IlmuKomputer.Com
Ikatlah ilmu dengan menuliskannya

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

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

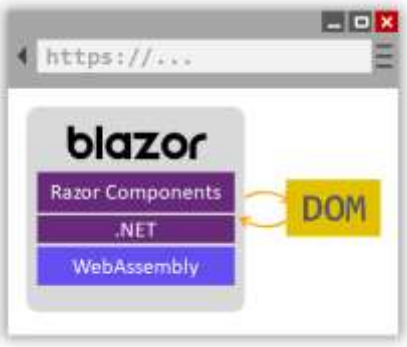
Abstrak

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.

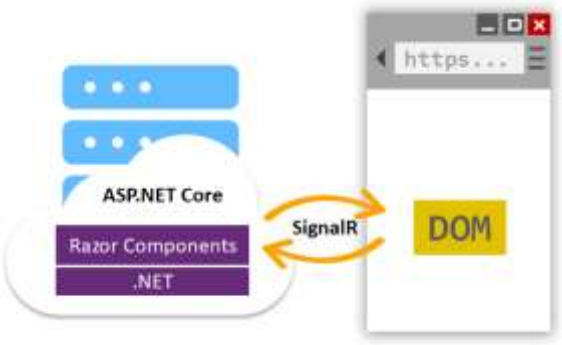
Pendahuluan

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 WebAssemby (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 dikirmkan 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.

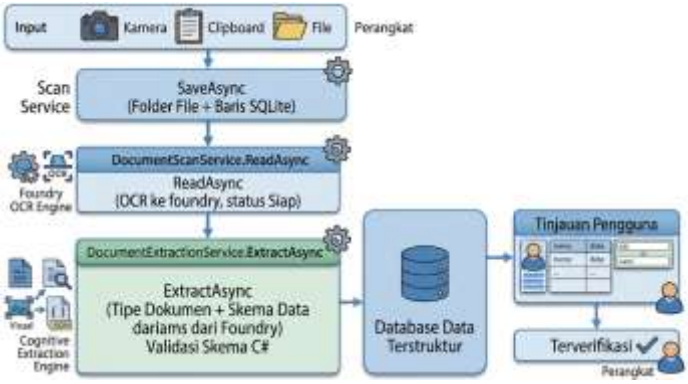


Terdapat beberapa keuntungan bagi pengembang yang ingin membangun aplikasi cross platform dengan menggunakan Blazor Hybrid seperti:

- 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.

Pada Part 1, aplikasi Document Assistant sudah bisa melakukan scanning document image dengan menggunakan file, clipboard, atau kamera dan membaca seluruh teksnya memakai model vision di Microsoft Foundry, lalu menyimpan gambar dan raw teks itu di SQLite. Hasilnya sudah bisa dibaca oleh pengguna, tapi masih berupa satu blok teks panjang, belum berupa field-field yang lebih terstruktur dan yang siap dipakai sistem. Pada Part 2 ini kita melanjutkan project yang sama dengan menambahkan tahap untuk melakukan ekstraksi data menjadi terstruktur. Aplikasi akan memperkirakan jenis dokumen (struk belanja, invoice, KTP, atau dokumen lain), lalu akan mengisi skema field sesuai jenisnya dengan menggunakan Microsoft Foundry, serta melakukan validasi dengan menggunakan aturan yang dibuat menggunakan sintaks dari C#, dilanjutkan dengan menampilkan hasilnya pada form sehingga bisa diperiksa, diedit, dan disetujui pengguna sebelum dianggap final.

Untuk Part 1, DocumentScanService.ReadAsync() berhenti pada titik Document dengan Text berisi hasil OCR dan Status menjadi Ready. Hasilnya cukup untuk ditampilkan pada panel hasil scan, tetapi belum cukup bila datanya mau dipakai untuk, seperti rekap pengeluaran. Prinsip Part 1, yaitu simpan dulu, baru baca, sekarang kita perpanjang menjadi simpan dulu, baru proses, dan simpan setiap tahap:



Alasannya sama dengan alasan OCR dipisah dari penyimpanan gambar di Part 1, panggilan ke Foundry bisa gagal kapan saja. Sehingga jika klasifikasi, pengisian skema, dan validasi digabung menjadi satu proses besar yang baru disimpan di akhir maka kegagalan di tengah jalan berarti akan menyebabkan kita harus mengulang semuanya, termasuk biaya untuk panggilan AI yang sudah berhasil. Dengan menyimpan di setiap tahap, maka dokumen yang gagal diekstrak tetap punya teks OCR-nya (status tetap Ready), dan tombol Extract data cukup mengulang dari tahap yang gagal.

Dokumen itu beragam skema atau struktur dan modelnya, seperti pada struk, memiliki daftar barang dan total, KTP punya NIK dan tempat lahir dan ada dokumen yang tidak memiliki semua field-field yang disebutkan sebelumnya. Memaksakan satu skema besar untuk semua jenis dokumen akan menghasilkan banyak field kosong dan aturan yang saling bertabrakan. Karena itu kita menggunakan konsep jenis dokumen (DocumentClassification), satu kelas kecil per jenis dokumen, yang mendefinisikan skema field-nya, aturan validasinya sendiri, dan cara membuat ringkasan dokumen. Sehingga jika kita ingin menambah jenis dokumen baru, nanti cukup menambah satu kelas, alur, form, dan penyimpanan tidak perlu diubah.

Memperluas Status Dokumen

Buka kembali Document.cs di folder Storage, project AzureFoundry.DocAssistant.Core. DocumentStatus dari Part 1 kita perluas dari dua status menjadi lima seperti pada sintaks di bawah.:

```
public enum DocumentStatus
{
    Unread = 0,
    Ready = 1,
    Structured = 2,
    NeedsReview = 3,
    Verified = 4,
}
public static class DocumentStatusExtensions
{
    public static bool HasText(this DocumentStatus status) => status !=
    DocumentStatus.Unread;

    public static bool HasStructuredData(this DocumentStatus status) =>
        status is DocumentStatus.Structured or DocumentStatus.NeedsReview
    or DocumentStatus.Verified;
}
```

Dua *extension meth*od di atas dipakai berulang kali pada seluruh aplikasi, agar pengecekan status tidak ditulis ulang sebagai *status == DocumentStatus.X || status == DocumentStatus.Y* di banyak tempat maka kita gunakan extension method di atas.

Document sendiri kita tambahkan empat properti baru: *docClassification* (hasil klasifikasi), *Data* (data terstruktur yang berlaku saat ini), *OriginalData* (hasil asli dari AI, sebelum dikoreksi pengguna yang dipakai untuk menandai field yang diedit dan tombol "Revert to AI result"), *Summary* (judul, tanggal, total, pihak, disimpan di kolom biasa agar bisa dicari

tanpa membongkar JSON), dan *HasFile* (penanda bahwa gambar aslinya sudah dihapus, khusus dokumen sensitif seperti KTP).

```
public sealed record Document(
    string Id,
    string Title,
    DocumentType Type,
    DateTime Time,
    DocumentStatus Status,
    string? Text,
    DocumentClassification? docClassification = null,
    DocumentData? Data = null,
    DocumentData? OriginalData = null,
    DocumentSummary? Summary = null,
    bool HasFile = true);

public sealed record DocumentFile(byte[] Bytes, string MediaType)
```

Skema untuk Jenis Dokumen (DocumentClassification)

Buat folder baru dengan nama "Extraction" di project Core. Di sinilah model data terstruktur dan definisi skema per-jenis dokumen diletakkan, terpisah dari folder "Ocr" dan "Storage" dari Part 1.

Tambahkan file pada folder "Extraction" dengan nama *DocumentClassification.cs* dan ketikkan sintaks di bawah.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;
public enum DocumentClassification
{
    Other = 0,
    Receipt = 1,
    Invoice = 2,
    IdCard = 3,
}
```

Lalu lanjutkan dengan file FieldSpec.cs, yang mendefinisikan satu field pada skema.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;
public enum FieldType
{
    Text,
    Date,
    Number,
    Money,
}

public sealed record FieldSpec(
    string Key,
    string Label,
    FieldType Type = FieldType.Text,
    bool Required = false,
    string? Hint = null,
    bool Sensitive = false);
```

Sensitive dipakai khusus untuk field seperti NIK atau alamat pada KTP, nilainya disamarkan di layar sampai pengguna memintanya tampil (dibahas di bagian UI).

