生産性向上ツール

ESS C# FrameWork 2021のご紹介

株式会社エスシステム

1 .ESS C# FrameWork 2021で実現するローコード開発

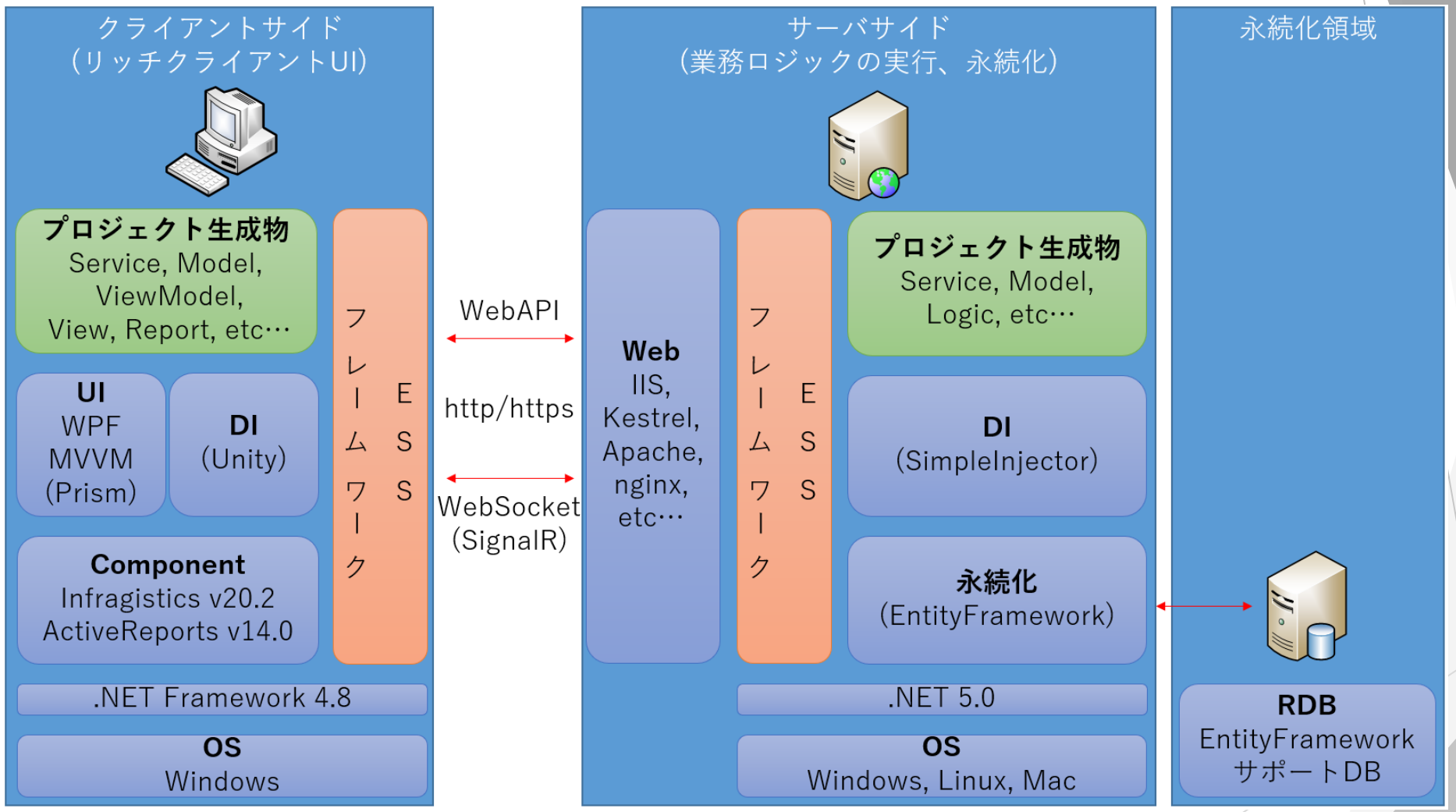
昨今、企業ををとりまくビジネス環境の変化が加速しており、よりタイムリーで、
柔軟で、迅速な対応が情報部門に求められる時代になっています。
そのような企業のニーズに応え、情報システムをビジネス成果に直結させるべく、
超高速開発手法の1つであるローコード開発が注目を集めています。

