

Aplikasi Desktop OCR Berbasis AI dengan Microsoft Foundry (Part 1)

Junindar, ST, MCPD, MOS, MCT, MVP

junindar@gmail.com

Lisensi Dokumen:

Copyright © 2003 IlmuKomputer.Com

Seluruh dokumen di IlmuKomputer.Com dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari IlmuKomputer.Com.

<http://junindar.blogspot.com>

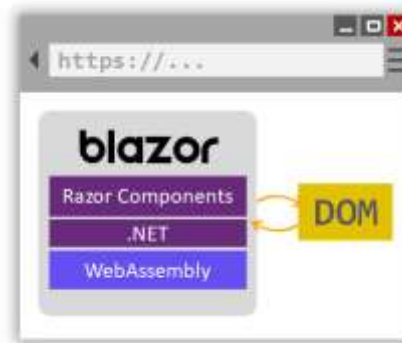
Abstrak

Blazor Hybrid adalah pendekatan yang menggabungkan framework Blazor dengan framework aplikasi native, seperti .NET MAUI, WPF, atau Windows Forms. Pada aplikasi Blazor Hybrid, komponen Razor berjalan secara native di dalam proses .NET dan dirender ke kontrol web view yang tertanam melalui local interop channel, sehingga komponen tidak berjalan di browser dan tidak melibatkan WebAssembly. Pada artikel ini, Blazor Hybrid digunakan bersama WPF, MainWindow pada WPF memuat kontrol BlazorWebView yang menampilkan halaman Razor menggunakan WebView2, sementara halaman tersebut memanggil layanan C# melalui dependency injection. Pendekatan ini dipakai untuk membangun aplikasi desktop Windows yang digunakan untuk melakukan scan dokumen, membaca isinya dengan OCR menggunakan model di Microsoft Foundry, dan menyimpan hasilnya di SQLite.

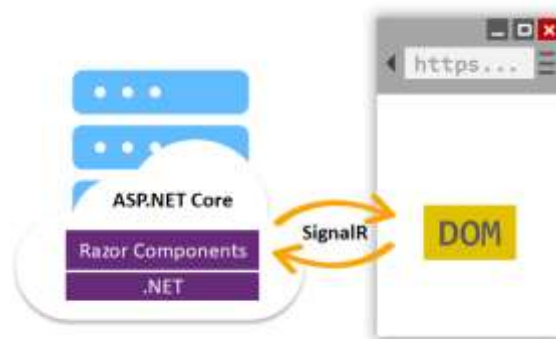
Pendahuluan

Blazor adalah Web Framework yang bersifat Open Source dimana aplikasi Web yang bersifat client-side interactive dapat dikembangkan dengan menggunakan .Net (C#) dan HTML. Pada saat ini C# biasa digunakan untuk melakukan proses back-end dari aplikasi web. Dengan menggunakan fitur baru dari ASP.NET Core yaitu Blazor, kita dapat membangun interactive WEB dengan menggunakan C# dan .NET .

Code .Net berjalan pada Web Assembly, yang artinya kita dapat menjalankan “NET” didalam browser (Client) tanpa harus menginstall plugin seperti Silverlight, Java mapun Flash.



Dengan menggunakan Blazor kita dapat memilih apakah menggunakan WebAssembly (Client Side), seperti yang telah dijelaskan di atas atau Blazor yang dijalankan diatas server. Dengan menggunakan cara kedua kita memerlukan SignalR untuk menghubungkan antara client (browser) dan server app. Sebagai contoh jika user melakukan proses klik button pada browser, maka data akan dikirimkan ke Server menggunakan SignalR dan hasilnya akan dikembalikan ke client dengan mengupdate DOM pada client.



Aplikasi Blazor menggunakan runtime .NET yang didasarkan pada Web Assembly atau disingkat WASM. WASM didukung secara native oleh mayoritas browser.

Blazor Hybrid adalah pendekatan inovatif yang menggabungkan framework Blazor dengan .NET MAUI (Multi-platform App UI). Yang memungkinkan kita sebagai *developer* untuk membangun aplikasi *cross platform* menggunakan teknologi web yang sudah dikenal. Dalam aplikasi Blazor Hybrid, komponen Razor berjalan secara native di perangkat dan dirender ke kontrol Web View yang tertanam melalui local interop. Sehingga komponen ini tidak berjalan di browser dan tidak melibatkan WebAssembly.



Terdapat beberapa keuntungan bagi pengembang yang ingin membangun aplikasi cross platform dengan menggunakan Blazor Hybrid seperti:

- Kita dapat menggunakan kembali komponen-komponen UI web yang sama di berbagai platform (seluler, desktop, dan web), sehingga mengurangi kebutuhan untuk menulis kode terpisah untuk setiap platform.
- Developer dapat menggunakan teknologi web yang sudah dikenal seperti HTML, CSS, dan C# untuk membangun aplikasi, yang dapat mempercepat proses development.
- Karena komponen Blazor Hybrid berjalan secara native di perangkat, komponen tersebut dapat menawarkan performa yang lebih baik dibandingkan aplikasi web tradisional yang dijalankan di browser.

- Blazor Hybrid memungkinkan kita untuk mengakses fitur dan API perangkat asli melalui .NET MAUI, sehingga dapat membuat aplikasi yang lebih canggih dan kaya akan fitur.

Dokumen hard-copy seperti nota, struk belanja, invoice, KTP, dan surat masih menjadi bagian dari pekerjaan sehari-hari. Masalahnya, data-data di dalamnya sulit dipakai ulang. Untuk mencatat satu struk belanja ke dalam sistem yang kita miliki, seseorang harus membaca dan mengetik ulang nama toko, tanggal, daftar barang, pajak, dan total. Proses manual ini sangat lambat dan sangat memungkinkan terjadi salah ketik. Dengan memanfaatkan model AI yang dapat membaca gambar, proses tersebut dapat dilakukan secara otomatis, pengguna cukup memotret atau memilih gambar dokumen, dan aplikasi mengubahnya menjadi data yang siap digunakan oleh sistem.

Microsoft Foundry adalah platform dari Microsoft untuk menggunakan model AI dari satu tempat. Salah satu kemampuannya adalah model vision, yaitu model yang dapat menerima gambar sebagai input dan membaca teks di dalamnya (OCR). Dalam artikel ini, aplikasi hanya perlu mengirim gambar ke Foundry melalui API dan menerima hasilnya. Tidak ada model yang harus dilatih atau diinstall di setiap komputer pengguna. Untuk menghubungkan aplikasi .NET dengan model tersebut, kita hanya perlu menggunakan Microsoft.Extensions.AI, yaitu sebuah library standar sehingga kode aplikasi tidak terikat langsung hanya pada satu provider model tertentu.

Hasil OCR berupa teks mentah atau tidak terstruktur, sedangkan sistem membutuhkan data yang terstruktur. Karena itu, proses tidak berhenti hanya pada pembacaan teks saja. Setiap jenis dokumen memiliki skema field sendiri. Struk belanja memiliki toko, tanggal, dan daftar barang, sedangkan KTP memiliki NIK, nama, dan alamat. Aplikasi pertama-tama menentukan jenis dokumen dari teks, lalu mengisi skema yang sesuai. Hasilnya divalidasi dengan aturan pada code, kemudian ditampilkan di form untuk diperiksa, diedit, dan disetujui oleh pengguna. Dengan cara ini, AI membantu kita untuk menginput data, tetapi keputusan akhir tetap berada di tangan pengguna.

Seluruh data disimpan di SQLite, yaitu database ringan yang berjalan sebagai satu file di komputer pengguna tanpa memerlukan server database terpisah. Gambar, teks hasil bacaan, dan data terstruktur disimpan di sana sehingga riwayat tetap ada setelah aplikasi ditutup. Aplikasi di design dengan menggunakan satu prinsip, yaitu simpan dulu, baru

proses, dan melakukan proses simpan setiap tahap. Saat melakukan pemanggilan ke Foundry bisa gagal kapan saja karena internet terputus, kuota habis, atau API key salah. Dengan menyimpan hasil di setiap tahap, dokumen yang gagal diproses tidak hilang. Pengguna cukup melanjutkan dari tahap yang gagal tanpa harus melakukan upload atau mengambil gambar/photo kembali.

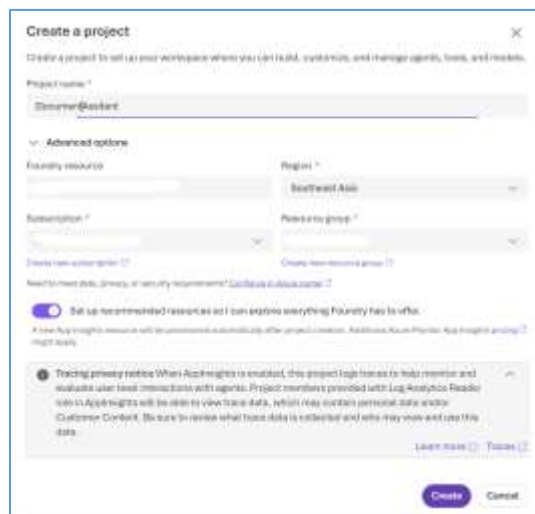
Untuk antarmukanya, kita menggunakan WPF yang digabungkan dengan Blazor Hybrid melalui kontrol BlazorWebView. Pendekatan ini memungkinkan kita membangun tampilan dengan komponen Razor dan C#, sementara aplikasi tetap berjalan sebagai aplikasi desktop Windows yang dapat mengakses fitur perangkat seperti kamera, clipboard, dan file system. Razor Component berjalan secara native dan dirender ke WebView yang tertanam, sehingga tidak berjalan di browser dan tidak melibatkan WebAssembly.

Pada artikel ini kita akan membangun aplikasi bernama “Document Assistant”, mulai dari solution kosong sampai aplikasi yang dapat memindai, membaca, mengekstrak, dan menyimpan dokumen. Pembahasan dibatasi pada fitur Scan beserta ekstraksi datanya.

Sebelum kita masuk ke coding, pertama-tama kita buat project di Microsoft Foundry .

Ikuti langkah-langkah dibawah ini.

- Buka <https://ai.azure.com> dan login dengan akun Azure.
- Pilih Create a new project (atau klik nama project di pojok kiri atas, lalu Create new project).

