View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by The University of Akron

The University of Akron
IdeaExchange@UAkron

Honors Research Projects

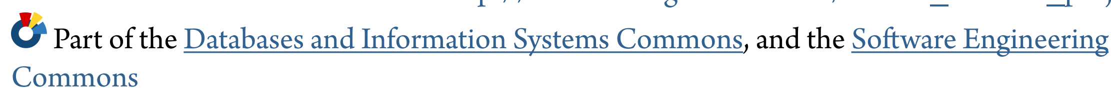
The Dr. Gary B. and Pamela S. Williams Honors College

Spring 2018

Exercising Efficiently with an Equipment Ticketing Mobile Application

Eric Merryman
erm38@zips.uakron.edu

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.
Follow this and additional works at: http://ideaexchange.uakron.edu/honors_research_projects


Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Recommended Citation
Merryman, Eric, "Exercising Efficiently with an Equipment Ticketing Mobile Application" (2018). *Honors Research Projects*. 710.
http://ideaexchange.uakron.edu/honors_research_projects/710

This Honors Research Project is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.

Exercising Efficiently with an Equipment Ticketing Mobile Application

Eric Merryman

The University of Akron

Honors Project

Spring 2018

Table of Contents

Introduction 1
Purpose 1
Methodology 2
Xamarin.Forms 3
History 3
Use 4
Languages Used 4
XAML (Extensible Markup Language) 4
C# 5
Object-Oriented Programming Principles Used 5
Interfaces 5
Encapsulation 5
Dependency Service 6
Database Use 7
SQLite 7
MongoDB 7
Object Relational Impedance Mismatch 9
RESTful Web Service 9
Node.js and Express 9
Application Workflow 10
Supplemental Management Web Application 15
MEAN Stack 15
Functionality 16
NuGet and npm 17
Conclusion 18
Native vs. Cross-Platform 18
Expectations for Xamarin.Forms 19
Personal Experience 19
Technical Requirements 20
References 21

Introduction

As technology continues to advance, we are seeing a movement toward compactness and availability of solutions. This has created a high demand for mobile products that allow us to perform everyday tasks more efficiently and conveniently. Also, companies are looking to mobile products in order to deliver content to their customers. This project will attempt to advance technology in the gym which is a market that has welcomed technology and has been able to continue growing because of it. Gyms and exercise facilities can be very intimidating to newcomers or those who have very basic knowledge of such an environment while mobile devices and their capabilities are something that we have become very familiar with. Therefore, for my project, I will be designing a mobile application that allows for a more welcoming and efficient exercise environment by allowing users to document any abnormalities with gym equipment.

Purpose

A solution that allows users to report broken or malfunctioning equipment will allow gym-goers to feel much more comfortable in a gym environment as they will not have to confront someone who may intimidate them with an embarrassing issue. Since many people are afraid to mention these incidents, equipment sees a much longer downtime than it should. This solution would be similar to the ticket system that IT departments have adopted in almost every organization. It would allow for the gym's management staff to document these incidents which in turn allows them to reduce downtime, possibly repair the equipment on their own in the future and generate reports in order to assess the machine and its worth. Other advantages of doing so include the following: increased customer satisfaction, more customers in the gym at a single moment, less of a dependency on a third-party to provide repairs/maintenance, reduced injury from attempting to use a malfunctioning machine, and reports that allow management to make future purchasing decisions based on the performance of a previous piece of equipment. Also, an application like this can then be used to deliver information to its users that is relevant to the company or further their customers' education of the gym and its equipment.

Methodology

This solution will be developed using Xamarin's cross-platform development toolset. Xamarin is becoming a popular choice when designing mobile applications as it allows for application developers to write code in only two languages that can then be transformed into the native code that will be run on different mobile operating systems. Applications are written using only C# and XAML which requires developers to learn only two languages before creating an application that can be ported to Android, iOS and Windows Phone. A SQLite database is packaged with the mobile application to serve static data about the gym's equipment to the application. Using the JavaScript execution environment Node.js, the NoSQL database engine MongoDB and the JavaScript server framework Express, a RESTful web service is used to serve dynamic data to and receive incident data from the mobile application. A supplemental web application, also making use of the RESTful web service, will be created in order to simulate the management aspect of a ticketing system. This web application allows administrators to check the status of each equipment and manage any incidents that have been reported. The MEAN Stack will be used to implement the web application which uses the frameworks of the RESTful web service and adds AngularJS as an MVC architecture. The architecture and interactivity of the project is given in Fig 1.