2. ローコード開発のメリット

１）開発工数の削減
FrameWorkによるローコード開発によって、実装コードが最大１／３
まで軽減することができ、実装工数やテスト工数の削減が可能です。

２）品質・保守性の向上
FrameWorkにより実装方法がルール化されるため、開発者のスキル差による
コード可読性が統一され、保守性が向上します。

3.C#オープンシステム概要図



４.ESS C# FrameWork 2021の特徴

＜旧来の開発手法と親和性の高い、データベースファーストによる開発＞

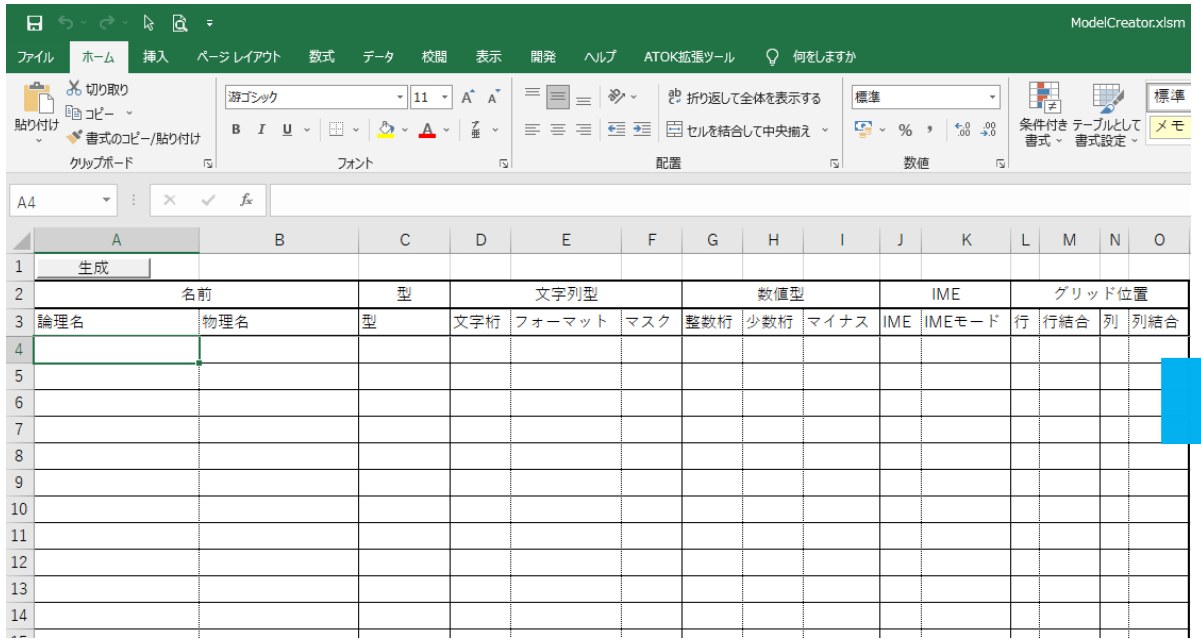
１）EntityFrameworkCoreによる、データベースモデルの自動生成(CLI)
　　また、データベーススキーマの影響箇所が、実行時エラーではなく、
　　ビルドエラーで把握できます。

２）中間モデルを利用し、データベースモデルと画面に表示するためのモデルを疎結合とします
　　※中間モデル生成用のExcelシートがあります。

３）中間モデルにアノテーションを設定することにより、コードを記述することなく
　　画面上での挙動を定義します。

4 .ESS C# FrameWork 2021の特徴

<中間モデル生成用のExcelシートのイメージ>




```
/// <summary>
/// 構成品構造のデータモデル
/// </summary>
[AddINotifyPropertyChangedInterface]
46 個の参照|Kazuyuki Yamauchi、1 日前|1 人の作成者、2 件の変更|1 件の作業項目
public class BomModel : BaseLoadModel {
    [GridField("構成品", FieldType = EGridFieldType.BIT, IsEditable = false)]
    3 個の参照|Kazuyuki Yamauchi、1 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public bool IsBom { get; set; }

    [GridField("表示順", FieldType = EGridFieldType.INTEGER, IsImeEnabled = false)]
    11 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public int? DispOrder { get; set; }

    [GridField("図面番号", FieldType = EGridFieldType.STRING, IsEditable = false)]
    20 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public string PartsCd { get; set; }

    [GridField("部材/構成品名", FieldType = EGridFieldType.STRING, IsEditable = false)]
    20 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public string PartsBomName { get; set; }

    [GridField("塗装色")]
    1 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public int? Color { get; set; }

    [GridField("員数", FieldType = EGridFieldType.FLOAT, IsImeEnabled = false, IntLength = 8, FloatLength = 2)]
    1 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public decimal ToCount { get; set; }

    [GridField("処理", FieldType = EGridFieldType.STRING, IsImeEnabled = true, ImeConversionMode = BaseConsts.IME_FULL_HIRAGANA, MaxLength = 50)]
    1 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public string Processing { get; set; }

    [GridField("備考", FieldType = EGridFieldType.STRING, IsImeEnabled = true, ImeConversionMode = BaseConsts.IME_FULL_HIRAGANA, MaxLength = 50)]
    1 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public string Remarks { get; set; }

    [GridField("")]
    9 個の参照|Kazuyuki Yamauchi、13 日前|1 人の作成者、1 件の変更|1 件の作業項目
    public ObservableCollection<BomModel> ChildItems { get; set; } = new ObservableCollection<BomModel>();
}
```

