Cody Cusey

CSCI 299 – Thesis/Capstone

Capstone Project

May 02, 2025

**Asset Logging and Reporting System for American Chemet Corporation**

**Introduction**

American Chemet Corporation is a company rich in history and innovation. In recent years, there has been a concerted effort to minimize the technology gap, normalize data, automate tasks, and better recognize cost centers throughout the production life cycle for each specific product. For systems like these to work, the data management needs to be handled in a very specific manner. Due to the disjointed series of workflows, creativity is required to solve complex problems across multiple digital platforms. This project is just a single piece of an overall larger vision of a new and digitized American Chemet Corporation. The following report outlines the project justification, work performed, summary and closing thoughts on the project.

**Why?**

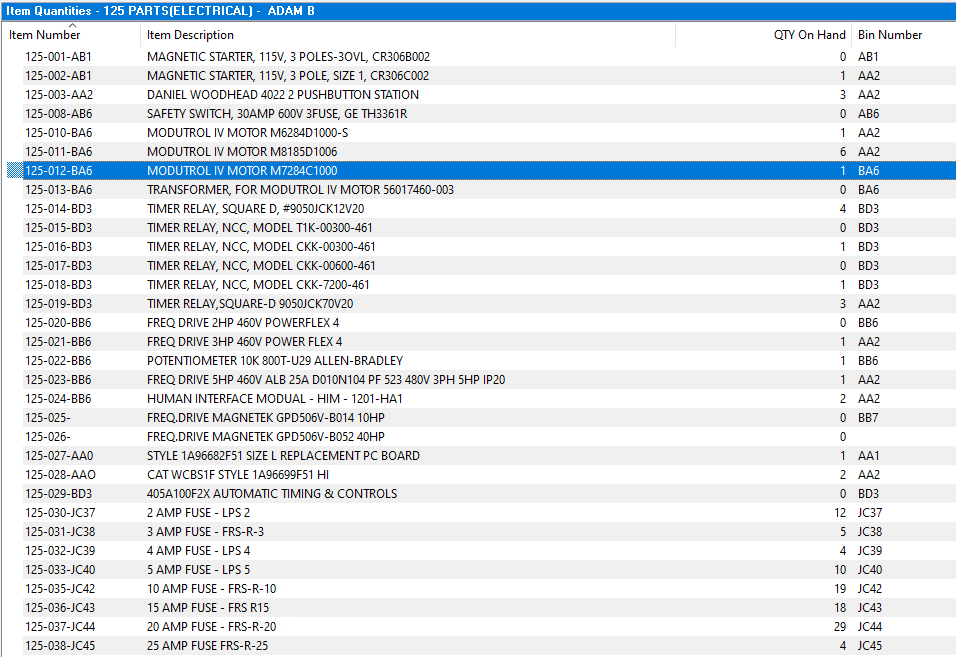
For any company or business to be successful and/or profitable, they should understand, monitor, and track all costs associated with operating. With a small company or relatively simple production model, capturing costs for manufacturing goods can be much simpler, and handled without the need for more advanced data management. For other companies, such as American Chemet Corporation, the amount of data to capture is seemingly unattainable. In an effort to migrate the current data logging system away from “I’ve got a spreadsheet for that,” to “It’s in the system,” American Chemet Corporation has begun the process of data normalization via online asset logging. This formal data collection system will allow American Chemet Corporation to better recognize cost centers. The frontend web application is written in C#, the backend database is Microsoft SQL Server (MSSQL), the Enterprise Resource Planning (ERP) software is Microsoft Great Plains (GP), the reporting service is SQL Server Reporting Service (SSRS).

**Start**

After a short introduction with the newly hired IS Manager, Wayne, we began the process of adding a new user to the sandbox, creating a Microsoft SQL Server Management Studio user, restricting and limiting permissions, and getting connected to the test database. This may seem like a trivial component to a project, but when working in the real world, there is no thing as too safe or too secure. Whether its user permissions or privileges, secure passwords, or a physical security measure, I believe it’s best to deny all first and allow only what’s necessary as it is needed. If your company uses user groups or a similar user grouping tool in their network infrastructure, you can create specific user rolls and assign each user as needed. This ensures data integrity and security, as well as assurance of each user and privileges. Additionally, if any changes are made to the user roll, they are passed down inherently to all of its users.