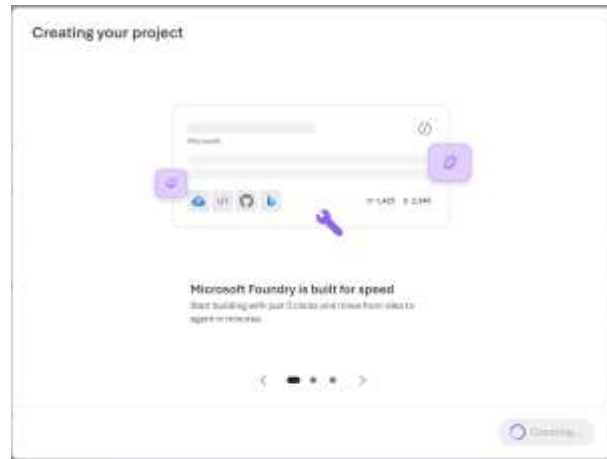
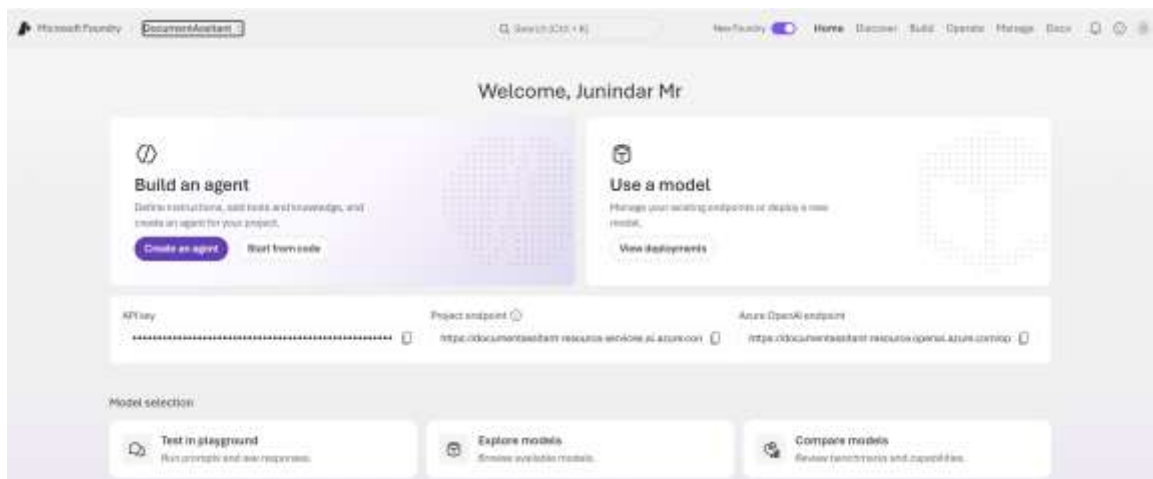


Beri nama, misalnya **DocumentAssistant**.

Buka Advanced options:

Resource group: buat baru, misalnya **DocumentAssistant**. Dengan resource group sendiri, semua resource mudah dihapus di akhir.

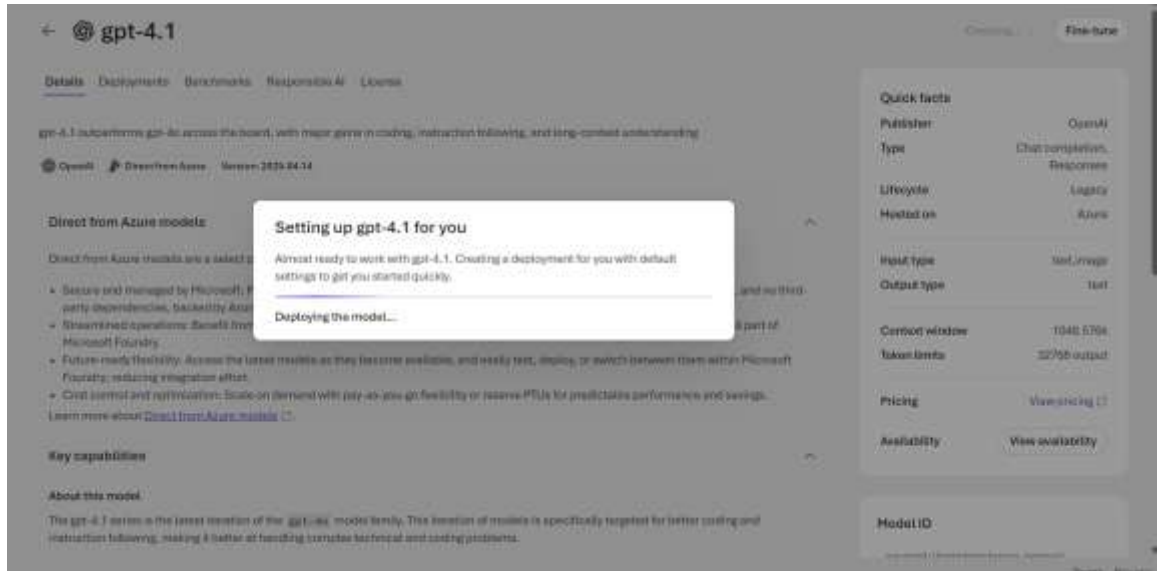
- Klik Create project. Portal otomatis membuat resource Foundry di background tunggu hingga selesai. Nama resource itu akan muncul di endpoint.



Deploy model yang mendukung gambar:

OCR kita memakai model multimodal, yaitu model yang bisa menerima gambar. Keluarga gpt-4.1, gpt-5-mini, dan gpt-5.4-mini mendukungnya. Pilih salah satu yang tersedia pada account kamu.

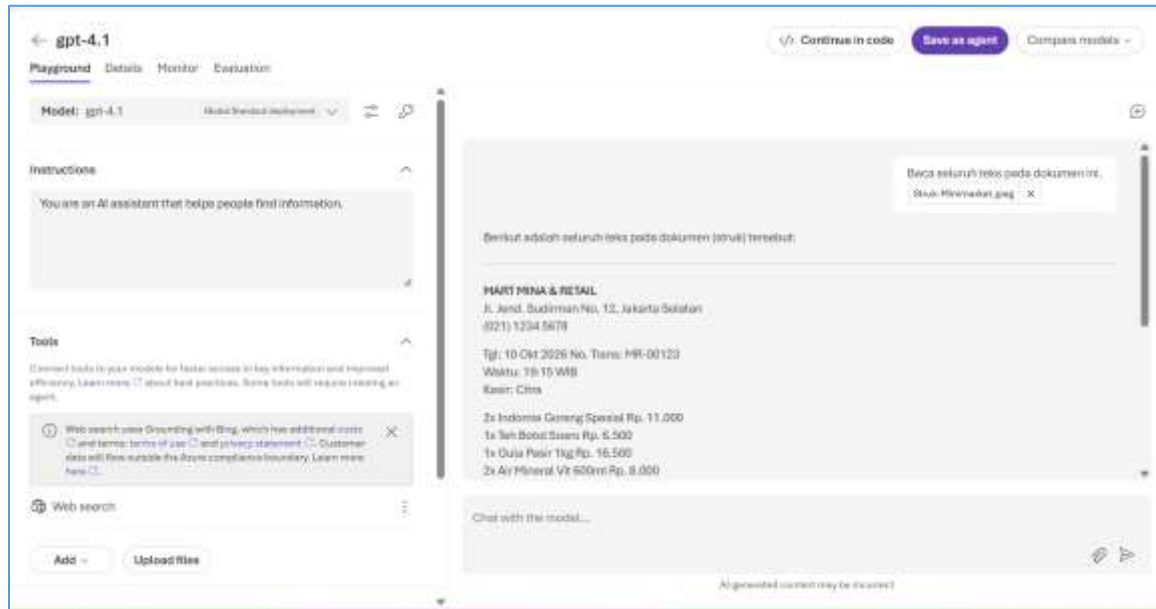
- Di portal Foundry, klik Discover, lalu klik Models.
- Cari gpt-4.1 (atau model lain di atasnya), lalu klik Card-nya untuk melihat detail.
- Klik Deploy dan pilih lalu Default settings.



Setelah selesai, kamu masuk ke playground. Catat nama deployment yang tertulis di sana. Nama deployment inilah (bukan nama model) yang dipakai aplikasi. Secara bawaan namanya sama dengan nama model, misalnya gpt-4.1.

Selanjutnya kita lakukan pengujian model di playground. Ikuti langkah-langkah dibawah ini.

- Pastikan deployment tadi terpilih.
- Lampirkan sample photo nota atau dokumen lewat ikon lampiran di kotak chat, lalu ketikkan: Baca seluruh teks pada dokumen ini.
- Bila teks terbaca, model dan kuotamu siap. Bila tombol lampiran gambar tidak ada, model itu tidak mendukung gambar. Deploy dengan menggunakan model lain.

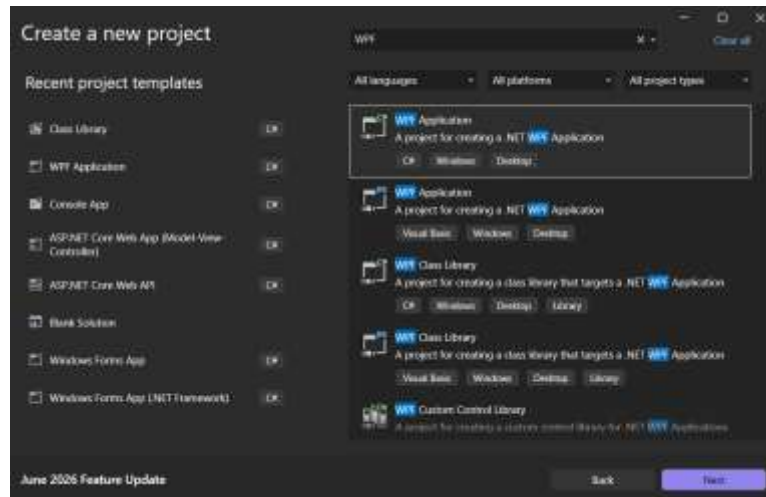


Langkah selanjutnya adalah mengambil endpoint dan API key pada Azure Portal.

