Build a Monitoring Portal Using .NET MVC (C#)

Senior Developer / DevOps – Technical Challenge

By Luke Rocco

Objectives

Brief

Objective:
Develop a lightweight monitoring portal using the .NET MVC framework. This portal will display monitoring data and alerts such as the ones provided in the sample csv file shared with this task. The focus should be on clean, maintainable code and creating a user-friendly interface with visualization capabilities.

Requirements:

1. Frontend Features:
   - ✓ Dashboard Page:
     - ✓ Use widgets, accordions, or collapsible panels to group and display monitoring information by system.
     - ✓ Each widget/accordion should be according to the system:
       - ✓ Status (OK/Warning/Error).
       - ✓ Alert Details
     - ✓ Include a color-coded status indicator for quick visualization (e.g., green for OK, yellow for warning, red for error).
   - ✓ Allow users to expand an accordion to see more details of alerts for a system.
   - ✓ Ensure the UI is responsive and works well across different screen sizes. You may use libraries like Bootstrap for styling and layout
2. Alert Details:
   - ✓ Each expanded view should display:
     - ✓ Timestamp of the alert.
     - ✓ Severity (Critical/Warning/Info).
     - ✓ Message describing the issue.
   - ✓ Implement a sorting feature for alerts (e.g., by timestamp or severity).
3. Data Source:
   - ✓ Alerts should be read from the provided flat file (i.e., CSV). Further alerts/dates/systems can be added to the provided file as desired. Should it be more comfortable to set a local database from which alert details are extracted, feel free to do so and provide us with migration data to set up environment.
   - ✓ Provide a method to simulate real-time updates (e.g., re-read the file every minute, as example).
4. Code Quality:
   - ✓ Write clean, maintainable, and modular code.
   - ✓ Use appropriate design patterns and techniques where applicable such as Repository Pattern, Interfaces, and DTOs

Follow .NET coding standards.

5. Scalability and Performance:
   - ✓ Ensure the design allows for easy addition of more systems or data sources in the future.

Optional Enhancements

- Ability to define ‘alerts’:

  - Solution can allow a user to define 'alerts' which essentially do a simple task (like a basic query) according to a given schedule to produce the result set.
- Search and Filtering:
  - ✓ Add a search bar to filter by date
  - ✓ Update search bar to filter alerts by keywords or criteria.
- Export Data:
  - ✓ Add functionality to export the displayed alerts to a CSV or JSON file.
- Appropriate Error Handling:
  - ✓ Implement error handling mechanisms

Deliverables:

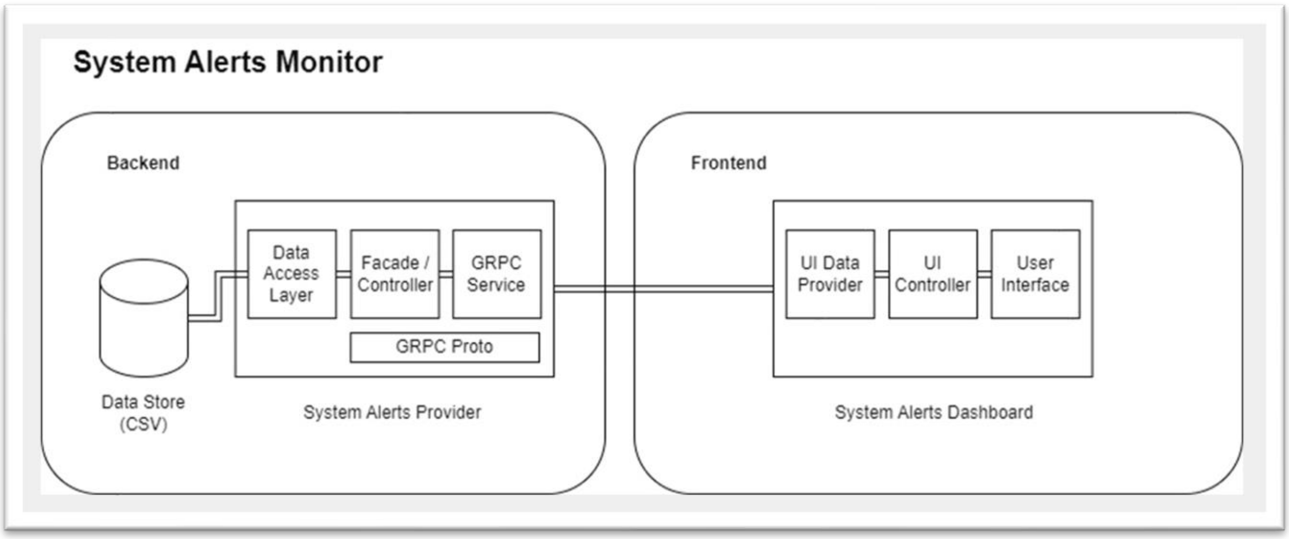
1. A functional .NET MVC application that fulfils the requirements.
2. A short README file with:
   - Steps to run the application.
   - Any assumptions made or additional features implemented.
3. A brief document explaining any architectural decisions such as the entity models implemented, and/or patterns used.