**SQL Work – Views for SSRS Report**

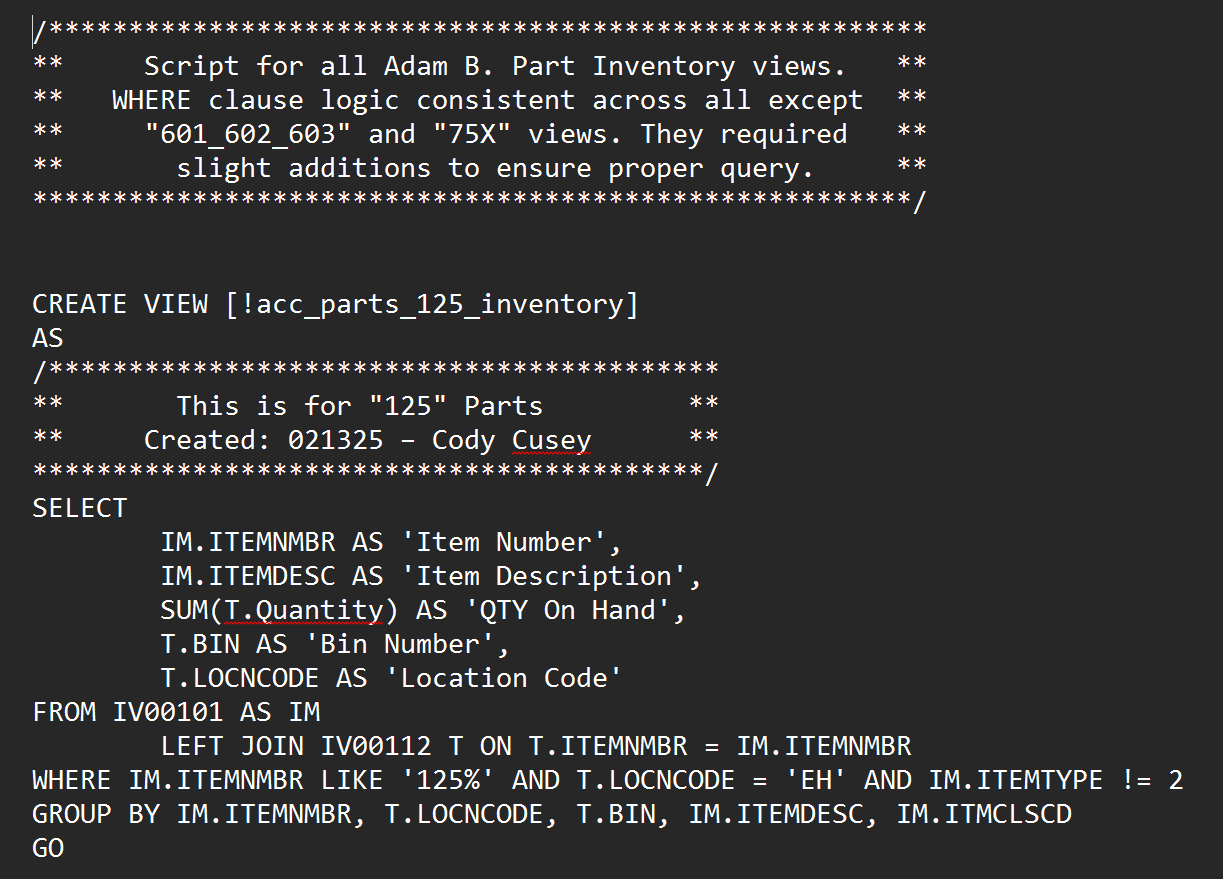
Once we had safely and securely allowed access for my new user, I was quickly tasked with my first piece of real work. I was instructed to create several SQL Views, each with very specific query parameters. This all sounded interesting and exciting so naturally I began to research about SQL Views. My search led me to sites like Microsoft.com, W3Schools.com, and Stackoverflow.com where I discovered that Views are a virtual table. They have rows and columns just like a real table but are only a reference to the SQL Query results from each real table. The major benefit to a view lies in the refresh of data. When you create a view, you’re telling a larger linked dataset to only return the specific items you have selected. This works well for inventory items with similar naming conventions.

*Example: Item# 125-000 through 125-999 are all electrical parts.* 

The query to return just those filtered items could be ran hundreds of times per day and it’s much more efficient to refresh the data in the view than to query the entire database each time. After spending time researching and learning about SQL Views, it was time to create my first View. Wayne had provided me with a document containing frequently used tables in Microsoft GP. By studying these tables, I was able to locate the correct item inventory tables from which to query my data. With a simple JOIN statement to link the “Master” item table with the “Inventory” item table, I specified my WHERE clause arguments based on the following criteria:

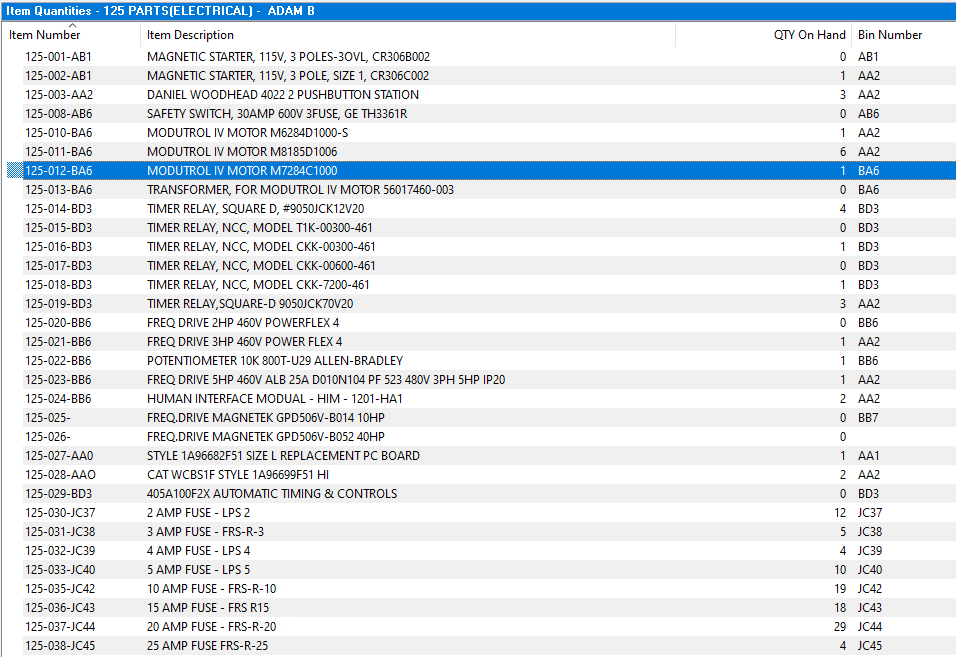
* ITEMNMBR LIKE ‘125%’ (begins with 125 and can have anything else following)
  + This gets the part numbers that start with 125
* T.LOCNCODE = ‘EH’ (Inventory Item Site in GP)
* ITEMTYPE != 2 (2 = Discontinued in GP)

