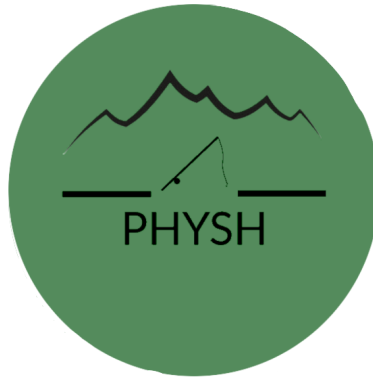


FISH - Fish Identification Search History



Tech Feasibility

Prepared by the members of Team Physh:

Ryan Mason, Scott Austin, Shelby Hagemann, Eduardo Martinez, and Jack Normand

Sponsored by David Rogowski, Ph.D.

Mentored by Vahid Nikoonejad Fard

Table of Contents

Section 1: Introduction	2
Section 2: Technical Challenges	3
Section 3: Technical Analysis	4
3a: Database	4
Introduction to the Issue:	4
Desired Characteristics:	4
Alternative Solutions:	5
Analysis	9
Chosen Approach:	10
Proving Feasibility:	10
3b: Cross-Platform Development	11
Introduction to the Issue:	11
Desired Characteristics:	11
Alternative Solutions:	12
Chosen Approach:	15
Proving Feasibility:	16
3c: Hosting the Database	17
Introduction to the Issue:	17
Desired Characteristics:	17
Alternative Solutions:	19
Analysis:	23
Chosen Approach:	23
Proving Feasibility:	25
3d: Bluetooth Connectivity	26
Introduction to the Issue:	26
Desired Characteristics:	27
Alternative Solutions:	27
Analysis:	29
Chosen Approach:	29
Proving Feasibility:	32
Section 4: Technology Integration	33
Section 5: Conclusion	34

Section 1: Introduction

Data collection is one of, if not the most, important technological focus in the modern era. This practice can give key insights into a population and help analysts come to conclusions that lead to actionable strategies in their research. Every large organization analyzes mass data to some degree. From the greatest tech corporations to local businesses, the analysis and collection of information will always be crucial to organizations reaching their goals.

One organization that relies on data collection is the Arizona Game and Fish Department (AZGFD). The AZGFD is responsible for managing and monitoring fish in the state of Arizona. One of the fisheries that they monitor is the Lees Ferry fishery. The data AZGFD collects here is extremely important because it not only allows Lees Ferry fishery to manage the work better, but also lets them observe how dam operations may be negatively or positively affecting the fish. This crucial form of data collection is not simple. In fact, AZGFD researchers can only sample a small area of the river several times a year. This hindrance makes it extremely hard for the AZGFD to understand what is happening within the fishery.

Currently, the AZGFD operates a program where citizen anglers who catch tagged fish can report data on them. This data is then analyzed by our sponsor, Dr. David Rogowski. David Rogowski is a Wildlife Specialist Regional Supervisor who collects and analyzes data on the fisheries in Arizona. The current data collection system works by having the angler use a PIT tag scanner to scan the fish and receive a code, which then must be cumbersome reported to an AZGFD scientist over the phone to store the data. Not only is it inefficient, but engagement in the program is also low. Arizona's Game and Fish Department needs accurate and consistent data to understand the details of the state's fish population, which is difficult when data collection is sparse.

Our solution to remedy this problem is a mobile application to streamline the data collection process. We envision a system that makes collecting and uploading data easy and more accessible to everyone. This app, called FISH (Fish Identification Search History), will be able to directly upload information on a given fish to a database, saving time from an inconvenient phone call to a researcher. Since the database (or part of the main database) will be directly accessible through the app, the user will be able to view data on the fish they have caught. This

feature will increase engagement and interest within the fishing community, resulting in even more data collection. The application will be available offline, as most fisheries are without internet access. On top of this requirement, FISH will be cross-platform, meaning Android and iOS users will both be able to collect and report data.

The development of an app with all these features will not happen without running into challenges. For this reason, we are conducting a detailed analysis in this paper of every technology we will be using and how they will help implement each feature. In the next section, we will outline some of the major challenges we anticipate while developing this application. We will analyze the major technological challenges for this project so we can identify all of their possible solutions and then select the most promising ones. We will begin this technological feasibility document by evaluating each challenge from a bird's eye view. After this, we will analyze each challenge in its subsection, going further into detail on what needs to be done to achieve our desired solution and what issues may arise in that process. Each potential solution will be evaluated systematically, resulting in a final decision on which solution is ideal, along with an explanation for why it would be the best option for a given challenge. Finally, we will summarize our design choices and describe how each technology will work together cohesively.

Section 2: Technical Challenges

In order to reach our goal for our fully developed application we must tackle the following four technological challenges that we have determined as a group. In this section, we will delve deeper into the different challenges we face, from the expected characteristics of the solution to the many different potential solutions we have come up with. Our technical challenges are as follows:

- We will need a way to store data locally.
- We will need to develop an app to run on both Android and iOS.
- We will need to host the database in some way.

- We will need to connect the Bluetooth PIT tag scanner to the app, collecting data from the scanning.

Section 3: Technical Analysis

3a: Database

Introduction to the Issue:

Our application's main task is to gather and view data, making our backend a crucial aspect of this project. We will need a way to store the fish data that we get from local anglers and send it to a database that the Arizona Game and Fish Department can view for quality control. We have a lot of data to store and our backend is one of the most important aspects of our project. Our database must allow for offline data collection and integrate easily into our wanted architecture. In addition, data from the database will need to be available for viewing in the application. We anticipate having to use a local caching database and a server-side database for our project. The main purpose of this application is seamless data synchronization between itself and the database.

Desired Characteristics:

→ **Size/Scalability:**

- ◆ Currently, the Arizona Game and Fish Department's main database holds around 2 million entries. While this does not accurately represent our storage necessities since we only need to worry about the most current entry for an individual fish. We know that we must allow for any number of entries in our database. We expect the database for the application will start with around 30,000 entries. We want to ensure that the database is scalable to allow for any number of future entries. Although, if a database can store a large enough amount of data without scaling, it is not as important to us compared to other desired characteristics.

→ **Offline Capability:**

- ◆ A very important part of our application is to collect and view data while offline and store it until connected to a network to push the database into the cloud. We will have to implement a local database/storage solution that interacts with the server-side database. With this, we want to ensure that we can store the data locally. Thus, it will be downloadable on a wide variety of mobile devices. We must also guarantee that we can run queries with limited amounts of processing power and memory on mobile devices.

→ **Integration:**

- ◆ Another important aspect is making sure that we can easily implement the database into our system and ensure proper integration with our other technical design choices. We must also keep in mind how the Arizona Game and Fish Department currently stores its data and minimize the need of restructuring or changing currently implemented systems.

→ **Ease of Use:**

- ◆ We want a database that is well-supported, well-documented, and easily maintained. A well-supported database has a variety of tools that make implementation and use simple. A well-documented database has abundant and helpful resources. As a result, developers can easily facilitate the management of the database systems. An easily maintained database will ensure the continued use and relevance of our mobile application.

Alternative Solutions:

SQLite:

“SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine” [3]. It is one of the most deployed databases in the field and is used by Adobe, Apple, Google, and more [3]. It is an embedded SQL database engine, and unlike other SQL databases, SQLite does not have a separate server process [3]. An entire database is contained in a single disk file [3]. The US Library of Congress recommends SQLite database files as a storage format [3]. SQLite is a very compact library, a

size of ~750KiB can be achieved depending on target platform settings [3]. The developer's main wish was to support products that are fast, reliable, and simple to use [3].

→ **Size/Scalability:**

- ◆ We expect to use SQLite as our local data storage while users are offline. It will have to store a condensed version of the fish catch data which is easily supported. The database will only have to store new data for a single fishing session meaning that SQLite will support more than enough storage for this caching model.

