S.A.R.C.I

Search And Rescue Coastal Intelligence

Feasibility Analysis

11/05/2021

Team Mentor: Han Peng

Group Members: Dylan Woolley, Vidal Martinez, Randy Duerinck, Jabril Gray

Team Sponsor: General Dynamics Mission Systems

Table Of Contents

1. Introduction -- 1
2. Technological Challenges --- 2
3. Technological Analysis -- 5
    3.1. Simulator Subsystem --- 7
    3.2. Reader Subsystem -- 11
    3.3. Orchestra Subsystem --- 14
    3.4. Database Subsystem -- 16
    3.5. Website Subsystem --- 20
4. Technological Integration -- 34
5. Conclusion --- 35

1. Introduction

Imagine someone is out deep sea fishing. They are catching fish, having a great time when suddenly, a massive storm rolls in. Humongous waves start crashing into the boat, water making its way onto the deck. They lost track of time, and now there is no way to make it back to shore in time to beat the storm. They reach for their radio and send a distress signal over the Very High Frequency (VHF) Marine Distress Channel 16. The Rescue21 system receives this call, and the Coast Guard is dispatched to their location. The Rescue21 system is a system of around 260 Remote Fixed Facilities (RFFs) which have towers with directional finding antennas to pick up VHF Marine band transmissions. Luckily, our client, General Dynamics Mission Systems, built and maintains Rescue21 for the Coast Guard. Though our story is hypothetical, it serves to show what Rescue21 and the Coast Guard can achieve. According to General Dynamic Mission System's website," As of January 15, 2016, the U.S. Coast Guard has performed 85,167 search and rescue (SAR) cases using the advanced capabilities of the Rescue 21 system." With this updated tally, between 06/16/2010 and 01/15/2016, the number of SAR cases involving Rescue 21 has an average of 1,074 SAR cases per month.". It is an impressive feat, but General Dynamics believes the Rescue21 system could be improved further.

Rescue 21 currently sends technicians to the RFF site if there is ever a problem and detecting problems remotely with RFF sites is not as easy as General Dynamics would like. Plus, there is no easy way to get local weather data. For these reasons, General Dynamics reached out to NAU to start a two-year capstone project that will address these issues.

The first-year capstone project built a device called the Site Weather and Power Recorder (SWAPR). A SWAPR device will sit at the RFF sites and monitor the power levels of the four VHF radios transmitted over one antenna, temperature, humidity, rain, wind speed, and wind direction. The problem with this project is that the output is horrible for humans to read and understand, so the next part of this project will focus on the software side.

We will be building off this device for our capstone project. We will solve the challenge of taking all of the data from the SWAPR devices and displaying this information in an easily understandable GUI by building a secure website. We will have two summary GUI's and one historical summary GUI, which can export the data to a .csv file. Lastly, we will need a way to notify the appropriate users of any problems occurring at RFF sites. By the end of this project, our client General Dynamics will be able to present this project to the Coast Guard. If they decide that they want the project's functionality, then General Dynamics will implement the project into the Rescue21 system.

2. Technological Challenges

Our project will consist of four different subsystems for our minimal viable product (MVP). An MVP is the smallest product that would be considered complete by our client. We also have a few challenges that are considered extra. By extra, we mean that they are challenges to be solved after we have completed the MVP. Below we have listed each subsystem and the challenges that we will face below them. If there is (MVP) next to it, that is a challenge required for the MVP. If there is (Extra) next to it, that is a challenge to be completed after the MVP is finished.

- (MVP) Simulator Subsystem
  - (MVP) Randomly generate SWAPR data for valid and invalid data
  - (MVP) Send data over virtualized com port on RS-232 protocol
- (MVP) Reader Subsystem
  - (MVP) Read data over virtualized com port on RS-232 protocol
  - (MVP) Establish a TCP Client connection with the database queue
- (Extra) Orchestra Subsystem
  - (MVP) Connects any number of simulator and Reader Subsystems over virtualized com ports for lab environment stress testing
- (MVP) Database Subsystem
  - (MVP) Establish a TCP Host connection with the Reader Subsystems
  - (MVP) Queue TCP data collected in Amazon Web Services Simple Queue Service (AWS SQS)
  - (MVP) Ability to create database entries from data in AWS SQS queue
  - (MVP) Hosting the database server off Amazon Web Services Relational Database Service (AWS RDS)
- (MVP) Website Subsystem
  - Backend Challenges
    - (MVP) Hosting Blazor Server on Amazon Web Services (AWS)
    - (MVP) Secure authentication between subsystems as well as “faking” user accounts and role based permissions
      - (Extra) Setup Windows Active Directory to manage user accounts and role-based permissions
    - (MVP) Ability to query the database server in a secure manner
    - (MVP) Notification system that detects when there is a problem with an RFF and creates a modal for a specific user group.
      - A modal is a pop-up window that contains text.
    - (MVP) When in the graphical view of historical SWAPR data the user should be able to export the selected data to a comma separated file
  - Frontend Challenges

- (MVP) Ability to create a list-view summary of data from a user-specified range (MVP) from all SWAPR devices in the network
- (MVP) Ability to create a map-view summary of the latest data from all SWAPR devices in the network
- (MVP) Ability to create a static graphic view of historical data of one SWAPR device
    - (Extra) Have the graphical view be interactive

Our project has many challenges that we will need to address in order to have an entirely successful implementation of our client's requested software solution. Our client's most significant request will be the secure website environment that will graphically display all of the data that the SWAPR device collects. Our client wants the website to be secure, which involves having authentication, user account management, and role-based permissions.

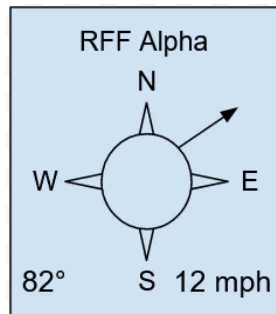

Figure 2.A A single SWAPR summary box for the list-based view

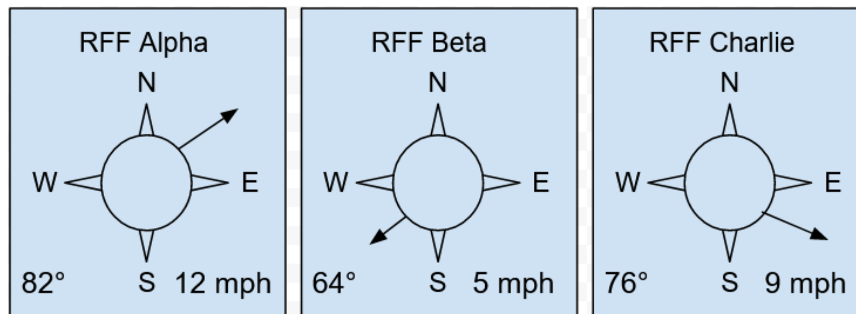

Figure 2.B A few SWAPR summary boxes for the list-based view

After a regular user has signed into our website, they will have a couple of "birds-eye" views of the SWAPR devices. The first view is a list view of the SWAPR devices, which will look like *Figure 2.A*. Each SWAPR summary box will include the site name (RFF Alpha), the temperature in Fahrenheit (82℉), the direction of the wind (NE), and the speed of the wind (12 MPH). There will be many of these boxes lined up horizontally, such as in *Figure 2.B,* and there

will be multiple rows of these boxes needed to display the 200+ SWAPR devices. Another type of view is the map view of the SWAPR network.

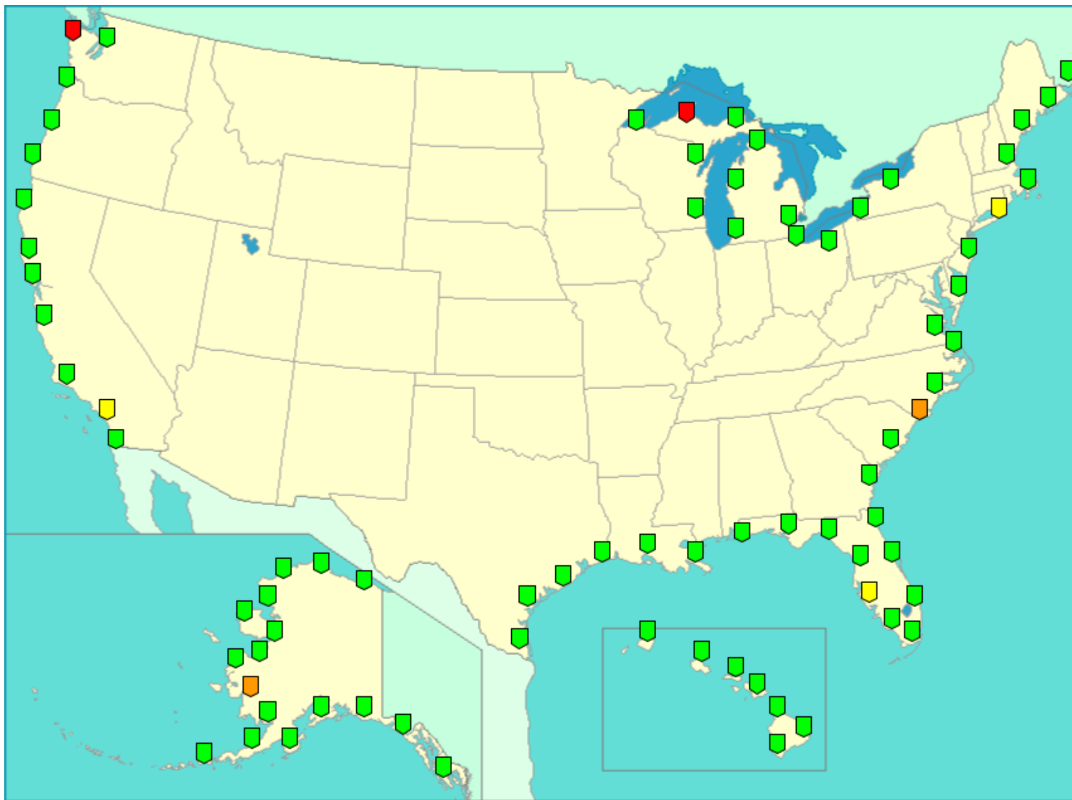

Figure 2.C An example of the map generated to display facilities with a SWAPR device. The green nodes represent fully functioning facilities while other colors suggest how the map can display issues.

