6-1-2017

# Developing Open Source Alternatives to Proprietary Software

Howard Passmore
*Western Oregon University*

# Developing Open Source Alternatives to Proprietary Software

By
Howard Passmore

An Honors Thesis Submitted in Partial Fulfillment of the
Requirements for Graduation from the
Western Oregon University Honors Program

Prof. Mitch Fry,
Thesis Advisor

Dr. Gavin Keulks,
Honor Program Director

June 2017

## Table of Contents

# Abstract

The invention of the smart phone has provided people the ability to share information instantly. One major development in the last decade is the ability to report problems such as potholes, malfunctioning lights, broke street signs, or graffiti in real time with exact GPS coordinates from a smartphone. Many cities and towns have contracted out to companies to build mobile apps that allow their citizens to report problems wherever they are. These applications often come at hefty prices. The focus of my thesis will be to develop an open source version of a reporting app that is being sold by SeeClickFix Inc. This will allow towns, cities, businesses, and colleges to adopt a real-time issue reporting application while minimizing costs. I will also discuss why adopting open source software can be beneficial.

## Introduction & Inspiration

Many cities and other large groups have a difficult time tracking infrastructure problems within their jurisdictions. For years, governments have provided the ability for citizens to write or call in to report issues like potholes, rusted out cars, clogged drains, and more. Some cities have set up websites where people can submit issues. People tend to not use these as they must go out of their way to do so. It is also difficult to get accurate location information and descriptions through these kinds of services. Anyone who spends a lot of time in a big city can tell you that there is still a lot of work left to be done.

The city of Detroit Michigan turned to mobile phone technology to help track and resolve problems around their city. In April 2015, the city contracted with SeeClickFix, a software company from Connecticut, to build a mobile phone application that people could use to send in reports of various issues around the city [15]. A report made from the Improve Detroit application contains the problem type (rusted car, broken pipe, etc...), a description provided by the person making the report, exact GPS coordinates based on the location of the phone, as well as a picture if the user decides to provide one. As of April 2017, the application has helped Detroit resolve more than 67,000 issues [16].

For years, cities have used their 311 lines, which are numbers people can call to get or report information, to let people report infrastructure issues. According to SeeClickFix's website, for a city to answer 10-15 calls per week in a 311 system costs $2,291 annually [17]. Cities then need to pay someone to locate the issue without exact GPS coordinates to verify the problem, costing an average of $14.41 per hour for a city works employee. On top

of that, cities must employ more community relations specialists to handle repeat callers and duplicate requests which costs $22.67 an hour (as of May 1, 2016) according to the Bureau of Labor Statistics [18]. Having the 311 process automated with mobile applications allows cities to verify and fix requests faster, provide feedback instantly to citizens who report problems, and be provided exact locations and descriptions of the problem.

Applications, like those from SeeClickFix, still come at a price however. A typical pricing model includes a hefty upfront cost to purchase software or become a registered user on the provider's site. Customers then need to spend money on ongoing maintenance and contracting costs, which includes either paying to store data on another company's servers or hire an employee to manage the application in house. While costs for operating 311 systems will never go away, utilizing open source technologies can help reduce costs tremendously. City planners must consider the cost and time needed to verify software functionality, time needed to gain adoption of the new platform, and accept the fact that all that time and money could be a waste if the project does not take off.

When I first read about the Improve Detroit application, I considered the possibility of developing an open source alternative that government and other large groups could use. Doing so would allow me to build valuable software engineering skills while assisting government and other groups in moving issue tracking into the twenty first century. At first, my plan was to create a software suite, whose name is CommunityCommunicator, that provided much the same functionality as SeeClickFix's solution. Governments and other groups would host an installation that would take in reports for their area of responsibility. Therefore, the original plan involved creation of an Android SDK, or Software Development

Kit, that cities could use to create their own mobile apps. After some consideration however, I scrapped this idea since it would require extra effort and expertise for each group to implement a custom solution. It also would slow down adoption, as people who often travel between multiple cities would need to have more than one application installed to report issues they find. I decided it best to try and create a universal program that could be hosted in the cloud where anyone could report issues for any location.

# Why Open Source

**Open vs. Closed Source**

Whether a piece of software should be open sourced is a tricky question to answer. What led to the open sourcing of the CommunityCommunicator project was the fact that it's purpose is to be as cheap to implement as possible. Since there is no charge to implement or extend the project, any entity will be able to take the code, customize it if needed, and deploy. There are benefits and detractions to this decision.

The first benefit, as previously discussed, is cost. A closed source solution would require anyone who wants to utilize the application to purchase it then pay for ongoing storage and support. There could be specific use cases that would cause a city or college to need to modify source code or data models to better match their needs, which can be costly for those purchasing a closed source system. When an application is released under an open source license, generally anyone can take it, modify it if need be, and deploy the solution for use. By making the CommunityCommunicator suite open source, the cost of entry for cities will be very low. If they later decide to keep, extend, or scrap the project there will be little to no lost money. With such a low barrier for entry, my hope is that even communities with little money available will try the suite out and adopt it.

