# ComPDFKit PDF SDK

## Developer Guides

**Comprehensive PDF SDK for Developers**

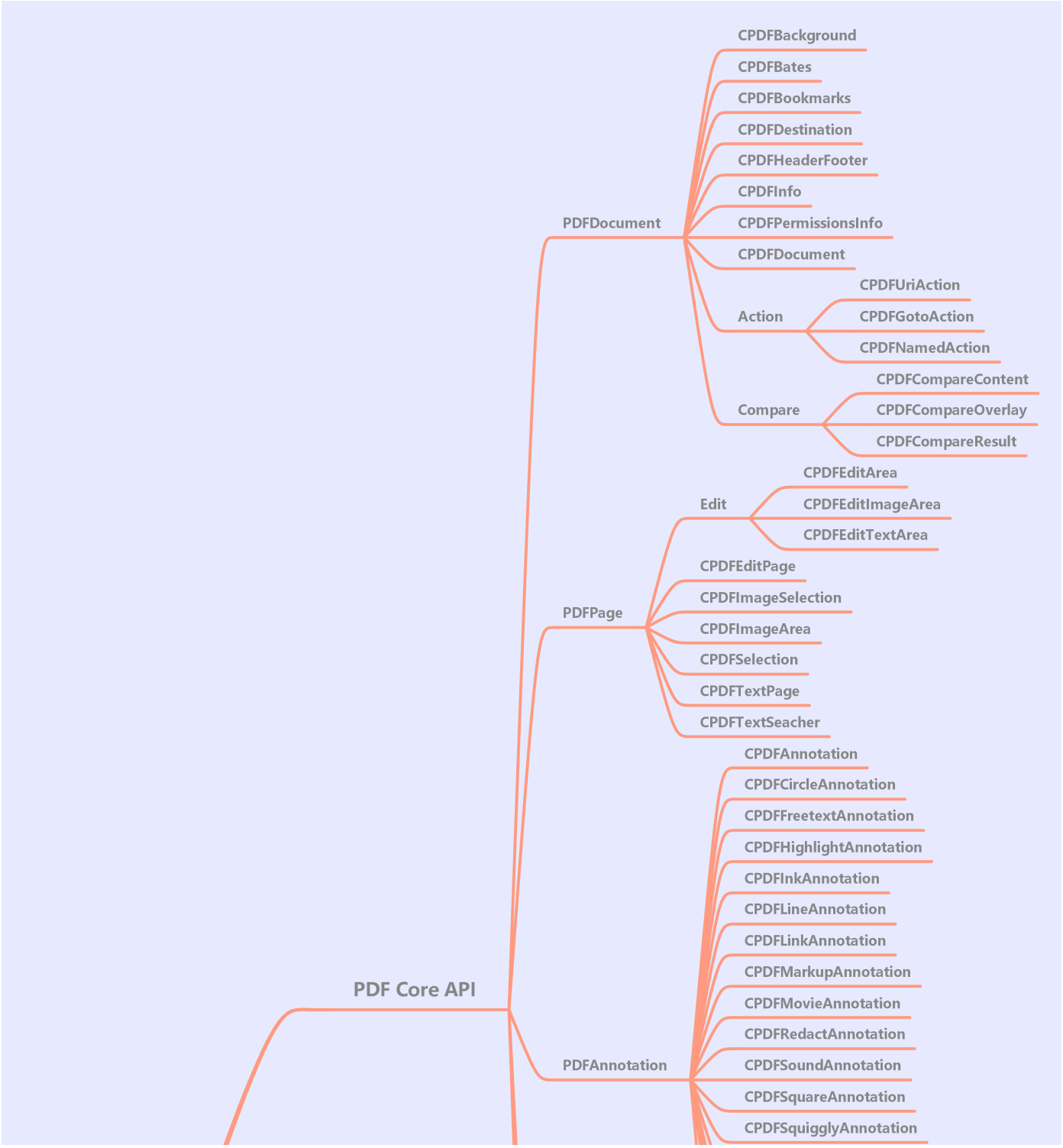
# Contents

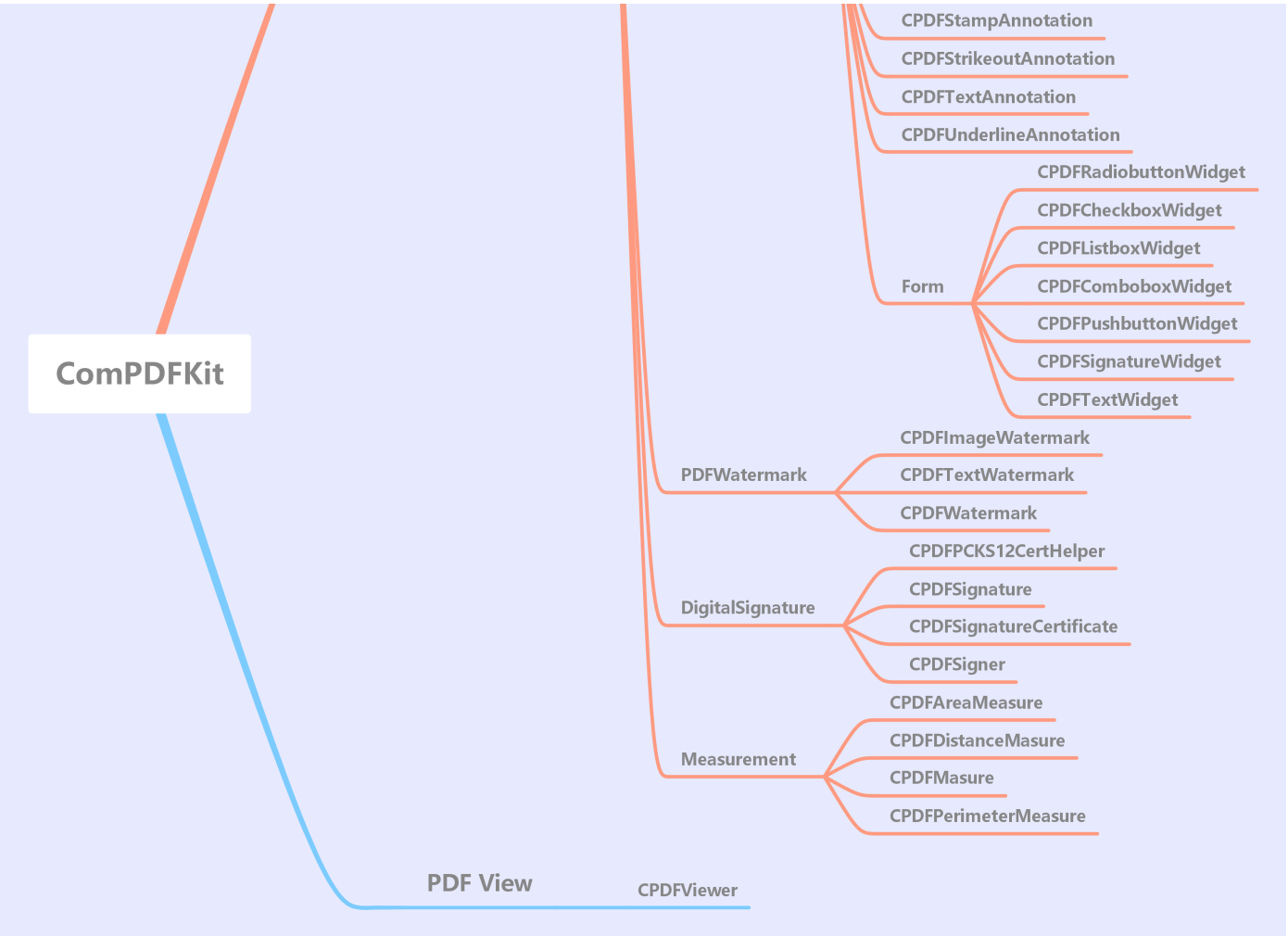
# 1 Overview

ComPDFKit PDF SDK for Windows is a powerful PDF library that ships with an easy-to-use C# interface. Developers can seamlessly integrate PDF rendering, navigation, creation, searching, annotation, PDF text extraction, form data collection, and editing capabilities into their applications and services.

## 1.1 ComPDFKit PDF SDK

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






The two elements for ComPDFKit:

- **PDF Core API**

  The ComPDFKit.NET can be used independently for document rendering, analysis, text extraction, text search, form filling, annotation creation and manipulation, and much more.

- **PDF View**

  The ComPDFKit.Viewer 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.

## 1.2 Key Features

**Viewer** component offers:

- Standard page display modes, including Single Page, Double Page, Scrolling, 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.

**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, include: Delete pages, Insert pages, Move pages, Rotate pages, Replace pages, and Exchange pages.
- Document information setting.
- Extract images.

**Content Editor** component offers:

- Programmatically add and remove text 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.
- Support editing PDF images like moving, rotating, scaling, mirroring, cropping, replacing, copying, and extracting.
- Search for a keyword and replace it with other content.
- Undo or redo any change.

**Security** component offers:

- Encrypt and decrypt PDFs, including Permission setting and Password protected.
- Create 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.

**Document Comparison** component offers:

- Compare different versions of a document, including overlay comparison and content comparison.

**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 PDF SDK in your Windows application with a few lines of C# code. Take just a few minutes and get started.

The following sections introduce the requirements, structure of the installation package, and how to make a Windows PDF Reader in C# with ComPDFKit PDF SDK.

## 2.1 Requirements

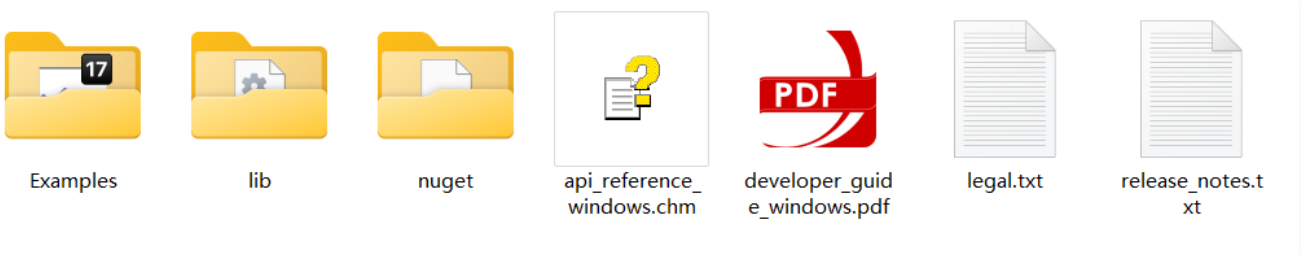
- Windows 7, 8, 10, and 11 (32-bit and 64-bit).
- Visual Studio 2017 or higher (Make sure the **.NET Desktop Development** is installed).

- .NET Framework 4.5 or higher.

## 2.2 Windows Package Structure

You can contact us and access our PDF SDK installation package. The SDK package includes the following files.

- ***"Examples"*** - A folder containing Windows 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.
  - ***"PDFViewer"*** - A multi-functional PDF program that integrates all of the above features.
  - ***"ComPDFKit.Tool"*** - An auxiliary utility class for CPDFViewer, used to implement a series of operations including annotation editing, form editing, content editing, and more.
  - ***"ComPDFKit.Controls"*** - A default control library for quickly building various function modules of PDF viewer.
  - ***"license_key_win.xml"*** - A xml file containing key and secret.
  - ***"TestFile"*** - A folder containing test files.
  - ***"Samples"*** - -A folder containing console application.
- ***"lib"*** - Include the ComPDFKit dynamic library (x86,x64).
- ***"nuget"*** - Include the ComPDFKit.NetFramework nuget package file.
- ***"api_reference_windows.chm"*** - API reference.
- ***"developer_guide_windows.pdf"*** - Developer guide.
- ***"legal.txt"*** - Legal and copyright information.
- ***"release_notes.txt"*** - Release information.




## 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 device 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.

ComPDFKit PDF SDK currently supports two types of verification methods: Online License and Offline License.

**Online License:** Online licensing supports dynamically updating license information. If we provide you with new services, you can obtain updates in real-time without any modifications.

**Offline License:** Offline licensing is suitable for scenarios with high security requirements and restricted network access. It allows you to use ComPDFKit PDF SDK when unable to connect to the internet.

## 2.3.1 Obtaining the License Key

If you want to use the ComPDFKit license for your application, we offer two types of licenses: trial license and full license. The full license must be bound to your developer device ID (How to Find the Developer Device ID).

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.

## 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:

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

Offline License:

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

3. Copy the value located at the **LICENSE_KEY** position within the `<key>LICENSE_KEY</key>` field. This is your license key.

## 2.3.3 Apply the License Key

Before using any classes from the ComPDFKit PDF SDK, you need to choose the corresponding scheme from the following two options based on the license type and apply the license to your application.

**Online License**

You can perform online authentication using the following method:

```
public static bool LicenseVerify()
{
  if (!CPDFSDKVerifier.LoadNativeLibrary())
  {
    return false;
  }
  LicenseErrorCode status = CPDFSDKVerifier.OnlineLicenseVerify("Input your license
here.");
  return status == LicenseErrorCode.E_LICENSE_SUCCESS;
}
```

Furthermore, if you need to confirm the communication status with the server during the current online authentication, you can achieve this by implementing the `CPDFSDKVerifier.LicenseRefreshed` callback.:

```
CPDFSDKVerifier.LicenseRefreshed += CPDFSDKVerifier_LicenseRefreshed;

private void CPDFSDKVerifier_LicenseRefreshed(object sender, ResponseModel e)
{
  if(e != null)
  {
    string message = string.Format("{0} {1}", e.Code, e.Message);
    Trace.WriteLine(message);
  }
  else
    {
      Trace.WriteLine("Network not connected.");
    }
}
```

**Offline License**

You can perform offline authentication using the following method:

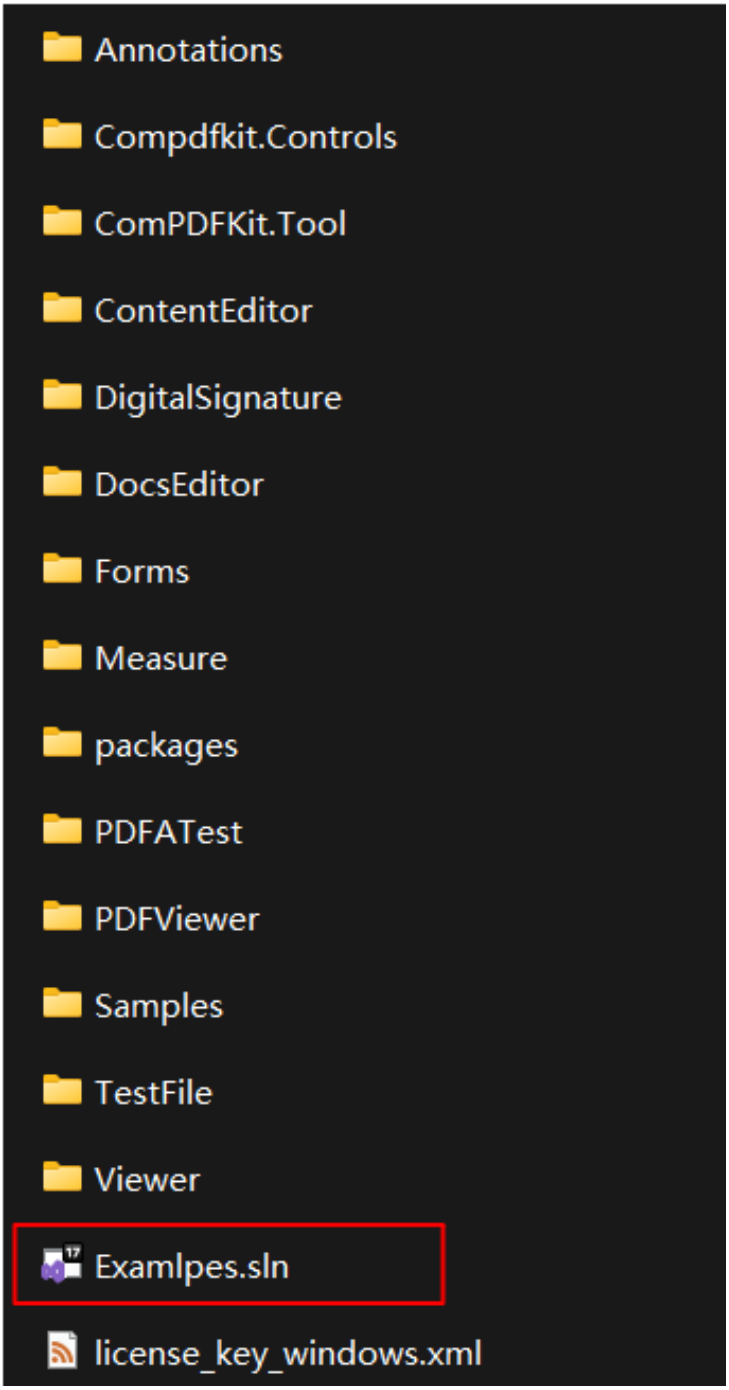
```
bool LicenseVerify()
{
    if (!CPDFSDKVerifier.LoadNativeLibrary())
        return false;

    LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your license
here.", false);
    return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
}
```

## 2.4 How to Run a Demo

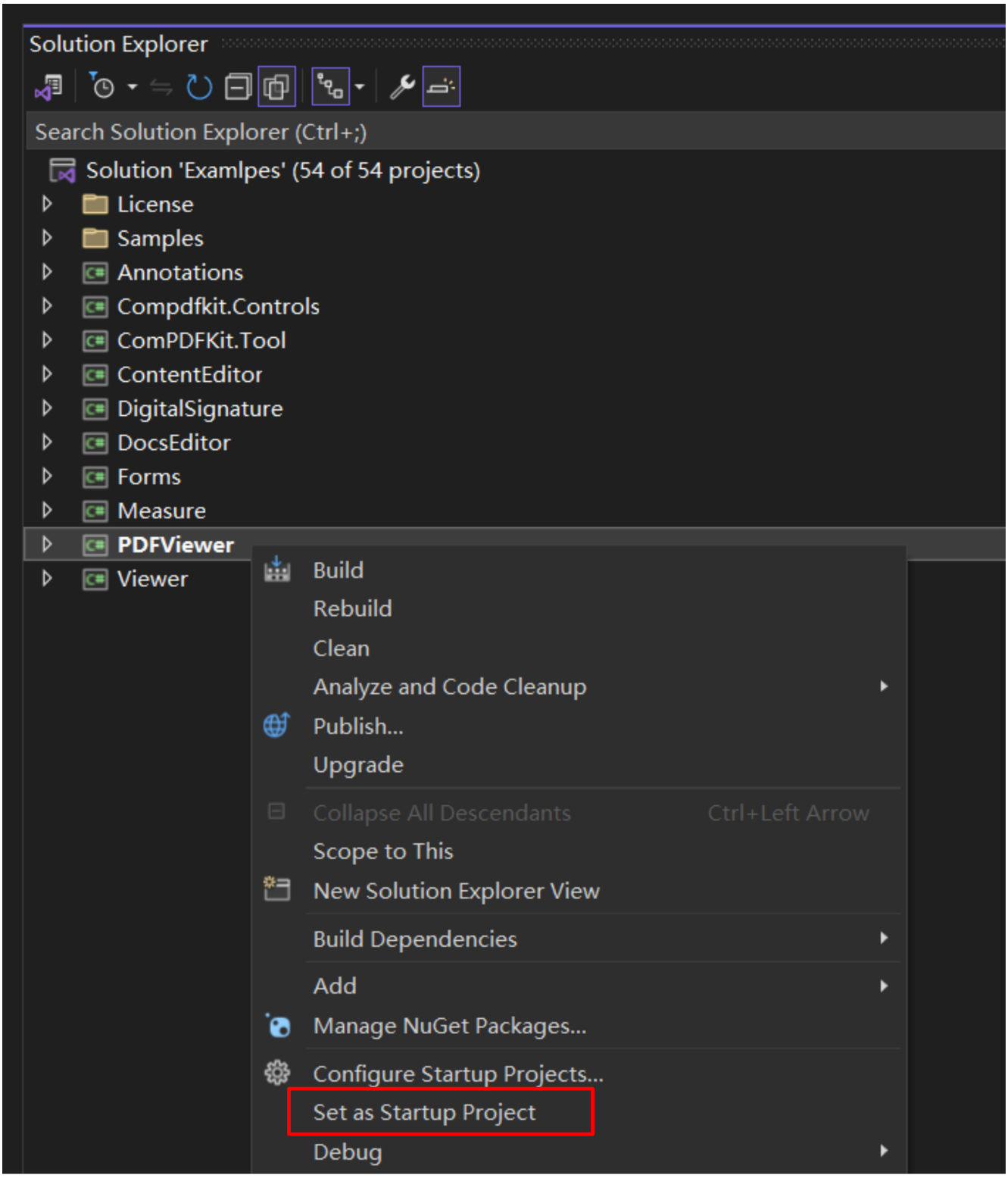
ComPDFKit PDF SDK for Windows provides multiple demos in C# for developers to learn how to call the SDK on Windows. You can find them in the ***"Examples"*** folder.

In this guide, we take ***"PDFViewer"*** as an example to show how to run it in Visual Studio 2022.

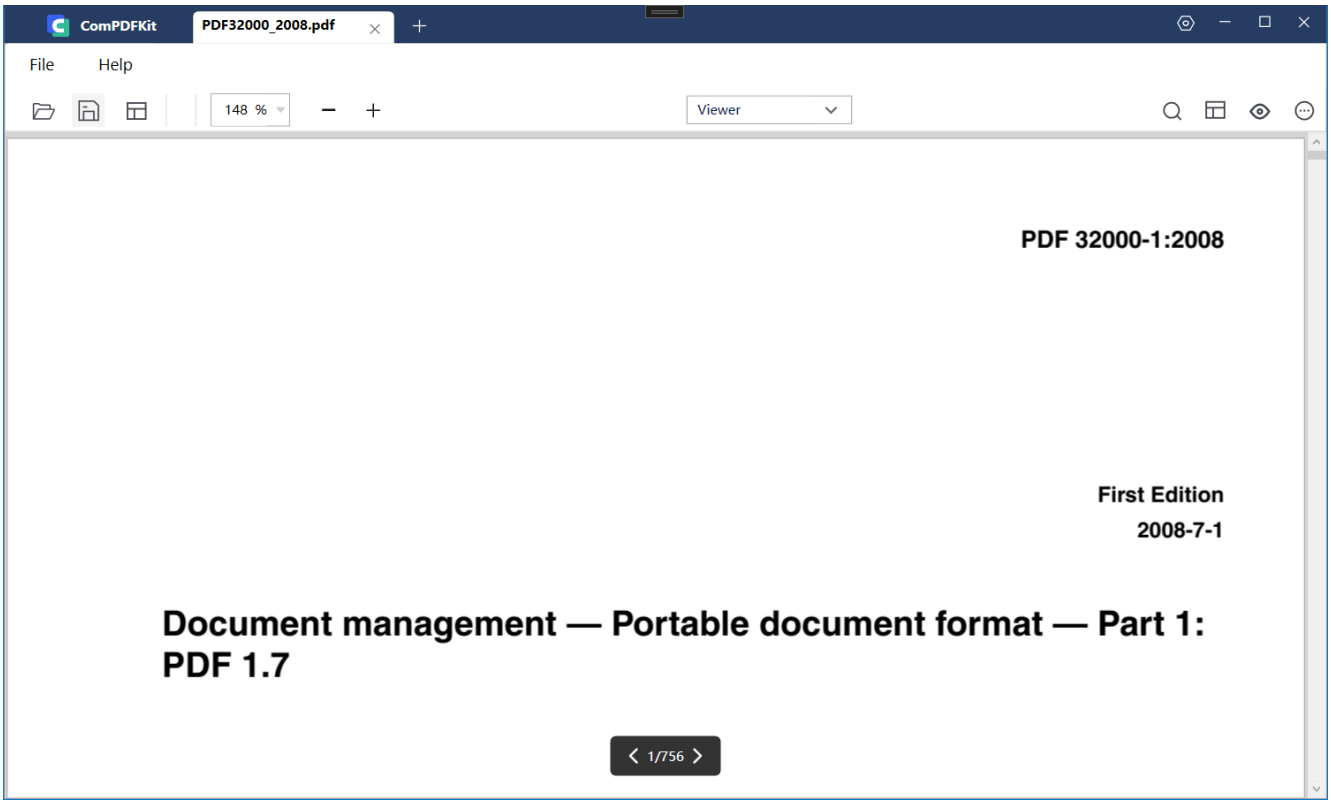
1. Copy your ***"license_key_windows.xml"*** to the ***"Examples"*** folder (The file is the license to make your project run).
2. Find ***"Examples.sln"*** in the ***"Examples"*** folder and open it in Visual Studio 2022.