Hasil ekstraksi dari semua jenis dokumen disimpan dalam format yang sama menggunakan class DocumentData. Class ini punya satu Dictionary untuk menyimpan field dokumen dan daftar baris untuk menampung data tabel, seperti daftar barang pada struk dan invoice.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;

public sealed class DocumentData
{
    public Dictionary<string, string?> Fields { get; init; } = [];
    public List<Dictionary<string, string?>> Items { get; init; } = [];

    public DocumentData Clone() => new()
    {
        Fields = new Dictionary<string, string?>(Fields),
        Items = [.. Items.Select(row => new Dictionary<string,
        string?>(row))],
    };

    public string? Value(string key) =>
        Fields.TryGetValue(key, out var value) &&
        !string.IsNullOrWhiteSpace(value) ? value : null;

    public bool EqualsData(DocumentData? other)
    {
        if (other is null || Items.Count != other.Items.Count)
        {
            return false;
        }

        return FieldsEqual(Fields, other.Fields) &&
        Items.Zip(other.Items).All(p => FieldsEqual(p.First, p.Second));
    }
    private static bool FieldsEqual(Dictionary<string, string?> a,
    Dictionary<string, string?> b)
    {
        foreach (var key in a.Keys.Union(b.Keys))
        {
            a.TryGetValue(key, out var x);
            b.TryGetValue(key, out var y);
            if (string.IsNullOrWhiteSpace(x) !=
                string.IsNullOrWhiteSpace(y) ||
                (!string.IsNullOrWhiteSpace(x) && x != y))
            {
                return false;
            }
        }

        return true;
    }
}
```

FieldsEqual() digunakan untuk membandingkan dua *Dictionary* apakah isinya sama atau tidak. Pertama, code akan mengambil semua *key* dari kedua dictionary, lalu mengeceknya satu per satu. Setiap *key* akan diambil value-nya menggunakan *TryGetValue*. Setelah itu,

code akan mengecek apakah value dari kedua dictionary sama-sama kosong (null, string kosong, atau hanya spasi), atau sama-sama memiliki isi dengan value yang sama persis. Kalau salah satu value kosong sementara yang lain berisi, atau keduanya berisi tapi nilainya berbeda, maka akan langsung mengembalikan false. Jika semua key sudah dicek dan tidak ditemukan perbedaan, maka hasilnya true.

Sebelum masuk ke IDocumentDefinition, satu file kecil lagi perlu disiapkan. ValidationFinding.cs, masih di folder yang sama. Isinya dua record yang dipakai bolak-balik oleh Validate() dan Summarize() pada setiap definisi jenis dokumen:

```
namespace AzureFoundry.DocAssistant.Core.Extraction;

public sealed record ValidationFinding(string? Key, string Message)
{
    public static string ItemKey(int rowNumber, string column) =>
    $"item:{rowNumber}:{column}";
}

public sealed record DocumentSummary(string? Title, DateOnly? Date,
decimal? Total, string? Party);
```

Sintaks ini mendefinisikan dua record untuk menyimpan data. ValidationFinding digunakan untuk menyimpan informasi error atau masalah validasi, sedangkan ItemKey membuat penanda berdasarkan nomor baris dan nama kolom. Sementara itu, DocumentSummary digunakan untuk menyimpan ringkasan dokumen seperti judul, tanggal, total, dan pihak terkait. Tanda ? menunjukkan bahwa nilainya boleh kosong (null).

Buat sebuah interface dengan nama IDocumentDefinition dan ketikkan sintaks seperti di bawah.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;

public interface IDocumentDefinition
{
    DocumentClassification docClassification { get; }
    string Name { get; }
    string Description { get; }
    bool Sensitive { get; }
    int SchemaVersion { get; }

    IReadOnlyList<FieldSpec> Fields { get; }
    string? ItemsName { get; }
    IReadOnlyList<FieldSpec> ItemColumns { get; }

    IReadOnlyList<ValidationFinding> Validate(DocumentData data);
    DocumentSummary Summarize(DocumentData data);
}
```

Sintaks diatas mendefinisikan sebuah interface bernama *IDocumentDefinition* yang berfungsi sebagai standar atau aturan untuk mendefinisikan sebuah dokumen dalam sistem. Interface ini menentukan informasi apa saja yang harus dimiliki oleh setiap jenis dokumen, seperti klasifikasi dokumen, nama, deskripsi, status sensitif, dan versi schema. Selain itu, *Fields* digunakan untuk menentukan field apa saja yang ada di dalam dokumen, sedangkan ItemsName dan ItemColumns digunakan untuk mengatur data yang berbentuk kumpulan item atau tabel. Ada juga dua method, yaitu *Validate()* untuk memeriksa apakah data dokumen sudah sesuai dengan aturan yang ditentukan, dan *Summarize()* untuk membuat ringkasan dari data dokumen. Jadi, sederhananya, *IDocumentDefinition* menjadi panduan agar setiap jenis dokumen memiliki struktur, aturan validasi, dan cara membuat ringkasan yang konsisten.

Kita lanjutkan dengan membuat Base Class *DocumentDefinitionBase* ketikkan seperti pada sintaks dibawah.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;

public abstract class DocumentDefinitionBase : IDocumentDefinition
{
    private const decimal _tolerance = 0.01m;

    public abstract DocumentClassification docClassification { get; }
    public abstract string Name { get; }
    public abstract string Description { get; }
    public virtual bool Sensitive => false;
    public virtual int SchemaVersion => 1;
    public abstract IReadOnlyList<FieldSpec> Fields { get; }
    public virtual string? ItemsName => null;
    public virtual IReadOnlyList<FieldSpec> ItemColumns => [];
    public abstract DocumentSummary Summarize(DocumentData data);

    public IReadOnlyList<ValidationFinding> Validate(DocumentData data)
    {
        ArgumentNullException.ThrowIfNull(data);
        var findings = new List<ValidationFinding>();

        //Detail sintaks pada project lampiran

        findings.AddRange(ValidateSpecific(data));
        return findings;
    }

    protected virtual IEnumerable<ValidationFinding>
    ValidateSpecific(DocumentData data) => [];


    protected static IEnumerable<ValidationFinding>
    ValidateArithmetic(DocumentData data)
    {
        //Detail sintaks pada project lampiran

    }

    protected static string? GetValue(Dictionary<string, string?> row,
    string key) =>
        row.TryGetValue(key, out var value) &&
        !string.IsNullOrWhiteSpace(value) ? value : null;

    protected static string CurrencyPrefix(string? currency) =>
        string.IsNullOrWhiteSpace(currency) || currency.Equals("IDR",
        StringComparison.OrdinalIgnoreCase)
            ? "Rp" : currency.Trim().ToUpperInvariant();

    private static void CheckValue(FieldSpec spec, string? value,
    string key, List<ValidationFinding> findings)
    {
        //Detail sintaks pada project lampiran


    }
}
```

Sinatks ini merupakan base class *DocumentDefinitionBase* yang berfungsi untuk melakukan validasi data dokumen sebelum diproses lebih lanjut. Class ini mengecek apakah setiap field

yang wajib diisi sudah memiliki nilai, apakah format data seperti tanggal dan angka sudah benar, serta memvalidasi data pada setiap baris Items. Selain validasi umum, terdapat *ValidateSpecific()* yang bisa di-override oleh class turunannya untuk menambahkan aturan validasi khusus sesuai jenis dokumen. Method *ValidateArithmetic()* digunakan untuk mengecek perhitungan matematika, seperti *quantity* × *unit_price* harus sesuai dengan subtotal, total seluruh subtotal harus sesuai dengan subtotal dokumen, dan *subtotal* + *tax* harus sesuai dengan total. Sementara itu, *ToDecimal(), ToDate(), GetValue(), dan CurrencyPrefix()* merupakan helper method untuk membantu proses konversi dan pengolahan nilai. Intinya, class ini menjadi fondasi validasi agar data dokumen yang diekstrak tetap lengkap, formatnya benar, dan perhitungannya sesuai.

Lalu pada folder *Extraction* tambahkan sebuah folder dengan nama *Definition*. Pada folder ini akan terdapat 4 class Definition yang menggunakan *DocumentDefinitionBase* sebagai base classnya. Sebagai contoh, tambahkan sebuah class dengan nama “ReceiptDefinition.cs“ lalu ketikkan sintaks di bawah.