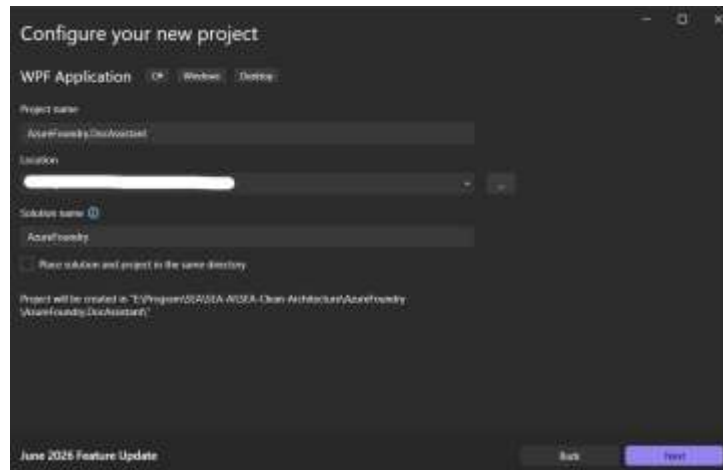
- Buka <https://portal.azure.com> , lalu buka resource group DocumentAssistant.
- Di menu kiri, buka Resource Management, lalu Keys and Endpoint.
- Salin KEY 1. (KEY 2 adalah cadangan untuk rotasi.)
- Endpoint yang kita pakai berbentuk Azure OpenAI v1 seperti dibawah ini.
`https://<nama-resource>-resource.openai.azure.com/openai/v1/`

Setelah selesai dengan Microsoft Foundry, sekarang kita lanjut bekerja dengan Visual Studio. Ikuti Langkah-langkah dibawah ini.

- Create a new project, pilih template WPF Application (bukan WPF Class Library), bahasa C#.



- Beri nama project AzureFoundry.DocAssistant, solution AzureFoundry, framework .NET 10.0.



- Klik kanan solution, Add, New Project, pilih Class Library, beri nama AzureFoundry.DocAssistant.Core, Pilih framework .NET 10.0. Ulangi untuk AzureFoundry.DocAssistant.Infrastructure.
- Lalu hapus Class1.cs pada kedua class library yang kira buat sebelumnya.
- Terakhir atur referensi (klik kanan project, Add, Project Reference):
 1. AzureFoundry.DocAssistant merujuk ke Core dan Infrastructure
 2. AzureFoundry.DocAssistant.Infrastructure merujuk ke Core
 3. Core tidak merujuk ke project manapun.

Lalu kita tambahkan Nuget Package untuk masing-masing project seperti pada table dibawah ini.

Project	Nuget Package
AzureFoundry.DocAssistant.Core	Microsoft.Extensions.AI.Abstractions
AzureFoundry.DocAssistant.Core	Microsoft.Extensions.DependencyInjection.Abstractions
AzureFoundry.DocAssistant.Infrastructure	Microsoft.Data.Sqlite
AzureFoundry.DocAssistant.Infrastructure	Microsoft.Extensions.AI.OpenAI
AzureFoundry.DocAssistant.Infrastructure	Microsoft.Extensions.Options.ConfigurationExtensions
AzureFoundry.DocAssistant (WPF)	Microsoft.AspNetCore.Components.WebView.Wpf
AzureFoundry.DocAssistant (WPF)	Microsoft.Extensions.Configuration.Json, .EnvironmentVariables, .UserSecrets

Bekerja dengan Project AzureFoundry.DocAssistant.Core

Buat tiga folder di project ini : Storage, Ai, dan Ocr. Dan pastikan namespace tiap file mengikuti nama foldernya. Ikuti Langkah-langkah dibawah ini (lakukan pada folder **Storage**):

- Tambahkan sebuah class dengan nama Document.cs , dan ketikkan sintaksnya seperti dibawah.

```
namespace AzureFoundry.DocAssistant.Core.Storage;

public enum DocumentType
{
    Image = 0,
}

public enum DocumentStatus
{
    Unread = 0,
    Ready = 1
}
```

```
public sealed record Document(  
    string Id,  
    string Title,  
    DocumentType Type,  
    DateTime Time,  
    DocumentStatus Status,  
    string? Text);  
  
public sealed record DocumentFile(  
    byte[] Bytes,  
    string MediaType);
```

Dua enum dan dua record. Document hanya membawa metadata dan teks. File gambar disimpan terpisah (di disk), dan isinya dibawa oleh DocumentFile hanya saat dibutuhkan. Dengan begitu, menampilkan history tidak perlu memuat banyak gambar ke memori.

DocumentStatus adalah inti dari prinsip "simpan dulu, baru baca". "Unread" berarti gambar aman tetapi teksnya belum ada.

- Selanjutnya kita tambah sebuah interface dengan nama "IDocumentStore" masih didalam folder yang sama. Seluruh aplikasi hanya tahu kontrak ini, tetapi tidak tahu pada implementasinya bahwa di balikny ada SQLite. Terdapat enam operasi yang dibutuhkan fitur Scan (tambah (CreateAsync) , ambil (GetAsync) , daftar (ListAsync), simpan teks (SaveTextAsync), ambil file (GetFileAsync), hapus (DeleteAsync)). Ketikkan sintaks dibawah.

```
namespace AzureFoundry.DocAssistant.Core.Storage;  
  
public interface IDocumentStore  
{  
  
    Task<Document> CreateAsync(string title, DocumentType type,  
        byte[] file, string mediaType,  
        CancellationToken ct = default);  
  
    Task<Document?> GetAsync(string id, CancellationToken ct = default);  
  
    Task<IReadOnlyList<Document>> ListAsync(DocumentType? type, int max,  
        CancellationToken ct = default);  
  
    Task SaveTextAsync(string id, string text, CancellationToken ct = default);  
  
    Task<DocumentFile?> GetFileAsync(string id, CancellationToken ct = default);  
  
    Task DeleteAsync(string id, CancellationToken ct = default);  
  
}
```

- Membuat class “StorageException”

Seluruh kegagalan teknis pada saat melakukan penyimpanan, seperti folder yang tidak dapat ditulis, disk penuh, atau database yang rusak, ditangani oleh satu jenis exception, yaitu StorageException, yang memiliki pesan aman untuk ditampilkan kepada pengguna. Sehingga UI cukup menangani StorageException tanpa perlu mengetahui detail exception dari SQLite, seperti SqliteException.

```
namespace AzureFoundry.DocAssistant.Core.Storage;  
  
public sealed class StorageException(string message, Exception? inner = null)  
    : Exception(message, inner);
```

Setelah selesai dengan folder “Storage” kita lanjutkan dengan folder “Ai”, ikuti Langkah-langkah dibawah ini.

- Buat sebuah interface dengan nama “IChatClientFactory”, dan ketikkan sintaks dibawah ini.

```
using Microsoft.Extensions.AI;  
namespace AzureFoundry.DocAssistant.Core.Ai;  
  
public interface IChatClientFactory  
{  
    IChatClient Create();  
    string GetErrorMessage(Exception ex);  
}
```

IChatClientFactory selain berfungsi untuk membuat "client", juga memiliki tugas untuk menerjemahkan exception teknis menjadi pesan yang dipahami pengguna, karena hanya provider AI yang tahu arti kode seperti 401 atau 429

- Lalu lanjutkan dengan membuat class “AiException” seperti dibawah

```
namespace AzureFoundry.DocAssistant.Core.Ai;  
  
public sealed class AiException(string message, Exception? inner = null) :  
    Exception(message, inner);
```

Untuk folder terakhir (Ocr), tambahkan beberapa file seperti dibawah

- Buat sebuah “IDocumentOcrService” dan class “OcrException” ketikkan sintaks seperti dibawah ini.

```
namespace AzureFoundry.DocAssistant.Core.Ocr;

public interface IDocumentOcrService
{
    Task<string> ReadTextAsync(byte[] image, string mediaType,
        CancellationToken ct = default);
}

public sealed class OcrException(string message, Exception? inner = null)
    : Exception(message, inner);
```

“ReadTextAsync” akan digunakan untuk membaca seluruh teks pada gambar dokumen dan mengembalikannya sebagai teks biasa.

- Tambahkan sebuah class dengan nama “DocumentScanService”

```
using AzureFoundry.DocAssistant.Core.Storage;
namespace AzureFoundry.DocAssistant.Core.Ocr;

public sealed class DocumentScanService(IDocumentStore store,
    IDocumentOcrService ocr)
{
    public Task<Document> SaveAsync(
        byte[] bytes, string mediaType, string name,
        CancellationToken ct = default) =>
        store.CreateAsync(name, DocumentType.Image, bytes, mediaType, ct);

    public async Task<Document> ReadAsync(
        string id, CancellationToken ct = default)
    {
        var document = await store.GetAsync(id, ct)
            ?? throw new OcrException($"Document '{id}' was not found.");

        if (document.Status == DocumentStatus.Ready)
        {
            return document;
        }

        var file = await store.GetFilesAsync(id, ct)
            ?? throw new OcrException(
                "The document image file was not found in storage.");

        var text = await ocr.ReadTextAsync(
            file.Bytes,
            file.MediaType,
            ct);

        await store.SaveTextAsync(id, text, ct);
        return document with
        {
            Text = text,
            Status = DocumentStatus.Ready
        };
    }
}
```

Sintaks ini berfungsi untuk menyimpan dokumen gambar dan membaca teks di dalam gambar menggunakan OCR (Optical Character Recognition). *DocumentScanService*

punya dua fungsi utama, yaitu *SaveAsync()* dan *ReadAsync()*. *SaveAsync()* digunakan untuk menyimpan gambar ke storage, sedangkan *ReadAsync()* digunakan untuk mengambil dokumen berdasarkan id. Kalau dokumen tidak ditemukan, proses akan menghasilkan *OcrException*. Di dalam *ReadAsync()*, kalau status dokumen sudah *Ready*, dokumen langsung dikembalikan. Kalau belum, gambar diambil dari storage lalu dikirim ke OCR untuk membaca teksnya. Hasil teks disimpan ke storage, kemudian dokumen dikembalikan dengan *Text* berisi hasil OCR dan *Status* menjadi *Ready*.