プログラムの知識がなくても中間モデルを作成することができ、
仕様書として機能させることもできます。

４.ESS C# FrameWork 2021の特徴

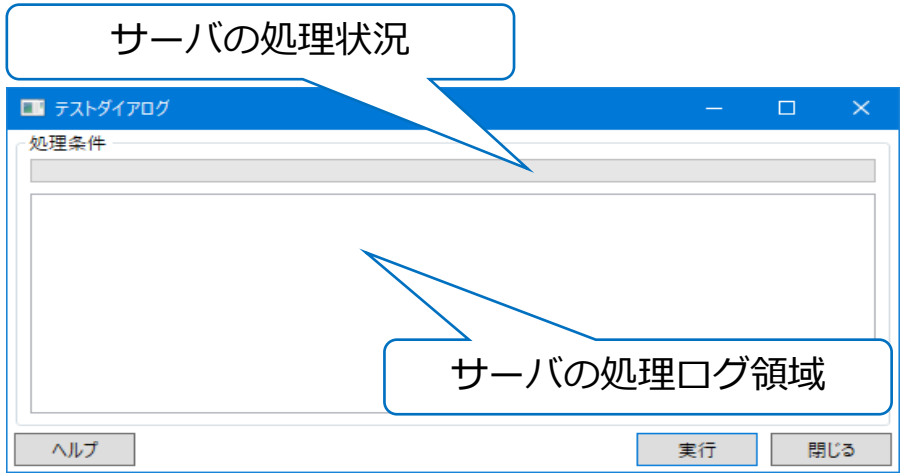
<フレームワーク、画面用テンプレートを駆使したローコード開発>

１）Swagを利用した、WebAPIクライアントコードを自動生成(CLI)します。

２）WebAPIへの意識を極力省き、クラ・サバライクでの開発が可能です。

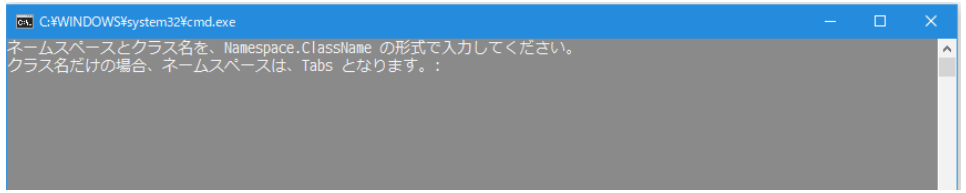
３）SignalR Coreを利用した、サーバプッシュが可能です。
Webアプリでは難しい、サーバ側処理の進捗表示などをフレームワークにて容易に実現します。

４）単テーブルのマスタメンテ画面であれば、30分程度で作成可能です。
※テーブルがデータベース上に作成されていることが前提となります。



5.画面作成例

1）CLIを利用して、データベースモデルを生成します。(初回のみ)
2）CLIを利用して、画面テンプレートからソースファイルを生成します。



3）生成されたソースで、中間モデルをExcelシートを利用して作成します。
4）WebAPI側の検索処理から中間モデルへ転記する処理を記述します。（赤枠内）

```
/// <summary>
/// テストの検索処理
/// </summary>
/// <param name="loginInfo">ログイン情報</param>
/// <param name="param">検索条件</param>
/// <param name="cancellationToken">キャンセルトークン</param>
/// <returns>検索結果</returns>
2 個の参照 | 変更箇所なし | 作成者なし、変更箇所なし
public async Task<IList<TestLoadModel>> LoadAsync(LoginInfo loginInfo, TestSearchParam param, CancellationToken cancellationToken) {
    IList<TestLoadModel> result = new List<TestLoadModel>();
    using var db = new PmsContext();
    db.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var targets = db.MTest.AsQueryable();

    // 共通列をクエリする
    // targets = targets.QueryCommonColumn(param);

    // グリッド表示データの作成
    foreach (var item in await targets.ToArrayAsync().ConfigureAwait(false)) {
        result.Add(new TestLoadModel {
            DispOrder = item.DispOrder,
            CreateUser = item.CreateUser,
            CreateTime = item.CreateTime,
            ModifyUser = item.ModifyUser,
            ModifyTime = item.ModifyTime,
            Key = Guid.NewGuid(),
        });
    }

    return result;
}
```