→ **Mobile Friendly:**

- ◆ SQLite is one of the common choices for local storage on mobile applications. It is a lightweight version of MySQL designed especially with mobile applications in mind [13]. It is well supported by both IOS and Android which have support for SQLite databases, and have their own management tools [8]. SQLite only uses a single disk file which is important for stable cross-platform use [13].

→ **Integration:**

- ◆ The Arizona Game and Fish Department already utilizes a SQLite database for their condensed fish data. This means we do not have to reformat or change the data in any way. As mentioned above, SQLite is very mobile-friendly and has support on both platforms. It is serverless and does not require internet access. As we have it, a SQLite database will easily integrate into our structure as the offline data storage model for our application.

→ **Ease of Use:**

- ◆ SQLite is serverless, self-contained, and requires no configuration. It is well documented and it is directly supported by both IOS and Android. Android provides thorough documentation for using SQLite on android applications [8].

MySQL:

MySQL is currently the world's most popular open-source database [10]. MySQL is very well known for its proven reliability, performance, and ease of use [10]. Some of the most used applications such as Facebook, Twitter, Netflix, and more are powered by MySQL [10]. It is a relational database management system. MySQL offers a rich and useful set of functions.

“MySQL’s connectivity, speed, and security make it highly suited for accessing databases on the internet” [10].

→ **Size/Scalability:**

- ◆ MySQL was designed as a single-node system and limited vertical scalability [12]. Thus, MySQL relies on sharding to horizontally scale the database across multiple nodes. MySQL Cluster will automatically shard tables across multiple nodes as necessary [7]. Scaling will not be an issue with MySQL, although we are not fans of having to share our data. Fortunately, MySQL can handle large volumes of data and we do not anticipate having to horizontally scale.

→ **Offline Capabilities:**

- ◆ MySQL would be our server-side database, and we would have to implement local storage for the offline data. With MySQL, we anticipate using a locally hosted SQLite database for local storage that syncs with the server-side MySQL database when connected to the internet. There are a few tools available that can help with the syncing between databases or we can create a simple API that handles the data synchronization.

→ **Integration:**

- ◆ Currently, the Arizona Game and Fish Department is utilizing a SQLite database to store condensed fish data. Both MySQL and SQLite are relational database management systems, meaning we will have structured data that MySQL supports. We can implement our offline data syncing strategy without needing to change or restructure the data.

→ **Ease of Use:**

- ◆ MySQL is one of the most popular database management systems in the world and offers a plethora of support and documentation, which are available for public use [6]. While the documentation can be slightly unclear at times and requires some higher-level knowledge of database management, it gives developers the necessary basic information on how to effectively use their system.

MariaDB:

MariaDB is an open-source, multi-threaded, relational database management system [2]. The lead developer is Michael Widenius who was part of the founding team of MySQL AB [2]. Two of MariaDB's most prominent features are speed and scalability [2], both of which would certainly be useful for this application. MariaDB can handle several thousands of tables and many billion rows of data [2]. The developers pride themselves on MariaDB being easily used on the web and having one of the fastest connectors [2].

→ **Size/Scalability:**

- ◆ A key feature of MariaDB is that it is highly scalable. The system utilizes multi-primary node clustering for both availability and read scalability [2]. MariaDB also employs sharding to partition tables across multiple database instances. MariaDB's tables can handle up to 64TB which is more than enough for our anticipated load of data [2].

→ **Offline Capabilities:**

- ◆ MariaDB would be our server-side database. We would still have to implement local storage for the offline data. As mentioned previously, we anticipate using a locally hosted SQLite database for storage on the mobile devices that will sync with the server-side MariaDB database when connected to the internet. Regarding syncing databases, our team can either use already existing tools and APIs, or we could create a simple data-synchronizing API that is more tailored to our project.

→ **Integration:**

- ◆ As mentioned, the Arizona Game and Fish Department is currently utilizing a SQLite database to store the condensed fish data. Both MariaDB and SQLite are relational database management systems, meaning we will already have structured data that MariaDB supports. Due to the compatibility of these two systems, we can implement our offline data syncing strategy without needing to change or restructure the data.

→ **Ease of Use:**

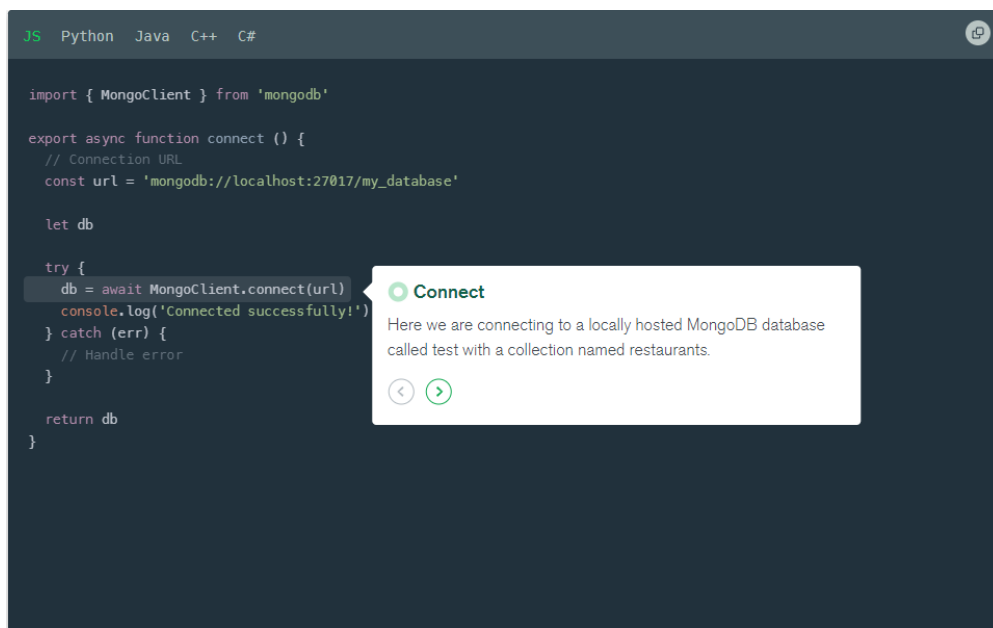
- ◆ MariaDB is one of the most popular database systems available, which means it is well-documented and offers many learning resources. MariaDB has a developer

hub that provides developer toolkits and other forms support [4], which certainly could be useful for this project.

MongoDB:

MongoDB is a non-relational document database that focuses on scalability and flexibility. MongoDB is free to use and is published under the Affero General Public License, or AGPL. This system provides support for 10+ languages, offering many other features through community support, such as drivers [9]. MongoDB stores data in a JSON-like format and is highly flexible which supports changes in data structures [9]. MongoDB is trusted by many companies in the technology industry, including Ebay, Google, SEGA, EA, and many more, showing that it is very reliable [9].

Figure 3a.1



```
JS Python Java C++ C#

import { MongoClient } from 'mongodb'

export async function connect () {
  // Connection URL
  const url = 'mongodb://localhost:27017/my_database'

  let db

  try {
    db = await MongoClient.connect(url)
    console.log('Connected successfully!')
  } catch (err) {
    // Handle error
  }

  return db
}
```

Connect
Here we are connecting to a locally hosted MongoDB database called test with a collection named restaurants.

Analysis

→ Size/Scalability:

- ◆ MongoDB decouples storage and computing to allow for easy scaling [9].
Regardless of the data size, you can increase read capacity to millions of requests per second. It supports horizontal scaling with native sharding. Data is stored in a

flexible JSON-like format that allows for many entries without taking up a ton of space. It is a distributed database at its core [9].

→ **Offline Capabilities:**

- ◆ MongoDB would be our server-side database and we would have to implement local storage for the offline data. As mentioned before, we anticipate using a locally hosted SQLite database for local storage that syncs with the server-side MongoDB database when connected to the internet. This solution could cause issues, because MongoDB and SQLite support different types of formatting.

