以下では、提示いただいたコード（CustomerManager_C.DocumentAsk）を前提に、外部依存（データベース/外部 API/HttpContext）に強く依存しない純粋なロジック部分を中心に単体テストを用意します。対象コードに private メソッドが多いため、リフレクションで呼び出す形にしています。テストフレームワークは MSTest V2 を想定しています。

前提

- WebForms（System.Web）プロジェクト CustomerManager_C を参照するテストプロジェクトを用意してください。

- テストは .NET Framework（WebForms と同一ターゲット）で動かしてください。

- 今回は外部依存の強い以下は対象外（または別途リファクタの上でモック推奨）

 - HTTP 通信（GetEmbedding, PostWithRetryAsync, btnAsk_Click 内の Chat API 呼び出し など）

 - DB アクセス（FindSimilarChunks, BuildRagContextAsync 内の DB 読み出しなど）

 - ページイベントとレスポンス書き込み（btnExportWord_Click, btnExportPDF_Click など）

テスト対象（リフレクションで呼び出し）

- CosineSimilarity(float[] v1, float[] v2)

- ExtractTitleLike(string text)

- CreateJsonContent(object body)

ポイント

- CosineSimilarity の基本ケース、直交ベクトル、長さ不一致ケースを確認

- ExtractTitleLike の拡張子含み抽出の基本と非該当ケースを確認

- CreateJsonContent が application/json ヘッダとなること、camelCase かつ null プロパティが省略されることを確認

サンプルテストコード（MSTest）

以下をテストプロジェクトに追加してください。（ファイル名例: DocumentAskTests.cs）

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using System.Linq;
using System.Net.Http;
using System.Reflection;
using System.Text;
using CustomerManager_C;

namespace CustomerManager_C.Tests
{
    [TestClass]
    public class DocumentAskTests
    {
        private object _page; // DocumentAsk のインスタンス
```

```
    [TestInitialize]
    public void Setup()
    {
        // WebForms の Page を継承していますが、今回のテストは HttpContext 非依存の private メソッドのみ呼び出します
        _page = Activator.CreateInstance(typeof(DocumentAsk));
    }

    // 汎用：private メソッド呼び出し（オーバーロードと null 引数にも柔軟に対応）
    private static T InvokePrivate<T>(object instance, string methodName, params object[] args)
    {
        var type = instance.GetType();
        var methods = type.GetMethods(BindingFlags.Instance | BindingFlags.NonPublic)
                          .Where(m => m.Name == methodName)
                          .ToList();

        foreach (var m in methods)
        {
            var ps = m.GetParameters();
            if (ps.Length != (args?.Length ?? 0)) continue;

            bool match = true;
```

```
            for (int i = 0; i < ps.Length; i++)
            {
                if (args[i] == null)
                {
                    // null は参照型/nullable なら可とする
                    match &= !ps[i].ParameterType.IsValueType ||
Nullable.GetUnderlyingType(ps[i].ParameterType) != null;
                }
                else
                {
                    match &= ps[i].ParameterType.IsAssignableFrom(args[i].GetType());
                }
            }
            if (!match) continue;

            var ret = m.Invoke(instance, args);
            return (T)ret;
        }

        throw new MissingMethodException($"{type.FullName}.{methodName} に一致するシグネチャが見つかりません。");
    }
```

```
[TestMethod]
public void CosineSimilarity_SameVector_ReturnsOne()
{
    var v1 = new float[] { 1f, 0f };
    var v2 = new float[] { 1f, 0f };

    double actual = InvokePrivate<double>(_page, "CosineSimilarity", v1, v2);

    Assert.AreEqual(1.0, actual, 1e-12, "同一ベクトルのコサイン類似度は 1 になるべき");
}

[TestMethod]
public void CosineSimilarity_Orthogonal_ReturnsZero()
{
    var v1 = new float[] { 1f, 0f };
    var v2 = new float[] { 0f, 1f };

    double actual = InvokePrivate<double>(_page, "CosineSimilarity", v1, v2);

