ComPDFKit PDF SDK

Developer Guides

Comprehensive PDF SDK for Developers

Contents

1 Overview

ComPDFKit PDF SDK for .NET Core 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 for .NET Core

ComPDFKit for .NET Core is cross-platform, now supports Windows, Linux, macOS.

1.2 Key Features

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.

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.

2 Get Started

2.1 .NET Core PDF Library for Windows

2.1.1 Requirements

- Visual Studio

  Make sure that the `.NET Core cross-platform development` workload and `MSBUILD` are part of your installation.

  This guide will use Visual Studio 2022. If you would like to use the NuGet integration for Windows x64 please make sure you have Visual Studio 2017 or later.

  Note: ComPDFKit PDF SDK is multi-targeting. Target Frameworks: .NET Core 2.1+, .NET Standard 2.0, .NET 5, .NET 6, .NET 7, .NET 8.

- ComPDFKit 's .NET Core PDF Library for Windows:

  Please contact us to get the ComPDFKit PDF SDK for .NET Core.

- ComPDFKit PDF SDK license

  Commercial license keys are required for use in production environments. If you do not have a valid license key, please contact us to obtain the license key.

  License keys are uniquely generated. Please make sure that it is not publicly available (e.g. in your public GitHub).

2.1.2 Run the Samples

1. Open the solution file 'Example. sln' in Visual Studio.
2. Select a sample project in Solution Explorer and set it as the startup project. CS and VB examples are currently available.
3. Build and run the example.

Or you can open the command line tool in the project folder and run the example through the command:

```
dotnet run
```

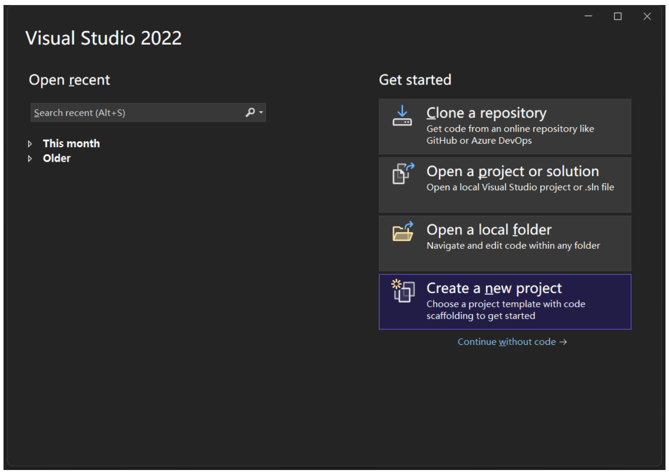
2.1.3 Integrate into Your Application

You can follow the manual or nuget integration described below. This section will help you build your first ComPDFKit application. If you can open, save, and close PDFDoc, you can easily integrate the rest of the ComPDFKit PDF SDK.

2.1.3.1 Integrate Manually

1. Open a new instance of Visual Studio 2022and create a new .NET Core console application project (`File > New > Project...`). You can find this under the Visual C# menu.