```
namespace AzureFoundry.DocAssistant.Core.Extraction.Definitions;

public sealed class ReceiptDefinition(TimeProvider clock) :
DocumentDefinitionBase
{
    public override DocumentClassification docClassification =>
    DocumentClassification.Receipt;
    public override string Name => "Receipt";
    public override string Description =>
        "A receipt or sales slip from a shop, convenience store,
        restaurant, or gas station: " +
        "store name at the top, a list of items or menu items with prices,
        then the total paid.";

    public override IReadOnlyList<FieldSpec> Fields { get; } =
    [
        new("store_name", "Store name", FieldType.Text, Required: true),
        new("date", "Date", FieldType.Date, Required: true),
        new("currency", "Currency", FieldType.Text),
        new("subtotal", "Subtotal", FieldType.Money),
        new("tax", "Tax", FieldType.Money),
        new("total", "Total", FieldType.Money, Required: true),
    ];
```

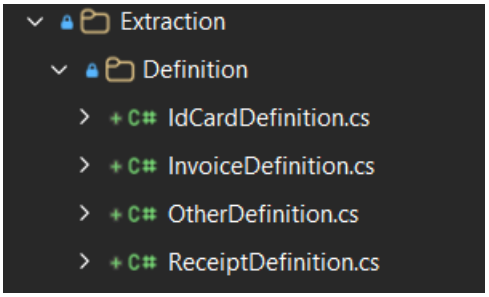
```
    public override DocumentSummary Summarize(DocumentData data)
    {
        var store = data.Value("store_name");
        var total = ToDecimal(data.Value("total"));
        var title = store is null ? null : $"Receipt from {store}";
        return new DocumentSummary(title, ToDate(data.Value("date")),
        total, store);
    }

    protected override IEnumerable<ValidationFinding>
    ValidateSpecific(DocumentData data)
    {
        var date = ToDate(data.Value("date"));
        var today = DateOnly.FromDateTime(clock.GetLocalNow().DateTime);
        if (date is not null && date.Value > today)
        {
            yield return new ValidationFinding("date", "The receipt date is
            in the future. Please check it again.");
        }

        foreach (var finding in ValidateArithmetic(data))
        {
            yield return finding;
        }
    }
}
```

Sintaks diatas merupakan class *ReceiptDefinition* yang khusus digunakan untuk mengatur struktur dan validasi dokumen receipt atau struk belanja. Class ini mewarisi *DocumentDefinitionBase*, sehingga sudah memiliki validasi dasar dari parent class. Pada bagian *Fields*, ditentukan data apa saja yang perlu diambil dari receipt, seperti *store_name, date, currency, subtotal, tax,* dan *total*, termasuk field mana yang wajib diisi. Bagian *ItemColumns* mengatur detail barang yang ada di dalam receipt, seperti nama barang, quantity, unit price, dan subtotal. Method *Summarize()* digunakan untuk membuat ringkasan dokumen berdasarkan nama toko, tanggal, dan total pembayaran. Sementara itu, *ValidateSpecific()* menambahkan validasi khusus receipt, yaitu memastikan tanggal receipt tidak berada di masa depan dan menjalankan *ValidateArithmetic()* untuk mengecek apakah perhitungan harga dan totalnya sudah sesuai. Intinya, class ini mengatur data apa saja yang harus diekstrak dari receipt, bagaimana ringkasannya dibuat, dan bagaimana memastikan datanya valid.

Untuk 3 class definition yang lain dapat dilihat sintaks detailnya pada project lampiran. Pastikan setelah membuat ketiga class tersebut tidak ada error pada saat di-Build.



Setelah selesai dengan ke-empat definition class diatas, selajutnya kita daftarkan ke-empat class tersebut kedalam satu Registry. Buat sebuah class seperti dibawah.

```
namespace AzureFoundry.DocAssistant.Core.Extraction;

public sealed class DocumentDefinitionRegistry
{
    private readonly Dictionary<DocumentClassification, IDocumentDefinition>
    _map;

    public DocumentDefinitionRegistry(IEnumerable<IDocumentDefinition>
    definitions)
    {
        All = [.. definitions.OrderBy(d => d.docClassification)];
        _map = All.ToDictionary(d => d.docClassification);

        if (!_map.ContainsKey(DocumentClassification.Other))
        {
            throw new InvalidOperationException("A definition for 'Other'
            is required as a safety net.");
        }
    }

    public IReadOnlyList<IDocumentDefinition> All { get; }

    public IDocumentDefinition Get(DocumentClassification
    docClassification) => _map.TryGetValue(docClassification, out var d) ? d :
    _map[DocumentClassification.Other];
}
```

Class ini berfungsi untuk mengelola semua *DocumentDefinition* berdasarkan jenis dokumennya. Data definition disimpan dalam Dictionary *_map* agar mudah dicari. Method *Get()* digunakan untuk mengambil definition sesuai *docClassification*. Jika jenis dokumen tidak ditemukan, sistem akan menggunakan *Other* sebagai fallback. Intinya, class ini membantu sistem menentukan aturan yang tepat untuk setiap jenis dokumen.

Dan di *CoreServiceCollectionExtensions* dari Part 1, kita daftarkan setiap *IDocumentDefinition* sebagai satu baris terpisah seperti pada sintaks dibawah ini.

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

        services.TryAddSingleton(TimeProvider.System);
        services.AddSingleton<IDocumentDefinition, ReceiptDefinition>();
        services.AddSingleton<IDocumentDefinition, InvoiceDefinition>();
        services.AddSingleton<IDocumentDefinition, IdCardDefinition>();
        services.AddSingleton<IDocumentDefinition, OtherDefinition>();
        services.AddSingleton<DocumentDefinitionRegistry>();
        services.AddSingleton<DocumentExtractionService>();
        return services;
    }
}
```

Mengikuti pola *DocumentScanService* dari Part 1, *DocumentExtractionService* mengatur alur klasifikasi, ekstrak danvalidasi, sekaligus menyediakan method untuk alur review oleh user (koreksi, setuju, hapus gambar):

```
public sealed class DocumentExtractionService(
IDocumentStore store, DocumentScanService scan,
IDocumentClassifier classifier,IDocumentExtractor extractor,
DocumentDefinitionRegistry registry)
{

    public const double ConfidenceThreshold = 0.6;

    public async Task<Document> ExtractAsync(string id, DocumentClassification?
    chosenClassification = null,  bool force = false,
    CancellationToken ct = default)
    {
        var document = await scan.ReadAsync(id, ct);
        //Sintaks detail pada project lampiran
    }

    public async Task<Document> SaveCorrectionAsync(string id,
    DocumentData data, CancellationToken ct = default)
    {
        var document = await GetOrThrowAsync(id, ct);
        //Sintaks detail pada project lampiran
    }

    public async Task<Document> ApproveAsync(string id,
    CancellationToken ct = default)
    {
        await store.ChangeStatusAsync(id, DocumentStatus.Verified, ct);
        return await GetOrThrowAsync(id, ct);
    }

    public async Task<Document> DeleteOriginalImageAsync(string id,
    CancellationToken ct = default)
    {
        var document = await GetOrThrowAsync(id, ct);
        //Sintaks detail pada project lampiran
        return await GetOrThrowAsync(id, ct);
    }
```

```
    public IReadOnlyList<ValidationFinding> Check(DocumentClassification
    docClassification, DocumentData data)
    {
        var definition = registry.Get(docClassification);
        return definition.Validate(data.Normalized(definition));
    }

    private async Task<DocumentClassification> GuessClassificationAsync(string
    text, CancellationToken ct)
    {
       //Sintaks detail pada project lampiran
    }

    //Sintaks detail pada project lampiran
}
```

DocumentExtractionService yang mengatur proses utama ekstraksi data dari dokumen. Pada method *ExtractAsync()*, sistem akan membaca dokumen, mengecek apakah data sudah pernah diekstrak, lalu mengambil file asli untuk diproses. Jika jenis dokumen belum ditentukan, *classifier* akan menebak jenis dokumen berdasarkan teks OCR. Setelah itu, *extractor* mengambil data sesuai definition yang dipilih, kemudian data dinormalisasi dan divalidasi. Hasil ekstraksi disimpan dengan status *Structured* jika tidak ada masalah, atau *NeedsReview* jika masih ada kesalahan.

Selain itu, terdapat beberapa method lain, yaitu *SaveCorrectionAsync()* untuk menyimpan perbaikan data, *ApproveAsync()* untuk menyetujui dokumen, *DeleteOriginalImageAsync()* untuk menghapus gambar asli setelah dokumen diverifikasi, dan *Check()* untuk mengecek validasi data. Method *GuessClassificationAsync()* digunakan untuk menentukan jenis dokumen berdasarkan confidence score, sedangkan *DefinitionFor()* dan *GetOrThrowAsync()* membantu mengambil definition dan dokumen yang dibutuhkan. Intinya, class ini mengatur seluruh alur ekstraksi, validasi, perbaikan, hingga persetujuan dokumen.

IDocumentStore dari Part 1 perlu ditambah beberapa method, cek pada project lampiran untuk detail sintaksnya.

