**Conducted By:** Grey Files

**PCR:** CUSV-8148 Create Fault Manager tool

**Choice: MySQL**

# Overview

When creating the Fault Management tool, we are faced with the decision of a database management system to host the fault data. One of the core ideas behind the tool is to centralize the storage and maintenance of the list of faults, which fits nicely with the concept of a database. The fault management tool will need to interact with the central database behind the scenes which will store all data in the system. The tool itself will be responsible for sending updates to the database based on what the user changes/deletes/adds on their local computer.

# Desired Outcomes

The chosen database management system will be able to manage all fault data in the system in one central location. It will be relatively easy to query and send updates to so we can build the tool to automatically interact with it. It will also be able to be accessed by any person on the local network in Hunt Valley while being secure enough to not be exposed to the internet. The chosen system should support usual database features like transactions and rollback although these features would only be useful if the tool was extended, or the database was accessed through the command line to make manual edits.

The selected database management system combined with the application will be able to support exporting into one of the available formats while offline. This may be useful when attempting to export when not on location in Hunt Valley or not connected to the internet to VPN in.

# XML Databases

XML Databases are databases that store large amounts of data formatted in XML. This means they are non-relational databases and therefore do not store their data in tables. The XML format is by nature a hierarchical data format and therefore lends itself well to hierarchical data. In our case, the data is not very hierarchical; The easiest way to organize it would be to have the software releases as the top level with associated faults at the second level and each fault’s associated data in the third level.

Performance of XML databases can be poor especially if it is an XML enabled database instead of a native XML database. This performance limitation is not a concern to us as the biggest transaction we are anticipating having with the database in our tool is about 1200 entries which almost any professionally used database system could easily handle.

Lastly, the APIs available for interacting with XML Databases exist but are not extremely common. We almost definitely would be able to find an open-source library that we would be allowed to use, but it probably wouldn’t have super expansive documentation and support. Because of the prominence of relational databases, use of an API to interact with an XML database may be a headache.

# SQLite

SQLite is one of many popular relational database management systems. It stores data in tables which can be defined, linked together in different ways, and slightly changed down the line. The nature of relational databases means they are limited to dealing with data that can be fit into a traditional table and are hard to change down the line once data formats have been defined. For our data, this type of database seems more appropriate because the faults and their attributes can easily be formatted to be stored in a table (and currently already are in the FM spreadsheet).

SQLite is not the fastest of the relational database systems as it has not been optimized for performance like some of the other ones have. Like I mentioned earlier, though, this will not be a large concern for our purposes because we do not expect large amounts of data or traffic to the database.

What makes SQLite unique from other systems is that is stores the data for a database as a simple file on the hard disk. This means that the actual database data will be visible as a file in the drive and could be copied to be accessible offline. Although this may be useful if someone wanted to access the fault data offline to export, it would create the issue of then allowing for changes to a database that is separate from the central one. This does not seem like an ideal way of making the fault data accessible offline.

Lastly, the API available for interacting with the SQLite database is easy to use and well maintained/documented. It would not be a problem to find an API that we could use in the tool.

# PostgreSQL

PostgreSQL is another relational database management system that is highly regarded in the industry. Like SQLite, it has all the expected features of a relational database system and would store our fault data for this tool naturally in table format.

PostgreSQL is considered a fast database management system that can handle many users and a lot of data quickly. As mentioned already, though, we are not anticipating this being relevant due to our use case not requiring lots of data or many users.

Unlike SQLite, PostgreSQL does not store the data as a visible file on the disk. The system stays running on the host computer and manages the data through API requests sent over the network. This option, therefore, would require a full host computer that has a storage medium to store the data. It would be easy to configure the database to listen for traffic on the local network but not be open to the internet.

Lastly, the API available for using PostgreSQL is very common and available. It also has plenty of documentation to make using it easy.

# MySQL

MySQL is a relational database management system that is very commonly used across the industry. Like the other relational database systems, it stores the data in tables and would suit our data nicely.

Although not the absolute best like PostgreSQL, MySQL still offers good speed when the database is under heavy use. Again, this is not a concern for us with small transactions and low traffic.

MySQL is simpler in features than PostgreSQL. It is not as simple as SQLite, though, and does not store its data on the disk in a single file like SQLite does. The lack of features compared to PostgreSQL is not a concern to us because we do not need the extra features. Things like the extra supported data formats and the ability to have table inheritance will not be used by us.

MySQL is probably the most documented and well known of the options and therefore has a plethora of resources to help us develop online. There are many readily available libraries that we can use in our tool to interact with a MySQL database.

# Decision

We will be using MySQL to host our database on a computer that will be on the local network. It offers fast, reliable, and easy use within our tool and can be set up how we envision. The main advantage of XML databases is the ability to store data in XML format, but because our data is not already in that format and doesn’t lend well to it, other options would be better. SQLite is a very simple database management system that does not have the complex features we would not use like something like MySQL or PostgreSQL would have. It also offers the simplicity of having a single file in the file system be your entire database. This feature, though, would introduce the opportunity to have multiple copies of the database and make changes to only a local copy which is what we are trying to avoid. Lastly, PostgreSQL works very well for our application but is slightly more complicated than MySQL which may introduce extra work and confusion. Lastly, MySQL would work very well just like PostgreSQL and is a simple and robust system for databases like the one we are going to create. Although both MySQL and PostgreSQL could be used successfully, we will go with MySQL to favor simplicity as we don’t need the more complicated features.

To address the problem of offering availability offline, we are planning to create a file format that can be saved locally. The application would allow the user to save the faults and associated data for the software release that they are currently viewing as a CSV (possibly with a different file extension) on their local computer. While offline, they can then load that CSV and export like they would normally be able to.

# Next Steps

Concerning the database specifically, we will first need to experiment with the engine ourselves and build our tool to be able to successfully interact with the database. Then, we will need to choose a host computer on the network and set up the database on that computer for LAN use.

## Sources

[Kasun Siyambalapitiya (Medium)](https://kasunsiyambalapitiya.medium.com/how-to-setup-a-mysql-server-on-local-area-network-lan-c3c5012c7d6b)

[MySQL Connector/C++](https://dev.mysql.com/doc/dev/connector-cpp/8.0/devapi_ref.html)

[Integrate.io](https://www.integrate.io/blog/postgresql-vs-mysql-which-one-is-better-for-your-use-case/#:~:text=The%20Critical%20Differences%20of%20Postgres%20vs%20MySQL%3A&text=MySQL%20is%20a%20simpler%20database,pure%20relational%20database%20(RDBMS).)

[IBM](https://www.ibm.com/docs/en/db2/11.5?topic=overview-comparison-xml-relational-models)

[tutorialspoint](https://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm)

[Java T point](https://www.javatpoint.com/what-is-xquery)