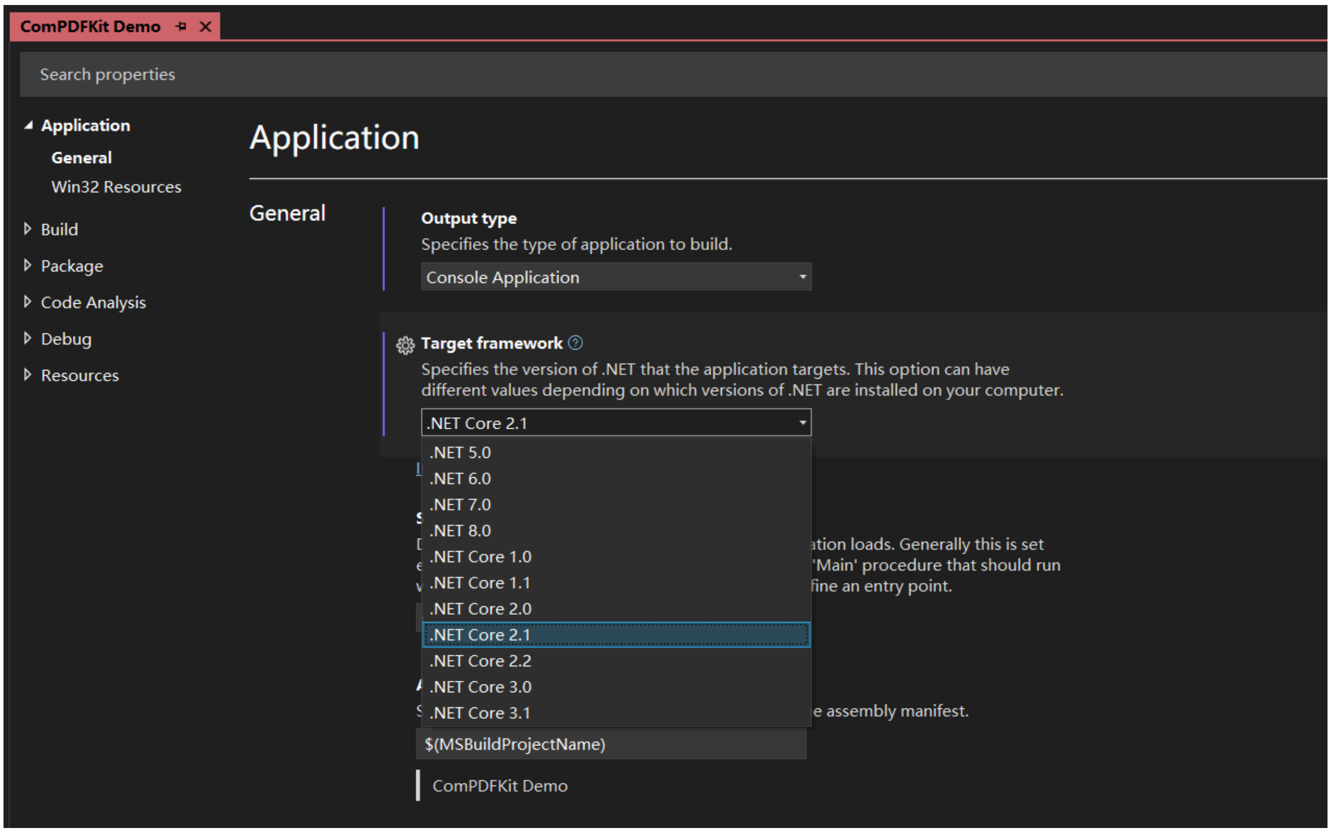
   Click on OK and allow the IDE to create the project.


Visual Studio 2022
Open recent
Search recent (Alt+S)
This month
Older
Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

2. Find the Solution Explorer on the right side of the screen. Select the project and press `Alt + Enter`. This will open the properties tab.

   Alternatively, you can right click on the project and find the properties option.

   Select **.NET Core 2.1 (or above)** as the target framework for your application.



Output type
Specifies the type of application to build.
Console Application

Target framework
Specifies the version of .NET that the application targets. This option can have different values depending on which versions of .NET are installed on your computer.
.NET Core 2.1

.NET 5.0
.NET 6.0
.NET 7.0
.NET 8.0
.NET Core 1.0
.NET Core 1.1
.NET Core 2.0
.NET Core 2.1
.NET Core 2.2
.NET Core 3.0
.NET Core 3.1

ation loads. Generally this is set
'Main' procedure that should run
fine an entry point.