Selanjutnya pada AzureFoundry.DocAssistant.Infrastructure folder Foundry tambahkan dua buah file “FoundryServiceBase.cs“ dan “FoundryDocumentClassifier.cs“ lalu ketikkan masing-masing class seperti dibawah.

```
public abstract class FoundryServiceBase(IChatClientFactory factory)
{
    private IChatClient? _client;

    private static readonly ChatOptions _jsonOptions = new() { ResponseFormat =
    ChatResponseFormat.Json };

    protected async Task<string> AskJsonAsync(IList<ChatMessage> messages,
    CancellationToken ct)
    {
        try
        {
            _client ??= factory.Create();
            var response = await _client.GetResponseAsync(messages,
            _jsonOptions, ct);
            var text = response.Text.Trim();

            return text.Length > 0
                ? text
                : throw new ExtractionException("The model did not return an
                  answer. Try processing it again.");
        }
        catch (OperationCanceledException) { throw; }
        catch (ExtractionException) { throw; }
        catch (AiException ex) { throw new ExtractionException(ex.Message, ex); }
        catch (Exception ex) {
              throw new
             ExtractionException(factory.GetErrorMessage(ex), ex); }
    }

    protected static string Truncate(string text, int maxChars) =>
        text.Length <= maxChars ? text : text[..maxChars];
}



public sealed class FoundryDocumentClassifier(IChatClientFactory factory)
    : FoundryServiceBase(factory), IDocumentClassifier
{
    internal const int MaxChars = 4_000;

    public async Task<ClassificationResult> ClassifyAsync(
        string text, IReadOnlyList<IDocumentDefinition> candidates,
        CancellationToken ct = default)
    {
        List<ChatMessage> messages =
        [
            new(ChatRole.System, DocumentPrompt.Classification(candidates)),
            new(ChatRole.User, "Document text:\n" + Truncate(text, MaxChars)),
        ];

        return AnswerParser.ParseClassification(candidates, await
        AskJsonAsync(messages, ct));
    }
}
```

FoundryServiceBase merupakan base class yang digunakan untuk menghubungkan aplikasi dengan AI model melalui *IChatClientFactory*. Method *AskJsonAsync()* bertugas mengirim messages ke AI dan meminta response dalam format JSON. Jika response kosong atau

terjadi error, sistem akan menangani error tersebut dan mengubahnya menjadi *ExtractionException*. Sementara itu, *Truncate()* digunakan untuk membatasi panjang teks agar tidak melebihi jumlah karakter yang ditentukan.

FoundryDocumentClassifier merupakan class yang bertugas menentukan jenis dokumen menggunakan AI. Class ini mewarisi *FoundryServiceBase* agar bisa menggunakan koneksi dan method yang sudah tersedia. Pada *ClassifyAsync()*, teks dokumen dibatasi maksimal 4.000 karakter, lalu dikirim ke AI bersama *DocumentPrompt.Classification* yang berisi instruksi dan pilihan jenis dokumen. Setelah AI memberikan response dalam format JSON, *AnswerParser.ParseClassification()* akan membaca hasilnya dan menentukan classification dokumen.

Lalu kita lanjutkan dengan membuat file "FoundryDocumentExtractor.cs" yang sintaks detailnya bisa dilihat pada project lampiran. *FoundryDocumentExtractor* berfungsi untuk mengambil data dari gambar dokumen menggunakan AI. Method *ExtractAsync()* mengirimkan instruksi, teks OCR (maksimal 20.000 karakter), dan gambar dokumen ke AI. Hasilnya berupa JSON yang kemudian diproses oleh *AnswerParser.ParseData()* menjadi DocumentData sesuai struktur dokumen yang ditentukan.

Yang menarik, isi instruksinya (prompt) sengaja diletakkan di Core, bukan di Infrastructure, lewat kelas statis *DocumentPrompt*. *DocumentPrompt* merupakan class yang menyimpan instruksi (prompt) untuk AI di bagian Core. Prompt dibuat berdasarkan struktur field dari *IDocumentDefinition*, sehingga tidak bergantung pada Azure Foundry. Dengan begitu, isi prompt bisa diuji tanpa harus memanggil AI, cukup memastikan semua field yang dibutuhkan sudah tercantum di dalam instruksi. Untuk sintaks detailnya ada pada project lampiran.

Lalu kita daftarkan semuanya di *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.AddSingleton<IDocumentClassifier, FoundryDocumentClassifier>();
        services.AddSingleton<IDocumentExtractor, FoundryDocumentExtractor>();

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

Memperbarui SQLite Tanpa Kehilangan Data Lama

PRAGMA *user_version* merupakan fitur bawaan SQLite yang digunakan untuk menyimpan versi database. Fitur ini membantu sistem mengetahui versi database yang sedang digunakan, sehingga perubahan struktur tabel bisa dilakukan tanpa menghapus data atau riwayat dokumen yang sudah ada dari versi sebelumnya.

Di SqliteDocumentStore.cs, tambahkan konstanta versi dan skrip migrasinya.

```
private const int _schemaVersion = 2;

private const string _migrateToV2 = """
ALTER TABLE documents ADD COLUMN kind INTEGER NULL;
ALTER TABLE documents ADD COLUMN schema_version INTEGER NULL;
ALTER TABLE documents ADD COLUMN data_json TEXT NULL;
ALTER TABLE documents ADD COLUMN original_data_json TEXT NULL;
ALTER TABLE documents ADD COLUMN document_date TEXT NULL;
ALTER TABLE documents ADD COLUMN total TEXT NULL;
ALTER TABLE documents ADD COLUMN party TEXT NULL;
CREATE INDEX IF NOT EXISTS ix_documents_document_date ON documents
(document_date);
""";
```

Lalu method *ApplyMigrationAsync()* dipanggil sekali di *InitializeAsync()*, sesudah *CREATE TABLE IF NOT EXISTS* dijalankan.

```
private static async Task ApplyMigrationAsync(SqliteConnection connection,
CancellationToken ct)
{
     long version;
     await using (var read = connection.CreateCommand())
     {
         read.CommandText = "PRAGMA user_version";
         version = Convert.ToInt64(await read.ExecuteScalarAsync(ct),
         CultureInfo.InvariantCulture);
     }

     if (version >= _schemaVersion)
     {
         return;
     }

     await using var transaction = (SqliteTransaction)await
     connection.BeginTransactionAsync(ct);
     await using (var cmd = connection.CreateCommand())
     {
         cmd.Transaction = transaction;

         if (version < 2)
         {
             cmd.CommandText = _migrateToV2;
             await cmd.ExecuteNonQueryAsync(ct);
         }
         cmd.CommandText = $"PRAGMA user_version = {_schemaVersion}";
         await cmd.ExecuteNonQueryAsync(ct);
     }
    await transaction.CommitAsync(ct);
}
```

ApplyMigrationAsync() berfungsi untuk memperbarui versi database SQLite. Pertama, sistem membaca versi database saat ini melalui PRAGMA *user_version*. Jika versinya sudah sesuai, proses langsung dihentikan. Jika belum, sistem menjalankan migration sesuai versi, misalnya *_migrateToV2*, lalu memperbarui angka versi database. Semua proses dilakukan dalam satu *transaction* agar perubahan tersimpan dengan aman dan tidak setengah jalan.

Perbarui konstanta (Const) *DocumentColumns*

```
private const string _documentColumns =
"id, title, type, timestamp, status, text, kind, data_json, original_data_json,
document_date, total, party, file_name";
```

Summary hanya dibuat jika dokumen sudah memiliki *DocumentClassification*, sehingga dokumen yang belum diekstrak tetap memiliki Summary bernilai null. Sementara itu, *ParseStoredData()* digunakan untuk membaca data JSON dan mengembalikan null jika terjadi error, agar satu data yang rusak tidak membuat seluruh riwayat dokumen gagal dimuat. Method *GetAsync()* dan *ListAsync()* juga perlu menggunakan *DocumentColumns*

dan *Map()* terbaru agar semua kolom dokumen bisa terbaca. Sedangkan *SaveExtractionAsync()* berfungsi menyimpan hasil ekstraksi pertama, menyimpan data aslinya ke *original_data_json* untuk fitur *Revert to AI result*, serta memperbarui judul dokumen berdasarkan ringkasan jika tersedia. Untuk sintaks detail pada file ”SqliteDocumentStore.cs” dapat dilihat pada project lampiran.