As previously stated, anyone can modify an open source project to match their needs. If administrators decide they need new data models, a different security approach, or location restriction they are free to add these as they please. If a bug is discovered in the source, anyone can report and patch the bug, which can result in a fast turnaround time on needed fixes. The typical workflow to obtain changes for a closed source application

requires requests to the owner of the application and time for development and testing. This can only happen if the company is still in business and the original purchase contains a maintenance or service clause (or one was purchased separately). If the company goes out of business, the city will be forced to live with the issue or purchase another project. Without a service clause, whoever requests the patch will be charged a large sum of money.

Even if no modification is needed, the ability to view source code can help build trust in an application. When someone purchases a closed source application, they must take it on faith that everything is implemented properly. The purchaser has no way of knowing about major flaws in areas like security, which can end up being costly. An open source application does not have this detraction. Any person with enough technical knowledge can view the source of the application to determine risk to their organization if the software is utilized. This can however be an issue, as even malicious users can review the code. If a malicious user finds a bug, they can exploit it until such time that someone else catches it.

Issues with open source are often mitigated by the fact that anyone can make changes. A software engineering company must take the time to review bugs and features, find time for engineers to make a patch, test the patch, then make a release. This cycle can be either short or incredibly long, depending on the company. All it takes for an issue to be fixed in an open source application is someone willing to put in the work. Many software engineers, myself included, take time to contribute to open source applications and libraries to help improve them. Once someone finds an issue with the source code of an application, they can fix it themselves and request their patch be combined with the original code. If the

person who finds the issue is not able to fix it, they can submit a report to the project and any contributor can fix it.

Every day different stories are published about breaches of systems. IT professionals are constantly trying to improve the security standing of their business. When a company purchases a closed source application, they must take it on good faith that the developers did not accidentally introduce any major security holes. While this can happen in an open source project, potential clients can gather technical people to review the source to determine if they want to introduce the new software into their environment. If the code has problems, the user will know and can determine mitigation techniques, if they choose to use it at all. The only thing a company loses out on is liability for issues. When purchasing a piece of software, a clause can be put into the contract that says if the software causes loss or profits or reputation, the developers of the software must pay a fee to the purchaser. No such clause can be made with something that is open sourced.

Even though open source technology is free, money can still be made from its existence. Companies often keep projects closed source as they believe that to be the only way for them to make money. While an open source project won't make money from the code itself, there are many ways to make money from open source. Companies can charge consultancy fees, data storage and cloud hosting, and sell extensions and additional services. Some of the best-selling software is made via extensions of open source. The most commonly used operating system in the world, Google's mobile Android OS is a completely free and open source project. Google can make money by providing apps and services for sale as well as showing ads for their products to users. By open sourcing code, companies

can also receive free engineering help and get other companies hooked on their product, which makes it much easier to sell proprietary technology down the road.

It is worth mentioning that not all free applications are open source. The term freeware refers to applications that are free to use. A common one that many people are familiar with is CCleaner for Microsoft Windows. The program is free to use for anyone, but only Piriform, CCleaner's creator, can tell you how it works. Many freeware applications are in fact closed source technology. Something only becomes open source when the code itself is released under an open source license, making it free for anyone to review, modify, and use.

# Licensing, data models, and API

**Licenses**

The first decision that I felt I had to make deals with the licensing of the project. Open source software is open for anyone to see, but there are different rules on how it can be used depending on how it is licensed. Organizations such as Apache, MIT, and GNU all have developed different licensing models. Some licenses, like the MIT, allow unconditional use in any way that a user would see fit as long as the license accompanies the derived software. Apache 2.0, a more stringent license, requires any changes made to the code to be documented as such. GNU GPLv3 is what is known as a copyleft license. According to GNU.org, “Copyleft is a general method for making a program (or other work) free (in the sense of freedom, not “zero price”), and requiring all modified and extended versions of the program to be free as well.” [5].

The distinction between copyleft and other licenses involves one key difference: the ability to commercialize the product. Most open source licenses allow for redistribution for monetization. Any software released under an MIT or GPL license can be taken by a third party and, if a copy of the license is provided, be sold to anyone. Software released under an MIT license, a non-copyleft code base, can be modified, copyrighted, and sold as proprietary software. Anyone who wishes to modify code released under GPL or other copyleft licenses must release their modifications under the same license as well. This prevents companies from releasing an extension of a copyleft product as proprietary [5].

Ultimately, I decided that the code for the CommunityCommunicator suite should be released under the MIT license. For most of the development time of this project, I was

planning on releasing under GPLv3, as I felt this would allow the suite to add features not thought of before and allow those features to be available, free of charge, to anyone wishing to use the suite. However, it later occurred to me that my goal should be as much adoption and cooperation in the project as I could get. I felt that the best way to achieve this would be to use the least restrictive license I could, even if it meant it could result in a closed source adaption.

**Data Models**

Modeling the data for the CommunityCommunicator project was an easy task. My initial goal was to make sure the data models could easily be added to or extended. The three models that were needed for the barebones CommunityCommunicator project are Users, Reports, and Comments. I decided that I wanted to perform relational operations on the data, so I knew that the models had to be compatible with a relational database system. I settled on the use of MySQL, which is a free to use relational database system which has widespread support. The top-level item in a relational database is called a database. The database holds tables, which define the names and types of data that are stored in them. Each table is used to describe one model, so in the case of CommunityCommunicator, the database, has three tables: User, Report, and Comment. Tables store items called rows (also referred to as records), which are actual pieces of data. The user John Smith would be a record in the User table.

