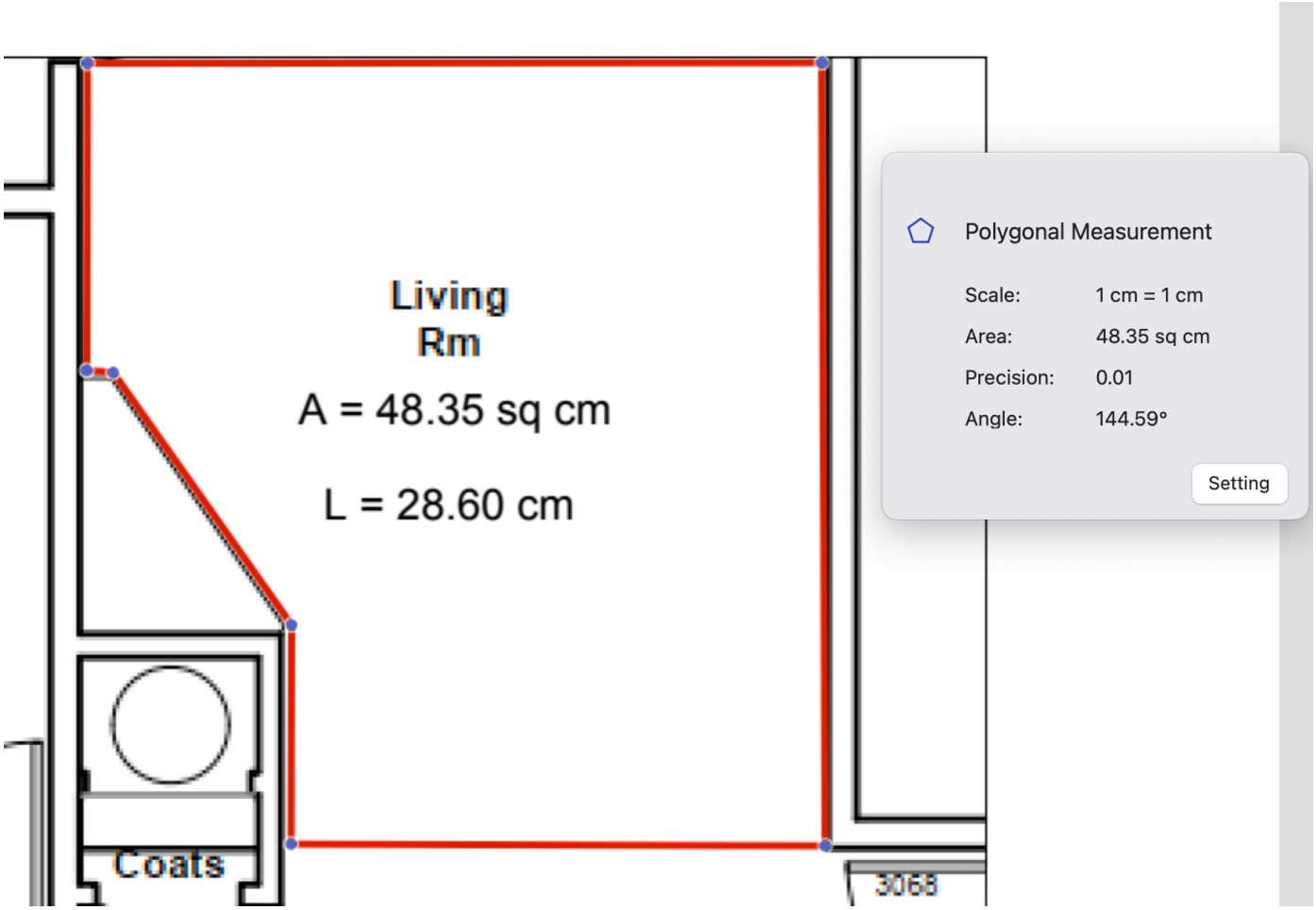
```
// Get the page object where the annotation needs to be created
if let page = document?.page(at: 0) {

  // Create a polyline annotation
  var anotation = CPDFPolylineAnnotation(document: document)

  // Set the vertex of the polyline
  var pointsArray: [NSValue] = []
  let point1 = CGPoint(x: 100, y: 100)
  pointsArray.append(NSValue(cgPoint: point1))
  let point2 = CGPoint(x: 100, y: 200)
  pointsArray.append(NSValue(cgPoint: point2))
  let point3 = CGPoint(x: 200, y: 200)
  pointsArray.append(NSValue(cgPoint: point3))
  anotation?.setSavePoints(pointsArray)

  // Set the measurement properties for the polyline annotation
  var measureInfo = CPDFPerimeterMeasureInfo()
  //Set MeasureInfo: This information will not be effective until it is added to the page
  anotation.measureInfo = measureInfo;

   // Update the appearance of the annotation to display it on the document
  page.addAnnotation(anotation!)
}
```

```
// Get the page object where the annotation needs to be created
CPDFPage *page = [document pageAtIndex:0];

// Create a polyline annotation
CPDFPolylineAnnotation *anotation = [[CPDFPolylineAnnotation alloc]
initWithDocument:document];

 // Set the vertex of the polyline
NSMutableArray<NSValue *> *pointsArray = [NSMutableArray array];
CGPoint point1 = CGPointMake(100, 100);
[pointsArray addObject:[NSValue valueWithCGPoint:point1]];
CGPoint point2 = CGPointMake(100, 200);
```

```
[pointsArray addObject:[NSValue valueWithCGPoint:point2]];
CGPoint point3 = CGPointMake(200, 200);
[pointsArray addObject:[NSValue valueWithCGPoint:point3]];
[anotation setSavePoints:savePoints];

 // Set the measurement properties for the polyline annotation
CPDFPerimeterMeasureInfo *measureInfo = [[CPDFPerimeterMeasureInfo alloc] init];
//Set MeasureInfo: This information will not be effective until it is added to the page
[anotation setMeasureInfo:measureInfo];

// Update the appearance of the annotation to display it on the document
[page addAnnotation:anotation];
```