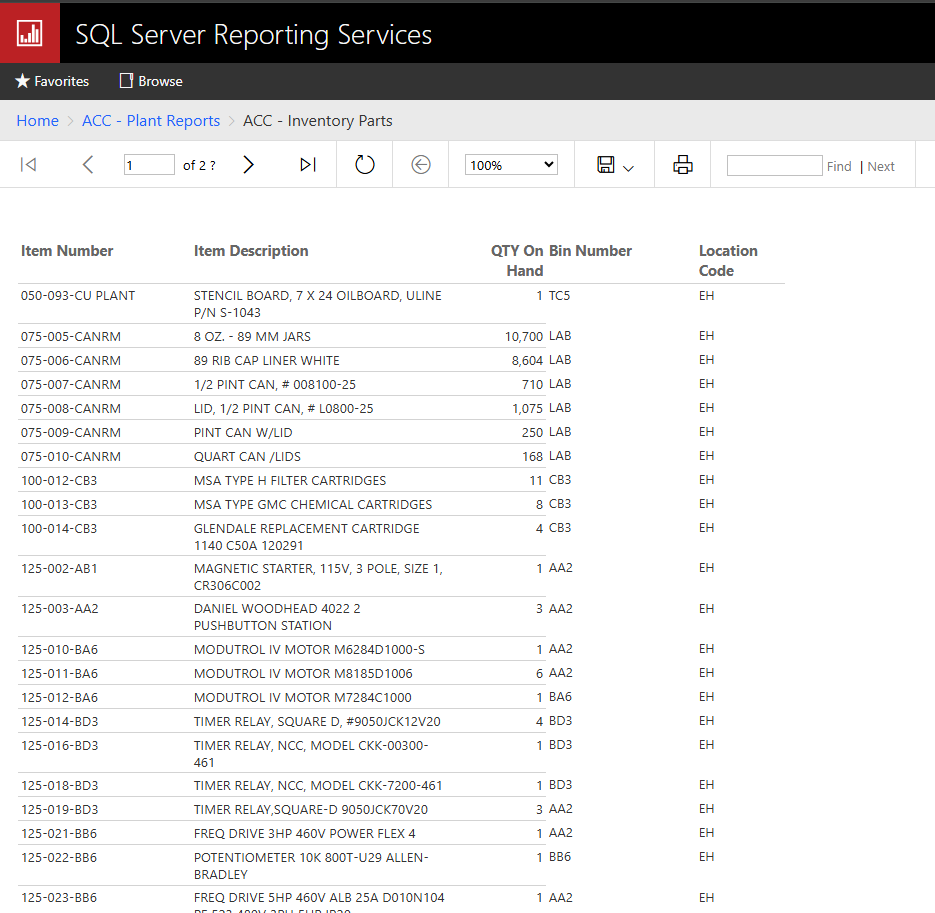
After I had my query ready, I followed the syntax from an example on W3Schools.com and with a little guidance from Wayne, I was able to successfully create my first View!

*Below is a screenshot from the Script File used to Create all the Views.*

I was able to successfully produce a working script that created over fifty views that are the backend for an SSRS parts report. By creating these views, the Microsoft GP inventory quantities can now be seen by front end users who do not (and should not) have access to inventory, accounting, or financial data. This style of reporting service provides greater companywide transparency, while retaining the data integrity by limiting user interaction.