    Assert.AreEqual(0.0, actual, 1e-12, "直交ベクトルのコサイン類似度は 0 になるべき");
}
```

```
    [TestMethod]
    public void CosineSimilarity_DifferentLength_ComputesOnMinDimension()
    {
        // dot = 1*1 + 2*2 = 5
        // ||v1|| = sqrt(1^2+2^2+3^2) = sqrt(14)
        // ||v2|| = sqrt(1^2+2^2)     = sqrt(5)
        // cos = 5 / (sqrt(14)*sqrt(5))
        var v1 = new float[] { 1f, 2f, 3f };
        var v2 = new float[] { 1f, 2f };

        double actual = InvokePrivate<double>(_page, "CosineSimilarity", v1, v2);

        double expected = 5.0 / Math.Sqrt(14.0 * 5.0);
        Assert.AreEqual(expected, actual, 1e-12, "長さ不一致でも短い方に合わせて計算されるべき");
    }

    [TestMethod]
    public void ExtractTitleLike_FindsFileNameWithExtension()
    {
        string text = "この件は Report.cs で定義されています。他にも参照があります。";
```

```
    string actual = InvokePrivate<string>(_page, "ExtractTitleLike", text);

    Assert.AreEqual("Report.cs", actual, "最初に見つかった拡張子付き語を返すべき");
}

[TestMethod]
public void ExtractTitleLike_NoHit_ReturnsNull()
{
    string text = "これは拡張子を含まない質問文です。";

    string actual = InvokePrivate<string>(_page, "ExtractTitleLike", text);

    Assert.IsTrue(actual == null, "該当がなければ null を返すべき");
}

[TestMethod]
public void CreateJsonContent_SerializesCamelCase_And_IgnoresNull()
{
    // CreateJsonContent は _jsonOpt（CamelCase + null 無視）を使う想定
    var body = new
    {
        FooBar = "X",
```

```
            Nest = new
            {
                A = (string)null
            }
        };

        var content = InvokePrivate<HttpContent>(_page, "CreateJsonContent", (object)body);

        Assert.IsNotNull(content, "HttpContent が返るべき");
        Assert.IsNotNull(content.Headers.ContentType, "Content-Type が設定されているべき");
        Assert.AreEqual("application/json", content.Headers.ContentType.MediaType, "application/json であるべき");

        var bytes = content.ReadAsByteArrayAsync().Result;
        var json = Encoding.UTF8.GetString(bytes);

        // camelCase になっていること
        StringAssert.Contains(json, "\"fooBar\":\"X\"", "camelCase でシリアライズされるべき");
        // null プロパティは出力されないこと（"a": が含まれない）
        Assert.IsFalse(json.Contains("\"a\":"), "null のプロパティは出力されないべき");
    }
```

```
    }
}
```

導入手順（例）

- ソリューションに MSTest テストプロジェクトを追加

- テストプロジェクトから WebForms プロジェクト（CustomerManager_C）を参照

- NuGet で MSTest.TestFramework / MSTest.TestAdapter を導入

- 上記コードをテストプロジェクトに追加し、テスト実行

補足と拡張案

- PostWithRetryAsync のリトライ検証や GetEmbedding/BuildRagContextAsync/FindSimilarChunks の検証は、現状の設計（private メソッド、static HttpClient、実 DB/実 API 依存）のままだとテストが重くなります。以下のリファクタを行うと、モック可能となりユニットテストが容易になります。

 - HttpClient を外部注入（例: コンストラクタまたはプロパティ注入）し、HttpMessageHandler をモック

 - DB アクセスを抽象化（リポジトリやインターフェース）し、テストではインメモリ実装を使用

 - private を internal に変更＋InternalsVisibleTo でテストから直接呼べるようにする、または純粋ロジックを別クラスに切り出す

- ExportWord/PDF のテストは結合/統合テストでの確認を推奨（ファイル出力のバイト列検証や、最低限の PDF/Docx 生成可否確認など）

このテストコードが最初の足掛かりとして、純粋ロジックの品質を確保するのに役立てば幸いです。外部依存部のユニットテスト化についても必要であれば具体的なリファクタやモック例をご提案できます。