→ **Integration:**

- ◆ As mentioned above, there will be issues using MongoDB with a local SQLite database. SQLite is relational and MongoDB is based on a non-relational document model, making them incompatible by default. The model for MongoDB differs fundamentally from standard relational databases. While integration is possible between the two, it would require much more work for us to sync offline data with the server.

→ **Ease of Use:**

- ◆ MongoDB's goal with its document model is to provide all capabilities necessary for complex requirements while being simple for developers to learn and use. Its documentation is extensive and is extremely user friendly, offering examples and different use cases for development [5].

Chosen Approach:

We decided as a team that the best approach is the combination of a local SQLite and a server-side MySQL database. The SQLite database will allow our application to function offline. It will hold the downloaded server-side data. Thus, no connection is required for data viewing. In addition, it will store new data that anglers enter while offline. Then, when a connection is reestablished, we can sync our server-side and local databases with each other. This chosen approach will allow us to deliver on our promise of an offline-capable mobile application.

Proving Feasibility:

The combination of SQLite and MySQL databases is the solution that we have chosen for storing data. They are the databases that are most likely to fit our needs and allow us to develop an application that relies on data collection. To prove this feature is effective, we plan to extensively test our chosen solution to ensure that it is exactly what we are looking for. Some features that we are looking to demonstrate regarding our databases include:

- Offline Syncing
- Small footprint in terms of size
- Support for both iOS and Android

3b: Cross-Platform Development

Introduction to the Issue:

Mobile application development is generally divided by the two most popular phone operating systems—Android and iOS. In order to effectively make an application usable and accessible, it must be available to both operating systems. Developers often have two paths for addressing this requirement—they can either create two versions of the same app (one for Android and one for IOS), or they can use cross-platform technology, creating an application that works on both operating systems. We are taking the latter approach, developing the FISH app for both IOS and Android simultaneously. There are several different cross-platform frameworks available to use; however, each with its strengths, weaknesses, and unique quirks. We will dive into several in the later subsections.

Desired Characteristics:

Cross-platform frameworks vary greatly in features, each offering its own desirable and undesirable characteristics. It is important that we take these characteristics into account when planning our development, whether they be ease-of-use, support of different libraries, or specific programming languages. The single most important characteristic of this technology is

accessibility to the libraries that we will need. One key factor in our app will be Bluetooth connectivity, meaning we will need to work with a well-developed Bluetooth library. Any framework with poor Bluetooth functionality will not be considered. This is because one of the main features of our application will be the ability to connect to the PIT tag scanner. The application needs to connect seamlessly and transfer data quickly.

Usability is also an important characteristic for our project. The application's framework needs to be easy to use so we can start development quickly without having to spend too much time learning the random quirks of the software. Some frameworks are complex and use their specific languages for development. If these languages are not written in a way that is similar to common programming languages, a lot of time could be wasted learning them.

Having the ability to quickly deploy and test applications is also very important for this project. It is important to this project that we can test and view our program running efficiently early and often. If the ability to deploy the application on devices were not simple and quick, then it would be difficult to test features and UI layout while developing. The ideal solution would be the ability to test the application with the click of a button or with one single command. Testing is even more important with the app being cross-platform. The ability to test the app on both an Android simulator and an iOS simulator is extremely important. Therefore, an even more ideal solution would be the ability to test the app on a personal smartphone, so we can see how the app feels on a real phone.

Alternative Solutions:

Flutter:

The first possible solution is Flutter. Flutter is a cross-platform app development framework created by Google. Flutter is one of the most recent, but also most popular frameworks to come out. Flutter utilizes its language called Dart, which is a c-style language. It is renowned for its simplicity and ease of use. It was released in 2017, so while it is new, it has released many supporting libraries. It also has a "hot reload" feature that updates the app live so you don't have to recompile it every time you make a change. It is generally considered to be the second most popular framework, behind React Native. Flutter can also be used to create a wide

variety of apps, and has a rendering engine that is not web-based like React Native. Apps such as eBay, Google Pay, and Alibaba all use Flutter.

React:

The next solution is React Native. React Native is a cross-platform framework created by Facebook. React Native was released in 2015, and many popular apps such as Skype, Bloomberg, and Shopify use it. Modules from it also exist on Instagram and Facebook. It has a fast refresh feature that allows developers to see their updates live, and makes debugging quite easy. There are countless libraries for React Native, as it is the most popular framework. It implements the Javascript language for programming, along with HTML and CSS for UI design. This makes it extremely easy to jump right into UI development, as most people are already familiar with HTML and CSS.

Kotlin:

Another solution is Kotlin Multiplatform Mobile. Kotlin is developed by JetBrains and is used by apps like Netflix, Philips, and Baidu. Kotlin was first officially released in 2016 and utilizes its very own Kotlin programming language. The main selling point of Kotlin is the ability to write cross-platform application code separately from platform-specific code. This allows you to organize things more easily during development, and could prove to be extremely useful.

Xamarin:

A final solution would be Xamarin, a framework developed by Microsoft. Some companies that utilize this framework include UPS, Alaska Airlines, and Academy Members. A key feature of Xamarin is that it uses the C# language, a language that many developers feel comfortable using and working with. Existing C# code can be compiled within a given program, which extends the functionality of the application. UI development is also well-supported in this framework, as it allows access to platform-specific UI elements (ex: a sliding touch bar from iOS or a button from Android).

Analysis:

Xamarin:

Though there are plenty of frameworks available, these four options are the most relevant to the application our team is creating. The first step to choosing the best approach would be to install the frameworks and gain a better understanding of how they work. But after doing some more research, we found that several of these frameworks do not need to be installed and further researched. For example, Xamarin can only properly be used in development if three IDEs are opened at once. Because every team member will be downloading and working with the desired framework, this requirement makes Xamarin no longer ideal for our project. In addition to this issue, most Xamarin packages tend to be out of date by several months, which would be a major hindrance to our application, as it will rely heavily on packages.

Kotlin:

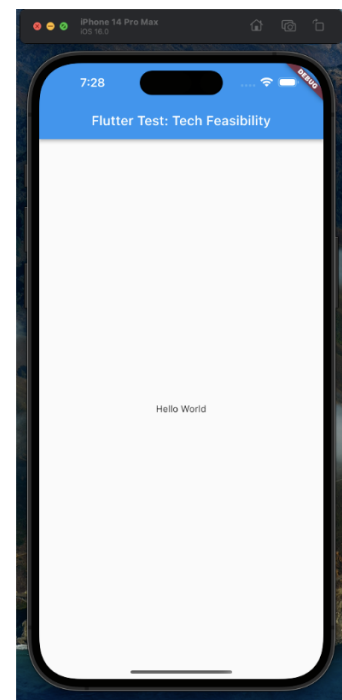
With Xamarin ruled out, it was time to look into Kotlin. Kotlin began to look promising, with its extremely readable Kotlin language and fast compile time. The issue lies in its native support. It would require a lot more effort and native code to be written for the application, which would waste valuable time. With the time to develop the app being so short, this ruled out Kotlin.

Flutter:

Now we move on to Flutter. Flutter looked extremely promising, and was even the first choice for a while. It has a simple, easy-to-understand language, and contains a lot of libraries to support different features. It appeared that the only real downside was the size of the application, and Bluetooth support. Although this was worrying, I decided to install the Flutter framework onto my computer along with all of its dependencies (XCode, Android Studio, VSCode). The environment was easy to set up, and developing simple layouts for applications was easy.

React Native:

Finally, it was time to look into React Native. React Native has the most popularity among app developers, which immediately



Flutter Test App

was a green flag as that means it likely also has the most support. On top of this, the libraries for React Native are countless. Because it is the most relevant cross-platform framework, it also would be a more valuable skill to have in the future. One of the group members, Jack, also already has a decent amount of experience working with React Native. React Native UI elements are also written in HTML and CSS, which the group all has experience with. As if this wasn't enough reason to choose React Native, it also has a library inside called Expo, which makes app development and testing easy and available to every developer on every platform (even Windows developers who want to test on an iOS device). Now onto the test: setting up the environment was just as easy as Flutter, and even allows the user to run the app on a personal phone.