*Below are screen shots of the GP Inventory and the SSRS Report.*

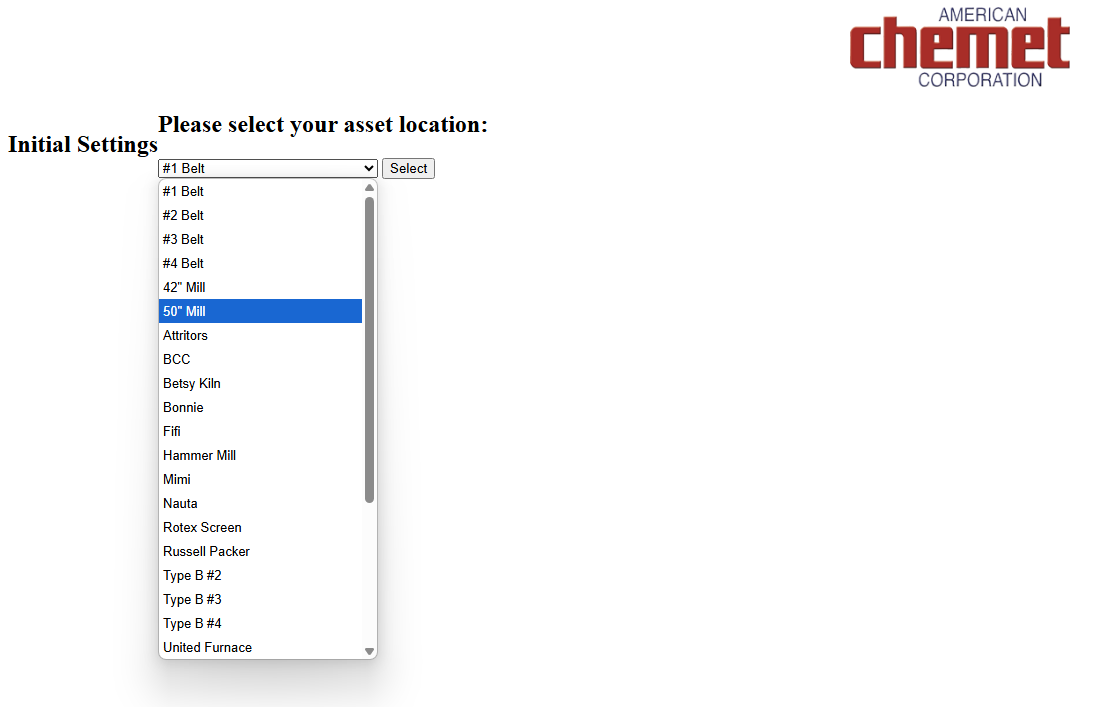




This first bit of SQL work proved my abilities, aptitude, and understanding of the data structure, databases, and overall SQL knowledge. The high quality of work immediately led Wayne to move forward with generating user credentials for the live system. We followed the same intensely cautious methodology while creating and securing my access. Any time working with real data, in a real company, it is paramount to exercise extreme caution.

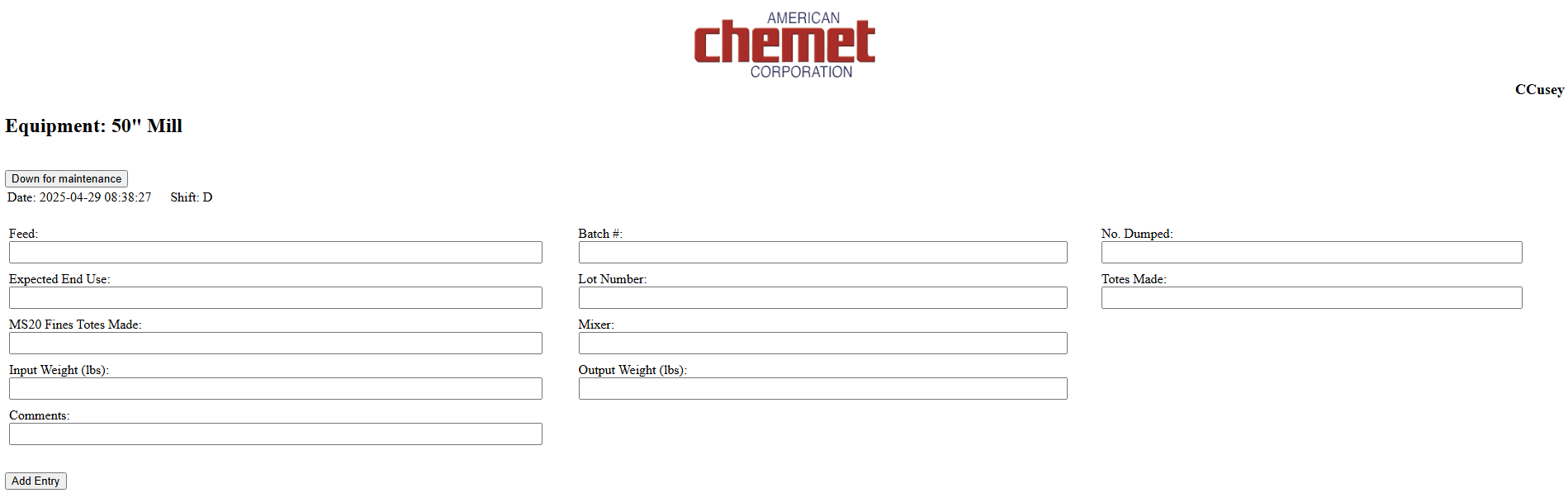
**SQL Work – Stored Procedures for SSRS Reports**

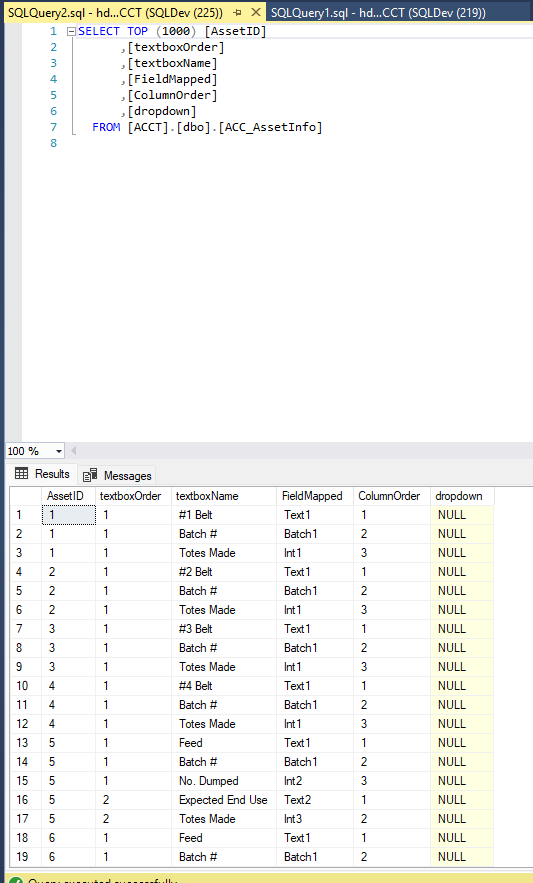
While Wayne worked on the front end web application, he required assistance on generating stored procedures for each of the company assets. The asset logging web application has a simple layout and UI, attempting not to distract the users from performing quality data entry. Each of the assets listed have variable input fields as seen in the screen shots below.

