Senior Design Final Report

QTC Smart Dashboard & Business Rule Engine

Team Members:

Bryan Gonzalez

Alvent Chang

Ashley Manese

Karina Pascual Zepeda

Adrian Salgado Lopez

Jonathan Diaz

Razin Khan

James Eddins

Anthony Tsui

Pokuong Lao

Faculty Advisor:

Huiping Guo

Liaisons:

Francisco Guzman

Julian Gutierrez

Table of Contents

Contents

1. Introduction (Smart-Dashboard) :

1.1. Background:

QTC is the nation's leading provider of medical, disability, and occupational health examination services. What once started as a small medical center grew into 70 medical clients with real-time access to reporting, tracking, and case information. To further this system, QTC has teamed up with California State University, Los Angeles to develop a smart dashboard to help service desk staff communicate with users and developers easily.

Smart Dashboard is a web application that helps consolidate errors for users and admins that have occurred in different systems and applications. This application will support multiple tenants and authenticate users using Windows authentication. The functionality of the dashboard will be determined based on what the user has access to view.

1.2. Design Principles:

The goal of the smart dashboard is to be the designated interface for QTC staff to view errors that have occurred in different systems. Based on the user's access level it will determine what is displayed. The data displayed on the dashboard will be errors obtained from different lines of businesses and their integration points through a data table. The application is made to be user-friendly and easy to read. This is achieved by drop-down menus for each line of business with their integration points, pagination, filter, sort, search functionalities, and error displays. The data of errors pulled from each line of businesses' databases were converted to easy-to-read information. Also, since the smart dashboard is still currently under development, the application needs to be simple in design so that future maintenance and expansions will not be too complicated.

1.3. Design Benefits:

The architecture used by the smart dashboard is great for future use because it allows the application to add additional lines of business. The aim of the smart dashboard is to display user and/or system errors from the different tenants to the correlating user. The data of errors pulled from each line of businesses' databases were converted and displayed for users to have readable information.

1.4. Achievements:

During the 2022-2023 academic year, by using Microsoft Visual Studio 22 the team created a functional web application for our sponsor QTC. This web application uses an N-tier

architecture, modular design, and supports multiple tenants. Additionally, the data displayed on the screen is not only from a specific data source but it also pertains to a specific tenant. For the design of the user interface. It uses the bootstrap theme from QTC which not only adheres to their ui/ux standard, but it uses the latest in front-end technology such as HTML, CSS, and Javascript. Our interfaces are coded in C# specifically for .NET 6 and they enable the functionality of the site such as retrieving and displaying the specific errors on the screen. Lastly, if a username and password are not passed as query parameters, then the user will see an unauthorized message on the screen.

2. Related Technologies:

2.1. Existing Solutions:

We investigated other smart dashboard solutions that are popular on the market and that are catered towards businesses. These include Table, Yellowfin, Oracle Analytics Cloud, and Domo.

While all of these dashboards provide a feature to connect to a database to retrieve data and display it, they are not fully customizable, the user access control is very limited, and the credentials to databases would get stored under one centralized account corresponding to an email address which poses a major security risk.

The dashboards listed above require the database information to be stored within their system. If their system gets breached then the data is at risk of exposure. Having an in-house dashboard without storing any database credentials remotely is ideal for security reasons. Additionally, it will match the current look and feel of QTC's other web applications. This is not possible with the dashboards we listed.