React Test App

Chosen Approach:

After researching each solution extensively, the decision of which framework to use was made obvious. Although Xamarin was tempting, with its ease of implementation of native UI elements, it simply was not well-supported enough and had too many poor reviews. Kotlin was also tempting, as it allows full control of the native elements of an application, but with that comes a lot of extra work. Flutter was extremely tempting as well, and had everything we wanted. But there is just a better option out there.

Table 3b.1

Solution	Pros	Cons
Flutter	Easy to use, widely supported, simple language	Not lightweight, potential Bluetooth problems, better options
Kotlin	Easy to develop native features of each operating system	Extra coding is required for each operating system you wish to run the app on
Xamarin	Runs with C# language, decent support	Horrible reviews and stories about packages lacking support
React Native	Easy to set up and use, widely supported, easy to learn languages, tons of packages, and use of Expo	Potentially not lightweight with Expo installed, lack of Expo packages, but Expo can be uninstalled

The above table displays the pros and cons of each framework. The Xamarin framework is the worst, simply because of its lack of support. Kotlin was considered but simply requires too much setup just to get a simple working app. Flutter and React Native are nearly a tie, but React Native wins out. We chose this framework over Flutter because it is more widely used, and has a much better chance of supporting the functions we wish to implement in the application.

Proving Feasibility:

React Native is the solution we have chosen. This means it's the framework that is most likely to fit our needs and allow us to develop an intuitive and user-friendly application. To prove this, we plan to extensively test this solution to ensure that it is exactly what we're looking for.

Some things we are looking to demonstrate with this framework include:

- Bluetooth connectivity
- Ability to create user-friendly interfaces

- Support for both iOS and Android platforms
 - Database connectivity
-

3c: Hosting the Database

Introduction to the Issue:

Cloud hosting is vital in meeting the Minimum Viable Product for our client and project. Currently, our goal for this project is to be able to host a database in the Cloud, to allow an angler to engage and add to the shared database managed by the Grand Canyon Monitoring and Research Center (GCMRC). This feature is vital for our final product since many of the anglers will not be connected to a network when collecting data. A cloud server will be able to sync data cached locally to a database hosted in the cloud. As well as maintain a larger data store.

Although many cloud hosting services are available, our project requires one that supports cross-platform development (IOS/Android), compatible with javascripts, and utilizes an infrastructure that uses Backend as a Services (Baas) for mobile deployments, and is able to handle offline data synchronization.

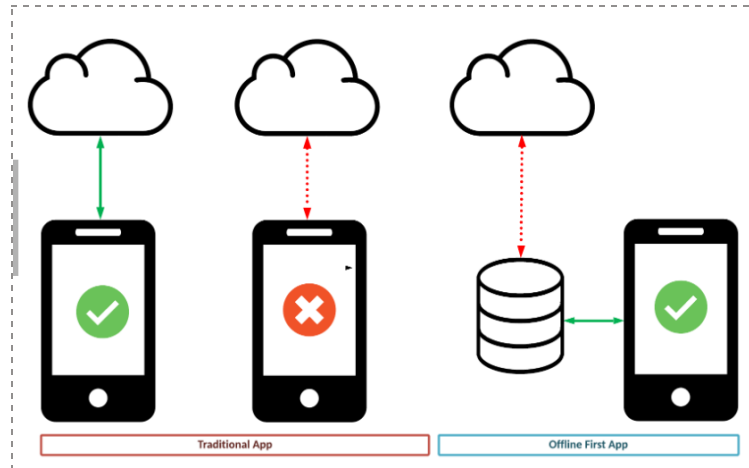
Desired Characteristics:

For the final version of our application, anglers and researchers will be able to update the database on their mobile devices locally, and when connected to a network, update the large cloud database with no difficulty. Although servers will be hosted in the cloud, they will be able to gather information from a locally stored cache in the user's device. Then, when connected to a network, the server will need to:

- Communicate between our mobile application and the database
- Persistently sync data
- Provide a reliable data store backing our clients' data that is highly accessible
- Secure authentication and authorization
- Handle a large real-time database

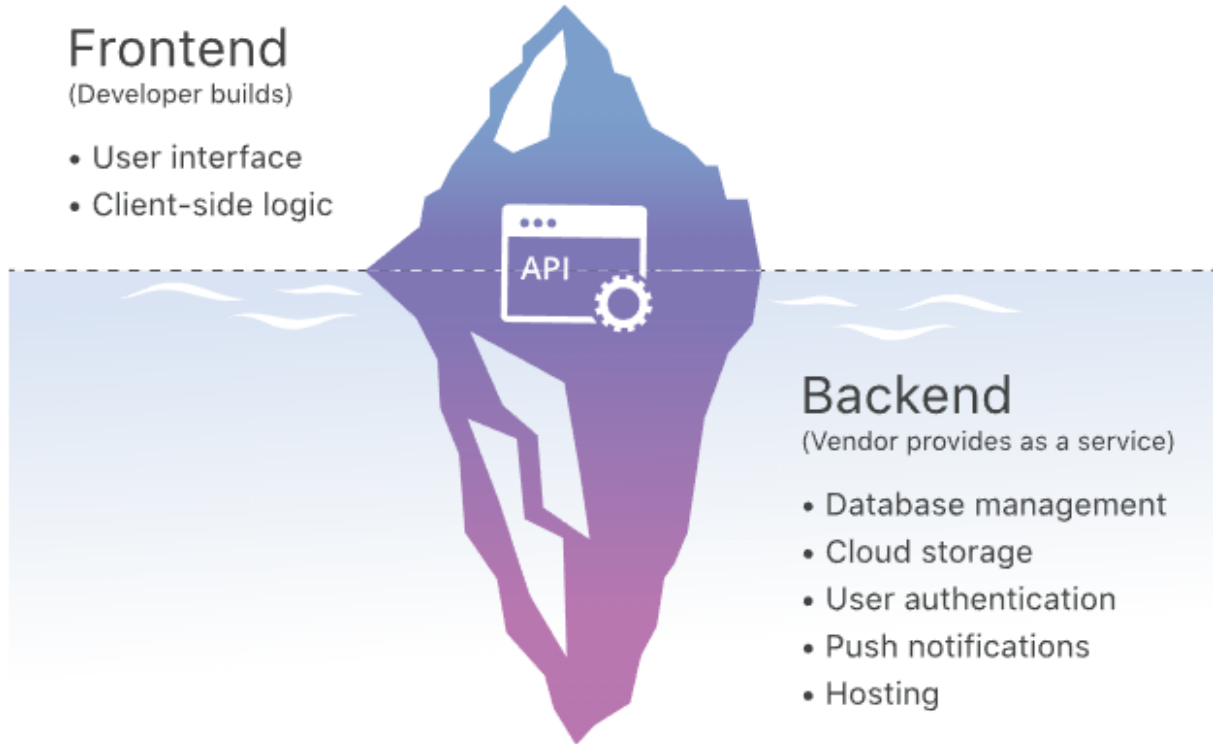
- Support scalability

Figure 3c.1



Overall, our desired solution would be a cloud service model that is built upon a Backend-as-a-Service (BaaS), where the platform would be able to handle database management, cloud storage, user authentication, push notifications, and hosting services. This infrastructure will allow our team to focus on developer builds, such as user interface, and client-side implementation, while preserving the integrity of our backend technologies. [1]

Figure 3c.2



Alternative Solutions:

The main issue with successfully syncing our data is to have a platform that provides efficient libraries to connect and update data persistently. In the case that our chosen approach fails to actively keep an offline persistence, either due to SDK library compatibility, or incorrect data manipulation. There are similar Cloud hosting services that are also built of Back end as a service, who specialize in mobile deployment.