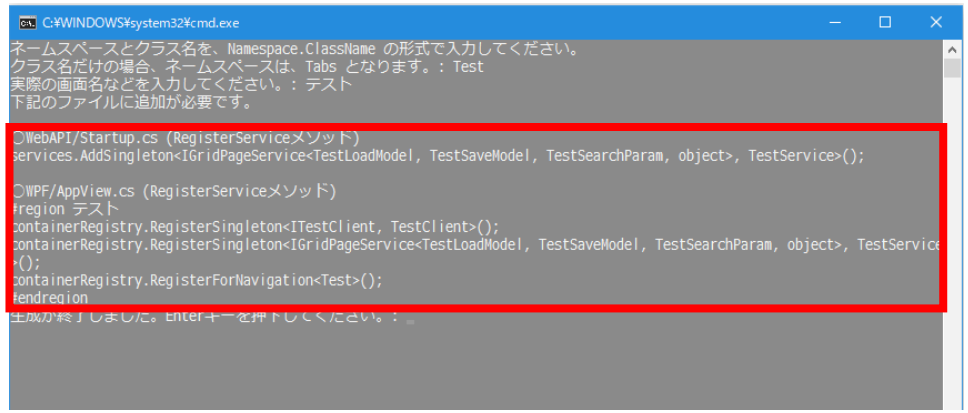
5.画面作成例

5）WebAPI側の保存処理を記述します。（赤枠内）

```
/// <summary>
/// テストの保存処理
/// </summary>
/// <param name="loginInfo">ログイン情報</param>
/// <param name="param">処理するデータ</param>
/// <param name="cancellationToken">キャンセルトークン</param>
/// <returns>処理結果</returns>
2 個の参照 | 変更箇所なし | 作成者なし、変更箇所なし
public async Task<BaseSaveResultModel<TestLoadModel>> SaveAsync(LoginInfo loginInfo, TestSaveModel param, CancellationToken cancellationToken) {
    var result = new TestSaveResultModel();

    try {
        using var db = new PmsContext();
        var now = DateTime.Now;

        foreach (var item in param.DataSource) {
            MTest target = null;

            #region 追加処理
            if (item.eDataStatus == EDataStatus.INSERT) {
                target = new MTest {
                    DispOrder = item.DispOrder,
                    CreateUser = loginInfo.UserId,
                    CreateTime = now,
                    ModifyUser = loginInfo.UserId,
                    ModifyTime = now,
                    DelFlg = item.DelFlg,
                };

                db.MTest.Add(target);
                continue;
            }
            #endregion

            // 更新・削除対象を検索する
            target = await db.MTest
                .FirstOrDefaultAsync(q => q.Test == item.Test)
                .ConfigureAwait(false);
            if (target == null) continue;   // あり得ないはず

            #region 削除処理
            if (item.eDataStatus == EDataStatus.DELETE) {
                db.MTest.Remove(target);
                continue;
            }
            #endregion

            #region 更新処理

            target.DispOrder = item.DispOrder;
            target.ModifyUser = loginInfo.UserId;
            target.ModifyTime = now;
            target.DelFlg = item.DelFlg;
            #endregion
        }

        await db.SaveChangesAsync(FuncId, loginInfo).ConfigureAwait(false);

        result.Saved = param.DataSource;
    } catch (Exception e) {
        result.ErrorMessage = e.GetFirstMessage();
    }

    return result;
}
```