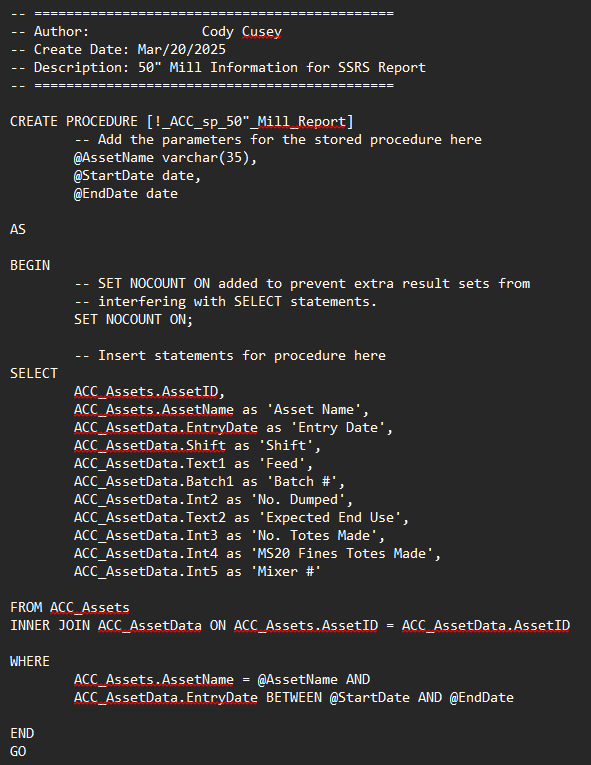




While I inspected the database tables, I discovered an interesting configuration of the data entry table and its’ linking to the asset data table. Each of the input boxes was setup as a Row / Column pair (as textboxOrder & ColumnOrder in the database), meaning that “Feed” was in Row 1 / Col 1, “Batch #” was R1 / C1, “Lot Number” was R2 / C2, etc. Reference images are below.





This means that each Asset (AssetID) could have anywhere from 1-X amount of rows and/or columns, and the data will still be normalized. This was an epiphanizing moment for me, as I was beginning to understand the creative problem solving happening to handle to extreme variability and volatility of the data in our systems. Once I had a firm grasp of how to tackle the problem, I began writing my first Stored Procedure for a single asset.

I was able to write the stored procedure above without much hassle or headache and test it by running the following command in the query window.

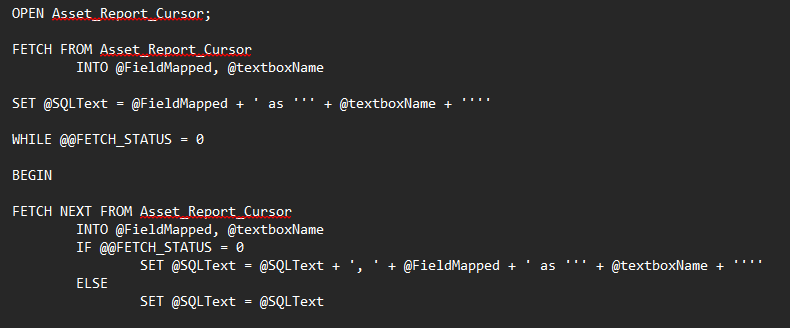
EXEC [!\_ACC\_sp\_50”\_Mill\_Report]

@AssetID = 6, @StartDate = ‘2025-02-14’, @EndDate = ‘2025-02-15’

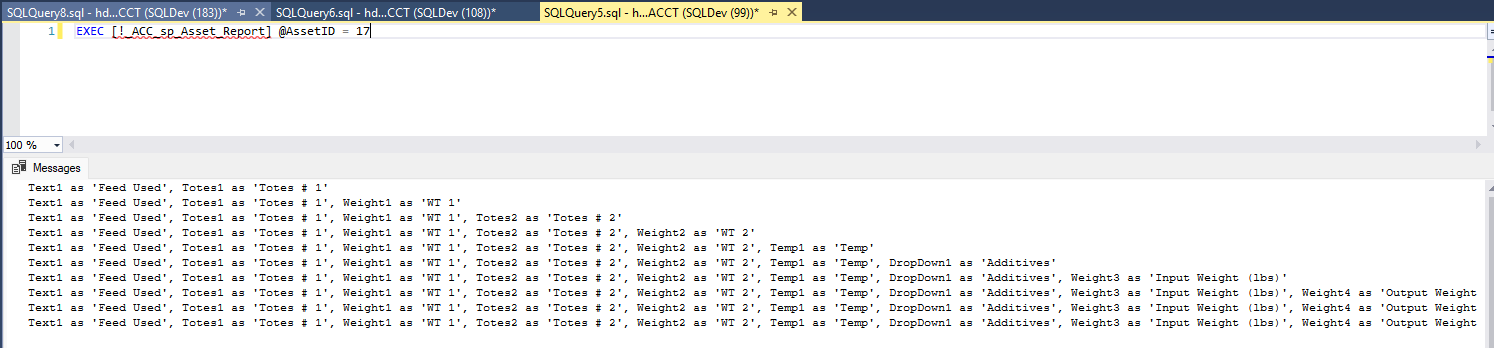
Based on the data that was in the test database, I was able to successfully validate the functionality of the Stored Procedure. I quickly realized that this strategy was going to be extremely cumbersome and tedious, as each asset would require its own Stored Procedure, with its own specific SELECT criteria and aliasing. This would have required a lot of repetitive coding and wasn’t very flexible. If anything changed, like data field name, field location, field order, or field additions or subtractions, the Stored Procedure would have to be updated for that specific Asset. I spent some time researching more dynamic ways to query non-normalized data or how to internally normalize it for a more robust system. After days of searching for different methods for performing dynamic SQL queries, I stumbled upon Cursors. A Cursor in SQL is a database object used to retrieve, process, and manipulate data one row at a time. While SQL is designed to handle large data sets in bulk (set-based processing), sometimes we just need to focus on one row at a time. It allows processing query results row-by-row instead of applying operations to the entire set. They give us more control over result sets and are especially useful in complex logic, conditional updates, or procedural programming.