3. Select ***"PDFViewer"*** and right-click to set it as a startup project.




4. Run the project and then you can open the multifunctional ***"PDFViewer"*** demo.

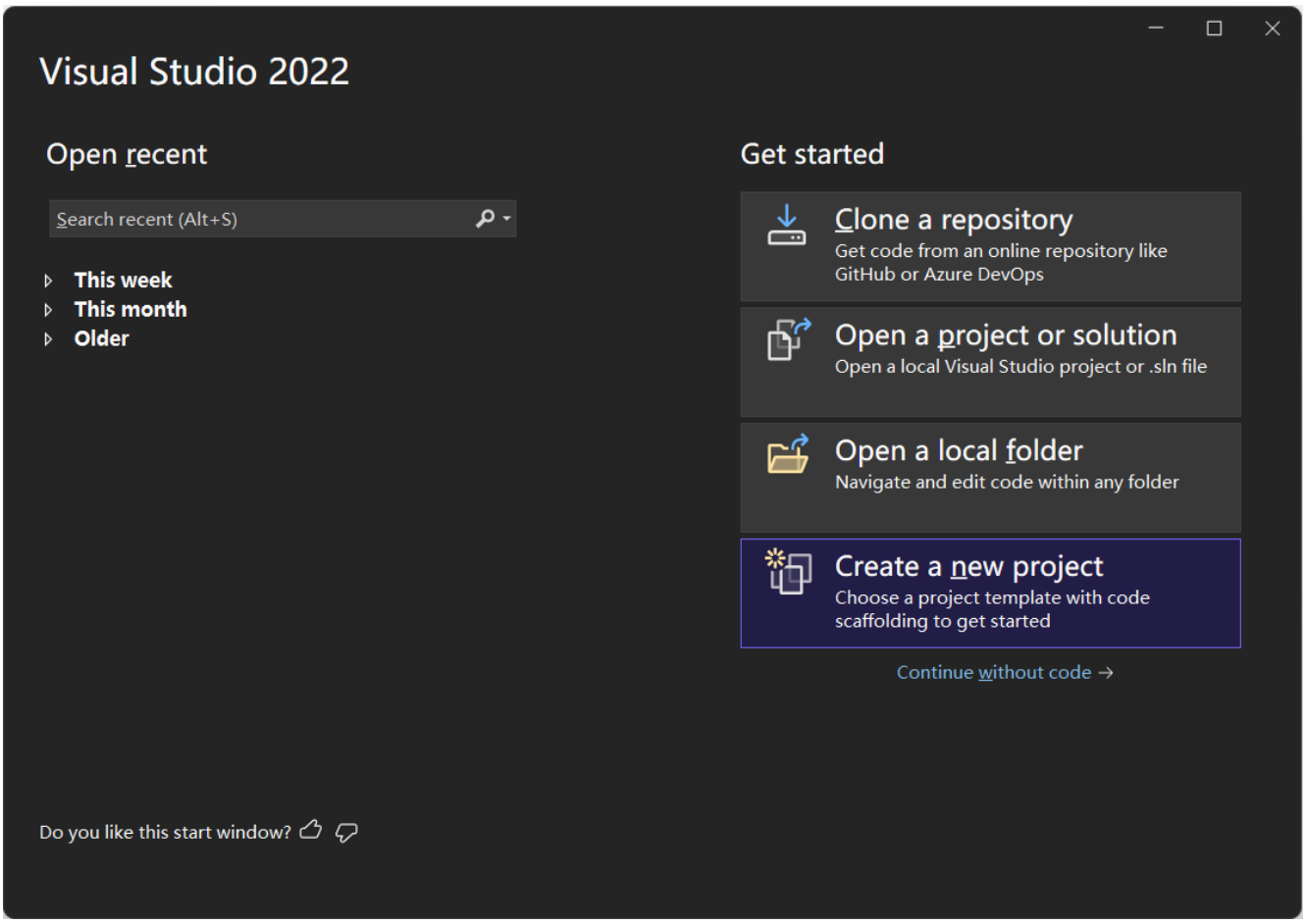



**Note:** *This is a demo project, presenting completed ComPDFKit PDF SDK functions. The functions might be different based on the license you have purchased. Please check that the functions you choose work fine in this demo project.*

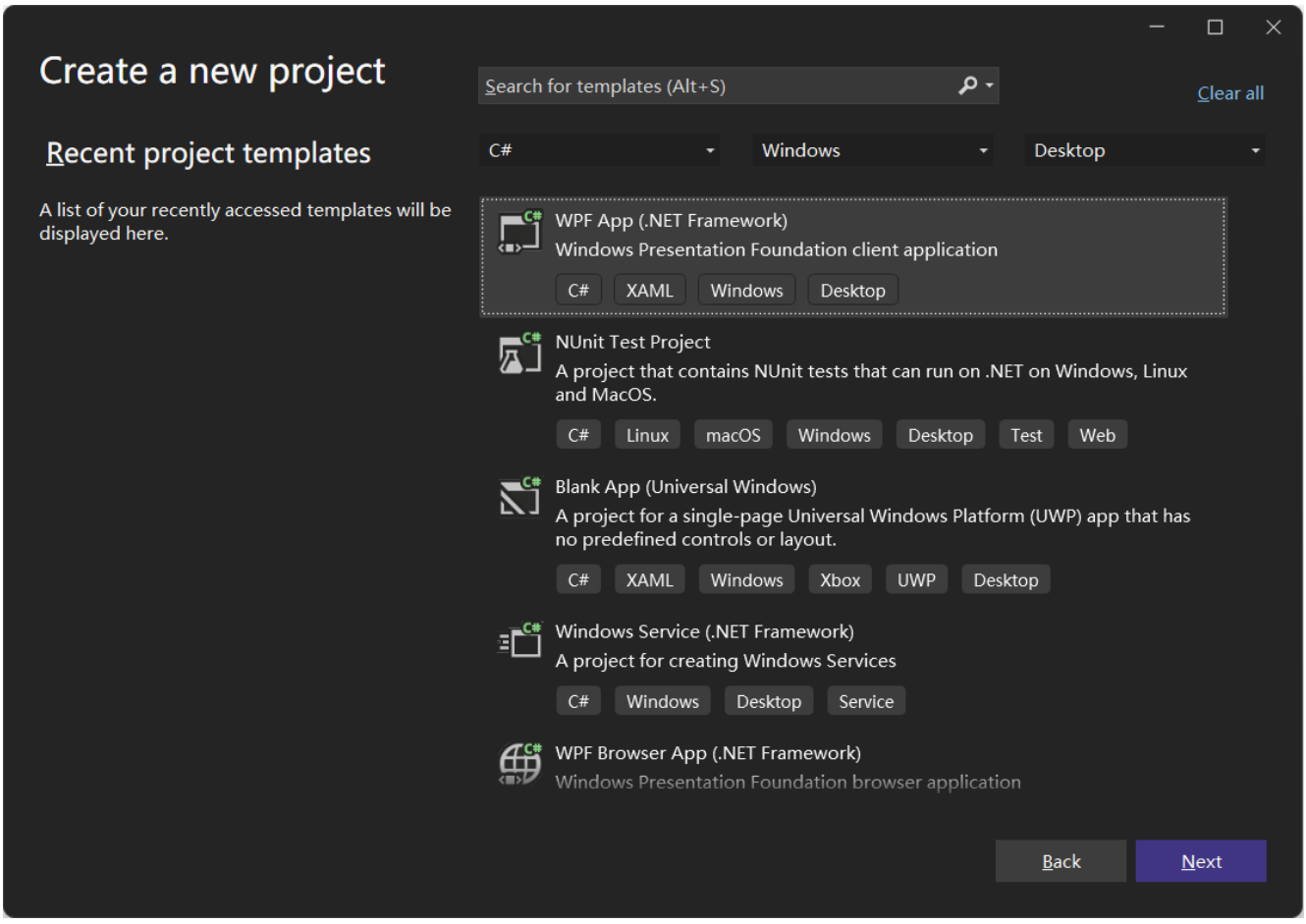
# 2.5 How to Make a Windows Program in C# with ComPDFKit PDF SDK

## 2.5.1 Create a New Project

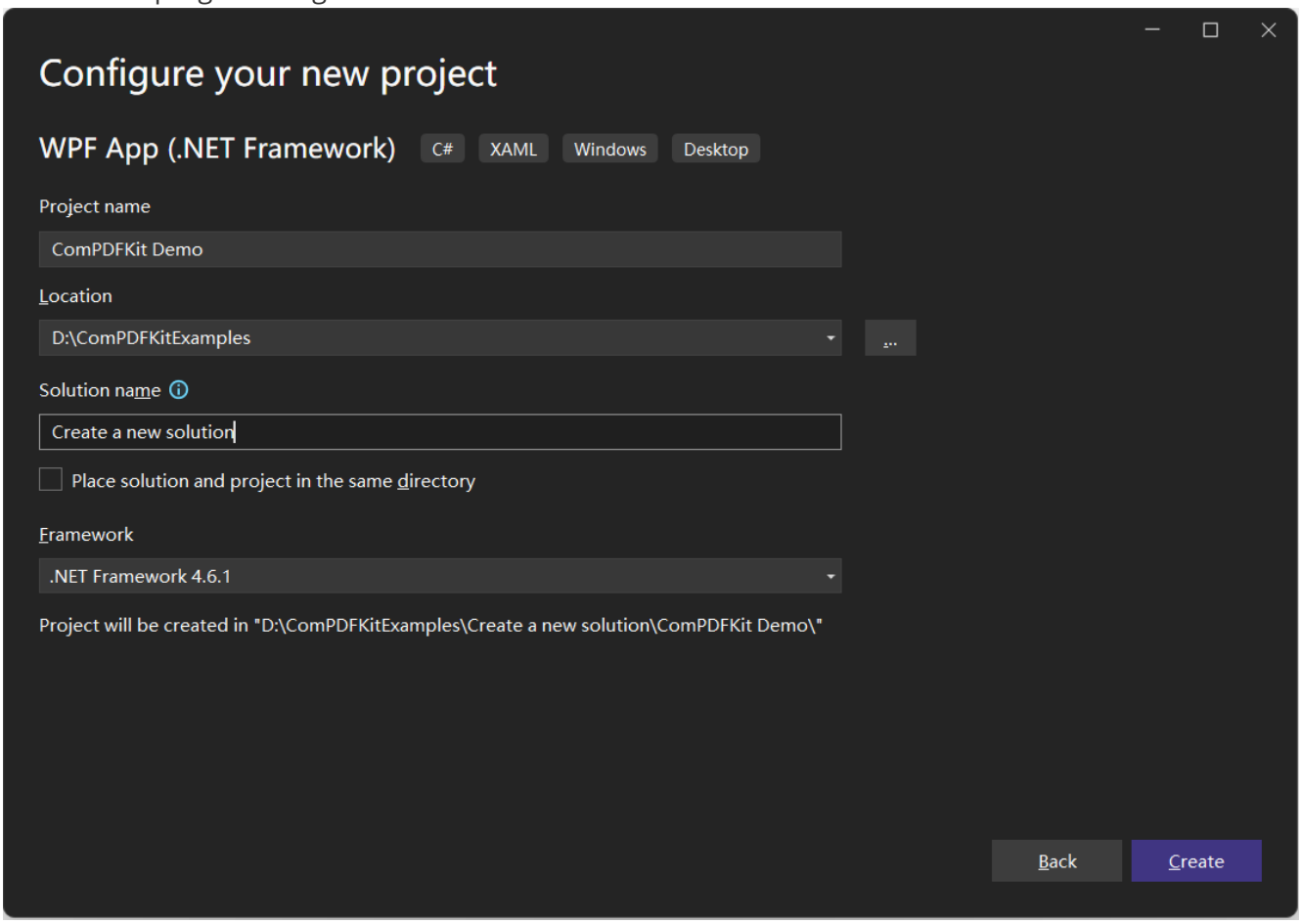
1. Fire up Visual Studio 2022, click **Create a new project**.




2. Choose **WPF App (.NET Framework)** and click **Next**.



3. Configure your project: Set your project name and choose the location to store your program. The project name is called "ComPDFKit Demo" in this example. This sample project uses .NET Framework 4.6.1 as the programming framework.



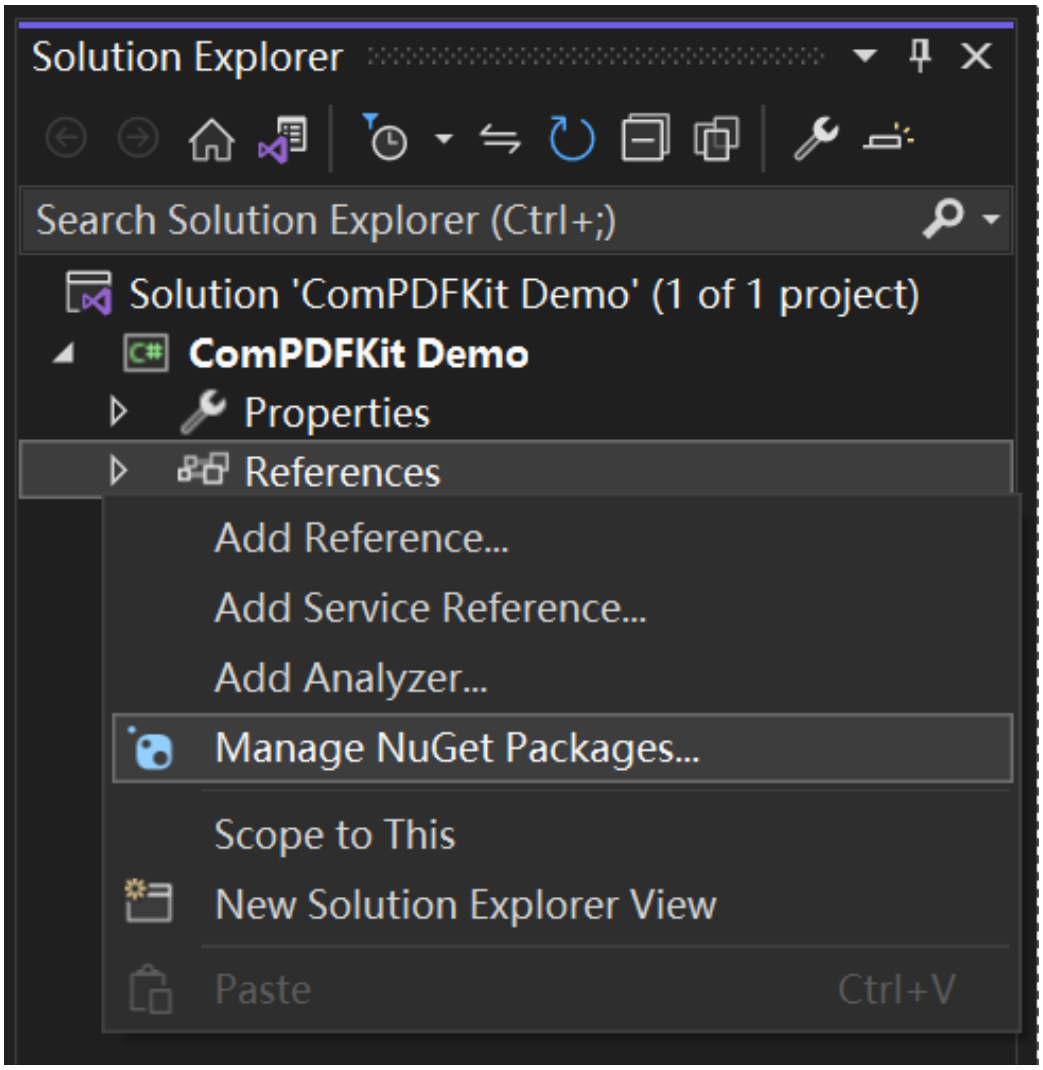

4. Click the **Create** button. Then, the new project will be created.

## 2.5.2 Add ComPDFKit to Your Project

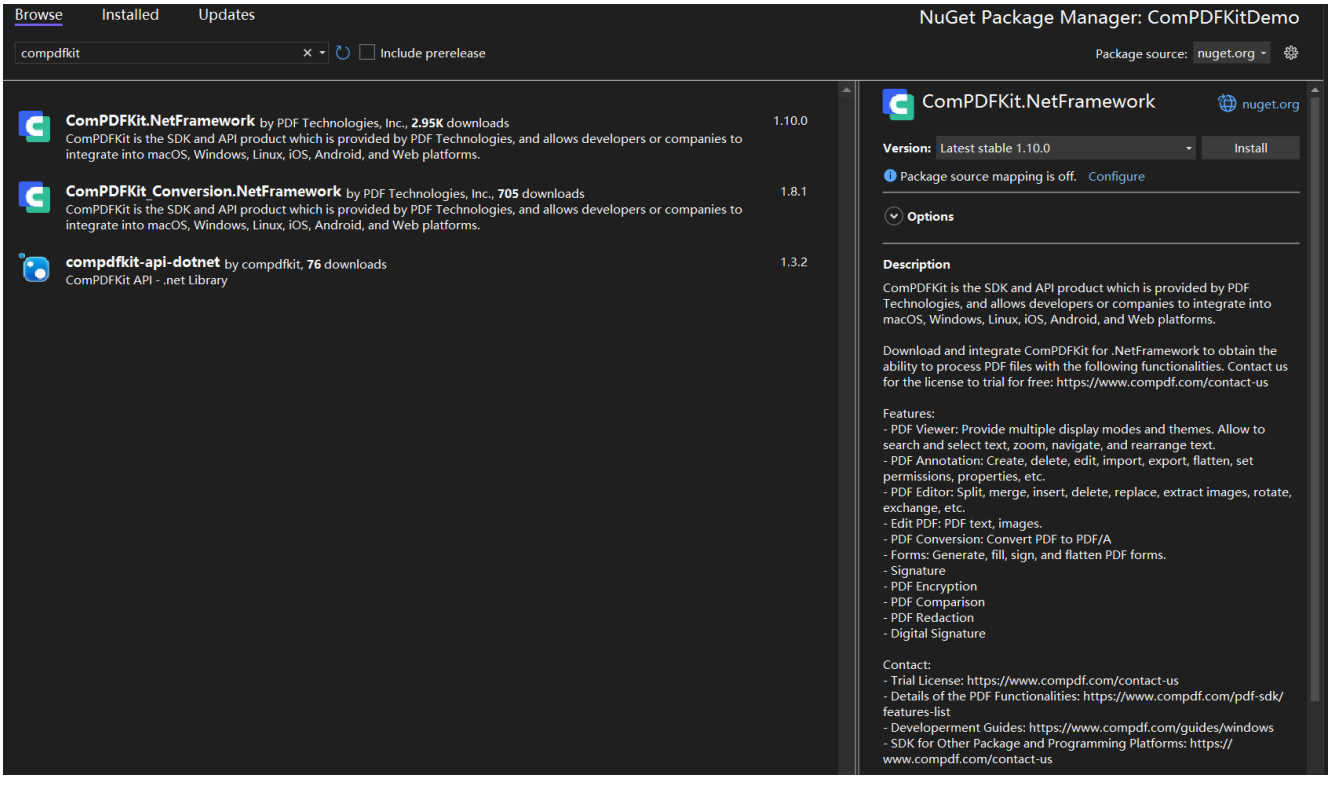
There are two ways to add ComPDFKit to your Project: `Nuget Repository` and `Local Package`, you can choose one or the other according to your needs.

**Nuget Repository**

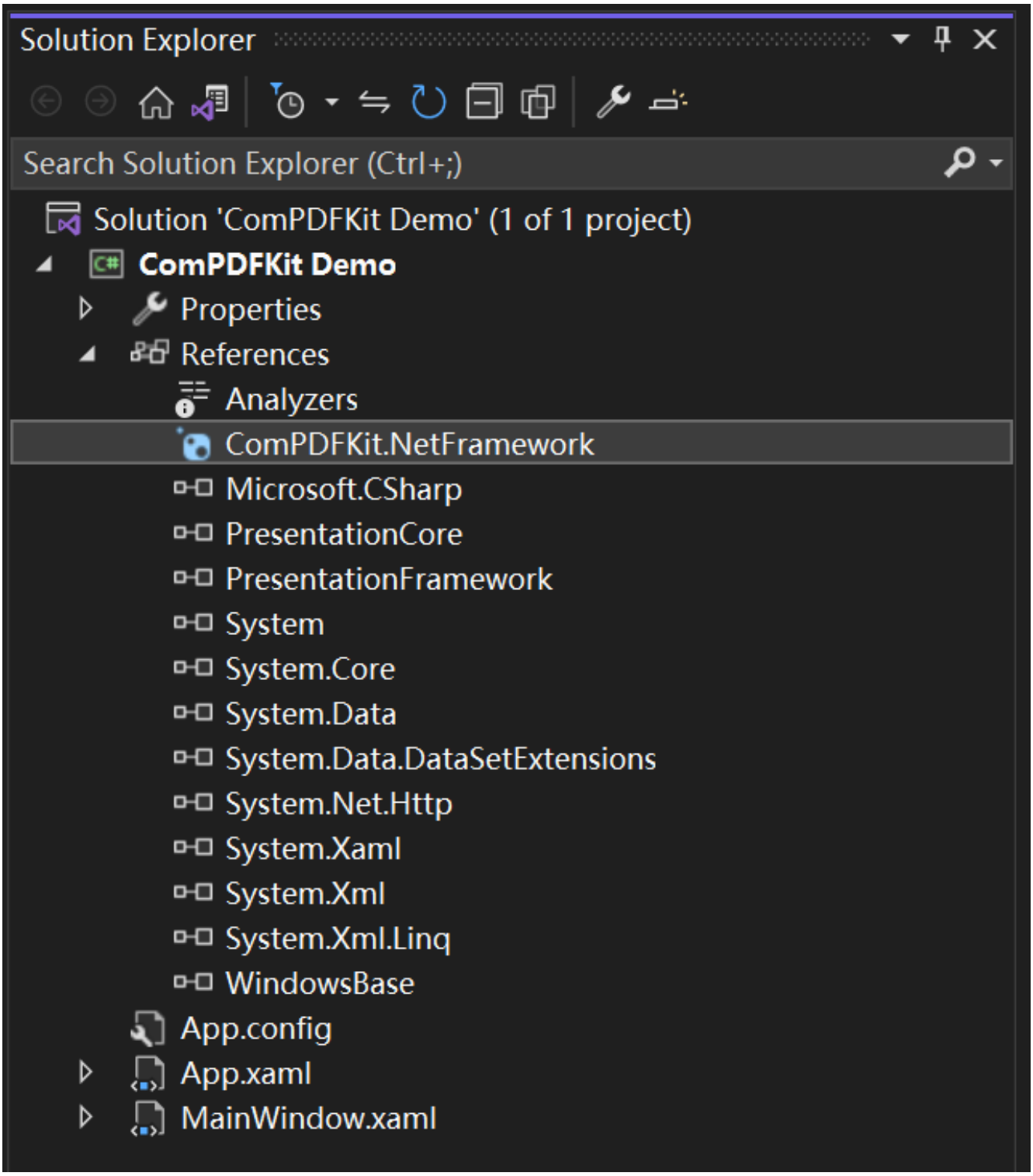
1. Open your project's solution, and in the Solution Explorer, right-click on **References** and click on the menu item **Manage NuGet Packages...**. This will open the NuGet Package Manager for your solution.




2. Search for `ComPDFKit.NetFramework`, and you'll find the package on nuget.org.
3. On the right side, in the panel describing the package, click on the **Install** button to install the package.



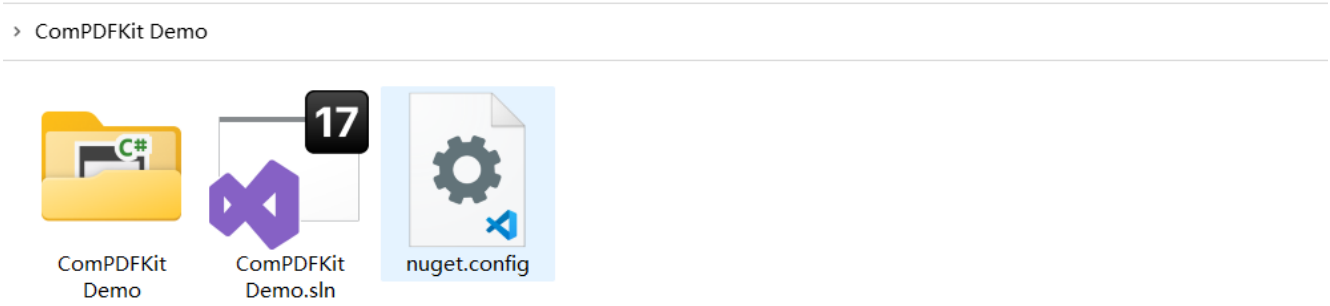
4. Once that is complete, you'll see a reference to the package in the Solution Explorer under **References**.




**Local Package**

Rather than targeting a package held at nuget.org, you may set up a configuration to point to a local package. This can be useful for some situations, for example, your build machines don't have access to the internet.

1. You can find ***"ComPDFKit.NetFramework....nupkg"*** file in the SDK Package
2. Create or edit a ***"nuget.config"*** file in the same directory as your solution file (e.g. ***"ComPDFKit Demo.sln"***).

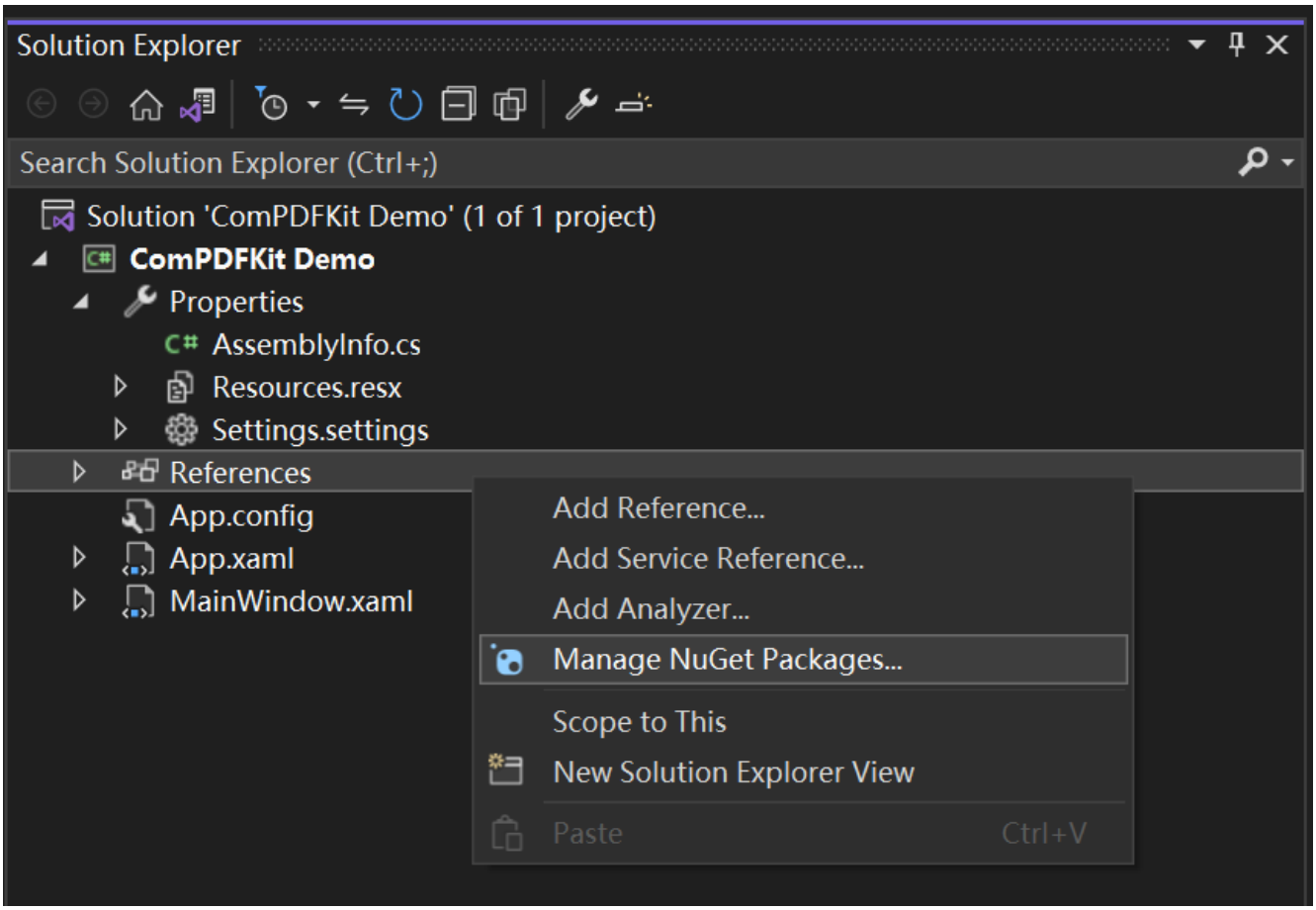



The contents of the file should contain an XML element, `packageSources` — which describes where to find NuGet packages — as a child of a root node named `configuration`. If the file already exists, add the extra `packageSources` entry shown below. If the file is blank, copy and paste the entirety of the following contents:

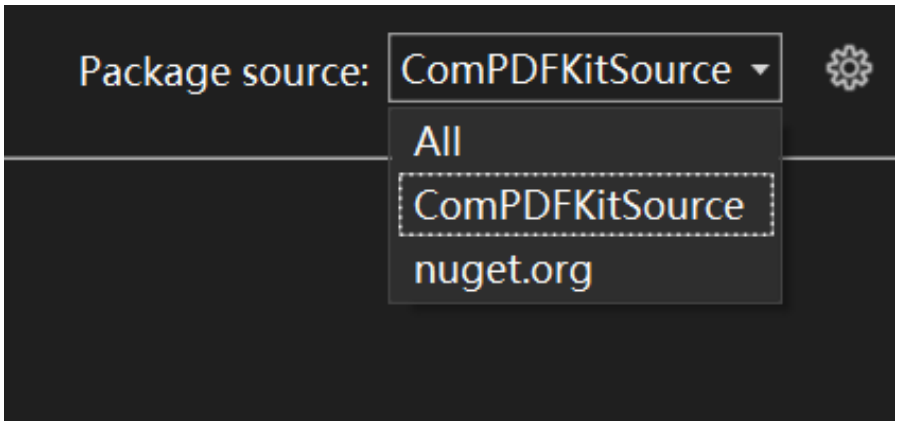
```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="ComPDFKitSource" value="path\to\directoryContainingNupkg" />
  </packageSources>
</configuration>
```

Edit the `value` of the contents to correctly refer to the location of the directory containing the ***"ComPDFKit.NetFramework....nupkg"*** package — for example, `C:\Users\me\nugetPackages\`. Now save the file, and close and reopen your solution for Visual Studio to force a read of the NuGet configuration.