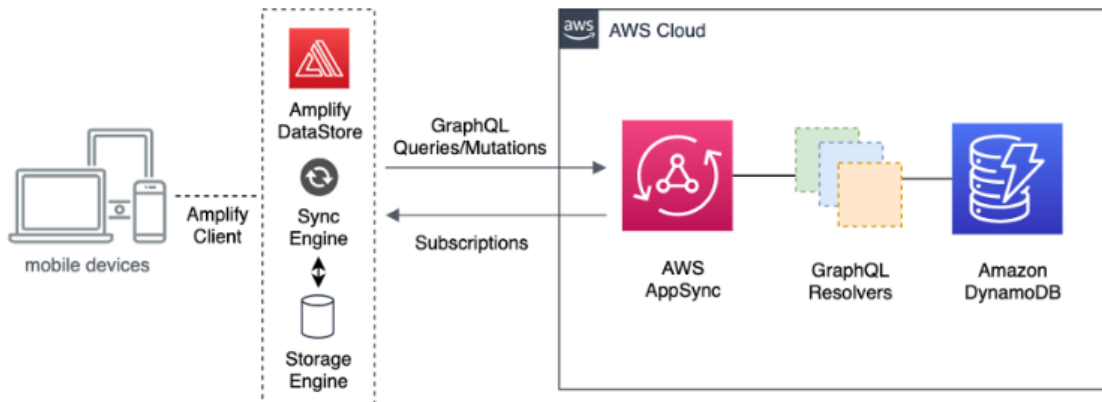
AWS Amplify:

AWS Amplify is a relatively new cloud hosting service for mobile development. It launched in November 2018 and was built on Amazon web services, however, it is tailored more for mobile development[2]. This solution offers an infrastructure known as Amplify Datastore which is a client-server that runs inside the mobile devices and exposes an API for developers to

interact with. In turn, it functions as a repository to store data locally and synchronize it to the cloud automatically when network connections are made.

Figure 3c.3

Reference architecture



As seen above, the Amplify DataStore is a subset of the larger AWS cloud server, that communicates with AWS cloud, handles data queries, and provides a pre-built sync engine to support developer needs. AWS Amplify provides many key features that are aligned with our desired characteristics, such as:

- Integration with React Native (our primary cross-platform development environment, as well as flutter and many other cross-platform environments)
- Real-Time and offline functionality
- Easy and fast deployment
- Ability to create/host an app backend and frontend user interface
- Provides BaaS infrastructure
- AWS RDS [compatible with SQL structures and MariaDB]

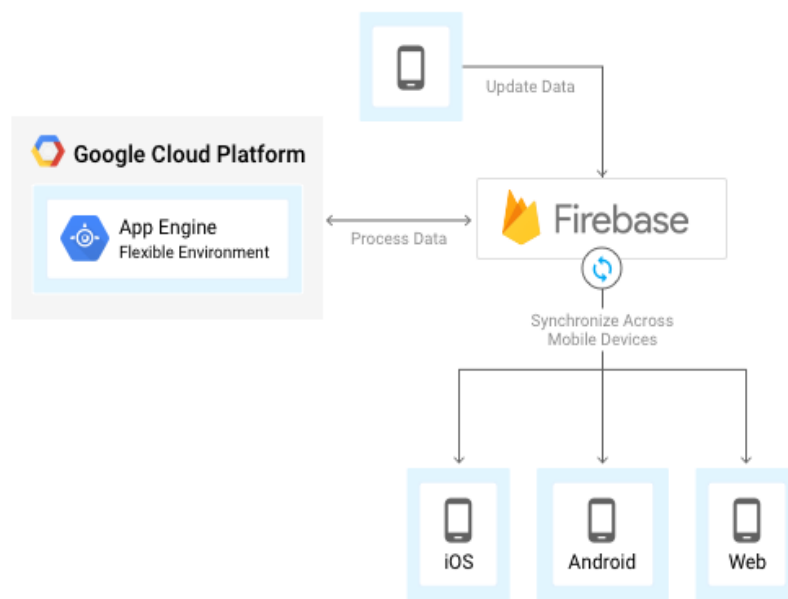
AWS Amplify provides a unique and favored alternative due to its pre-built engines, storage capacity, and ease of deployment. However, its drawbacks are its relatively new nature, a high learning curve for developers, and pre-built architecture that makes it difficult to manipulate.

However, the unexpected cost of data transferring can cause difficulty when collaboration between developers begins.

Google Cloud Platform:

Google's cloud platform is renowned for its support efficiency of frequent offline and online data persistence. It offers a feature that caches a copy of its Cloud Firestore data that our project can benefit from. In turn, our mobile application will be able to access the data when the device is offline and set listeners() to persistently sync data when connected to a network. This allows the ability to write, read, listen to, and query the locally cached data that is stored on the user's mobile. As the user's device reconnects to a network (back online), Cloud Firestore is then able to synchronize any local changes made in the application to the Cloud Firestore backend. [3]

Figure 3c.4



As shown in the figure above, Firebase acts as an engine that synchronizes data across all mobile devices when an update has been made. The Firebase engine also enables google to synchronize data when collected offline. The google Cloud Platform provides many means to achieve our desired functionality, such as:

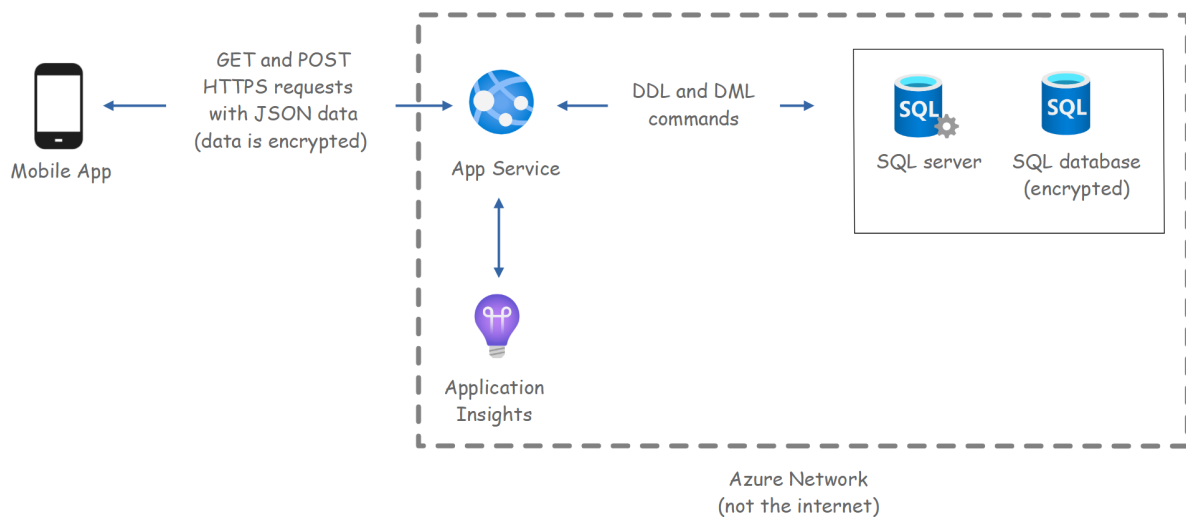
- Configuring cache size, the local data that will be synchronized during network connection
- Listen, get, and query offline data
- Integration with React Native and a large amount of documentation support
- Designed to scale
- BaaS infrastructure
- Provides a relational database known as Cloud SQL, which simplifies migration from MySQL, PostgreSQL, and SQL Server.

Overall, Google’s cloud platform provides solutions to the many issues we will be facing. However, its drawbacks, such as limited support for IOS features and manipulation querying abilities, might hinder the process of our cross-platform development goal.