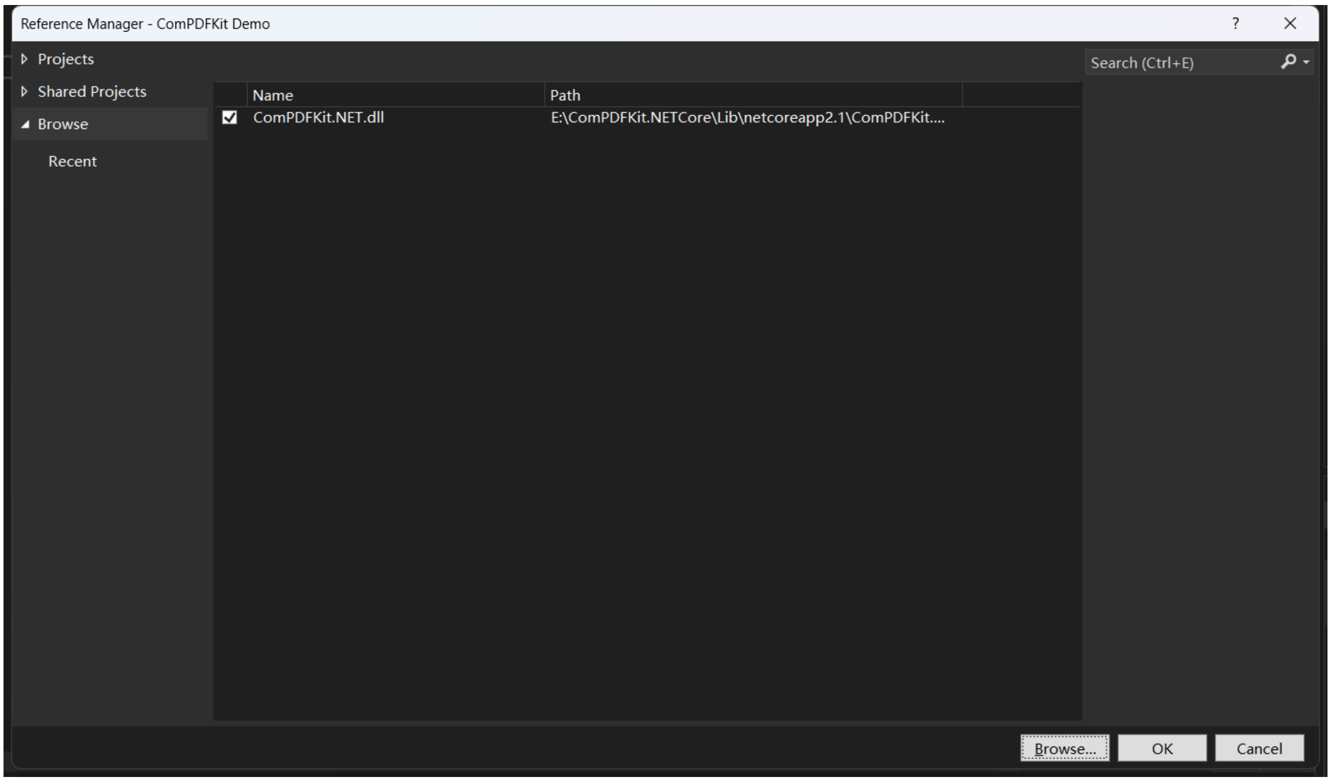
e assembly manifest.

$(MSBuildProjectName)

ComPDFKit Demo

If you are using a Windows x64 machine for your .NET Core development, you can use NuGet package manager to get the ComPDFKit PDF SDK. Otherwise, please integrate the SDK manually.

3. Right-click "Dependencies" and click "Add Project Reference", which will open the "Reference Manager" dialog box. Click the option at the bottom of the dialog box, find the corresponding "ComPDFKit.NET.dll", select it and click Add.



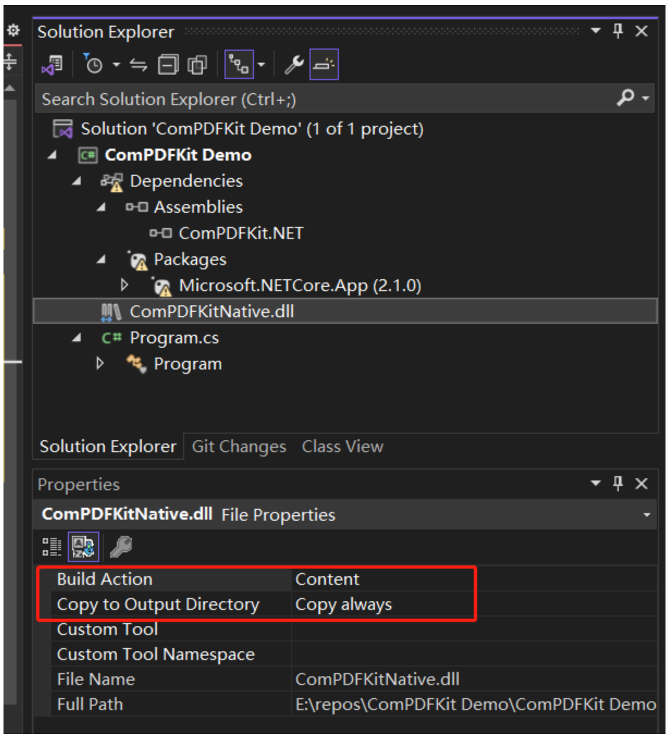
	Name	Path
✓	ComPDFKit.NET.dll	E:\ComPDFKit.NETCore\Lib\netcoreapp2.1\ComPDFKit....

