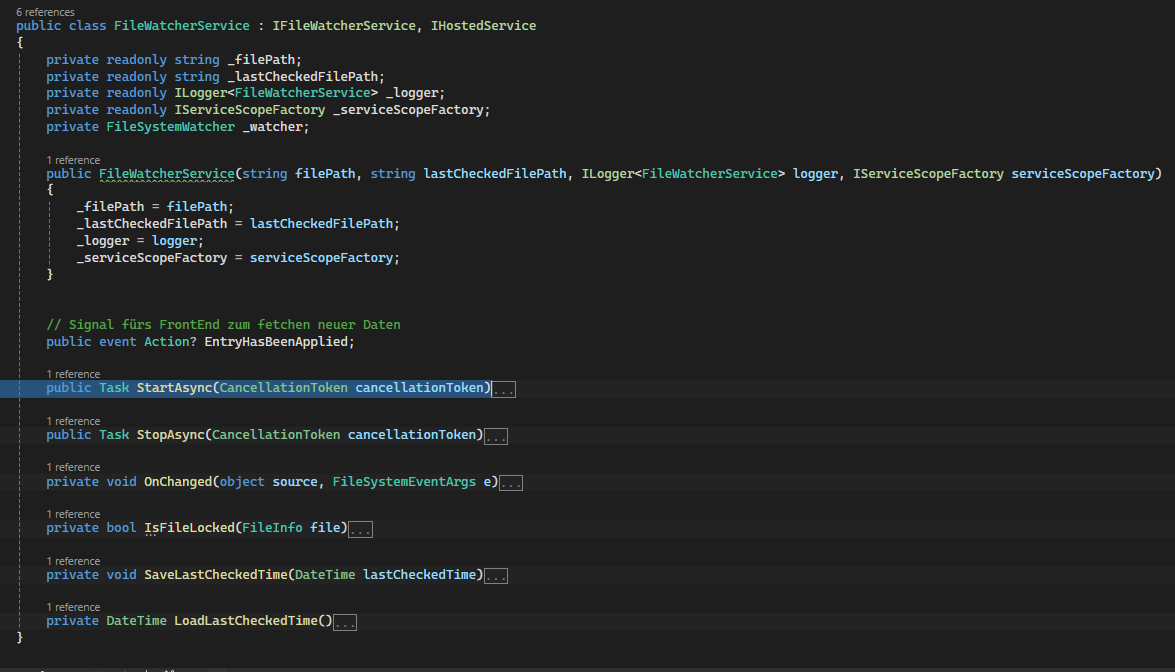
**1. Projektdefinition**

**Projektname**: Qualitätsbewertung und Visualisierung von Checkpoint-Daten **Projektziel**: Entwicklung einer Software-Plattform zur Erfassung, Integration und Visualisierung von Qualitätsdaten an einem Checkpoint (Blazor Server App)