The central model of the system is User. Everything in the CommunityCommunicator system requires interaction from a logged in person. I decided to, for now, exclude anonymous reporting to mitigate misuse of the system. The User object contains 5 fields: ID

(integer), email (string), password (string), created date (date string), and active (integer). To create a user, a person supplies both an email and password to the REST API. While all fields are required, the server generates ID, created date, and active since they are used for administration purposes and not user interaction. Both ID and email are unique for this model, meaning no two records can have the same value for either. All data models need a unique key which will never change, which in this case is ID. ID is an auto incrementing integer, meaning the database itself fills the field. The first user created gets 1, second gets 2, third gets 3, and so on. Emails are unique since they shouldn't be duplicated as they are used to login. This will allow users to change their email while preserving their accounts. Both the created date and active are both auto added by the server. Active is used to track whether the person has deactivated/deleted their account. On creation, Active is assigned 1 to indicate. When the user deactivates, the Active is flipped to -1. Upon reactivation, the flag is flipped to 1 so all their data is preserved. The number 1 was picked arbitrarily for active and -1 for inactive since it is the opposite of 1.

The next model that is important is Report. It contains 8 fields: ID (integer), ReporterID (integer), Date (datetime), Longitude (decimal), Latitude (decimal), Description (string), LocationInfo (string) Image Location (string), and Active (integer). Of these, ID, Reporter, Longitude, Latitude, and Description are required. Image Location is optional as reports are not required to contain images. When a user creates a report, they supply the longitude, latitude, description, and optionally an image or additional location info. How image storage is handled will be discussed later. Active and Date are handled the same as in Users. The server will then generate an ID for the report, add the reporting user's ID to the

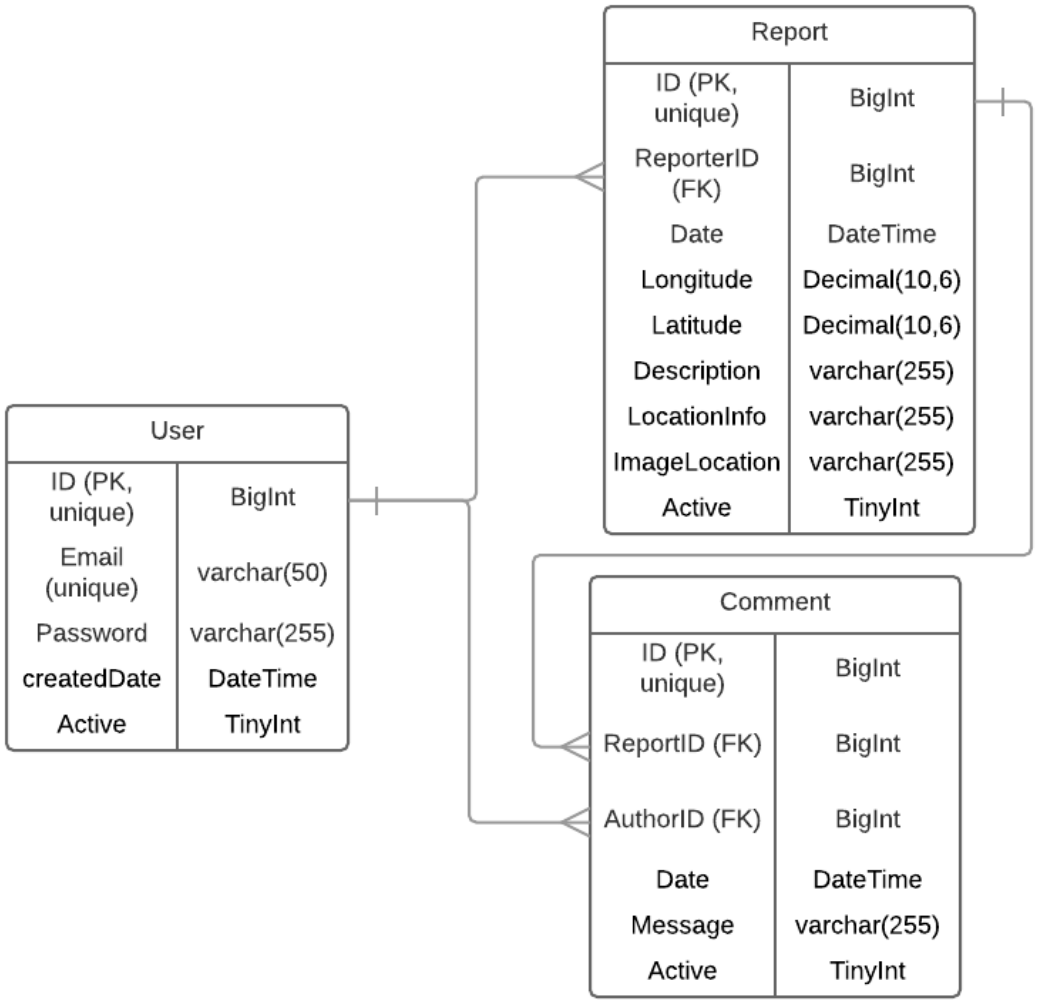
ReporterID field, then store the report in the database. In this case, ReporterID is what is known as a foreign key. A foreign key is a column in a relational database that uniquely identifies a record or row in another table of the database. This allows someone viewing the report to retrieve information about the person who created it. If a user deactivates their account, all reports they make remain active so those reported issues can still be resolved.