Selesai dengan ketiga folder diatas kita lanjutkan dengan melakukan register service. Buat sebuah static class pada project dengan nama “*CoreServiceCollectionExtensions*” dan ketikkan sintaks dibawah ini.

```
using AzureFoundry.DocAssistant.Core.Ocr;
using Microsoft.Extensions.DependencyInjection;

namespace AzureFoundry.DocAssistant.Core;

public static class CoreServiceCollectionExtensions
{
    public static IServiceCollection AddDocAssistantCore(this
        IServiceCollection services)
    {
        services.AddSingleton<DocumentScanService>();
        return services;
    }
}
```

Project ini (*DocAssistant.Core*) mendaftarkan service-nya sendiri lewat extension method, jadi App tidak perlu tahu class apa saja yang ada di dalamnya. Namespace-nya sengaja *Microsoft.Extensions.DependencyInjection*, supaya App cukup memanggil *AddDocAssistantCore()* tanpa using tambahan.

Bekerja dengan Project AzureFoundry.DocAssistant. Infrastructure

Buat dua buah folder “*Foundry*” dan “*Storage*” terlebih dahulu. Pada project ini kita akan bekerja pada folder *Foundry* terlebih dahulu, ikuti langkah-lagkah dibawah ini.

- Buat sebuah class dengan nama “*FoundryOptions*” dan ketikkan sintaks lengkapnya seperti dibawah.

```
namespace AzureFoundry.DocAssistant.Infrastructure.Foundry;
public sealed class FoundryOptions
{
    public const string SectionName = "Foundry";
    public string Endpoint { get; set; } = "";
    public string ApiKey { get; set; } = "";
    public string ModelDeployment { get; set; } = "";
}
```

FoundryOptions adalah class yang digunakan untuk menyimpan konfigurasi Azure AI Foundry. *SectionName* digunakan untuk menentukan nama section konfigurasi, yaitu "Foundry", sedangkan *Endpoint* berisi alamat endpoint, *ApiKey* berisi API key untuk akses, dan *ModelDeployment* berisi nama model yang digunakan. Nilai awal ketiga property tersebut dibuat string kosong (""), dan nantinya bisa diisi melalui file konfigurasi seperti appsettings.json.

```
using System.ClientModel;
using System.Text.RegularExpressions;
using AzureFoundry.DocAssistant.Core.Ai;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Options;
using OpenAI;

namespace AzureFoundry.DocAssistant.Infrastructure.Foundry;

public sealed class FoundryClientFactory(
    IOptions<FoundryOptions> options)
    : IChatClientFactory
{
    private readonly FoundryOptions _options = options.Value;

    public IChatClient Create()
    {
        if (string.IsNullOrEmpty(_options.Endpoint) ||
            string.IsNullOrEmpty(_options.ApiKey) ||
            string.IsNullOrEmpty(_options.ModelDeployment))
        {
            throw new AiException(
                "Foundry is not configured. Please set Foundry:Endpoint, " +
                "Foundry:ApiKey, and Foundry:ModelDeployment in user-secrets.");
        }

        var client = new OpenAIClient(
            new ApiKeyCredential(_options.ApiKey),
            new OpenAIClientOptions
            {
                Endpoint = NormalizeEndpoint(_options.Endpoint)
            });

        return client.GetChatClient(_options.ModelDeployment)
            .AsIChatClient();
    }

    public static Uri NormalizeEndpoint(string endpoint)
    {
        var text = endpoint.Trim().TrimEnd('/');
        if (!text.EndsWith("/openai/v1",
            StringComparison.OrdinalIgnoreCase))
        {
            text += "/openai/v1";
        }
        return new Uri(text + "/");
    }

    public string GetErrorMessage(Exception ex)
    {
        var text = ex.ToString();
        // Detail sintaks pada project lampiran
        return "Failed to connect to Foundry: " + ex.Message;
    }

    private static bool ContainsStatusCode(string text, int code) =>
        Regex.IsMatch(text, $"(?:?!\\d){code}(?!\\d)");
}
```

FoundryClientFactory berfungsi untuk membuat koneksi ke Azure AI Foundry. Class ini mengambil konfigurasi dari *FoundryOptions*, seperti Endpoint, ApiKey, dan ModelDeployment. Kalau ada konfigurasi yang masih kosong, proses akan menghasilkan *AiException*.

Method *Create()* membuat *OpenAIClient* menggunakan ApiKey dan Endpoint, lalu memilih model berdasarkan *ModelDeployment*. Sementara itu, *NormalizeEndpoint()* memastikan format endpoint sudah sesuai dengan menambahkan /openai/v1 jika belum ada.

Method *GetErrorMessage()* digunakan untuk mengenali error dan memberikan pesan yang lebih mudah dipahami, misalnya API key salah (401), akses ditolak (403), model tidak ditemukan (404), atau request terlalu banyak (429). Kalau error tidak cocok dengan kondisi tersebut, method akan mengembalikan pesan error umum.

- Lanjutkan dengan membuat class “FoundryDocumentOcrService”.

FoundryDocumentOcrService berfungsi untuk membaca teks dari gambar dokumen menggunakan AI di Azure AI Foundry. Pada bagian *Instructions* berisi aturan agar AI menyalin semua teks yang terlihat, mempertahankan urutan teks, tidak menebak tulisan yang tidak terbaca, dan menampilkan pesan “no text” jika gambar tidak memiliki teks.

Method *ReadTextAsync()* menerima gambar dan mediaType, lalu mengirimkannya ke AI melalui *_client.GetResponseAsync()* untuk diproses. Hasilnya diambil sebagai teks, lalu dilakukan *Trim()* untuk menghapus spasi di awal dan akhir. Jika hasilnya kosong atau terjadi error, proses akan ditangani melalui *catch* dan menghasilkan *OcrException* dengan pesan yang sesuai. *_client ??= factory.Create()* memastikan client dibuat saat pertama kali dibutuhkan dan digunakan kembali berikutnya.

Untuk detail sintaks dapat dilihat pada project lampiran.

Setelah selesai dengan folder “Foundry” kita lanjutkan dengan folder “Storage”, ikuti Langkah-langkah dibawah ini.

- Tambahkan sebuah class dengan nama “StorageOptions”

```
namespace AzureFoundry.DocAssistant.Infrastructure.Storage;

public sealed class StorageOptions
{
    public const string SectionName = "Storage";
    public string Folder { get; set; } = "";
}
```

Lokasi penyimpanan bisa diatur melalui konfigurasi *Folder*. Dan jika kosong, aplikasi akan menggunakan %LocalAppData%\DocAssistant (biasanya C:\Users\<nama>\AppData\Local\DocAssistant).

- Lalu buat sebuah class dengan nama “SqliteDocumentStore”. Class ini digunakan untuk melakukan implementasi dari interface “IDocumentStore”.

```
using System.Globalization;
using AzureFoundry.DocAssistant.Core.Storage;
using Microsoft.Data.Sqlite;
using Microsoft.Extensions.Options;

namespace AzureFoundry.DocAssistant.Infrastructure.Storage;
public sealed class SqliteDocumentStore : IDocumentStore, IDisposable
{
    //detail sintaks pada project lampiran
}
```

SqliteDocumentStore berfungsi untuk mengelola penyimpanan dokumen menggunakan database SQLite dan menyimpan file gambar di folder lokal. Constructor mengatur lokasi penyimpanan, folder files, dan file database docassistant.db. Bagian *_schema* digunakan untuk membuat tabel *documents* dan index jika belum tersedia.

Method *CreateAsync()* digunakan untuk menyimpan dokumen baru. Method ini membuat id unik, menyimpan file gambar ke folder, lalu menyimpan informasi dokumen ke database dengan status *Unread*. Kalau proses penyimpanan ke database gagal, file gambar akan dihapus agar tidak menjadi file yang terbengkalai.

Method *GetAsync()* mengambil satu dokumen berdasarkan id, sedangkan *ListAsync()* mengambil daftar dokumen berdasarkan tipe dan jumlah maksimal. *SaveTextAsync()* menyimpan hasil OCR sekaligus mengubah status menjadi *Ready*. Sementara itu, *GetFileAsync()* digunakan untuk mengambil file gambar beserta *mediaType* dari storage.

Method *DeleteAsync()* menghapus data dokumen dari database sekaligus file gambarnya dari folder. *InitializeAsync()* memastikan folder dan tabel database sudah

tersedia sebelum digunakan, sedangkan *OpenAsync()* membuka koneksi SQLite. *Map()* digunakan untuk mengubah data dari database menjadi object *Document*.

Terakhir, *Protect()* menangani error terkait SQLite dan file, lalu mengubahnya menjadi *StorageException* agar lebih mudah ditangani. *GetExtension()* menentukan ekstensi file berdasarkan *mediaType*, sedangkan *Dispose()* membersihkan resource yang digunakan oleh *_initializationLock*.

Selesai dengan kedua folder diatas kita lanjutkan dengan melakukan register service.

Buat sebuah static class pada project dengan nama "InfrastructureServiceCollectionExtensions" dan ketikkan sintaks dibawah ini.

```
using AzureFoundry.DocAssistant.Core.Ai;
using AzureFoundry.DocAssistant.Core.Ocr;
using AzureFoundry.DocAssistant.Core.Storage;
using AzureFoundry.DocAssistant.Infrastructure.Foundry;
using AzureFoundry.DocAssistant.Infrastructure.Storage;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace AzureFoundry.DocAssistant.Infrastructure;
public static class InfrastructureServiceCollectionExtensions
{
    public static IServiceCollection AddDocAssistantInfrastructure(this
        IServiceCollection services, IConfiguration configuration)
    {
        services.Configure<FoundryOptions>(configuration.GetSection(FoundryOptions.
            SectionName));
        services.AddSingleton<IChatClientFactory, FoundryClientFactory>();
        services.AddSingleton<IDocumentOcrService,
            FoundryDocumentOcrService>();

        services.Configure<StorageOptions>(configuration.GetSection(StorageOptions.
            SectionName));
        services.AddSingleton<IDocumentStore, SqliteDocumentStore>();
        return services;
    }
}
```

AddDocAssistantInfrastructure() digunakan untuk mendaftarkan service ke Dependency Injection, mulai dari konfigurasi Azure AI Foundry dan storage, *FoundryClientFactory* untuk koneksi AI, *FoundryDocumentOcrService* untuk OCR, hingga *SqliteDocumentStore* untuk penyimpanan dokumen.