1PT = 8H Arbeitszeit

**2. Aufgabenliste**

1. **Projektvorbereitung ( 1-2 PT )**
   * Anforderungen und Spezifikationen klären
     1. Diese wurden laut Anforderungen an mich übermittelt
   * Projektplan erstellen
2. **Infrastruktur einrichten (1-2 PT)**
   * Entwicklungsumgebung aufsetzen
     1. Entscheidung fiel auf ein klassisches „CA-Design“ mit den Layern
        1. Applikation
        2. Domain
        3. Infrastructure
        4. Presentation Layer
     2. Diese Entscheidung traf ich aufgrund meiner wenigen Erfahrung mit Softwarearchitektur bisher und um mein Skill-set hier zu festigen. Im Nachhinein betrachtet wäre es eine bessere Entscheidung gewesen, die „Vertical Slice Architektur“ einzusetzen
        1. CA ist konzeptionell nicht schwer zu verstehen aber schwer zu erstellen und zu warten. Man läuft bei einer komplexeren Aufgabenstellung oft in die Frage „Ist das ein Infrastruktur Objekt oder ein Domain Objekt? Sollte ich diesen Service in der Infrastruktur oder in der Applikation einbinden? Usw.
        2. Dies führt dazu, dass man Zeit verschwendet und ultimativ vom eigentlichen Ziel ablenkt. Viele Abstraktionen auf verschiedenen Layern verkomplizieren das Projekt unnötig speziell bei einer trivialen Aufgabenstellung wie dieser.
        3. Auch habe ich das „Repository Pattern“ aufgesetzt, welcher später für die Controller und der „API“ dienlich sein sollten
           1. Speziell mit modernem Framework wie „EntityFramework“, welcher unter der Haube schon das Repository-Pattern anwendet, ist dieser Weg denke ich nicht so optimal gewesen.
        4. Speziell bei Änderungen muss man dann wiederrum das Interface die Implementierung und den Controller ändern was zusätzlichen Arbeitsaufwand bedeutet und es erschwert die Wartung.
        5. Interfaces sollten nur dann benutzt werden, wenn es mehrere Implementationen gibt und es unumgänglich ist.
        6. Eventuell wäre der Ansatz mit der „Server App“ + einem „Services-Projekt“ effizienter gewesen in dieser hätte man dann z.B: in einer Ordnerstruktur /Features/Entries, /Features/ArticleCodes etc. unkomplizierter veranschaulichen können.
   * Research für Architektur
3. **Hintergrunddienst 1 ( 1PT )**
   * Entwicklung eines Hintergrunddienstes zur regelmäßigen Datenimportierung und -integration
     1. Hier war der Plan von der. json Datei Änderungen mittels Microsofts „FileWatcher“ Klasse zu erkennen und immer, wenn ein save/change etc. auf dem überwachten File passiert diese zu erkennen.
     2. Dieser Service ist als „HostedService“ registriert und hat ein Interface „IFileWatcherService.cs“, welches auch in der „CHK17.razor“ Datei injeziert wird um das Event zu subscriben, welches mir mitteilt, wenn es Änderung gab, um das dynamische Rendering anzustoßen und in „Echtzeit“ die Daten zu visualisieren mithilfe von MudBlazor und einem Chart.
     3. Eventuell wäre es besser dem Fremdsystem eine API-Schnittstelle zur Verfügung zu stellen, um die Einträge direkt in die DB einzutragen. In meinem Beispiel habe ich eine extra Column mit „CheckpointId“ die beim Erstellen per default auf 17 gesetzt ist erstellt.
        1. Für spätere „Skalierung“ auf mehrere Produktionslinien, um jeweils die Checkpoints und die Einträge dazu zu verlinken.
   * Testen des Hintergrunddienstes wurde bisher nur manuell gemacht und es sind keine Unit Tests geschrieben.



1. **WebAPI-Endpunkt 1 & Blazor Page 1 ( 1-3 PT) je nach Konzept**
   * Entwicklung der WebAPI zur Bereitstellung der Daten
     1. Hierzu steht der Controller „MonitorEntryController.cs“ mit den CRUD-Schnittstellen zur Verfügung
     2. Zusätzlich gibt es auch eine .razor View mit CRUD über DBContext unter „MonitorEntryPages“
     3. Visualisierung mittels MudBlazor
     4. Viel Code in der .Razor Komponente die man sicher noch besser woanders aufheben kann.
     5. Derzeit nur ein Chart mit 24H in der Vergangenheit bis DateTime.Now , könnte man sicherlich auch auf mehrere Konzepte wie Weekly/Monthly etc. anpassen ohne viel Aufwand.
     6. Mini „site.css“ File für den Chart erstellt.
   * Entwicklung der Blazor Page zur Visualisierung der Qualitätsdaten am TV
     1. Unter „CheckpointPages/CHK17.razor“
        1. Hier könnte man einen generischen dynamischeren Ansatz überlegen, um einfach mehrere Checkpoints aufsetzen zu können. z. B: über eine Konfigurationsdatei oder einem Admin-Dashboard.
   * Integration und Testen des Web-Frontends
2. **WebAPI Endpunkt 2 & Blazor Page 2 ( 0,5 – 1 PT )**
   * Entwicklung der WebAPI zur Pflege der Artikelcodes
     1. Hierzu steht der Controller „ArticleCodesController.cs“ mit den CRUD-Schnittstellen zur Verfügung.
     2. Zusätzlich gibt es auch eine .razor View mit CRUD über DBContext unter „ArticleCodePages“ zur Verfügung.
   * Entwicklung der Blazor Pages zur Verwaltung der Artikelcodes
   * Integration und Testen des Verwaltungs-Frontends
3. **Dokumentation und Schulung**
   * Projektdokumentation erstellen
     1. Wurde sehr einfach gehalten.
   * Schulung der Endbenutzer