The last model is Comment. This model stores all the information that a user would need to leave comments on existing reports. Comments allow someone to provide additional information to a report made by themselves or another person. A comment object contains five fields: an ID (integer), a ReportID (integer), an AuthorID (integer), Date (datetime), Message (string), and Active (integer). ID is the only unique field for a comment as a user can make multiple comments on the same report, and a report can have multiple comments. ReportID and AuthorID are both foreign keys; ReportID corresponds to the report the comment is placed on and AuthorID corresponds to the ID of the user making the comment. The reporting user only supplies the message when making a comment. As before, ID is auto incremented by the database on creation. Active and Date are handled the same as the previous models.

All the models are created to be easily used by any future models that need to be created. If a future model needs any of their information, that model would just need to store the ID of the user, report, or comment as a foreign key in their table. These models only represent the bare minimum that is needed to create an application to report and store information about infrastructure issues. There are possibly more models that will be added on to this project later. I did not want to commit to doing work that I could not

deliver on, so I thought it best to make a bare bones system work well as opposed to making a complex system that had a lot of problems.

The following chart is an entity diagram showing the relationships between the different models in the CommunityCommunicator system. The first column is the name as it appears in the database. The second is the exact data type the database uses. The lines between them refer to items that reference each other, showing how ID in User is the same as ReporterID in Report and AuthorID in Comment.



**API**

Next, I decided how exactly the data models would be exposed to the world for interaction. The way this is done is using a Representational State Transfer (REST) web based Application Programming Interface (API). Consumers of a RESTful API make stateless calls that amount to them saying “Get me this, put this here, delete that”. Each call made to the API is to either create, modify, get, or delete information. RESTful APIs expose endpoints, or URL paths (www.google.com is an example of a URL path) that, when accompanied by an HTTP (HyperText Transfer Protocol – how information is exchanged through the internet) verb, tell a web server to do something. A good example is when a person types https://www.google.com into their web browser. They are really telling their browser to perform a GET operation on the root directory, denoted by /, on the google server. Typing https://www.google.com/coolStuff tells the browser to perform a GET action on whatever google says is at /coolStuff.

The CommunityCommunicator server application was designed to expose several endpoints that are used to access a database described in the Data Models section. This program only concerns itself with the four HTTP verbs: GET, POST, PUT, and DELETE. RFC 7231 by the Internet Engineering Task Force defines what HTTP verbs are supposed to be used for given operations [6]. GET is used to denote that the action is to retrieve something, whether that something is a web page or piece of data. PUT is used to update or create a resource when the user knows the storage location. For example, if a user is creating a dog object and the dogs are denoted by name, they could use PUT https://storeDogs.com/Henry to put a dog named Henry. If they do not know how the dog

will be stored or denoted, they would instead use a POST to https://storeDogs.com/ and the server itself would figure out where to put the dog from there. If they then wanted to DELETE the dog, they would call DELETE https://storeDogs/Henry. The area after the slash denotes the unique resource that is being deleted. RFC 7231 does not state any data needs to be deleted from a DELETE call. It is common for resources to just be deactivated so they appear deleted [6].

Before the endpoints are discussed, a few terns should be covered. A query string is an optional piece of data that can accompany a call. Query strings come after a question mark at the end of a URL path. In the previous example, getting all dogs that are brown could look like GET https://storeDogs.com/?color=brown. The API endpoint would need knowledge of all possible queries. It will look for those it knows and disregard those it does not know. A path variable is a non-optional variable that is in the actual path of the call. For example, a programmer would expose all the dogs owned by a user in this manner: GET https://storeDogs.com/user/{userId}/dogs. A word between curly braces is replaced by the actual variable. To get all of user 1's dogs, a user would call GET https://storeDogs.com/user/1/dogs.

The following list represents the minimal REST request endpoints the CommunityCommunicator application needs to function:

- GET /user/{userID}: Gets the information for the user represented by the passed in userID.
- GET /user?email= : Gets the user who has the passed in email, if they exist. Email in this case is a query string.

- GET /report: Gets all the active reports on the system.
- GET /user/{userID}/report: Gets all the active reports made by the given user.
- GET /report/{reportId}: Gets the details of the report represented by the reportID.
- Get /report/{reportID}/comment: Gets all the comments for the given report
- POST /report: Creates a report.
- PUT /report/{reportID}: Update an existing report. Can only be made by person who created the report.
- POST /report/{reportID}/Image: Uploads an image to a report.
- POST /report/{reportID}/Comment: Creates a comment on the given report.
- PUT /report/{reportID}/Comment/{commentID}: Updates an existing comment. Can only be made by the comments author.
- POST /user: Creates a user.
- PUT /user: Updates the email and/or password for a given user. Can only be made by the user themselves.
- DELETE /user/{userId}: Deactivates the given user. Can only be made by the user.
- DELETE /report/{reportID}: Deletes the report represented by reportID. Can only be made by the user who made the report. Also deletes all comments.
- DELETE /report/{reportID}/Comment/{commentID}: Delete the comment. represented by the commentID. Can only be made by the author of the comment.