Bekerja pada project “AzureFoundry.DocAssistant”

Sebelum menyentuh *ScanDocument.razor*, buka dulu *_Imports.razor* di folder Components dan tambahkan satu baris *@using* untuk namespace baru yang kita buat sebelumnya.

```
@using AzureFoundry.DocAssistant.Core.Extraction
```

Baris ini diperlukan agar tipe seperti *DocumentClassification, DocumentData*, dan *IDocumentDefinition* bisa dikenali oleh compiler Razor saat digunakan di *@code* pada page dan form. Selain itu, pastikan *_Imports.razor* sudah memiliki baris yang dibutuhkan dari Part 1 agar komponen seperti *StatusBadge.razor* bisa digunakan tanpa masalah.

```
@using AzureFoundry.DocAssistant.Components.Shared
```

Baris kedua diperlukan agar komponen *DocumentForm* dan *DocumentStatusChip* bisa dikenali dan digunakan di *ScanDocument.razor*. Tanpa baris ini, compiler akan menganggap nama komponen tersebut sebagai elemen HTML yang tidak dikenal. Pada project App, gunakan kembali *ScanDocument.razor* dari Part 1 tanpa mengubah struktur Bootstrap yang sudah ada, lalu tambahkan dropdown jenis dokumen, tab Data/Raw text, komponen DocumentForm, dan gunakan *DocumentExtractionService.ExtractAsync()* untuk menggantikan proses OCR langsung.

Buka file *ScanDocument.razor* lalu ikuti Langkah-langkah dibawah ini.

- Tambahkan DocumentExtractionService dan DocumentDefinitionRegistry seperti sintaks di bawah.

  ```
  @inject DocumentExtractionService Extraction
  @inject DocumentDefinitionRegistry Registry
  ```

  Extraction bertugas membaca teks, mengenali jenis dokumen, lalu mengisi data yang diperlukan. Sementara itu, Registry digunakan untuk mengambil daftar jenis dokumen yang tersedia, seperti struk, invoice, dan KTP, untuk ditampilkan pada dropdown.

- Menambahkan dropdown Document Classification

  Dengan begitu, pengguna bisa memilih jenis dokumen sendiri, seperti struk, invoice, atau KTP. Jika tidak ingin memilih, pengguna bisa menggunakan opsi "Guess

automatically" agar AI mengenali jenis dokumen secara otomatis. *KindOptions* dan *OnKindChosen* akan dibuat nanti di bagian *@code*.

```
<div class="d-flex align-items-center gap-2 mb-3">
    <label for="document-kind" class="form-label mb-0">Document kind</label>
    <select id="document-kind" class="form-select form-select-sm w-auto"
    @onchange="OnKindChosen" disabled="@IsProcessing">
        <option value="" selected="@(_chosenKind is null)">
        Guess automatically</option>
        @foreach (var d in KindOptions)
        {
            <option value="@d.docClassification"
            selected="@(_chosenKind == d.docClassification)">@d.Name</option>
        }
    </select>
</div>
```

- Tambahkan kondisi pada Area Preview

```
else if (_selected is { HasFile: false })
{
     <p class="text-body-secondary text-center mb-0">The original image for this
    document has been deleted. Its data and text remain.</p>
}
```

Di light-table/preview-area, sebelumnya ada tiga kondisi yang mungkin terjadi, kamera sedang aktif, gambar dokumen sedang ditampilkan, atau belum ada dokumen yang dipilih. Sekarang, ada satu kondisi tambahan, yaitu ketika gambar asli dokumen sudah dihapus melalui fitur "Delete original image" yang baru ditambahkan pada artikel ini.

- Dynamic label pada button

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

Pada artikel sebelumnya, tombol "Read this document again" hanya ditampilkan ketika status dokumen sama dengan *Unread*. Sementara itu, untuk saat ini, tombol yang sama juga ditampilkan untuk dokumen dengan status *Ready*, yaitu dokumen yang teksnya sudah berhasil dibaca, tetapi datanya belum diekstrak. Perbedaannya hanya pada teks tombol yang ditampilkan. Sebaiknya gunakan satu tombol dengan satu @onclick. Teks pada tombol akan berubah sesuai kondisi dokumen, sehingga tidak perlu membuat dua tombol yang berbeda.

- Bagian hasil dibagi menjadi dua bagian: hasil sebelumnya dan hasil terbaru.

```
@if (_selected is { docClassification: not null } result &&
result.Status.HasStructuredData() && result.Data is not null)
 {
     <section class="card mt-4" aria-label="Result">
         <div class="card-header d-flex
         justify-content-between align-items-center">
             <div class="btn-group btn-group-sm" role="tablist">
                 <button type="button" class="btn @(_tab == ResultTab.Data ?
                 "btn-primary" : "btn-outline-secondary")"
                  @onclick="() => _tab = ResultTab.Data">
                     Data
                 </button>
                 <button type="button" class="btn @(_tab == ResultTab.Text ?
                 "btn-primary" : "btn-outline-secondary")"
                 @onclick="() => _tab = ResultTab.Text">
                     Raw text
                 </button>
             </div>
             @if (_tab == ResultTab.Text)
             {
                 <button type="button" class="btn btn-sm btn-outline-secondary"
                 @onclick="CopyResultAsync">Copy text</button>
             }
         </div>
         <div class="card-body">
             @if (_tab == ResultTab.Data)
             {
                 <DocumentForm Document="result"
                 Definition="Registry.Get(result.docClassification.Value)"
                 Busy="IsProcessing" OnSave="SaveCorrectionAsync"
                 OnApprove="ApproveAsync"OnDeleteImage="DeleteOriginalImageAsync"
                 OnReExtract="ReExtractAsync" />
             }
             else
             {
                 <pre class="result-text mb-0">@result.Text</pre>
             }
         </div>
     </section>
 }
 else if (_selected is { Status: DocumentStatus.Ready })
 {
     <!-- blok "Reading result"seperti pada artikel sebelumny -->

 }
```

Pada artikel sebelumnya, hanya ada satu blok "Reading result" yang ditampilkan ketika status Ready. Untuk saat ini ditambahkan satu blok baru di atas blok tersebut. Blok baru ini akan muncul ketika data sudah berhasil diekstrak, yaitu dengan status *Structured*, *NeedsReview*, atau *Verified*.

Logika tampilannya menjadi seperti berikut:

1. Jika data sudah diekstrak (Structured / NeedsReview / Verified):

   - Tampilkan tab Data dan Raw text.

- Tampilkan form DocumentForm (komponen baru). Akan dijelaskan tersendiri dibawah..

2. Jika data belum diekstrak, tetapi teks sudah tersedia (Ready):
   - Tampilkan blok Reading result yang lama.
   - Hanya tampilkan teks mentah (raw text).

Catatan: Blok Reading result yang lama tidak dihapus. Blok tersebut hanya dipindahkan ke kondisi else if agar tampil sesuai status dokumen.

- History: menampilkan jenis dokumen dan membuat badge menjadi komponen

```
<!-- tambahan: nama jenis dokumen -->
@if (item.docClassification is { } kindItem)
{
    <small class="d-block text-body-
    secondary">@Registry.Get(kindItem).Name</small>
}

<!-- ganti badge manual jadi komponen -->
<!-- Artikel 1: -->
@if (item.Status == DocumentStatus.Unread)
{
    <span class="badge rounded-pill text-bg-warning mt-1">Not read yet</span>
}

<!-- Artikel Part II: -->
<DocumentStatusChip Status="item.Status" />
```

Pada artikel sebelumnya hanya memiliki 2 status (Unread dan Ready), sehingga badge bisa ditulis langsung (inline) tanpa membuat kode menjadi rumit. Sementara itu, untuk part 2 memiliki 5 status (Unread, Ready, Structured, NeedsReview, dan Verified). Jika semuanya ditulis secara inline, kode akan menjadi terlalu panjang dan sulit dibaca. Karena itu, dibuat komponen terpisah bernama *DocumentStatusChip* untuk menangani kelima status tersebut agar kode lebih rapi, sederhana, dan mudah dikelola.

- @code: field dan konstanta baru

```
// diganti: dari 90 detik jadi 180 detik, karena sekarang ada 3 panggilan AI
// (baca teks + tebak jenis + isi data), bukan cuma 1
private static readonly TimeSpan _processingTimeout = TimeSpan.FromSeconds(180);

// baru: buat nentuin tab mana yang lagi aktif
private enum ResultTab
{
    Data,
    Text,
}

// baru: jenis dokumen yang dipilih user di dropdown (null = auto)
private DocumentClassification? _chosenKind;
private ResultTab _tab = ResultTab.Data;
```

_ocrTimeout dari part 1 dihapus, diganti *_processingTimeout*.

```
private IEnumerable<IDocumentDefinition> KindOptions =>
Registry.All.OrderBy(d => d.docClassification == DocumentClassification.Other ?
int.MaxValue : (int)d.docClassification);

private void OnKindChosen(ChangeEventArgs e) =>
    _chosenKind = Enum.TryParse<DocumentClassification>(e.Value?.ToString(), out
    var kind) ? kind : null;
```

KindOptions mengatur urutan pilihan jenis dokumen di dropdown. Pilihan *Other* sengaja diletakkan paling bawah karena hanya digunakan sebagai pilihan cadangan, bukan pilihan utama. *OnKindChosen* mengubah pilihan user dari dropdown (teks) menjadi enum yang sesuai. Jika user memilih *Guess automatically*, nilainya akan menjadi null.

```
// Artikel part 1
await RunOcrAsync(document);
// ...
await RunOcrAsync(_selected);

// Artikel part 2
await RunProcessingAsync(document.Id, _chosenKind, force: false);
// ...
await RunProcessingAsync(_selected.Id, _chosenKind, force: false);
```

ReadAgainAsync menjadi lebih singkat pada part ini, karena beberapa baris kode, yaitu *_errorMessage = null;*, *_status = ProcessStatus.Processing;*, dan *StateHasChanged();*, sudah dipindahkan ke dalam *RunProcessingAsync*. Dengan begitu, kode yang sama tidak perlu ditulis berulang kali di dua tempat.