-

Bekerja dengan Project AzureFoundry.DocAssistant (Wpf dan Blazor)

Setelah selesai dengan dua project library diatas, sekarang kita lanjut dengan project yang focus pada user interface. Gambar dibawah adalah struktur project pada “AzureFoundry.DocAssistant”.



Penjelasan Struktur Project (WPF (Blazor Hybrid)):

- Components: Berisi komponen-komponen antarmuka berbasis Razor (.razor).
 - Layout: Berisi tata letak utama aplikasi seperti MainLayout.razor.
 - Pages: Berisi halaman utama aplikasi, contohnya ScanDokumen.razor.
 - Shared: Berisi komponen yang dapat digunakan kembali, seperti StatusBadge.razor.
 - Routes.razor & _Imports.razor: File routing dan konfigurasi namespace global untuk komponen Razor.
- Services : Tempat untuk meletakkan logika bisnis, layanan API, atau integrasi dengan Azure Foundry
- Wwwroot : Menyimpan aset web statis seperti file CSS (css/app.css) , JavaScript (js/) dan index.html
- File Root :
 - App.xaml & MainWindow.xaml: Konfigurasi tingkat aplikasi desktop.
 - appsettings.json: File konfigurasi untuk menyimpan kunci API, string koneksi, atau pengaturan environment.

Untuk langkah awal kita lakukan perubahan untuk file-file yang sudah ada pada saat project ini dibuat.

- Buka file “App.xaml”, secara default Template WPF memiliki atribut `StartupUri="MainWindow.xaml"`. Hapus atribut, tersebut karena kita akan membuka jendela secara manual melalui `OnStartup`. Jika atribut ini tidak dihapus, aplikasi akan membuka dua jendela sekaligus. Sehingga seperti pada sintaks dibawah

```
<Application x:Class="AzureFoundry.DocAssistant.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:AzureFoundry.DocAssistant">
    <Application.Resources>

    </Application.Resources>
</Application>
```

- Buka file “App.xaml.cs”, lalu ketikkan seperti pada sintaks dibawah
Sintaks dibawah adalah *composition root*, tempat semua layanan aplikasi didaftarkan dan disiapkan. Setelah semuanya siap, aplikasi membuka `MainWindow`.
 - `OnStartup()` untuk menjalankan proses inisialisasi saat aplikasi mulai.
 - `ConfigurationBuilder()` membaca konfigurasi dari `appsettings.json`, `User Secrets`, dan `environment variables`. Nilai dari sumber terakhir menimpa nilai sebelumnya jika key sama.
 - `AddWpfBlazorWebView()` menyiapkan layanan (service) agar Blazor bisa berjalan di WPF.
 - `AddBlazorWebViewDeveloperTools()` mengaktifkan DevTools untuk debugging saat mode Debug.
 - `AddDocAssistantCore()` dan `AddDocAssistantInfrastructure()` mendaftarkan service dari layer Core dan Infrastructure.
 - `AddSingleton<IClipboardImageService, WpfClipboardImageService>()` mendaftarkan service untuk mengakses gambar dari clipboard Windows.
 - `BuildServiceProvider()` menyiapkan semua service agar bisa digunakan aplikasi.
 - `Resources["ServiceProvider"]` menyimpan `ServiceProvider` agar bisa diakses melalui resource WPF.
 - `new MainWindow().Show()` membuka jendela utama aplikasi.

- *OnExit()* membersihkan ServiceProvider saat aplikasi ditutup agar resource bisa dilepas dengan benar.

```
using System.Windows;
using AzureFoundry.DocAssistant.Core;
using AzureFoundry.DocAssistant.Infrastructure;
using AzureFoundry.DocAssistant.Services;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace AzureFoundry.DocAssistant;

public partial class App : Application
{
    private ServiceProvider? _serviceProvider;
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        // Urutan: appsettings.json (default kosong) -> user-secrets
        // (nilai asli) -> environment variable.
        // Nilai yang belakangan menimpa yang sebelumnya.
        var configuration = new ConfigurationBuilder()
            .SetBasePath(AppContext.BaseDirectory)
            .AddJsonFile("appsettings.json", optional: true)
            .AddUserSecrets<App>(optional: true)
            .AddEnvironmentVariables(prefix: "DOCASSISTANT_")
            .Build();

        var services = new ServiceCollection();
        services.AddWpfBlazorWebView();

#if DEBUG
        services.AddBlazorWebViewDeveloperTools();
#endif

        services.AddDocAssistantCore();
        services.AddDocAssistantInfrastructure(configuration);

        services.AddSingleton<IClipboardImageService,
WpfClipboardImageService>();

        _serviceProvider = services.BuildServiceProvider();
        Resources["ServiceProvider"] = _serviceProvider;

        new MainWindow().Show();
    }

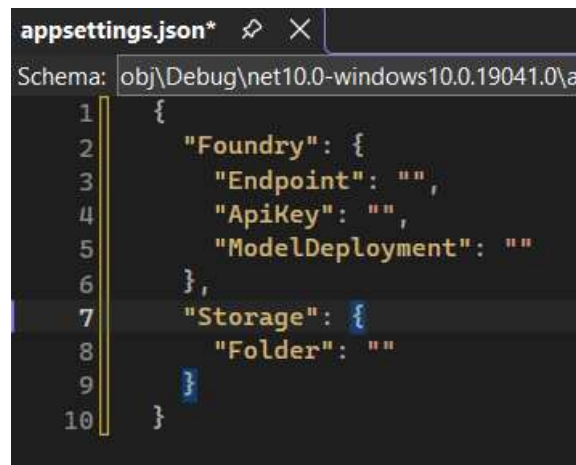
    protected override void OnExit(ExitEventArgs e)
    {
        _serviceProvider?.Dispose();
        base.OnExit(e);
    }
}
```

- Buka file “MainWindow.xaml”. Sintaks pada file ini mengatur tampilan jendela utama aplikasi WPF yang menampilkan UI Blazor. *Title*, *Height*, dan *Width* mengatur

judul serta ukuran jendela. Grid menjadi container utama untuk menampung *BlazorWebView*. *HostPage="wwwroot\index.html"* menentukan halaman HTML awal yang digunakan Blazor, sedangkan *Services="{DynamicResource ServiceProvider}"* menghubungkan Blazor dengan layanan yang sudah didaftarkan di *App.xaml.cs*. Terakhir, *RootComponent* menampilkan komponen Routes pada elemen HTML dengan selector *#app*, sehingga halaman Blazor bisa tampil di dalam jendela WPF.

```
<Window x:Class="AzureFoundry.DocAssistant.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:blazor="clr-
namespace:Microsoft.AspNetCore.Components.WebView.Wpf;assembly=Microsoft.AspNetCore.Components.WebView.Wpf"
        xmlns:components="clr-
namespace:AzureFoundry.DocAssistant.Components"
        Title="Document Assistant" Height="720" Width="1080">
    <Grid>
        <blazor:BlazorWebView HostPage="wwwroot\index.html"
                            Services="{DynamicResource ServiceProvider}">
            <blazor:BlazorWebView.RootComponents>
                <blazor:RootComponent Selector="#app"
                                    ComponentType="{x:Type
                                    components:Routes}" />
            </blazor:BlazorWebView.RootComponents>
        </blazor:BlazorWebView>
    </Grid>
</Window>
```

- Buka file "appsettings.json" ketikkan json dibawah ini.



```
appsettings.json* X
Schema: obj\Debug\net10.0-windows10.0.19041.0\appsettings.json
1  {
2      "Foundry": {
3          "Endpoint": "",
4          "ApiKey": "",
5          "ModelDeployment": ""
6      },
7      "Storage": {
8          "Folder": ""
9      }
10 }
```

- Setelah selesai dengan existing file, kita lanjutkan dengan membuat file-file yang diperlukan oleh project ini.

Buat sebuah file “ProcessStatus.cs” dan ketikkan sintaks untuk enum seperti di bawah. karena ProcessStatus ini ditampilkan di layer dan merupakan bagian dari UI, sehingga file ini berada di project App.

```
namespace AzureFoundry.DocAssistant;
public enum ProcessStatus
{
    Idle,
    Processing,
    Done,
    Failed,
}
```

- Kita lanjutkan dengan membuat sebuah folder dengan nama “Services”, diikuti dengan membuat file “IClipboardImageService.cs” dan ketikkan sintaksnya seperti di bawah ini.

```
namespace AzureFoundry.DocAssistant.Services;

public sealed record ClipboardImage(byte[] Bytes, string MediaType, string Name);

public sealed class InvalidImageException(string message) : Exception(message);

public interface IClipboardImageService
{
    Task<ClipboardImage?> GetImageAsync(long maxBytes);
}
```

Interface ini mendefinisikan kontrak untuk fitur mengambil gambar dari clipboard. *ClipboardImage* menyimpan data gambar dalam bentuk *byte[]*, tipe file di *MediaType*, dan nama file di *Name*. *InvalidImageException* digunakan untuk menandai error saat gambar tidak valid. Sementara itu, *IClipboardImageService* menyediakan method *GetImageAsync(long maxBytes)* untuk mengambil gambar dari clipboard secara asynchronous dengan batas ukuran file tertentu.

Masih dalam folder yang sama buat sebuah file “WpfClipboardImageService.cs”, file ini berupa class yang mengimplementasikan dari interface diatas.

```
using System.IO;
using System.Windows;
using System.Windows.Media.Imaging;

namespace AzureFoundry.DocAssistant.Services;

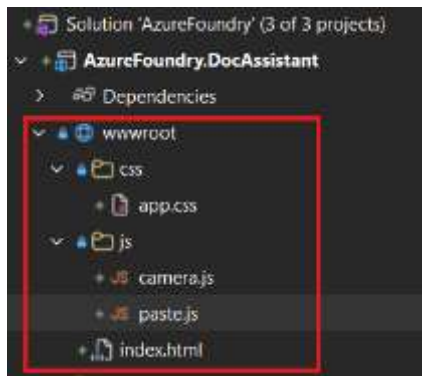
public sealed class WpfClipboardImageService : IClipboardImageService
{
    public async Task<ClipboardImage?> GetImageAsync(long maxBytes)
    {
        //sintaks detail pada project lampiran
    }

    private static ClipboardImage? GetImage(long maxBytes)
    {
        //sintaks detail pada project lampiran
    }
}
```

Sintaks diatas berfungsi untuk mengambil gambar dari clipboard Windows, misalnya gambar yang di-copy menggunakan Ctrl + C. Method *GetImageAsync(long maxBytes)* mengecek batas ukuran gambar, mengambil Dispatcher dari aplikasi WPF, lalu menjalankan proses pengambilan gambar melalui *GetImage(maxBytes)* di thread yang sesuai.