Sample Alerts:

Sample_Monitoring_Alerts.csv

Project Specifications

Project Diagram



The Minimum Viable Product (MVP) of the project will be divided into two sections.

The Front End which is named the "*System Alerts Dashboard*" and the Back End which is named the "*System Alerts Provider*".

- **Front End**, named **“System Alerts Dashboard**” is a .Net MVC web interface enabling users to review and investigate the status of monitored systems.
- **Back End**, named “**System Alerts Provider**“ is a backend system serving as the entry point for data access and interaction.

System Alerts Monitor

For the purposes of this MVP, the backend will be implementing the following features:

Project Data Store

Optional Enhancement: SQLite Integration

As per the explanation in the above brief, the data store being used is a CSV export containing a snapshot of the System Status Alerts. Upon further revisions or enhancements, however, this backend source can be retrofitted to work with either a standard Relational Database, NoSQL Database or Data Lake datastores. It is even possible to have the System Alerts Provider query directly the Live Status of the systems being monitored.

Apart from the CSV Controller, an SQLite Controller was also implemented. The provided CSV file has been loaded into an SQLite database. This approach simplifies data manipulation by allowing direct SQL queries, reducing code complexity while showcasing the system's flexibility. By implementing a custom Data Access Layer (DAL) tailored to this alternative data source, the integration seamlessly maintains the functionality of the Controllers and User Interface (UI) without requiring any modifications.

System Alerts Provider

The backend component of the MVP, called the System Alerts Provider, will serve as an entry point for front end and other systems to query the project’s data store. This access point is being envisaged as a **gRPC Service**, which although seems like an overkill for the time being, leverages the benefits of Scalability, High Performance and Streaming Support which would be useful for real-time monitoring of systems should the system be developed further.

Other alternatives include developing the System Alerts Provider as a REST API which is simpler, easier to debug, while remaining scalable, however might not be suitable for real-time monitoring.

Alternatively, the System Alerts Provider could be developed as a simple DLL Library, and possibly deployed as a NUGET Package, to be referenced and consumed by the front end and/or other systems, thus giving direct access to the datastore. This option would not be viable if the system in question does not have direct access to the datastores being consumed.

System Alerts Dashboard

The frontend component of the MVP, called the System Alerts Dashboard, will serve as the Web Interface a user can make use of to review and investigate the Status of the Systems being monitored.

For the purposes of the MVP, no Login and Authentication will be implemented on the Dashboard, so the information will be immediately accessible to anyone who has access to the URL of the Dashboard. Future enhancements to the project may include the need to Login prior to accessing the Dashboard page, and even Role Based access to the different functionalities available.

Project Review

Architecture and Frameworks used.

Onion Architecture Pattern

Both front-end and back-end utilize the Onion Architecture Pattern, ensuring a clear separation of concerns across its components. This architecture enhances maintainability, readability, and flexibility, particularly when scaling the project to interact with additional data endpoints. Should the project be developed further with the implementation of Unit and Integration testing, the Onion Architecture Pattern facilitates isolated testing of individual sections or libraries. This approach minimizes dependencies on external or third-party components, promoting a modular and testable design.