- Mengganti RunOcrAsync dengan 3 method baru

  Ini adalah bagian utama dari perubahan di artikel part 2.

  Pada part 1, *RunOcrAsync(Document document)* hanya menjalankan *Scan.ReadAsync()*, lalu memperbarui riwayat secara manual. Di artikel part 2, fungsi tersebut diganti

dengan *RunProcessingAsync* dan dibantu oleh dua helper baru, yaitu *FailAsync* dan *Update*. Tujuannya agar proses lebih terstruktur, kode lebih rapi, dan pengelolaan status serta error lebih mudah dilakukan.

```
private async Task RunProcessingAsync(string id, DocumentClassification? kind,
bool force)
{
     _status = ProcessStatus.Processing;
     _errorMessage = null;
     StateHasChanged();

     try
     {
         using var cts = new CancellationTokenSource(_processingTimeout);
         var result = await Extraction.ExtractAsync(id, kind, force, cts.Token);

         Update(result);
         _tab = ResultTab.Data;
         _status = ProcessStatus.Done;
     }
     catch (OcrException ex) { await FailAsync(id, ex.Message); }
     catch (ExtractionException ex) { await FailAsync(id, ex.Message); }
     catch (StorageException ex) { _status = ProcessStatus.Failed;
      _errorMessage = ex.Message; }
     catch (OperationCanceledException) { await FailAsync(id, "Processing took too
     long and was cancelled."); }
}
private async Task FailAsync(string id, string message)
{
    //sintaks detail pada project lampiran

}
private void Update(Document document)
{
    var index = _history.FindIndex(d => d.Id == document.Id);
    if (index >= 0)
    {
        _history[index] = document;
    }
    _selected = document;
}
```

Perbedaan dengan *RunOcrAsync* yang lama

- Menggunakan *Extraction.ExtractAsync()* sebagai pengganti *Scan.ReadAsync()*. Dengan satu pemanggilan ini, sistem bisa langsung menjalankan beberapa proses sekaligus, membaca teks (jika belum ada), menentukan jenis dokumen, dan mengekstrak datanya.
- Menangani *ExtractionException*. Ini adalah exception baru yang digunakan khusus untuk menangani error pada tahap penentuan jenis dokumen dan ekstraksi data.

- Menambahkan *FailAsync()*. Helper ini mengambil ulang data dokumen dari database, lalu menentukan pesan error yang sesuai. Tujuannya agar user tahu tombol mana yang harus ditekan untuk mencoba kembali:
  - ✓ *Read again* jika proses membaca teks yang gagal.
  - ✓ *Extract data* jika teks sudah berhasil disimpan, tetapi proses ekstraksi data gagal.
- Menambahkan *Update()*. Helper ini menjadi satu-satunya tempat untuk memperbarui *_history* dan *_selected* sekaligus. Dengan begitu, logika untuk mencari index di list, mengganti data, dan memperbarui selected tidak perlu ditulis berulang kali di beberapa bagian kode.

- "Review" — koreksi, setuju, hapus gambar, ekstrak ulang

```
private Task SaveCorrectionAsync(DocumentData data) =>
    RunReviewAsync(() => Extraction.SaveCorrectionAsync(_selected!.Id, data));

private Task ApproveAsync() =>
    RunReviewAsync(() => Extraction.ApproveAsync(_selected!.Id));

private async Task DeleteOriginalImageAsync()
{
    await RunReviewAsync(() =>
    Extraction.DeleteOriginalImageAsync(_selected!.Id));
    if (_selected is { HasFile: false })
    {
        _previewSrc = null;
    }
}

private async Task ReExtractAsync(DocumentClassification docClassification)
{
    if (_selected is not null)
    {
        await RunProcessingAsync(_selected.Id, docClassification, force: true);
    }
}
```

```
private async Task RunReviewAsync(Func<Task<Document>> work)
{
    if (_selected is null)
    {
        return;
    }

    _errorMessage = null;
    try
    {
        Update(await work());
    }
    catch (ExtractionException ex)
    {
        _errorMessage = ex.Message;
    }
    catch (StorageException ex)
    {
        _errorMessage = ex.Message;
    }
}
```

Keempat method ini digunakan sebagai parameter *OnSave*, *OnApprove*, *OnDeleteImage*, dan *OnReExtract* pada komponen *<DocumentForm>* yang sudah ditambahkan sebelumnya. Pola kerja *RunReviewAsync* hampir sama dengan *RunProcessingAsync*, tetapi lebih sederhana karena tidak menggunakan timeout atau *cancellation token*. Method ini hanya menggunakan *try-catch* untuk menangani error dari keempat aksi tersebut. Keempatnya memiliki pola yang sama, yaitu menjalankan proses dan mengembalikan hasil berupa *Task<Document>*.

- Buat sebuah komponen razor pada Shared folder dengan nama “DocumentForm.razor”.

  File DocumentForm.razor di folder Shared juga menggunakan komponen Bootstrap seperti form-control, table, alert, dan badge, bukan CSS custom. Untuk diketahui pada artikel ini hanya menjelaskan bagian-bagian penting pada komponen. Untuk detail sintaksnya dapat dilihat pada project lampiran.

  Dependency dan parameter

```
@using System.Globalization
@inject DocumentExtractionService Service
@inject DocumentDefinitionRegistry Registry
@inject IJSRuntime JS
```

```
[Parameter, EditorRequired]
public Document Document { get; set; } = default!;

[Parameter, EditorRequired]
public IDocumentDefinition Definition { get; set; } = default!;

[Parameter]
public bool Busy { get; set; }

[Parameter]
public EventCallback<DocumentData> OnSave { get; set; }

[Parameter]
public EventCallback OnApprove { get; set; }

[Parameter]
public EventCallback OnDeleteImage { get; set; }

[Parameter]
public EventCallback<DocumentClassification> OnReExtract { get; set; }
```

Komponen ini menggunakan tiga dependency:

- DocumentExtractionService (Service) untuk menjalankan validasi ulang setiap kali user mengubah isi form.
- DocumentDefinitionRegistry (Registry) untuk mengambil daftar jenis dokumen yang ditampilkan pada dropdown re-extract.
- IJSRuntime (JS) untuk menampilkan confirm() bawaan browser sebelum menjalankan aksi yang tidak bisa dibatalkan.

Ada dua parameter wajib (EditorRequired) yang menjadi sumber data utama komponen ini:

- Document Document: dokumen yang sedang dibuka.
- IDocumentDefinition Definition: skema field yang digunakan sesuai jenis dokumen, misalnya struk, invoice, KTP, dan lainnya.

Parameter Busy digunakan untuk menonaktifkan semua kontrol ketika ada proses lain yang sedang berjalan. Selanjutnya, ada empat *EventCallback*, yaitu *OnSave, OnApprove, OnDeleteImage*, dan *OnReExtract*. Keempatnya menghubungkan *DocumentForm* dengan parent component. Perlu dipahami bahwa DocumentForm tidak menyimpan data langsung ke database. Komponen ini hanya mengirimkan permintaan kepada parent melalui *OnSave.InvokeAsync(_edit)*, beserta data yang ingin disimpan. Selanjutnya, parent yang akan memanggil service untuk memproses penyimpanan tersebut.

Header: judul, badge, dan status chip

```
<div class="d-flex flex-wrap align-items-center gap-2 mb-3">
    <h2 class="h5 mb-0">@Definition.Name</h2>
    <span class="badge text-bg-info-subtle
     text-info-emphasis border border-info-subtle">AI-extracted</span>
    <DocumentStatusChip Status="Document.Status" />
</div>
```

Bagian ini menampilkan nama jenis dokumen melalui Definition.Name, badge AI-extracted untuk memberi tahu user bahwa data tersebut merupakan hasil ekstraksi AI, bukan input manual, serta <DocumentStatusChip Status="Document.Status" /> untuk menunjukkan status dokumen saat ini. Perlu diingat bahwa status dokumen berbeda dengan status proses. Status dokumen menunjukkan kondisi dokumen, sedangkan status proses menunjukkan apakah suatu proses sedang berjalan atau tidak.