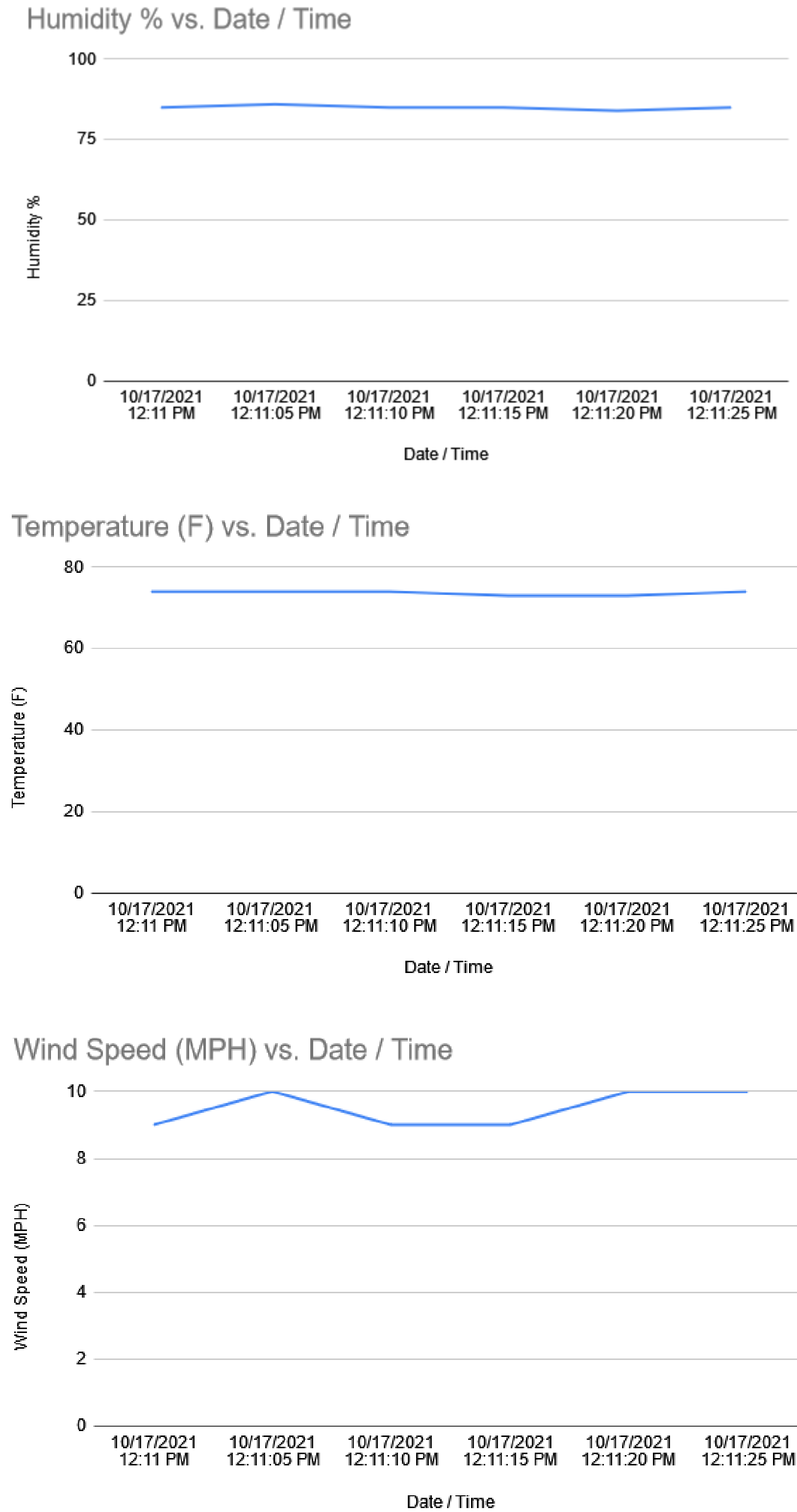

Figure 2.D Three examples of data visualizations of historical data in given ranges for specific types: Humidity, Temperature, and Wind Speed.

An example of this view is shown in *Figure 2.C.* Each of the marked nodes on the map would be an RFF with a SWAPR device installed. These points would be interactive, allowing the user to transition to our last type of view, which looks at an individual SWAPR device. We call this view the historical chart view. This view will show all the historical data of a SWAPR device in various graphical views. We will need the type of data that the user wants to display and the date ranges for said data to display graphs. Our MVP will include static images generated on the server side based on what the client has provided. If we have time, we would like to include an interactive graphical view where the user can make changes to the graph in real-time and see the changes on their end. *Figure 2.D* shows examples of various graphs that will display the variety of historical data that our SWAPR devices will generate. Not only will we have to generate graphs, but we will also need functionality that allows the user to export the data ranges used to generate the graphs.

If the user is an administrator, they will have a slightly different view when on the website. Admins will be able to view everything that normal users can, except they will also be able to view notifications when special events or loss of functionality happens. These notifications are just simple pop-up windows that tell the administrator what SWAPR device the notification is about and the reason for the notification.

The last thing that our client has requested we create is a way to simulate the SWAPR device for stress testing in a lab environment. The client wants us to create a program that will create the output that a SWAPR device would generate and send it over the RS-232 protocol. They also want us to create reader software that will sit on a computer at the RFF site, read the data coming in over the RS-232 protocol, package said data, and send it to a remote server to be used by the website. We decided with our client that the remote server would be a database server that extracts the data sent from the reader software and creates a database object for the data. The website will then make calls to the database for its objects to use them to do the functionality requested above.

We have divided this project into four subsystems. Those subsystems are the simulator, reader, database, and website. The simulator will act as a SWAPR device randomly generating output and sending the output in a list over the RS-232 protocol to the Reader Subsystem. The Reader Subsystem will take in a list over the RS-232 protocol, evaluate the data, and send it over a TCP connection to the Database Subsystem. The simulator and reader software will encompass the stress-testing in a lab environment challenge. The database server will read data coming in over a TCP connection and build and store a database object representing the data that a SWAPR device recorded. This database is not a direct challenge given by the client, but it is an indirect challenge that will help solve the direct challenges. The website contains the remainder of the challenges that we will face in this project. Our website will take data from the database and use that data to solve the challenges we will face. The last subsystem is an extra subsystem used for lab environment stress testing, which is called the Orchestra Subsystem. It is called the Orchestra Subsystem because it will orchestrate any number of simulator and Reader Subsystems connected over virtual com ports. We will analyze all the subsystems in more detail.

3. Technological Analysis

3.1 Simulator Subsystem

We will start our analysis with the Simulator Subsystem shown below *in Figure 3.1.A*. The Simulator Subsystem will randomly generate valid and invalid SWAPR output data, sending the data in a list over the RS-232 protocol to the reader software. The randomized data will be as shown in *Figure 3.1.B.* That means that our Simulator Subsystem will have two main challenges to face. Those challenges are generating random numerical values and sending data over the RS-232 protocol. Should time permit, we will have an additional challenge, a simulator orchestra that will create any number of simulator reader pairs that communicate over a virtualized com port. We will now analyze the minimal viable product (MVP) challenges for the Simulator Subsystem.

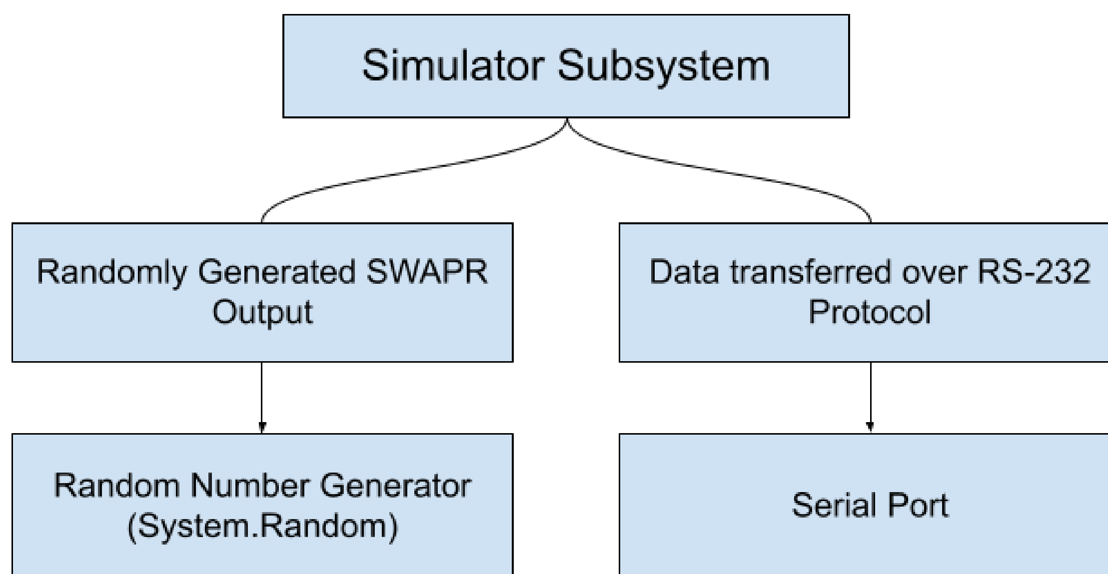

Figure 3.1.A This is an outline of the Simulator Subsystem's inner components for generating and transferring SWAPR output.