Model View Controller pattern

Complementing the Onion Architecture, the front-end employs the Model-View-Controller (MVC) pattern to manage the User Interface (UI) and associated usability logic. This pattern accelerates development once the data models are established in the Data Access Layer (which interacts with the gRPC service). By decoupling the UI and controller from the underlying data sources, MVC enhances code reusability and maintainability. Separation of concerns is further reinforced by isolating the UI controller from the data controller. This distinction ensures that UI and usability tasks are handled independently of data operations, improving testability and allowing each component to be tested in isolation.

AI Tools Used

The following is the list of AI Tools used when working on this project.

○ v0 by Vercel	https://v0.dev/chat
○ Gemini by Google	https://gemini.google.com/
○ ChatGpt by OpenAI	https://chatgpt.com

User Interface

To develop the HTML, CSS, and JavaScript required for the User Interface, **v0 by Vercel** was utilized. This tool facilitated the rapid translation of project ideas and requirements into a foundational skeleton structure, forming the basis for building the dashboards. After drafting the initial structure and reviewing the prototype, further refinements and enhancements were made by iteratively improving individual sections of the interface.

The generated code was reviewed and tailored to ensure seamless integration with the project's components and frameworks. Additionally, **Gemini AI** was consulted to aid in developing and refining

JavaScript functionalities implemented within the UI, providing valuable insights and recommendations.

Development of Basic Routines

Several essential routines, such as **Read From CSV**, **Export to CSV**, and **SQLite Logic**, were initially generated using **ChatGPT**. This approach expedited development compared to traditional methods like querying Google or Stack Overflow. The generated code was rigorously verified and adjusted to align with the specific requirements of the MVP project. Cross-referencing with **Gemini AI** further enhanced the quality and accuracy of the implementations.

Debugging of Errors and Issues encountered.

Both **ChatGPT** and **Gemini AI** were instrumental in resolving errors and debugging issues encountered during development. This streamlined the troubleshooting process, significantly reducing downtime and turnaround time. As a result, more time was allocated to implementing the project's core requirements and exploring additional functionalities outlined in the original specifications.

Review of Documentation.

Both **ChatGPT** and **Gemini AI** were utilized to review and refine the project documentation. These tools ensured that the language used was of a high standard, while maintaining clarity and professionalism. Throughout the process, careful verification was conducted to ensure that the refinements aligned with the original intent and spirit of the documentation, preserving its accuracy and consistency.

Code Quality

Design Patterns and Techniques

The **Repository Pattern** is extensively utilized to ensure a clear separation between the Controller and Data components. This decoupling allows for multiple data source factories, such as CSV and SQLite, to seamlessly utilize the same Models and data flows without requiring modifications.

By leveraging well-defined **Interfaces** and **DTOs**, the project maintains a streamlined and consistent data transfer mechanism across the various layers of the Onion Architecture. This approach enhances maintainability and flexibility in extending the system. At the moment, the Interfaces and DTOs implemented are still a bit barebone, however these can be enhanced in due course, as more functionality is added to the project.

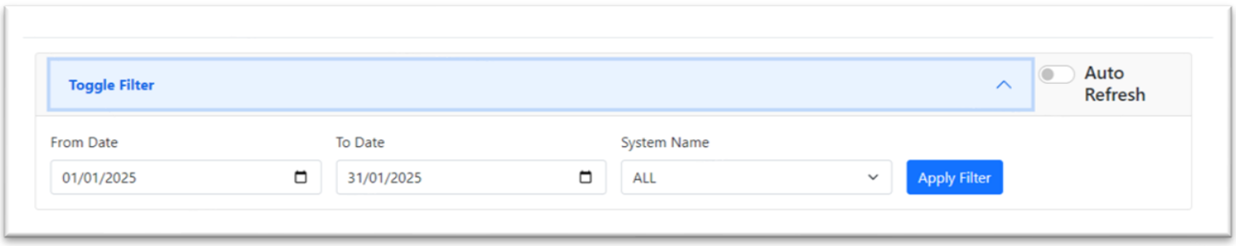
RIOK Mapperly has been employed to simplify and automate the translation between different models across the project's layers. This tool ensures efficient and accurate mapping, reducing boilerplate code and potential errors in data transformation processes

This combination of design patterns and tools contributes to a clean, maintainable, and scalable codebase, ready for future enhancements or additional data sources.

