1) Overview and recommended stack

* **Goal:** View any PDF, add comments/annotations, select text, copy images, draw freehand, fill forms, insert text/images, and share securely (metadata scrub, encrypt, sign).# Build
* **Platform:** Windows 11, offline only.
* **Language/UI:** .NET 8 + WPF (beginner‑friendly, stable).
* **PDF engine:** PDFium via PdfiumViewer.Core (battle‑tested viewer/selection/forms).
* **PDF creation:** QuestPDF (templates or simple “image to PDF” build).
* **Security:** qpdf (encrypt/sign), ExifTool (metadata scrub), optional Ghostscript (flatten/sanitize).
* **Name:** PdfSafe (you can rename later).

Direct path for new programmers: stick to PdfiumViewer for viewing/selection/forms; keep “content editing” to insert text/images as overlays; use reliable external tools for security.

2) Prerequisites and installs

* **Core tools:**
  + **.NET SDK:** 8.x
  + **Git:** Latest
  + **Winget:** Built into Windows 11
* **External utilities (bundled locally, offline):**
  + **qpdf:** Encryption, permissions, signing Metadata scrub
  + **ExifTool:** - **Ghostscript (optional):** Flatten/sanitize
* **PDFium native binaries:** x64, placed under your project’s native runtimes folder.

Use this once in PowerShell (Run as Administrator):

winget install Microsoft.DotNet.SDK.8 winget install GnuPG.QPDF winget install PhilHarvey.ExifTool winget install Ghostscript.GPL

If winget prompts you to agree, accept and continue. You can also manually download portable zips and place exe files under /tools.

3) Repository prompt)

Use this scaffold (Cursor to create the solution, folders, and projects. Paste into Cursor and run it step‑by‑step.

Create a new git repo named PdfSafe with a .NET 8 WPF app and supporting class libraries. Use MVVM. Solution layout: - src/PdfSafe.App (WPF shell, MVVM, DI) - src/PdfSafe.Core (domain models, services interfaces, undo/redo) - src/PdfSafe.Pdfium (PDFium integration: open, render, selection, forms) - src/PdfSafe.Annotations (annotation/ink overlay layer and export) - src/PdfSafe.Creation (QuestPDF document generator) - src/PdfSafe.Securitypdf/exiftool, secure (wrappers for q-share pipeline) - src/PdfSafe.Tests (xUnit tests) -iftool, ghostscript tools/ (qpdf, eximes/win-x64/native binaries) - runt/ (PDFium native dlls) - scripts/ (dev-setup.ps1, encrypt-pdf.ps1, sign-pdf.ps1, scrub-metadata.ps1) - README.md Initialize Directory.Build.props to enforce LangVersion=latest, nullable enable, treat warningseditorconfig with as errors. Add . consistent formattingGet dependencies.



Now add Nu```text Add these (Cursor prompt):

NuGet packages:

* PdfiumViewer.Core (for PDF rendering/selection/forms)
* PdfiumViewer.Native.x64 (for PDFium native runtime)
* QuestPDF
* CommunityToolkit.Mvvm
* Microsoft.Extensions.DependencyInjection
* Microsoft.Windows.Compatibility (cert and crypto)
* xunit, xunit.runner.visualstudio, FluentAssertions helpers if needed (in Tests)

> After restore, verify that Pdfium native dlls are copied to bin at build. If not, place official PDFium binaries into runtimes/win-x64/native. --- ## 4) App shell and MVVM wiring (Cursor prompt + code) Prompt: ```text In PdfSafe.App, create a WPF MainWindow with: - Left: page thumbnails - Center: viewer canvas hosted via WindowsFormsHost for PdfiumViewer control - Right: actions panel with buttons: Open, Save, Select, Highlight, Note, Ink, Text Box, Image, Form Fill, Redact, Encryptly. Use MVVM (Community, Sign, Share SecureToolkit.Mvvm). Create ICommand properties MainViewModel with for each action. Wire DI with Microsoft.Extensions.DependencyInjection in App.xaml.cs.