3.1 Desired Characteristics

We have three desired characteristics for this subsystem. We chose ease of Maintenance, Tech Maturity, and Documentation. We have ease of maintenance as one of our desired characteristics because General Dynamics Mission Systems will implement our system for the Coast Guard. We want to make sure that, if possible, the technology that we choose is one that they are familiar with so that our client has a smooth process for implementing our project into their architecture and maintaining it for the future. Tech maturity is one of our characteristics

because new technologies have not been tested enough to know all their flaws. Therefore, we want to choose technology that has been around for a while so that most of the bugs have been ironed out. That way, we do not unknowingly or unintentionally introduce any bugs or flaws into our system. Our last characteristic will be documentation because we want to make sure that both us and our client will understand how to use and implement the alternative into our systems.

3.1.1 Introduction

Our Simulator Subsystem will have two main challenges to face. One of those challenges is generating random numerical values. We will now analyze the minimal viable product (MVP) challenges for the Simulator Subsystem. The data we are randomizing will be as shown in Figure *3.1B.*

Rain output:	value from 0 to 1023	(1023 = no rain : 0 = Heavy rain)
Direction:	Output range from 0 to 360 deg	
Power levels range:	Range from -105 dBm to -85 dBm	
Humidity sensor levels:	0 to 100%	
Temperature:	Range from -40 to 80 Celsius	(-40 to 176 Fahrenheit)
Wind Speed:	Range from 0 to 200 mph	

Figure 3.1.B A display of the format for the data which the reader software will analyze.

3.1.1 Alternatives

We are using C# for this project so we will be looking at the .NET random class. There are possible alternatives, but they don't make sense because they will require more work to implement and will act the same.

3.1.1 Analysis

We will start our analysis with the Simulator Subsystem. The Simulator Subsystem will randomly generate valid and invalid SWAPR output data, sending the data in a list over the RS-232 protocol to the reader software. The class we chose best valued our desired characteristics**.** It is integrated into the .NET framework, so it will be a class that has been around since the beginning of .NET. That means that the security is dependable since Microsoft has had over twenty years of commercial use and development to mature it. Since this class is part of the .NET framework, our customer has likely used the class before, which will make it easier for them to maintain. The class has some maturity to it since it has been around for over twenty years. There are no additional fees required for our client or us to use the class. The performance

is also going to be fine-tuned because of the maturity of the tech, and Microsoft documents the class very well. All of these reasons are why we chose the system.random class to solve the challenge of generating randomized output for a SWAPR.

3.1.1 Chosen Approach

We will start with randomizing numerical values first. Because we are using C# as our language for this project, the choice for randomly generating numerical values is easy. We will be using the system.random class, which is a class under the system namespace.

3.1.1 Proving Feasibility

We will prove the feasibility of the system.random class by creating a simple C# program that will randomly generate the output of a SWAPR device and display the data in a list to the console. By doing this program, we will be able to show that generating SWAPR output at random is possible, and therefore the system.random class would be able to solve this challenge.

3.1.2 Introduction

The main challenge is to send data over the RS-232. The RS-232 protocol is a standard interface between a Data Terminal Equipment (DTE) and Data Communications Equipment (DCE) device establishing serial binary data interchange. It was developed by the Electronic Industries Association (EIA) and Telecommunications Industry Association (TIA). The origin of this protocol comes from electromechanical typewriters and modems dating back to the 1960s. The Simulator Subsystem will take in a list over the RS-232 protocol, evaluate the data, and send the list to the Reader Subsystem.

3.1.2 Alternatives

There are alternatives for sending data rather than the System.IO.Ports.SerialPort (RS-232) protocol, but it would be inefficient to transmit data through any other medium with our existing architecture. It will integrate well with the existing SWAPR device code from last year's capstone project.

3.1.2 Analysis

We will be using a .NET class that will give all our desired characteristics. That means the documentation will be better since Microsoft has had over twenty years to develop and improve their tool. Also, being part of the .NET framework means that our client will more than likely have experience with the class making it easier to maintain. The tech has some maturity since it has been around for over twenty years, and Microsoft still maintains it. Again, since the class is a part of the .NET framework, we will not have additional licensing fees. The speed and performance of the class are promising because of the maturity and purpose of the technology. Microsoft has developed and enhanced this tool for high-performance programming to draw in customers for their product with years of updates to gradually maximize the benefits. As our team uses .NET we will be able to verify this characteristic ourselves. Lastly, the class is well documented, just as most of the classes are provided by the .NET framework. These reasons explain why we choose the System.IO.Ports.SerialPort class to solve the challenge of communicating over the RS-232 protocol.

3.1.2 Chosen Approach

We will be using the System.IO.Ports.SerialPort class, which is given to us by the .NET framework. We choose this class to solve this challenge because it is a class that has been around since the beginning of the .NET framework.

3.1.2 Proving Feasibility

We will prove the feasibility of the System.IO.Ports.SerialPort class by creating a C# program that takes in a list and sends the data over a virtual com port to another C# program that just prints the list to the console. By creating this program, we will prove that communication over a virtual com port using the System.IO.Ports.SerialPort class is possible, and therefore the class can be used to solve the challenge.

3.2 Reader Subsystem

The next subsystem that we will be analyzing is the reader software. The reader software will be responsible for reading data over a virtual com port on the RS-232 protocol coming from the Simulator Subsystem. The reader software will then package the data received over the RS-232 protocol into a TCP packet and send it over a TCP connection with the Database Subsystem. Therefore, we will have two main challenges in this subsystem, and one of them was already solved in the Simulator Subsystem, as seen in *Figure 3.2.A.*

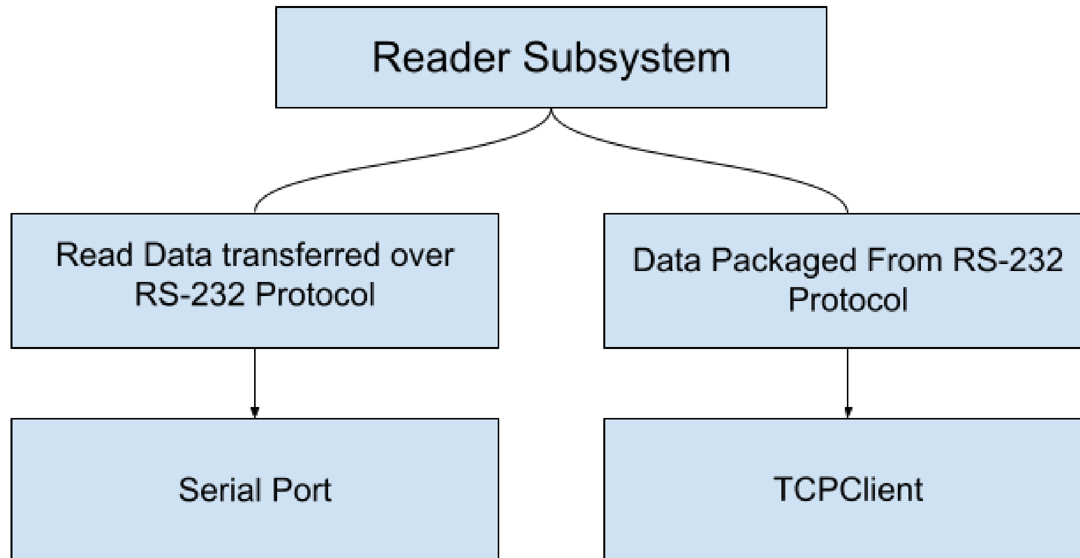

Figure 3.2.A This is an outline of the Reader Subsystem's inner components for generating and transferring SWAPR output

3.2 Desired Characteristics

We will be evaluating this subsystem based on four different characteristics. Those characteristics are security, ease of maintenance, tech maturity, and documentation. Security is important for this subsystem because this subsystem will have a TCP connection established with our Database Subsystem. Therefore, we are vulnerable to outside actors because we will be connected to the internet. For that reason, we want to make sure that there are not any security vulnerabilities that can be taken advantage of. Ease of maintenance refers to how easy it will be for our client to build, integrate, and maintain our project into their systems. Therefore, ease of maintenance refers to how well our client knows the alternatives that we are looking at and the compatibility of the alternative with their system. Tech maturity refers to the age of the alternative that we are looking at. We want our alternatives to have some maturity so that the maintainers of the alternative have had time to detect and fix any bugs or issues with their alternative. Lastly, we will be evaluating our alternatives based on the documentation of the alternatives. We want to make sure that we can easily find and learn how to use and implement the alternatives into our project. The best alternative will have all of these characteristics met.

3.2.1 Introduction

That challenge is reading data over the RS-232 protocol. We solved this problem in the Simulator Subsystem by using the System.IO.Ports.SerialPort class. The only difference this time is that we will be reading data instead of sending data.

3.2.1 Alternatives

Luckily for us, the System.IO.Ports.SerialPort class can read and send data through the RS-232 protocol. Though there are alternatives for data reading, our client requested that we use the RS-232 protocol.

3.2.1 Analysis

We have already done an analysis on why we picked this class which can be found in the Simulator Subsystem.

3.2.1 Chosen Approach

Using RS-232 is the most logical solution for us due to client request and smoother integration with the SWAPR Arduino device created by last year's capstone group. That device uses RS-232 protocol for transmitting serial data, and our team doing the same (though limited to virtual transmission and emulation) would prevent having to refactor a significant portion of code completed prior. No alternative solution would offer any advantage to us.