2.2. Reused Products:

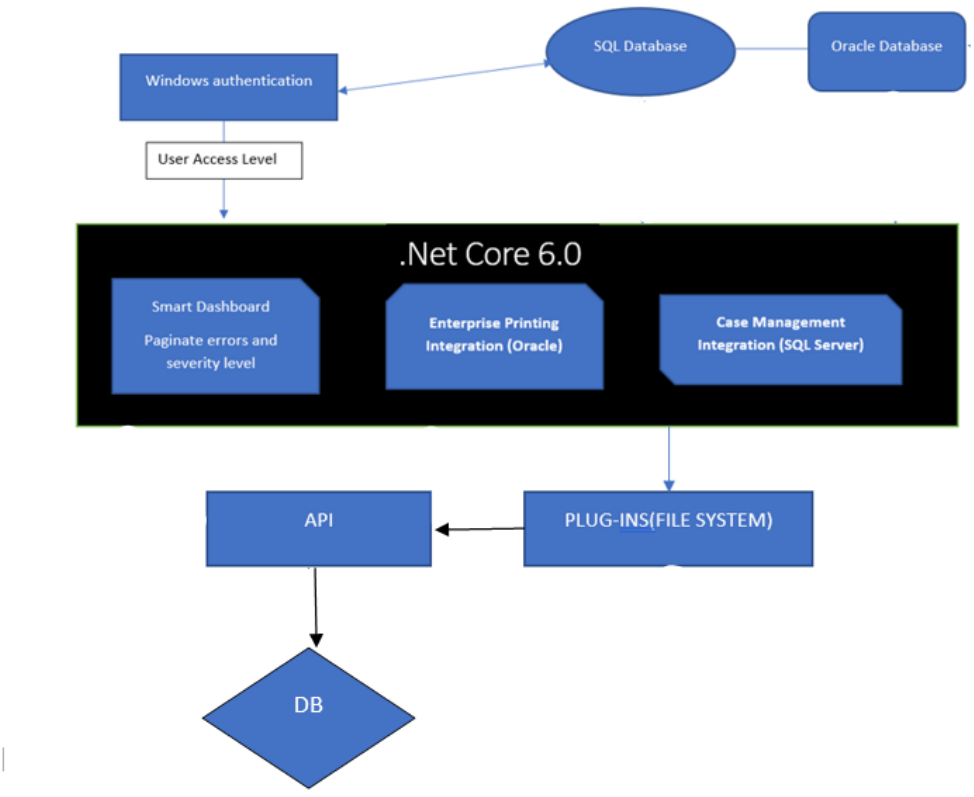
This smart dashboard will be developed using ASP.NET MVC, C#, Entity Framework using stored procedures, and SQL as the backend. QTC provided the architecture and framework which was divided into multiple layers, each section pertaining to the Model-View-Controller (MVC) pattern. The scripts, styling, and coding environment can be reused respectively. Lastly, our reusable class library can be used for other projects.