Microsoft Cloud:

Another possible alternative for our solution would be to use Microsoft Azure. Azure is a well-known cloud computing service operated by Microsoft for application management. Up and running since 2010, its robust cloud hosting capabilities have many diverse functions, including offline synchronization and an integrated autoscale system. Microsoft Azure contains a specific platform built on the BaaS infrastructure known as Azure Mobile Apps Services. [4]

Figure 3c.5



Much like the other cloud services, it offers offline data sync to build responsive apps. However, the uniqueness of Microsoft Azure is its data access framework that provides another solution to accessing data offline. It features a mobile-friendly entity framework known as ASP.NET core developed by Microsoft[5], which supports high-performance and cross-platform development for modern, cloud-based applications. This framework aligns with many of our desired characteristics, including but not limited to:

- Offline data synchronization
- Supports SQL data provider
- Client SDK libraries to handle both HTTP requests/responses for developer support

Analysis:

Through further discussion, we reviewed the pros and cons of each cloud hosting platform to distinguish which would benefit the integrity database maintenance and offline persistence. Through running and deploying test servers to evaluate the connection between database and server, we analyzed the different offline capabilities for each cloud platform, supporting libraries which they provide, scalability, integration with our chosen cross-platform development, and the learning curve each platform contains. The following section demonstrates the logic of our chosen decision and the consideration of the characteristics of each platform and what they had to provide

Chosen Approach:

Although each hosting service provides a unique solution to our hosting issues and offline synchronization, Amazon Web Services Amplify and Relational Database (RDS) best integrate an SQL-like database with offline persistence. This approach will ensure security, scalability, ease of deployment, and real-time/offline functionality that would best suit the minimal viable product for our project. The most considerable drawback to this approach is the learning curve of the AWS system. However, due to its extensive documentation and built off a trusted and reliable service (AWS), we believe we'll be able to implement all necessary offline features and functionality.

The primary reason Google’s cloud platform was not selected was due to its offline integration with an SQL database. Google’s cloud platform does allow SQL databases to be built but integration with Firebase, the offline sync engine, primarily works with NoSQL cloud-hosted databases, making integration with our chosen approach of using MySQL difficult. We steered away from the Microsoft Cloud alternative, due to its offline capabilities with mobile clients run on Xamarin, and integration with react native will require plugins adding to the level of abstraction and taking away modularity in our design.

Table 3c.1

Alternative Solution Hosting	Pros	Cons
Amazon Web Services Amplify and RDS	<ul style="list-style-type: none"> ● Cross-platform compatible ● Real-time and offline functionality ● Compatible with many different types of databases and environments ● Expansive documentation on integration with react native 	<ul style="list-style-type: none"> ● Learning curve ● Pre-built architectures can be difficult to manipulate
Google Cloud Platform	<ul style="list-style-type: none"> ● Cross-platform compatible ● Offline synchronization persistence ● Real-time and offline synchronization ● Expansive documentation 	<ul style="list-style-type: none"> ● Does not support SQL database ● Limited support for IOS functionality ● Pre-built data querying is difficult to manipulate

Microsoft Cloud (Azure)	<ul style="list-style-type: none"> ● Cross-platform compatible ● Offers many pre-SDK libraries that would be useful for development ● Expansive documentation 	<ul style="list-style-type: none"> ● Not specifically written for React Native, Xamarin ● An added level of abstraction might hinder our development process
-------------------------	--	--

Proving Feasibility:

Now to see how the implementation of AWS Amplify and RDS will integrate, we will test how the interaction from the GraphQL of Amplify will interact with a test SQL database.

Figure 3c.6

```

1 USE Marketplace;
2 CREATE TABLE Customers (
3   id int(11) NOT NULL PRIMARY KEY,
4   name varchar(50) NOT NULL,
5   phone varchar(50) NOT NULL,
6   email varchar(50) NOT NULL
7 );
8 CREATE TABLE Orders (
9   id int(11) NOT NULL PRIMARY KEY,
10  customerId int(11) NOT NULL,
11  orderDate datetime DEFAULT CURRENT_TIMESTAMP,
12  KEY `customerId` (`customerId`),
13  CONSTRAINT `customer_orders_ibfk_1` FOREIGN KEY (`customerId`) REFERENCES `Customers` (`id`)
14 );

```

Here we will create and connect to a database, then by running amplify API add-graphql-data source from our amplify project, and pushing to AQS we can now interact with the SQL database from amplified GraphQL. We can test this by creating a data point and seeing the queries in the SQL database. [6]

Figure 3c.7

```
1 mutation CreateCustomer {
2   createCustomers(createCustomersInput: {
3     id: 1,
4     name: "Hello",
5     phone: "111-222-3333",
6     email: "customer1@mydomain.com"
7   }) {
8     id
9     name
10    phone
11    email
12  }
13 }
```

```
{
  "data": {
    "createCustomers": {
      "id": 1,
      "name": "Hello",
      "phone": "111-222-3333",
      "email": "customer1@mydomain.com"
    }
  }
}
```

3d: Bluetooth Connectivity

Introduction to the Issue:

Bluetooth connection is key for meeting the Minimum Viable Product for this project. One of the key goals for FISH is to allow anglers to connect their mobile devices to their PIT scanners, where they will then update the data on their caught fish, sending new information to the main database. This feature is especially useful in that it can be utilized by anglers even when they are without internet access, which is common in fishing in Northern Arizona.

Bluetooth involves creating a “server” on a device that other devices can be paired with. In the case of FISH, this entails establishing a connection between our mobile app and the PIT scanner, in which the mobile app would function as the server. To achieve this function, “one device [the app] must open a server socket, and the other [the PIT scanner] must initiate the connection using the server device’s MAC address” [1]. A server socket within the app will search for other devices for connection [2], finding the PIT scanner. Once the angler selects their PIT scanner for device pairing, they will be able to transfer data from the scanner to their phones, updating the fish database.

Desired Characteristics:

Bluetooth connection will be swift and easy to activate within our application. When a user opens the FISH app for the first time, they will be prompted with messages asking for certain permissions which will be necessary for activating Bluetooth connection [5]. These permissions will allow the app to access their phone's Bluetooth components and either their MAC address (Android device) or their UUID (IOS device). Then, the user will be able to scan for Bluetooth devices to connect with, where they will then choose their PIT scanner. Once paired, the app will store the data for the PIT scanner, allowing for quick and seamless pairing in the future.

Alternative Solutions:

Ideally, our app will act as a Bluetooth server that the PIT scanners can connect to. However, there are other ways to implement Bluetooth connectivity if the desired route is not possible or efficient. For example, one API that allows for Bluetooth connection in React Native-developed apps is react-native-bluetooth-serial [4]. This API is developed based off of Bluetooth-Serial, another API that primarily works with Android-developed apps. The primary difference between these two APIs is that react-native-bluetooth-serial has functions designed specifically for cross-platform development. An alternative option for implementing Bluetooth connectivity with React Native is React Native BLE PLX [6]. According to this library's documentation, it supports "making connections to peripherals" [6], which are "devices [that] can connect to [a] computer using Bluetooth, [ranging] from input devices... to output devices" [7]. For this reason, BLE PLX could be useful in pairing with PIT scanners, which are unorthodox in terms of what generally gets paired to a smartphone.

While the chosen approach for app development is React Native, there are several methods for implementing Bluetooth connection if Flutter ends up becoming a more ideal solution for programming. For example, flutter_blue is a new API being developed that is designed to make Bluetooth connection on flutter apps simple and quick.