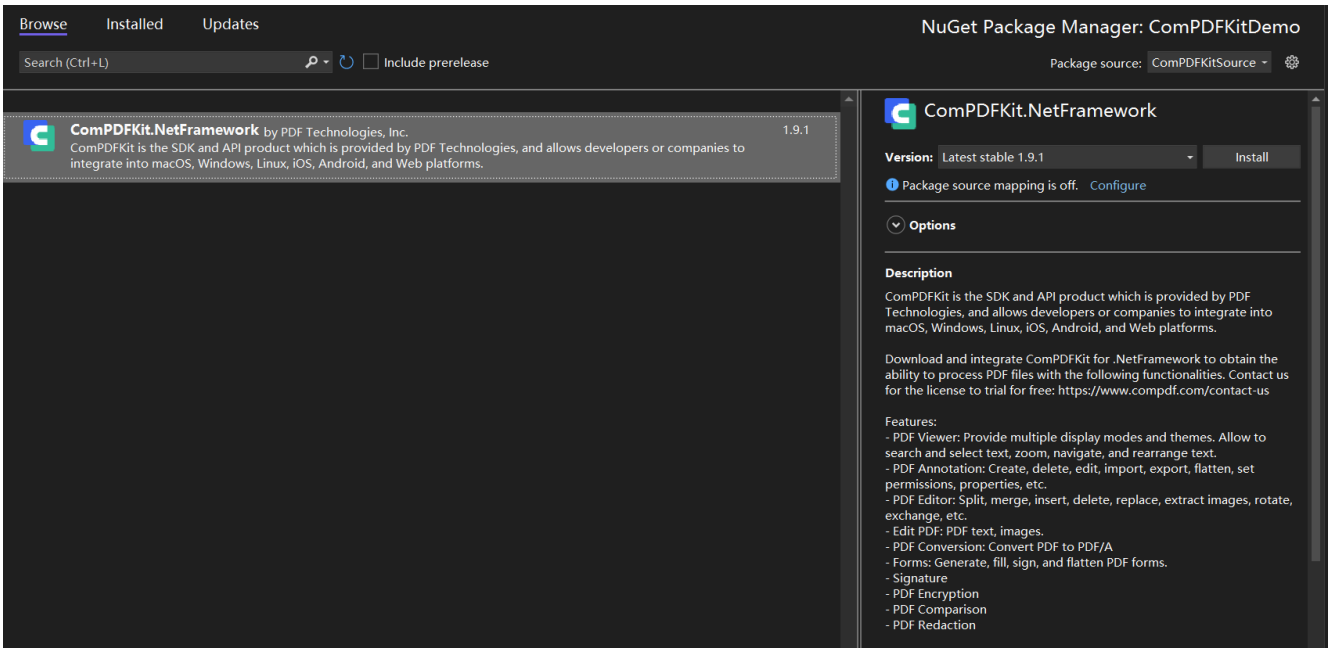
3. Open your project's solution, and in the Solution Explorer, right-click on **References** and click on the menu item **Manage NuGet Packages...**. This will open the NuGet Package Manager for your solution.



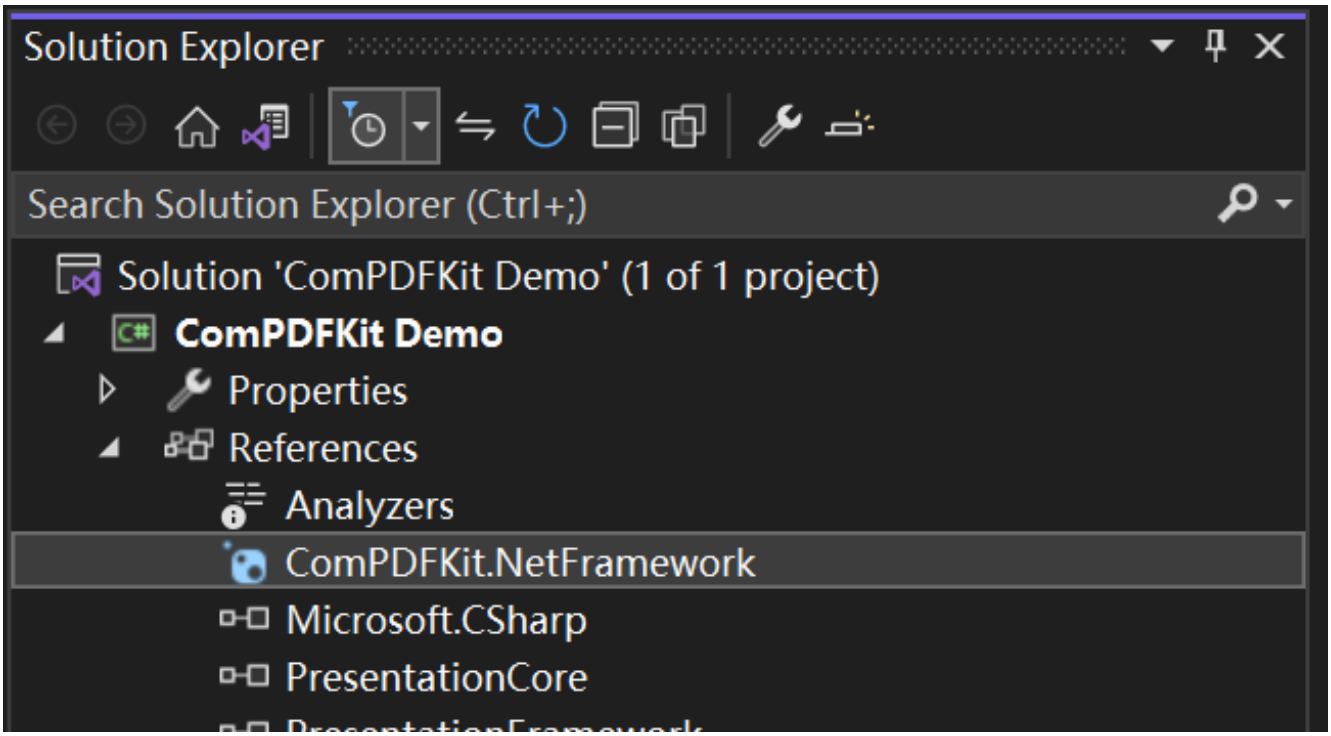

4. On the right-hand side of the manager in the Package source dropdown window, choose the entry `ComPDFKitSource` (or whatever you decided to name it). You should then see the entry for ***"ComPDFKit.NetFramework"***.




5. On the right side, in the panel describing the package, click on the **Install** button to install the package.

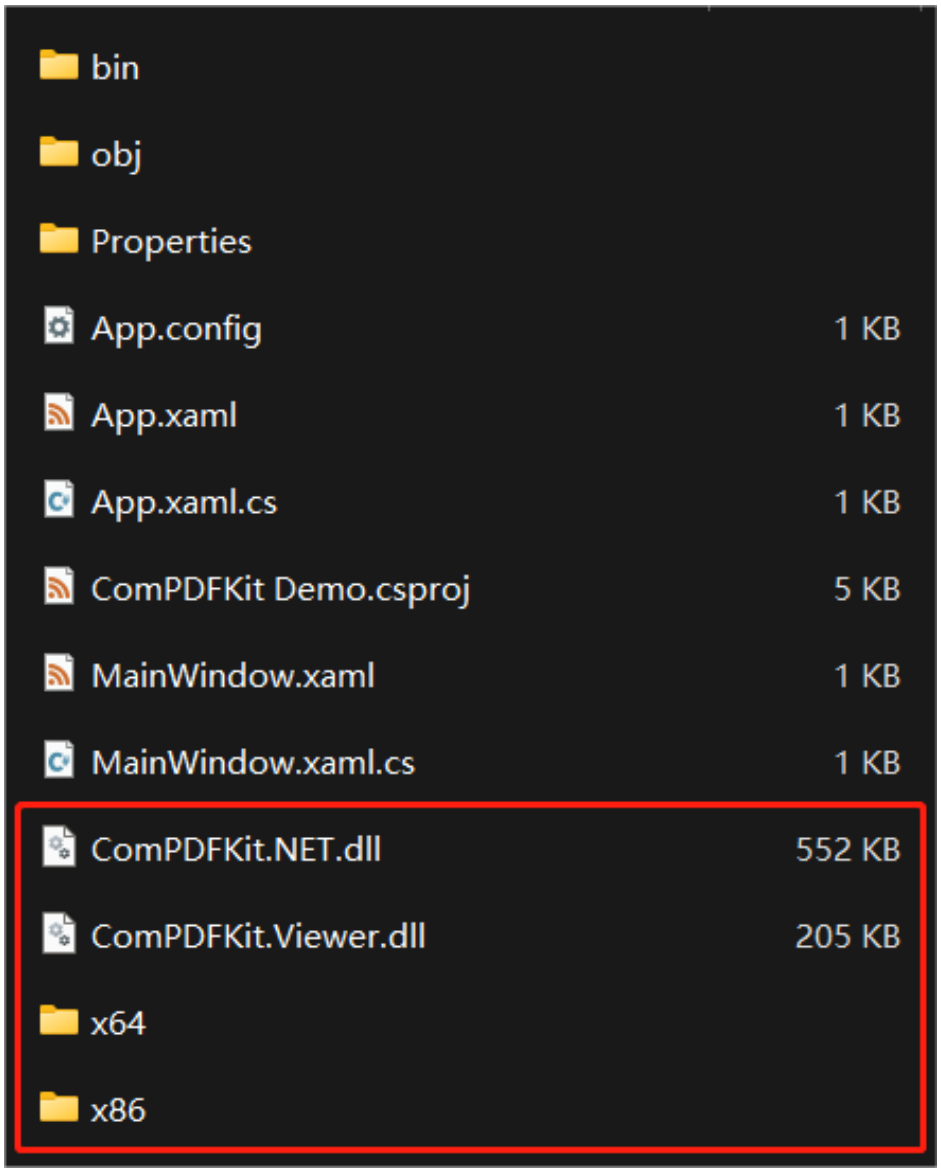



6. Once that's complete, you'll see a reference to the package in the Solution Explorer under **References**.

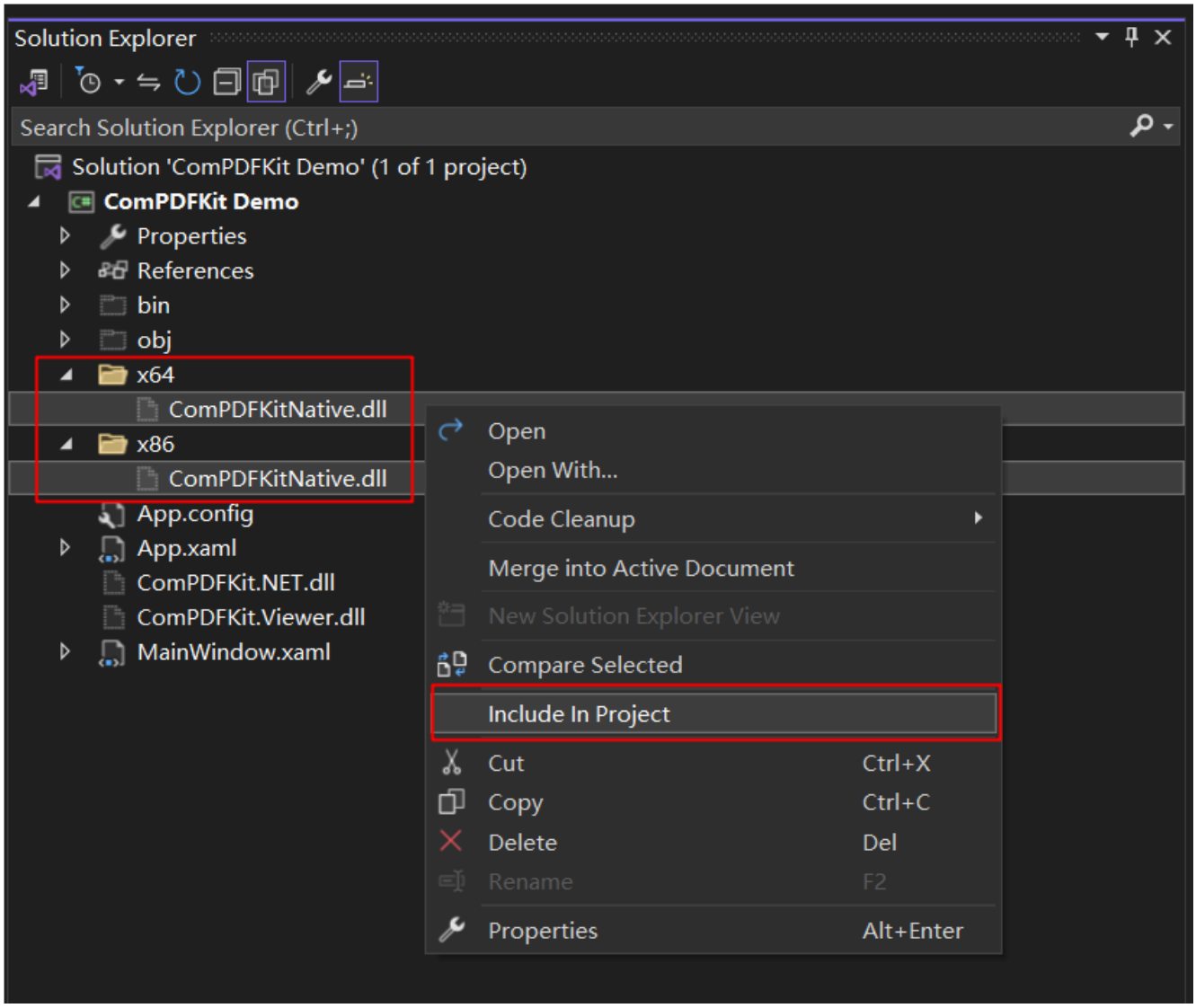



Alternatively, you can manually integrate ComPDFKit's dynamic libraries into your project.

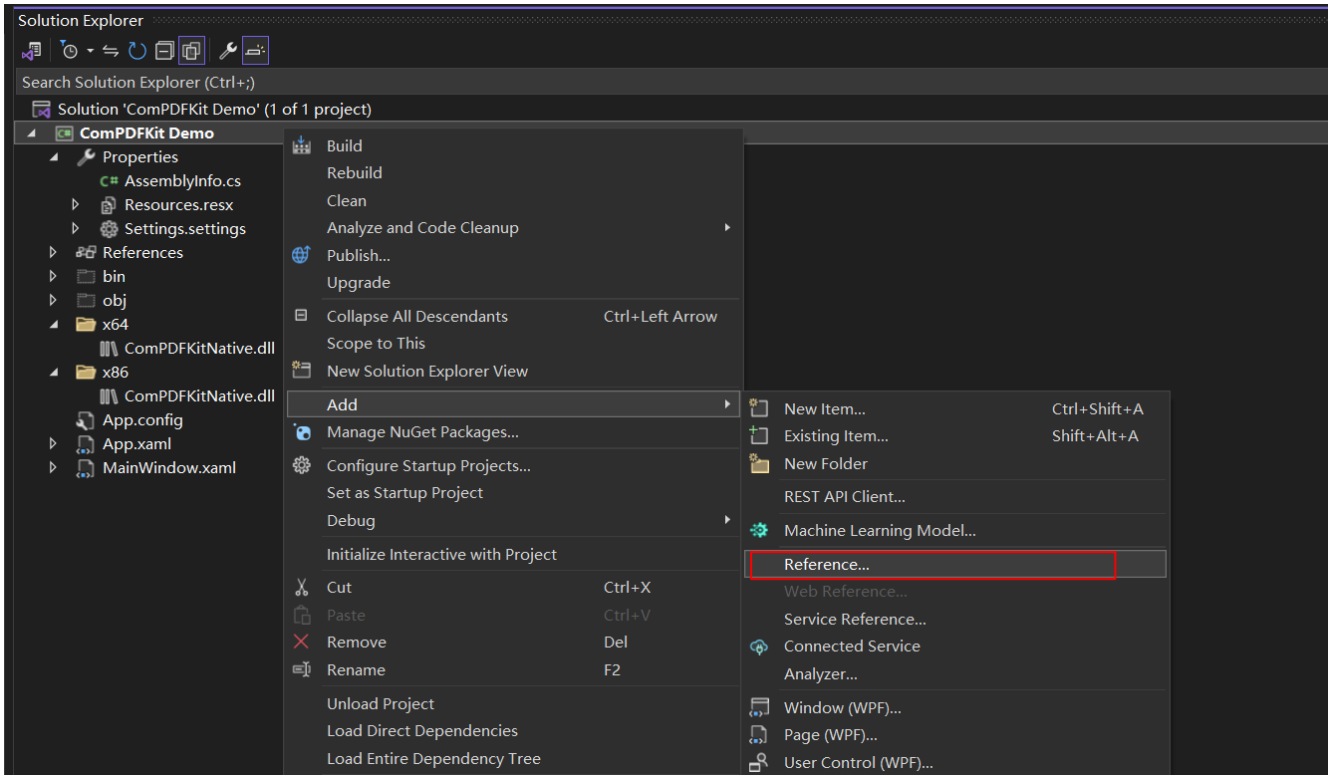
1. Open the ComPDFKit SDK package that you have extracted, and go to the ***"lib"*** folder. Copy the ***"x64"*** folder, ***"x86"*** folder, ***"ComPDFKit.NET.dll"*** file, and ***"ComPDFKit.Viewer.dll"*** file to the folder with the same name as your project that you created in  2.4.1. In this project, the folder is named "ComPDFKit Demo". Now, your folder should look like this:



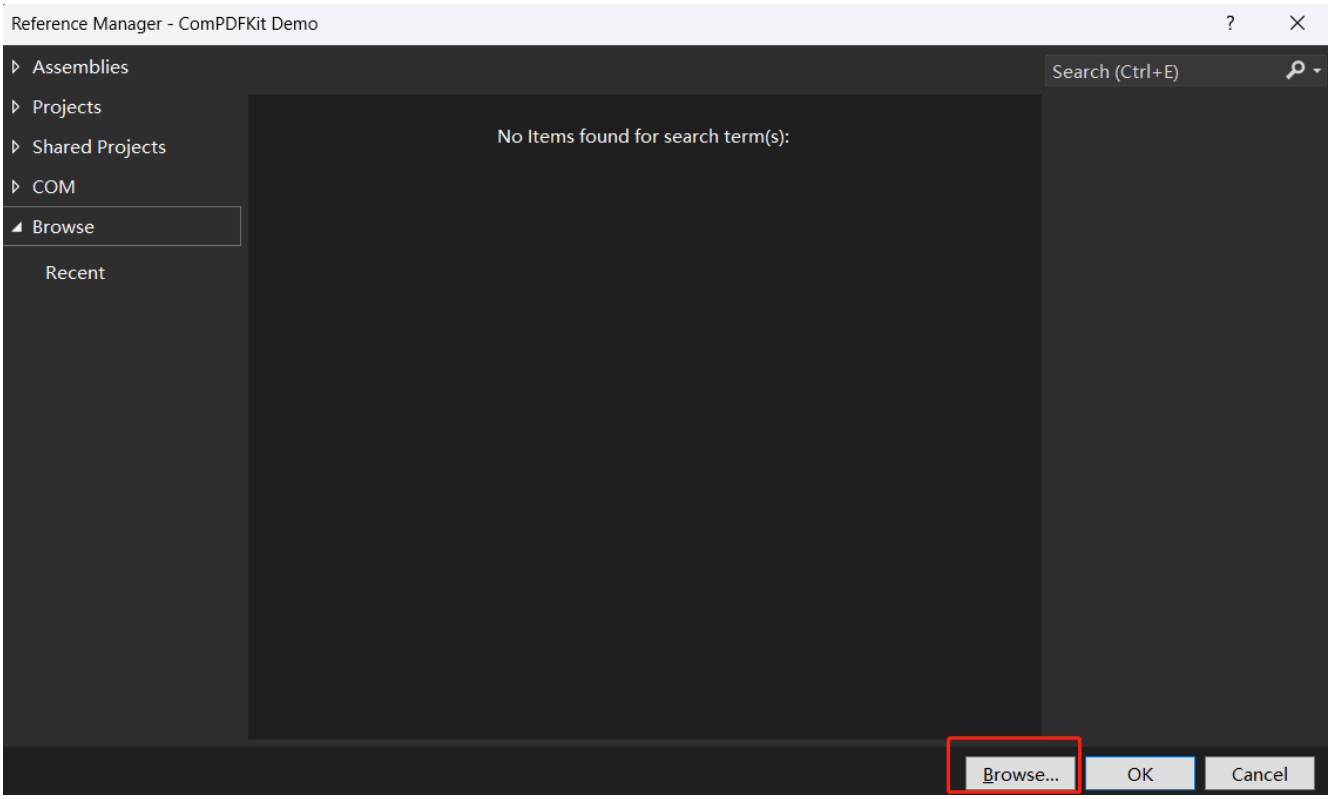

2. Then click the button **Show All Files** in the **Solution Explorer** menu. Find and right click the files you added before, and choose **Include In Project**.

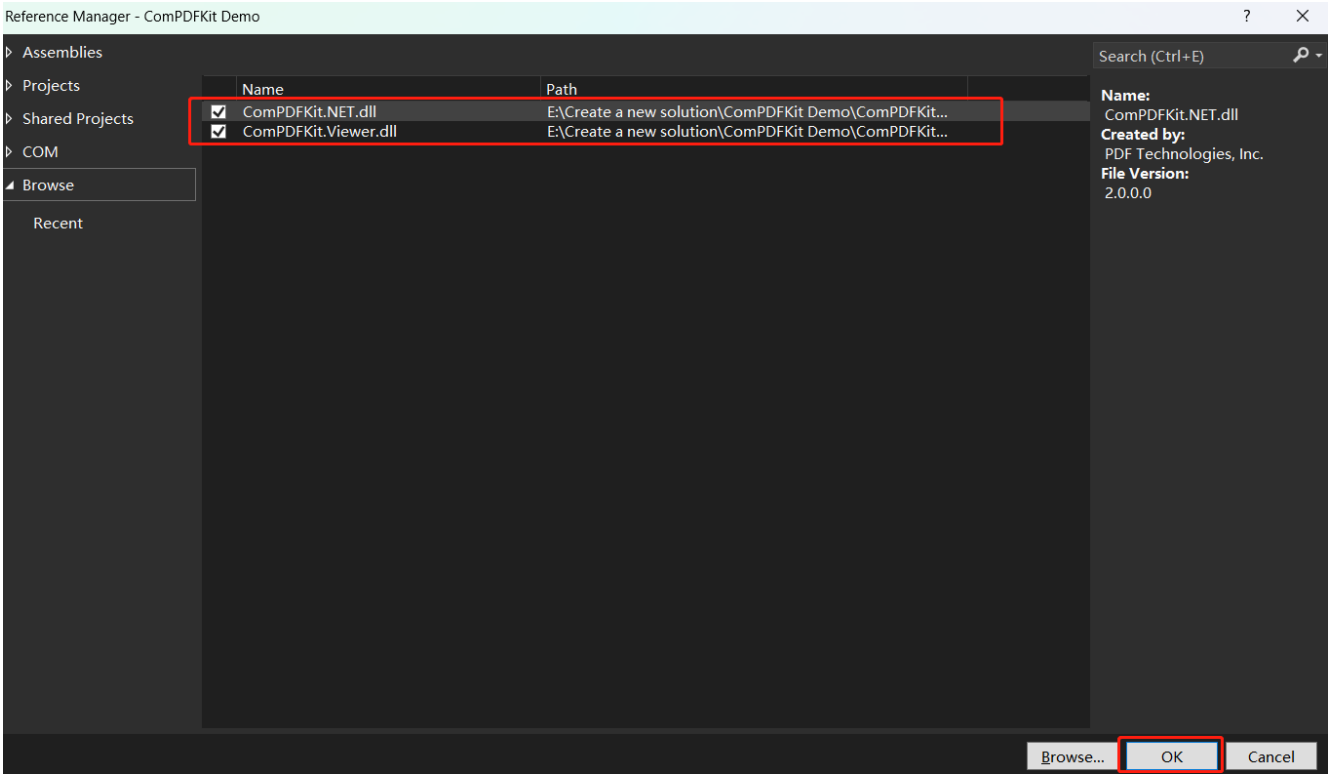



3. To use the APIs of ComPDFKit PDF SDK in your project, follow the instructions and add them to **Reference**. Right click the project that you need to add ComPDFKit PDF SDK and click **Add**. Then, click **Reference**.

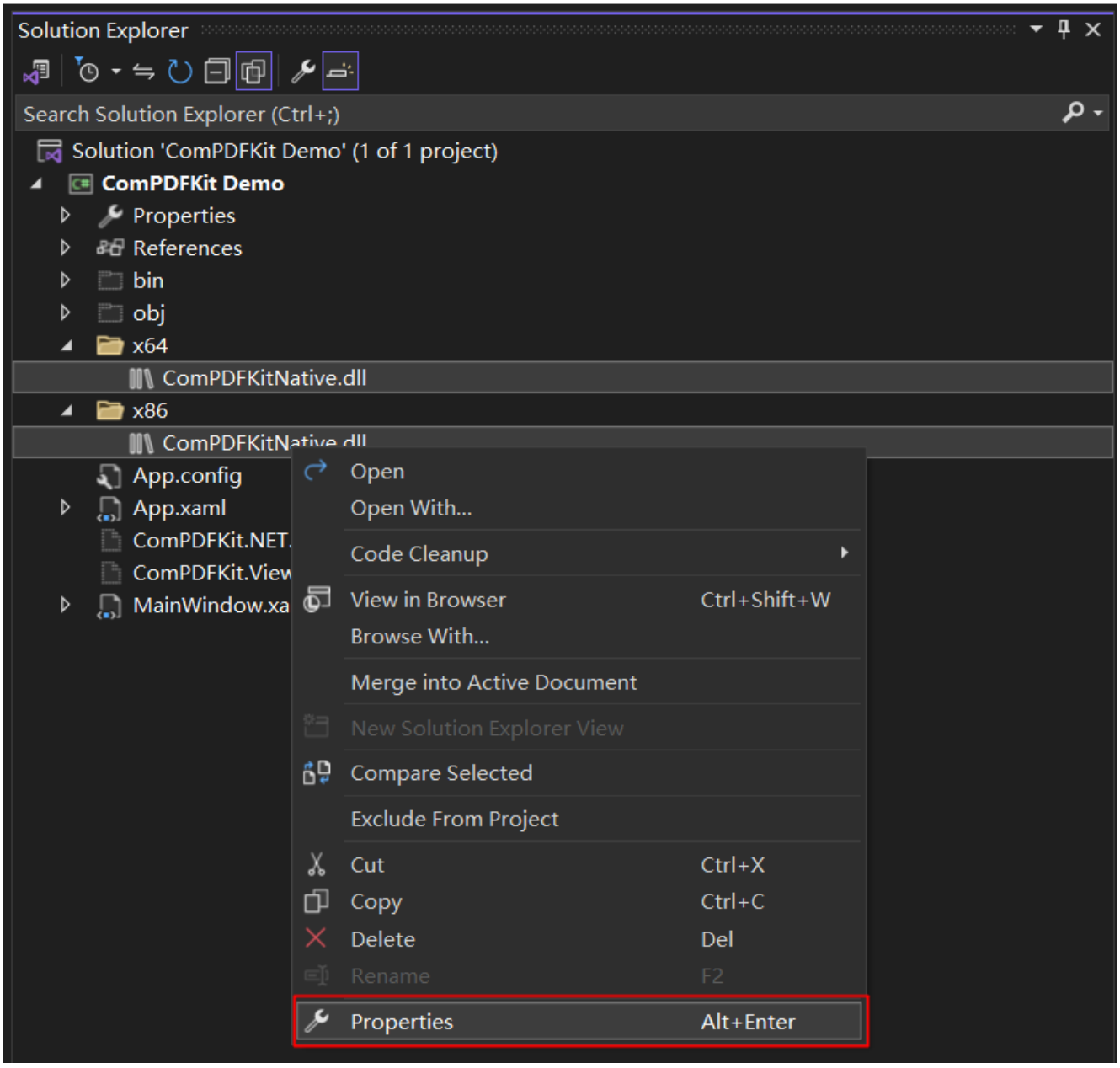



In the Reference dialog, choose **Browse**, click another **Browse** button in the bottom right corner, and navigate to the ***"ComPDFKit Demo"*** folder which is in your project. Select ***"ComPDFKit.NET.dll"*** and ***"ComPDFKit.Viewer.dll"*** dynamic library. Then, click **OK**.