6）WPF側へ追加されたソースファイルを
プロジェクトへ追加します。

7）画面テンプレートの作成CLIで出力されたコード
を指示されたソースに追加します。（赤枠内）



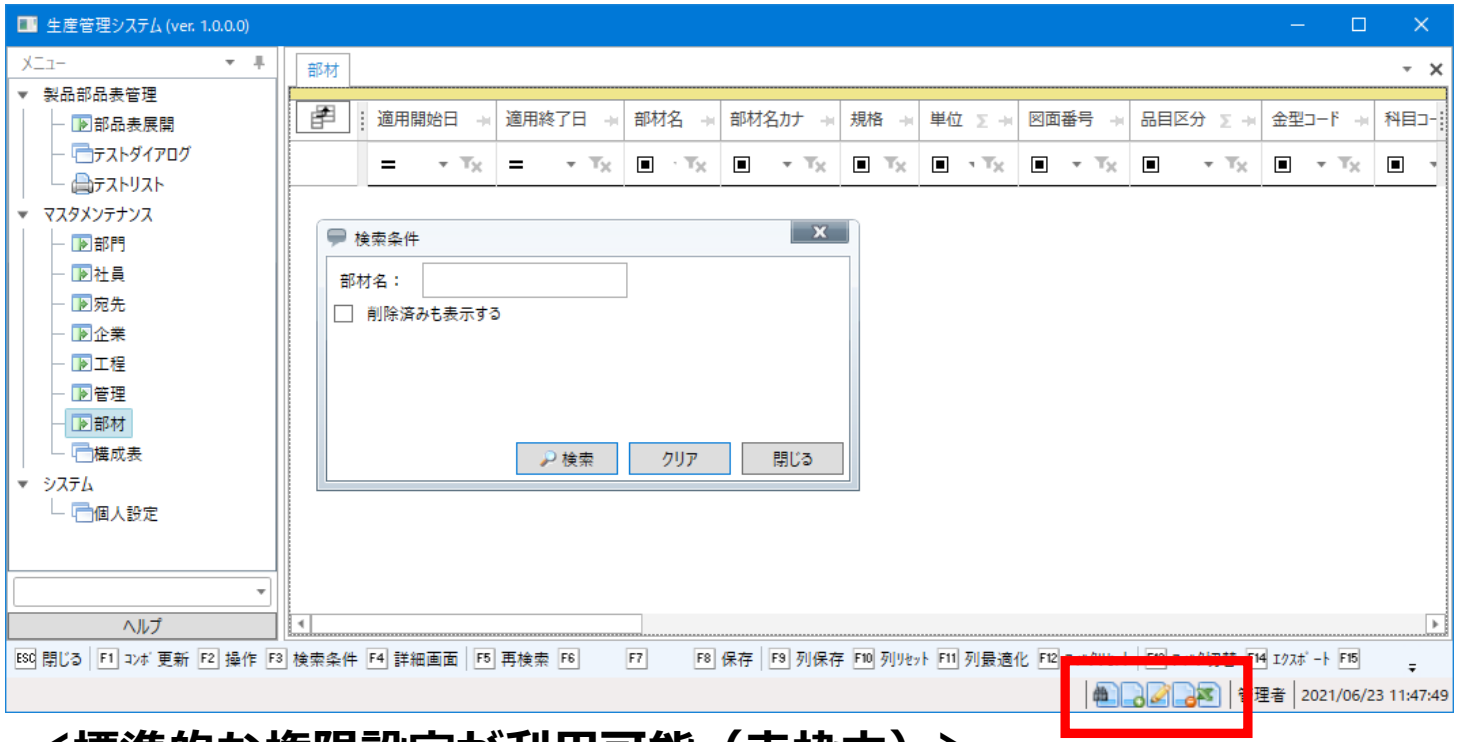
8）メニューのマスタに作成した画面を追加します。

6.画面の特徴

<グリッド機能を利用した、Excelライクな画面デザイン>

1）フィルタ機能、並び替え機能、列の入替などが標準機能として利用できます。

2）ログインユーザ毎に、画面の状態（フィルタ、並び替え、列位置情報など）が保存可能です。



<標準的な権限設定が利用可能（赤枠内）>

1）画面毎に、検索・追加・更新・削除・エクスポートをコード追記不要で適切な動作が可能です。

※コードを追加することで、動作をカスタマイズすることも可能です。

７.動作環境

Microsoft Visual Studio 2019 + インターネット接続 (Nugetに必要)
EntityFrameworkCore に対応したデータベース (SQL-Serverを推奨)
Infragistics Professional 2020 vol.2以降 (画面、Excel帳票で必須)
GrapeCity ActiveReports for .NET 12.0以降 (Excel帳票のみの場合は不要)

８． 製品価格

ソースコード：500,000円

※ 開発者あたりの年間サブスクリプション価格は検討中

９.バージョンアップ方法

サポートサイトより、最新版をダウンロード

１０.納品方法

当社サイトからモジュールとマニュアルをダウンロードしていただきます。

１１．オンサイト研修制度

お客様のご要望によりオンサイトでの研修を行います。

受講可能人数：最大１０人／回
時間：２時間程度／回 （プロジェクトの作成～1画面の作成まで）
費用：50,000円／回

※前提条件として以下の基本的スキルを有していることとします。

C#, EntityFramework, DI, WPF + MVVM

１２．その他