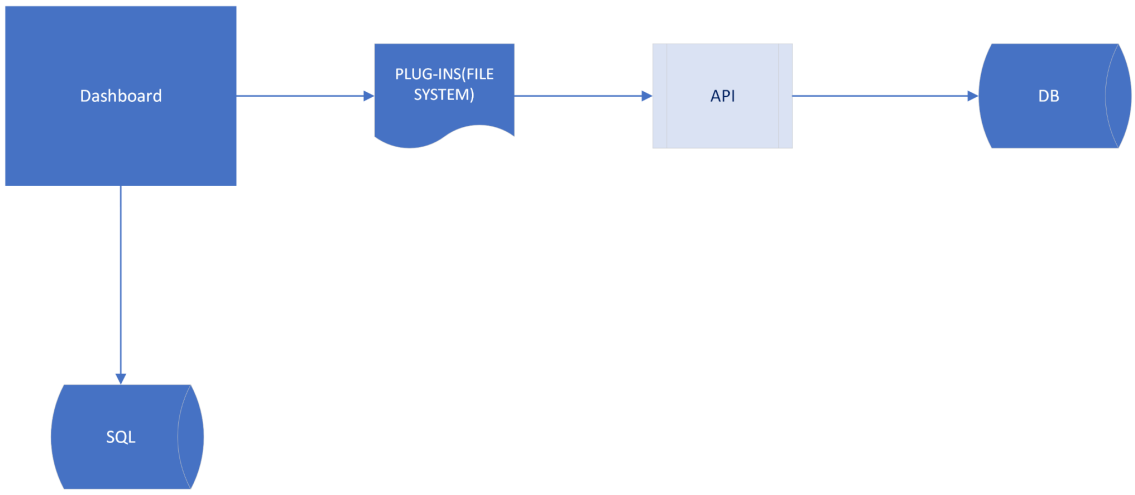
3. System Architecture

3.1. Overview:

QTC provided the framework which is what we used for the project. Each line of business is isolated, their business logic is housed in an assembly plugin, ending with the file extension DLL. Upon initial run, the application will scan a certain folder, look for the assemblies, load them, extract the business logic, and inject them into the interfaces of our website component. Each tenant may pull data from a different data source, which we refer to as integration points. If maintenance or changes are necessary for one of the assembly plugins, it can be done so without affecting the other plugins.

The diagram below demonstrates the overview of our system architecture





3.2. Data Flow:

3.2.1 Database:

The data necessary to populate the dashboard UI is stored in a database.

3.2.2 Back End:

Manipulates business logic with reference to the saved data in a database, converting and populating the views.

3.2.3 Front End:

The visual aspect of the website that interacts with the user after the data (derived from the database) is manipulated via the back end.

3.3. Implementation:

We implemented features into our smart dashboard UI and a database to hold all data displayed using said features.

3.3.1 Features:

The web page serves medical record files and the errors they contain that are implemented through multiple interfaces.

- **Side Navigation:** The user can direct themselves through the web pages and different tenant databases. The smart dashboard has a side navigation bar that has

built in links, which will then redirect the user to the correlating interface. The Smart Dashboard sidebar has two integration points listed under “Applications” and when the user clicks on either point or just the Dashboard link they will get redirected to the correlating webpage.

- **Help Button:** Allows the users access to a guide on how every feature is used. The Smart Dashboard has a button built into the side navigation with a question mark on it to hint users and if they have any questions they can find the answer on how to use the website's features there.
- **Authentication:** Allows only certain users access or permissions to view or use certain web pages. The Smart Dashboard only allows certain users to view some specific pages based on the user’s login information.
- **Pagination:** Allow the user to expand or contract the number of errors displayed on the web page. The Smart Dashboard displays tables that list all the errors from the data submitted. Below the table in the center the amount of errors are split into multiple pages and allows the user to control what page they are viewing.
- **Filtering:** Allows the user to filter errors by categories. The Smart Dashboard displays tables that list all the errors from the data submitted. The tables have a built-in filtering system where when the user chooses a category from the table the error reorganizes themselves alphabetically, numerically, lowest to highest vice versa, etc..
- **Search Filtering:** Allows the users to filter through all the errors that retrieve the closest matching error to the keyword the user typed. The Smart Dashboard displays tables that list all the errors from the data submitted. Above the table to the right is a text box where the user can type in a query that the table will start to reorganize itself to find the closest matching error to the query typed.
- **View More Info:** Allows the user to view the full error message. The Smart Dashboard displays all errors from the data submitted. In every error there contains a column with an eye icon that will allow the user to see more details about the error when clicked.

4. Conclusions:

4.1. Results:

The Smart Dashboard was developed in order to help QTC consolidate errors that have occurred in different systems from different applications. Our design allows the application to load new business logic from different tenants by compiling their code into an assembly and placing it in a folder. At runtime, the web application will scan the folder, load the assembly, extract the business logic, and inject it into the interfaces of our web component.

The errors displayed on the screen are from the specific integration point and tenant. It could be the case that the tenant pulls errors from a different data source apart from SQL and Oracle which is possible by coding it into a new assembly and placing it into our assemblies folder in the website component.

Multi-tenancy was accomplished as well as error retrieval from different data sources for the specific tenant. Lastly, front-end features were implemented to improve the user experience such as data limiting, pagination, help modal, long error message truncation, view button to see the long error, and partial authentication.