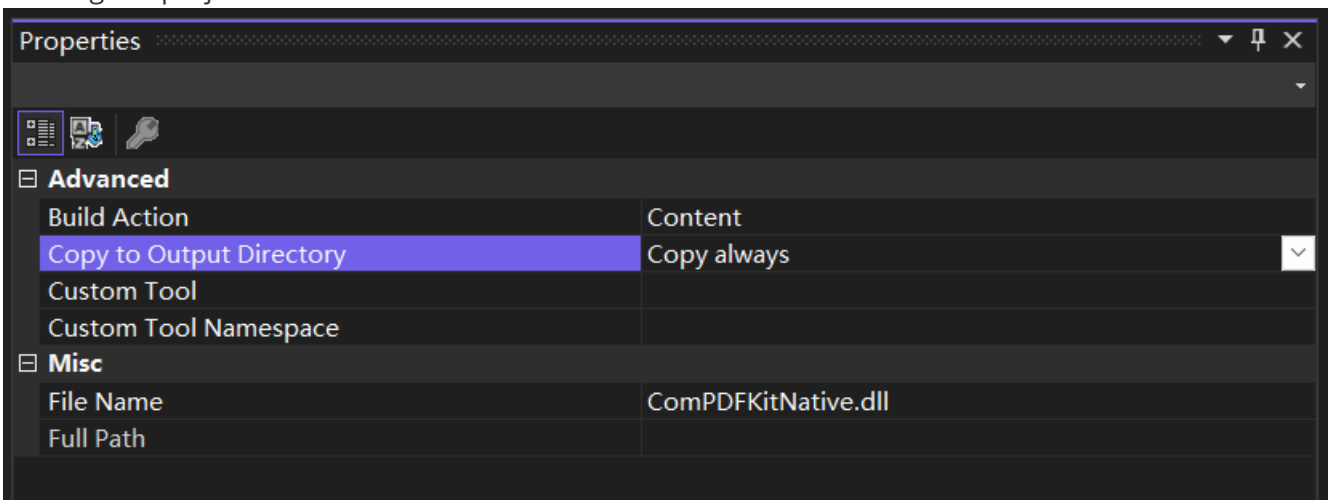





Right click ***"ComPDFKit.dll"*** -> click **Properties**.




Please make sure to set the property **Copy to Output Directory** of ***"ComPDFKitNative.dll"*** to **Copy if newer**. Otherwise, you should copy it to the same folder with the executable file manually before running the project.



## 2.5.3 Apply the License Key

You can contact the ComPDFKit team to obtain a trial license. Before using any classes from the ComPDFKit PDF SDK, you need to choose the corresponding scheme from the following two options based on the license type and apply the license to your application.

**Online Authentication**

You can perform online authentication using the following approach:

```
public static bool LicenseVerify()
{
  if (!CPDFSDKVerifier.LoadNativeLibrary())
  {
    return false;
  }
  LicenseErrorCode status = CPDFSDKVerifier.OnlineLicenseVerify("Input your license
here.");
  return status == LicenseErrorCode.E_LICENSE_SUCCESS;
}
```

Additionally, if you need to confirm the communication status with the server during online authentication, you can implement the `CPDFSDKVerifier.LicenseRefreshed` callback:

```
CPDFSDKVerifier.LicenseRefreshed += CPDFSDKVerifier_LicenseRefreshed;

private void CPDFSDKVerifier_LicenseRefreshed(object sender, ResponseModel e)
{
    if(e != null)
    {
        string message = string.Format("{0} {1}", e.Code, e.Message);
        Trace.WriteLine(message);
    }
    else
    {
        Trace.WriteLine("Network not connected.");
    }
}
```

**Offline Authentication**

The following is the `LicenseVerify()` method for implementing offline authentication:

```
bool LicenseVerify()
{
    if (!CPDFSDKVerifier.LoadNativeLibrary())
        return false;

    LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your license
here.");
    return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
}
```

## 2.5.4 Display a PDF Document

We have finished all prepare steps. Let's display a PDF file.

Add the following code to ***"MainWindow.xaml"*** and ***"MainWindow.xaml.cs"*** to display a PDF document. Please make sure to replace "ComPDFKit_Demo" with the name of your project. Now, all you need to do is to create a `CPDFViewer` object, and then display the `CPDFViewer` object in the Grid (component) named "PDFGrid" using the `OpenPDF_Click` method.

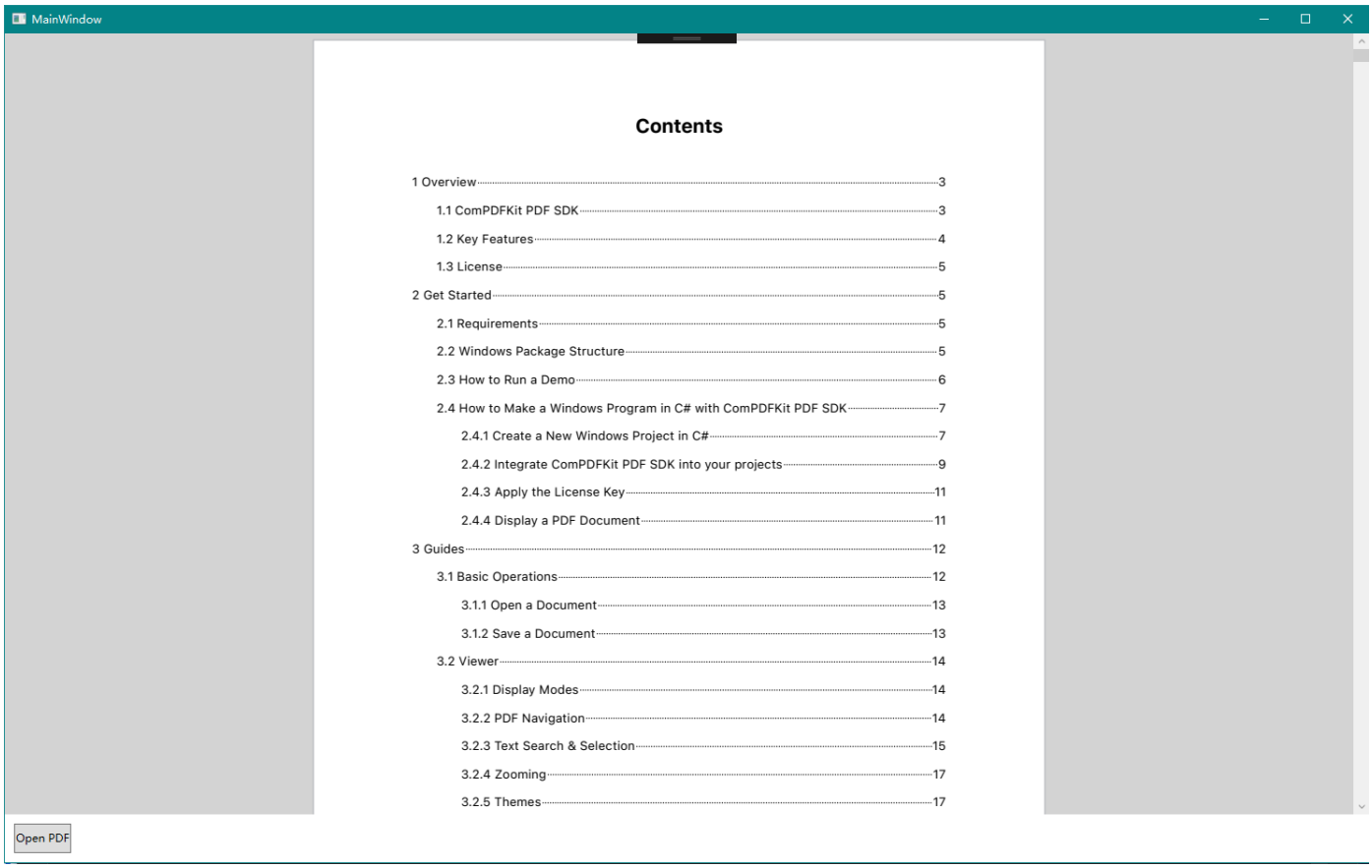
Now your ***"MainWindow.xaml"*** should look like the following code.

```
<Window x:Class="ComPDFKit_Demo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ComPDFKit_Demo"
        xmlns:compdfkitviewer="clr-namespace:ComPDFKitViewer;assembly=ComPDFKit.Viewer"
        mc:Ignorable="d"
        Focusable="True"
        Title="MainWindow" Height="600" Width="800" UseLayoutRounding="True">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="52"/>
        </Grid.RowDefinitions>
        <Grid Name="PDFGrid" Grid.Row="0">
            <ScrollViewer Focusable="False" CanContentScroll="True"
HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">
                <compdfkitviewer:CPDFViewer x:Name="PDFViewer"/>
            </ScrollViewer>
        </Grid>
        <Button Content="Open PDF" Grid.Row="1" HorizontalAlignment="Left" Margin="10"
Click="OpenPDF_Click"/>
    </Grid>
</Window>
```

Now your ***"MainWindow.xaml.cs"*** should look like the following code. Please note: You need to enter your license key. All the places that need to be modified in the code have been marked with comments in the code below. You just need to replace the string content below the comments by yourself.

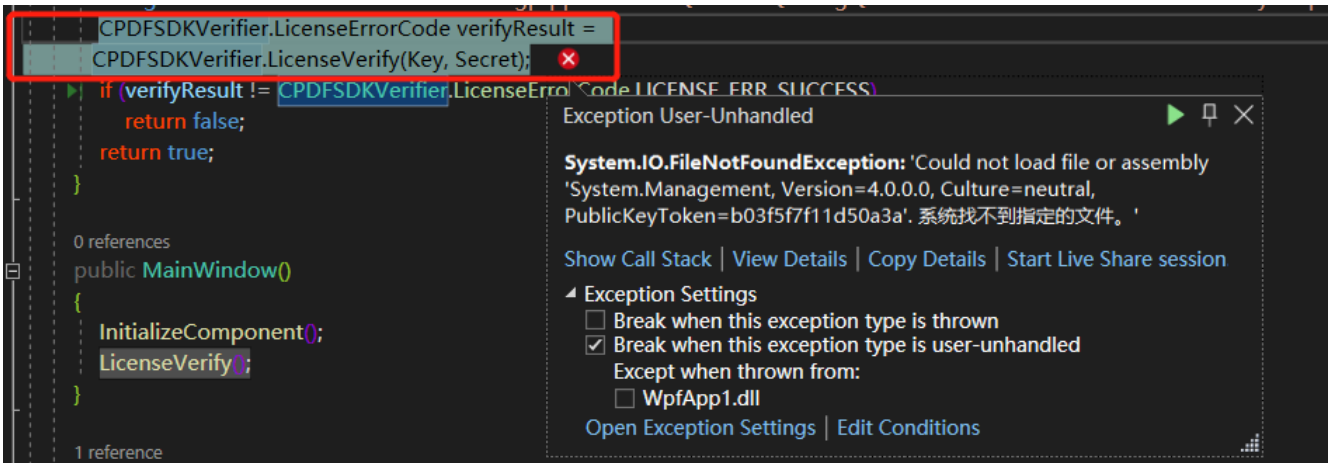
```
using ComPDFKit.NativeMethod;
using ComPDFKit.PDFDocument;
using Microsoft.Win32;
using System.Windows;

namespace ComPDFKit_Demo
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            LicenseVerify();
        }

        bool LicenseVerify()
        {
            if (!CPDFSDKVerifier.LoadNativeLibrary())
                return false;

            // Input your license.
            LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your
license here.");
            return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
        }

        private void OpenPDF_Click(object sender, RoutedEventArgs e)
        {
            // Get the path of a PDF file.
            var dlg = new OpenFileDialog();
            dlg.Filter = "PDF Files (*.pdf)|*.pdf";
            if (dlg.ShowDialog() == true)
            {
                // Use the PDF file path to open the document in CPDFViewer.
                CPDFDocument doc = CPDFDocument.InitWithFilePath(dlg.FileName);
                if (doc != null && doc.ErrorType ==
CPDFDocumentError.CPDFDocumentErrorSuccess)
                    PDFViewer.InitDoc(doc);
            }
        }
    }
}
```

Now run the project and you will see the PDF file that you want to display.  The PDF Viewer has been created successfully.

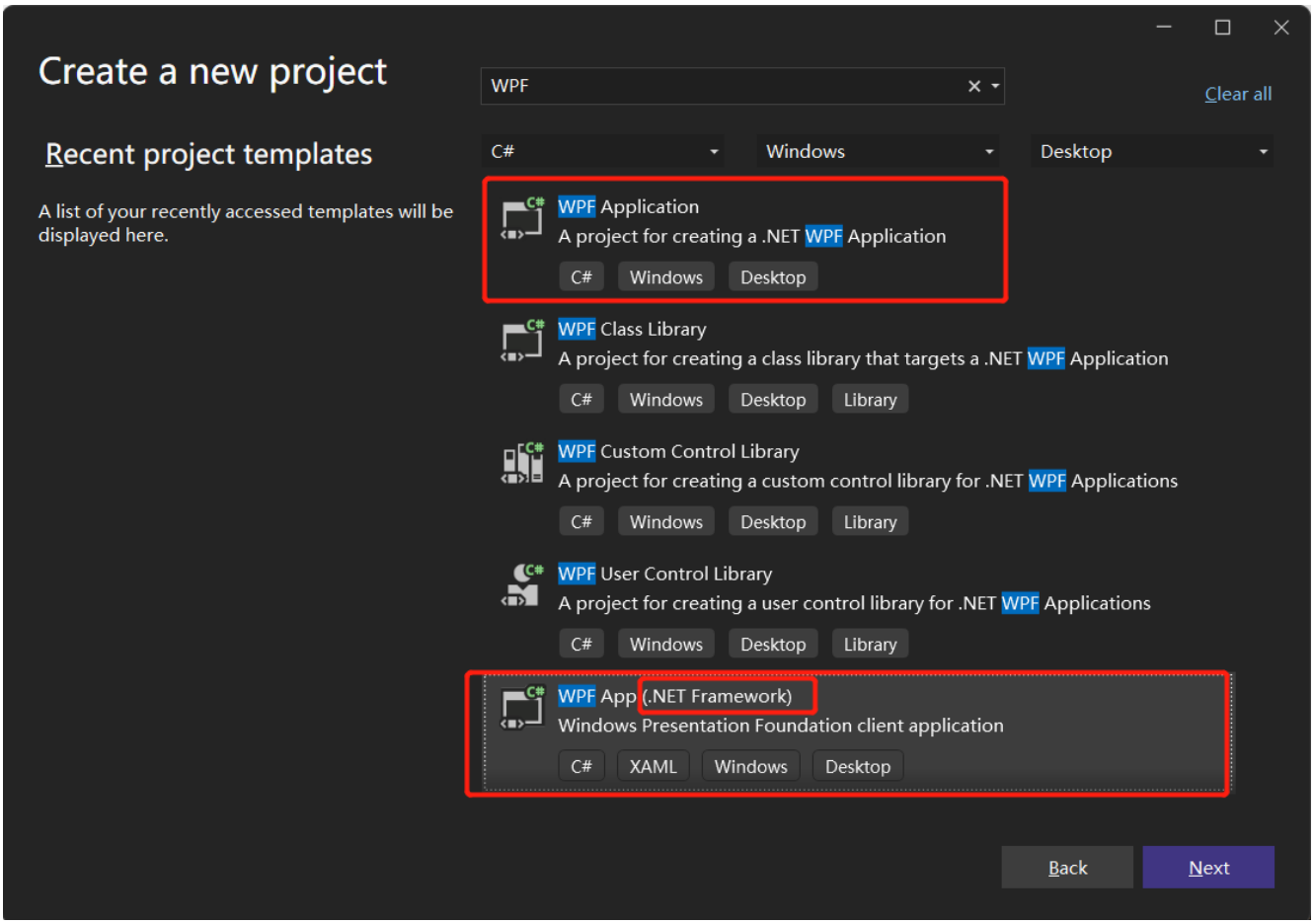



## 2.5.5 Troubleshooting

1. If "System.IO.FileNotFoundException" occurred in the `LicenseVerify()` function like this:




Check your WPF project and ensure that you chose **WPF App(.NET Framework)** instead of **WPF Application** when creating the project.




2. Other Problems

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

## 2.6 UI Customization

The folder of ***"ComPDFKit.Controls"*** includes the UI components to help conveniently integrate ComPDFKit PDF SDK. We have also built six standalone function programs, namely **Viewer**, **Annotations**, **ContentEditor**, **Forms**, **DocsEditor**, and **Digital Signature**, 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.Controls"*** 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.

## 2.6.1 Overview of *"ComPDFKit.Controls"* Folder

There are seven modules in ***"ComPDFKit.Controls"***: ***"Common"***, ***"Viewer"***, ***"Annotations"***, ***"ContentEditor"***, ***"Forms"***, ***"DocsEditor"***, and ***"Digital Signatures"***. Each of them includes the code and UI components like the following table to process PDFs.

| Folder | SubFolder | Description |
| --- | --- | --- |
| Common | BaseControl | Basic components used to compose other components, such as the value component to control the value of opacity, font size, border width, etc. |
| | Convert | Data converter |
| | Helper | Static classes and static methods that provide assistance for common functions, such as a static method that can invoke a file open dialog and get the selected PDF file path: GetFilePathOrEmpty. |
| | PasswordControl | Include the UI components and interaction of typing file passwords. |
| | PropertyControl | Include the UI components and interaction of handling specified data type inputting. |
| PDFView | PDFBookmark | Include the UI components and interaction of editing bookmarks and jumping pages. |
| | PDFInfo | Include the UI components and interaction of document information. |
| | PDFDisplaySettings | Include the UI components and interaction of PDF viewing like setting themes, display modes, etc. |
| | 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. |
| | PDFThumbnail | Include the UI component and interaction of PDF thumbnails. |
| Annotations | PDFAnnotationBar | Include the toolbar that indicates the required annotation type and order. Clicking on the navigation bar will pass the corresponding comment type enumeration through an event. |
| | PDFAnnotationPanel | When creating or modifying annotations, specific property panels are displayed, and controls for |

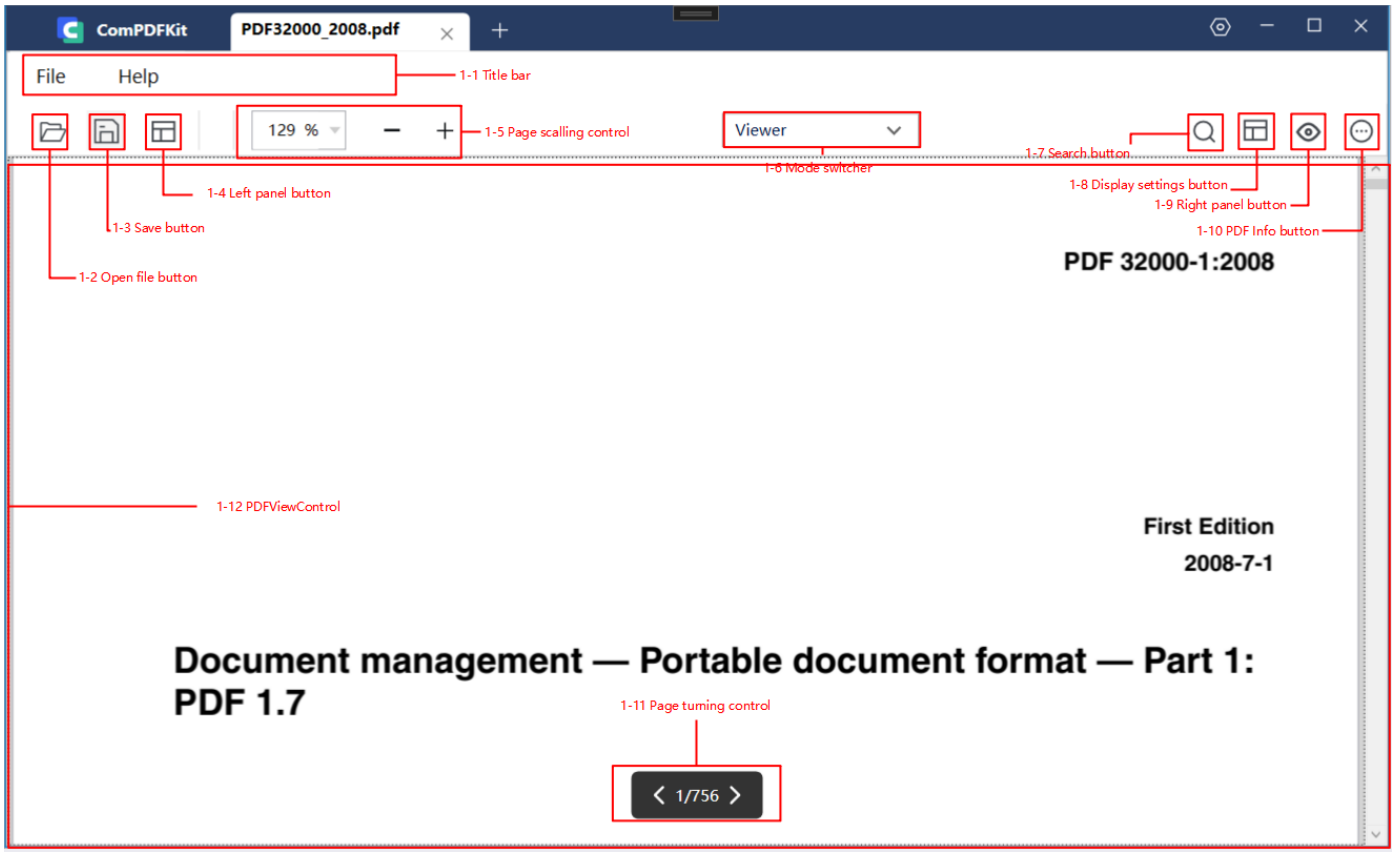
| | | |
|---|---|---|
| | | handling data are provided. |
| | PDFAnnotationList | Include the UI component and interaction of displaying all annotations in a list, selecting and deleting specific annotations/all annotations. |
| ContentEditor | PDFImageEditControl | Include the toolbar to edit PDF images and undo/redo the processing of editing PDF images. |
| | PDFTextEditControl | Include the toolbar to edit PDF text and undo/redo the processing of editing PDF text. |
| Forms | FormBarControl | Include the UI component and interaction of specifying needed form fields and the order of displaying the form field types in UI. |
| | FormPropertyControl | Include the property panel and interaction to set the properties of forms. |
| Docs Editor | PDFPageEditBar | Include the toolbar for creating, replacing, rotating, extracting, and deleting PDF pages |
| | PDFPageEdit | Include the UI component and interaction of document editing like thumbnails, drag, right-click menu, etc. |
| | PDFPageExtract | Include the popup window of page extraction. It only processes and transfers the data. You can refer to PDFPageEdit for inserting pages. |
| | PDFPageInsert | Include the popup window of page insertion. It only processes and transfers the data. You can refer to PDFPageEdit for inserting pages. |
| Digital Signatures | AddCertificationDialog | Include the popup window to create new certificates or using existing certificates. |
| | CPDFSignatureListControl | Include the UI component and interaction of displaying the list of digital signatures and their status, with options to navigate to a specific signature location or open a signature status popup. |
| | VerifyDigitalSignatureControl | Include the popup window to display the signature status. |
| | SignatureStatusBarControl | Include the popup window to display all the signature statuses in this file. |
| | FillDigitalSignatureDialog | Include the popup window to create the signature appearance. |
| | | |

| | | |
|---|---|---|
| | ViewCertificateDialog | Include the popup window to view the signature certificates. |

## 2.6.2 UI Component

This section mainly introduces the connection between the UI components and API configuration of ***"ComPDFKit.Controls"***, 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.

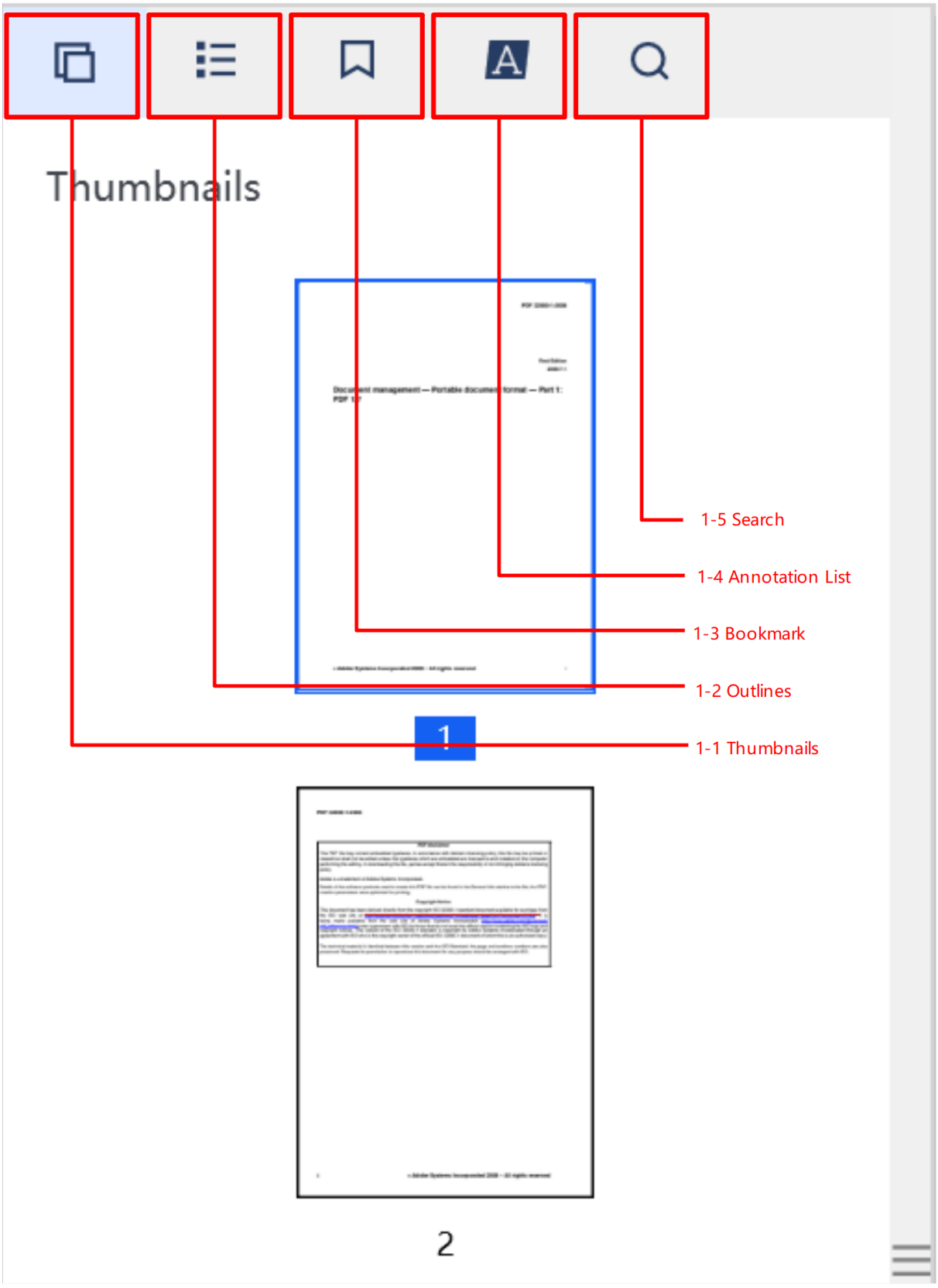
**Part 1:**




The picture above shows the main UI components associated with the API of Viewer, which are also the fundamental UI components of ***"ComPDFKit.Controls"***. The following table shows the details of the connection between UI components and APIs.