Pada method *GetImage(long maxBytes)*, code mengecek apakah clipboard berisi gambar. Jika ada, gambar diubah ke format PNG menggunakan *PngBitmapEncoder* dan disimpan sementara di *MemoryStream*. Ukurannya kemudian dicek, dan jika melebihi batas akan muncul *InvalidImageException*. Kalau berhasil, gambar dikembalikan dalam bentuk *ClipboardImage* yang berisi data gambar, MIME type, dan nama file berdasarkan waktu pengambilan.

- Lanjutkan dengan membuat folder “wwwroot”



Buat struktur folder dan file seperti pada gambar. Untuk sintaks detail pada setiap file dapat dilihat pada project lampiran.

- Setelah selesai dengan folder `wwwroot`, kita lanjutkan dengan membuat folder “Components”. Lalu buat dua buah razor file dengan nama masing-masing `_Imports.razor` dan `Routes.razor`. `_Imports.razor` berfungsi untuk menyimpan daftar `@using` yang bisa digunakan oleh banyak file `.razor`, jadi kita nggak perlu menulis `@using` yang sama berulang kali di setiap component. Sementara itu, `Routes.razor` berfungsi untuk mengatur routing di aplikasi Blazor Hybrid, yaitu menentukan component mana yang harus ditampilkan berdasarkan URL yang sedang diakses. File ini biasanya menggunakan `Router`, `Found`, dan `NotFound` untuk menangani halaman yang ditemukan atau URL yang tidak memiliki halaman yang sesuai. Untuk sintaks detail pada dua file tersebut dapat dilihat pada project lampiran.

Lau buat folder “Layout” dan tambahkan sebuah file razor “MainLayout.razor”. `MainLayout.razor` berfungsi sebagai layout utama yang digunakan bersama oleh beberapa halaman, seperti untuk menampilkan *sidebar*, *navbar*, dan menu navigasi. `@Body` menjadi tempat untuk menampilkan konten halaman yang sedang dibuka.

```
@inherits LayoutComponentBase
```

```
<main class="container py-4">
    @Body
</main>
```

Selanjutnya tambahkan folder “Pages” dan didalam folder tersebut buat sebuah file razor dengan nama “ScanDocument.razor”.

```
@page "/"
@inject DocumentScanService Scan
@inject IDocumentStore Store
@inject IClipboardImageService ClipboardSvc
@inject IJSRuntime JS

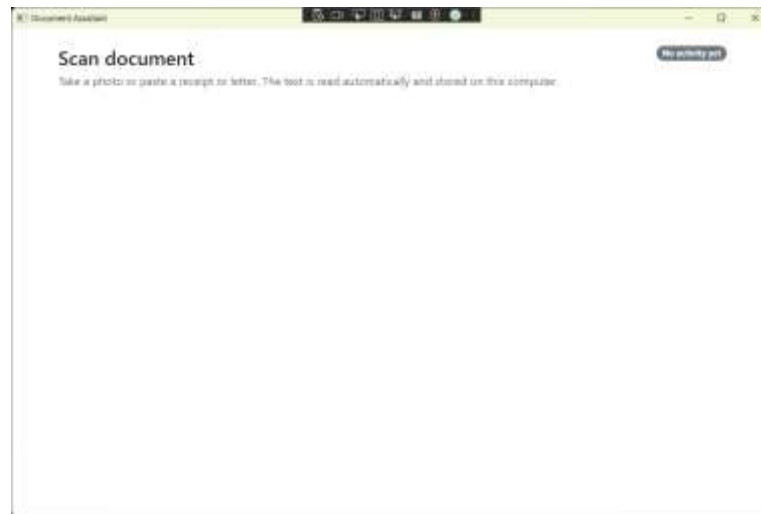
<div class="d-flex justify-content-between align-items-start gap-3 mb-4">
    <div>
        <h1 class="h3 mb-1">Scan document</h1>
        <p class="text-body-secondary mb-0">Take a photo or paste a receipt or letter. The text is read automatically and stored on this computer.</p>
    </div>
    <StatusBadge Status="_status" />
</div>

@code {
    private ProcessStatus _status = ProcessStatus.Idle;
}
```

Sintaks diatas digunakan untuk membuat halaman Scan Document, di mana user bisa melakukan scan dokumen lewat foto atau paste gambar struk dan surat. Bagian `@page "/"` menentukan alamat halaman, sedangkan `@inject` mengambil service yang

dibutuhkan untuk memproses scan, menyimpan dokumen, mengambil gambar dari clipboard, dan menjalankan JavaScript. Untuk bagian `<div>` mengatur tampilan judul, deskripsi, dan status scan agar rapi. Sementara itu, `@code` mendefinisikan `_status` dengan nilai awal `ProcessStatus.Idle`, yang menandakan proses scan belum dimulai. Untuk sintaks detail “StatusBadger” dapat dilihat pada folder “Shared” file “StatusBadge.razor”

Jalankan program untuk melihat hasilnya. Pastikan mendapatkan hasil seperti gambar dibawah.

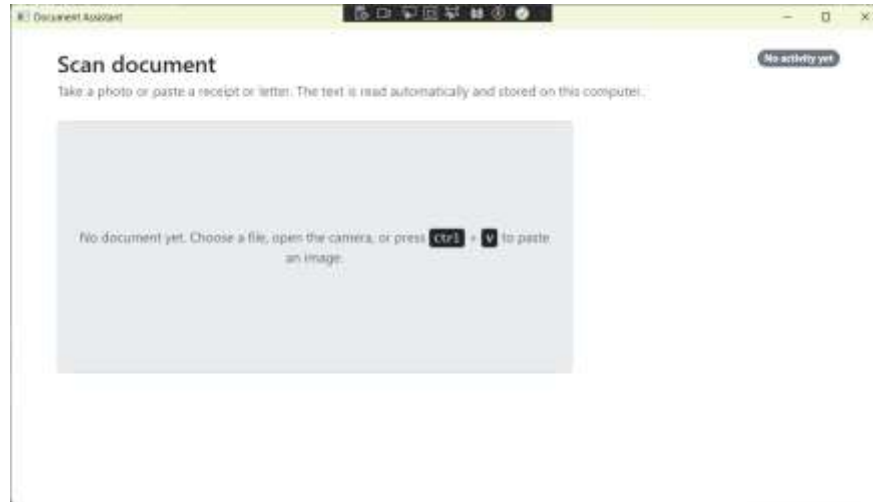


Selanjutnya kita tambahkan sintaks untuk preview gambar, jika user melakukan upload atau mengambil gambar menggunakan kamera.

```
<div class="row g-4">
  <section class="col-lg-8" aria-label="Preview and result">
    <div class="card">
      <div class="preview-area bg-body-secondary d-flex align-items-center
        justify-content-center p-3">
        @if (_cameraActive)
        {
          <video id="@VideoId" class="preview-media bg-black" autoplay
            playsinline muted></video>
        }
        else if (_previewSrc is not null)
        {
          
        }
        else
        {
          <p class="text-body-secondary text-center mb-0">No document
            yet. Choose a file, open the camera, or press
            <kbd>Ctrl</kbd> + <kbd>V</kbd> to paste an image.</p>
        }
      </div>
    </div>
  </section>
</div>
```

Sintaks ini digunakan untuk menampilkan preview dokumen yang akan di-scan. Bagian *row* dan *col-lg-8* mengatur layout agar area preview terlihat rapi. Lalu, *@if* (*_cameraActive*) mengecek apakah kamera sedang aktif; kalau iya, elemen *<video>* akan menampilkan video dari kamera. Kalau kamera tidak aktif tetapi *_previewSrc* memiliki nilai, elemen ** akan menampilkan gambar dokumen yang dipilih. Jika keduanya tidak terpenuhi, aplikasi menampilkan pesan bahwa belum ada dokumen dan user bisa memilih file, membuka kamera, atau menekan *Ctrl + V* untuk paste gambar. Untuk back end code tambahkan sintak berikut. Lalu jalankan program untuk melihat hasilnya.

```
private const string VideoId = "doc-camera";
private string? _previewSrc=null;
private ProcessStatus _status = ProcessStatus.Idle;
private bool _cameraActive=false;
```



Selanjutnya kita lanjutkan dengan sintaks pada UI untuk membuat beberapa button untuk proses scanning. Tapi saat ini kita akan focus pada komponen “InputFile”. Ketikkan seperti pada sintaks di bawah.

```
<div class="card-body">
  @if (_cameraActive)
  {
    <div class="d-flex flex-wrap gap-2">
      <button type="button" class="btn btn-primary"
        disabled="@IsProcessing">Capture and read</button>
      <button type="button" class="btn btn-link"
        disabled="@IsProcessing">Cancel</button>
    </div>
  }
  else
  {
    <div class="d-flex flex-wrap gap-2">
      <label class="btn btn-primary mb-0" aria-disabled="@IsProcessing">
        Choose file
        <InputFile class="d-none" OnChange="OnFileSelectedAsync"
          accept=".jpg, .jpeg, .png, .webp"
          disabled="@IsProcessing" />
      </label>
      <button type="button" class="btn btn-outline-secondary"
        disabled="@IsProcessing">Paste from clipboard</button>
      <button type="button" class="btn btn-outline-secondary"
        disabled="@IsProcessing">Open camera</button>
    </div>
    <p class="form-text mb-0 mt-2">To capture the screen, press
      <kbd>Win</kbd> + <kbd>Shift</kbd> + <kbd>S</kbd>, then <kbd>Ctrl</kbd> +
      <kbd>V</kbd> here.</p>
  }
}
```

```
@if (IsProcessing)
{
    <div class="progress mt-3" role="progressbar" aria-label="Reading
document">
        <div class="progress-bar progress-bar-striped progress-bar-animated
w-100"></div>
    </div>
}

@if (_errorMessage is not null)
{
    <div class="alert alert-danger mt-3 mb-0"
role="alert">@_errorMessage</div>
}

@if (_selected is { Status: DocumentStatus.Unread } && !IsProcessing &&
!_cameraActive)
{
    <button type="button" class="btn btn-primary mt-3" >Read this document
again</button>
}
</div>
```