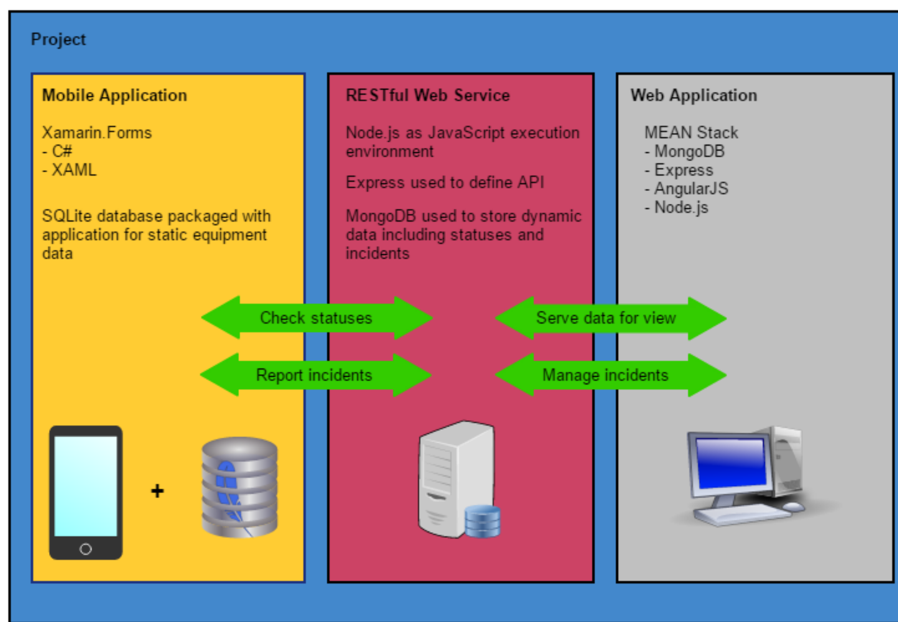


Figure 1: Architecture of GymTicket solution

This application will be debugged and tested on two physical devices, a Samsung Galaxy S4 and an iPhone 6. The operating systems installed on the Samsung Galaxy S4 and the iPhone 6 are Android Lollipop (5.0.1) and iOS 10.2 respectively.

In order to deploy to and debug on the Android device, the developer options has to be enabled which is done by navigating to the "About Phone" section in the settings and then clicking on the "Build Number" label 5 times in rapid succession. This allows for hidden functions to be used such as accessing various logs and many other options useful for debugging such as touch feedback. When connected to the development machine running Visual Studio, the option to debug via this device is readily available as the Android SDK is installed with the Xamarin.Forms plugin. Once built and deployed, the application can be used from the home screen with or without a connection to the development machine, but requires connection in order to deploy a newer build

For the iPhone, a separate machine running MacOS is needed with Xcode installed in order to build the iOS project. A Mac Mini will be used as the build machine for this project which runs MacOS Sierra (10.12.6). Within Visual Studio for Mac, the repository on GitHub will be cloned to the Mac machine to be built and the project will then be deployed to the device. In order to debug on a device as opposed to a simulator, the proper code signing keys are needed and the device needs provisioned for debugging. When deployed, the application needs to be verified in the settings before being used. As with the Android device, the application could be used from the home screen with or without a connection to the development device and only needed reconnected to deploy a newer build.

Xamarin.Forms

History

Xamarin began as a project of Miguel de Icaza and Nat Friedman's known as Ximian. Ximian's primary project was Mono, an attempt to recreate the .NET framework so that it could be used to develop for other operating systems with their primary focus being Android. In 2003, Novell acquired Ximian

whose team was later relieved of their duties in 2011 after Novell's acquisition of Attachmate. This left many worried about Mono and the Ximian team responded by founding Xamarin who would focus on tools for mobile development. After gaining a foothold in the mobile-application market, Microsoft purchased Xamarin in order to bolster its software sales by providing applications that run on more than Windows machines (Finley). Microsoft's acquisition of Xamarin was completed in 2016 and to this day, Microsoft continues to support the framework and makes it available as a Visual-Studio plugin allowing more and more developers to use and be exposed to the framework.

Use

Xamarin.Forms has become popular due to its cross-platform nature. Xamarin's website boasts a possibility of 96% code coverage with certain applications which is a metric to determine how much of an application's code was able to be reused among each of the platforms. This framework lends itself well to applications that require little platform-specific functionality. Also available are the Xamarin.iOS and Xamarin.Android which are platform-specific and better for invoking native behavior while still using C# and the .NET platform.

Languages Used

XAML (Extensible Markup Language)

XAML (often pronounced "zammel") has become the standard markup language for defining user interfaces in almost every application built on Microsoft's .NET framework. The markup language was introduced alongside Windows Presentation Foundation (WPF) in 2006 which was a graphical user-interface framework that could be used to build Windows desktop applications. Windows Presentation Foundation was introduced with the release of the .NET 3.0 framework and was marketed as a smarter API than its popular predecessor, Windows Forms.

C#

C# (pronounced C-Sharp) has become a very popular object-oriented programming language in a relatively short period of time. Microsoft developed C# within its .NET initiative which was an attempt to enter competition with Java. Java at the time had become very popular due to its "compile once, run anywhere" motto. This allowed for Java to be run on all types of machines regardless of its specifications. Microsoft's idea was to design C# to meet the Common Language Infrastructure (CLI). The Common Language Infrastructure was a technical standard for different parts of the programming process to be generalized for different machines without having to be rewritten for certain systems.