3.12.4 Measure Perimeter and Area

Polygon measurement annotations can be used to measure the perimeter and area of a polygonal region selected by the user.



Below is an example code for creating a polygon measurement annotation:

```
// Get the page object where the annotation needs to be created
if let page = document?.page(at: 0) {

  // Create a polygon annotation
```

```
  var anotation = CPDFPolygonAnnotation(document: document)

   // Set the vertex of the polygon
  var pointsArray: [NSValue] = []
  let point1 = CGPoint(x: 100, y: 100)
  pointsArray.append(NSValue(cgPoint: point1))
  let point2 = CGPoint(x: 100, y: 200)
  pointsArray.append(NSValue(cgPoint: point2))
  let point3 = CGPoint(x: 200, y: 200)
  pointsArray.append(NSValue(cgPoint: point3))
  anotation?.setSavePoints(pointsArray)

  // Set the measurement properties for the polygon annotation
  var measureInfo = CPDFAreaMeasureInfo()
  // (for closed graphics where the area can be measured) Setting displays the area and
circumference of the graph in the comment appearance.
  measureInfo.captionType = [.length, .area]

  //Set MeasureInfo: This information will not be effective until it is added to the page
  anotation.measureInfo = measureInfo;
   // Update the appearance of the annotation to display it on the document
  page.addAnnotation(anotation!)
}
```

```
// Get the page object where the annotation needs to be created
CPDFPage *page = [document pageAtIndex:0];

// Create a polygon annotation
CPDFPolygonAnnotation *anotation = [[CPDFPolygonAnnotation alloc]
initWithDocument:document];

// Set the vertex of the polygon
NSMutableArray<NSValue *> *pointsArray = [NSMutableArray array];
CGPoint point1 = CGPointMake(100, 100);
[pointsArray addObject:[NSValue valueWithCGPoint:point1]];
CGPoint point2 = CGPointMake(100, 200);
[pointsArray addObject:[NSValue valueWithCGPoint:point2]];
CGPoint point3 = CGPointMake(200, 200);
[pointsArray addObject:[NSValue valueWithCGPoint:point3]];
[anotation setSavePoints:savePoints];

 // Set the measurement properties for the polygon annotation
CPDFAreaMeasureInfo *measureInfo = [[CPDFAreaMeasureInfo alloc] init];
// (for closed graphics where the area can be measured) Setting displays the area and
circumference of the graph in the comment appearance.
measureInfo.captionType = CPDFCaptionTypeLength | CPDFCaptionTypeArea;

//Set MeasureInfo: This information will not be effective until it is added to the page
[anotation setMeasureInfo:measureInfo];

 // Update the appearance of the annotation to display it on the document
```

```
[page addAnnotation:anotation];
```

3.12.5 Adjust Existing Measurement Annotations

For existing measurement annotations, when changing the measurement points (such as `savePoints` for `CPDFPolylineAnnotation`), it's necessary to refresh the measurement properties to obtain the new measurement values.

Note: You cannot remove the measurement properties by setting them to nil.