**3. Zeitplan**

**Tag 1**: Projektvorbereitung und Infrastruktur einrichten

**Tag 2**: Entwicklung und Testen des Hintergrunddienstes

**Tag 3-4**: Entwicklung der WebAPI und Blazor Page für die Qualitätsdatenvisualisierung

**Tag 5**: Entwicklung der WebAPI und Blazor Page für die Artikelcode-Verwaltung

**Woche 2**: Integration und Systemtests

**Woche 2**: Dokumentation und Schulung

**4. Verantwortlichkeiten**

**Projektleiter**: Umut

**Entwicklerteam**: Umut

**Tester**: Umut

**5. Ressourcen**

**Entwicklungsumgebung**:

Visual Studio 22, .NET,

**Technologien und Frameworks:**

EntityFramework, Mudblazor, Blazor, .NET, C#

**Datenquelle**: Netzlaufwerk mit JSON-Dateien | Lokale Datenbank „SQL Server 2022“

**Hardware**: TV mit Chrome-Browser

**Fazit**

**Herausforderungen:**

**1.**

Wie so üblich in der Programmierung waren in den Stunden der Entwicklung öfter Fehler, die man sich nicht erklären konnte. Ein Fehler davon war, dass ich über die Referenz in meiner Presentation Layer in einer Klasse kein resolving der Namespacing bekommen habe auf meinen FileWatcher und dessen Interface. Das Ganze hat mich in etwa 4-5 Stunden gekostet und der Code-Stand konnte von anderen Systemen und zwei Personen aus der „C#-Community“ in welcher ich in einem Discord Server bin nicht reproduziert werden.

Abhilfe hat geschaffen, indem ich „Jet Brains Rider“ installiert habe auf meinem System und magischerweise hat dieses etwas auf meinem System repariert und die Auflösung der Namen ging dann doch. Ich denke mal es war ein Problem der Assembly oder meiner SDK.

**2.**

Herausfordernd und spannend war auch wieder tiefer unter die Haube von .NET zu tauchen speziell im Front-End da ich dort am wenigsten Erfahrung habe und mit den neuen Funktionen wie dem Interactive Rendering von. Razor Komponenten und anderen Interessanten Technologien, wo ich viel lernen konnte.

**Ideen und Gedanken:**

**Fremdystem und Netzlaufwerk:**

* + 1. Das Fremdsystem sollte im Idealfall über eine API-Schnittstelle liefern anstatt in eine Datei am Laufwerk
       1. Hier wäre je nach Produktionskapazität eventuell Performance ein Thema, wenn die Datei nicht regelmäßig „geleert“ wird, könnte diese zu groß werden. Auch wäre das eine manuelle Tätigkeit die eventuell vergessen werden könnte.
    2. Idealfall wäre dann natürlich auch die Checkpoint ID mitzuliefern um später beim auslesen die Verbindung zu welchem „Fernseher“ man dieses Visualisieren soll, herzustellen.
    3. Mein Verständnis für das .JSON File war, dass die „articleDescription“ gleich den „Produktnamen“ mitliefert und die Tabelle „Artikelcodes“ derzeit nicht in der Logik berücksichtigt wird. Diese Tabelle ist mit der Voraussetzung, dass Sie richtig gewartet und aktualisiert wird (vom User) im Stande über die DMC den Produktnamen selbstständig zu referenzieren. Der Artikelcode ist im Beispiel: mit „**55322234“** als Key „String“ in der DB hinterlegt und hat eine „Bezeichnung“ als zweite Column.
    4. Die DMC dient bei mir als „Primary Key“ und ist eindeutig und wir gehen davon aus, dass es keine duplizierten Einträge seitens „Fremdsystem“ geben kann.

**Visualisierung auf mehreren „Fernseher bzw. Produktionslinien“:**

* + 1. Ich denke es wäre sinnvoll, falls eine Skalierung erwünscht, ist auch das automatische Generieren von neuen End-Points wie in meinem Fall z. B: unter der Ordnerstruktur in dem Presentation-Layer „/CheckpointsPages/<Checkpoint> Hier könnte man durch die CheckPointID die auf der MonitorEntry gespeichert wird auch differenzieren auf welcher Linie das Produkt erzeugt wurde und angezeigt wird.

In weiterer folge wären sicherlich noch tausende andere Sachen, die mir einfallen würden, aber ich denke mit diesem Prototyp ist die Grundfunktionalität der Anforderung gedeckt. Es wäre definitiv noch sinnvoll ein Refactoring und Error-Handling Konzept zu haben, um alle möglichen Fehler auszuschließen, um eine robuste Software zu haben.

Falls es Fragen gibt, einfach schreiben oder gerne anrufen 😊 Danke euch für eure Aufmerksamkeit und das coole Beispiel.