I felt like I was on the right path to creating something flexible, robust, and dynamic so I consulted Wayne to solicit feedback on the login and implementation. He was thrilled to hear that I was exploring dynamic SQL, going beyond his simple request for dozens of stored procedures. With confirmation and validation on my side, I set forth on a multi-day learning quest, exploring many sites and video content looking for good examples and explanations of Cursors and dynamic SQL.

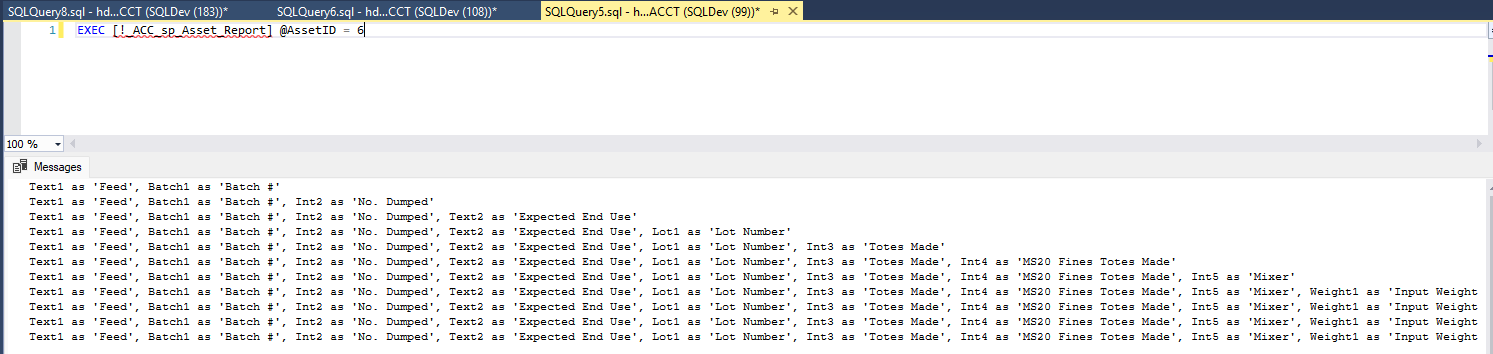
Once I had “the idea” logically programmed in my head, I attempted to create my first dynamic SQL stored procedure. As one would expect, the results were less than extraordinary. Through continual learning and trial and error, I was finally able to return the specific value desired! Now equipped with a quasi-functional stored procedure, it was time to attempt to dynamically build a string based on the returned values from each row by row looping. Cursors effectively act as a WHILE LOOP and by the use of WHILE LOOP logic, I was able to successfully build a SELECT statement via string concatenation.



The Cursor is fetching just the specified parameters @FieldMapped and @textboxName for each row that exists. WHILE @@FETCH\_STATUS = 0 just means “while there is still more data to fetch”. When there is no longer data in the database, @@FETCH\_STATUS = -1. The @SQLText variable is being used to build the select statement, casting aliases to each of the SELECTED fields. With this simple looping logic and dynamic string building, I was able to successfully create a dynamic Stored Procedure using Cursors! This means that with a single parameter input, I can query every field associated with the AssetID input. This is the perfect scenario to handle the non-normalized datasets within this data structure.

*Printing out @SQLText each time through the loop showing the dynamic build.*

*Using a different AssetID parameter to demonstrate the flexibility.*



Now that I have my dynamic SQL Cursor stored procedure properly collecting the data and building the SELECT string, I can return that string as an OUTPUT variable. The magic happens when you nest this stored procedure inside of another. This will pass the same single AssetID into both stored procedures, generating the SELECT string and passing that OUTPUT variable into the outer stored procedure. The outer stored procedure performs a simple SELECT statement but passes in the AssetID parameter along with the OUTPUT variable to generate the final SELECT statement used for the query.