4. In the Solution Explorer, select the project and press `Shift + Alt + A`. With this you can Add an Existing Item.

Alternatively, you can right click on the project and find the `Add an existing item...` option under the `Add` submenu.

Navigate to the library location again, select the file type as `All Files (*.*)` and select `ComPDFKitNative.dll`. Click Add.

5. Select ComPDFKit.NET.dll in the solution explorer. A properties window should appear below. In it, change the Build Action setting to `Content` and the Copy to Output Directory setting to `Copy always`. To avoid errors, use the drop-down menus available for those fields.



Build Action	Content
Copy to Output Directory	Copy always
Custom Tool	
Custom Tool Namespace	
File Name	ComPDFKitNative.dll
Full Path	E:\repos\ComPDFKit Demo\ComPDFKit Demo

2.1.3.2 Integrate with NuGet

1. Perform the first 2 steps of integrating manually.
2. Right click on project Dependencies and click on `Manage NuGet Packages...`. This will open the NuGet Package Manager
3. Click on the `Browse` tab near the top of the package manager. In the search bar enter:

```
ComPDFKit.NetCore
```

4. Select the `ComPDFKit.NetCore` package by PDF Tecnologies Inc. and click on the `Install` button in the panel with the package information. If you're prompted or an external dialog is opened for confirmation, click on `Ok`.

   After installation, you will see the reference to the package under Dependencies in Solution Explorer.

2.1.3.3 Create PDF Document

We have completed all the preparation steps. Now let's use the ComPDFKit PDF SDK to create a PDF file with a blank page, and replace your Program. cs file with the following code. Note that you need to replace your license in the 'LicenseVerify()' method.

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

namespace ComPDFKit_Demo
{
    public class Program
    {
        private static bool LicenseVerify()
        {
            if (!CPDFSDKVerifier.LoadNativeLibrary())
                return false;
            LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your
license here");
            return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
        }

        public static void Main()
        {
            LicenseVerify();
            CPDFDocument document = CPDFDocument.CreateDocument();
             // Insert to the first page
            int pageIndex = 0;
            int pageWidth = 595;
            int pageHeight = 842;
            //The InsertPage method can specify an image path. When the image path is
empty, a blank page will be inserted.
            document.InsertPage(pageIndex, pageWidth, pageHeight, "");
            document.WriteToFilePath("new_file.pdf");// Save the entire document
object to the current path.
            Console.WriteLine("Done. Results saved in new_file.pdf");
        }
    }
}
```

After running the program, please check the output file generated by the program. You will find that a PDF file with blank pages has been generated, and by default, it should be in the bin/Debug directory where your project files are located.

2.2 .NET Core PDF Library for Linux

2.2.1 Requirements

- NET Core SDK
  Note: ComPDFKit PDF SDK is multi-targeting. Target Frameworks : .NET Core 2.1+, .NET Standard 2.0, .NET 5, .NET 6, .NET 7, .NET 8。
- ComPDFKit 's .NET Core PDF Library for Linux:
  Please contact us to obtain the ComPDFKit PDF SDK for NET Core.
- ComPDFKit PDF SDK license

  Commercial license keys are required for use in production environments. If you do not have a valid license key, please contact us to obtain the license key.

  License keys are uniquely generated. Please make sure that it is not publicly available (e.g. in your public GitHub).

2.2.2 Run the Samples

Enter the sample folder and run the example from the command line:

- Input/ Run. sh all to run all examples.
- Input/ Run. shRun a specific example. For example/ Run. sh BatesTest.

If you receive an error message bash:/ Test. sh: Permission denied, you need to enter the following command:

```
chmod+x run .sh
```

2.2.3 Integrate into Your Application

You can follow the manual or nuget integration described below for operation. This section will help you build your first ComPDFKit application. If you can open, save, and close PDFDoc, you can easily integrate the rest of the ComPDFKit PDF SDK.

2.2.3.1 Integrate Manually

1. Create a new project named ComPDFKit Demo_ through the console:

   ```
   mkdir ComPDFKitDemo
   cd ComPDFKitDemo
   dotnet new console -o "ComPDFKit Demo"
   ```

2. Copy the ComPDFKitNative.so file and ComPDFKit Copy the NET.dll file to the project folder.
3. Add the following code to the. csproj file:

```
<ItemGroup>
 <Reference Include="ComPDFKit.NET.dll">
   <HintPath>ComPDFKit.NET.dll</HintPath>
 </Reference>
</ItemGroup>

<ItemGroup>
    <Content Include="ComPDFKitNative.so">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
</ItemGroup>
```

Your final ComPDFKit Demo.csproj file should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
        <Reference Include="ComPDFKit.NET.dll">
            <HintPath>ComPDFKit.NET.dll</HintPath>
        </Reference>
    </ItemGroup>

    <ItemGroup>
        <Content Include="ComPDFKitNative.so">
            <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        </Content>
    </ItemGroup>

</Project>
```

2.2.3.2 Integrate with NuGet

1. Create a new project named ComPDFKit Demo_ through the console:

```
mkdir ComPDFKitDemo
cd ComPDFKitDemo
dotnet new console -o "ComPDFKit Demo"
```

2. Enter the project folder and install ComPDFKit Core NuGet package:

```
dotnet add package ComPDFKit.NetCore
```

2.2.3.3 Create PDF Document

We have completed all the preparation steps. Now let's use the ComPDFKit PDF SDK to create a PDF file with a blank page, and replace your Program. cs file with the following code. Note that you need to replace your license in the 'LicenseVerify()' method.

```
using ComPDFKit.NativeMethod;
using ComPDFKit.PDFDocument;
using System.Reflection.Metadata;

namespace ComPDFKit_Demo
{
    public class Program
    {
        private static bool LicenseVerify()
        {
            if (!CPDFSDKVerifier.LoadNativeLibrary())
                return false;
            LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your
license here");
            return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
        }

        public static void Main()
        {
            LicenseVerify();
            CPDFDocument document = CPDFDocument.CreateDocument();
            int pageIndex = 0;
            int pageWidth = 595;
            int pageHeight = 842;
            document.InsertPage(pageIndex, pageWidth, pageHeight
      , "");
            document.WriteToFilePath("new_file.pdf");
            Console.WriteLine("Done. Results saved in new_file.pdf");
        }
    }
}
```

Now you can run the program from the command line:

```
dotnet run
```

Now you will find the 'new_file. pdf' file in the program output directory, which is a PDF file with a blank page.

2.3 .NET Core PDF Library for macOS

2.3.1 Requirements

- .NET Core SDK

  Note: ComPDFKit PDF SDK is multi-targeting. Target Frameworks : .NET Core 2.1+, .NET Standard 2.0, .NET 5, .NET 6, .NET 7, .NET 8。

- ComPDFKit 's .NET Core PDF Library for macOS:
  Please contact us to obtain the ComPDFKit PDF SDK for NET Core.

- ComPDFKit PDF SDK license

  Commercial license keys are required for use in production environments. If you do not have a valid license key, please contact us to obtain the license key.

  License keys are uniquely generated. Please make sure that it is not publicly available (e.g. in your public GitHub).

2.3.2 Run the Samples

Enter the sample folder and run the example from the command line:

- Input/ Run. sh all to run all examples.
- Input/ Run. shRun a specific example. For example/ Run. sh BatesTest.

If you receive an error message bash:/ Test. sh: Permission denied, you need to enter the following command:

```
chmod +x run.sh
```

2.3.3 Integrate into your Application

You can follow the manual or nuget integration described below for operation. This section will help you build your first ComPDFKit application. If you can open, save, and close PDFDoc, you can easily integrate the rest of the ComPDFKit PDF SDK.