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Title bar | CPDFTitleBarControl | The toolbar at the top of the PDF view window: Include the help center and file center. |
| 1-2 | Open file button | / | Control to switch a new document. |
| 1-3 | Save button | / | Control to save the current file. |
| 1-4 | Right panel button | CPDFBOTABarControl | Control the display status of the property panel. |
| 1-5 | Page scalling control | CPDFScalingControl | Control to change the zoom ratio of PDF. |
| 1-6 | Mode switcher | / | Switch the feature modules. |
| 1-7 | Search button | CPDFBOTABarControl | Enter the searching mode. |
| 1-8 | Display settings button | CPDFDisplaySettingsControl | Control to show or hide the setting panel. |
| 1-9 | Left panel button | CPDFAnnotationControl、FromPropertyControl、PDFContentEditControl | Control the displaying status of property panel. |
| 1-10 | PDF info button | CPDFInfoControl | Control the popup window of document information. |
| 1-11 | Page turning control | PageNumberControl | Control to jump to other specific pages quickly. |
| 1-12 | PDFViewControl | PDFViewControl | Basic interactions like zooming PDF view with mouse and executing page jumping or push button actions. |

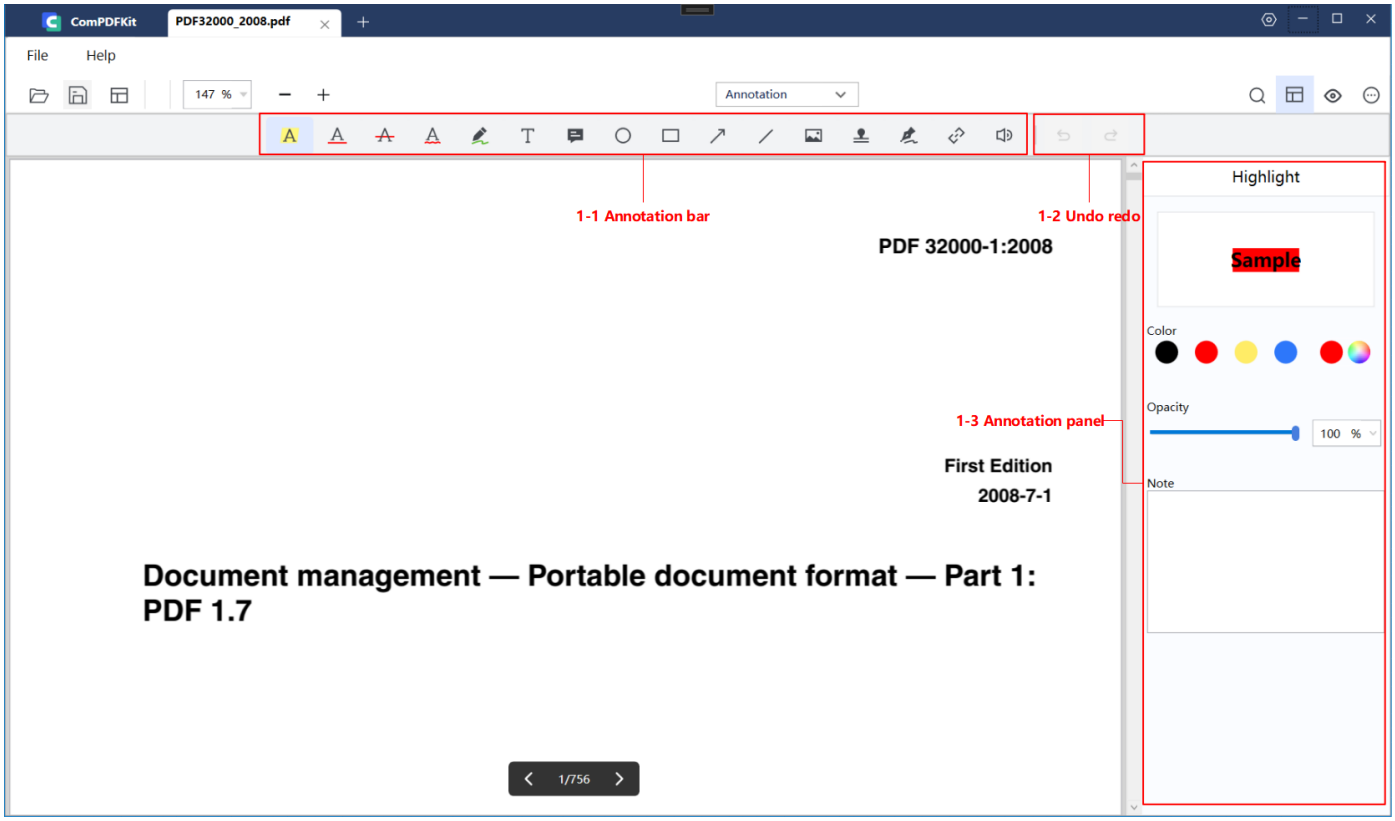
**Part 2:**




The picture above shows the UI components associated with the API of outline, bookmark, thumbnail, annotation list, and searching. The following table shows the details of the connection between UI components and APIs.

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Thumbnails | CPDFThumbnailControl | Enter the thumbnails of PDFs. |
| 1-2 | Outlines | CPDFOutlineControl | Enter the outlines of PDFs. |
| 1-3 | Bookmark | CPDFBookmarkControl | Enter the bookmark list of PDFs. |
| 1-4 | Annotation List | CPDFThumbnailControl | Enter the annotation list of PDFs. |
| 1-5 | Search | CPDFSearchControl | Enter the PDF keywords searching. |

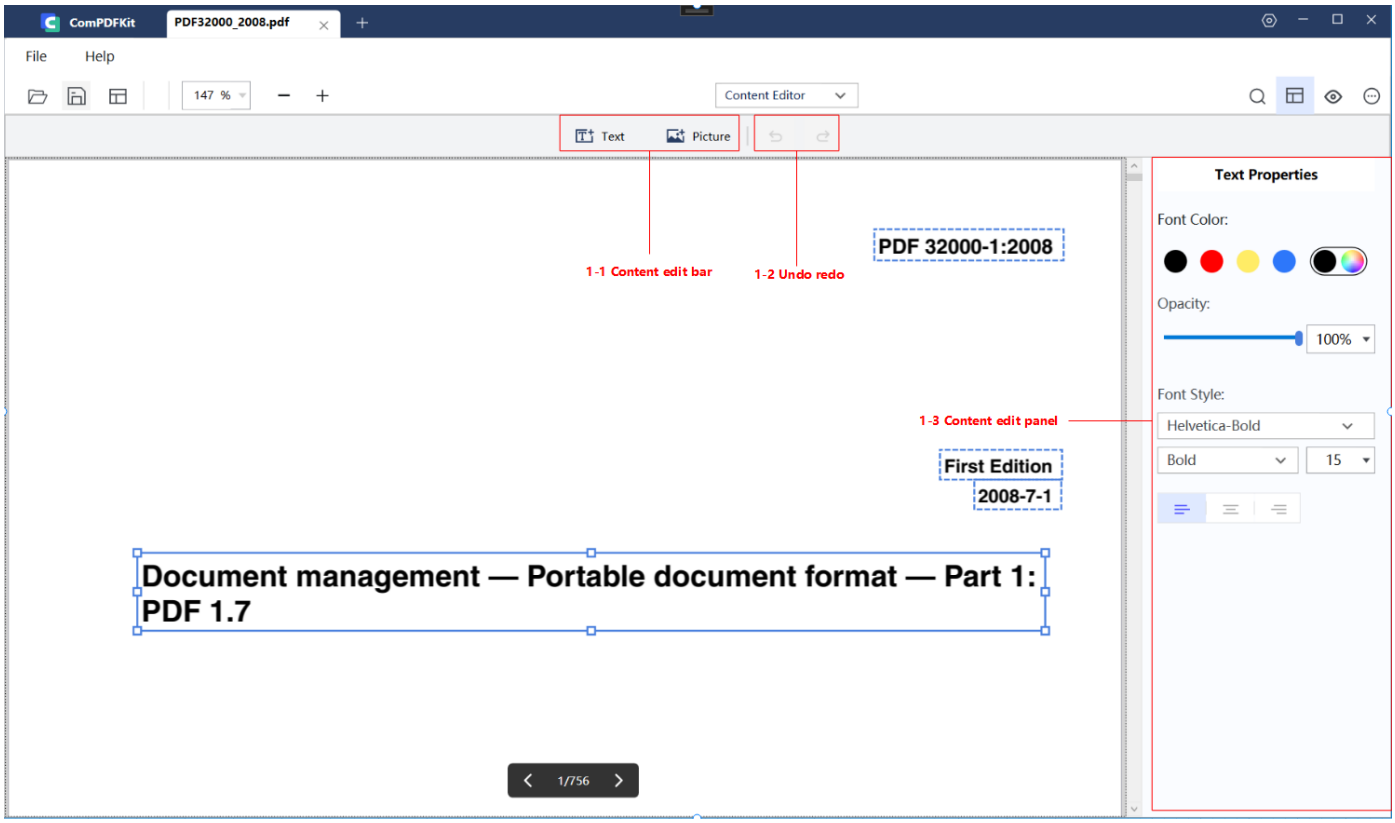
**Part 3:**




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

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Annotation bar | CPDFAnnotationBarControl | Annotated toolbar, allowing specifying the annotation types and the order of displaying the annotation types in UI. |
| 1-2 | Undo redo | / | Undo/redo the processing of annotations. |
| 1-3 | Annotation panel | CPDFAnnotationControl | Preset annotation properties for creating annotation. |

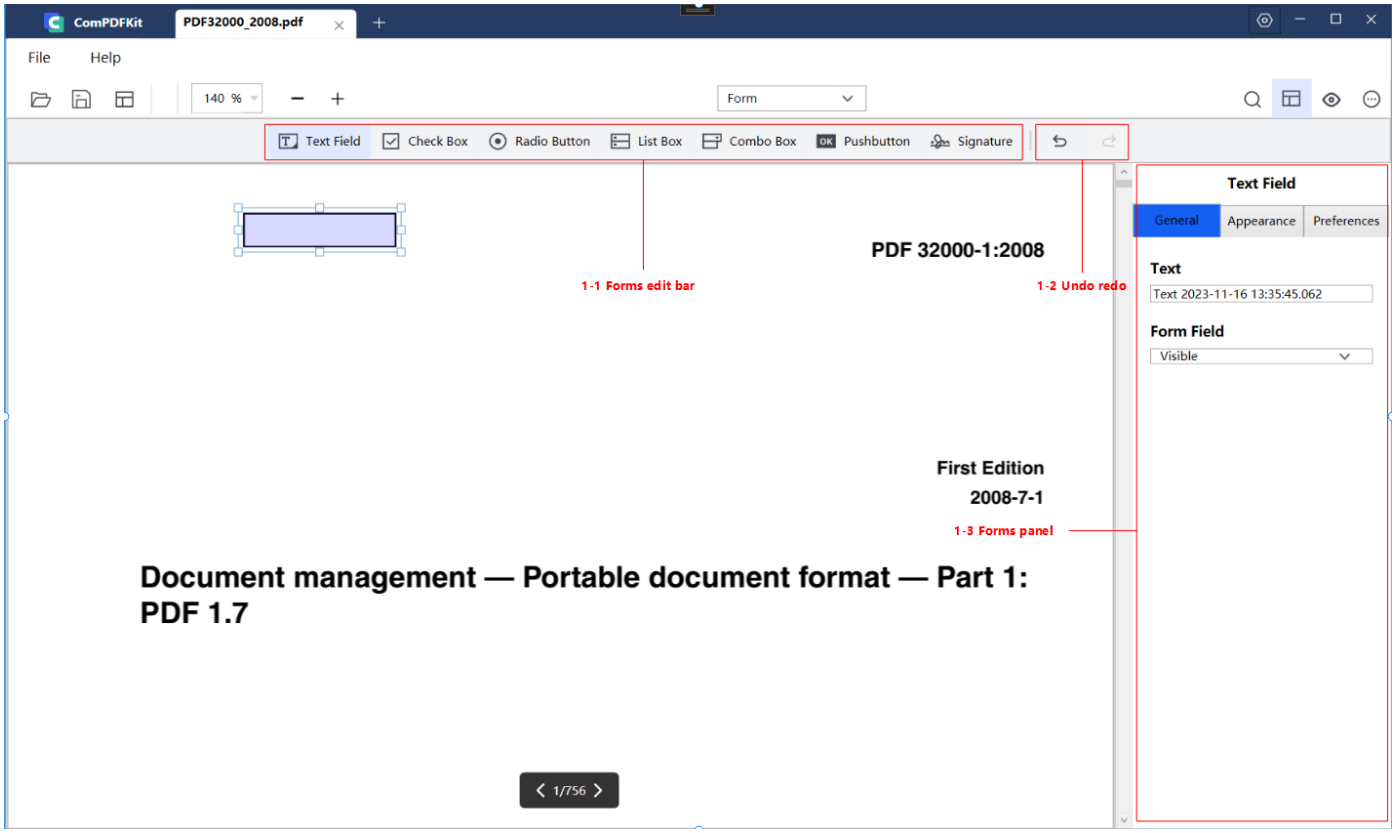
**Part 4:**




The picture above shows the UI components associated with the API of the content editor. The following table shows the details of the connection between UI components and APIs.

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Content edit bar | / | The tool bar of content editor. Enter a state of creating text and only text editing, after clicking on text editing. After clicking on image editing, you can add images. After adding the images, you will enter the default mode, in which both images and text can be edited. |
| 1-2 | Undo redo | / | Undo redo the processing of editing PDF text/images. |
| 1-3 | Content edit panel | PDFContentEditControl | Preset text properties for adding text. After clicking on text or image, you will get the attributes of the currently selected object and can modify them. |

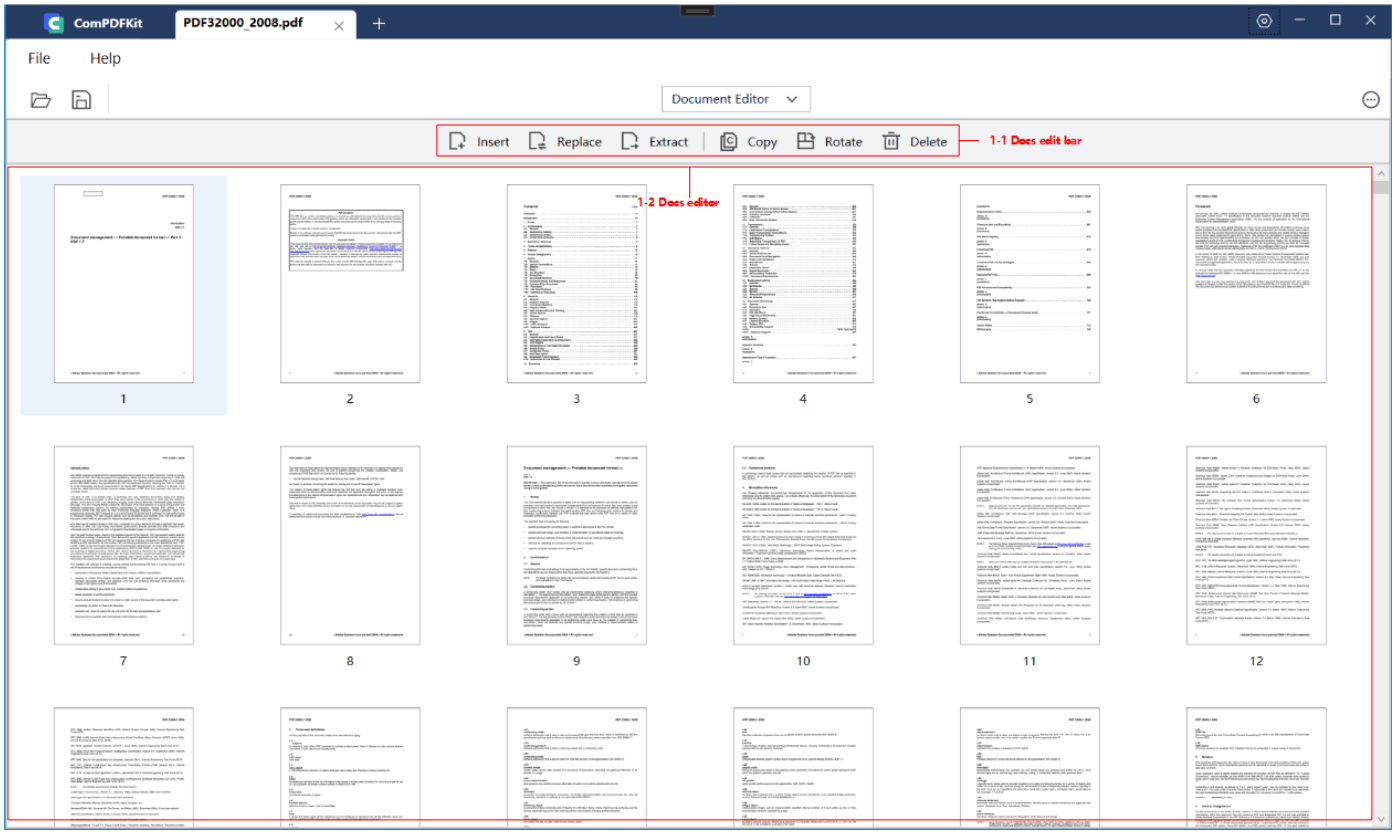
**Part 5:**




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

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Forms edit bar | CPDFFormBarControl | Tool bar of PDF forms. |
| 1-2 | Undo redo | / | Undo/redo the processing of forms. |
| 1-3 | Forms panel | FromPropertyControl | Set the properties of forms. |

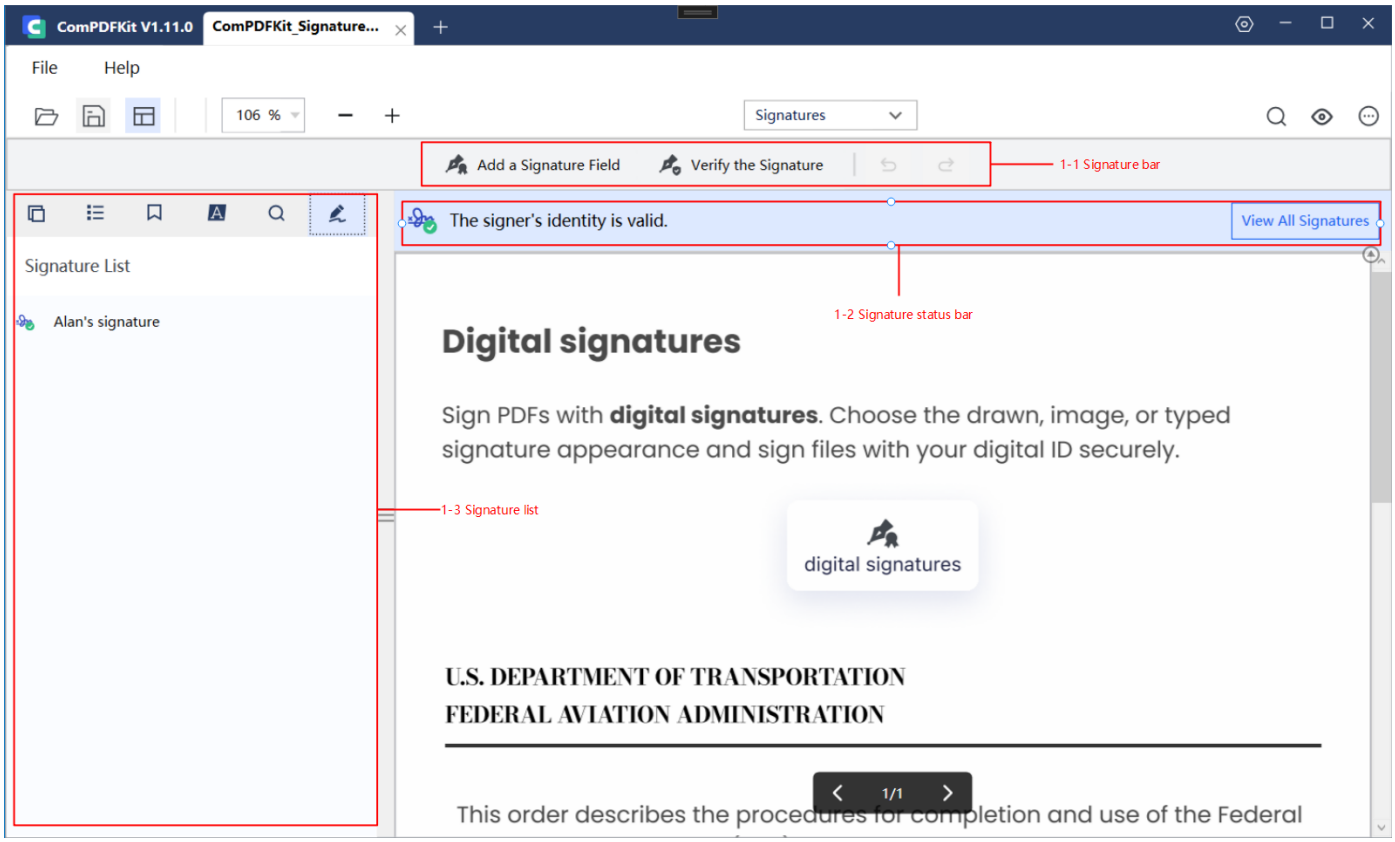
**Part 6:**




The picture above shows the UI components associated with the API of the document editor. The following table shows the details of the connection between UI components and APIs.

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Docs edit bar | CPDFPageEditBarControl、CPDFPageExtractWindow、CPDFPageInsertWindow | Tool bar of document editor. |
| 1-2 | Docs editor | CPDFPageEditControl | Show the thumbnails of PDF pages and interaction for editing PDF pages. |

**Part 7:**




The picture above shows the UI components associated with the API of the digital signature. The following table shows the details of the connection between UI components and APIs.

| Number | Name | Functionality | Description |
| --- | --- | --- | --- |
| 1-1 | Signature bar | AddCertificationDialog、FillDigitalSignatureDialog | Add signature field, and verify all the signatures. |
| 1-2 | Signature status bar | SignatureStatusBarControl | Show the status of all the digital signatures. |
| 1-3 | Signature list | ViewCertificateDialog | A list to display all the digital signatures in PDFs. |

# 2.7 Samples

The Samples use preset parameters and documentation to call the API of ComPDFKit PDF SDK for each function without UI interaction or parameter settings. This is achieved through modular code examples. 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, and access the existing 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. |
| DocumentInfo | Display 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. |
| 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. |
| DocumentCompare | Compare different documents and show the result in new documents. |
| DigitalSignatures | Create, fill, and verify the signatures and certificates. Read the details of signatures and certificates. Remove digital signatures. |

# 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 Windows apps using our 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.

- **Font Management**

  Retrieve all fonts available on the local system. These fonts can be utilized for font settings across multiple modules.

## 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:

```
// Initialize a `CPDFDocument` object using the PDF file path.
CPDFDocument document = CPDFDocument.InitWithFilePath(myFilePath);

if(document.ErrorType != CPDFDocumentError.CPDFDocumentErrorSuccess
  && document.ErrorType != CPDFDocumentError.CPDFDocumentPasswordError)
{
    return;
}

// 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:

```
CPDFDocument document = CPDFDocument.CreateDocument();
```

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 |
| --- | --- |
| CPDFDocumentErrorSuccess | Document opened successfully. |
| 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 :

```
myCPDFDocument.WriteToLoadedPath();// Incrementally save the document object to the current path.

myCPDFDocument.WriteToFilePath(newFilePath);// Save the document object to the current path in a non-incremental manner.
```

## 3.1.4 Document Information

This example shows how to get document information:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
CPDFInfo info = document.GetInfo();
string title = info.Title;          // Document title.
string author = info.Author;        // Document author.
string subject = info.Subject;      // Document subject.
string creator = info.Creator;      // Application name that created the document
string producer = info.Producer;    // Application name that generated PDF data
string keywords = info.Keywords;    // Document keywords.
string creationDate = info.CreationDate;  // Document creation date
string modificationDate = info.ModificationDate;    // Document last modified date
```

## 3.1.5 Font Management

The ComPDFKit PDF SDK supports reading the font families and their styles available on your device, and setting them as fonts for various functionalities such as annotations, forms, watermarks, headers, footers, Bates numbering, and more. This will assist you in designing aesthetically pleasing PDF files or adjusting and enhancing your PDF files with fonts that comply with certain specifications.

When setting fonts using font management, you need to:

1. Retrieve the names of all font families in the system.
2. Select the font you need and obtain the style names for the font family.
3. After selecting the style name, obtain the PostScript name of the font based on the font family name and style name.
4. The PostScript name can then be used to set the font.

This example shows how to use font management:

```
int familyNameIndex = 0;
int styleNameIndex = 0;

// Get the list of font families and choose a font family
List<string> fontFamilyNames = CPDFFont.GetFontNameDictionary().Keys.ToList();
string fontFamilyName = fontFamilyNames[familyNameIndex];

// Get the list of font styles corresponding to the font family and choose a font style
List<string> fontStyleNames = CPDFFont.GetFontNameDictionary()[fontFamilyName];
string fontStyleName = fontStyleNames[styleNameIndex];

// Get the PostScript name based on the font family and font style
string postScriptName = string.Empty;
CPDFFont.GetPostScriptName(fontFamilyName, fontStyleName, ref postScriptName);

// Apply the PostScript name to the functionality that requires font setting.
// For specific attribute settings, refer to the documentation of the corresponding
functionality.
CPDFWatermark watermark = document.InitWatermark(C_Watermark_Type.WATERMARK_TYPE_TEXT);
watermark.SetFontName(postScriptName);
```

**About Font Family, Font  Style, and PostScript Name**

1. Font Family:

   Font Family refers to the group or series name of a font, typically representing a collection of fonts sharing a similar design style.

   For example, the Helvetica font family comprises various styles of fonts such as Helvetica Regular, Helvetica Bold, Helvetica Italic, etc., all belonging 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.

3. PostScript Name:

   The PostScript name is the unique identifier for a font. It is typically a distinct string used to specify a unique combination of font family and style.

   It serves as a standardized name for fonts, allowing them to be accurately referenced across different systems and platforms.

# 3.2 Viewer

## 3.2.1 Overview

ComPDFKit for Windows 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 Windows 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 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 and split-view 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.

- **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 content in the viewer, including text and images.