Object-Oriented Programming Principles Used

Interfaces

Interfaces are an object-oriented paradigm that leverage inheritance to help define objects. An interface is a structure that contains method signatures. An interface cannot be instantiated like an object can be instantiated from a class. Therefore, classes that inherit a certain interface are responsible for implementing each method defined by the interface in a way that is unique to the object. For a class to inherit an interface, the class must implement each function within the interface.

For this project, each platform needed a class that would copy the SQLite file to local memory for use and then define how to connect to the database. These classes inherited the ISQLiteConnector interface that contains a method GetConnection() which returns a SQLiteConnection object. This SQLiteConnection object is then used to connect to the database and store the data in a collection data structure. This interface plays a vital role in the dependency service which is covered later.

Encapsulation

Encapsulation is the technique of bundling similar data into a data structure. In C#, this is done using classes and structs. This can be done in order to create abstractions, simplify maintenance and

increase re-usability. Access modifiers can be used in order to limit or increase the visibility of properties and methods to other objects.

In order to bind the data from the SQLite database to the mobile application's view, classes needed to be created in order to store the data for access later. Classes were used since structs are typically for smaller, immutable collections of data. Since this class was being modeled after relational data, the class required additional syntax unique to the SQLite drivers which aid in mapping the data to objects. The class members were prefaced by value constraints if there were any and each member was typed similarly to the data in the SQLite database. For example, C#'s "string" would refer to SQLite's "text" and C#'s "int" matches SQLite's "integer". An object is created for each record in the database and stored in an ObservableCollection at runtime in order to dynamically bind data to the application's view. When selected from the view, the object selected is then passed from page to page in order to populate the different views and continue referencing the object attributes in order to interact efficiently with the RESTful web service which will be covered later.

Dependency Service

A dependency service is something that is unique to Xamarin.Forms, but closely resembles polymorphism. This functionality allows shared code to utilize platform specific code at runtime. Interfaces play a major role in utilizing a dependency service. First, an interface must be defined in the shared code. Second, each platform-specific project must implement the interface that was defined in the shared code. Third, the platform specific implementation must be registered as a dependency which allows the dependency service to retrieve the platform-specific code when called. Once all of this is in place, the application at run-time will be able to determine the type of platform the application is being run on and call the necessary function.

Database Use

SQLite

SQLite has become one of the most widely used relational database management systems (RDBMS). SQLite has many unique characteristics that make it a favorable choice when deciding on a database to support an application. One of the most significant characteristics of the SQLite platform is that it does not require a server. This differs from many of the commercial enterprise database systems including Microsoft's SQL Server and Oracle's products. A SQLite database is designed as a single file that can be integrated into any application. This removes the need for maintaining a database engine and the many processes that work in concert to serve the data. These files are identified by the .db3 extension.

Due to the compactness and portability of SQLite databases, SQLite was chosen to be used as the database that would contain the information for each piece of gym equipment and is packaged with the application when installed. Since gym equipment is costly and cumbersome to replace, the database would only need to change on a limited basis at which point an update can be released. The database file used to store the equipment info contained the following attributes: equpID, equipName and imageURL. As opposed to storing images on the device and increasing the size of the application by packaging unnecessary data, an imageURL was used to locate the image on a web server that hosted the images. The Data Definition Language (DDL) used to generate this table is given in Fig. 2 below:

```
CREATE TABLE "Equipment" (`equipID` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
                          `equipName` TEXT NOT NULL,
                          `imageURL` TEXT )
```

Figure 2: SQLite DDL used to create Equipment table

MongoDB

NoSQL, sometimes referred to as "not only SQL" or "non-SQL", databases have become very popular with large-scale applications where the data may not be structured to fit a strict schema. These databases stray away from traditional relational databases with tabular constructs. MongoDB has become a very popular open-source NoSQL database that makes use of collections of documents. With MongoDB,

a database consists of collections which are similar to tables in the relational model. A collection will contain zero or more documents that contain the data and are similar to records. Documents in MongoDB are stored as JSON objects which contain one or more key-value pairs. Each document can have unique fields and it is not required that each document contain the same fields. This allows for flexibility in how the data is stored within the database. An example of a single MongoDB document is given in Fig. 3 below.

```
{
        "_id" : ObjectId("<Generated Id>"),
        "equipId" : 1,
        "equipName" : "Free Weight Bench",
        "status" : 1,
        "imageURL" : "freeweightbench.jpg"
}
```

Figure 3: Example MongoDB document in Status collection

MongoDB was an ideal choice for the database that would support the back-end of the web server that not only hosted the images, but also played a vital role in the checking on the status of a piece of equipment and tracking reported incidents. A database titled "gymapp" was used to store a collection titled "Status" which stored documents containing information for each piece of equipment. Each document used the exact equipment ID, name and imageURL that is found in the SQLite database to allow interactivity between the two and so the supplemental web application can reference the same images. Documents stored in the collection contained an additional "status" field which is stored as an integer that acts as a Boolean data type. Another collection was also created for the incidents that are reported by the users. This collection was titled "tickets" and each document contained the following attributes: ticketId, equipName, dateReported, dateResolved, userDesc and imageURL. Additional attributes could be introduced in order to generate more useful reports. Seguin claims that "document-oriented databases are probably the most similar to relational databases" (31) in regards of modeling your data which makes MongoDB a good choice for certain applications.