Field dirender dari skema, bukan di-hardcode

```
@foreach (var spec in Definition.Fields)
{
     var inputId = "f-" + spec.Key;
     var value = _edit.Fields.TryGetValue(spec.Key, out var v) ? v : null;
     var masked = spec.Sensitive && !_showSensitive;
     var notes = _findings.Where(f => f.Key == spec.Key).ToList();

     <div class="col-md-6">
      //Sintaks detail pada project lampiran
     </div>
}
```

Inilah yang membuat satu komponen ini bisa digunakan untuk berbagai jenis dokumen, seperti struk, invoice, dan KTP. Field yang ditampilkan otomatis mengikuti Definition.Fields, jadi tidak perlu menuliskan setiap field satu per satu di markup. Setiap kali user mengetik, nilainya langsung disimpan ke _edit.Fields melalui @bind. Sementara itu, @bind:after="Check" membuat validasi berjalan otomatis setiap kali ada perubahan input, tanpa harus menunggu user menekan tombol submit.

IsDirty — nentuin tombol mana yang aktif

```
protected override void OnParametersSet()
 {
     if (_sourceId != Document.Id)
     {
         _showSensitive = false;
     }

     if (_sourceId != Document.Id || !ReferenceEquals(_sourceData, Document.Data))
     {
         _sourceId = Document.Id;
         _sourceData = Document.Data;
         _edit = (Document.Data ?? new DocumentData()).Clone();
         _reextractKind = Document.docClassification ??
         DocumentClassification.Other;
         FillMissingKeys();
         Check();
     }
 }


private void Check()
{
     _normalized = _edit.Normalized(Definition);
     _findings = Service.Check(Definition.docClassification, _edit);
}
```

OnParametersSet() hanya memperbarui _edit ketika referensi Document.Data benar-benar berubah, misalnya saat user membuka dokumen lain atau setelah proses ekstraksi ulang. Jadi, _edit tidak akan di-reset setiap kali komponen melakukan render.

Sementara itu, Check() menjadi satu-satunya tempat untuk menjalankan validasi melalui Service, bukan hanya mengandalkan validasi di sisi client. Karena itu, Check() harus dipanggil setiap kali ada perubahan pada _edit, termasuk saat menambah baris, menghapus baris, atau mengembalikan data ke kondisi sebelumnya (revert).

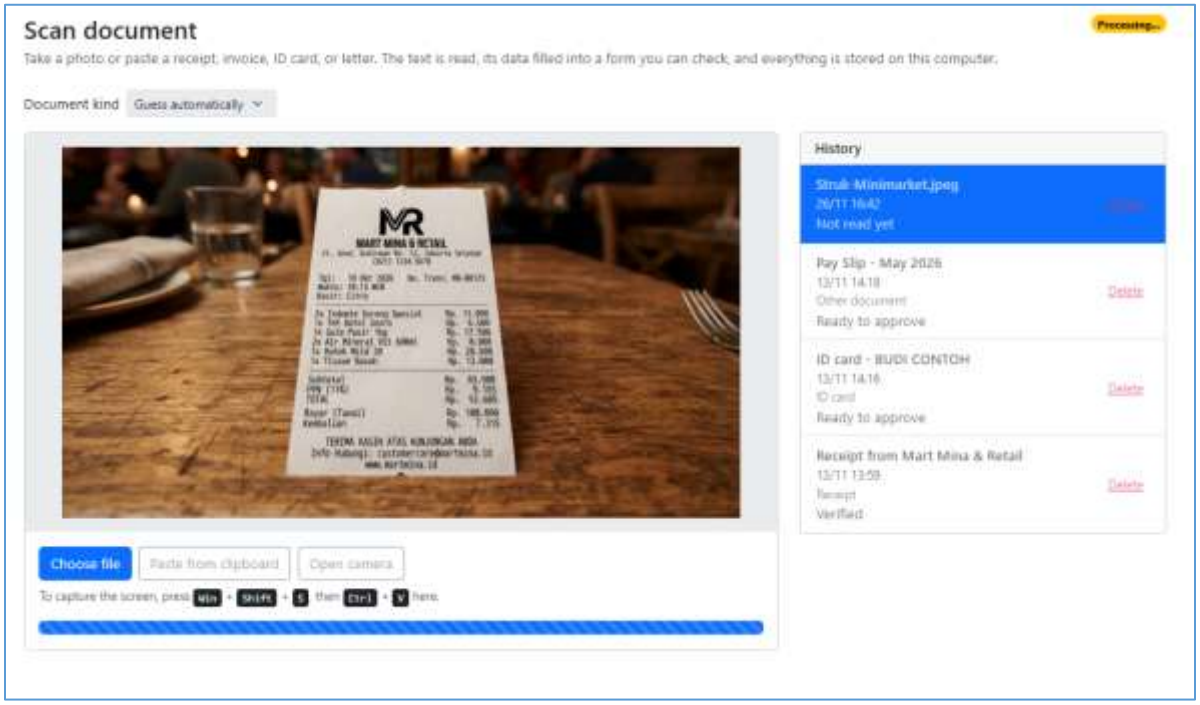
```
private async Task ApproveAsync()
{
    if (_findings.Count > 0)
    {
        var confirmed = await JS.InvokeAsync<bool>("confirm", $"There are still
        {_findings.Count} notes. Approve the data as it is?");
        if (!confirmed)
        {
            return;
        }
    }

    await OnApprove.InvokeAsync();
}
```

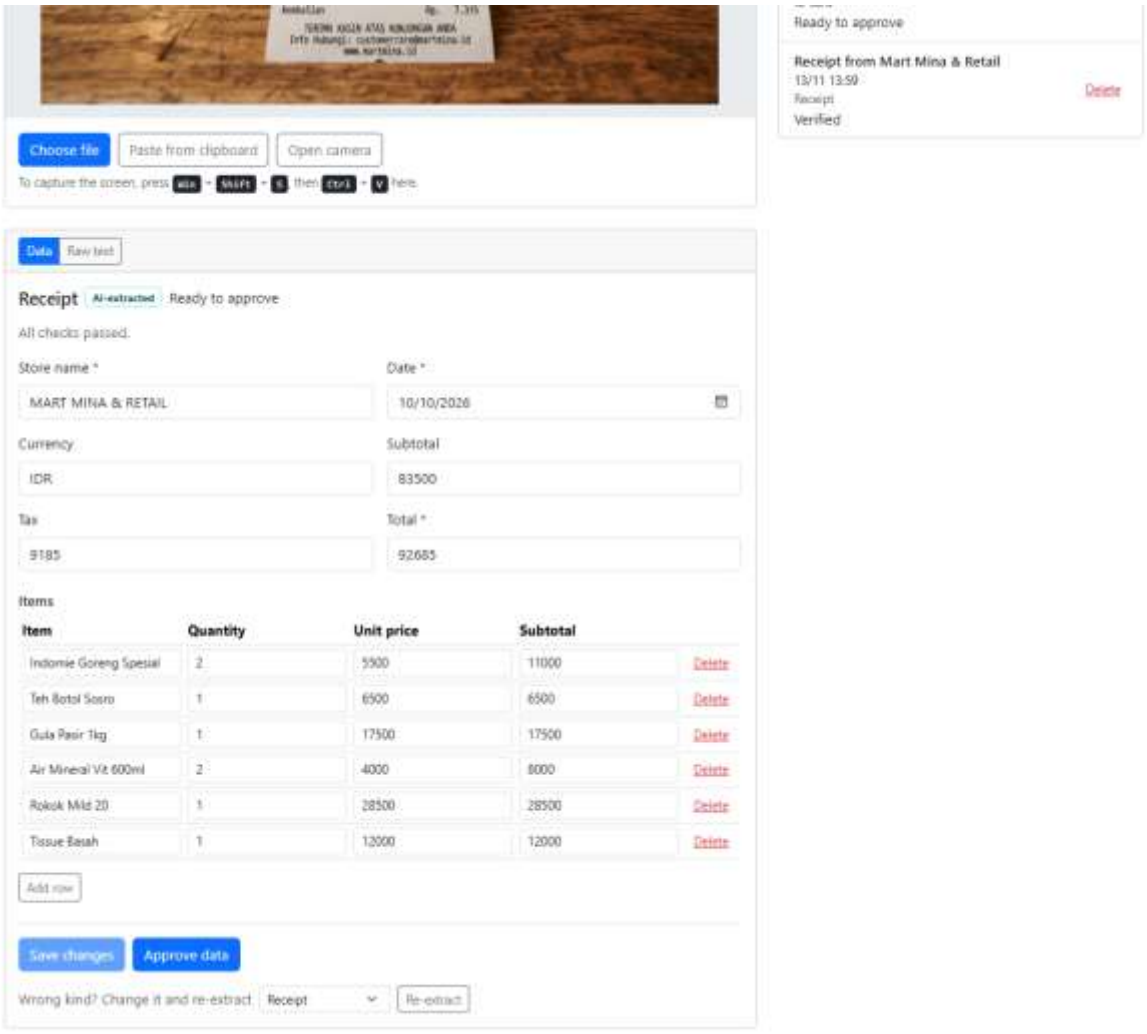
Pola yang sama juga digunakan pada *DeleteImageAsync()* dan *ReExtractAsync()*. Pada *DeleteImageAsync()*, sistem selalu menampilkan konfirmasi karena gambar yang dihapus tidak bisa dikembalikan. Sementara pada *ReExtractAsync()*, konfirmasi hanya muncul jika ada perubahan data yang berisiko tertimpa. Dengan begitu, komponen ini