A normal HTTP request contains headers and a body. Headers tell both the client and the server important information like what language or country a request is made from, whether the request should be cached, and any relevant cookie or session information. The

body of a request usually contains information the client is attempting to retrieve, create, or update. Note that if all the information the server needs to complete a call can be expressed in a URL path, a body is not required. For example, for the GET /user/{userID} call, no request body is required as the only thing the server needs to know to complete the request is the user's ID, which is already in the URL. However, to create a report, someone needs to provide a lot of information that should not be transferred through a URL, thus a request body is used.

The server component of the CommunityCommunicator application makes use of JSON, or JavaScript Serializable Object Notation, for its request and response bodies. JSON is a standard format for request bodies in a RESTful API. It is lightweight and very easy to understand. Here is a sample JSON object, used to create a report in the CommunityCommunicator system:

```
{
        "Longitude" : -123.456,
        "Latitude" : 23.567,
        "Description" : "Pothole on the corner of fourth and main",
        "Image" : null
}
```

Attributes of JSON objects are denoted by strings. Here, Longitude is an example of an attribute. A colon separates an attribute from the value assigned to it. In JSON objects, strings must be enclosed by double quotes. Note that Image has an assigned value of null, meaning the requestor did not provide one. Also notice how these fields do not line up with the expected values for a report model. Quite often, creation of an object and what is stored are decoupled from each other. In the CommunityCommunicator program, the Image in this request would be an actual picture, but the database only stores a string. As

seen later, when a user makes a report, the image is uploaded through a separate call. Once uploaded, the server stores the place the image is kept.

The only call that does not use JSON is POST /report/{reportID}/Image. This is because sending image data inside of a JSON body requires extra overhead. The call to upload an image has a content type of multipart/form-data. This is the only way to upload raw files to a server without first turning them into another kind of data, which causes larger uploads as well as processing overhead for both the requestor and the server. When uploading a report, a user first sends a request to POST /report. If that request is successful, the server returns a JSON object that has the ID for the given report. The user then uploads the image using that ID. It is possible to send both image data and report information through with one form-data call. Splitting report creation into two steps prevents large image uploads in situations where report creation fails, which cuts down on lost network resources and potential errors.

# Implementation

**Server Architecture**

The easiest decision I had to make was what language the RESTful API would be in. Before starting the project, a friend of mine, Brady Sullivan, introduced me to Golang, also known as Go, a language created and maintained by Google. Go was built to address "the problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model. [19]" Golang has strong support for both concurrency (executing multiple programming threads simultaneously) as well as dependency management (managing external code or projects your code needs). Before beginning to develop the project, I gave Go a trial run and found that the code was shorter and cleaner than Java (my default language) while getting the job done. These conditions make it good for writing server APIs. While it would be stepping out of my Java comfort zone, I decided Golang was the best choice for the project while also forcing me to learn a new language.

As far as operating system goes, I decided to build and test the CommunityCommunicator application on Linux. The main reason for this is that Linux is also a free open source project so there would be no licensing overhead like there would be if I went with Microsoft Windows, one of the other big player in server operating systems. It also helps that it is lightweight, easily customized, and supported by every cloud provider for very cheap. Normally this would mean that I would need to either develop on a machine running the same version of Linux or write all my code on one machine and build from another, a process which can be time consuming given how small the actual code base

is. Luckily, Golang has great support for cross-compilation. This means that, even though Golang creates a program made to run only on the operating system it is compiled for, a developer can compile for any operating system from the same machine. Even though all the actual coding was done from a MacBook Pro, by specifying the target operating system as Linux when compiling, my machine would build a fully functioning Linux binary. This meant I could do all my incremental testing by running locally from my mac, and then when I had a workable product I could deploy it to a Linux machine for testing there.

For deploying and running the CommunityCommunicator software I settled on using Amazon's AWS service. AWS is Amazon's cloud computing department. I settled on AWS because I found the system to be very straightforward to work with. AWS provides a free tier of service for all users which can often be augmented for students which helped me keep costs down during development. Their free tier provides the ability to run one Linux system with 1 CPU core and 1 Gigabyte of RAM, which is plenty for development and testing, 24/7 for an entire year. Normally it is bad practice to not keep a database on a dedicated system. Keeping a database separate removes a single point of failure while making it easier to replicate and backup whole or parts of data. Since the early stages were just for development purposes, I ran MySQL on the same system that housed the server software. Once it came time to finalize the software, I moved the database to its own dedicated system in AWS.

**Server Implementation**

Now that all the design decisions had been made I was able to begin the actual implementation of the system. The first thing that was done was identifying third-party

libraries that would be helpful in writing the code. A library is code that a programmer writes to perform specific tasks that other developers can use. To talk to a database, a program needs what is called a driver. Most relational databases make use of Structured Query Language, or SQL, to take in commands to create or access data. Even still, each type of database like Oracle, Microsoft SQL, and MySQL have nuances that a program needs to be able to navigate through. Luckily a GitHub user had publicly released a MySQL driver for Golang, keeping me from having to develop one myself. The only other third party library CommunityCommunicator server makes use of is “mux router” by the Gorilla Web Toolkit. This library made it easier to link RESTful paths with the functions that would handle them. The standard Golang libraries could take care of everything else that I needed.