Object Relational Impedance Mismatch

One of the greatest challenges with object-oriented programs that use databases as a backend is the concept of object relational impedance mismatch. This problem arises from the practice of object-relational mapping (ORM) which is when data is coerced into data types in object-oriented programming languages. Mapping the tabular data to data types becomes difficult since the data types in the database engine are almost never identical to the programming languages data types. Also, since object-oriented languages utilize the defining technique of inheritance, the data in a tabular model may be broken apart based on the inheritance hierarchy. Relational data is usually meant to define a single record and breaking this information apart eliminates the definition of the tuple.

Luckily, this difficulty is almost non-existent for NoSQL database systems. One of the key characteristics of NoSQL databases is the abandoning of the traditional relational formatting of data for other structures of data.

RESTful Web Service

A RESTful Web Service is a service that allows for data to be served on the internet. This is usually done by one system being set up to accept requests from a client and then serve a response that contains the necessary data in the format needed. A well-formed web service should support all major HTTP methods such as POST, PUT or DELETE requests. The web service can also respond with a confirmation that the transaction took place on the server. Since these act as an API for a site, the interaction is usually hidden and not seen by the user. RESTful web services are commonly used as backbones for popular sites since they are lightweight.

Node.js and Express

Recently, Node.js has become a very popular environment for web development done in JavaScript. This JavaScript runtime is open-source and is used by many large companies such as Netflix, PayPal and Uber. Prior to Node.js, JavaScript was used for client-side scripting that would be run using the user's web

browser. The framework used to handle the client-server interactions is Express which has become a very popular server-side framework for defining APIs. An Express server defines routes which are URLs that clients can make requests to. These routes also define the expected request method, any actions that need to be taken as a result of receiving this request and the response to send to the requestor. The RESTful web service implemented for this project uses Node.js and which also defines the API for the supplemental web application.

This project makes use of the deployed Node.js server by handling HTTP requests made by the mobile application then returning certain information depending on the request. For example, each time a piece of equipment is selected from the application's list, a popup will be created with a "Report an Issue" button. A user should not be able to use this button should a ticket already exist for the particular piece of equipment. In order to determine whether or not the button is enabled, an HTTP request is sent to the server at the URL of "http://<server IP address>:<server port>/api/equipmentstatus/<equipmentID>" at which point the server handles and queries the MongoDB database for the status. Once the status is determined, the server replies with a "1" or a "0" which the application uses to enable or disable the button depending. This service is used for multiple functionalities and each is covered in depth in the application workflow section.

Application Workflow

The first page that a user is presented with when using this application is a log in screen. Due to privacy constraints, the sign-in functionality is not actually implemented and is simply used as a splash screen. As for now, the application can only report issues within a private network where the RESTful server is deployed requiring the user to be on the same network which was adequate for this project. The sign-in functionality could be implemented by a business in order to ensure that members are the only ones submitting tickets and not those who are briefly connected to the wireless. To continue to the next step, the user simply clicks the "Sign In" button at the bottom of the page given in Fig. 4.

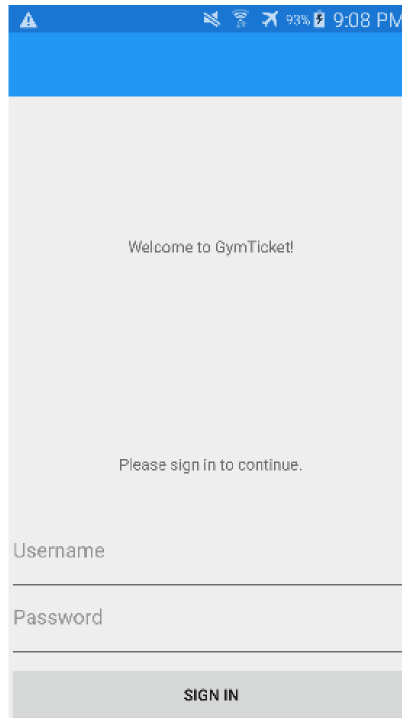


Figure 4: Sign-in page

After passing the sign-in screen, the user is presented with a list of equipment that is in the gym. This list is populated using the SQLite database that is packaged with the application. When the connection is opened, a “SELECT * FROM Equipment” query is executed to get all of the equipment. This query can be modified using .NET’s object-oriented query language LINQ. Once the query is executed, a data collection object is used to store objects of type “Equipment” which the ListView UI element uses to bind data from each object to the application view. Each object in the ListView displays the name, the ID and an image so that users can easily differentiate the equipment. The imageURL member in the Equipment class is used by the UI in order to fetch the associated images added to the view. These images are stored on the RESTful web server so that the application can remain minimal in size so in order to fetch these images, a class method adds the server’s name and port to the URL which the UI references. Some take time to download due to size with the largest image taking around 4 seconds on average to load. To advance, the user simply selects a piece of equipment to view and/or report an incident for. The equipment list with the associated images is given in Fig. 5.