## 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 and split-view mode.

**Set the Display Mode**

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

```
myCPDFViewer.SetViewMode(ViewMode.SingleContinuous);
```

**Explanation of Display Mode Types**

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

| Type | Description | Parameter Name |
| --- | --- | --- |
| Single | Display one page at a time, flipping to the next when scrolling to the page edge. | ViewMode.Single |
| Single Continuous | Display one page at a time, with pages arranged continuously in the vertical direction. | ViewMode.SingleContinuous |
| Double | Display two pages side-by-side at a time, flipping when scrolling to the page edge. | ViewMode.Double |
| Double Continuous | Display two pages side-by-side at a time, with pages arranged continuously in the vertical direction. | ViewMode.DoubleContinuous |
| Book | Cover occupies a row alone, with the rest consistent with the double-page mode. | ViewMode.Book |
| Book Continuous | Cover occupies a row alone, with the rest consistent with the double continuous mode. | ViewMode.BookContinuous |

**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:

```
myCPDFViewer.SetIsCrop(true);
```

## 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.
myCPDFViewer.GoToPage(0);

// Navigate to a specific position on the first page.
myCPDFViewer.GoToPage(0, new Point(100, 100));
```

**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 top-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:

```
static private void TraverseOutline(List<CPDFOutline> outlineList)
{
  foreach (var outline in outlineList)
  {
        // Print indentation, add 4 spaces for each nested level.
    for (var i = 0; i < outlineCounter; i++)
    {
      Console.Write("    ");
    }
        // Print outline titles.
    Console.Write("-> " + outline.Title + "\n");
        // Increase outline numbering.
    outlineNumber++;
        // Retrieve the list of child outlines for the current outline.
    var childList = outline.ChildList;
    if (childList != null && childList.Count != 0)
    {
          // If there are child outlines, increase the indentation count.
      outlineCounter++;
          // Recursively traverse child outlines.
      TraverseOutline(childList);
    }
    else
```

```
    {
            // Current level traversal completed, return to the parent outline.
      if (outlineList.IndexOf(outline) + 1 == outlineList.Count)
      {
                // If the current outline is the last one at the current level, decrease
the indentation count.
        outlineCounter--;
      }
    }
  }
}
```

### 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:

```
// Locate the parent outline where the new outline needs to be added.
CPDFOutline outline = document.GetOutlineRoot();
// Create the new outline.
CPDFOutline childOutline = null;
outline.InsertChildAtIndex(document, 0, ref childOutline);
// Add outline action; in this case, it is to navigate to the first page.
CPDFGoToAction gotoAction = new CPDFGoToAction();
CPDFDestination dest = new CPDFDestination();
dest.PageIndex = 0;
gotoAction.SetDestination(document, dest);
childOutline.SetAction(gotoAction);
// Set properties
childOutline.SetTitle("New outline");
```

### Adjust the Order of the Outlines

Move a specific outline to be a child outline of the target outline.

The steps for moving an outline are as follows:

1. Obtain the outline object for the target outline.
2. Specify the position index of the child outline under the target outline, and move the outline to the corresponding position.

This example shows how to adjust the order of the outlines:

```
// Retrieve the outline object for the target outline.
CPDFOutline targetOutline = document.GetOutlineList()[1];
// Specify the index of the child outline to move to and move the designated outline to
the specified position.
targetOutline.MoveChildAtIndex(document, outline, targetOutline.ChildList.Count);
```

#### 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:

```
CPDFOutline outline = document.GetOutlineList()[0];
outline.RemoveFromParent(document);
```

## 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:

```
List<CPDFBookmark> bookmarkList = document.GetBookmarkList();
```

#### 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:

```
// Create a bookmark object.
CPDFBookmark bookmark = new CPDFBookmark();
// Set bookmark properties.
bookmark.Title = "new bookmark";
bookmark.PageIndex = 4;
// Add the bookmark to the document.
document.AddBookmark(bookmark);
```

#### 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(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 search and select 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. Specify the storage location for search results and the content container.
2. Create `CPDFTextPage` and `CPDFTextSearcher` objects.
3. Specify the `CPDFTextPage` object to be searched, the keywords for the search, search options, and the number of characters before and after the exact search result.
4. Use a temporary variable to store the context of the search results.
5. Record the contents of the temporary variable in the container whenever a search result is found.

This example shows how to  search specified text:

```
// instructing the storage location and content variables of search results
List<CRect> rectList = new List<CRect>();
List<string> stringList = new List<string>();
string searchKeywords = "searchKeywords";
int findIndex = 0;

// Specify the `CPDFTextPage` object to be searched, the keywords to be searched, the
search options for case sensitivity and whole word matching, and the starting index for
recording search results.
CPDFTextPage textPage = myCPDFPage.GetTextPage();
CPDFTextSearcher searcher = new CPDFTextSearcher();
if (searcher.FindStart(textPage, searchKeywords, C_Search_Options.Search_Case_Sensitive |
C_Search_Options.Search_Match_Whole_Word, 0))
{
// Use temporary variables to store the context and position of search results,  and
record them in the container whenever a result is found.
  CRect textRect = new CRect();
  string textContent = "";
```

```
  while (searcher.FindNext(page, textPage, ref textRect, ref textContent, ref findIndex))
  {
    // Record the content of the temporary variables in the container whenever a search
result is found.
    stringList.Add(textContent);
        rectList.Add(textRect);
  }
}
```

**Explanation of Search Settings**

| Option | Description | Value |
| --- | --- | --- |
| C_Search_Options.Search_Case_Insensitive | Case Insensitive | 0 |
| C_Search_Options.Search_Case_Sensitive | Case Sensitive | 1 |
| C_Search_Options.Search_Match_Whole_Word | Match Whole Word | 2 |

**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.

Create a rectangle by specifying two diagonal coordinates (Points), and capture the text covered by the rectangle along with the position of the text area. This simulates the action of dragging the mouse or finger to select text.

This example shows how to select specified text:

```
void SelectForPage(CPDFPage page, CPoint fromPoint, CPoint toPoint, ref List<CRect>
rects, ref string textContent)
{
    CPDFTextPage textPage = page.GetTextPage();
    // Form a rectangle by specifying two diagonal coordinate points and retrieve the
text content within the rectangular space defined by these two points.
    textContent = textPage.GetSelectText(fromPoint, toPoint, new CPoint(10, 10));
    // Create a rectangle by specifying two diagonal coordinate points and obtain the
position of the rectangular space between these two points, allowing a deviation of
(10,10).
    rects = textPage.GetCharsRectAtPos(fromPoint, toPoint, new CPoint(10, 10));
}
```

## 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:

```
CPDFPage page = myCPDFDocument.PageAtIndex(0);
string myText = page.GetString();
```

## 3.2.8 Zooming

Page zoom refers to adjusting the level of magnification or reduction of a page within a PDF document to accommodate the user's visual preferences or device screen size. `CPDFViewer` offers various automatic zoom modes and also supports manual zooming.

In automatic zoom mode, the document zoom level automatically adjusts to changes in the display area size as it evolves.

This example shows how to  set automatic zooming:

```
// Set the zoom mode to make the entire page visible within the display area.
myCPDFViewer.SetFitMode(FitMode.FitOriginal);
```

You can also set the zoom mode to custom and specify the scaling ratio.

This example shows how to  set custom zooming:

```
// Set the zoom mode to custom.
myCPDFViewer.SetFitMode(FitMode.FitZoom);

// Set the zoom mode to custom.
myCPDFViewer.SetZoom(5.0)
```

**Explanation of Zooming Modes**

| Mode | Description | Option Value |
| --- | --- | --- |
| Fit Width | Adjusts the page width to fit within the display area. | FitMode.FitWidth |
| Fit Height | Adjusts the page height to fit within the display area. | FitMode.FitHeight |
| Fit Original | Makes the entire page visible within the display area. | FitMode.FitOriginal |
| Fit Zoom | Does not adjust based on display area size; allows custom zoom value. | FitMode.FitZoom |

## 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:

```
myCPDFViewer.SetDrawModes(DrawMode.Dark);
```

**Explanation of Themes**

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

## 3.2.10 Custom Menu

The `CPDFViewer` supports a right-click menu feature. Right-clicking within the `CPDFViewer` area will trigger a custom menu.

You can set different custom menus for users in various contexts. For instance, setting the page zoom level when right-clicking in a blank area, enabling copy options when right-clicking on text, images, or annotations, etc.

The steps for setting up a custom menu are as follows:

1. Register for the custom menu event.
2. Create a context menu on `CPDFViewerTool` and add menu items (Copy, Cut, Paste, Delete).

This example shows how to  create a custom menu:

```
// Register custom menu event
toolManager.MouseRightButtonDownHandler += ToolManager_MouseRightButtonDownHandler;

 private void ToolManager_MouseRightButtonDownHandler(object sender, MouseEventObject e)
 {
     // Create a context menu on CPDFViewerTool and add menu items (Copy, Cut, Paste,
Delete).
     ContextMenu contextMenu = new ContextMenu();
```

```
     contextMenu.Items.Add(new MenuItem() { Header = "Copy", Command =
ApplicationCommands.Copy, CommandTarget = (UIElement)sender });
     contextMenu.Items.Add(new MenuItem() { Header = "Cut", Command =
ApplicationCommands.Cut, CommandTarget = (UIElement)sender });
     contextMenu.Items.Add(new MenuItem() { Header = "Paste", Command =
ApplicationCommands.Paste, CommandTarget = (UIElement)sender });
     contextMenu.Items.Add(new MenuItem() { Header = "Delete", Command =
ApplicationCommands.Delete, CommandTarget = (UIElement)sender });

     CPDFViewerTool tool = sender as CPDFViewerTool;
     tool.ContextMenu = contextMenu;
 }
```

### 3.2.11 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:

```
// Enable form field highlighting.
myCPDFView.SetFormFieldHighlight(true);
// Enable hyperlink highlighting.
myCPDFView.SetLinkHighlight(true);
```

### 3.2.12 Get the Selected Content

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

This example shows how to  Get the selected content:

```
// Extract the selected text.
TextSelectInfo textSelectInfo = viewerTool.GetTextSelectInfo();
textSelectInfo.PageSelectText.TryGetValue(pageIndex, out string text);

// Extract the selected image.
string tempPath = Path.Combine(Path.GetTempPath(), Guid.NewGuid() + ".jpg");
myCPDFPage.GetImgSelection().GetImgBitmap(imageIndex, tempPath);
```

## 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**

- **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 markers, comments, and text replies to enable 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.

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
int pageCount = document.PageCount;
if(pageCount>0)
{
    for(int pageIndex = 0;pageIndex < pageCount;pageIndex++)
    {
      // Obtain the page object and Access the list of annotations within the page
object.
        List<CPDFAnnotation>annotations =
document.PageAtIndex(pageIndex).GetAnnotations();
        if(annotations != null && annotations.Count != 0)
        {
          //  Iterate through each annotation object in the list.
          foreach(CPDFAnnotation annotation in annotations)
          {
            //  Perform relevant operations on the obtained annotation objects.
          }
        }
    }
}
```

## 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:

```
// Obtain the page object where the note needs to be created
CPDFPage page = document.PageAtIndex(0);

// Create a text annotation for the note on that page
CPDFTextAnnotation textAnnotation =
    page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_TEXT) as CPDFTextAnnotation;
```

```
// Set properties for the text annotation
textAnnotation.SetColor(new byte[] { 255, 0, 0 });
textAnnotation.SetTransparency(255);
textAnnotation.SetContent("ComPDFKit");
textAnnotation.SetRect(new CRect(300, 650, 350, 600));

// Update the annotation appearance to display it on the document
textAnnotation.UpdateAp();
```

### 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:

```
// Get the page object for which the annotation needs to be created.
CPDFPage page = document.PageAtIndex(0);
// Create a link annotation on the page.
CPDFLinkAnnotation link =
page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_LINK) as CPDFLinkAnnotation;
CPDFDestination dest = new CPDFDestination();
// Set the link to navigate to the second page using `CPDFDestination`.
dest.PageIndex = 1;
// Set annotation properties and attach the `CPDFDestination` object to the annotation.
link.SetRect(new CRect(0, 50, 50, 0));
link.SetDestination(document, dest);
```

### 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 free text annotation are as follows:

1. Obtain the page object where the annotation needs to be created.
2. Create a 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:

```
// Get the page object for which the annotation needs to be created.
CPDFPage page = document.PageAtIndex(0);
// Create a free text annotation on the page.
CPDFFreeTextAnnotation freeText =
page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_FREETEXT) as CPDFFreeTextAnnotation;
// Set annotation properties.
string str = "ComPDFKit Samples";
freeText.SetContent(str);
freeText.SetRect(new CRect(0, 100, 160, 0));
CTextAttribute textAttribute = new CTextAttribute();
textAttribute.FontName = "Helvetica";
textAttribute.FontSize = 12;
byte[] fontColor = { 255, 0, 0 };
textAttribute.FontColor = fontColor;
freeText.SetFreetextDa(textAttribute);
freeText.SetFreetextAlignment(C_TEXT_ALIGNMENT.ALIGNMENT_CENTER);
// Update the annotation in the document.
freeText.UpdateAp();
```

### 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:

```
// Get the page object for which annotations need to be created.
CPDFPage page = document.PageAtIndex(0);
float[] dashArray = { 2, 1 };
byte[] lineColor = { 255, 0, 0 };
byte[] bgColor = { 0, 255, 0 };

// Create a square annotation on the page.
CPDFSquareAnnotation square = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_SQUARE) as
CPDFSquareAnnotation;
// Set annotation properties.
square.SetRect(new CRect(10, 250, 200, 200));
square.SetLineColor(lineColor);
square.SetBgColor(bgColor);
square.SetTransparency(120);
square.SetLineWidth(1);
square.SetBorderStyle(C_BORDER_STYLE.BS_DASHED, dashArray);
// Update the annotation appearance to display it in the document.
```

```
square.UpdateAp();

// Create a circle annotation on the page.
CPDFCircleAnnotation circle = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_CIRCLE) as
CPDFCircleAnnotation;
// Set annotation properties.
circle.SetRect(new CRect(10, 410, 110, 300));
circle.SetLineColor(lineColor);
circle.SetBgColor(bgColor);
circle.SetTransparency(120);
circle.SetLineWidth(1);
circle.SetBorderStyle(C_BORDER_STYLE.BS_DASHED, dashArray);
// Update the annotation appearance to display it in the document.
circle.UpdateAp();

// Create a line annotation on the page.
CPDFLineAnnotation line = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_LINE) as
CPDFLineAnnotation;
// Set annotation properties.
line.SetLinePoints(new CPoint(300, 300), new CPoint(350, 350));
line.SetLineType(C_LINE_TYPE.LINETYPE_NONE, C_LINE_TYPE.LINETYPE_CLOSEDARROW);
line.SetLineColor(lineColor);
line.SetTransparency(120);
line.SetLineWidth(1);
line.SetBorderStyle(C_BORDER_STYLE.BS_DASHED, dashArray);
// Update the annotation appearance to display it in the document.
line.UpdateAp();
```

**Explanation of Line Types**

| Name | Description |
| --- | --- |
| LINETYPE_UNKNOWN | Non-standard or invalid line segment endpoint. |
| LINETYPE_NONE | No line segment endpoint. |
| LINETYPE_ARROW | Two short lines intersect at a sharp angle, forming an open arrow. |
| LINETYPE_CLOSEDARROW | Two short lines intersect at a sharp angle, similar to style one, and are connected by a third line, forming a triangular closed arrow filled with the interior color of the annotation. |
| LINETYPE_SQUARE | A square filled with the interior color of the annotation. |
| LINETYPE_CIRCLE | A circle filled with the interior color of the annotation. |
| LINETYPE_DIAMOND | A diamond shape filled with the interior color of the annotation. |
| LINETYPE_BUTT | A short line perpendicular to the line itself, located at the endpoint. |
| LINETYPE_ROPENARROW | Two short lines in opposite directions. |
| LINETYPE_RCLOSEDARROW | A triangular closed arrow in opposite directions. |
| LINETYPE_SLASH | A short line, located at the endpoint, rotated approximately 30 degrees clockwise from the direction perpendicular to the line itself. |

**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:

```
// Get the page object for which annotations need to be created.
CPDFPage page = document.PageAtIndex(0);
// Create a text object using the page object.
CPDFTextPage textPage = page.GetTextPage();
// Use this text object to obtain the positions of the text to be marked.
List<CRect> cRectList = textPage.GetCharsRectAtPos(new CPoint(0,0),new
CPoint(500,500),new CPoint(10,10));
```

```
// Create corresponding highlight annotation objects on the page.
CPDFHighlightAnnotation highlight =
page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_HIGHLIGHT) as CPDFHighlightAnnotation;
// Set properties for the highlight annotation object.
byte[] color = { 0, 255, 0 };
highlight.SetColor(color);
highlight.SetTransparency(120);
highlight.SetQuardRects(cRectList);
// Update the annotation appearance to display it in the document.
highlight.UpdateAp();
```

### 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:

```
// Get the page object for which annotations need to be created.
CPDFPage page = document.PageAtIndex(0);

// Create a standard stamp.
CPDFStampAnnotation standard = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_STAMP) as
CPDFStampAnnotation;
standard.SetStandardStamp("Approved");
standard.UpdateAp();

// Create a text stamp.
CPDFStampAnnotation text = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_STAMP) as
CPDFStampAnnotation;
text.SetTextStamp("test", "detail text", C_TEXTSTAMP_SHAPE.TEXTSTAMP_LEFT_TRIANGLE,
C_TEXTSTAMP_COLOR.TEXTSTAMP_RED);
text.UpdateAp();

// Create an image stamp.
CPDFStampAnnotation image = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_STAMP) as
CPDFStampAnnotation;
byte[] imageData = new byte[500 * 500];
image.SetImageStamp(imageData, 500, 500);
image.UpdateAp();
```

### Create Ink Annotations

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

The steps to create a ink 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 freehand annotations:

```
// Get the page object for which annotations need to be created.
CPDFPage page = document.PageAtIndex(0);
// Create a ink annotation on the page.
CPDFInkAnnotation ink = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_INK) as
CPDFInkAnnotation;
// Set the path traversed by the ink annotation.
List<List<CPoint>> points = new List<List<CPoint>>();
points.Clear();
points.Add(new List<CPoint>()
{
    new CPoint(10, 100),
    new CPoint(100, 10),
});
ink.SetInkPath(points);
// Set other properties of the annotation.
ink.SetInkColor(new byte[] { 255, 0, 0 });
ink.SetBorderWidth(2);
ink.SetTransparency(128);
ink.SetInkPath(points);
ink.SetThickness(8);
// Update the ink annotation in the document.
ink.UpdateAp();
```

**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:

```
// Get the page object for which annotations need to be created.
CPDFPage page = document.PageAtIndex(0);
// Create a sound annotation on the page.
CPDFSoundAnnotation sound = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_SOUND) as
CPDFSoundAnnotation;
// Set the path to the audio file.
sound.SetSoundPath(null, "soundFilePath");
// Set other properties.
sound.SetRect(new CRect(0, 50, 50, 0));
// Update the annotation appearance to display it in the document.
sound.UpdateAp();
```

## 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:

```
CPDFTextAnnotation textAnnotation = annotList[0] as CPDFTextAnnotation;
CTextAttribute textAttribute = new CTextAttribute()
{
  FontColor = new byte[] { 255, 0, 0 },
  FontSize = 20,
  FontName = "Helvitical"
};
textAnnotation.SetTextAttribute(textAttribute);
textAnnotation.UpdateAp();
```

## 3.3.6 Update Annotation Appearances

Annotations may include properties describing their appearance, such as annotation color or shape. However, these properties do not guarantee consistent display across different PDF readers. To address this issue, each annotation can define an appearance stream applied for rendering.

When modifying annotation properties, you must invoke the `UpdateAp()` method within the `CPDFAnnotation` class:

```
bool UpdateAp();
```

Setting a custom appearance stream for annotations is straightforward. This operation is commonly performed in stamp annotations, particularly because they don't have other properties. Stamp annotations used in this manner are often referred to as image annotations.

## 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:

```
// Get the page object for which annotations need to be deleted.
CPDFPage page = document.PageAtIndex(0);
// Get the list of annotations for this page.
List<CPDFAnnotation> annotList = page.GetAnnotations();
// Find the annotation to be deleted in the list and remove it.
annotList[0].RemoveAnnot();
```

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
document.ImportAnnotationFromXFDFPath("xfdfPath","tempPath");
```

**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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
document.ExportAnnotationToXFDFPath("xfdfPath","tempPath");
```

**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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("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

The annotation replies feature allows you and other users to engage in written discussions directly within the document. ComPDFKit provides a convenient API for accessing replies in the document, along with 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. Obtain the page object.
2. Retrieve annotations on the page.
3. Create a text reply annotation on the annotation and add the reply content.

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

```
// Obtain the page object.
CPDFPage page = document.PageAtIndex(0);

// Retrieve annotations on the page.
CPDFAnnotation annotation = page.GetAnnotations()[0];

// Create a text reply annotation on the annotation and add the reply content.
annotation.CreateReplyAnnotation();
```

**Create State Replies**

The status reply feature allows you to mark the status of an annotation, with options for checked and commented states. The status types can be distinguished using the `CPDFAnnotationState` enumeration.

The steps to create a status reply are as follows:

1. Obtain the page object.
2. Retrieve the annotation on the page.
3. Create a marked state reply.
4. Create a review state reply.

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

```
// Obtain the page object.
CPDFPage page = document.PageAtIndex(0);

// Retrieve the annotation on the page.
CPDFAnnotation annotation = page.GetAnnotations()[0];

// Create a marked state reply.
CPDFAnnotation markedReply =
annotation.CreateReplyStateAnnotation(CPDFAnnotationState.C_ANNOTATION_MARKED);

// Create a review state reply.
CPDFAnnotation stateReply =
annotation.CreateReplyStateAnnotation(CPDFAnnotationState.C_ANNOTATION_ACCEPTED);
```

**Enumeration Types for State Reply Annotations**

| Name | Description |
| --- | --- |
| C_ANNOTATION_MARKED | Checked state, used for checked status replies |
| C_ANNOTATION_UNMARKED | Unchecked state, used for checked status replies |
| C_ANNOTATION_ACCEPTED | Accepted state, used for commented status replies |
| C_ANNOTATION_REJECTED | Rejected state, used for commented status replies |
| C_ANNOTATION_CANCELLED | Cancelled state, used for commented status replies |
| C_ANNOTATION_COMPLETED | Completed state, used for commented status replies |
| C_ANNOTATION_NONE | No state, used for commented status replies |
| C_ANNOTATION_ERROR | State error |

### Getting All Replies

The Get All Replies feature allows you to retrieve all replies under an annotation, including text replies and status replies. The reply types can be distinguished using the `CPDFReplyAnnotationType` enumeration.

The steps to get all replies are as follows:

1. Obtain the page object.
2. Retrieve the annotation object on the page.
3. Retrieve all reply annotations and classify them.

```
// Obtain the page object.
CPDFPage page = document.PageAtIndex(0);

// Retrieve the annotation object on the page.
CPDFAnnotation annotation = page.GetAnnotations()[0];

// Obtain the status replies of the annotation.
foreach (CPDFAnnotation replyAnnot in annotation.GetReplies())
{
    if(replyAnnot.IsMarkedStateAnnot())
    {
        // Marked state reply.
    }
    else if(replyAnnot.IsReviewStateAnnot())
    {
        // Review state reply.
    }
    else
    {
        // Text reply.
    }
}
```

## 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 the annotation programmatically, you set it using two methods: `SetSourceRect` in the `CPDFAnnotation` class and `SetRotation` in the `CPDFAnnotationRotator` class.

- **CPDFAnnotation.SetSourceRect**

  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.

- **CPDFAnnotationRotator.SetRotation**

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

Here is the example code for annotation rotation:

```
// Obtain the page object.
CPDFPage page = document.PageAtIndex(0);

// Create standard stamp annotation.
CPDFStampAnnotation standard = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_STAMP) as
CPDFStampAnnotation;
standard.SetStandardStamp("Approved");

// Sets the rectangle before annotation rotation.
standard.SetSourceRect(new CRect(100, 150, 250, 100));

// Sets the rotation Angle of the annotation.
standard.AnnotationRotator.SetRotation(45);

standard.UpdateAp();
```

## 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 polygonal annotation:

```
// Obtain the page object.
CPDFPage page = document.PageAtIndex(0);

// Create a polygonal annotation.
```

```
CPDFPolygonAnnotation polygon = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_POLYGON)
as CPDFPolygonAnnotation;
polygon.SetPoints(new List<CPoint> { new CPoint(100, 100), new CPoint(200, 100), new
CPoint(200, 200), new CPoint(100, 200) });
polygon.SetRect(new CRect(100,200,200,100));
polygon.SetLineWidth(1);
polygon.SetLineColor(new byte[] { 255, 0, 0 });

// Set the border dashed style.
polygon.SetBorderStyle(C_BORDER_STYLE.BS_DASHDED, new float[]{2,1});

// Set the cloud border style.
CPDFBorderEffector borderEffector = new
CPDFBorderEffector(C_BORDER_TYPE.C_BORDER_TYPE_Cloud,
C_BORDER_INTENSITY.C_INTENSITY_ONE);
polygon.SetAnnotBorderEffector(borderEffector);

polygon.UpdateAp();
```

**Cloud Border style Type Enumeration**

| Name | Description |
| --- | --- |
| C_INTENSITY_ZERO | The influence intensity ranges from 0 to 2, and when it is 0, it is a solid line |
| C_INTENSITY_ONE | |
| C_INTENSITY_TWO | |

| Name | Description |
| --- | --- |
| C_BORDER_TYPE_STRAIGHT | The border style style is a solid line |
| C_BORDER_TYPE_Cloud | The border style style is cloud style |

# 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 `CPDFViewer` 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. | CPDFCheckBoxWidget |
| Radio Button | Select one option from predefined choices. | CPDFRadioButtonWidget |
| Push Button | Create custom buttons on the PDF document that acts when pressed. | CPDFPushButtonWidget |
| List Box | Select one or more options from a predefined list. | CPDFListBoxWidget |
| Combo Box | Select one option from a drop-down list of available text options. | CPDFComboBoxWidget |
| Text | Input text content such as name, address, email, etc. | CPDFTextWidget |
| Signature | Digitally sign or electronically sign the PDF document. | CPDFSignatureWidget |

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);
CPDFTextWidget textField = page.CreateWidget(C_WIDGET_TYPE.WIDGET_TEXTFIELD) as
CPDFTextWidget;
textField.SetRect(new CRect(28, 75, 235, 32));
textField.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
textField.SetWidgetBgRGBColor(new byte[] {240,255,240});
```

**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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);

// Create button to jump to the second page.
CPDFPushButtonWidget pushButton1 = page.CreateWidget(C_WIDGET_TYPE.WIDGET_PUSHBUTTON) as
CPDFPushButtonWidget;
pushButton1.SetRect(new CRect(28, 150, 150, 100));
pushButton1.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
pushButton1.SetWidgetBgRGBColor(new byte[] { 180, 180, 220 });
pushButton1.SetButtonTitle("Go To Page 2");
CTextAttribute attribute = new CTextAttribute();
attribute.FontColor = new byte[] { 0, 0, 0 };
attribute.FontSize = 14;
attribute.FontName = "Helvetica";
pushButton1.SetTextAttribute(attribute);
CPDFGoToAction gotoAction = new CPDFGoToAction();
CPDFDestination dest = new CPDFDestination();
dest.PageIndex = 1;
gotoAction.SetDestination(document, dest);
pushButton1.SetButtonAction(gotoAction);
pushButton1.UpdateFormAp();

// Create button to jump to the web page.
CPDFPushButtonWidget pushButton2 = page.CreateWidget(C_WIDGET_TYPE.WIDGET_PUSHBUTTON) as
CPDFPushButtonWidget;
pushButton2.SetRect(new CRect(168, 150, 290, 100));
pushButton2.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
pushButton2.SetWidgetBgRGBColor(new byte[] { 180, 180, 220 });
pushButton2.SetButtonTitle("Go To ComPDFKit");
CTextAttribute attribute2 = new CTextAttribute();
attribute2.FontColor = new byte[] { 0, 0, 0 };
attribute2.FontSize = 14;
attribute2.FontName = "Helvetica";
pushButton2.SetTextAttribute(attribute);

CPDFUriAction uriAction = new CPDFUriAction();
uriAction.SetUri("https://www.compdf.com/");

pushButton2.SetButtonAction(uriAction);
pushButton2.UpdateFormAp();
```