Example XAML (simplified):

Width="\*"/> Width="280"/> FormsHost Grid.ColumnHost" /> ="{Binding Encrypt



CodeiumViewer (Cursor‑behind to host Pdf prompt):

Add a PdfiumViewer control into ViewerHost in MainWindow.PdfViewer WinForms.xaml.cs. Expose it to MainViewModel via an IPdfViewAdapter interface so the VM can call OpenDocument, RenderThumbnail, SetZoom, etc.



Sample:

using PdfiumViewer; using System.Windows.Forms.Integration; public partial class MainWindow : Window { private PdfViewer \_pdfViewer; public MainWindow() { InitializeComponent(); \_pdfViewer = new PdfViewer { Dock = System.Windows.Forms.DockStyle.Fill, ShowToolbar = false, ShowBookmarks = false }; (ViewerHost.Child as System.Windows.Forms.Control)?. ViewerHost.ChildDispose(); = \_pdfViewer; DataContextChanged += (\_, \_\_) => { if (DataContext is IPdfViewAdapterConsumer c) c.Attach(new PdfViewAdapter(\_pdfViewer)); interface IPdfView }; } } publicAdapter { void Open(string path); Bitmap RenderPage(int page, int dpi); int PageCount { get; } void SetZoom(double factor); } public class PdfViewAdapter : IPdfViewAdapter { private readonly PdfViewer \_viewer; public Pdf viewer) { \_viewerViewAdapter(PdfViewer = viewer; } public void Open(string path) => \_viewer.Document = PdfDocument.Load(path); public Bitmap RenderPage(int page, int dpi) => \_viewer.Document.Render(page, dpi, dpi, true); public int PageCount => \_viewer.Document?.PageCount ?? 0; public void SetZoom(double factor) => \_viewer.ZoomMode = PdfViewerZoomMode.FitBest; // keep simple for v1 } public interface IPdfViewAdapterConsumer { void Attach(IPdfViewAdapter adapter); }



5) Open, thumbnails, select text, copy images (Cursor prompts + code)

Prompt:

Implement OpenCommand: - Show file picker. - Open via PdfViewAdapter.Open(path). - Generate pages, RenderPage thumbnails: iterate(page, 96 dpi), convert to ImageSource, and bind to Thumbnails. - Update current page selection. Implement Select and Copy: - Use PdfiumViewer’s built-in selection/copy for text: Ctrl+A/Ctrl+C on the viewer; expose CopySelectedText() via adapter if needed. - For images: on right-click, hit-test by rendering a high-dpi crop around mouse and offer "Copy image region".



Thumbnail generation (example):

public ObservableCollection Thumbnails { get; } = new(); private void LoadThumbnails(IPdfViewAdapter.Clear(); for (int i = 0; i < pdf.PageCount; i pdf) { Thumbnails++) { using var bmp = pdf.RenderPage(i, 96); Thumbnails.Add(ConvertToImageSource(bmp)); } } private static ImageSource ConvertToImageSource(Bitmap bitmap) { using var ms = new MemoryStream(); bitmap.Save(ms, System.Drawing.Imaging.ImageFormat.Png); ms.Position = 0; var img = new BitmapImage(); img.BeginInit(); img.CacheOption = BitmapCacheOption.OnLoad; img.StreamSource = ms; img.EndInit(); img.Freeze(); return img; }

6) Annotations: highlight, note, freehand ink (CursorPrompt:

Create an overlay layer on top of the viewer for annotations: - Highlight: store prompts + code) (page, quad points or rectangles). Draw semi-transparent yellow overlay. - Note: store (page, position, text). Show icon; display popup on click. - Ink: use WPF InkCanvas layered above the viewer in Grid; on Save, convert strokes to vector annotations as PDF paths. Persist annotations: - For highlight and note, export appearance streams via Pdfium if available As, flatten the; otherwise, on Save overlay by re-rendering the page and compositing overlays, then rebuild a new PDF with QuestPDF (safe fallback).