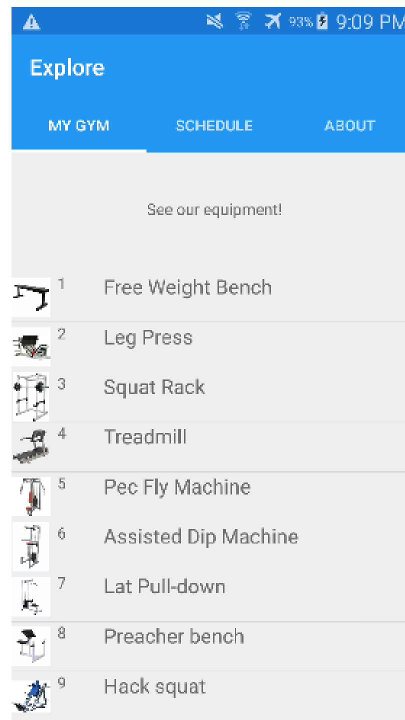


Figure 5: Selectable equipment list

When an item is selected, a new page is generated which displays the image in a larger window and lists additional attributes besides its name. At the bottom of this new page, there is an option to "REPORT AN ISSUE". This button only enabled under two conditions: if the application is able to connect to the RESTful web service and if there is not already an open ticket for the piece of equipment being viewed. If a user is on the network and no ticket exists for that piece of equipment, the button is enabled allowing the user to click it. This was done so that multiple tickets cannot exist simultaneously for the same piece of equipment and so that users cannot invoke undefined behavior off the network. To accomplish this, upon creation of the new page, the application sends an HTTP request to the server at the URL "/api/equipmentstatus/<id>", where "id" is that of the equipment in question. First, the server will parse the URL in order to get the ID of the equipment and then queries the MongoDB database's "Status" collection for the status of the equipment. A 1 is returned to let the application know that the equipment is currently considered working and a 0 should it be considered broken and a ticket exists. The user can

simply use this page for information or they can advance by clicking the “REPORT AN ISSUE” button. The enabling or disabling of the “REPORT AN ISSUE” button is given in Fig. 6.

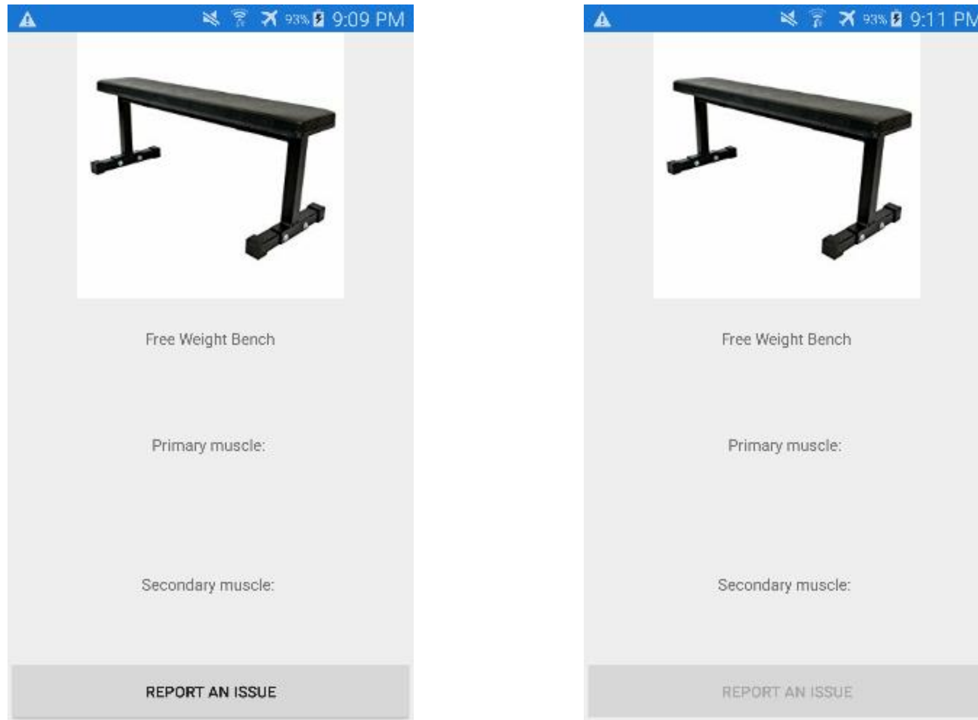


Figure 6: The issue button's usability depending on the equipment's status

Should the user decide to report an issue with the equipment, a new page that contains a form will be loaded for the user. This form includes text input as well as the option to take a picture with the device’s camera. In order to add a textual description to the form, the user simply selects the text area at which point the device’s default keyboard appears for input. To add an image to the form, the user selects the “Take Picture” button which initiates a multi-step process. First the application checks that a camera is available and that the application has permission to access the camera. This permission is granted by modifying an XML document that is checked by the device at deployment. Second, the application takes the image and stores it in a file object. The image is temporarily stored on the device and is deleted when out of scope. Next, the image is loaded into the image UI object in the form so that the user can see the image taken and the complete form prior to reporting. Selecting an image from the device’s storage was not implemented in order to dissuade users from posting old images that may or may not be relevant. Once the user has

appropriately filled out the form, they can choose to submit the form using the "Submit" button. The information on the page is packaged into a multi-part form and then sent over the network to "/api/reportIncident/<id>" and the response is parsed to determine if the action was successful. An example incident form and the prompt on success are given in Fig. 7.