4.2. Future:

With this application, QTC can see different errors from different data sources for multiple tenants. Each page that displays the errors is specific to that one data source and tenant which makes it easy to quickly identify and view errors for a specific client (tenant) and a data source such as a SQL database. Although we accomplished the latest revision of requirements, there are some future improvements that can be done to further improve the application.

- **User view** - Currently the dashboard displays all of the information for the errors, it should be the case that if the user is not an admin, they will only see a subset of those errors with less information.
- **Windows authentication** - Users should be authenticated via Windows authentication and not via passing the username and password as queries on the browser.
- **Homepage text** - The homepage should be cleaned up and not include any information that was used for development. Ideally, it should display the list of errors for the tenant that shows up first in the left hand side menu.
- **User roles/role mapping** - There needs to be two roles, admins and users. Via role mapping, admins should have the ability to see both system errors and application errors whereas regular users should only be able to see application errors. This role mapping should not be hard coded.
- **More errors button** - Button that when clicked will pull in 200 more errors.
- **Error filtering** - The admin should have a way to filter errors by system or application level.

5. Introduction (Business Rule Engine):

5.1. Background:

The Business Rule Engine (BRE) is an application that is designed to execute business rules at run time using predetermined logic. It is meant to automate business rules and functions which would otherwise be done manually. The functionality of the Business Rule engine may be accessed through either the Swagger or Postman UI. The motivation behind this project is providing QTC with an easier way to handle, classify and review, among other things, important medical data, documents and other relevant information.

5.2. Design Principles:

BRE is designed so that it recursively re-evaluates or executes a rule, until a final outcome has been reached. Using the JSON objects also made it convenient to pass on the parameter values based on which the Rule Engine will undertake in its process. The UI also makes it easy to access other functionalities of the Rule Engine, such as adding or deleting a rule.

5.3. Design Benefits:

The purpose of the Business Rule Engine is to classify, sort and evaluate patient records, among other things, with a centralized database. As it is an automated application, it reduces the chance of human error when categorizing patients. The Rule Engine being automated also allows for a faster response time, and provides customers a more efficient and personalized experience. Another benefit of it is that it allows Business Analysts themselves to make necessary changes to the business rules, procedures etc. without needing the assistance of the IT professionals of their organizations.

5.4. Achievements:

Throughout the course of the year, the team had created a functional database that could be accessed through the Swagger or Postman API. The Rule engine can create, delete and edit rules and expressions, whilst also verifying if the expressions are within the rule engine to check if the user can edit or delete an expression. The Engine can also notify users if there are any exceptions or missing information in the database. Users can also utilize the Rule Engine's functionality with their own databases so long as they connect it through Azure.

6. Related Technologies:

6.1. Existing Solutions:

The Business Rule Engine, being an automation tool, since it is a very common tool used in a whole host of industries. But, for our purpose, there were not many implementations similar to what we were doing. Therefore, we did not use existing solutions as references for our system.