Berikut adalah penjelasan sintaks di atas, yang digunakan untuk mengatur tombol dan informasi yang muncul di bagian bawah preview dokumen. Jika `_cameraActive` bernilai true, akan muncul button “Capture and read” dan “Cancel”. Kalau kamera tidak aktif, user bisa memilih file gambar, paste gambar dari clipboard, atau membuka kamera. Properti `disabled="@IsProcessing"` digunakan agar button tidak bisa digunakan saat proses sedang berjalan. Bagian `@if (IsProcessing)` menampilkan progress bar, sedangkan `_errorMessage` menampilkan pesan error jika terjadi masalah. Terakhir, tombol Read this document again muncul jika dokumen berstatus *Unread* dan proses scan maupun kamera sedang tidak aktif.

```
private const long MaxSizeBytes = 10 * 1024 * 1024; // 10 MB
private static readonly TimeSpan _ocrTimeout = TimeSpan.FromSeconds(90);
private static readonly string[] _supportedMediaTypes = ["image/jpeg",
"image/png", "image/webp"];
private string? _errorMessage;
private Document? _selected;
private List<Document> _history = [];
private bool IsProcessing => _status == ProcessStatus.Processing;
```

```
private async Task OnFileSelectedAsync(InputFileChangedEventArgs e)
{
    var file = e.File;
    var mediaType = MediaTypeFor(file);

    if (!_supportedMediaTypes.Contains(mediaType))
    {
        _status = ProcessStatus.Failed;
        _errorMessage = "File format is not supported. Use jpg, png, or webp.";
        return;
    }

    if (file.Size > MaxSizeBytes)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = "Maximum file size is 10 MB.";
        return;
    }

    using var ms = new MemoryStream();
    await using (var stream = file.OpenReadStream(MaxSizeBytes))
    {
        await stream.CopyToAsync(ms);
    }

    await ProcessImageAsync(ms.ToArray(), mediaType, file.Name);
}
```

Untuk menangani file gambar yang dipilih user. Pertama, *e.File* mengambil file dan *MediaTypeFor(file)* menentukan tipe file tersebut. Selanjutnya, code mengecek apakah format file didukung dan ukurannya tidak lebih dari 10 MB. Kalau tidak sesuai, *_status* diubah menjadi *Failed* dan *_errorMessage* menampilkan pesan error. Jika file valid, isinya dibaca menggunakan *MemoryStream*, lalu diubah menjadi byte array melalui *ms.ToArray()*. Terakhir, *ProcessImageAsync()* dipanggil untuk memproses gambar berdasarkan isi file, tipe media, dan nama file.

```
private async Task ProcessImageAsync(byte[] bytes, string mediaType, string name)
{
    _errorMessage = null;
    _status = ProcessStatus.Processing;
    _selected = null;
    _previewSrc = DataUri(bytes, mediaType);
    StateHasChanged();

    try
    {
        var document = await Scan.SaveAsync(bytes, mediaType, name);
        _history.Insert(0, document);
        _selected = document;
        StateHasChanged();

        //await RunOcrAsync(document);
    }
    catch (StorageException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
    }
}

private static string DataUri(byte[] bytes, string mediaType) =>
    $"data:{mediaType};base64,{Convert.ToBase64String(bytes)}";

private static string MediaTypeFor(IBrowserFile file)
{
    if (!string.IsNullOrEmpty(file.ContentType))
    {
        return file.ContentType.ToLowerInvariant();
    }

    return System.IO.Path.GetExtension(file.Name).ToLowerInvariant() switch
    {
        ".jpg" or ".jpeg" => "image/jpeg",
        ".png" => "image/png",
        ".webp" => "image/webp",
        _ => "",
    };
}
```

Pada *ProcessImageAsync()*, aplikasi akan menghapus pesan error, mengubah status menjadi Processing, lalu menampilkan preview gambar melalui *_previewSrc*. Setelah itu, *Scan.SaveAsync()* menyimpan gambar dan hasilnya dimasukkan ke *_history* serta *_selected*. Jika terjadi *StorageException*, status berubah menjadi Failed dan pesan error ditampilkan. Method *DataUri()* digunakan untuk mengubah gambar menjadi format Base64 agar bisa ditampilkan sebagai preview, sedangkan *MediaTypeFor()* bertugas untuk menentukan tipe gambar berdasarkan ContentType atau ekstensi file seperti JPG, PNG, dan WEBP.



Jika berhasil, hasilnya akan terlihat seperti pada gambar di atas. Selanjutnya, kita akan membuat section History di sebelah gambar preview. History ini berguna untuk melihat dokumen apa saja yang pernah kita scan. Selain itu, kita juga bisa melakukan re-scan langsung dari History tanpa perlu meng-upload dokumen kembali. Untuk membuat History ini, ketikkan sintaks di bawah dengan benar.

```
<aside class="col-lg-4" aria-label="Scan history">
  <div class="card">
    <div class="card-header">
      <h2 class="h6 mb-0">History</h2>
    </div>
    @if (_history.Count == 0)
    {
      <div class="card-body text-body-secondary">
        Documents you scan appear here and remain after the application
        is closed.
      </div>
    }
    else
    {
      <ul class="list-group list-group-flush">
        @foreach (var item in _history)
        {
          <li class="list-group-item d-flex align-items-center gap-2"
            @(item.Id == _selected?.Id ? "active" : "")>
            <button type="button" class="btn btn-link text-start
              text-decoration-none flex-grow-1 p-0 text-reset"
                @onclick="() => SelectHistoryAsync(item)"
                disabled="@IsProcessing">
              <span class="d-block fw-semibold text-
                truncate">@item.Title</span>
              <small class="d-block">@item.Time.ToString("dd/MM
                HH:mm")</small>
              @if (item.Status == DocumentStatus.Unread)
              {
                <span class="badge rounded-pill text-bg-warning
                  mt-1">Not read yet</span>
              }
            </button>
            <button type="button" class="btn btn-sm btn-link text-
              danger" aria-label="Delete @item.Title"
                @onclick="() => DeleteAsync(item)"
                disabled="@IsProcessing">Delete</button>
          </li>
        }
      </ul>
    }
  </div>
</aside>
```

Jika `_history` masih kosong, maka akan muncul pesan bahwa dokumen yang di-scan akan ditampilkan di sini dan tetap tersimpan meskipun aplikasi ditutup. Jika ada dokumen, `@foreach` akan menampilkan setiap item beserta judul, waktu scan, dan status dokumennya jika belum berhasil dibaca. Dokumen yang dipilih akan diberi tampilan berbeda melalui pengecekan `item.Id == _selected?.Id`. Tombol pada judul dokumen menjalankan `SelectHistoryAsync(item)` untuk memilih dokumen dari History, sedangkan tombol Delete menjalankan `DeleteAsync(item)` untuk menghapusnya. Kedua tombol akan dinonaktifkan selama proses berlangsung melalui

IsProcessing, sehingga pengguna tidak bisa menjalankan aksi lain saat aplikasi sedang memproses dokumen.

```
protected override async Task OnInitializedAsync()
{
    try
    {
        _history = (await Store.ListAsync(DocumentType.Image, 50)).ToList();
    }
    catch (StorageException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
    }
}

private async Task SelectHistoryAsync(Document item)
{
    if (IsProcessing)
    {
        return;
    }
    _selected = item;
    _errorMessage = null;
    _status = item.Status == DocumentStatus.Ready ? ProcessStatus.Done :
    ProcessStatus.Idle;
    try
    {
        var file = await Store.GetFileAsync(item.Id);
        _previewSrc = file is null ? null : DataUri(file.Bytes, file.MediaType);
        if (file is null)
        {
            _errorMessage = "The image file for this document was not found in
            storage. Delete this document from the history and scan it again.";
        }
    }
    catch (StorageException ex)
    {
        _previewSrc = null;
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
    }
}
```

Method *OnInitializedAsync()* dijalankan saat halaman pertama kali dibuka. Di dalam *try*, *Store.ListAsync(DocumentType.Image, 50)* digunakan untuk mengambil maksimal 50 dokumen bertipe image dari storage, lalu hasilnya disimpan ke *_history* dalam bentuk list. Jika terjadi *StorageException*, aplikasi akan mengubah *_status* menjadi *Failed* dan menyimpan pesan error ke *_errorMessage*.