### 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);
CPDFListBoxWidget listbox = page.CreateWidget(C_WIDGET_TYPE.WIDGET_LISTBOX) as
CPDFListBoxWidget;
listbox.SetRect(new CRect(28, 330, 150, 230));
listbox.AddOptionItem(0, "1", "ComPDFKit1");
listbox.AddOptionItem(1, "2", "ComPDFKit2");
listbox.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
listbox.SetWidgetBgRGBColor(new byte[] { 200, 180, 180 });
```

### 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:

```
CPDFPage page = document.PageAtIndex(0);
CPDFSignatureWidget signatureField =
page.CreateWidget(C_WIDGET_TYPE.WIDGET_SIGNATUREFIELDS) as CPDFSignatureWidget;
signatureField.SetRect(new CRect(28, 420, 150, 370));

signatureField.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
signatureField.SetWidgetBgRGBColor(new byte[] { 150, 180, 210 });
```

### 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);
CPDFCheckBoxWidget checkBox = page.CreateWidget(C_WIDGET_TYPE.WIDGET_CHECKBOX) as
CPDFCheckBoxWidget;
checkBox.SetRect(new CRect(28, 470, 48, 450));
checkBox.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
checkBox.SetWidgetBgRGBColor(new byte[] { 150, 180, 210 });
```

**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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);
CPDFRadioButtonWidget radioButton = page.CreateWidget(C_WIDGET_TYPE.WIDGET_RADIOBUTTON)
as CPDFRadioButtonWidget;
radioButton.SetRect(new CRect(28, 500, 48, 480));
radioButton.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
radioButton.SetWidgetBgRGBColor(new byte[] { 210, 180, 150 });
radioButton.SetWidgetCheckStyle(C_CHECK_STYLE.CK_CIRCLE);
```

**Create ComboBoxes**

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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPage page = document.PageAtIndex(0);
CPDFComboBoxWidget comboBox = page.CreateWidget(C_WIDGET_TYPE.WIDGET_COMBOBOX) as
CPDFComboBoxWidget;
comboBox.SetRect(new CRect(28, 330, 150, 230));
comboBox.AddOptionItem(0, "1", "ComPDFKit1");
comboBox.AddOptionItem(1, "2", "ComPDFKit2");
comboBox.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
comboBox.SetWidgetBgRGBColor(new byte[] { 200, 180, 180 });
```

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
CPDFPage page = document.PageAtIndex(0);

List<CPDFAnnotation> annotList = page.GetAnnotations();
foreach (CPDFAnnotation annot in annotList)
{
    if (annot.Type==C_ANNOTATION_TYPE.C_ANNOTATION_WIDGET)
    {
        CPDFWidget widget = annot as CPDFWidget;
        switch(widget.WidgetType)
        {
            case C_WIDGET_TYPE.WIDGET_TEXTFIELD:
                {
                    CPDFTextWidget text = widget as CPDFTextWidget;
                    text.SetText("test");
                    text.UpdateFormAp();
                }
                break;

            case C_WIDGET_TYPE.WIDGET_RADIOBUTTON:
                {
                    CPDFRadioButtonWidget radio = widget as CPDFRadioButtonWidget;
                    radio.SetChecked(true);
                    radio.UpdateFormAp();
```

```
                }
                break;

            case C_WIDGET_TYPE.WIDGET_LISTBOX:
                {
                    CPDFListBoxWidget listBox = widget as CPDFListBoxWidget;
                    listBox.SelectItem(0);
                    listBox.UpdateFormAp();
                }
                break;

            default:
                break;
        }
    }
}
```

## 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:

```
CPDFTextWidget textWidget = myCPDFWidget as CPDFTextWidget;
textWidget.SetWidgetBorderRGBColor(new byte[] {255, 0, 0});
textWidget.UpdateFormAp();
```

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
CPDFPage page = document.PageAtIndex(0);

List<CPDFAnnotation>annotList = page.GetAnnotations();
annotList[0].RemoveAnnot();
```

### 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("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:

```
// Insert after the first page.
int pageIndex = 1;
int pageWidth = 595;
int pageHeight = 842;
// The `InsertPage` method allows specifying an image path, and when the image path is
empty, it inserts a blank page.
document.InsertPage(pageIndex, pageWidth, pageHeight, "");
```

**Insert Pages from other PDFs**

This example shows how to insert pages from other PDFs:

```
CPDFDocument documentForInsert = CPDFDocument.InitWithFilePath("OtherPDF.pdf");
document.ImportPagesAtIndex(documentForInsert, "1", 1);
```

## 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:

```
CPDFDocument documentPart1 = CPDFDocument.CreateDocument();
// Split the first and second pages into separate sections.
documentPart1.ImportPagesAtIndex(document, "1-2", 0);

CPDFDocument documentPart2 = CPDFDocument.CreateDocument();
// Split the third, fourth, and fifth pages into the second section.
documentPart2.ImportPagesAtIndex(document, "3-5", 0);
```

## 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:

```
CPDFDocument documentforMerge = CPDFDocument.CreateDocument();

CPDFDocument document1 = CPDFDocument.InitWithFilePath("filePath");
CPDFDocument document2 = CPDFDocument.InitWithFilePath("filePath2");

documentforMerge.ImportPagesAtIndex(document1,"1-10",document.PageCount);
documentforMerge.ImportPagesAtIndex(document2,"1-10",document.PageCount);
```

## 3.5.5 Delete Pages

This example shows how to delete pages:

```
List<int> pageNumbersToRemove = new List<int>(){1};
// Delete the first page of the document.
document.RemovePages(pageNumbersToRemove.ToArray());
```

### 3.5.6 Rotate Pages

This example shows how to rotate pages:

```
// Rotate the first page 90 degrees clockwise, with each unit of rotation representing a 90-degree clockwise turn.
document.RotatePage(0, 1);
```

### 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:

```
List<int> pageList = new List<int>() { 0 };
// Remove the first page from the document.
document.RemovePages(pageList.ToArray());
CPDFDocument documentForInsert = CPDFDocument.InitWithFilePath("OtherPDF.pdf");
// Insert the first page of another document into the original document's first-page position to complete the replacement.
document.ImportPagesAtIndex(documentForInsert, "1", 0);
```

### 3.5.8 Extract Pages

This example shows how to extract pages:

```
CPDFDocument extractDocument = CPDFDocument.CreateDocument();
// Extract the first page of the original document and save it in a new document.
extractDocument.ImportPagesAtIndex(document, "1", 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPermissionsInfo permission = new CPDFPermissionsInfo()
{
    AllowsCopying = true,
    AllowsPrinting = false
};
CPDFDocumentEncryptionLevel algorithm =
CPDFDocumentEncryptionLevel.CPDFDocumentEncryptionLevelRC4;
document.Encrypt("userPassword", "ownerPassword", permission, algorithm);
```

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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFPermissionsInfo permission = document.GetPermissionsInfo();
Console.WriteLine("Allows Printing: " + permission.AllowsPrinting);
Console.WriteLine("Allows Copying: " + permission.AllowsCopying);
```

**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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
if (document.IsLocked)
{
    document.UnlockWithPassword(password);
}
document.Decrypt("Save Path");
```

## 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:

```
CPDFBackground background = document.GetBackground();
background.SetBackgroundType(C_Background_Type.BG_TYPE_COLOR);
background.SetColor(new byte[] { 255, 0, 0 });
background.SetOpacity(255);// 0-255
background.SetScale(1);// 1 == 100%
background.SetRotation(0);// Units: Radians
background.SetHorizalign(C_Background_Horizalign.BG_HORIZALIGN_CENTER);
background.SetVertalign(C_Background_Vertalign.BG_VERTALIGN_CENTER);
background.SetXOffset(0);
background.SetYOffset(0);
background.SetPages("0-2");
background.Update();
```

### Set Image Background

The steps to set an image background are as follows:

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:

```
CPDFBackground background = document.GetBackground();
background.SetBackgroundType(C_Background_Type.BG_TYPE_IMAGE);
byte[] imageData = new byte[500 * 500];
background.SetImage(imageData, 500, 500, ComPDFKit.Import.C_Scale_Type.fitCenter);
background.SetOpacity(128);// 0-255
background.SetScale(1);// 1 == 100%
background.SetRotation(1f);// Units: Radians
background.SetHorizalign(C_Background_Horizalign.BG_HORIZALIGN_CENTER);
background.SetVertalign(C_Background_Vertalign.BG_VERTALIGN_CENTER);
background.SetXOffset(0);
background.SetYOffset(0);
background.SetPages("0-2");
background.Update();
```

**Remove Background**

The steps to remove the background are as follows:

1. Obtain the document background object.
2. Delete the document background.

This example shows how to remove background:

```
CPDFBackground background = document.GetBackground();
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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File Path");
CPDFHeaderFooter headerFooter = document.GetHeaderFooter();
headerFooter.SetPages("1-3");
headerFooter.SetText(0, @"<<1,2>> page");
byte[] color = { 255, 0, 0 };
headerFooter.SetTextColor(0, color);
headerFooter.SetFontSize(0, 14);
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:

```
CPDFHeaderFooter headerFooter = document.GetHeaderFooter();
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 numbers.

**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 the Bates number：

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
CPDFBates bates = document.GetBates();
bates.SetText(0, @"<<#3#5#Prefix-#-Suffix>>");
byte[] color = { 255, 0, 0 };
bates.SetTextColor(0, color);
bates.SetFontSize(0,14);
bates.SetPages("0-" + (document.PageCount - 1));
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 remove 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：

```
CPDFHeaderFooter headerFooter = document.GetHeaderFooter();
headerFooter.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 in PDF documents.

## 3.7.2 Redact PDFs

The steps to redact PDFs are as follows:

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 `SetQuardRects` or `SetRect` 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 redaction annotations：

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");

CPDFPage page = document.PageAtIndex(0);
CPDFRedactAnnotation redact = page.CreateAnnot(C_ANNOTATION_TYPE.C_ANNOTATION_REDACT) as
CPDFRedactAnnotation;

redact.SetRect(new CRect(0, 50, 50, 0));
redact.SetOverlayText("REDACTED");
CTextAttribute textDa = new CTextAttribute();
textDa.FontName = "Helvetica";
textDa.FontSize = 12;
```

```
byte[] fontColor = { 255, 0, 0 };
textDa.FontColor = fontColor;
redact.SetTextDa(textDa);
redact.SetTextAlignment(C_TEXT_ALIGNMENT.ALIGNMENT_LEFT);
byte[] fillColor = { 255, 0, 0 };
redact.SetFillColor(fillColor);
byte[] outlineColor = { 0, 255, 0 };
redact.SetOutlineColor(outlineColor);

redact.UpdateAp();
```

**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：

```
document.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 the `CPDFWatermark` object, specifying the type as text.
CPDFWatermark watermark = document.InitWatermark(C_Watermark_Type.WATERMARK_TYPE_TEXT);
// Set the properties for text content, font, color, and font size.
watermark.SetText("test");
watermark.SetFontName("Helvetica");
byte[] color = { 255, 0, 0 };
// Configure the properties required for the text watermark, including content, font,
color, and font size.
watermark.SetPages("0-3");
watermark.SetTextRGBColor(color);
watermark.SetScale(2);
watermark.SetRotation(0);
watermark.SetOpacity(120);
watermark.SetVertalign(C_Watermark_Vertalign.WATERMARK_VERTALIGN_CENTER);
watermark.SetHorizalign(C_Watermark_Horizalign.WATERMARK_HORIZALIGN_CENTER);
watermark.SetVertOffset(0);
watermark.SetHorizOffset(0);
watermark.SetFront(true);
watermark.SetFullScreen(true);
watermark.SetVerticalSpacing(10);
watermark.SetHorizontalSpacing(10);
// Create the watermark in the document.
watermark.CreateWatermark();
```

## 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 and specify the watermark type as an image.
CPDFWatermark watermark = document.InitWatermark(C_Watermark_Type.WATERMARK_TYPE_IMG);
```

```
// Set the image source and scaling ratio for the image watermark.
byte[] imageData = new byte[500 * 500];
watermark.SetImage(imageData, 500, 500);
watermark.SetScale(2);
// Set general properties for the watermark.
watermark.SetPages("0-3");
watermark.SetRotation(1);
watermark.SetOpacity(128);
watermark.SetVertalign(C_Watermark_Vertalign.WATERMARK_VERTALIGN_CENTER);
watermark.SetHorizalign(C_Watermark_Horizalign.WATERMARK_HORIZALIGN_CENTER);
watermark.SetVertOffset(0);
watermark.SetHorizOffset(0);
watermark.SetFront(false);
watermark.SetFullScreen(true);
watermark.SetVerticalSpacing(10);
watermark.SetHorizontalSpacing(10);
// Create a watermark in the document.
watermark.CreateWatermark();
```

### 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 all watermarks from the document.
watermarkDocument.DeleteWatermarks();
```

## 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:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
document.WritePDFAToFilePath(CPDFType.CPDFTypePDFA1a,"savePath");
```

**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, images and path, allowing users to easily insert, adjust, and delete text, images and path. 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 in a document, such as cut, copy, move, delete, and other operations.
- **Search and Replace:** Search for a keyword and replace it with other content.
- **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, Images and Path

  Create, move, and delete text, images and path.
- 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.
- Search and Replace

  Set keywords and search settings, check and replace specified matches one by one, or replace all matches with a single click.

## 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:

```
CPDFDocument doc = CPDFDocument.InitWithFilePath("filePath");
CPDFViewerTool tool = new CPDFViewerTool();
tool.GetCPDFViewer().InitDoc(doc);
CPDFToolManager toolManager = new CPDFToolManager(tool);
toolManager.SetToolType(ToolType.ContentEdit);
tool.SetCurrentEditType(CPDFEditType.EditText | CPDFEditType.EditImage |
CPDFEditType.EditPath);
```

**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. | CPDFEditType.EditText |
| 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. | CPDFEditType.EditImage |
| Path Mode | In path mode, the path in the PDF document can be selected, which can be cut, copied, moved, deleted, and so on. | CPDFEditType.EditPath |
| 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, images and path. | CPDFEditType.EditText\|CPDFEditType.EditImage\|CPDFEditType.EditPath |

## 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 `CPDFViewer`

`CPDFViewer` 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, text blocks and path, resize images, text blocks and path, etc. by using the mouse and keyboard.

### Configure the Context Menu

If you need to copy, paste, cut, or delete text, images and path, you can add these methods to the context menu through the `MouseRightButtonDownHandler` event of `CPDFToolManager`.

This example shows how to configure the context menu:

```
// Register custom menu event
toolManager.MouseRightButtonDownHandler += ToolManager_MouseRightButtonDownHandler;

private void ToolManager_MouseRightButtonDownHandler(object sender, MouseEventObject e)
{
    // Create a context menu on CPDFViewerTool and add menu items (Copy, Cut, Paste,
Delete).
    ContextMenu contextMenu = new ContextMenu();
    contextMenu.Items.Add(new MenuItem() { Header = "Copy", Command =
ApplicationCommands.Copy, CommandTarget = (UIElement)sender });
    contextMenu.Items.Add(new MenuItem() { Header = "Cut", Command =
ApplicationCommands.Cut, CommandTarget = (UIElement)sender });
    contextMenu.Items.Add(new MenuItem() { Header = "Paste", Command =
ApplicationCommands.Paste, CommandTarget = (UIElement)sender });
    contextMenu.Items.Add(new MenuItem() { Header = "Delete", Command =
ApplicationCommands.Delete, CommandTarget = (UIElement)sender });

    CPDFViewerTool tool = sender as CPDFViewerTool;
    tool.ContextMenu = contextMenu;
}
```

### Inserting Text and Images

You can specify the ability to insert text and image blocks through the `SetPDFEditCreateType` method of `CPDFViewer`. The following code shows how to achieve this:

```
// Insert Image
myCPDFView.SetPDFEditCreateType(CPDFEditType.EditImage);
// Insert Text
myCPDFView.SetPDFEditCreateType(CPDFEditType.EditText);
// Cancel
myCPDFView.SetPDFEditCreateType(CPDFEditType.None);
```

## 3.10.4 Edit Text and Image Properties

ComPDFKit supports to modify the properties of text and images. To configure text and image properties, you can utilize the `MouseLeftButtonDownHandler` event of `CPDFToolManager`.

### Edit Text Properties

ComPDFKit supports modifying text properties, such as text font size, name, color, alignment, italics, bold, transparency, etc.

This example shows how to set text to 12pt, red, and bold:

```
toolManager.MouseLeftButtonDownHandler += ToolManager_MouseLeftButtonDownHandler;
private void ToolManager_MouseLeftButtonDownHandler(object sender, MouseEventObject e)
{
    int pageIndex = -1;
    CPDFToolManager manager = sender as CPDFToolManager;
    CPDFEditArea editArea = manager.GetSelectedEditAreaObject(ref pageIndex);
    if (editArea != null && editArea.Type == CPDFEditType.EditText)
    {
        CPDFEditPage editPage =
manager.GetDocument().PageAtIndex(pageIndex).GetEditPage();
        CPDFEditTextArea textArea = editArea as CPDFEditTextArea;
        Rect textRect = DataConversionForWPF.CRectConversionForRect(textArea.GetFrame());

        // Text properties
        textArea.SetCharsFontColor(255,0,0);
        textArea.SetCharsFontTransparency(255);
        textArea.SetCharsFontSize(12,true);
        textArea.SetCharsFontName("Arial");
        textArea.SetCharsFontBold(true);
        textArea.SetCharsFontItalic(true);
        textArea.SetTextAreaAlign(TextAlignType.AlignLeft);
        editPage.EndEdit();

        manager.GetCPDFViewerTool().UpdateRender(textRect, textArea);
    }
}
```

Below is the example code for creating and removing underlines and strikethroughs for text:

```
// creating underlines and strikethroughs for text
textArea.AddUnderline();
textArea.AddStrikethrough();

// removing underlines and strikethroughs for text
textArea.RemoveUnderline();
textArea.RemoveStrikethrough();
```

**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:

```
toolManager.MouseLeftButtonDownHandler += ToolManager_MouseLeftButtonDownHandler;
private void ToolManager_MouseLeftButtonDownHandler(object sender, MouseEventObject e)
{
    int pageIndex = -1;
    CPDFToolManager manager = sender as CPDFToolManager;
    CPDFEditArea editArea = manager.GetSelectedEditAreaObject(ref pageIndex);
    if (editArea != null && editArea.Type == CPDFEditType.EditText)
    {
        CPDFEditPage editPage =
manager.GetDocument().PageAtIndex(pageIndex).GetEditPage();
        CPDFEditImageArea imageArea = editArea as CPDFEditImageArea;
         Rect imageRect =
DataConversionForWPF.CRectConversionForRect(imageArea.GetFrame());
      //Image properties.
    imageArea.Rotate(90);
        imageArea.HorizontalMirror();

        manager.GetCPDFViewerTool().UpdateRender(imageRect, imageArea);
        editPage.EndEdit();
    }
}
```

## 3.10.5 Undo and Redo

The `UndoManager` class of `CPDFViewer` provides undo and redo functionalities.

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(myCPDFViewer.UndoManager.CanUndo)
{
    myCPDFViewer.UndoManager.Undo();
}
// redo
if (myCPDFViewer.UndoManager.CanRedo)
{
    myCPDFViewer.UndoManager.Redo();
}
```

## 3.10.6 End Content Editing and Save

After completing edits, you can exit editing mode by using `SetMouseMode` with `CPDFToolManager` to switch out of edit mode. To save the changes, utilize either `WriteToLoadedPath` or `WriteToFilePath` save the modified document.

```
// Exit edit mode and switch to PanTool mode.
toolManager.SetToolType(ToolType.Pan);

if(tool.IsDocumentModified)
{
    // Save changes to the local document.
    doc.WriteToLoadedPath();
}
```

## 3.10.7 Find and Replace

In content editing mode, users can perform text search and replace within PDF documents.

**Text Search**

After entering the keywords that need to be replaced and configuring the search parameters, you can navigate throughout the entire document by using forward and backward search, selecting and highlighting the matching items.

This example shows how to  entering keywords and search settings, as well as selecting the next and previous matches:

```
// Enter the keywords to be replaced along with the search settings.
tool.StartFindText(keyWord, searchOption);
// Find and select the next match.
tool.FindNext();
// Find and select the previous match.
tool.FindPrevious();
```

**Explanation of Search Options**

| options | Discription | value |
| --- | --- | --- |
| C_Search_Options.Search_Case_Insensitive | Match Case Insensitive | 0 |
| C_Search_Options.Search_Case_Sensitive | Match Case Sensitive | 1 |
| C_Search_Options.Search_Match_Whole_Word | Match Whole Word | 2 |

**Replace Single**

Users can choose to replace the currently selected match.

This example shows how to  replacing the currently selected match:

```
// Replace the currently selected match
tool.ReplaceText(replaceWord);
```

**Replace All**

Users can choose to replace all matching items.

This example shows how to replacing all matching items throughout the document:

```
// Replace all matching items
tool.ReplaceAllText(replaceWord);
```

# 3.11 Compare Documents

## 3.11.1 Overview

The document comparison feature is utilized to compare the differences between two documents, highlighting modifications, additions, or deletions. It assists users in identifying changes in complex drawings, text, and image content.

**Benefits of ComPDFKit Document Comparison**

- **Overlay Comparison：** Highlight document differences using colored lines, suitable for comparing variances in complex drawings.
- **Content Comparison：** Retrieve differential content and locations, suitable for document comparison with substantial text and image differences.

**Guides for Document Comparison**

- Overlay Comparison

  Generate a comparative result document based on two documents for comparison, marking document differences with colored lines.

- Content Comparison

  Compare the content of two documents (including text and images) and present the differences in a list format.

## 3.11.2 Overlay Comparison

Overlay Comparison is used to visually compare pages of different documents. It's helpful for things such as construction plans and detailed drawings, as well as other content that requires precise placement.

Setting a different stroke color is usually necessary when trying to compare documents. During the document comparison, the two documents overlap, and repetitive positions become a uniform color due to color overlay. Inconsistent parts retain distinct stroke colors, thereby displaying all differences between the two documents prominently.

The color settings here only affect the stroke objects and do not alter the color of other elements, such as text or images.

The steps to perform overlay comparison are as follows:

1. Open the two documents to be compared.
2. Create a document comparison object.
3. Compare the two documents and generate a comparison document.
4. Save the comparison document.

This example shows how to perform overlay comparison:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File1.pdf");
CPDFDocument dewDocument = CPDFDocument.InitWithFilePath("File2.pdf");
CPDFCompareOverlay compareOverlay = new CPDFCompareOverlay(document, "1-5", dewDocument,
"1-5");
compareOverlay.Compare();
CPDFDocument comparisonDocument = compareOverlay.ComparisonDocument();
comparisonDocument.WriteToFilePath("Save Path");
```

- You can modify the stroke colors for the two versions of the document by using the `SetOldDocumentStrokeColor` and `SetNewDocumentStrokeColor` properties.
- You can also change the blend mode used to overlay the new version of a document on top of the old one by changing the `SetBlendMode` property.
- Trying out various stroke colors and blend modes will result in different-looking comparison documents, and you can make sure the final result fits your needs.

## 3.11.3 Content Comparison

Comparing the content of two versions of PDF files, including text and images, enables the identification of deleted, added, and replaced content. Displaying the differences in a list format supports the ability to click and navigate to the specific change points within the PDF documents.

The steps to perform content comparison are as follows:

1. Open the two documents for comparison.
2. Create a content comparison object.
3. Compare the specified pages.

4. Retrieve the results of the content comparison.

This example shows how to perform content comparison:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("File1.pdf");
CPDFDocument dewDocument = CPDFDocument.InitWithFilePath("File2.pdf");
CPDFCompareContent compareContent = new CPDFCompareContent(document, dewDocument);
int pageCount = Math.Min(document.PageCount, dewDocument.PageCount);
for (int i = 0; i < pageCount; i++)
{
    Console.WriteLine("Page: {0}", i);

    CPDFCompareResults compareResults = compareContent.Compare(i, i,
CPDFCompareType.CPDFCompareTypeAll, true);
    Console.WriteLine("Replace count: {0}", compareResults.ReplaceCount);
    Console.WriteLine("TextResults count: {0}", compareResults.TextResults.Count);
    Console.WriteLine("Delete count: {0}", compareResults.DeleteCount);
    Console.WriteLine("Insert count: {0}", compareResults.InsertCount);
}
```

- You can compare different content types in a document by setting the `type`. For example, using `CPDFCompareTypeText` will only compare text, while using `CPDFCompareTypeAll` will compare all content.
- One of the most important steps in generating the comparison results is the ability to change the highlight colors, which makes it easier to see the differences between two versions of a document. The highlight colors of both versions of a document can be changed using the `SetReplaceColor`, `SetInsertColor`, and `SetDeleteColor` properties.

# 3.12 Digital Signatures

## 3.12.1 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 Signature SDK**

- **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.