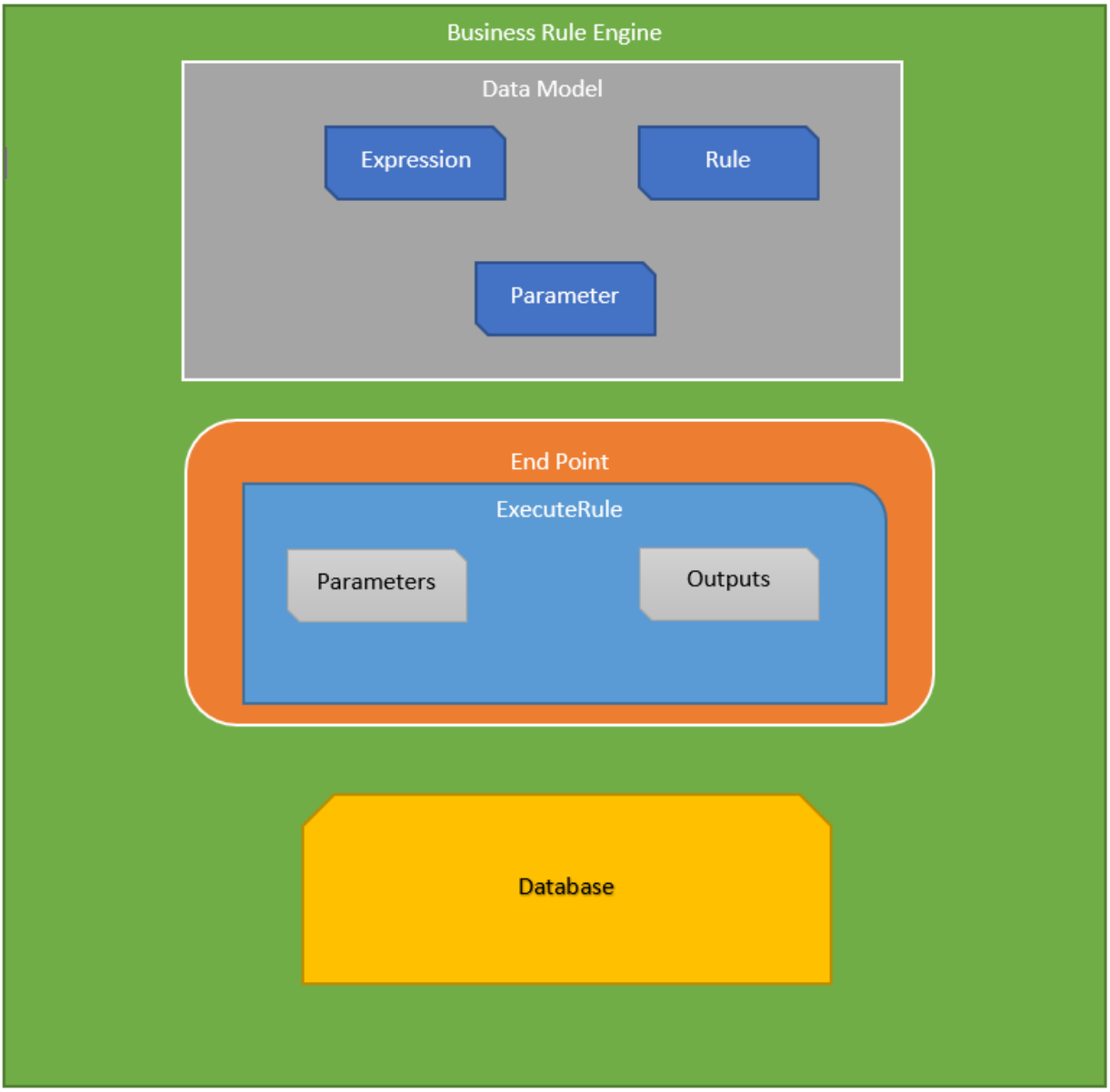
6.2. Reused Technologies:

Like the Smart Dashboard, the Business Rule engine was developed using ASP.NET, C#, and utilizes JSON and SQL Server for the backend. To organize the API methods, Swagger User Interface was used. The Swagger UI provides an efficient way for the user to get access to the API resources without much hassle. In addition to Swagger UI, another tool called the Postman API platform was used, which made it easier to pass JSON objects along with the necessary parameters into the Rule Engine.