Optional Enhancements Implemented

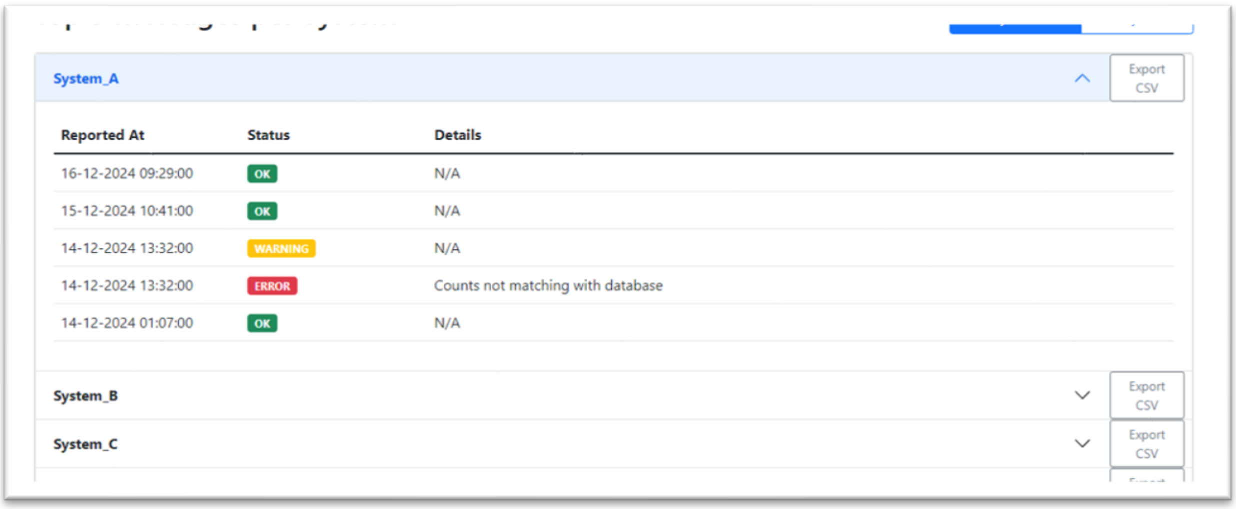
The following optional enhancements have been implemented in the project:

Search and Filtering



A Filter functionality has been implemented which allows the user to specify the FROM date, the TO date, and/or the NAME of the System. This can be further enhanced by having the 'System Name' field of the filter work as a FREE TEXT Search in both System Name and Further Details columns of the records available.