Below is an example code for modifying an existing polyline distance measurement annotation:

```
// Get the page object where the annotation needs to be created
if let page = document?.page(at: 0) {

  // Create a polyline annotation
  var anotation = CPDFPolylineAnnotation(document: document)

   // Set the vertex of the polyline
  var pointsArray: [NSValue] = []
  let point1 = CGPoint(x: 100, y: 100)
  pointsArray.append(NSValue(cgPoint: point1))
  let point2 = CGPoint(x: 100, y: 200)
  pointsArray.append(NSValue(cgPoint: point2))
  let point3 = CGPoint(x: 200, y: 200)
  pointsArray.append(NSValue(cgPoint: point3))
  anotation?.setSavePoints(pointsArray)

  // Set the measurement properties for the polyline annotation
  var measureInfo = CPDFPerimeterMeasureInfo()
  anotation.measureInfo = measureInfo;

   // Update the appearance of the annotation to display it on the document
  page.addAnnotation(anotation!)

  // Add a new vertex to the polyline
  let point4 = CGPoint(x: 200, y: 300)
  pointsArray.append(NSValue(cgPoint: point4))
  anotation?.setSavePoints(pointsArray)

  // Refresh the measurement properties to recalculate
  // Set MeasureInfo: This information will not be effective until it is added to the
page
  anotation.measureInfo = measureInfo;

  // Refresh the appearance of the annotation
  anotation.updateAppearanceStream()
}
```

```
// Get the page object where the annotation needs to be created
CPDFPage *page = [document pageAtIndex:0];

// Create a polyline annotation
CPDFPolylineAnnotation *anotation = [[CPDFPolylineAnnotation alloc]
initWithDocument:document];

// Set the vertex of the polyline
NSMutableArray<NSValue *> *pointsArray = [NSMutableArray array];
CGPoint point1 = CGPointMake(100, 100);
[pointsArray addObject:[NSValue valueWithCGPoint:point1]];
CGPoint point2 = CGPointMake(100, 200);
[pointsArray addObject:[NSValue valueWithCGPoint:point2]];
CGPoint point3 = CGPointMake(200, 200);
[pointsArray addObject:[NSValue valueWithCGPoint:point3]];
[anotation setSavePoints:savePoints];

// Set the measurement properties for the polyline annotation
CPDFPerimeterMeasureInfo *measureInfo = [[CPDFPerimeterMeasureInfo alloc] init];
[anotation setMeasureInfo:measureInfo];

 // Update the appearance of the annotation to display it on the document
[page addAnnotation:anotation];

// Add a new vertex to the polyline
CGPoint point4 = CGPointMake(200, 300);
[pointsArray addObject:[NSValue valueWithCGPoint:point4]];
[anotation setSavePoints:savePoints];

// Refresh the measurement properties to recalculate
// Set MeasureInfo: This information will not be effective until it is added to the page
 anotation.measureInfo = measureInfo;

// Refresh the appearance of the annotation
[anotation updateAppearanceStream];
```

3.13 Optimization

3.13.1 Introduction

The ComPDFKit PDF SDK optimizes PDF documents by reducing file size, optimizing page content, removing redundant information, and compressing data streams using the latest image compression technologies, while allowing control over the compression accuracy of images.

Advantages of ComPDFKit Optimization

- **Precision Control**: Regulate the precision of optimization by controlling the compression accuracy of images during document optimization.

- **Redundant Information Removal**: Remove redundant information from documents, such as invalid links and bookmarks, through configuration parameters.
- **Page Content Optimization**: Optimize the page content of documents through configuration parameters.
- **Rapid UI Integration**: Achieve rapid integration and customization through extensible UI components.

Functions Supported by ComPDFKit Document Optimization

- **Compress and Optimize PDF Files**

  Examples of how to use the ComPDFKit PDF SDK to optimize documents through code.

3.13.2 Compress & Optimize a PDF File

Control the compression accuracy of images and observe the compression progress in real-time by setting the `CPDFDocumentOptimizeOption` parameter.

Below is an example code for configuring the compression and optimization of PDF files:

```
 // Compress & Optimize a PDF File
document?.writeOptimize(to: url, withOptions:
[CPDFDocumentOptimizeOption.imageQualityOption : 60], progressHandler: { pageIndex,
pageCount in
   // Monitor the current compression progress of the page index value and the total
number of pages in the document
}, cancelHandler: {
  // Whether to cancel the current compression
    return NO
}, completionHandler: { isSuccess in
  // Compression success
   }
})
```

```
  // Compress & Optimize a PDF File
[document writeOptimizeToURL:[NSURL fileURLWithPath:targetPath]
                         withOptions:@{CPDFDocumentImageQualityOption:compressType}
                     progressHandler:^(float cPageIndex, float totalPageIndex) {
      // Monitor the current compression progress of the page index value and the total
number of pages in the document
  }
                       cancelHandler:^BOOL{
      // Whether to cancel the current compression
      return NO;
  }
                   completionHandler:^(BOOL finished) {

      // Compression success
  }];
```

4 Support

4.1 Reporting Problems

Thank you for your interest in ComPDFKit PDF SDK, an easy-to-use and powerful development solution to integrate high quality PDF rendering capabilities to your applications. If you encounter any technical questions or bug issues when using ComPDFKit PDF SDK for iOS, please submit the problem report to the ComPDFKit team. More information as follows would help us to solve your problem:

- ComPDFKit PDF SDK product and version.
- Your operating system and IDE version.
- Detailed descriptions of the problem.
- Any other related information, such as an error screenshot.

4.2 Contact Information

Website:

- Home Page: https://www.compdf.com
- API Page: https://api.compdf.com/
- Developer Guides: https://www.compdf.com/guides/pdf-sdk/ios/overview
- API Reference: https://developers.compdf.com/guides/pdf-sdk/ios/api-reference/index
- Code Examples: https://www.compdf.com/guides/pdf-sdk/ios/examples

Contact ComPDFKit:

- Contact Sales: https://api.compdf.com/contact-us
- Technical Issues Feedback: https://www.compdf.com/support
- Contact Email: support@compdf.com

Thanks,
The ComPDFKit Team