3.2.1 Proving Feasibility

We will be proving the feasibility of the System.IO.Ports.SerialPort class for the challenge of reading data over a virtual com port through the RS-232 protocol by creating a simple C# program that will connect to the proof of feasibility for the Simulator Subsystem, read the data sent, and print it to the console. This will prove that we can send and receive data over the RS-232 protocol using the System.IO.Ports.SerialPort class, which will prove that the class will solve our challenge.

3.2.2 Introduction

The final challenge will be establishing and sending data over a TCP connection with the Database Subsystem. We choose a TCP connection because we want to be guaranteed a response from every SWAPR device each time that the device sends out data. This is because if we are not getting responses from the device, then we will assume that the device is not working properly and that a technician needs to be sent to the site. We do not want to incorrectly conclude that a site is not working when the packet was actually just lost or corrupted. Therefore, we choose the TCP connection for the guarantees that we are provided.

3.2.2 Alternatives

We found another .NET class that will solve this challenge for us. The class we found to solve the challenge is called TCPClient and it is from the System.Net.Sockets namespace.

3.2.2 Analysis

We choose this class because it is another .NET class that has been around since the beginning of .NET. Therefore, our security and tech maturity characteristics are covered since Microsoft has had over twenty years to improve and remove all bugs from the class. Also, because the class comes directly from .NET, that means that our client will probably be familiar with the class, making for easier maintenance, and there will be no additional licensing fees. The class also has extensive documentation since it is from Microsoft

3.2.2 Chosen Approach

For all of these reasons, we decided to use the System.Net.Sockets.TCPClient class to solve this challenge.

3.2.2 Proving Feasibility

We will be proving the feasibility of the System.Net.Sockets.TCPClient class by creating a simple C# program that will establish a connection to the simple C# program to prove the database challenge. The program will then send some data that will be printed to the console on the other side. This will prove that System.Net.Sockets.TCPClient will be able to send data over a TCP connection, and the data will be properly received on the other side. If that is done, then we will have proved that the System.Net.Sockets.TCPClient class can solve this challenge.

3.3 Orchestra Subsystem

Our architecture thus far has covered the technology needed to simulate our system's expected environment and the challenges of transmitting data between the SWAPR device and reader software. Now we consider the subsystem which will allow us to construct instances of the communication between an individual SWAPR device and a receiving end. The following are characteristics we see value in for this subsystem and its challenges.

3.3 Desired Characteristics

The envisioned solution to this challenge is a tool capable of easily reproducing the simulation of a single device's communication with the reader software—a technology with established effectiveness and development history. The technology must be mature enough for us to be confident in its usability and reliability. For this, we must also consider a technology that is maintained and supported by a reliable developer. We must also seek a tool with documentation capable of providing examples of and references to this system's functions. We have chosen ease of maintenance, tech maturity, and documentation as our three desired characteristics.

3.3.1 Introduction

The challenge we face is developing an orchestra system to provide multiple instances of a simulated SWAPR device. Our system referred to the Simulator Subsystems' need for emulating randomness in data output and device communication defined with the RS-232 protocol as the interface. Now we are considering the need for a method of virtually simulating this case in multiple instances. You may consider up till now the idea of our simulator as a simulation of a single SWAPR device communicating with our server. This challenge considers the difficulty in creating this single action of one device and multiplies this to the size of tens or hundreds of devices all communicating with our server. This can be useful in stress testing our system in a more complex and scalable environment.

3.3.1 Alternatives

Though it is possible there are available alternatives to using the Null-modem emulator(com0com) to simulate SWAPR devices, our client has specifically requested for us to use com0com to virtualize serial ports on Windows. Our client went into detail on the functionality of the null-modem emulator and explained how it can utilize our Simulator Subsystem. This information was introduced in Section 3.1.

3.3.1 Analysis

The Null-modem emulator(com0com) is an open-source project created by Vyacheslav Frolov and S Hatchett in 2005 and has a most recent update from April 2018. This technology will allow us to create an unbounded number of virtual COM port pairs for communication between two COM port-based applications with one applications port as input and the other as applications port as output. com0com will feature the ability to create any COMx ports and multiple null-modem schemes to establish a simple emulator. This satisfies our desired characteristic of being able to simulate communication between a device and reader software.

We desired a technology that was maintained and supported by a reliable developer. The last desired characteristic which our approach fills is documentation. There are links on the main download page on SourceForge to a user manual as well as a readme and forums that explain how to use the Null-modem emulator.

3.3.1 Chosen Approach

Our chosen approach for the challenge of developing an orchestra system is to use the com0com tool for emulating COM port transmission. As noted previously, this choice to use the com0com protocol was made by our client, and for a good reason. Our chosen approach has satisfied all of our desired characteristics shown in Analysis 3.3.1: ease of maintenance, tech maturity, and documentation.

3.3.1 Proving Feasibility

To prove the feasibility of the Null-modem emulator, we will create a demo using the user manuals and reading the documentation on SourceForge.com. We will install the com0com tool and create virtual ports that will emulate serial com ports using RS-232 protocol from our simulated SWAPR devices. We do not have access to the physical Arduino SWAPR device; to work around this, using virtual ports will allow us to emulate the function and data transmission sent and received from our devices. Once we are able to connect to one device, we will create hundreds of COMx ports to stress test our system, which will need to support at least 260 of these simulations.

3.4 Database Subsystem

We will now analyze the Database Subsystem. Our Database Subsystem will be responsible for maintaining and listening to TCP connections with all of the reader software that is on the RFF sites. The database will take the data and extract the important information into a database entry. The database will also be responsible for taking queries from the website and serving the requested data to the website. Therefore, there are two challenges that the Database Subsystem will have to deal with. The first challenge is connecting and listening to the 260+ Reader Subsystems that will be at the RFF sites over a TCP connection. The second challenge will be dealing with the data received and turning them into database entries. The challenge is more with dealing with the large amount of data that the database will receive because it will be receiving data from each SWAPR device every 5 seconds. We want to make sure that we can receive all that data without losing any of it and make the database entries from each packet received.

3.4 Desired Characteristics

This subsystem is focused on six characteristics: security, ease of maintenance, tech maturity, licensing fees, speed and performance, and documentation. The Database Subsystem's purpose is for storing all important and private data retrieved from the SWAPR devices. Because of this, security is a necessity in ensuring the transmission of data to the database and data access is controlled and secure. This subsystem will need to be non-challenging in development and maintenance to avoid impacting our team's productivity. For this reason, ease of maintenance is a consideration in selecting our tools. Additionally to maintainability, we are considering the development of our tools for the database. That is: we want our technology to be mature. The technology we select must be clearly supported and developed consistently in order for our team to consider the technology to be an adequate tool to implement in our Database Subsystem. An inconvenience to our team and our client is technology costs. Any license fee or subscription we may run into for a viable product may be feasible. However, with a limited budget, we prefer to avoid unnecessary expenses. The nature of the Database Subsystem is data creation, access, deletion, and manipulation. For any actions done in our subsystem, we need responsive and accurate technology. The speed and performance of the database is a characteristic for verifying its efficiency. Finally, clear documentation and examples for technologies we implement will give our team helpful explanations and explicit cases which we can use better to understand the technology and aid in our productivity.

3.4.1 Introduction

We will first look at the challenge where our database needs to build and listen to a TCP connection for data being sent from the Reader Subsystem.

3.4.1 Alternatives

We found a .NET class for solving the challenge of sending data over a TCP connection, and there is a counterpart to this class called TCPListener, which is a part of the System.Net.Sockets.Socket namespace.

3.4.1 Analysis

Since the beginning, the TCPListener has been a part of the .NET framework, so we know that the security and tech maturity characteristics are met because Microsoft has had over twenty years to perfect the class. Since the class comes from .NET, we can also say that the maintenance characteristic is met because our client will have likely used the class before, and we can say the licensing fee characteristic is met because there will be no additional cost to use the class. Speed and performance are more than likely met because of Microsoft's amount of time to work on the class. The documentation characteristic is also met because the class has extensive documentation from Microsoft.

3.4.1 Chosen Approach

For all of these reasons, we have decided to use System.Net.Sockets.Socket.TCPListener class to solve this challenge.

3.4.1 Proving Feasibility

We will prove the feasibility of System.Net.Sockets.Socket.TCPListener by creating a simple C# program that is hosted by amazon web services (AWS) that will establish a connection with the proof of feasibility program that will be made for the Reader Subsystem, and the data received will be printed to the console. This will prove that the System.Net.Sockets.Socket.TCPListener class can build the TCP connection and listen for data over the connection

3.4.2 Introduction

The next challenge that we will look at involves receiving the large amounts of data from the SWAPRs and ensuring that we can process all the data without losing data.

3.4.2 Alternatives

This challenge will be solved by using AWS's Amazon Simple Queue Service (SQS), a technology that our client requested that we use to solve this challenge. Amazon Simple Queue Service will allow us to separate this subsystem into two parts one that will receive the data from the TCP connections and put them into a queue, and another that will process the data from the head of the queue and insert a database entry into the database which will be hosted on another AWS server. That way, we can separate the task of inputting data into the database and receiving data from the database for the website. That way, we can reduce stress on the database server and compartmentalize the subsystem so that failure will only break that one subpart and not the entire Database Subsystem.