2.2.3.1 Integrate Manually

1. Create a new project named ComPDFKit Demo_ through the console

   ```
   mkdir ComPDFKitDemo
   cd ComPDFKitDemo
   dotnet new console -o "ComPDFKit Demo"
   ```

2. Copy the libComPDFKit.dylib file and the ComPDFKit.Desk.Core.dll file to the project folder.

3. Add the following code to the .csproj file:

```
<ItemGroup>
    <Reference Include="ComPDFKit.NET.dll">
        <HintPath>ComPDFKit.Desk.Core.dll</HintPath>
    </Reference>
</ItemGroup>

<ItemGroup>
    <Content Include="libComPDFKit.dylib">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
</ItemGroup>
```

Your final ComPDFKit Demo.csproj file should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
        <Reference Include="ComPDFKit.Desk.Core.dll">
            <HintPath>ComPDFKit.Desk.Core.dll</HintPath>
        </Reference>
    </ItemGroup>

    <ItemGroup>
        <Content Include="libComPDFKit.dylib">
            <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        </Content>
    </ItemGroup>

</Project>
```

2.2.3.2 Integrate with NuGet

1. Create a new project named ComPDFKit Demo in the console:

```
mkdir ComPDFKitDemo
cd ComPDFKitDemo
dotnet new console -o "ComPDFKit Demo"
```

2. Go to the project folder and install the ComPDFKit.Core NuGet package:

```
dotnet add package ComPDFKit.NetCore
```

2.3.3.3 Create PDF Document

We have completed all the preparation steps. Now let us use ComPDFKit PDF SDK to create a PDF file with a blank page. Replace your Program.cs file with the following code. Note: You need to replace your license into the `LicenseVerify()` method.

```
using ComPDFKit.NativeMethod;
using ComPDFKit.PDFDocument;
using System.Reflection.Metadata;

namespace ComPDFKit_Demo
{
    public class Program
    {
        private static bool LicenseVerify()
        {
            if (!CPDFSDKVerifier.LoadNativeLibrary())
                return false;
            LicenseErrorCode verifyResult = CPDFSDKVerifier.LicenseVerify("Input your
license here");
            return (verifyResult == LicenseErrorCode.E_LICENSE_SUCCESS);
        }

        public static void Main()
        {
            LicenseVerify();
            CPDFDocument document = CPDFDocument.CreateDocument();
            int pageIndex = 0;
            int pageWidth = 595;
            int pageHeight = 842;
            document.InsertPage(pageIndex, pageWidth, pageHeight
      , "");
            document.WriteToFilePath("new_file.pdf");
            Console.WriteLine("Done. Results saved in new_file.pdf");
        }
    }
}
```

Now you can run the program from the command line:

```
dotnet run
```

You will now find the `new_file.pdf` file in the program output directory, which is a PDF file with a blank page.

2.4 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.

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. Call `CPDFFont.InitFont();` to initialize the font library. This function only needs to be called once during the project's lifecycle.
2. Retrieve the names of all font families in the system.
3. Select the font you need and obtain the style names for the font family.
4. After selecting the style name, obtain the PostScript name of the font based on the font family name and style name.
5. The PostScript name can then be used to set the font.

This example shows how to use font management:

```
// Initialize the font library
CPDFFont.InitFont();

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 Annotations

3.2.1 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.

3.2.2 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

3.2.3 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.
          }
        }
    }
}
```

3.2.4 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();
```

3.2.5 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();
```

3.2.6 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.

3.2.7 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();
```

3.2.8 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.

3.2.9 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 Forms

3.3.1 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.

3.3.2 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

3.3.3 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 });
```

3.3.4 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;
        }
    }
}
```

3.3.5 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();
```

3.3.6 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();
```

3.3.7 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.

3.4 Document Editor

3.4.1 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.

3.4.2 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);
```

3.4.3 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);
```

3.4.4 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);
```

3.4.5 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());
```

3.4.6 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);
```

3.4.7 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);
```

3.4.8 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);
```

3.5 Security

3.5.1 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.

3.5.2 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");
```

3.5.3 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();
```

3.5.4 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();
```

3.5.5 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();
```

3.6 Redaction

3.6.1 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.

3.6.2 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 PDF 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();
```

3.7 Watermark

3.7.1 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.