Method *SelectHistoryAsync(Document item)* digunakan saat user memilih dokumen dari History. Jika aplikasi sedang memproses dokumen, method langsung berhenti melalui pengecekan *IsProcessing*. Jika tidak, dokumen akan disimpan ke *_selected*, error sebelumnya di-reset, dan *_status* diatur berdasarkan status dokumen yang

dipilih. Selanjutnya, *Store.GetFilesAsync(item.Id)* digunakan untuk mengambil file dari storage, lalu *_previewSrc* diisi dengan *DataUri()* agar gambar bisa ditampilkan di preview. Jika file tidak ditemukan, aplikasi akan menampilkan pesan error. Sementara itu, jika terjadi *StorageException*, preview akan dikosongkan, status diubah menjadi *Failed*, dan pesan error disimpan ke *_errorMessage*.

```
private async Task DeleteAsync(Document item)
{
    var confirmed = await JS.InvokeAsync<bool>("confirm", $"Delete
    \"{item.Title}\" along with its reading result?");
    if (!confirmed)
    {
        return;
    }

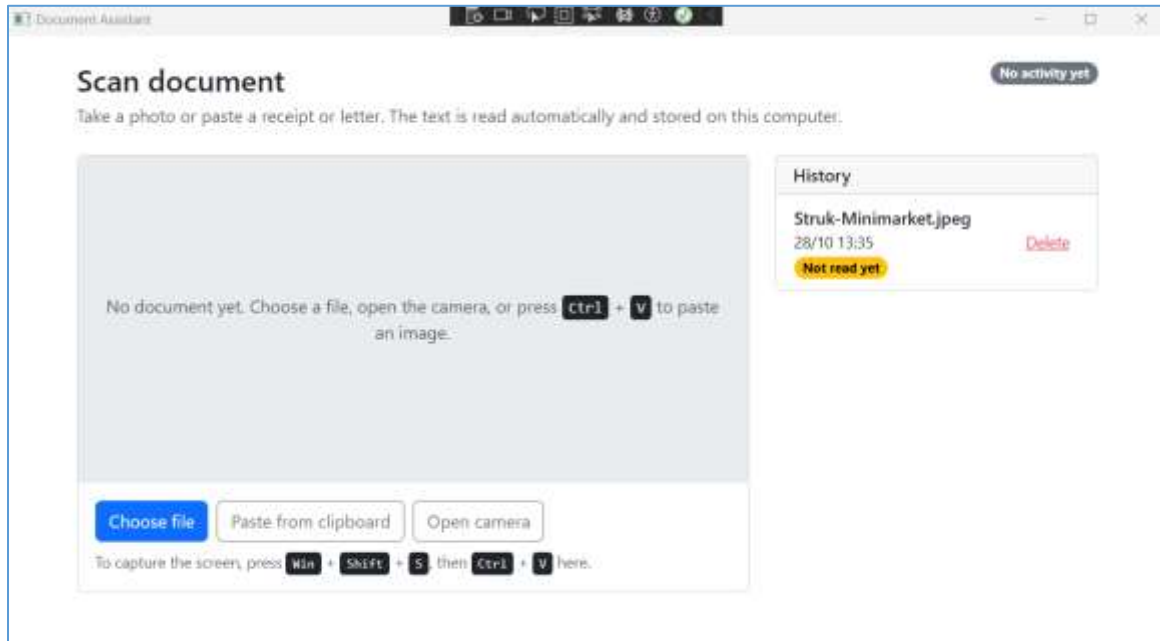
    try
    {
        await Store.DeleteAsync(item.Id);
    }
    catch (StorageException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
        return;
    }

    _history.RemoveAll(d => d.Id == item.Id);

    if (_selected?.Id == item.Id)
    {
        _selected = null;
        _previewSrc = null;
        _errorMessage = null;
        _status = ProcessStatus.Idle;
    }
}
```

Method *DeleteAsync(Document item)* digunakan untuk menghapus dokumen dari History setelah user melakukan konfirmasi. Jika berhasil, dokumen dihapus dari storage dan *_history*. Jika dokumen tersebut sedang dipilih, preview dan status akan di-reset. Jika terjadi error, aplikasi akan menampilkan pesan error.

Untuk hasil dapat dilihat pada gambar di bawah ini.



```
private async Task RunOcrAsync(Document document)
{
    try
    {
        using var cts = new CancellationTokenSource(_ocrTimeout);
        var result = await Scan.ReadAsync(document.Id, cts.Token);

        var index = _history.FindIndex(d => d.Id == result.Id);
        if (index >= 0)
        {
            _history[index] = result;
        }

        _selected = result;
        _status = ProcessStatus.Done;
    }
    catch (OcrException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message + " The document has been saved. Press \"Read this document again\" to retry.";
    }
    catch (StorageException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
    }
    catch (OperationCanceledException)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = "Reading took too long and was cancelled. The document has been saved. Press \"Read this document again\" to retry.";
    }
}
```

Method *RunOcrAsync(Document document)* digunakan untuk menjalankan proses OCR pada dokumen. *CancellationTokenSource* membatasi waktu proses sesuai *_ocrTimeout*, lalu *Scan.ReadAsync()* membaca teks dari dokumen. Jika berhasil, data di *_history* diperbarui, *_selected* diisi dengan hasil terbaru, dan status diubah menjadi *Done*. Jika terjadi error atau proses melebihi batas waktu, status menjadi *Failed* dan *_errorMessage* diisi dengan pesan yang sesuai.

Lalu letakkan method *RunOcrAsync* kedalam method *ProcessImageAsync* seperti pada gambar di bawah.

```
private async Task ProcessImageAsync(byte[] bytes, string mediaType, string name)
{
    _errorMessage = null;
    _status = ProcessStatus.Processing;
    _selected = null;
    _previewSrc = DataUri(bytes, mediaType);
    StateHasChanged();

    try
    {
        var document = await Scan.SaveAsync(bytes, mediaType, name);
        _history.Insert(0, document);
        _selected = document;
        StateHasChanged();

        await RunOcrAsync(document);
    }
    catch (StorageException ex)
    {
        _status = ProcessStatus.Failed;
        _errorMessage = ex.Message;
    }
}
```

Selanjutnya untuk sintaks yang lain seperti “OpenCamera”, “PasteFromClipboardAsync” dan lain-lain dapat dilihat project lampiran. Pada artikel ini hanya menjelaskan fungsi-fungsi utama dari aplikasi ini. Jalankan project, lalu upload sebuah dokumen seperti struk belanja. Pastikan hasil dari OCR berhasil seperti pada gambar di bawah ini.

Scan document

Take a photo or paste a receipt or letter. The text is read automatically and stored on this computer.



Choose file

Paste from clipboard

Open camera

To capture the screen, press **Win** + **Shift** + **S** then **Ctrl** + **V** here

Reading result

Copy text

MR
PART MINA & RETAIL
Jl. Jend. Sudirman No. 12, Jakarta Selatan
(021) 1234 5678

Tgl: 18 Okt 2025 No. Trans: MR-00123
Waktu: 10:15 WIB
Kasir: Citra

2x Indomie Goreng Spesial	Rp. 11.000
1x Teh Botol Sosro	Rp. 6.500
1x Gula Pasir Ing	Rp. 17.000
2x Air Mineral Vit 600ml	Rp. 8.000
1x Rotok M3id 3D	Rp. 20.500
1x Tissue Basah	Rp. 13.000

TERIMA KASIH ATAS KEBERHUKAN ANDA
Info Kontak: 021-1234567890-30
www.mrpartminaretail.id

History

Struk-Minimarket.jpeg
28/10 14:51

Struk-Minimarket.jpeg
28/10 13:35

Read

Delete

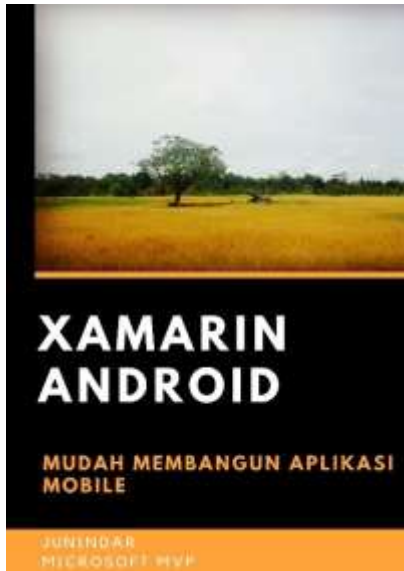
Read read pdf

Penutup

Sedangkan untuk memudahkan dalam memahami isi artikel, maka penulis juga menyertakan dengan full source code project latihan ini, dan dapat di download disini

<https://junindar.blogspot.com/2026/10/aplikasi-desktop-ocr-berbasis-ai-dengan.html>

Referensi



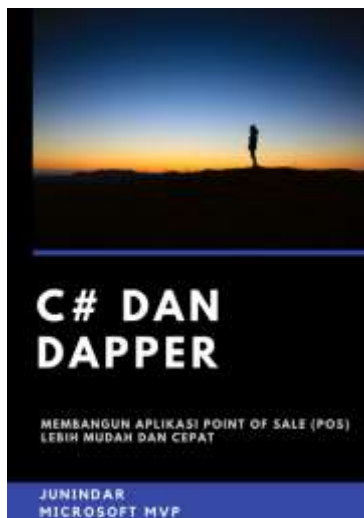
<https://play.google.com/store/books/details?id=G4tFDgAAQBAJ>



<https://play.google.com/store/books/details?id=VSLiDQAAQBAJ>



https://play.google.com/store/books/details/Junindar_Xamarin_Forms?id=6Wg-DwAAQBAJ



https://play.google.com/store/books/details/Junindar_C_dan_Dapper_Membangun_Aplikasi_POS_Point?id=6TErDwAAQBAJ



<https://play.google.com/store/books/details/Junindar ASP NET MVC Membangun Aplikasi Web Lebih?id=XLlyDwAAQBAJ>



<https://play.google.com/store/books/details/Junindar ASP NET CORE MVC?id=x Ee5DwAAQBAJ>

Biografi Penulis.



Junindar Lahir di Tanjung Pinang, 21 Juni 1982. Menyelesaikan Program S1 pada jurusan Teknik Inscreenatika di Sekolah Tinggi Sains dan Teknologi Indonesia (ST-INTEN-Bandung). Junindar mendapatkan Award Microsoft MVP VB pertanggal 1 oktober 2009 hingga saat ini. Senang mengutak-atik computer yang berkaitan dengan bahasa pemrograman. Keahlian, sedikit mengerti beberapa bahasa pemrograman seperti : VB.Net, C#, SharePoint, ASP.NET, VBA. Reporting: Crystal Report dan Report Builder. Database: MS Access, MY SQL dan SQL Server. Simulation / Modeling Packages: Visio Enterprise, Rational Rose dan Power Designer. Dan senang bermain gitar, karena untuk bisa menjadi pemain gitar dan seorang programmer sama-sama membutuhkan seni. Pada saat ini bekerja di salah satu Perusahaan Consulting dan Project Management di Malaysia sebagai Senior Consultant. Memiliki beberapa sertifikasi dari Microsoft yaitu Microsoft Certified Professional Developer (MCPD – SharePoint 2010), MOS (Microsoft Office Specialist) dan MCT (Microsoft Certified Trainer) Mempunyai moto hidup: **“Jauh lebih baik menjadi Orang Bodoh yang giat belajar, dari pada orang Pintar yang tidak pernah mengimplementasikan ilmunya”**.