WPF overlay setup:

Ink mode toggle:

public void EnterInkMode() => InkLayer.EditingMode = InkCanvaspublic void ExitEditingMode.Ink; InkMode() => Ink = InkCanvasEditingLayer.EditingModeMode.None;



Flattening ink (safe fallback):

* **Approach:** For each page, render at high DPI (e.g., onto a System.Drawing 200–300), draw strokes.Bitmap using stroke geometry translated into page pixels, then rebuild PDF pages from those selectable text, but safe if you images (lossy of need to “burn in” annotations).

7) Form filling (AcroForm) and basic content insert

Prompt:

Add IFormService in PdfSafe.Pdfium: - List fields: name, type, page, rect, value. - Get/Set values for text, checkbox, radio, combo. - Add "Flatten forms" (render appearances into content and fields). In the remove interactive UI: - Form Mode toggles field highlighting; clicking a field focuses it; pressing Save performs incremental save.



Basic content insert (overlay approach):

* **Insert text:** Place a box; capture string and font; on Save As, render page and draw the text onto) OR use Pdfium text the bitmap (safe object API if exposed by wrapper (many wrappers don’t expose low‑level writes; safe route is flattening).
* **Insert image:** Choose an image; place/scale; on Save As, composite the image onto the page bitmap or embed via QuestPDF rebuild.

For a new programmer, start with “flatten on Save As” for content inserts/annotations. Keep “Save” as a project file (.pfs) that stores overlay data; “Save As (Flattened PDF)” generates a final PDF for sharing.

8) Security: scrub, encrypt, sign, verify (Cursor prompts + code)

Place tools:

* **tools/qpdf/qpdf.exe**
* **tools/exiftool/exiftool.exe**
* **tools/ghostscript/gswin64c.exe** (optional)

Prompt ISecurityService:

Create in PdfSafe.Security that wraps externalStartInfo. Methods tools via Process(input, output) : - ScrubMetadata- Encrypt(input, output, password, allowPrint, allowModify) - Sign(input, output, pfxPath, pfxPassword, reason, location) - Verify, terms) -> boolRedactions(input Implement "Share Securely" pipeline: 1) Save As Flattened (if overlays/redactions exist) 2) Scrub metadata 3) Encrypt with AES-256 (owner=user pw) 4) Optional: Sign with PFX 5) Verify: open output and grep for forbidden terms (simple text search), block if found.



C# wrappers:

static int Run(string exe, string args, string workingDir) { var p = new Process { StartStartInfo(exe, argsInfo = new Process) { WorkingDirectory = workingDir, = false, UseShellExecute CreateNoWindow = true, RedirectStandardOutput = true, RedirectStandardError = true } }; p.Start(); p.WaitForExit(); return p.ExitCode; } // ExifTool: strip metadata public void ScrubMetadata(string input, string output) { File.Copy(input, output, true); Run(Path.Combine(ToolsDir, "exiftool", "exiftool.exe"), $"-overwrite\_original -all= \"{output}\"", ToolsDir); } // qpdf: encrypt AES-256 public void Encrypt(string input, string output, string password, bool allowPrint = false, bool allowModify = false) { var perm = $"{(allowPrint ? "--print=full " : "--print=none ")}{(allowModify ? "--modify=all " : "--modify=none ")}"; Run(Path.Combine(ToolsDir, "qpdf", "qpdf.exe"), $"--encrypt {password} {password} 256 -- {perm}\"{input}\" \"{output}\"", ToolsDir); } // qpdf: sign with PFX public void Sign(string input, string output, string pfxPath, string pfxPassword, string reason, string location) { Run(Path.Combine(ToolsDir, "qpdf", "qpdf.exe"), $"--sign \"{input}\"sign-p12=\"{pfxPath \"{output}\" --}\" --sign-p12-password=\"{pfxPassword}\" --sign-reason=\"{reason}\" --sign-location=\"{location}\"", ToolsDir); }



Create a self‑signed certificate (one‑time, PowerShell):