To make it so the server could easily be set up by anyone, I decided it best to read in a configuration file in JSON format upon startup of the system. Here is an example of one of those configuration files:

```
{
        "port":"8443",
        "db":{
                "address":"localhost",
                "port":"3306",
                "username":"commadmin",
                "userpass":"Password1@"
        },
        "key":"~/serverKey.pem",
        "cert":"~/cert.pem",
        “secret”:”MySecret@1#”
}
```

The port field denotes the actual port that the server will be listening on for requests. Ports are what computers use to identify how they are talking to each other. HTTP uses port 80 and HTTPS uses port 443. When you type https://www.google.com, you are telling your

browser to communicate with whatever device is at www.google.com using port 443. The db field holds all the information needed to communicate with the database that holds the application's data. The value of db is a nested JSON object as db contains enough related information it is helpful to show it as a standalone object. Address stores where the database is, port tells which port the server will connect to the database through, and username and userpass are the credentials the server uses to authenticate with the database. Key and cert are what the server uses to establish secure communications with clients using HTTPS. Secret is the string that is used to sign all tokens issued to users (discussed later).

The next issue that needed to be tackled is how best to store user passwords. Lately there has been a lot of news articles about leaked or cracked accounts. Often this can be caused by the inability of a site to store their password properly. If you have ever gone to reset your password and the site just emails you your password, that is a strong indication that they are not stored properly. The proper way to store a password is with hashing. Hashing functions take an input, in this case a password, and runs that input through what is called a one-way function, or a function where the input cannot be derived from the output. When a user first creates their account, or resets their password, the password should be ran through one of these functions. The result of the function is what should be stored in the database. Then when a user goes to login, the system takes the provided password and runs it through the same function. If the output matches the stored value in the database, the user is granted access to the system.

Not all hash functions are created equally though. When a system combines a strong hash function with strong passwords, the hashes, even if stolen, are virtually uncrack able. After much research, I settled on the use of the BCrypt algorithm. BCrypt is built to be intentionally slower than most hashing algorithms, which is a good thing. According to the original creators of BCrypt, it is roughly 10,000 times slower the SHA1 algorithm [7, 8]. Under the hood, BCrypt is just executing an internal hash algorithm many times repeatedly [7]. The number of times it is executed is defined by the user. When a program runs a BCrypt function, they pass in a number N that tells BCrypt to run the function $2^N$ times. If the program tells BCrypt 10, then the function executes $2^{10} = 1024$ hashes on the supplied data. If in the future hardware becomes suddenly more powerful, all that needs to be done is increase N until the algorithm is again sufficiently slow. BCrypt also provides a unique randomly generated value called a salt which is combined with the password before hashing. Salting a value makes hashes unique, even if the passwords themselves are the same. This makes hash cracking software crack each hash individually instead of relying on rainbow tables or hash tables, which use pre-computed hashes to crack passwords faster.

Now that password storage was solved I could begin figuring out exactly how calls could be authenticated. The first thing that I thought of would be to have the user supply their credentials on every call. While this would technically work, it would be cumbersome, make the code ugly, and provide more opportunities for credentials to be stolen. A brief search revealed that JWT's, or JSON Web Tokens, could be the solution to my problem [11]. JWT's, or more generally "tokens", store information that a server uses to authenticate a user. When a user first logs in, their credentials are verified and they are issued a token. The

token stores certain pieces of information about the user as well as when the token expires. On each subsequent request, the client provides the token to the server, which is used to determine whether the user is valid and can perform the action they are attempting to. The server cryptographically signs the token to prevent the token from being tampered with [11]. CommunityCommunicator tokens store the user's ID and email as well as an expiration as these are the minimal needed to figure out who the token belongs to. The expiration is a month to prevent the user from being annoyed with frequent logins. Since the data is not sensitive or harmful, this is sufficient for now.

The last problem that needed to be solved was how image storage would work. Nowadays, pictures taken by cell phone cameras can be several megabytes in size. Any storage solution I selected must be able to store and serve these larger images rapidly. The first solution that I tried was storing the pictures as raw bytes in the MySQL database. At the base level, all data in a system is represented in bytes. I figured that all I would need to do would be to take in a picture and store the raw bytes for a report image inside of the report table. At first I tried this, and while the performance was not bad, I could tell that the images were taking too long to load. After a quick google search, it turned out that there is some overhead that I had not considered when it came to storing images in a database. Retrieving lots of data from a database can be a costly operation which can slow down other requests.

After watching a talk and reading a paper by Jason Sobel and several other Facebook Engineers, it occurred to me that, for this application, dedicated filesystem or blob storage would work better than database storage for my use [9]. Web servers and browsers are

built to mainly serve static content [9]. By storing images on a filesystem, the files are stored exactly how they are sent to the requestor. All the server needed to do would be to take the image and return it to the client. During initial development, I found it sufficient to take the uploaded pictures and store them on the server's machine in a specific folder. Once it came time to ready the application to store images for production use, I migrated the application to use Amazon's S3 storage. S3 allows users to store files as static objects on Amazon's servers, where they take care of the backup for you. When someone submits a report, the server takes the image they provide, uploads it to Amazon S3, and then stores the URL Amazon returns as the picture's location. When a user then wants to view the image, the server returns the location and the CommunityCommunicator application retrieves the picture directly from S3.