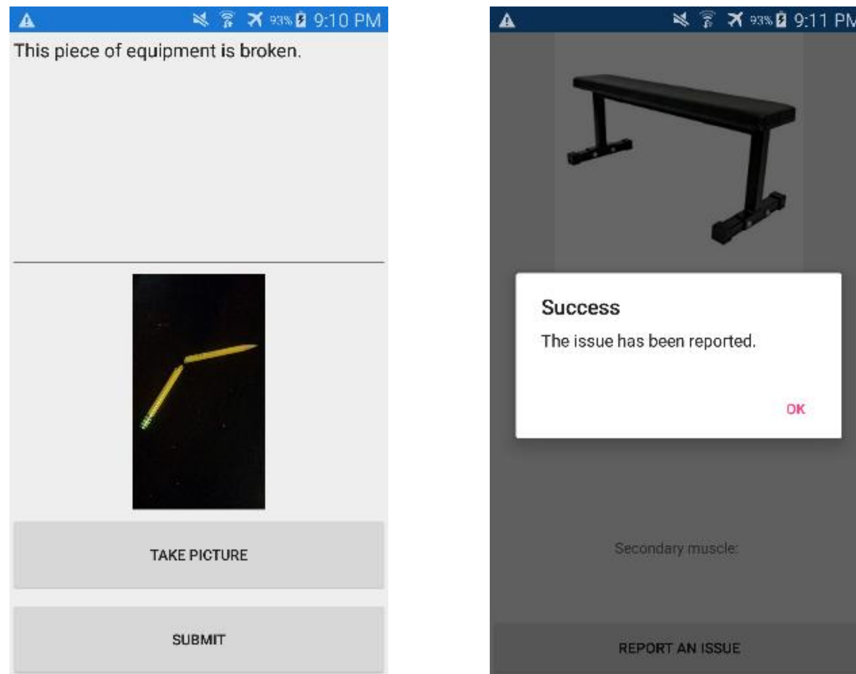


Figure 7: Example form and confirmation of submission

Multi-part forms are used for sending data over the internet that may be more than text. The RESTful web service receives the multi-part form and then decodes the form based on the encoding designated by the HTTP request. Once the form is decoded, the server parses the request and searches for the necessary fields contained within the body of the request. When an HTTP request is received by the server at the particular URL, a ticket is generated in the MongoDB database's "tickets" collection in which the ticket receives an ID in order to uniquely identify it. This ticket ID is generated and maintained by the MongoDB server as an autoincremented value. Once the ticket is created and stored in the database, other attributes are added including as the name of the equipment in question and the date the ticket was reported. At this point, the body of the request is parsed and if the userDesc and image field of the HTTP request is not blank, an attribute is added for user's comments as well as the image's URL on the hard-drive. The

image is fetched from the body of the request and is stored on the hard-drive of the machine that the server is running on. In order to dynamically bind the image to the table in the supplemental web application, the image file on the hard-drive is renamed to that of the <ticketID>.jpg where ticketID is the ticket that the image belongs to. Once all of the database insertions and updates are made, the server sends an e-mail to the administrator's e-mail address mentioning that an issue had been reported and the ID of the equipment it was reported for. An example e-mail sent by the server is given in Fig. 8.

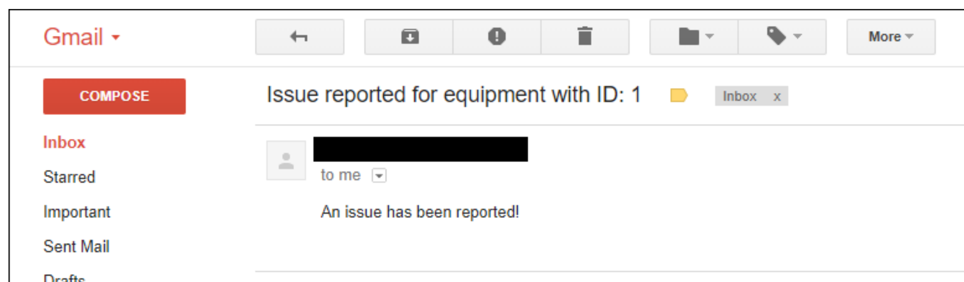


Figure 8: E-mail sent by server when issue is reported

Supplemental Management Web Application

The Xamarin.Forms application allows for users to efficiently report issues with gym equipment, but it does not offer a management aspect for those who will be servicing the tickets. Therefore, a web application was created using the MEAN Stack to display all of the equipment, tickets and allow an administrator to act upon each ticket.