Export Data

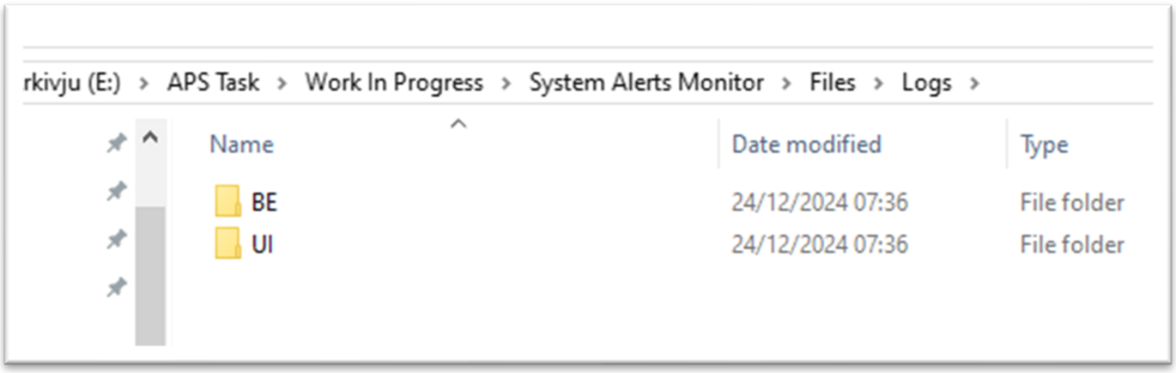


Reported At	Status	Details
16-12-2024 09:29:00	OK	N/A
15-12-2024 10:41:00	OK	N/A
14-12-2024 13:32:00	WARNING	N/A
14-12-2024 13:32:00	ERROR	Counts not matching with database
14-12-2024 01:07:00	OK	N/A

A basic EXPORT TO CSV functionality has been implemented in the UI. Basically, the system extracts the *FILTERED* data of the Current System being reviewed, and compiles them into a CSV file, which is then provided to the user to download. This functionality can be further enhanced by having the Export Format change to JSON, or Excel File (xls/xlsx). However, it was ascertained that the CSV format is a reasonably good option, satisfying the discussed requirements.

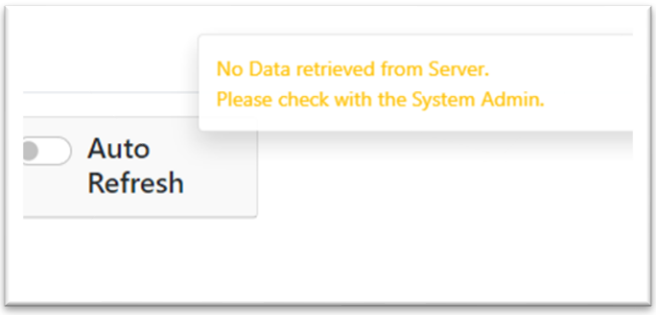
A further enhancement to the Export functionality could be implemented by moving the Compilation of the data onto the Server Side, instead of compiling it on the UI side. This would be better suited to the task, should the volume of data being exported be considerably larger, thus requiring more resources to be compiled.

Error Handling

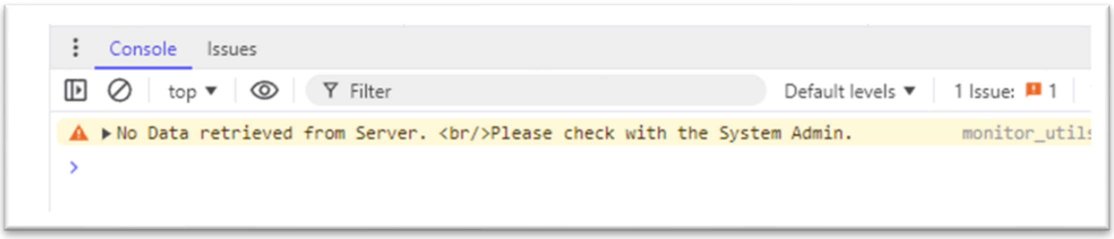

rkivju (E:) > APS Task > Work In Progress > System Alerts Monitor > Files > Logs >

Name	Date modified	Type
BE	24/12/2024 07:36	File folder
UI	24/12/2024 07:36	File folder

Basic Try/Catch logic has been implemented across the whole system, both Frontend and Backend. A basic text logger has been implemented which dumps the error message onto Text Files found under the appropriate 'Logs' folder, as seen above.


No Data retrieved from Server.
Please check with the System Admin.

On the Usability side, a 'Toast' notification system has been implemented to inform the user of any warnings or information messages which the system will provide. Apart from the Toast Notification, the messages are also logged onto the Console.Log for debugging/status review.



```
▸No Data retrieved from Server. <br/>Please check with the System Admin.          monitor_util:
```

These features can be refined, by implementing a Verbosity controller which will define what level of messages are reported to the client, and logged. Having a high usage system which logs a huge number of messages, which are not mission critical, is not ideal, especially in a distributed environment, since it will shoot up maintenance costs and storage space requirements. But this should be reviewed and revised on a case-by-case basis.