3.7.2 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();
```

3.7.3 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();
```

3.7.4 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();
```

3.8 Conversion

3.8.1 PDF/A

ComPDFKit PDF 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.

3.9 Compare Documents

3.9.1 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.

3.9.2 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.

3.9.3 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.

3.10 Digital Signatures

3.10.1 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.

3.10.2 Concepts of Digital Signatures

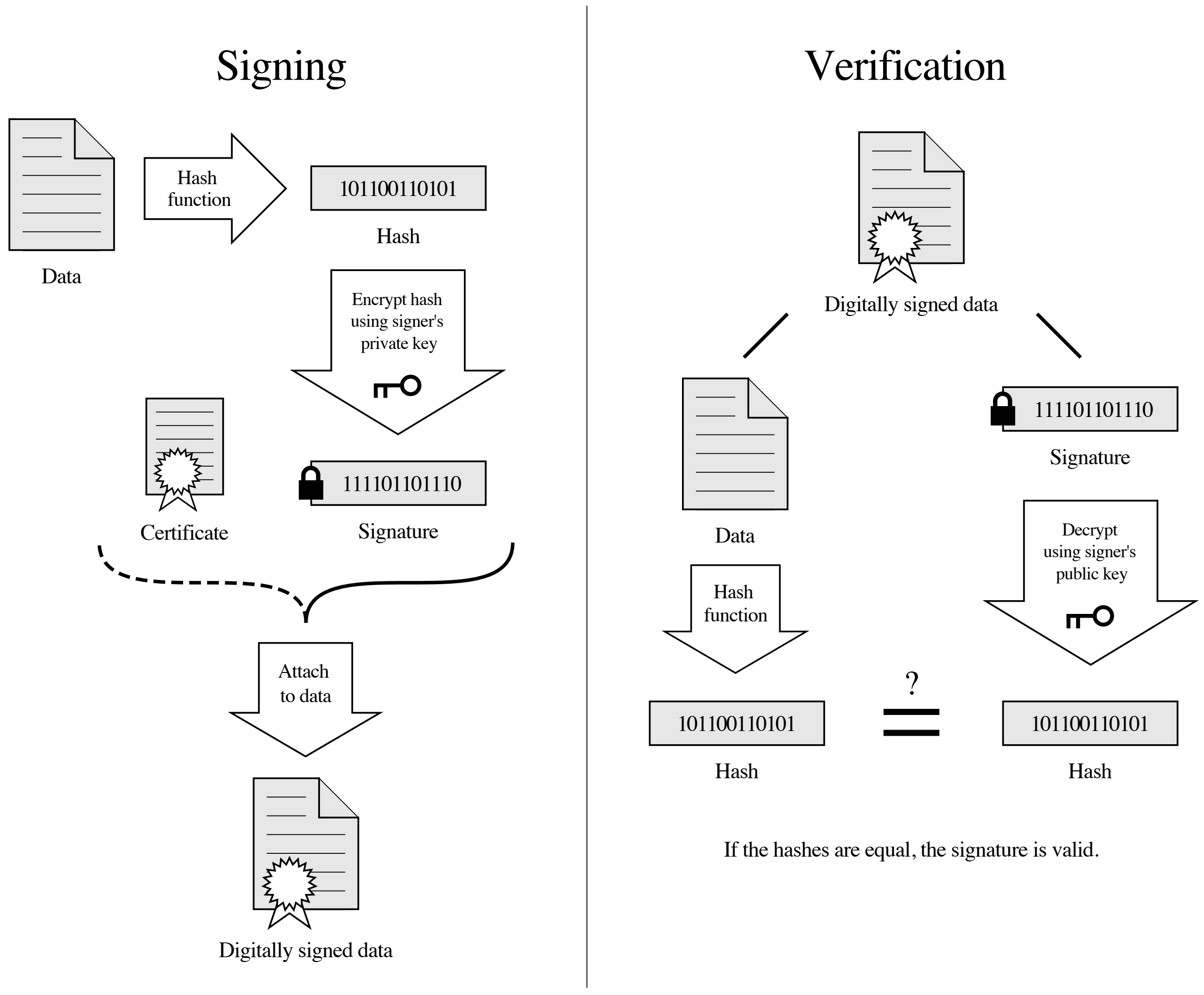
How Digital Signatures Work

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
```

3.10.3 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);
```

3.10.4 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 PDF 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;
}
```

3.10.5 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.

3.10.6 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.
}
```

3.10.7 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.
          }
      }
}
```

3.10.8 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 PDF 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());
```

3.10.9 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);
```

3.10.10 Troubleshooting

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.

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