**Android Implementation**

The main challenge in creating the Android application was the design of the user interface. If the interface was too simple it may be difficult to convey the information that needs to be conveyed. However, if the interface was too complex, users would not adopt the application, which means the CommunityCommunicator suite will not have reached its desired goal to help streamline the 311 process. I decided I should make the interface as functional and intuitive as possible in the first iteration. I decided that on load, the user will be presented with a map centered on their current location. On that same screen, there would be three buttons, one labelled Report, one labelled My Repots, and one labelled Re-center Map. If the user is logged in, Report would direct them to a page where they can file a report. If they are not logged in, they will be taken to a page where they can either create

an account or log in. Once logged in, the user would be redirected to the Report screen to file a report. My Reports makes the map display only the reports made by the user. If not logged in, My Reports redirects the user to the previously mentioned login screen.

The report screen has fields for description and location details, as well as the option to provide an image. In the background, the application collects the exact longitude and latitude. Location details is optional and is only used to augment GPS location, just in case the GPS coordinates could mean multiple locations, such as in a multi-story building. Once a report is submitted, the user is once again sent back to the main screen where their report shows on the map. The map itself will be populated with all active reports represented as pins on the map. If a user selects a pin on the map, they are taken to a report details page that shows the description, longitude, latitude, an image if one was provided, and a button to view comments. If the user is the creator of the report, a delete button will also show on the repot details to remove the report. The comments screen lists all comments on the report with the option to add one of your own.

Since the Android application is rather simple with little logic, the only technical issue that had to be tackled beyond learning Android development was the retrieval of location information. GPS works by triangulating your location based on proximity to different satellite systems. The longer the system has been polling, the more accurate the information is. At first, I thought it would be sufficient to start polling for GPS information when the user entered the reports screen. This caused two issues. The first is that, when it worked, the points that were being reported were not accurate enough. The second is that the Android OS is not very fast about retrieving location information. Requesting GPS

coordinates takes several seconds, so if they are only requested when the report screen is brought up, the user may be forced to wait to make a report. To avoid these problems, I decided to begin polling for location data as soon as the application started up and store it in a global value. Then, in the background, the CommunityCommunicator application updates the global location value every 3 seconds with new GPS coordinates.

## Extensions

The CommunityCommunicator suite in its current form is the barebones needed for an application used to track and solve city infrastructure problems. There are many features and extensions that I would like to have developed but did not have the time for. The most obvious problem with the suite right now is its lack of an iOS application. As of 2017, Apple's iOS holds just under 45% of total smartphone usage in the United States, meaning the suite as it stands is only open to roughly half the potential users [12, 13]. Time constraints ultimately forced the iOS application to be put off. As of May 2017, I am working with a former coworker, who is a senior iOS engineer, on plans to begin development of the iOS version of CommunityCommunicator.

There is also no central page for people to gain information about the application. Creating a web application to house information about the application could have possibly over extended me and made the quality of the product suffer. Once an iOS application is built, I plan on building a website that people can use to preview the application and talk about why CommunityCommunicator is necessary. The site will also contain statistics about the number of issues reported and resolved in the system as well as a map of all current open reports. Once the basics are covered, the ability for users of the suite to log in to the site to view their own reports and modify account details will be added.

Currently, there is no way for cities to communicate with people who make reports. The next feature that will be added to the system is the ability for different cities and towns to register accounts to be able to manage issues in their area. This would allow them to communicate with users, resolve issues that have already been taken care of, and improve

their own internal tracking. Until this is done, cities will not be able to use the CommunityCommunicator application as a full replacement for 311 reporting services. For this to be done, I will need to implement a separate authentication service for administration purposes. Setting up Keycloak by Red Hat should be sufficient, as it would allow the ability to create admins over a specific geographical area [14].

The last extension that is planned is one to prevent abuse of the system. Currently there is no filtering of images based on content type. If the description of the issue states that the issue is a clogged drain pipe, there is nothing stopping the uploaded picture from being a cute puppy. While most would likely use the application for what it is meant for, some may take the opportunity to upload images that are offensive in nature. Google has an API called Vision that processes information and tells you what the image most likely contains. Vision can detect what objects are in a specific image, what the focus of the image is, as well as whether it contains explicit content. It is, however, not free to use. The first 1,000 images can be processed for free. From there, Google charges $1.50 per thousand images, until five million, when the cost becomes $1 per thousand. Since open source projects often depend on donations and server hosting costs can get expensive, I may have to come up with a custom solution. If enough money can be raised however, Google's Vision API is an amazing solution.

# Conclusion, Acknowledgment, and Project Code

There are a few things that I wish I had done differently throughout this project. The first is spending more time on architecting the server API. While I left it open for easy extension, I found myself having to constantly go back and add REST calls or functions I did not consider that became necessary. In that same vein, my original set of data models only contained half of the fields they ended up with. Even though it is easy to add columns to a database, it forced me to wipe out and re-add test data every time a model changed. I also wish I had started with Test Driven Development, or TDD. TDD is a software engineering method where an engineer first writes tests for expected behavior and edge cases for a function, then writes the function to make those tests pass. This could have prevented many mistakes. The last mistake was not sticking with a more formal engineering methodology. Over the course of the project, I set vague goals that had no deadlines. This led to spurts of productivity followed by times of no work. Had a more rigid method with deadlines been followed, the quality of the work may have been better. I am still satisfied with the outcome however.