$cert = New-SelfSignedCertificate -Type CodeSigningCert -Subject "CN -KeyExportPolicy=PdfSafe Family" Exportable -CertStoreLocation Cert:\CurrentUser\My $pwd = ConvertTo-SecureString -String "YourStrongPfxPassword" -Force -AsPlainText Export-PfxCertificate -Cert $cert -FilePath "$env:USERPROFILE\PdfSafeFamily.pfx" -Password $pwd



“Share Securely” flow (VM pseudo):

var tmp1 = Path.GetTempFileName().Replace(".tmp",".pdf"); var tmp2 = Path.GetTempFileName().Replace(".tmp",".pdf"); // 1) Flatten if neededIfNeeded(inputPdf -> tmp1 Flatten, tmp1); // 2) Scrub -> tmp2 \_security.ScrubMetadata(tmp1, tmp2); // 3) Encrypt -> enc.pdf \_security.Encrypt(tmp2, encPath, userPassword); // 4) -> final.pdf if) Sign (optional (sign) \_security.Sign(encPath, finalPath, pfxPath, pfxPassword, reason, location); else File.Copy(encPath, finalPath, true); // 5) Verify (search sensitive terms) if (ContainsTerms(finalPath, termsToCheck)) { ShowError("Redaction/cleanup failed. Sensitive terms still present."); }



9) Redaction: keep it safe and simple

* **Mark redactions:** Let the user draw rectangles on pages and add patterns (emails/phones) to a queue.
* **Apply (safe default):** Rasterize those regions or entire pages and rebuild PDF pages (no hidden text left).
* **Verify:** After building, search the final PDF for the terms you intended to redact.

Prompt:

Implement RedactCommand: - Allow drawing rectangles on the overlay with a “Redact” brush. - On ApplyRedactions: for each page, render at 300 DPI, draw solid rectangles over marked areas, then rebuild the PDF with QuestPDF (one image per page) to produce a safe redacted PDF. Add VerifyRedactions: open final PDFium text page per page), search for patterns; if any hits, warn user, extract text (Pdf.



QuestPDF “image per page” build:

public byte[] BuildPdfFromImages(List pageImagePaths.Create(container) { return Document => { container.Page(page => { page.Size(PageSizes.A4); // or match original via DPI math page.Margin(0); page.Content().Element { (ctx => foreach (var img in pageImagePaths ctx.Image(img, ImageScaling.FitArea); }); }); }).) GeneratePdf(); }



This approach trades selectable text for guaranteed redaction safety. For family use, that’s a good default.

10) Create new PDFs (templates)

Prompt:

In PdfSafe.Creation: - Simple letter (logo, address, body text) - Blank - Form-like page lined note page (labels, text boxes; you can print and as drawn rectangles fill or keep digital via overlays) Expose ICreationService.Generate) -> byte[] and a(templateId, data UI dialog to fill basic fields and save a new PDF.



Sample QuestPDF snippet:

public byte[] GenerateLetter(string title, string body) { return Document.Create(container => { container.Page(page => { page.Size(PageSizes.A4); page.Margin(40); (title).SemiBold page.Header().Text().FontSize(20); page.Content().Text(body).FontSize(12).LineHeight(1.4f); page.Footer().AlignCenter().Text(x => x.CurrentPageNumber().Slash().Total }); }).GeneratePages()); Pdf(); }



11) Saving model: project vs flattened PDF

* **Project save (.pfs):**
  + **What:** Original PDF path + overlay annotations + ink + insert text/images + redaction rectangles + form values.
  + **Why:** Lets you re‑edit later without damaging the original.
* **Flattened PDF (for sharing):**
* **What:** Composite overlays and redactions, scrub, encrypt, and sign.
* **Why:** Delivers the secure final document.

Prompt:

Add a PfsDocument class with JSON serialization that stores overlay data and references the source PDF. Implement SaveProject/LoadProject. Add "Save As Flattened PDF" that runs the compose pipeline and writes a final shareable file.