3.4.2 Analysis

We are considering the Amazon Simple Queue Service because the security provided is top notch using Amazon SQS to transmit data from application to application with server-side encryption to encrypt the message. Plus, we can use AWS Key Management Service to have the applications talking to each other authenticate before accepting or sending any data. Since our client was the one to request that we use this software, we can assume that they are familiar with the service and they can maintain it after we turn the project over. The SQS service started being provided by amazon in late 2004, so we can say that the service meets our tech maturity

characteristics. Licensing fees will play into using this software; however, as a student, we get a $300 credit, so we will not have to pay anything to use the server to start. However, our client will have to consider the cost of using the server. Looking at the current cost of use ($.50 per 1 million requests/month) and with an estimated 1,639,872,000 messages that will be received and put into the queue in a year, we estimate the cost of using AWS SQS for one year will be $820. $820 per year is not a lot of money when considering the size of the Rescue21 system, so we will consider this licensing fee to be irrelevant. The speed and performance of AWS SQS have been proven by case studies done by EMS Driving Fuel IQ, NASA, BMW, CapitalOne, Change HealthCare, RedBus.in, and oyster hotel reviews. Because of these case studies shown on the amazon SQS page, we know that the speed and performance characteristics are met. Lastly, we were able to find great documentation that even includes best practices and tutorials. Not to mention that there are a number of guides on using the software.

3.4.2 Chosen Approach

All of these reasons explain why we have decided to use AWS SQS for solving our second challenge.

3.4.2 Proving Feasibility

We will prove the feasibility of using AWS SQS by spinning up an instance that will host the proof of feasibility that we made for the first challenge so that we can read data coming in from the TCP connections established with the Reader Subsystem. If we can read this data and put it into the queue, and receive it, we will have proved that AWS SQS will solve the challenge of not losing any data from the large amount of information that our SWAPR devices generate.

3.4.3 Introduction

Lastly, we will talk about how we will host the database server.

3.4.3 Alternatives

We are evaluating the MySQL database, which will be hosted on AWS Relational Database Service (RDS). We will be doing this because our client has requested that we use a MySQL database and that we host it off of AWS.

3.4.3 Analysis

We are considering AWS RDS because the service was built specifically to host MySQL databases. AWS RDS was initially released in October of 2009, so we know that the service meets our tech maturity characteristic. The security for AWS RDS is great because it includes network isolation using Amazon Virtual Private Cloud, authentication between services using AWS Key Management Service, and the data is encrypted in transit using SSL. The maintenance for our client will be met since our client has requested that we host the database off AWS, suggesting that they are familiar with the platform. We will have to consider the licensing fee for our client; however, we will not have to pay anything because we get a $300 credit because we are students. With our previous estimates of requests being 1,639,872,000 messages a year, we will run up a data request cost of $163 estimate if we request for every message once. We will also have $.10 per GB / Month. Each SWAPR entry will be at most 46 bytes. That means that with 1,639,872,000 entries in a year, then we would be storing 75.5 GB's at the end of the year.

That would cost the client an additional $7.5-9.5 for the last month of the year. That would cost the client about $120 a year (overestimate) if we only kept one year of historical data. The last cost to consider is the server that we will host the database off. There are a large number of servers to pick from, but I suspect that we will need a server that will average a cost of $.25 per hour (it may be less depending on the needs and performance that we find). Therefore, if we ran the server in the on-demand server pool, then we could expect to spend roughly $2,200 on hosting the database server for the year. Therefore, the total estimate for hosting the database server for one year will be roughly $2500. This is a lot of money, but it isn't a lot of money for the size of the project that the Rescue21 system is. Plus, both of these costs can be mitigated should General Dynamics decide to host these instances on their own equipment. For those reasons, we will consider the licensing fees to be irrelevant. The speed and performance can be scaled with the server we host the database server on, so the characteristic has been met. Lastly, the documentation for AWS RDS is pretty solid. We were able to find tutorials for building an instance and documentation on AWS RDS's functionality.

3.4.3 Chosen Approach

We must consider that we will be doing our hosting on AWS, so we want to have all our hosting done off AWS. For that reason and all the reasons listed above, we picked AWS RDS for hosting our MySQL database server.

3.4.3 Proving Feasibility

For proving the feasibility of using AWS RDS for hosting our MySQL server, we will spin up a simple instance that we will connect with the simple instance of the AWS SQS. We will test that we can retrieve data from AWS SQS, build a database object, and store it in the MySQL database. We will then query the database for the simple Blazor server instance that we will make for the Website Subsystem. If we can do both of these tasks, we will have proved that we can use AWS RDS to host our MySQL database to solve the challenges we face in the Database Subsystem.

3.5 Website Subsystem

Last but not least, we will be analyzing the Website Subsystem. The Website Subsystem is the biggest subsystem in our project. The Website Subsystem will be responsible for two types of challenges: backend and frontend. Backend challenges will be challenges that the customer has no control of and has no idea how we solve the challenges. Backend functionality will be the computational functionality. Meaning the backend will be doing the work to make the frontend work. The frontend challenges will be functionality that the user sees and interacts with to change the way that the backend responds to a point. For this subsystem, we will first examine the framework that we will be using, then segue into the backend analysis, and we will finish the analysis of this subsystem by examining the frontend analysis.

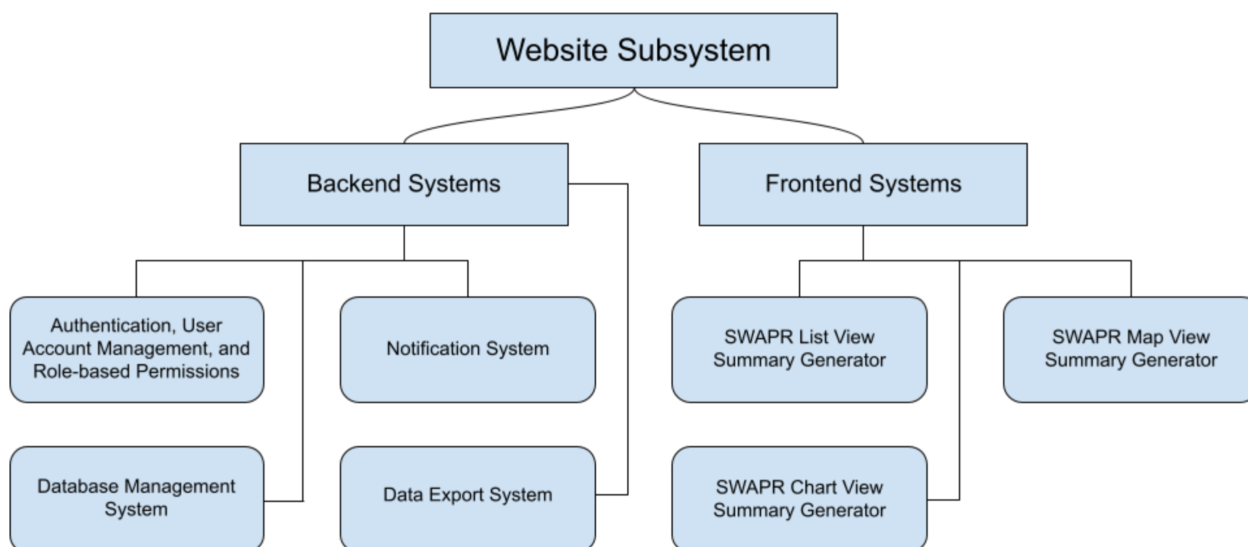

Figure 3.5.A This is an outline of the Website Subsystem's inner components for completing all of the challenges we face in the backend and frontend.

3.5 Desired Characteristics

Our desired characteristics for the Website Subsystem are security, ease of maintenance, tech maturity, licensing fees, speed, and performance. Security is important in the Website Subsystem because sensitive location information and the status of the Remote Fixed Facilities (RFFs) from our database are being accessed. Our Website Subsystem must be easy for users to access and add any functionality if necessary. We want to spare our client as much financial expense as possible, so we need a solution with minimal to free licensing fees. Performance is the most important concern for our website because the Coast Guard needs to be able to access information quickly and efficiently without significantly long wait times, which is crucial.

3.5.1 Backend

We will now discuss the backend challenges we will face in the Website Subsystem. There are four main challenges that we will face. We must have authentication, user management, and role-based permissions, which as an MVP, will be faked because we cannot easily get an Active Directory running without a hefty licensing fee. For that reason, our client has decided to let us fake user management and role-based permissions unless we have extra time and money still in the budget. Authentication will be handled by our hosting solution, which is why we do not consider it a challenge. Authentication will happen between the website and the database. We will still evaluate the alternatives that we have for Active Directories just in case we do end up setting up a real user management and role-based permission system. The next challenge involves our database queries. We will need the ability to securely access data from our Database Subsystem. The next challenge involves notifying the appropriate user accounts that there are problems occurring at any of the RFF sites or any of the SWAPR devices. The last challenge for the backend will be exporting data to a .csv file. When the user gets historical data from one SWAPR device, they will be given a choice of a date range (for the MVP), and we will generate charts to show that data. Our client has requested that we also provide a way to export that data to a .csv file. If we have additional time, we will also provide a more interactive chart generator that will be able to select what data a SWAPR generated to display, and the export functionality will need to only export the requested data. We will now examine each of these challenges more closely in the order that they were talked about.

3.5.1.1 Introduction

We need to build a website capable of handling live updates, as it will eventually take in weather and status information of SWAPR devices in real-time. This will effectively be a live data dashboard.

3.5.1.1 Alternatives

Our client has made it a requirement that the website is created using Blazor Server as our website framework. Blazor Server was released in 2018, but it was built on top of the .NET framework, so it uses code that has been around for over twenty years. Blazor Server is built on top of the .NET 5 framework, so we will have access to most of the tools provided by the .NET 5 framework. Blazor Server is the correct framework for us to use because Blazor Server will work well with updating the overview of our SWAPR dashboard. We are also going to be required to program in C# for this project because our client has made it a requirement and because Blazor Server uses C# as its language.