The following table displays functions from the APIs and libraries described above, which may be useful for this project:

Table 3d.1

API/Library	Function	Description
react-native-bluetooth-serial	enable() (Android only)	Allows the app to search for Bluetooth devices (only applicable for Android devices)
react-native-bluetooth-serial	disable() (Android only)	Stops the app from searching for Bluetooth devices (only applicable for Android devices)
react-native-bluetooth-serial	isEnabled()	Verifies that the app/device is ready for Bluetooth pairing, which would be necessary for this project.
react-native-bluetooth-serial	list()	Returns a list of devices that can be paired via Bluetooth. This would be useful for this project during the pairing stage, allowing the app to parse through devices and select the PIT scanner.

react-native-bluetooth-serial	connect(String id)	Connects the app to a device using either the MAC address (Android devices) or the UUID (IOS devices). This is imperative for pairing via Bluetooth.
react-native-bluetooth-serial	write(Buffer String data)	Used to write data to a device paired via Bluetooth.

Analysis:

To determine which method would be most ideal for this project, the development team met and discussed the pros and cons of each possible solution. Each of the possible solutions discussed in this paper have the basic desired characteristics necessary for obtaining the minimum viable product, such as pre-existing methods for connection and cross-platform compatibility. However, some solutions posed better than others, offering more potentially useful features and being easier to work with for less seasoned app developers.

Our team met to go over these different options and take a vote on what the chosen approach would be. We each read the descriptions and documentation for the APIs addressed above and evaluated each of the potential issues they posed for development. The following section addresses our decision-making method and explains the logic that led us to our final choice for implementing Bluetooth connectivity.

Chosen Approach:

The primary reason why Bluetooth Serial was not chosen is that it is not tailored to React Native. While this API offers very extensive documentation and many useful functions, react-native-bluetooth-serial is built off of it, offering better compatibility with the chosen programming language. Therefore, there are better options for development than Bluetooth Serial.

While it is not a key factor in the minimum viable product, support for bonding peripherals is a feature that would enhance the experience of anglers using the FISH app. Some of the potential solutions for Bluetooth connectivity do not support this feature, including React Native BLE PLX. Because support for bonding peripherals would significantly improve the experience of users, these APIs were not the chosen approach.

React-Native-Bluetooth-Manager is an API that offers many useful features and obtains all of the desired characteristics for the minimum viable product. However, this API offers less documentation and tutorials compared to other potential solutions. Because each of the developers on this project is not an expert on app development, we decided it would be best that we choose an approach that offers more resources for learning, ensuring that we develop the product to the best of our abilities.

Several of the options addressed so far for Bluetooth connectivity utilize Bluetooth Low Energy connection. While this method of creating Bluetooth connections works very well at conserving energy, it would create some issues regarding compatibility with the PIT scanners. This is because PIT scanners are primarily compatible with classic Bluetooth features and not the Bluetooth Low Energy features. For this reason, react-native-bluetooth-classic would be ideal. This API is also very useful in that it offers substantial documentation and example code on GitHub, which can be used as we teach ourselves how to code with React Native. If a different approach is used for Bluetooth connectivity, we would need to write more tools ourselves to get the Bluetooth Low Energy APIs to function with the PIT scanners.

Table 3d.2

Alternative Solution (APIs and Packages)	Pros	Cons
React-native-bluetooth-serial library	<ul style="list-style-type: none">• Cross-platform compatible• Same installation process for IOS and Android• Compatible with many different types of hardware• Expansive documentation	<ul style="list-style-type: none">• GitHub says device connection feature on IOS needs to be improved• Certain functions only work on Android devices• Uses Bluetooth Low Energy
React Native BLE PLX	<ul style="list-style-type: none">• Cross-platform compatible	<ul style="list-style-type: none">• Does not support bonding peripherals• Uses Bluetooth Low Energy
BluetoothSerial	<ul style="list-style-type: none">• Cross-platform compatible• Offers many pre-written functions that would be useful for development• Expansive documentation	<ul style="list-style-type: none">• Not specifically written for React Native, may require extra work• Uses Bluetooth Low Energy

flutter_blue	<ul style="list-style-type: none"> • Cross-platform compatible • Provides function for writing and working with characteristics • Has a function to set notifications for user 	<ul style="list-style-type: none"> • Actively still being developed, could have bugs since it's new • Designed for Flutter, which is not the chosen language for development • Uses Bluetooth Low Energy
react-native-ble-manager	<ul style="list-style-type: none"> • Cross-platform compatible 	<ul style="list-style-type: none"> • Uses Bluetooth Low Energy
react-native-bluetooth-classic	<ul style="list-style-type: none"> • Offers online examples as learning resources • Easy testing with the Universal WorldScan Reader 	

Proving Feasibility:

To ensure that we understand how to effectively implement Bluetooth connectivity and that our chosen approach achieves our desired characteristics, we will create a simple application that will demonstrate Bluetooth usage by connecting to a demo PIT scanner. We have already verified that PIT scanners can easily connect to Bluetooth-enabled devices, such as mobile phones. In order to test that the scanners can connect with the FISH app, in particular, we will need to run the app on our phones.

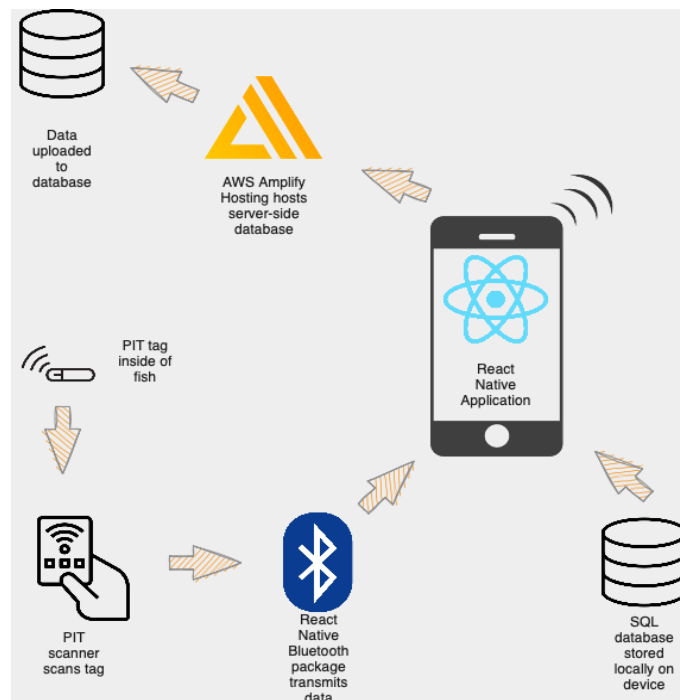
While many of the other features in the FISH app will be able to be tested through emulators simulating mobile phones on computers, Bluetooth cannot by default be used or tested on an emulator. This is because Bluetooth connectivity relies on the hardware of a device and emulators do not possess hardware themselves. Due to this restriction, we will be testing

Bluetooth connection by running the FISH app on our own mobile devices with a PIT scanner leased to us by our sponsor.

Section 4: Technology Integration

Now that we have all these various technologies laid out, we have to make these solutions work together cohesively. These technologies form the basis of our entire application and will work together seamlessly to create an intuitive, user-friendly app. They will all work together in a way that satisfies the project requirements, and maintain an attractive appearance. Below is a diagram that illustrates this.

Figure 4.1



As the diagram above depicts, all the relevant pre-existing fish data is first stored locally on the mobile device. This allows the angler to view data on a caught fish that is tagged. Once the angler catches a fish, they can use the PIT scanner to check if it is tagged. If it is, the scanner will transmit the tag ID to the user's iPhone either by hand or by using Bluetooth. Then, the user will use the application to enter any more data such as the measured size of the fish, or the date/time caught. The user can use the locally stored data to see any available information about

the fish they just caught. Finally, the user will store the data on the device until an internet connection is reached when the data will be uploaded to the server-side database hosted on AWS Amplify.

Section 5: Conclusion

The primary issue with the current system of monitoring is when an angler catches a fish they need to contact a scientist that has access to the database to manually locate that fish's information, overall making this a very inefficient process. Our goal is to fully develop this app to assist the everyday angler by allowing a Bluetooth transfer of the scanned PIT tag ID number. The app will then use this ID number to locate and display the fish's respective information for the angler as well as provide a way to update this information for future catches, removing the current system's middleman.