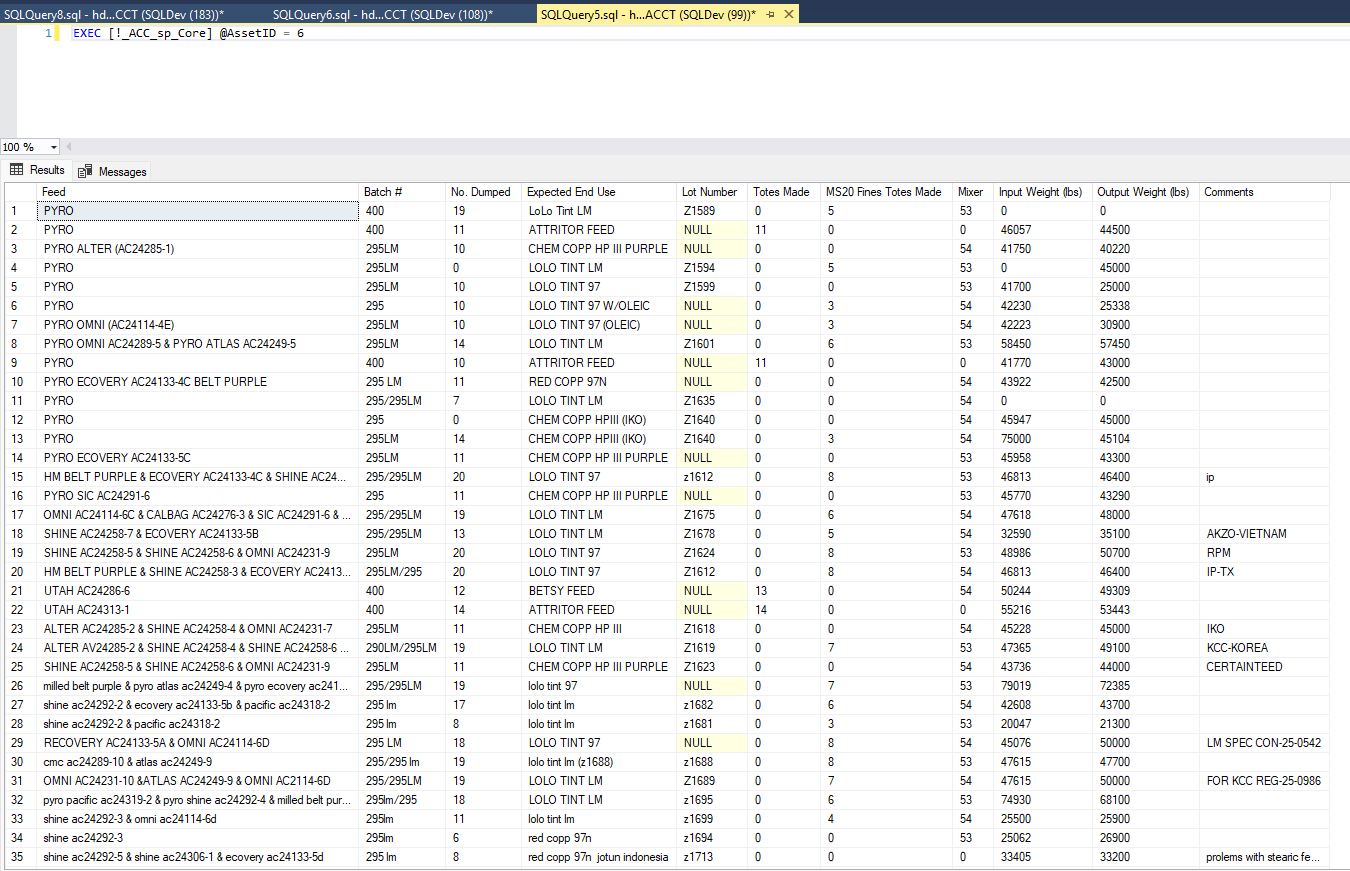
The screenshot below shows the nested stored procedure syntax.

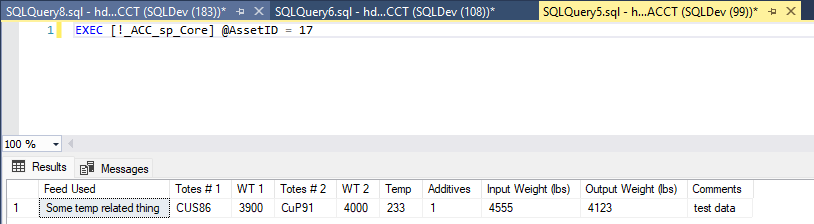
A screenshot of a computer

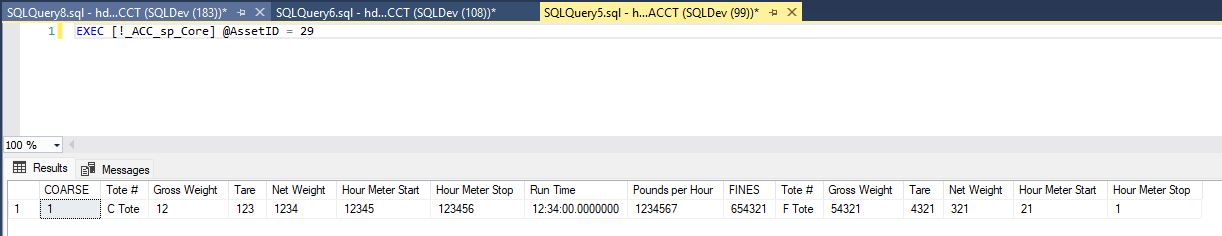
AI-generated content may be incorrect.

I was able to test my final Stored Procedure by running the following code in the query editor.

EXEC [!\_sp\_ACC\_Core] @AssetID = (6, 17, 29)





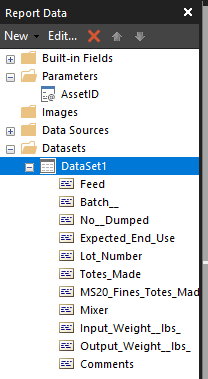


These were the only three assets with a full dataset so they were primarily used as my test set. The “50” Mill” (AssetID = 6) was the primary test set. I was receiving the desired output given the input, so I am assuming it will continue for the rest and more as we continue to implement and improve the system. As you can see above, each one of the unique AssetID’s queried against provides an exact match to the web application data field labels and their corresponding user entered value. This simple, single value, entry point allows for easy use and interaction with our data reporting system, SSRS. Additionally, the flexibility and versatility of the nested dynamic SQL logic, easily allows for adjustments and growth withing the existing data structure, without having to rewrite or alter any stored procedures.