3.5.1.1 Analysis

It is difficult to say Blazor Server meets our requirement of tech maturity; however, our client, who is very security conscious, requested we use Blazor Server. Considering this, Security and Tech Maturity requirements are met. We also meet the ease of maintenance characteristic because our client requested we use Blazor Server, so they already know the framework. There will not be any additional licensing fees because Blazor Server is a part of the .NET framework, which we have access to through Visual Studios. The speed and performance of our Blazor Server will depend on the code that we write and the server that we host it from. Both are

dependent on our design decisions, so we can confirm Blazor Server meets the speed and performance metrics. Lastly, Blazor Server is well documented by Microsoft, and there are plenty of other internet resources on using Blazor Server.

3.5.1.1 Chosen Approach

For all of these reasons, we decided to use Blazor Server over other .NET frameworks like Blazor WebAssembly or the Model View Controller model.

3.5.1.1 Proving Feasibility

To prove feasibility for Blazor Server, we will create a simple Blazor Server app that will grab and display information retrieved from the database that we will eventually host. If we can grab data and display the data to the website, then we will know that we can make Blazor Server work just as long as the proof of feasibility for the other challenges we are about to go over also work.

3.5.1.2 Introduction

In our project, we are dealing with very sensitive location information in addition to even more sensitive account information for The Coast Guard and General Dynamics. We need to ensure that only authorized users are able to access SWAPR devices and their data. We need a secure authentication system that consists of two roles: user and administrator. Ideally, we would like a low-priced or free licensing cost. We want authentication that works well with Windows because that is what General Dynamics uses currently. We want a tried and tested system that has been around for a while.

3.5.1.2 Alternatives

There are a few approaches that we researched that may serve us well. The first one of these is Active Directory. It was developed and is currently maintained by Microsoft. It was originally released in 1999 and is commonly used in professional environments. Active Directory (AD) allows you to create and control users and roles with the backbone of Kerberos, LDAP, and NTLM security protocols proven to be effective and secure. It is by far the most supported and well-documented account management system. Our second alternative we looked into was Azure Active Directory (AzureAD). This was originally released in 2008 by Microsoft as Windows Azure before being called Microsoft AzureAD. It is a cloud-based account management system that is accessible by anyone for cheap licensing fees depending on the version. The free version is six dollars monthly. AzureAD has a managed firewall, backup, and multi-factor authentication. This is very similar to the use of Active Directory but is not used by General Dynamics, so it is not our first choice for role-based account creation. It is well documented, so it would be easy to integrate within a windows environment. An open-sourced approach we looked at is called JumpCloud. Rajat Bhargava and Larry Middle created it in 2008, who wanted to create an open-source alternative to Active Directory with Windows, Mac, and Linux support. Jumpcloud is entirely cloud-based, works well with Active Directory, and is well documented. For security, JumpCloud uses Cloud LDAP protocol

3.5.1.2 Analysis

In order to narrow down which system we wanted to use for our Website Subsystem, we analyzed AzureAD, JumpCloud, and Active Directory-based on our desired characteristics of security, ease of maintenance, tech maturity, licensing fees, speed, and performance. Jumpcloud claims to be secure, as it is completely cloud-based and uses Cloud LDAP protocol for encryption. It has been around since 2008, so it has reasonable tech maturity. JumpCloud is open-source, so there are no licensing requirements. Speed is dependent on network connection speed. This software is usable on Mac, Linux, and Windows, so it has high performance. AzureAD is cloud-based, and for security, it uses Azure Security Center. If used in conjunction with Active Directory, more security options could be enabled. Though our client already has experience with Active Directory and would be able to easily maintain AzureAD, they currently do not use AzureAD, so it would be an inconvenience to use this. Licensing is six dollars a month for a membership or available for free with a Microsoft 365 account. Performance is high since it integrates well with Active Directory and Windows products, but it would have to be paired with AD and not used on its own for proper security. For Active Directory, The security is by far the best out of the alternatives that we found. The software has been around for over twenty years, so it meets our tech maturity characteristic. The software does have a licensing fee should we get time and money left to implement a real Active Directory system. The cost would be the price of a windows pro edition key which is around $130. The speed and performance of Active Directory has been perfected over the last twenty years, so we will say that we have met this characteristic. Lastly, the documentation is pretty good for Active Directory, so we can confirm that this characteristic is met.

3.5.1.2 Chosen Approach

We decided to choose Active Directory to solve this challenge because it was the software that our client uses, so it meets our maintenance requirement. For all of these reasons, we have decided to pick Active Directory over the alternative options.

3.5.1.2 Proving Feasibility

For proving feasibility, we will first build our MVP for the challenge, which will involve creating a database table for users, account type, and usernames and credentials. We will also need a simple login page that takes in a username and credentials. If we can get our simple Blazor Server app to compare the credentials and if they are true, then bring the user to a new page that shows them their account type, then we will know that we can fake Active Directory functionality. Of course, if the credentials are wrong, then we will need to prompt the user to try again because the account wasn't found. If we get the time and have the money left, then we will prove feasibility with the same proof of concept, but we will use Active Directory instead of a database table.

3.5.1.3 Introduction

The next challenge that we will discuss involves database queries. We will need two types of database queries. One query for the latest entry from all the SWAPR devices in the network for use by the summary GUI's. The other query will be for retrieving data from one SWAPR device over some date range and the set or subset of data that a SWAPR generates.

3.5.1.3 Alternatives

The functionality for both of these can be done using a .NET class. We can solve the challenge using the System.Data.SQLClient namespace courtesy of the ADO.NET framework. More specifically the SQLDataAdapter and SQLDataReader classes are used for retrieving data from a stream. Due to the nature of this subsystem and this challenge there is no alternative approach to accessing the database. These classes are built into .NET guaranteeing no financial cost as well as compatibility with Blazor and our website. This functionality has been a component of .NET since .NETs initial release. The DataAdapter features three useful methods: insert, delete, and update. The DataAdapter is used for retrieving data from a data source and updating SQL server databases. What this means is our project will be able to load all the desired data from the database, store it in an in-memory cache in a DataSet, and then close the connection, saving on performance. This efficient option will keep our website agile and efficient. The DataSet object is a collection of DataTable objects otherwise considered a table itself. By accessing the data in this form, as a DataSet, data can be read and stored easily also with the option of being written as an XML document. This simplifies transportation of data over the website.

3.5.1.3 Analysis

Previously stated in our Alternatives section 3.5.1.3, the System.Data.SQLClient namespace within the ADO.NET framework already provides two classes that will allow us to handle this challenge. SQLDataAdapter and SQLDataReader have no viable competition for us to consider because .NET and its internal components are to be used at our client's request. Since the .NET framework has been around for over twenty years, this meets our security and tech maturity needs, as this is ample time for Microsoft to iron out security flaws. Our client is familiar with the .NET framework, so it is easily maintainable. There are no licensing fees for this alternative.

3.5.1.3 Chosen Approach

We plan on using DataReader for accessing the latest database entries from each SWAPR device because DataReader is the faster option, and we typically only iterate over it once, which better fits the use case. We plan on using DataAdapter and DataSet to grab the data over a certain date range and set of SWAPR data output because we will need all of the data before moving on, which is what this class does. It also stores the data in memory, making it easier for us to create charts and export the data. For all of these reasons, we have decided to use the system.Data.SQLClient namespace to solve this challenge.

3.5.1.3 Proving Feasibility

To prove feasibility, we will create a simple Blazor App that will query the database using both the DataReader and DataAdapter classes to query for their relevant data (latest entry / historical data) and display this on the webpage. If we can successfully do this, then we will know that the system.Data.SQLClient will work to solve this

3.5.1.4 Introduction

The next challenge that we will analyze is notifying the appropriate users that there is a problem with a particular RFF or SWAPR device. We will do this by evaluating the data that we

get from the SWAPR devices, and should any of the data received be outside of the acceptable data ranges, then we know that there is a problem with the SWAPR (if weather data is invalid) or with the RFF antennas (if the transmission power is invalid). If we detect this to happen (using simple if-else or switch statements), then we will need to queue notifications to be displayed to the next admin user that signs in, and only when an admin clears a notification does it stop displaying at sign-in. This can be done using simple C# data structures, so that is not the challenge. The challenge is displaying the notifications. There is really only one good solution to the problem, which involves using a NuGet package.

3.5.1.4 Alternatives

A NuGet package is a package that has been made for .NET applications. The NuGet package we chose was specifically made to be used by Blazor products which is why it is the clear-cut solution. The NuGet package we will be using is called Blazored.Modal. A model is a notification pop-up.

3.5.1.4 Analysis

The NuGet package, Blazored.Modal was released in early 2019, so it is relatively new, but over four hundred thousand people have used it, and so the application has had some time to mature. This package will be easier for our client to implement because Microsoft already approves a NuGet package. However, it isn't as likely that they have used the package. There isn't a licensing fee associated with the package, meaning we meet the licensing fee characteristic. The speed and performance of the package looks promising because the package is frequently referenced in user forums asking about notification pop-ups. There is documentation on how to use the package, but it isn't nearly as comprehensive as Microsoft documentation, but it will be enough for us to get the package to work.

3.5.1.4 Chosen Approach

For all of these reasons, we decided to use Blazored.Model to solve this challenge.

3.5.1.4 Proving Feasibility

We will prove feasibility by creating a simple Blazor server app that will create a pop-up on the website page. We will keep showing the pop-up until the user clicks a button that says "Clear Notification" instead of "Close Pop-up". If we can do this, then we will have proved that Blazored.Model will work to solve the problem of showing pop-ups.

3.5.1.5 Introduction

The last challenge that we must analyze for the backend challenges involves exporting data to a .csv file.