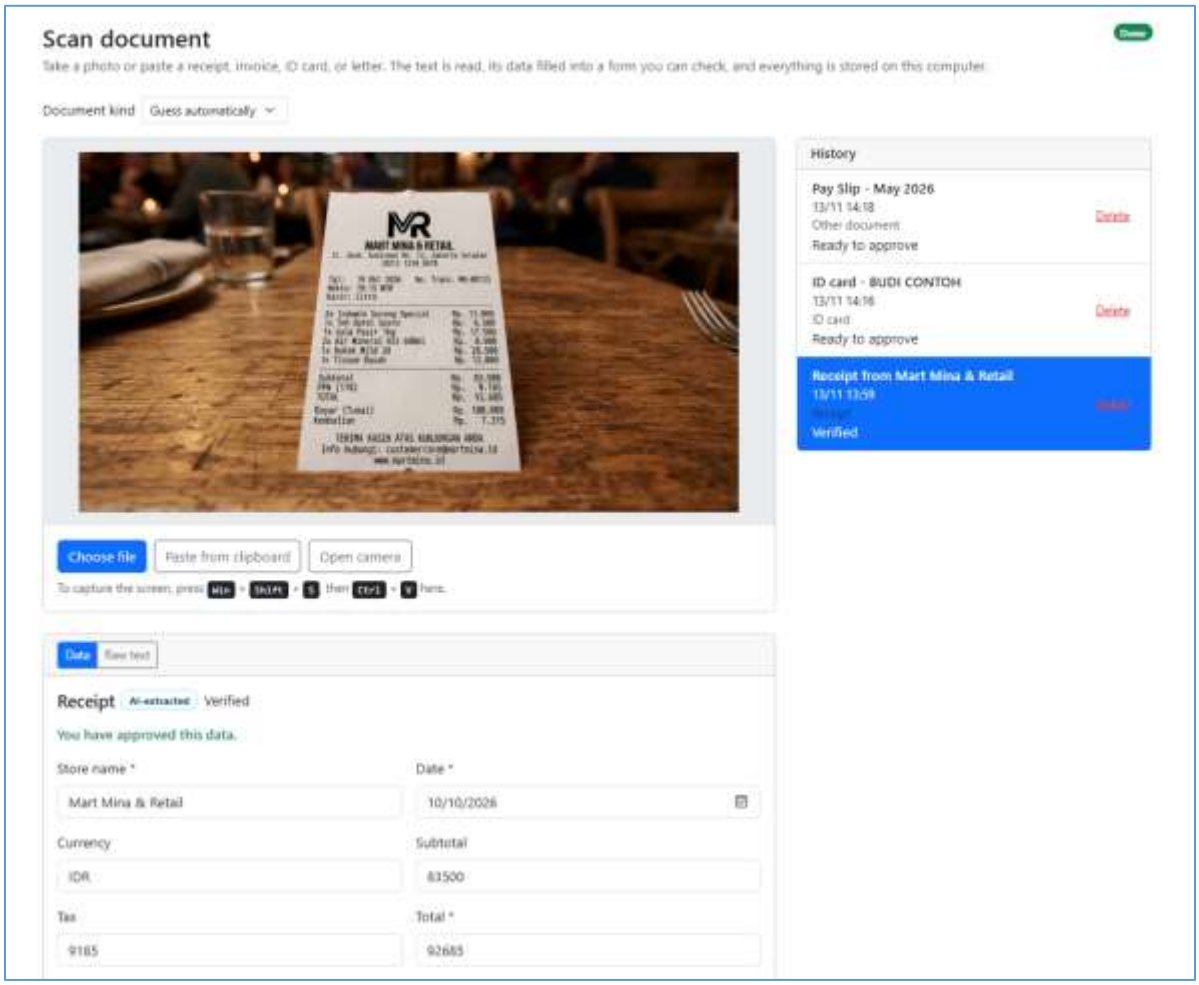
menggunakan *confirm()* bawaan browser untuk mencegah aksi yang berisiko, tanpa menampilkan konfirmasi yang tidak diperlukan.

Setelah selesai dengan Langkah-langkah di atas, lalu build dan jalankan project, lalu upload sebuah dokumen seperti struk belanja. Pastikan hasil dari OCR berhasil seperti pada gambar di bawah ini.



S

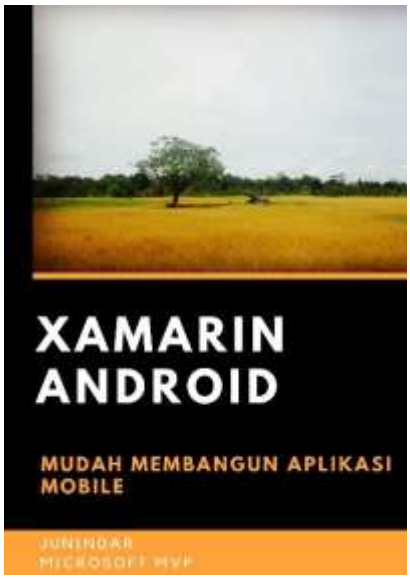

Ready to approve
Receipt from Mart Mina & Retail
13/11 13:59
Receipt
Verified
Delete
Choose file
Paste from clipboard
Open camera
To capture the screen, press Win + Shift + S, then Ctrl + V here.
Data
Raw text
Receipt
AI-extracted
Ready to approve
All checks passed.
Store name *
MART MINA & RETAIL
Date *
10/10/2026
Currency
IDR
Subtotal
83500
Tax
9185
Total *
92685
Items
Item
Quantity
Unit price
Subtotal
Indomie Goreng Spesial
2
5500
11000
Teh Botol Sosro
1
6500
6500
Gula Pasir 1kg
1
17500
17500
Air Mineral Vit 600ml
2
4000
8000
Rokok Mild 20
1
28500
28500
Tissue Basah
1
12000
12000
Delete
Add row
Save changes
Approve data
Wrong kind? Change it and re-extract
Receipt
Re-extract


Scan document
Take a photo or paste a receipt, invoice, ID card, or letter. The text is read, its data filled into a form you can check, and everything is stored on this computer.
Done
Document kind
Guess automatically
History
Pay Slip - May 2026
13/11 14:18
Other document
Delete
Ready to approve
ID card - BUDI CONTOH
13/11 14:16
ID card
Delete
Ready to approve
Receipt from Mart Mina & Retail
13/11 13:59
Delete
Verified
Choose file
Paste from clipboard
Open camera
To capture the screen, press Win + Shift + S then Ctrl + V here.
Data
Raw text
Receipt
AI-extracted
Verified
You have approved this data.
Store name *
Mart Mina & Retail
Date *
10/10/2026
Currency
IDR
Subtotal
83500
Tax
9185
Total *
92685

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/2026/11/aplikasi-desktop-ocr-berbasis-ai-dengan.html

Referensi



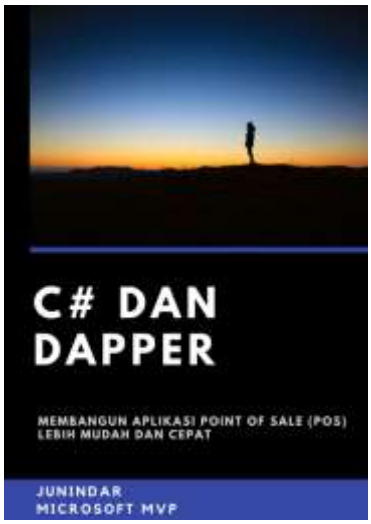
https://play.google.com/store/books/details?id=G4tFDgAAQBAJ



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=xEe5DwAAQBAJ

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".**