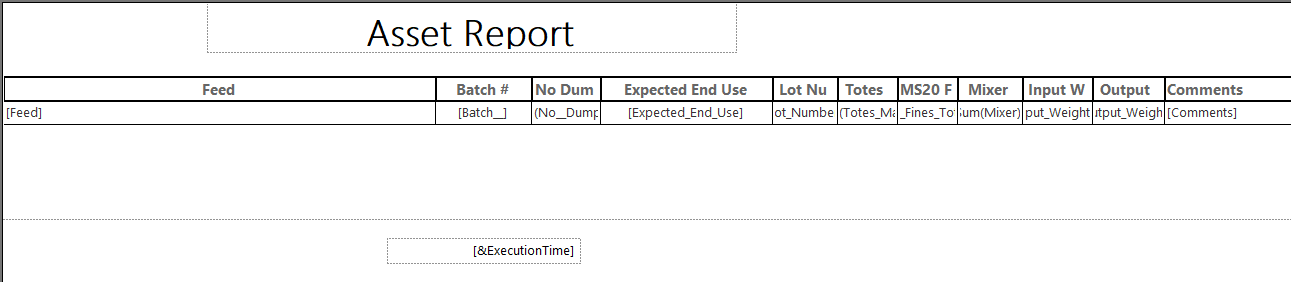
**Final Step – SSRS Reporting**

Now that I have an extremely flexible, versatile, efficient, and adaptable nested dynamic SQL stored procedure, using it to generate formal reports is a breeze with SSRS. By design, this program is designed to bring in data from SQL data sources so by having such a robust stored procedure used to query the data, creating datasets for reports can happen quickly and efficiently. Below are a collection of screenshots showing how a well designed stored procedure can easily detect all the fields, table headers, and values for any given parameter.

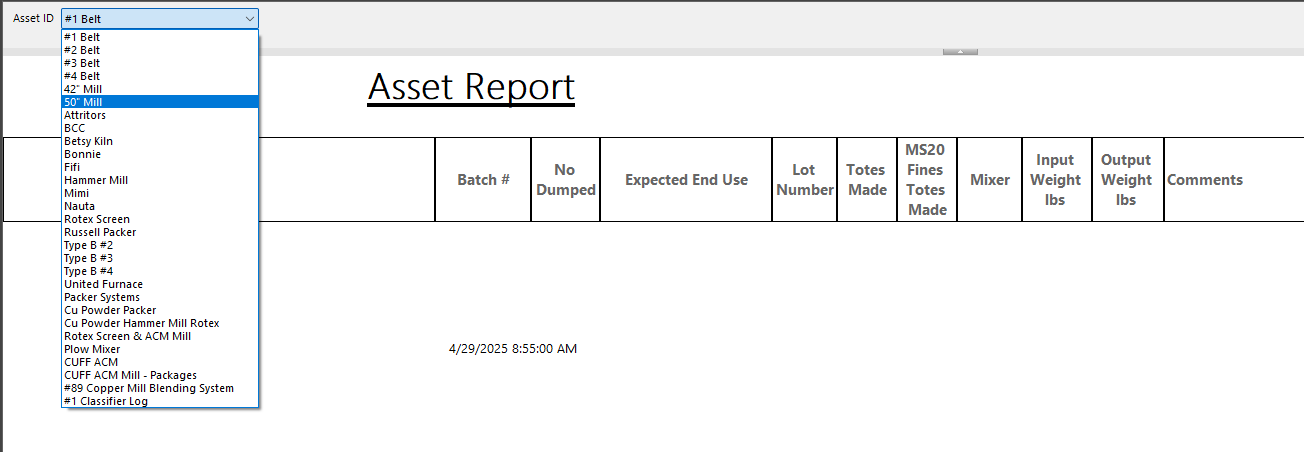
*Dataset Import – Follow the connection wizard to connect to your target database. One connected, you can navigate to your stored procedure and select that as the query criteria. The wizard will run the query against the defined stored procedure and sense each of the fields being returned. You can select and deselect certain fields to further customize the data import prior to reporting.*



*Report Designer Mode, Table Headers were populated from Stored Procedure. The blocked cells are the values corresponding to the table header field names.*



*Selecting an Asset from the dropdown list. Notice the label says Asset ID, since the parameter required by the stored procedure is an integer value, we can assign common naming conventions to each Unique ID to further enhance user satisfaction and minimize confusion.*



*Running the Final report! The built in formatting tools allow the report designer to customize the appearance of their report easily. The formatting interface mimics that of other popular spreadsheet and word processing software.*

A screenshot of a computer

AI-generated content may be incorrect.

Now that the report has been designed, built, and deployed, we can make it public to the appropriate user groups on our network. In the future, I would like to explore reporting options that incorporate the same dynamic nature of the stored procedures that allow it to function so well. For now, the dynamic nature allows the creation of datasets very quickly and simple but cannot dynamically build the report table based on unique datasets. My goal was to have a functional Asset Reporting System finished and I feel like I accomplished my goal!

**Closing Thoughts**

Throughout the duration of my coursework, I have always been actively pursuing technologies, systems, programs, and ideas that could add value to American Chemet Corporation. I quickly gravitated to databased and database management as it is something that we have multiple of, and virtually everything operates utilizing. Data is embedded in everything we do and the only way most of these thing’s function is by having data integrity and data security. Data integrity and human error is a curse that has plagued American Chemet Corporation for as long as I have been there. Archaic methodologies such as: printing forms, filling out by hand, then scanning back for digital record, a spreadsheet for everything, and multi point human data handling. This system is a step in the right direction for American Chemet Corporation. While crude and young, it’s the correct way for our company to capture all of this important and unique data. With a simple UI, asset specific naming conventions, and ease of use, this simple tool will allow American Chemet Corporation to begin tracking the cost / lb. for each asset and its’ utilization. I am proud of the creative solutions I was able to implement to make the reporting system as simple and efficient as possible. I am excited to see what the future has in store for the digital age of American Chemet Corporation and any additional projects I may be so blessed to work on.