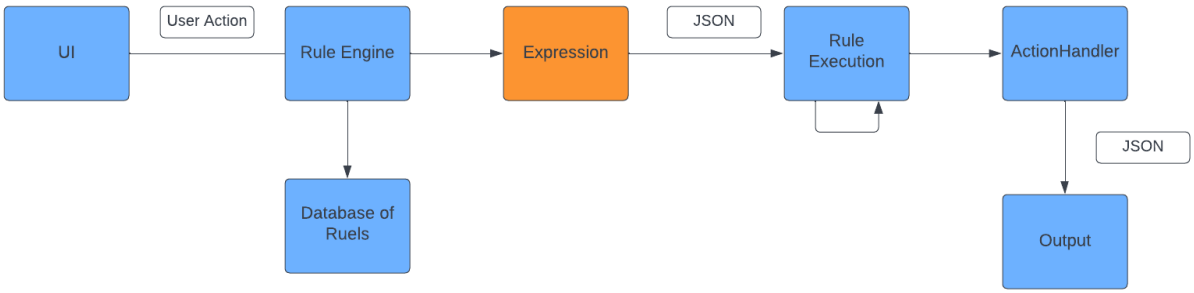
7. System Architecture

7.1. Overview:

The Business Rule Engine will consist of multiple components, which all work in tandem. One of those components is the database, which holds the data necessary for the evaluation process. In addition, it also contains a plugin (file system) as well as an authentication method. The plugin will make the API call to the database to retrieve the necessary data, which will be used accordingly depending on the parameters and inputs that are provided.


Business Rule Engine
Data Model
Expression
Rule
Parameter
End Point
ExecuteRule
Parameters
Outputs
Database

7.2. Data Flow:


UI
User Action
Rule Engine
Expression
JSON
Rule Execution
ActionHandler
JSON
Database of Ruels
Output

7.2.1 UI:

Using the Swagger UI, the user will be able to add, delete or edit a rule. Using Postman, the user is able to pass parameters into the Rule Engine for the execution of rules

7.2.2 Rule Engine:

Checks the Database of Rules to see if it exists and sends the expression to be executed if it does. Contains all the necessary logic to execute the rule. The final output will be displayed to the user based on either a “Positive Action” or “Negative Action” which the Rule Engine will evaluate and determine

7.2.3 Database of Rules:

All saved rules and expressions are stored in a database.

7.2.4 Rule Execution:

Executes rules accordingly and will execute recursively when required. Afterwards it passes it on to the ActionHandler.

7.2.5 ActionHandler:

Handles the Executed rule and outputs the result to the user.

7.3. Implementation:

The rule engine can be accessed through Swagger as the base UI or through the Postman interface. Both UIs allow users to test each individual function and rules.

7.3.1 Features:

- **AddExpression:** Users can add expressions to the database unless it already exists.
- **EditExpression:** Edits an expression but returns an error if it doesn’t exist.
- **DeleteExpression:** Deletes an existing expression.
- **GetAllExpressions:** Shows a list of all expressions.
- **AddRule:** Similarly to AddExpression, users can add rules to the database so long as it doesn’t exist.
- **EditExpression:** Edits an existing rule but returns an error if it doesn’t exist.
- **DeleteRule:** Deletes an existing rule.
- **GetAllRules:** Shows all existing rules.
- **ExecuteRule:** Checks to see if the rule exists in the database before running the rule’s functionality.

- **Schemas:** Lists the variables each expression or rule incorporates, for example CreateRule works with only string values.

8. Conclusions:

8.1 Results:

The basic functionalities of the Rule Engine were successfully implemented. The Rule Engine is able to take in multiple parameters with one or more JSON objects. Despite there technically being no User Interface, the Rule Engine is easy to access using the Swagger UI and the Postman API. The one feature that was not able to be implemented was Nested Expressions, which are expressions within other expressions, which requires more complex logic to be able to evaluate. There was a lot of trial and error involved in the process, and eventually we found what worked.

8.2 Future:

The Business Rule Engine could be scaled up to a larger level to be able to support multiple types of rules and multiple use cases. Organizations could have multiple implementations of it depending on their needs. Instead of Swagger UI, a more dynamic and fluid system could also be used on the Front End. Overall, the BRE could also be made more user friendly.

9. References:

- C# Documentation:

  C# Tutorial (C Sharp)

- Visual Studio 2022 Documentation:

  Visual Studio documentation | Microsoft Learn

- SQL Server Management Studio Documentation:

  Download SQL Server Management Studio (SSMS)

- Swagger API Documentation

  Swagger Tutorial

- Postman API Documentation

  Postman Download