## 3.12.2 Concepts of Digital Signatures

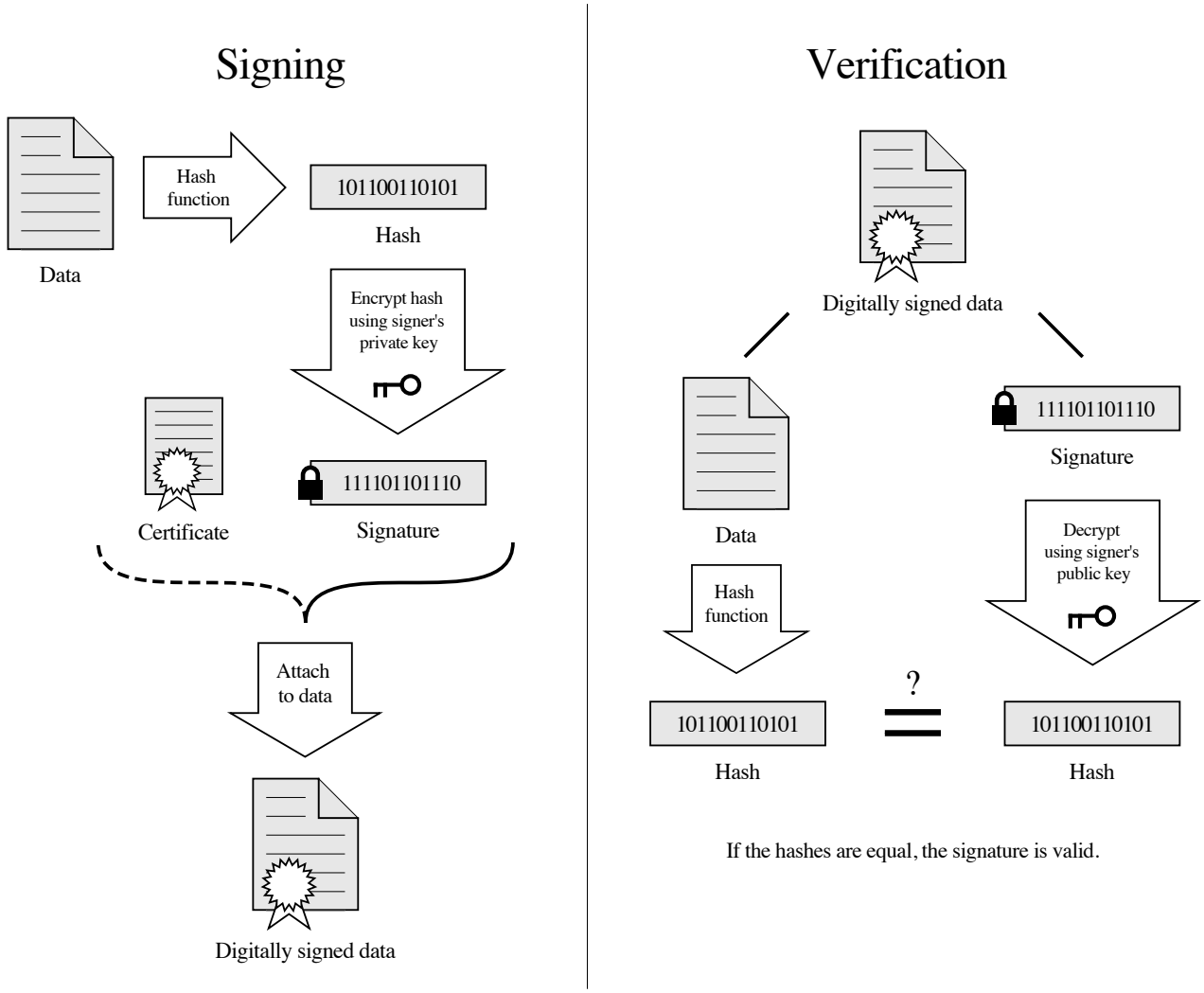
**How Digital Signatures Work**

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
```

## 3.12.3 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
//
string password = "ComPDFKit";
string info = "/C=SG/O=ComPDFKit/D=R&D Department/CN=Alan/emailAddress=xxxx@example.com";
string filePath = outputPath + "\\Certificate.pfx";
CPDFPKCS12CertHelper.GeneratePKCS12Cert(info, password, filePath,
CPDFCertUsage.CPDFCertUsageAll, false);
```

## 3.12.4 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 }
//
CPDFPage page = document.PageAtIndex(0);
            CPDFSignatureWidget signatureField =
page.CreateWidget(C_WIDGET_TYPE.WIDGET_SIGNATUREFIELDS) as CPDFSignatureWidget;
            signatureField.SetRect(new CRect(28, 420, 150, 370));
            signatureField.SetWidgetBorderRGBColor(new byte[] { 0, 0, 0 });
            signatureField.SetWidgetBgRGBColor(new byte[] { 150, 180, 210 });
            signatureField.UpdateAp();
```

**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 the grantor's name from the 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"
//
CPDFSignatureCertificate certificate =
CPDFPKCS12CertHelper.GetCertificateWithPKCS12Path("Certificate.pfx", "ComPDFKit");
string name = GetGrantorFromDictionary(certificate.SubjectDict) + "\n";
string date = DateTime.Now.ToString("yyyy.MM.dd HH:mm:ss");
string reason = "I am the owner of the document.";
string location = certificate.SubjectDict["C"];
string DN = certificate.Subject;
CPDFSignatureWidget signatureField =
page.CreateWidget(C_WIDGET_TYPE.WIDGET_SIGNATUREFIELDS) as CPDFSignatureWidget;
CPDFSignatureConfig signatureConfig = new CPDFSignatureConfig
{
    Text = GetGrantorFromDictionary(certificate.SubjectDict),
    Content =
        "Name: " + name + "\n" +
        "Date: " + date + "\n" +
        "Reason: " + reason + " \n" +
        "Location: " + location + "\n" +
        "DN: " + DN + "\n",
    IsContentAlignLeft = false,
    IsDrawLogo = true,
    TextColor = new float[] { 0, 0, 0 },
    ContentColor = new float[] { 0, 0, 0 }
};

byte[] imageData = new byte[500 * 500];
signatureConfig.LogoData = imageData;
signatureConfig.LogoHeight = 500;
signatureConfig.LogoWidth = 500;

signatureField.UpdataApWithSignature(signatureConfig);
document.WriteSignatureToFilePath(signatureField,
    "filePath",
    "certificatePath", "password",
    location,
```

```
    reason,
    CPDFSignaturePermissions.CPDFSignaturePermissionsNone);


public static string GetGrantorFromDictionary(Dictionary<string, string> dictionary)
{
    string grantor = string.Empty;
    dictionary.TryGetValue("CN", out grantor);
    if (string.IsNullOrEmpty(grantor))
    {
        dictionary.TryGetValue("OU", out grantor);
    }
    if (string.IsNullOrEmpty(grantor))
    {
        dictionary.TryGetValue("O", out grantor);
    }
    if (string.IsNullOrEmpty(grantor))
    {
        grantor = "Unknown Signer";
    }
    return grantor;
}
```

## 3.12.5 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：

```
foreach (var signature in document.GetSignatureList())
{
    signature.VerifySignatureWithDocument(document);
    Console.WriteLine("Name: " + signature.Name);
    Console.WriteLine("Location: " + signature.Location);
    Console.WriteLine("Reason: " + signature.Reason);
    foreach (var signer in signature.SignerList)
    {
        Console.WriteLine("Date: " + signer.AuthenDate);
        foreach (var certificate in signer.CertificateList)
        {
            Console.WriteLine("Subject: " + certificate.Subject);
        }
    }
}
```

**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.

## 3.12.6 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.

This example shows how to verify digital certificates：

```
// Verify certificate.
//
// 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 call the "CPDFSignatureCertificate.CheckCertificateIsTrusted" first. Then
you can view the "CPDFSignatureCertificate.IsTrusted" property.
//
CPDFSignatureCertificate certificate =
CPDFPKCS12CertHelper.GetCertificateWithPKCS12Path(certificatePath, password);
certificate.CheckCertificateIsTrusted();
if (certificate.IsTrusted)
{
  // Certificate is trusted.
}
else
{
    // Certificate is not trusted.
}
```

## 3.12.7 Verify Digital Signatures

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：

```
foreach (var signature in document.GetSignatureList())
{
```

```
    signature.VerifySignatureWithDocument(document);
    foreach (var signer in signature.SignerList)
    {
        Console.WriteLine("Is the certificate trusted: " +
signer.IsCertTrusted.ToString());
        Console.WriteLine("Is the signature verified: " +
signer.IsSignVerified.ToString());
        // Take appropriate actions based on the verification results.
        if (signer.IsCertTrusted && signer.IsSignVerified)
        {
            // Signature is valid and the certificate is trusted.
            // Perform the corresponding actions.
         }
         else if (!signer.IsCertTrusted && signer.IsSignVerified)
         {
              // Signature is valid but the certificate is not trusted.
              // Perform the corresponding actions.
         }
         else
         {
               // Signature is invalid.
               // Perform the corresponding action.
          }
      }
}
```

## 3.12.8 Trust Certificate

Trusting certificates involve two steps:

1. Specify the trust path (folder) for certificates. This path serves as the location where certificates are placed when they are trusted. Additionally, when checking the trustworthiness of certificates, the SDK will look for the corresponding certificates within this folder. Please ensure that this path is valid. If the path does not exist or is inaccessible, the ComPDFKit SDK will not automatically create the trust path folder.
2. Execute the method to trust certificates, and the certificates will be added to the trust path.

This example shows how to trust certificates：

```
CPDFSignature signature = document.GetSignatureList()[0];
            CPDFSignatureCertificate signatureCertificate =
signature.SignerList[0].CertificateList[0];
 Console.WriteLine("Certificate trusted status: " +
signatureCertificate.IsTrusted.ToString());
Console.WriteLine("---Begin trusted---");
string trustedFolder = AppDomain.CurrentDomain.BaseDirectory + @"\TrustedFolder\";
if (!Directory.Exists(trustedFolder))
{
    Directory.CreateDirectory(trustedFolder);
```

```
}
// Set your trust path as a folder path.
CPDFSignature.SignCertTrustedFolder = trustedFolder;
// Add your certificate to the trust path.
signatureCertificate.AddToTrustedCertificates();
Console.WriteLine("Certificate trusted status: " +
signatureCertificate.IsTrusted.ToString());
```

## 3.12.9 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：

```
// Remove digital signature.
// You can choose if you want to remove the appearance.
CPDFSignature signature = document.GetSignatureList()[0];
document.RemoveSignature(signature, true);
string filePath = outputPath + "\\" + document.FileName + "_RemovedSign.pdf";
document.WriteToFilePath(filePath);
```

## 3.12.10 Trouble Shooting

**Inaccurate Signature Information Retrieval**

Before retrieving signature information, it is necessary to call the `VerifySignatureWithDocument` method within the `CPDFSignature` class. This method refreshes the document's integrity and checks the validity of the certificate. Failure to call this method before retrieving the signature information may result in obtaining outdated or incorrect results.

**Failure to Add Certificate to Trust Path**

Before calling the `AddToTrustedCertificates` method within the `CPDFSignatureCertificate` class, you must first set the value of the `SignCertTrustedFolder` parameter in the `AddToTrustedCertificates` class to your trust path folder. Failure to do so will result in the addition process failing. When you trust a certificate, it is added to the specified path, and the SDK also checks for the existence of certificates in that path to determine their trustworthiness. The trust path folder is not automatically created by the SDK, so you should ensure it exists and is a valid path.

# 3.13 Measurement

## 3.13.1 Overview

ComPDFKit PDF SDK supports drawing lines, polylines, and polygons with measurement information in PDF documents. This allows measuring distances, perimeters, and areas of specified sections in the document. By setting a scale factor, for example, defining one inch on the PDF document to correspond to one foot in actual distance, you can measure the actual dimensions of objects in architectural design drawings.

**Benefits of ComPDFKit Measurement**

- **Comprehensive Tools:** Utilize tools for straight lines, polylines, rectangles, and polygons to precisely draw and measure dimensions of complex graphics.
- **Data Richness:** Measure distances, perimeters, and areas, applicable to a wide range of scenarios.
- **Configurable Properties:** Set global scales freely, convert units, and adjust precision to meet specific requirements.
- **Adjust Measurement Annotations:** Adjust the scale, unit conversion, and precision of existing measurement annotations.
- **Appearance Settings:** Customize the appearance of measurement annotations, including lines, colors, fonts, and more.
- **Fast UI Integration:** Achieve swift integration and customization through extensible UI components.

**ComPDFKit Measurement Features**

**Configure Measurement Properties**

Before utilizing the measurement features, it is necessary to configure the scale, units, and precision.

**Measure Distance**

The distance measurement tool allows your users to measure the distance between two points in a PDF.

**Measure Perimeter and Area**

Utilize the perimeter and area measurement tools to measure the perimeter and area of enclosed graphics in a PDF.

**Adjust Existing Measurement Annotations**

Fine-tune properties related to measurements in existing measurement annotations.

## 3.13.2 Configure Measurement Properties

To set measurement scale and precision, you can utilize the class `CPDFMeasureInfo` within the `ComPDFKit.Measure`.

This example shows how to  configuring measurement properties:

```
MeasureSetting measureSetting = new MeasureSetting();

// Set the measurement scale in the PDF file to 1.0 inch.
measureSetting.RulerBase = 1.0;
measureSetting.RulerBaseUnit = CPDFMeasure.CPDF_IN;
```

```
// Set the translation of 1.0 inch in the PDF to correspond to 10.0 feet in the actual object.
measureSetting.RulerTranslate = 10.0;
measureSetting.RulerTranslateUnit = CPDFMeasure.CPDF_FT;

// Set the precision to 0.01.
measureSetting.Precision = CPDFMeasure.PRECISION_VALUE_TWO;

// (For enclosed graphics with measurable area) Set to display the area in the annotation appearance.
measureSetting.IsShowArea = true;

// (For enclosed graphics with measurable perimeter) Set to display the perimeter in the annotation appearance.
measureSetting.IsShowLength = true;
```

#### Supported Measurement Units

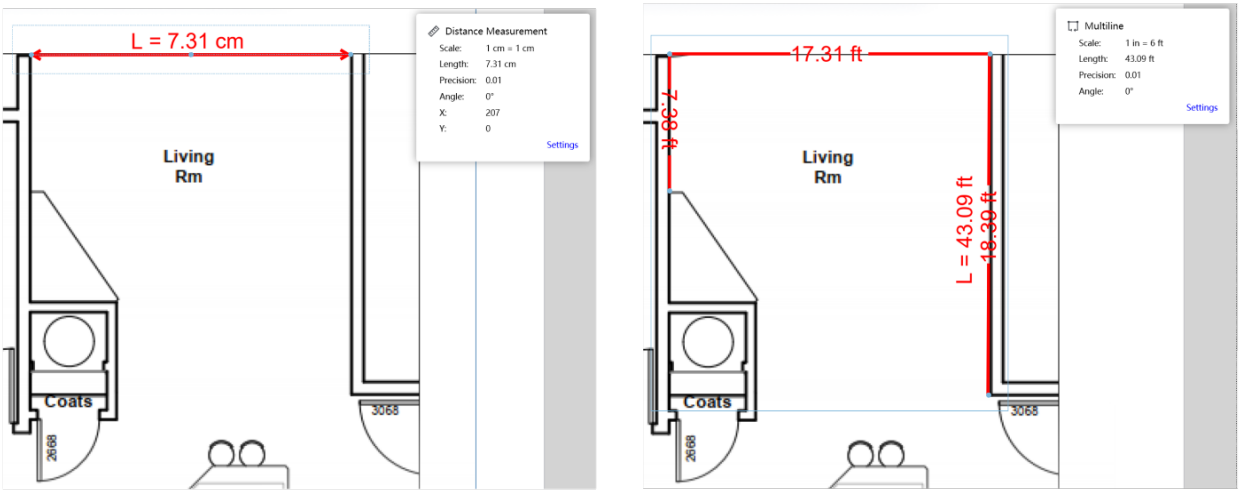
You can set the units for the measurement ruler on the PDF file and the size units for the actual object using `CPDFMeasureInfo.RulerBaseUnit` and `CPDFMeasureInfo.RulerTranslateUnit`. The supported units and corresponding parameter values are listed in the table below:

| Unit | Constant | Value |
| --- | --- | --- |
| Point | CPDFMeasure.CPDF_PT | pt |
| Inch | CPDFMeasure.CPDF_IN | in |
| Millimeter | CPDFMeasure.CPDF_MM | mm |
| Centimeter | CPDFMeasure.CPDF_CM | cm |
| Meter | CPDFMeasure.CPDF_M | m |
| Kilometer | CPDFMeasure.CPDFO_KM | km |
| Foot | CPDFMeasure.CPDF_FT | ft |
| Yard | CPDFMeasure.CPDF_YD | yd |
| Mile | CPDFMeasure.CPDF_MI | mi |

## 3.13.3 Measure Distance

The distance measurement tool allows your end-users to measure the distance between two points representing objects in a planar diagram, such as houses, streets, or walls. Upon selecting this tool, users only need to click on the starting and ending points with the pointer or finger to obtain the distance between the two points.

There are two types of distance measurement tools: the line segment measurement tool and the polyline measurement tool. The line segment measurement tool measures the distance between the starting and ending points, while the polyline measurement tool measures the distance between all adjacent points during the polyline drawing process and calculates the total length.




Taking the line segment measurement tool as an example, after configuring the measurement properties, you can set up the `CPDFViewerTool` to create a distance measurement tool mode through the following steps:

1. Set the tool type to annotation creation mode.
2. Create a `LineMeasureParam` object (use `PolyLineMeasureParam` if creating a polyline tool), which can be used to set properties for the distance measurement tool.

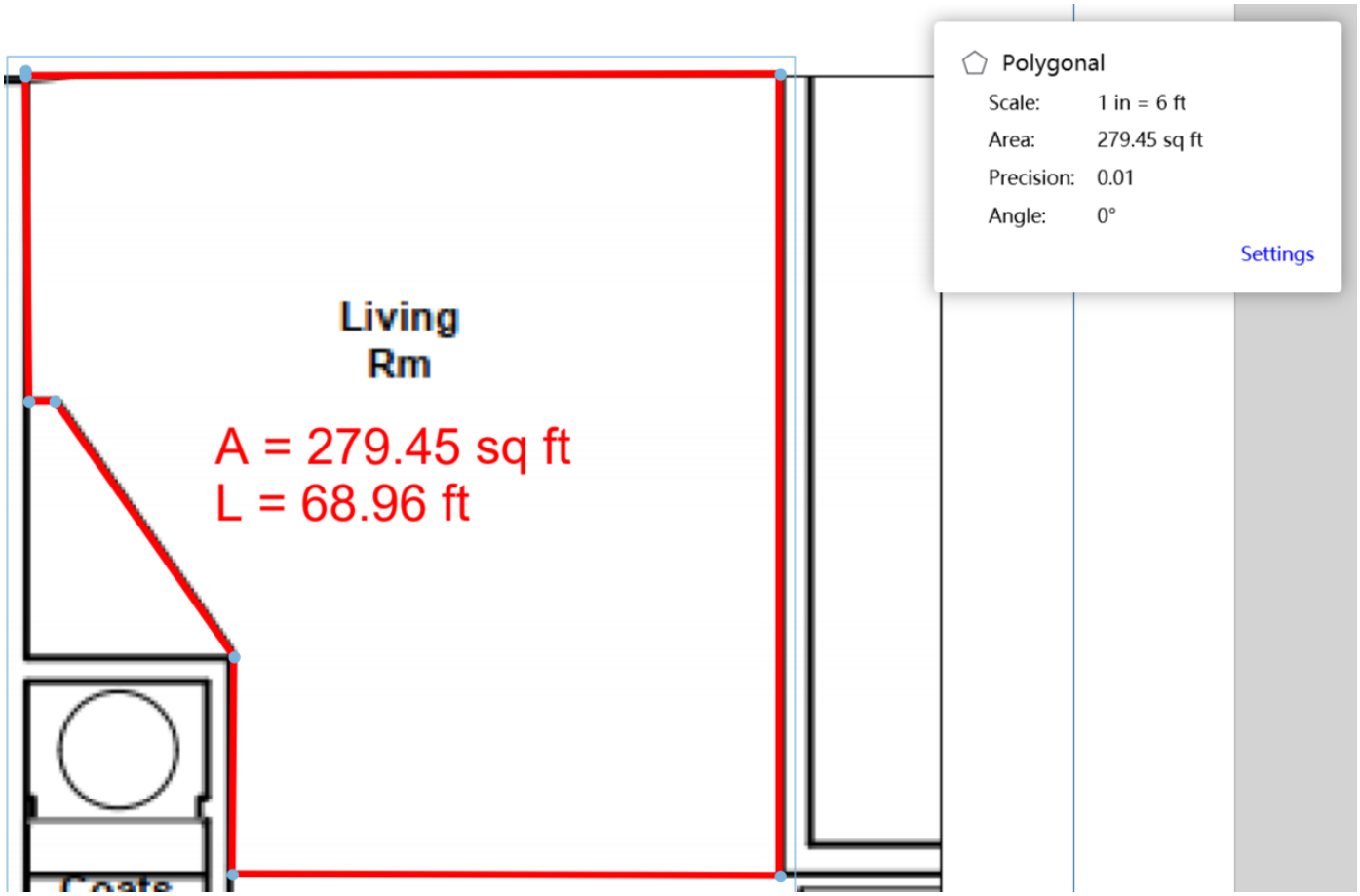
Here is an example code for creating the distance measurement tool:

```
// Initializing the CPDFViewerTool.
CPDFDocument doc = CPDFDocument.InitWithFilePath("filePath");
CPDFViewerTool tool = new CPDFViewerTool();
tool.GetCPDFViewer().InitDoc(doc);
CPDFToolManager toolManager = new CPDFToolManager(tool);

// Set the mouse mode to annotation creation mode.
toolManager.SetToolType(ToolType.CreateAnnot);
toolManager.SetCreateAnnotType(C_ANNOTATION_TYPE.C_ANNOTATION_LINE);

// Create a LineMeasureParam object to set properties for the distance measurement tool.
LineMeasureParam lineMeasureParam = new LineMeasureParam();
lineMeasureParam.CurrentType = C_ANNOTATION_TYPE.C_ANNOTATION_LINE;
lineMeasureParam.LineColor = new byte[] { 255, 0, 0, };
lineMeasureParam.LineWidth = 2;
lineMeasureParam.Transparency = 255;
lineMeasureParam.FontColor = new byte[] { 255, 0, 0, };
lineMeasureParam.FontName = "Arial";
lineMeasureParam.FontSize = 14;
lineMeasureParam.HeadLineType = C_LINE_TYPE.LINETYPE_ARROW;
lineMeasureParam.TailLineType = C_LINE_TYPE.LINETYPE_ARROW;
```

```
lineMeasureParam.measureInfo = new CPDFMeasureInfo
{
    Unit = CPDFMeasure.CPDF_CM,
    Precision = CPDFMeasure.PRECISION_VALUE_TWO,
    RulerBase = 1,
    RulerBaseUnit = CPDFMeasure.CPDF_CM,
    RulerTranslate = 1,
    RulerTranslateUnit = CPDFMeasure.CPDF_CM,
    CaptionType = CPDFCaptionType.CPDF_CAPTION_LENGTH,
};
tool.GetDefaultSettingParam().SetAnnotParam(lineMeasureParam);
```

## 3.13.4 Measure Perimeter and Area

The perimeter and area measurement tool can be used to measure the perimeter and area of a polygonal region selected by the user.




After configuring the measurement properties, you can set up the `CPDFViewerTool` to create a perimeter and area measurement tool mode through the following steps:

1. Set the mouse mode to annotation creation mode.
2. Create a `PolygonMeasureParam` object, which can be used to set properties for the perimeter and area measurement tool.

Here is an example code for creating the perimeter and area measurement tool:

```
// Set the tool type to annotation creation mode.
```

```
toolManager.SetToolType(ToolType.CreateAnnot);
toolManager.SetCreateAnnotType(C_ANNOTATION_TYPE.C_ANNOTATION_POLYGON);

// Create a PolygonMeasureParam object to set properties for the perimeter and area
measurement tool.
PolygonMeasureParam polygonMeasureParam = new PolygonMeasureParam();
polygonMeasureParam.CurrentType = C_ANNOTATION_TYPE.C_ANNOTATION_POLYGON;
polygonMeasureParam.LineColor = new byte[] { 255, 0, 0, };
polygonMeasureParam.LineWidth = 2;
polygonMeasureParam.Transparency = 255;
polygonMeasureParam.FontColor = new byte[] { 255, 0, 0, };
polygonMeasureParam.FontName = "Arial";
polygonMeasureParam.FontSize = 14;
polygonMeasureParam.measureInfo = new CPDFMeasureInfo
{
    Unit = CPDFMeasure.CPDF_CM,
    Precision = CPDFMeasure.PRECISION_VALUE_TWO,
    RulerBase = 1,
    RulerBaseUnit = CPDFMeasure.CPDF_CM,
    RulerTranslate = 1,
    RulerTranslateUnit = CPDFMeasure.CPDF_CM,
    CaptionType = CPDFCaptionType.CPDF_CAPTION_LENGTH |
CPDFCaptionType.CPDF_CAPTION_AREA,
};
tool.GetDefaultSettingParam().SetAnnotParam(polygonMeasureParam);
```

## 3.13.5 Adjust Existing Measurement Annotations

For existing measurement annotations, you can reset measurement-related information such as scale, units, and precision without redrawing the annotation.

Taking the line segment measurement tool as an example, you can adjust the properties of an existing measurement annotation through the following steps:

1. Get the `CPDFAnnotation` object from `CPDFViewerTool`, check the `Type` parameter of the `CPDFAnnotation` object representing the line measurement tool. If it is `C_ANNOTATION_TYPE.C_ANNOTATION_LINE`, convert the `CPDFAnnotation` type of the line measurement tool to a `CPDFLineAnnotation` object.
2. Use the `IsMersured` method to determine if the object is a measurement tool. If true, call the `GetDistanceMeasure` method to obtain the corresponding `CPDFDistanceMeasure` object.
3. Retrieve the `MeasureInfo` from the `CPDFDistanceMeasure` object and assign values to it.
4. Call the `SetMeasureInfo`, `SetMeasureScale`, and `UpdateAnnotMeasure` methods of `CPDFDistanceMeasure` to complete parameter updates.
5. Finally, call the `UpdateAp` method of `CPDFAnnotation` to update the annotation appearance, and then call `UpdateAnnotFrame` method of `CPDFViewer` to update the drawing.

Here is an example code for adjusting existing measurement annotations:

```
BaseAnnot baseAnnot = tool.GetCacheHitTestAnnot();
```

```
CPDFAnnotation annot = baseAnnot.GetAnnotData().Annot;
switch (annot.Type)
{
    case C_ANNOTATION_TYPE.C_ANNOTATION_LINE:
        {
            // Convert CPDFAnnotation type of the line measurement tool to
CPDFLineAnnotation object.
            CPDFLineAnnotation lineAnnot = (CPDFLineAnnotation)annot;
            // Check if the object is a measurement tool.
            if (lineAnnot.IsMersured())
            {
                // Get the CPDFDistanceMeasure object corresponding to the annotation.
                CPDFDistanceMeasure lineMeasure = lineAnnot.GetDistanceMeasure();

                // Set new measurement-related properties.
                CPDFMeasureInfo measureInfo = lineMeasure.MeasureInfo;
                measureInfo.Precision = measureSetting.GetMeasureSavePrecision();
                measureInfo.RulerBase = (float)measureSetting.RulerBase;
                measureInfo.RulerBaseUnit = measureSetting.RulerBaseUnit;
                measureInfo.RulerTranslate = (float)measureSetting.RulerTranslate;
                measureInfo.RulerTranslateUnit = measureSetting.RulerTranslateUnit;

                // Complete parameter updates.
                lineMeasure.SetMeasureInfo(measureInfo);
                lineMeasure.SetMeasureScale(
                    measureInfo.RulerBase,
                    measureInfo.RulerBaseUnit,
                    measureInfo.RulerTranslate,
                    measureInfo.RulerTranslateUnit);
                lineMeasure.UpdateAnnotMeasure();

                // Update annotation appearance.
                lineAnnot.UpdateAp();

                // Update drawing.
                tool.GetCPDFViewer().UpdateAnnotFrame();
            }
        }
        break;

    case C_ANNOTATION_TYPE.C_ANNOTATION_POLYLINE:
    case C_ANNOTATION_TYPE.C_ANNOTATION_POLYGON:
        // To do.
        break;
}
```

## 3.14 Optimization

## 3.14.1 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.

## 3.14.2 Compress and Optimize PDF Files

The compression accuracy is controlled by setting the `imageQuality` parameter of the `CompressFile_Init` interface, and the `getPageIndexDelegate` parameter allows us to monitor the compression progress in real time.

Below is an example code of compression and optimization of PDF files:

```
CPDFDocument document = CPDFDocument.InitWithFilePath("filePath");
//Image quality.
float imageQuality = 40;

//Gets a callback for the progress of document compression.
CPDFDocument.GetPageIndexDelegate getPageIndex = GetPageIndex;

//Compress a document.
IntPtr compressPtr = document.CompressFile_Init(imageQuality, getPageIndex);
bool result = document.CompressFile_Start(compressPtr, "saveFilePath");

//If you need to uncompress, you can call the following method.
document.CompressFile_Cancel(compressPtr);
```

# 4 Support

## 4.1 Reporting Problems

Thank you for your interest in ComPDFKit PDF SDK, an easy-to-use but 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 Windows, 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/windows/overview
- API Reference: https://developers.compdf.com/guides/pdf-sdk/windows/api-reference/html/3a1f08b6-6ac4-f8b5-bad1-a31c98e96105.htm
- Code Examples: https://www.compdf.com/guides/pdf-sdk/windows/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