3.5.1.5 Alternatives

We have looked at a number of NuGet packages that can solve this problem; however, none of them really were what we wanted. For that reason and because it isn't terribly hard to create comma-separated files, we have decided to create our own functionality to create .csv files. In order to do this, we will need a way to create and edit files which is functionality that is provided by the System and System.IO namespaces.

3.5.1.5 Analysis

These namespaces have been around since the beginning of the .NET framework, so Microsoft has had over twenty years to ensure that the classes have proper security and speed, and performance. Therefore, the namespaces meet our security, tech maturity, and speed, and performance characteristics. Since System and System.IO is such an important namespace to .NET, we can also assume that our client knows and has used the namespaces, so it meets the maintenance characteristic. There aren't any licensing fees because we will be making the functionality ourselves. There is ample documentation on the System and System.IO namespaces too.

3.5.1.5 Chosen Approach

For all of these reasons, we have decided to use the System and System.IO namespaces for solving this challenge.

3.5.1.5 Proving Feasibility

For proving feasibility, we will create a simple Blazor server application that will create a file, put some data in it, and display it on the website. If we can do this, then we will know that we can solve the challenge of exporting data to a file, and all we will have to do is name the file .csv and follow the format of a .csv file.

3.5.2 Frontend

We will now analyze the frontend challenges that we will face while building the Website Subsystem. There will be three challenges that we need to overcome. The first challenge will be creating a list summary view of all the SWAPR devices connected to the system. The list view will show the latest data from each SWAPR device. Each SWAPR device will show its name, temperature, wind direction, and wind speed. The next challenge will be another summary view, but it will be on a map. The map will have a bunch of color-coded markers that show the location of the SWAPR device. The color of the marks will be green, orange, yellow, or red. Green means the site is working as expected. Orange means there is a problem with the antennas on the RFF site. Yellow means there is a problem with the SWAPR equipment. Red means that the site is not responding at all and that a technician will need to be sent to the site to figure out what is going on. Each marker on the map will be clickable, and it will take you to that SWAPR device's historical information page. The historical information page is the last challenge we face in the frontend challenges. The historical page will at first only take in a date range which will be used to retrieve the data for that SWAPR, generate static graphics to display that information, and that will be returned to the user. This page will also be where you can export the data to a .csv file. If we have additional time, then we would like to make the graphics generator interactive where the user can select the set or subset of data values that a SWAPR generates, and they can also select the type of graph that is generated to display said data.

Our project is concerned with two forms of viewing our SWAPR devices and each device's data. This challenge covers determining adequate technology to display the devices and their data in a list view of summary objects. For a visual example of this concept, see *Figure 2.B*. We need a tool with functions capable of creating simple 2D geometry, color, and dynamically changing values such as integers or floats.

3.5.2.1 Introduction

We are looking for an efficient method of creating graphics for our listview within a web page. Preferably a system with no financial expense, fast and clean rendering to create a user-friendly and pleasant display, proper documentation to make learning and testing the tool easier, and established development history to give our team confidence in the functionality of the product and compatibility with .NET/Blazor. These graphics will likely be created in real-time versus being created and stored beforehand. Because these graphics will need to be generated as they are invoked, our tool will need to be as optimized and as fast as possible with as little memory cost as possible. Potentially hundreds of SWAPR devices will be listed in an instance and will need to be visible without impacting the website or users' local machine performance-wise.

3.5.2.1 Alternatives

In researching the available avenues, we found two paths: Windows products and JavaScript-based packages that were potentially capable of meeting our needs. The first product is Windows Forms Designer (WFD). This was discovered by searching for .NET-related tools capable of producing graphics in 2D. WFD is used for creating Windows-based applications that can utilize the feature-rich Windows operating system. The WFD toolkit is accessible via the

.NET Core as a UI framework in Windows, meaning it is built into the primary system we are using, is a product of Microsoft, and is capable of doing exactly what we are looking for, creating UI on our website. WFD became available with the .NET Core in 2019.

The second Windows product discovered is Windows Presentation Foundation (WPF). Another product of Microsoft, WPF, provides an extensive toolkit of application development features. WPF is used to create vector-based graphics in both 2D and 3D, animating graphics, and more. It can be used to develop and design Windows applications and web applications with extensive customizability, potentially greater than WFD. It is now available with the latest version of .NET, .NET 5, but has been in development with full releases since .NET Core 3.0 with over 3,000 GitHub commits to its repository by almost 100 contributors. WPF provides an open-source GitHub repository, but WPF is still a product of Microsoft with all of the support and documentation.

Windows products, however, are not our only alternatives. Knowing that JavaScript can be used to render graphics in a browser, research was specifically done to find alternatives that used JavaScript to make comparisons to Windows toolkits. There is the Blazor extension, Canvas. Canvas is an API that can be integrated with Blazor to create web-based graphics with either Canvas 2D or WebGL (Web Graphics Library). Currently, in version 1.1.1, Canvas extends the capabilities of Blazor by adding HTML5 APIs, new classes, and methods in order to give Blazor the power to create JavaScript-based graphics. The JavaScript will be injected into C# using JavaScript interoperability referred to as JS interop for short. This only refers to calling JavaScript functions from .NET methods but is a specific feature of ASP.NET, a framework used by Blazor for web applications. Although the Blazor extension Canvas is integratable with Blazor, it appears to be purely a community-developed tool based on the GitHub Repository. The components of this tool are products of Mozilla and Khronos. Khronos Group is the developer of OpenGL and other popular and powerful graphics rendering frameworks. This suggests the potential for a powerful tool.

Finally, D3.js, another tool that utilizes JavaScript to create web graphics. D3.js is also an open-source tool available on GitHub and also provides an official site. In version 7.1.1, with extensive development history, D3.js is used to create web-based visuals with SVG, Canvas 2D, and HTML. It uses DOM manipulation to produce graphic interfaces with interactive capabilities. D3.js uses JS interop to inject JavaScript calls into .NET C# to invoke JavaScript functionality for Blazor web applications. D3.js was appealing because of the impressive examples found for creating wind compasses and other types of graphics. It appears to be more than capable of creating static and dynamic graphics for the summary of SWAPR devices.

3.5.2.1 Analysis

We will start by evaluating D3.js, which can be implemented using the ASP.NET framework. After importing the D3.js package, integration of D3.js only requires .NETs JS interop feature to object JavaScript actions into Blazor to create desired functionality. D3.js has a large development community with recent and frequent commits in their GitHub repository. This shows investment by its developers and its community. On the topic of its community, D3.js has over 200 thousand users and over 100 open-source contributors. Their official site, separate from their GitHub, looks professional and up-to-date. The performance of D3.js is also promising. The only negative is warnings from performance testing conducted in blogs online stating that very

large datasets may begin to create lag. This is definitely something we will need to stress test to find the bounds to which our site can effectively generate graphics. D3.js also supports multiple file formats: vector graphics, HTML-based graphics, as well as Mozilla's Canvas API if necessary. A very extensive and well-maintained product with a strong and active community. Not only do the sites look pleasant and display the involvement of their community and developers, but their documentation and examples are extensive. Whether it is on their official D3.js site, GitHub, or other forums and blogs, the D3.js features are all thoroughly covered with fairly concise information and examples.

The aforementioned characteristics for D3.js are what make it stand out in comparison to the other alternatives. Another proponent to this decision is D3.js' cross usage for other challenges. This may potentially be used in other components of our subsystem. The alternatives, such as the two Windows Products, pose limitations and steeper learning curves. In the case of WPF, while it may be advertised as a powerful tool and a product of Microsoft, it is also said to be difficult to pick up and use due to the structure of all of its functionality. WFD, on the other hand, appears to be outdone by WPF because of its capabilities. Having only functionality with Windows applications poses an issue in the case we want only browser-based features. Finally, the Canvas extension for JavaScript lacks strong developer support and has a small enough community and development history that it appears too new and underdeveloped. With our project to consider, a technology that has only been developed for a year looks less appealing compared to a technology that has been in development for almost ten years currently in version 7.1.1.

3.5.2.1 Chosen Approach

By researching information for each tool based on these criteria, we were able to create a table overviewing all of this information. This chart can now be evaluated to establish ratings and reach a conclusion.

Graphical List SWAPR Summary	Speed (10 total)	Documentation (8 total)	Ease of Maintenance (5 total)	Tech Maturity (5 total)	Fees (2 total)	**Total (out of 30)**
Windows Forms Designer	4	6	3	5	2	**20**
Windows Presentations Foundations	6	8	5	5	2	**26**
Blazor.Extensions.Canvas + HTML Canvas (+JavaScript)	10	3	2	2	2	**19**
Blazor with JavaScript package D3.js	**10**	**8**	**5**	**5**	**2**	**30**

Table 3.5.2.1.A This is a table overviewing the final scoring for the Graphical List View of SWAPR summary data.

Using D3.js is the most viable option for this project. It was chosen based on our desired characteristics *Table 3.5.2.1.A* . It scored the highest out of all the current options; it excelled in documentation, rating, and maintenance. The benefits of D3.js far outweigh any of the downsides. D3.js was appealing because of the impressive examples found for creating wind compasses and other types of graphics. It appears to be more than capable of creating static and dynamic graphics for the summary of SWAPR devices.

3.5.2.1 Proving Feasibility

A quick and concise way to test this tool will be to create a simplified version of the graphic, viewable in the browser, to verify the general concept can be built. See *Figure 2.A* or *2.B* for a mockup of the widget. We will create a similar, likely less detailed version of this and see that when values are updated, the image properly updates.

3.5.2.2 Introduction

We will now analyze the map summary view challenge. This challenge will have us put markers on a map of the United States that are color-coded where the color represents the type of issue, if any, that is occurring with the RFF or SWAPR device. These markers will also be clickable, and they will take you to the historical SWAPR data summary page. We will be making a static map for our MVP that will just be a picture of the United States with clickable objects drawn on top of the map. We would like to have an interactive map should time permit, but this would be a stretch goal.