This application suite has come a long way since its inception over a year ago. Even now, both the server and Android applications are in early alpha stages of functionality and usability. The project is still far from complete. As is the case with a software project, more features will be requested. It would not be surprising that, as time goes on, this project morphs into something that barely resembles what it is today. Even if the project is not adopted, I still learned, and will continue to learn, a lot from its creation. Over the course of the last year, I have learned a lot about software engineering and design. I got hands on

experience with Amazon's AWS and other cloud providers, which is an essential skill for developers now. This project has given me experience that has already served useful in my career of software engineering while allowing me to do some good for the world. I knew I would not be making money while at the same time exposing my work to the world's scrutiny. Hopefully, the project being open sourced will attract other people and the CommunityCommunicator suite can grow enough that it helps fix infrastructure problems all over the world.

**Acknowledgements**

I would first like to thank my advisor, Mitch Fry, for being an incredible advisor and teaching me a lot throughout my computer science education. A special thanks to my friend Brady Sullivan who was the original inspiration for the project, introduced me to my now second favorite programming language, and helped me solved several software bugs throughout the duration of this project.

**Project code**

The code for the entire CommunityCommunicator suite is kept at the following URL: https://github.com/CommunityCommunicator. All future code changes, extensions, and bug fixes will be kept and tracked there.

# Bibliography

[1] Donovan, Alan A. A., and Brian W. Kernighan. 2015. The Go Programming Language. 1st ed. Addison-Wesley.

[2] “Improve Detroit | Mobile Apps | How Do I | City of Detroit MI.” Retrieved November 28, 2015 from http://www.detroitmi.gov/How-Do-I/Mobile-Apps/ImproveDetroit.

[3] Noyes, Katherine. Five Reasons Linux Beats Windows for Servers. Retrieved December 7, 2015 from http://www.pcworld.com/article/204423/why_linux_beats_windows_for_servers.html.

[4] “Gartner Says Tablet Sales Continue to Be Slow in 2015.” Gartner Says Tablet Sales Continue to Be Slow in 2015. Gartner Inc. Web. 8 Dec. 2015.

[5] Anon. What is Copyleft? Retrieved May 12, 2017 from https://www.gnu.org/licenses/copyleft.en.html

[6] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Retrieved May 21, 2017 from https://tools.ietf.org/html/rfc7231

[7] Niels Provos and David Mazières. 1999. A future-adaptive password scheme. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 32-32.

[8] Dustin Boswell. Storing User Passwords Securely: hashing, salting, and Bcrypt . Retrieved April 17, 2016 from http://dustwell.com/how-to-handle-passwords-bcrypt.html

[9] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in Haystack: facebook's photo storage. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, Berkeley, CA, USA, 47-60.

[10] Ado Kukic. 2016. Authentication in Golang with JWTs. (April 2016). Retrieved May 5, 2016 from https://auth0.com/blog/authentication-in-golang/

[11] auth0.com. JSON Web Tokens Introduction. Retrieved May 5, 2016 from https://jwt.io/introduction/

[12] John Koetsier. 2017. Surprise: Google Reveals Apple's iOS Market Share Is 65% to 230% Bigger Than We Thought. (May 2017). Retrieved May 20, 2017 from https://www.forbes.com/sites/johnkoetsier/2017/05/18/surprise-google-reveals-apples-ios-market-share-is-65-to-230-bigger-than-we-thought/#c3e54cf58903

[13] Anon. US mobile smartphone OS market share 2012-2017 | Statistic. Retrieved May 20, 2017 from https://www.statista.com/statistics/266572/market-share-held-by-smartphone-platforms-in-the-united-states/

[14] Keycloak Team. About. Retrieved May 21, 2017 from http://www.keycloak.org/about.html

[15] Matt Helms. 2015. New app lets Detroiters report potholes, other problems. (April 2015). Retrieved May 25, 2017 from http://www.freep.com/story/news/local/michigan/detroit/2015/04/08/detroit-launches-improve-detroit-app/25465923/

[16] Anon. 2017. City Reports 67,000 Neighborhood Issues Resolved Thanks To 'Improve Detroit' App. (April 2017). Retrieved May 25, 2017 from http://detroit.cbslocal.com/2017/04/03/city-reports-67000-neighborhood-issues-resolved-thanks-to-improve-detroit-app/

[17] Anon. 2016. SeeClickFix Helps Save Tax Dollars. (February 2016). Retrieved May 25, 2017 from https://gov.seeclickfix.com/2015/12/01/seeclickfix-helps-save-tax-dollars/

[18] Anon. May 2016 National Occupational Employment and Wage Estimates. Retrieved May 25, 2017 from https://www.bls.gov/oes/current/oes_nat.htm

[19] Rob Pike. Go at Google: Language Design in the Service of Software Engineering. Retrieved May 25, 2017 from https://talks.golang.org/2012/splash.article

[20] “What Is Open Source?” Opensource.com. Retrieved April 18th, 2017 from https://opensource.com/resources/what-open-source.