MEAN Stack

As mentioned before, Node.js has become the backbone for many enterprise applications. The MEAN Stack is a software stack that utilizes MongoDB, Express.js, Angular.js and Node.js in order to create client and server-side code using solely JavaScript. The only framework that the web application introduces to the project is Angular. Angular is specifically designed for developing an application's front-end by applying the model-view-controller (MVC) architecture. This architecture makes data binding and handling user input relatively simple. To make this happen, each page makes use of a controller which interacts with the model in order to create a view for the user.

Functionality

One of the main features of the supplemental web application is the ability to see a list of the equipment as well as check the status of each piece of equipment. The table used for this page displays the equipment's ID, name, a link to an image and the status. The list and statuses are acquired using a controller to query the data from the server and then bind it to the web page. This can be used to easily create HTML tables without having to define each row individually as Angular provides a function that allows a portion of HTML to be repeated for each record returned by the query. Attributes of each object are accessed by the notation {{ object.attribute }} which can be used to populate the table and was used as a method to generate unique URLs which created the unique links to each image. Also, the status is shown as red or green which is easier to read than words for each record. This is accomplished by using conditional styling which can be done using Angular in the HTML. Each object's "status" attribute is checked and Angular populates the background color of the table element appropriately. The view created and conditional styling by Angular is given in Fig. 9.

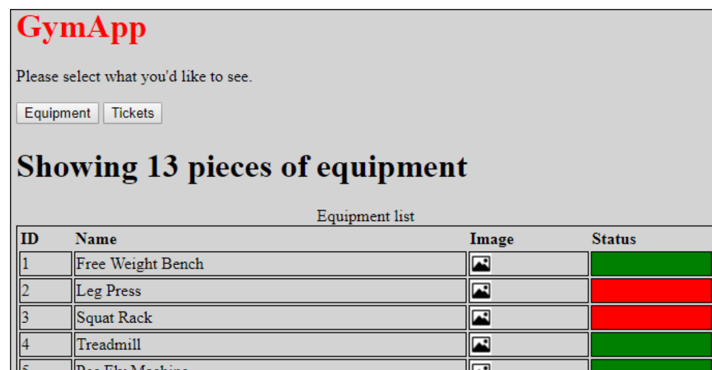


Figure 9: Equipment list as seen in web application

The most important functionality is the ability to check a list of tickets submitted. Each ticket listed contains its ID, the equipment it was reported for, the date it was reported, a date at which the ticket was resolved, the comment submitted by the user, a link to the image provided by the user and a button that allows the administrator to close the ticket when ready. The table is generated in the same fashion as the equipment table using Angular to query the tickets from the MongoDB database and is given in Fig. 10. In

the last column of the table, there is a button that allow users to close the ticket. Conditional styling is used again to style the button to only be visible if there is no resolution for the ticket enforcing only one resolution per ticket. On clicking the "Close" button, a form is created for the ticket which has input for the name of the user who resolved the issue as well as the actions taken and is given in Fig. 11. Once the form is complete, the user can submit the form at which point an Angular controller takes the form and updates the ticket in the MongoDB database. A resolution date is added at the time of submit which officially closes the ticket and hides the button for closing the ticket.

Showing 4 tickets

Ticket list

ID	Equipment	Date Reported	Date Resolved	User Comment	Image	Interactive	Close Ticket
1	Free Weight Bench	Fri Mar 02 2018 18:35:10 GMT-0500 (Eastern Standard Time)	Fri Mar 02 2018 18:35:10 GMT-0500 (Eastern Standard Time)				
2	Leg Press	Fri Mar 02 2018 18:35:10 GMT-0500 (Eastern Standard Time)					Close
3	Squat Rack	Fri Mar 02 2018 18:35:10 GMT-0500 (Eastern Standard Time)					Close
4	Free Weight Bench	Sun Mar 04 2018 21:10:57 GMT-0500 (Eastern Standard Time)		This piece of equipment is broken.			Close

Figure 10: Ticket list as seen in web application

Resolution for: Ticket 4 (Free Weight Bench)

Name:

Actions Taken:

Submit

Figure 11: Form offered to user when closing ticket

NuGet and npm

This project would not have been possible without the use of both the NuGet and npm package managers. NuGet and npm are both package managers that connect to a content delivery network (CDN) in order to install open-source software packages and libraries to use with your project. Microsoft supports and provides the NuGet package manager as a Visual Studio plug-in while npm is used with the Node.js environment. CDNs such as these have become very prominent in the development community as it accelerates development with the help of code re-use.

Conclusion

Native vs. Cross-Platform

I chose to use a cross-platform framework since I was the sole developer on this project and so that I could be exposed to multiple platforms to increase my knowledge base of each. As a sole developer with no experience creating applications for either Android or iOS, I was able to create an application that could be used on both. The development process took about seven months which might have only been enough time to learn one platform. Also, I was very familiar with C# and the .NET framework at the time this project was being proposed which also led me to the Xamarin.Forms solution.