3.5.2.2 Alternatives

We have found two ways to solve the MVP challenge that we face. The first way that we could solve the challenge is by using the R language with tmap or ggplot2 with +sf, raster, dplyr, spData, and spDataLarge packages to support tmap and/or ggplot2. The second way that we could solve the challenge is with an image of the United States and with the HTML Map Tag along with the Area Tag and the color attribute.

We will first look at using the R language with the tmap or ggplot2 packages. R was created in 1995, and Tmap was released in Nov of 2013 and is still actively updated. GGPlot2 was originally released in 2005 and is still actively updated. Tmap is more used for creating maps, while ggplot2 is more used for creating graphics. Tmap closely resembles the syntax of ggplot2, which will be more important later.

We will now look at the HTML Map Tag, Area Tag, and Color attribute alternative. This alternative is very basic, and it is not a solution that can be easily changed for new data. However, it will be able to overcome our challenge even if we have to have a clunky solution.

3.5.2.2 Analysis

For solving this challenge, we will look more at Tmap. Tmap will be great for creating maps. Tmap can also have clickable objects on the map, which is exactly what we need. Tmap security isn't so important because it will generate a static map that has clickable links; however, the security flaws will have, more than likely, been ironed out since they have had 17 years to detect and fix them. Our client is not likely to have worked in R, so, unfortunately, this alternative doesn't meet our maintenance characteristics. However, we will be trying to use a

package that we found which will convert C# code into R, which will make it easier for our client to maintain. Should this workaround not work, then we will just try using the R language instead. It does however meet our tech maturity characteristic because of the 17 years that the project has been around. There are not any licensing fees which mean we also meet that characteristic. The speed and performance for Tmap is great when using it for smaller maps like we will be using it for, which means we also meet that characteristic. Lastly, the documentation for the package is ok, but the internet tutorials make up for any shortcomings.

We will now look at the HTML Map Tag, Area Tag, and Color attribute alternative. The security of HTML is met because of the rigorous testing that happens before new functionality can be added to HTML. Our client has most definitely used HTML, so we know that they could maintain this alternative. Also, the tech has had time to mature because of the age of HTML. There are no licensing fees for using HTML too. The speed and performance is great because it is just like rendering any other HTML document. Lastly, there is plenty of documentation on using these tags and even more internet resources.

3.5.2.2 Chosen Approach

Both of these alternatives are good choices; however, we definitely like one approach more than the other. We would have to pick the tmap package using R as the winner because of the increased flexibility that we get from the package. However, if we cannot use tmap for whatever reason, then we have the HTML alternative to fall back on.

3.5.2.2 Proving Feasibility

We will prove the feasibility of tmap by making a basic Blazor server app and loading it in the package. We will try to use a module provided by Microsoft that allows us to convert C# code to R code, but if we cannot get that to work, then we will just use R. If we can load the package, create a map of the United States, and create one clickable object that takes us to another URL, then we will have proved that we can use tmap to solve our map summary view challenge.

3.5.2.3 Introduction

We will now look at the challenge for our graph view of historical SWAPR data. We will need to take in data ranges, lists of values, and generate images to create graphs, plots, and other types of displays for quantitative data. This will be useful for users to see summaries of historical data from a SWAPR device in a convenient and organized format.

3.5.2.3 Alternatives

Our first tool considered is the R programming language, specifically the implementation of R using R Shiny aka Shiny. The R programming language is a popular language for computing statistical data and generating visualizations. The Shiny app is capable of defining objects in R and providing a clean application of these visualizations into our web pages without any web development tools. .NET itself covers the integration of the R programming language into C#. This appealed to us because the R programming language is valued for its computing power and purpose as a statistical computing and visualizing tool. Having a tool with support from our primary library gives reason to consider R Shiny. This tool is a package for R

developed by an institute, Swarthmore College, in Pennsylvania. First released in 2012, it is available officially via the RStudio site as well as in a GitHub repository.

As our other alternative, a JavaScript approach may be implemented. As previously mentioned, JavaScript is integratable with .NET using ASP.NET. Using JS interop, the implementation of JavaScript in our web pages will be possible. D3.js can be used in this case to create the visualizations we desire. D3.js has been in development for about ten years and is currently in version 7.1.1 with documentation on both an official site as well as a GitHub Repository. To produce the graphs and plots specifically, there are a number of packages that can be imported to achieve this. Importation of packages in JavaScript is a straightforward process but will require multiple dependencies. We will later discuss in more detail these options

3.5.2.3 Analysis

In our evaluation considering JavaScript and R programming language approaches. In terms of ease of maintenance, the R programming language route utilizes tools built for .NET integration and is supported by established developers. Consistency with our tools reduces dependencies our team will need to consider. Both tools are to be implemented in our system, and for this reason, does not propose any complication. To give ourselves an overview of these characteristics and to provide clear comparisons between the two avenues, we created a comparison table for the alternatives. See *Table 3.5.2.3.A*.

3.5.2.3 Chosen Approach

Historical view for summary data	Speed (10 total)	Documentation (8 total)	Ease of Maintenance (5 total	Tech Maturity (5 total)	Fees (2 total)	**Total (out of 30)**
JavaScript (and accompanying packages)	7	8	4	5	2	**26**
R programming language with R Shiny (and accompanying packages)	**10**	**8**	**5**	**5**	**2**	**30**

Figure 3.5.2.3.A This is a table overviewing the final scoring for generating graphics of SWAPR device historical summary data.

Although we plan to use D3.js elsewhere in this subsystem, we find the R route to be more appealing as it stands and considering we plan on using R for map creation, we still get to uphold consistency with our tools. R is a high-performance computing language made to do exactly what we want. .NET features implementation for using R within C# without us needing to write JavaScript or HTML or in any kind of web development language. The extensive package list available for R with R Shiny, while it has more dependencies, also provides functionality to achieve exactly what we want. The development history for R and R Shiny is substantial, with many resources at our disposal. Especially noted are blogs available that

provide their found results of test cases and tutorials for creating interesting graphs. Both D3.js and R Shiny are free tools with supposedly easy implementation and an array of functions, but because R Shiny is built as a solution for our challenge specifically, we decided to select it as a feasible tool to implement visuals of historical summary data in our website.

3.5.2.3 Proving Feasibility

This tool will need to be tested extensively with datasets ranging in size to properly evaluate its potential use in our prototype. We will create test datasets and run these through a simple C# program in .NET. This will give us a chance to see how the implementation of R in C# results as well as test the performance capabilities of R Shiny in a C# application.

Technological Integration

In the last section of our feasibility analysis, we will discuss how all of these solutions to the challenges we face are connected. As you know, we have four subsystems: The Simulator, Reader, Database, and Website. The simulator will be tied with the reader. The simulator will only generate output and send it to the reader. The reader will take the data from the simulator and send it to a queuing server, where the data will wait to be put into the database. We do this to ensure that our system is never overwhelmed. The Database Subsystem encompasses the queuing server as well as the database server. The queuing server will take in the data from all of the SWAPR devices, and another program will pop the data off the front of the queue and build an entry to the database server using the data retrieved. The database server will only get entries from the queue. Nothing else is allowed to create entries to the database server. The database server will also be responsible for serving data to the website. The Website Subsystem is the only subsystem that will be allowed to query the database. The website will query the database in one of two ways. The first way is to get the latest entry from every SWAPR device. The second way is by getting all historical SWAPR data entries from a certain date range and possibly a set or subset of the possible data that a SWAPR generates. There may also be a third way which is to check if credentials entered match credentials stored in the database for faking user account management and role-based permissions. However, this third method of querying the database may not be used depending on whether we get an active directory or not. The website will use the first type of database query to provide the summary view functionality. That functionality includes the list view and map view summaries. The first database query type will also provide the functionality to our notification challenge. The second type of database query will be used to provide the functionality to the historical data summary view as well as the export data challenge.

Conclusion

Our client General Dynamics has identified a couple of additions they'd like to add to the Rescue21 system to be a non-trivial way of detecting issues with the RF equipment on the RFF sites as well as the inability to get the local weather data. These additions will assist the Coast Guard as they need a way to ensure their equipment is functioning 24/7, and without accurate weather data, the Coast Guard can do their job however the more information available means a better choice can be made. That is why General Dynamics reached out to us to discover and build a solution to add to the Rescue21 system to create these additions. Our solution will involve the device that last year's capstone made, which will sit on the site and record data about the antennas transmission power and the weather. Our project will create the software that will tie all of the SWAPR devices together and provide a nice graphical interface for the customer to use to view all of the SWAPR data. We have outlined in great detail throughout this document the challenges that we will face as well as how we plan to overcome those challenges. We also did a general overview of how each of these systems will tie themselves together to create a complete and working project that meets all of the requirements set forth by our client. We will be moving forward with this project by proving the feasibility of each challenge solution, ensuring that the technologies that we choose will actually solve the challenge. If we need to make changes because one of these technologies ends up not working, then we will find the next best alternative and prove that it will work. Once we know what technologies we will be using, we will then start to prototype each subsystem getting feedback from our client along the way. Should our client want changes made then, we will comply. The step after that is to finalize the functionality and UI with our customer until they are happy with our project solution. Once that happens, if time and budget permit, then we will try to get our stretch goals completed and completed to our customers' liking.

Once we complete the project, we hope that the Coast Guard likes the functionality that our project will provide. If so, then they will give the go-ahead to general dynamics to implement the project into the Rescue21 system. Our hope is that it will be as smooth as possible for general dynamics, which is why we emphasized security and ease of maintenance just as general dynamics will when they implement the system.

.