Since we are developing a cross-platform application we needed to find a framework that would support this. Ultimately we landed on using the React-Native framework to develop the front end of the application. Alongside the application itself, we need to be able to have a hosted database that holds all rainbow and brown trout information as well as a compressed database that will be stored locally on the angler's smartphone to be accessed without cellular data. To accomplish all of this we decided to use the AWS Amplify and RDS services to host our SQL-based database that will contain all fish information and we will use a SQLite database to store the necessary fish data on a smartphone locally. The last major technical challenge we have been faced with is the implementation of Bluetooth connectivity. This decision weighed heavily on the choice of the cross-platform framework. Because of this, we will use the react-native Bluetooth serial module which supports Bluetooth connectivity over iOS and Android.

From this point forward we are planning to complete the Requirements Specification document listing the functional and non-functional requirements for our project. On top of this document, we will be working on the creation of our presentation poster that will be presented at the December 2 conference. Overall we hope to be able to develop a reliable application that the

everyday angler can use to view their catch's information within seconds, a much smoother and faster process than what is currently in place.

Sources

Bluetooth Relevant Sources:

[1] Android Developer. 2021. Connect Bluetooth Devices. (October 2021). Retrieved October 10, 2022 from

<https://developer.android.com/guide/topics/connectivity/bluetooth/connect-bluetooth-devices>

[2] Android Cookbook. 2022. Opening a Bluetooth Server Socket Listener. (June 2022). Retrieved October 10, 2022 from

<https://www.androidcookbook.info/application-development/opening-a-bluetooth-server-socket-listener.html#:~:text=A%20Bluetooth%20Server%20Socket%20is,requests%20from%20remote%20Bluetooth%20Devices.>

[3] Android Developer. 2022. BluetoothAdapter. (July 2022). Retrieved October 11, 2022 from

[https://developer.android.com/reference/android/bluetooth/BluetoothAdapter#listenUsingRfcommWithServiceRecord\(java.lang.String,%20java.util.UUID\)](https://developer.android.com/reference/android/bluetooth/BluetoothAdapter#listenUsingRfcommWithServiceRecord(java.lang.String,%20java.util.UUID))

[4] Daniel Lima, Medium. 2018. How to create a Bluetooth App with React Native. (October 2018). Retrieved October 11, 2022 from

<https://medium.com/@daniel.lima.nascimento/how-to-create-a-bluetooth-app-with-react-native-5212d8590e6b>

[5] Android Developer. 2022. Bluetooth Permissions. (October 2022). Retrieved October 10, 2022 from <https://developer.android.com/guide/topics/connectivity/bluetooth/permissions>

[6] Dotintent. 2021. React-native-ble-plx. (November 2021). Retrieved October 10, 2022 from <https://github.com/dotintent/react-native-ble-plx>

[7] Robot Powered Home. 2022. Bluetooth Peripheral Device: What is it? (March 2022). Retrieved October 11, 2022 from

<https://robotpoweredhome.com/bluetooth-peripheral-device/#:~:text=Bluetooth%20peripheral%20devices%20are%20devices,multimedia%20devices%20like%20digital%20cameras.>

Database Hosting Relevant Sources:

[1] AWS. 2022 AWS Amplify: Build Full-stack web and mobile apps in hours. Easy to start, easy to scale. Retrieved October 13, 2022 from

https://aws.amazon.com/amplify/?trk=41731cf6-f5eb-4611-81ef-f2914ec706b5&sc_channel=ps&s_kwid=AL!442213!588971138365!e!!g!!aws%20amplify&ef_id=Cj0KCCQjwkt6aBhDKARIs

[AAyeLJ1y7VO_rWG15me5WZgKDAB58F60V_Zsz2QsujbidCUfGP4esoHPx04aAhssEALw_wcB:G:s&s_kwcid=AL!4422!3!588971138365!e!!g!!aws%20amplify](https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/)

[2] CLOUDFLARE. 2022 What is BaaS: Backend-as-a-Service vs serverless. Retrieved October 15, 2022 from

<https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/>

[3] Firebase. 2022 Documentation > Firestore: Access data offline. Retrieved October 10, 2022 from

<https://firebase.google.com/docs/firestore/manage-data/enable-offline>

[4] Azure. 2022. Azure Cloud Services. Retrieved Oct 12, 2022 from

<https://azure.microsoft.com/en-us/products/cloud-services/#overview>

[5] Microsoft. 2022 Documentation: Offline Data Sync. Retrieved October 12, 2022 from

<https://learn.microsoft.com/en-us/azure/developer/mobile-apps/azure-mobile-apps/howto/data-sync>

[6] AWS. 2022 Front-End Web & Mobile: Connect Amplify DataStore with existing SQL data sources; adding offline and sync features in your application. Retrieved Oct 24, 2022 from

<https://aws.amazon.com/blogs/mobile/connect-amplify-datastore-with-existing-sql-datasources-adding-offline-and-sync-features-in-your-application/>

Database Relevant Sources:

[1] MariaDB. 2022. About mariadb server. (September 2022). Retrieved October 25, 2022 from

<https://mariadb.org/about/>

[2] MariaDB. About mariadb software. Retrieved October 25, 2022 from

<https://mariadb.com/kb/en/about-mariadb-software/>

[3] SQLite. About SQLite. Retrieved October 25, 2022 from <https://www.sqlite.org/about.html>

[4] MariaDB. MariaDB developer hub: Database developer resources. Retrieved October 25, 2022 from <https://mariadb.com/developers/resources/toolkits/>

[5] MongoDB. MongoDB Documentation. Retrieved October 25, 2022 from

<https://www.mongodb.com/docs/manual/>

- [6] MySQL. MySQL documentation. Retrieved October 25, 2022 from <https://dev.mysql.com/doc/>
- [7] MySQL. MySQL NDB Cluster: Scalability. Retrieved October 25, 2022 from <https://www.mysql.com/products/cluster/scalability.html>
- [8] Android. Save data using sqlite : android developers. Retrieved October 25, 2022 from <https://developer.android.com/training/data-storage/sqlite>
- [9] MongoDB. What is mongodb? Retrieved October 25, 2022 from <https://www.mongodb.com/what-is-mongodb>
- [10] Oracle. What is mysql? Retrieved October 25, 2022 from <https://www.oracle.com/mysql/what-is-mysql/#one-choice>
- [11] Geekboots. 2019. How sqlite helps in Mobile App Development. (March 2019). Retrieved October 25, 2022 from <https://www.geekboots.com/story/how-sqlite-helps-in-mobile-app-development>
- [12] Mike Benshoof. 2021. A horizontal scalability mindset for mysql. (September 2021). Retrieved October 25, 2022 from <https://www.percona.com/blog/horizontal-scalability-for-mysql/>
- [13] Yuliia Panasenko, Maksym Karpovets, and Daria Bulatovych. Why make your business software available offline? Retrieved October 25, 2022 from <https://yalantis.com/blog/offline-mode-application/>

Cross Platform Relevant Sources

- [1] Rabe, Jan. Everything that is wrong with Xamarin and why it is bad for you. Retrieved October 23, 2022 from <https://medium.com/@kibotu/everything-that-is-wrong-with-xamarin-and-why-it-is-bad-for-you-a5399075e50a>
- [2] React Native Getting Started Guide. Retrieved October 23 from <https://reactnative.dev/docs/getting-started>

[3] Flutter Tutorials. Retrieved October 23 from <https://docs.flutter.dev/reference/tutorials>

[4] Kotlin FAQ. Retrieved October 23 from

<https://kotlinlang.org/docs/faq.html>

[5] React Native Vs. Flutter: Which one is better in 2022?. Retrieved October 23 from

<https://fireart.studio/blog/flutter-vs-react-native-what-app-developers-should-know-about-cross-platform-mobile-development/>