Even though this project was able to prosper from a cross-platform framework, this method is not always ideal for developing a mobile application though. Mobile developers find themselves in one of two camps when it comes to choosing the method for creating a mobile application, either native or cross-platform development. Native applications use a specific platform's software development kit (SDK) alongside the programming language best-suited for the operating system. For example, a native application for Android would be written using Java while native iOS applications would be written using Swift or Objective-C. Rami Assi, a software engineer at Etohum, does a great job of arguing both sides in his article titled "Mobile app development: Native vs cross-platform vs hybrid" which also emphasizes the large market that mobile applications have become by citing sources that expected the market to be worth $77 billion by the end of 2017. A native approach is favored for many reasons including access to all features provided by the operating systems well as better performance and use of resources. There are still downsides to developing native applications which include the need for a development process and/or team for each platform as well as a very deep understanding of the targeted system. Cross-platform development seems favorable when it comes to cost as it only requires a single development team and code can be re-used fairly easily. Where cross-platform falls to native is the ability to use the features provided by the operating system to their full potential. These features must be accessed using plugins or a method specific to the framework that is ultimately not as efficient with the device's resources.

Expectations for Xamarin.Forms

Xamarin.Forms was a very useful tool for creating cross-platform tools quickly and easily. The framework could easily become very popular among those who wish to develop mobile applications using the .NET framework which is gaining popularity with those who favor object-oriented development. In October of 2017, Microsoft ‘s Corporate Vice President in the Operating Systems Group Joe Belfiore announced that the Windows Phone platform has been moved into a maintenance state and will not receive any new features and current users will only see security updates and bug fixes (Warren). In response to this move, the Xamarin team may no longer continue support for the Universal Windows Platform in which case they will be able to focus all of their efforts on Android and iOS. Once the framework supports iOS and Android development as well as the other available frameworks, Xamarin could gain popularity even quicker than expected.

Personal Experience

This project was very enjoyable and enlightening. I learned about mobile development, the two most popular mobile operating systems, embedded databases and much more. The application met the standards I set at the beginning of the project as well as the deadlines put in place by myself and the Honors College. Adding the supplemental web application was a thought that came later in the development process and not only added value to the project, but allowed me to continue practicing web development in JavaScript. The Xamarin.Forms framework was a technology that I was unfamiliar with and was slightly challenging to learn and build on especially when attempting to interact with the embedded database and utilize native features. Apart from the difficulties, I was able to start and finish a relatively useful mobile application that can be used on three different platforms which was a neat experience.

Technical Requirements

The following equipment was used to produce this application:

HP Envy TouchSmart m6 Sleekbook (Development machine)

- Memory – 8 GB
- Operating System – Windows 10 Home (64-bit)
- Processor – Intel Core i5-4200U (1.6GHz base frequency, 2.3GHz max frequency)

Mac Mini – Mid 2010 (iOS build machine)

- Memory – 4 GB
- Operating System – macOS Sierra Version 10.12.6
- Processor – Intel Core 2 Duo (2.4GHz)

Samsung Galaxy S4 (Android device used for debugging)

- Operating System – Android Lollipop (5.0.1)

iPhone 6 (iOS device used for debugging)

- Operating System – iOS 10.2

Microsoft Visual Studio Enterprise 2017 (IDE used for development)

- License acquired via The University of Akron's Microsoft Imagine subscription

MongoDB

- Community Server for Windows Server 2008 R2 64-bit and later with SSL Support
- Version 3.4.6

Node.js

- Version 6.11.0

DB Browser for SQLite

- Open source tool used to visualize, create and edit SQLite database files
- Version 3.9.1
- Licensed under the Mozilla Public License as well as the GNU General Public License

Git was used as version control for this project. The source code for the mobile application can be found at https://github.com/ermerryman/HonorsProj. The web application source without image and package files can be found at https://github.com/ermerryman/HonorsProjServer.

References

Assi, Rami. "Mobile app development: Native vs cross-Platform vs hybrid." LinkedIn, 11 Oct. 2017, www.linkedin.com/pulse/mobile-app-development-native-vs-cross-platform-hybrid-rami-assi/.

Britch, David. *Enterprise Application Patterns using Xamarin.Forms*. Redmond, Washington, DevDiv, .NET and Visual Studio produc teams, 2017.

Finley, Klint. "Microsoft Buys Xamarin to Expand Its Empire Beyond Windows." Wired, 24 Feb. 2016, www.wired.com/2016/02/microsoft-expands-empire-beyond-windows-xamarin-buy/.

Petzold, Charles. *Creating Mobile Apps with Xamarin.Forms*. Redmond: Microsoft Press, 2016. Web.

Seguin, Karl. *The Little MongoDB Book.*: Syncfusion Inc., 2014. Web.

Warren, Tom. "Microsoft finally admits Windows Phone is dead." The Verge, 9 Oct. 2017, www.theverge.com/2017/10/9/16446280/microsoft-finally-admits-windows-phone-is-dead.