12) Packaging and tests

* **Publish self - Command:‑contained:**  
  dotnet publish src/PdfSafe.App -c Release -r win-x64 --self-contained true -p:PublishSingleFile=true
  + **Bundle tools:** Copy tools/qpdf, tools/exiftool, tools/ghostscript into the publish folder.
  + **Test on another Windows machine:** Double‑click PdfSafe.exe, fill, share securely; verify open, annotate.
* **Automated tests (Cursor prompt):**

In PdfSafe.Tests, add: - OpenAndRender\_SmokeTest: open sample.pdf, render page 1 bitmap not null. - QuestPdf\_Generate\_Letter: generates bytes without error. , opens via Pdfium- Security\_ScrubEncrypt\_Roundtrip: scrub+encrypt, assert output exists = 0. - Redaction\_Verify\_NoTerms: and qpdf exit code mark/redact “email@example.com”, verify search returns false. Use xUnit and FluentAssertions.



13) Starter “Cursor” prompts you can paste as you go

* **Scaffold solution and projects:**

Create a .NET 8 WPF solution named PdfSafe with these projects: - PdfSafe.App (WPF) - PdfSafe.Core (class lib) - PdfSafe.Pdfium (class lib) - PdfSafe.Annotations (class lib) - PdfSafe.Creation (class lib) - PdfSafe.Security (class lib) - PdfSafe.Tests (xUnitiumViewer.Core, Pdf) Add NuGet: PdfiumViewer.NativeToolkit.Mvvm, Microsoft.x64, QuestPDF, Community.Extensions.DependencyInjection, xunit, FluentAssertions. Set up DI in App MainWindow.



.xaml.cs, MVVM in- **Integrate Pdfium viewer via WindowsFormsHost:**

Add a WindowsFormsHost to MainWindow and embed PdfiumViewer.PdfViewer. Expose IPdfViewAdapter with Open(string), RenderPage(int,dpi), PageCount, SetZoom(double). Bind OpenCommand to file picker and load thumbnails.



* **Annotations and Ink overlay:**

Add InkCanvas overlay for freehand. Commands: InkCommand toggles EditingMode. Add simple Highlight and Note using a Canvas overlay storing rectangles and popup SaveProject (JSON text. Implement) and SaveAsFlattenedPdf that composites overlays onto rendered bitmaps and rebuilds PDF pages.



* **Forms support:**

Add IFormService in PdfSafe.Pdfium: enumerate fields, Get/Set values for text/checkbox: Form mode highlights fields; clicking/radio/combo. UI edits values. Add FlattenForms() to burn values into content for final PDFs.



* **Security pipeline:**

Create ISecurityService wrapping tools: - ScrubMetadata -> exiftool -all= - Encrypt -> qpdf --encrypt 256 -- ... - Sign -> qpdf --sign ... --sign-p12= Add ShareSecurelyCommand: flatten (if needed) -> scrub -> encrypt -> (optional sign) -> verify (search terms). Show progress dialog and final path.



* **Redaction safe default:**

Add RedactCommand to draw rectangles. On Apply: render pages at 300 DPI, draw black boxes, rebuild PDF with QuestPDF. Run Verify for redacted terms using Pdfium text extraction; if found, block sharing. Redactions: search``` - \*\*Publish and smoke tests:\*\* ```text Add dotnet publish profile for self-contained win-x64 single file. Copy tools into publish folder. Add xUnit smoke tests for renderer, creation, and security wrappers.



14) What to build first (weekend plan)

* **Day 1:**
  + **Set up solution and packages.**
  + **Open PDFs and show thumbnails.**
  + **Text selection and copy.**
* **Day 2:**
  + **Ink + highlight + notes overlays.**
  + **Project save/load (JSON).**
  + **Save As Flattened PDF (overlay compositing).**
* **Day fill + flatten. 3:**
* **Form**
* **Security, encrypt, sign). pipeline (scrub**
* **Redaction (rasterize) + verify.**

15) Final tips

* **Keep two saves:** project (.pfs) for editing, flattened (.pdf) for sharing.
* **Prefer safety over features:** when in doubt, rasterize redactions and overlays.
* **No network:** don’t add auto‑updates or telemetry; keep tools local.
* **Version pin:** once stable, pin NuGet versions in Directory.Packages.props to avoid surprises.