2016

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

AN EXPLORATION IN BUILDING OFFLINE WEB APPLICATIONS IN EMBER

Brook Bigford

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2015

Approved by

Advisory Committee

Douglas Kline	Eric Patterson

Thomas Janicki

Accepted By

Dean, Graduate School

Table of Contents

CHAPTER 1: INTRODUCTION

As technologies like web browsers, mobile phones, HTML, CSS and JavaScript mature, there are major shifts in the approach of engineering software to meet consumer's needs. Traditionally web applications have been client-server driven, meaning the client software relies heavily on a connection to a server to run the application whether it is serving HTML or querying data these applications do not function without a reliable connection. If a software company does not actively embrace the evolution of these technologies it is easy to fall behind in design and functionality, limiting the capability of existing and future web applications. Around 2005, Ajax (Asynchronous JavaScript) was gaining popularity in web applications and created an easier way for developers to update web pages without reloading the entire DOM (Document Object Model). Recently in 2014, the HTML5 standards were released and with them came many more API's that all major browsers supported such as local storage, application cache and web workers.

This study will investigate the advantages of several of these new features and discuss implementation strategies of these features in the popular open source JavaScript frameworks Angular, Backbone and Ember to determine which one is a suitable solution for developing single page offline web applications. At the end of the research, a real world application is proposed with a project timeline to flesh out Ember, the chosen framework, in a fully functional production ready application. After the proposed timeline is an updated review of how the application was implemented with a description of the completed functionality, an analysis of what was learned, and potential future work.

CHAPTER 2: BACKGROUND WORK & PROBLEM

Quintify, the company where I work was created in 2005 by Reid Wilson and develops ERP/CRM type web applications for businesses on a LAMP (Linux, Apache, MySQL, PHP) stack. Quintify uses a code generator designed and maintained internally that is named WM. It creates basic CRUD (Create, Read, Update, Delete) pages and the framework needed to run the pages. A developer creates a configuration file that defines some options and the data model and then it does the rest, it generates all the SQL, PHP and HTML needed for the page to function and interact with related data. WM was written in 2005 and only gets major upgrades when it directly relates to a client's request or problem. Due to the lack of new technologies being incorporated into the framework it has a lot of pitfalls in functionality. Clients substitute missing functionality with things such as Microsoft Excel spreadsheets, Access Databases and other products that are native to the operating system and can run offline. This causes pain for developer and client when there are problems between the communication of these technologies and the web applications actually responsible for aggregating all the data**.**

Another downside of the WM framework is it creates traditional HTML pages that require PHP from the server to do all the work in between each page load, so depending on how much information is sent or received and the underlying network connection, each page will have to send the full HTML and data each time it renders, as well as JavaScript code. As developers utilize JavaScript libraries each one will have to be individually loaded and initialized every page load so as Quintify incorporates new libraries into the WM framework, each one sacrifices crucial time as a crutch for missing client side functionality in the framework. Our firm is not the only firm to experience these problems with code generators and the leaps in technology.

CHAPTER 3: ANALYSIS

3.1 Models, Views, and Controllers

A popular approach to this problem is to offload all the functionality and template rendering on the client by loading all HTML and JavaScript for an entire application first and then only sending and receiving data from the server when needed. The open source community is exploding with frameworks in every language to accomplish these goals. All frameworks listed in Table 1 have complete source code with history available on GitHub.[1]

Framework	Language	Initial Release
CakePHP	PHP	2005
Laravel	PHP	2011
Django	Python	2005
Ruby on Rails	Ruby	2005
Angular	JavaScript	2009

Table 1: List of open source frameworks

One of the most popular approaches to web design is the MVC (Model, View, and Controller) architecture, in this approach all concerns are separated into 3 areas. The model contains the data and defines the structure of the data while providing some functionality, such as data formatting. A view is an HTML template that can render variables and provides control structures like for loops and if statements to render grids, tables, and input/output entities. A controller is the glue that puts the model and view together, a controller is responsible for fetching the model and placing it on the template. Events are also defined in the controller as well as specific page functionality. Another important piece that all frameworks are coming to realize is the router, it hasn't made it into the MVC acronym yet but it's just as important, a

router reads the browser's URL and loads the appropriate resources for the application which allows the browser to track the different states of the application in its history, so the forward and back buttons work.

These design approaches create a new type of web application called a Single Page Application or SPA. These SPA's consist of a JavaScript framework that loads on the initial page load and then manages the entire application without re-rendering the DOM, creating a much faster web page that almost feels native to the operating system. One of the pitfalls of the WM framework is that it was created in-house and rarely receives updates due to a lack of resources and the 'it works' mentality. In order to avoid that same problems as WM I will research frameworks maintained by the open source community.

3.2 Open Source Reasons

Open source projects are maintained by communities that provide full source code as well as historical data such as commit messages and example code. Another benefit of open source JavaScript frameworks is browser compatibility, the hassle of making JavaScript code behave the same way on each browser is not our concern.

This transparency creates a community where anyone can use the code, talk about the code, modify the code, report bugs in the code and fix bugs in the code all while being scrutinized by the public. As the communities grow, so does the amount of knowledge on the given framework, causing more access to guides, APIs, documentation, plug-ins, tutorials, examples, videos, chat rooms, forums, etc. So depending on which framework Quintify chooses not only will it matter that it has the required functionality but it also matters how popular it is and how well the community thrives. The popularity of the project is important in the open

source world because there is usually no monetary backing and as soon as the community loses interest, the project will become outdated causing problems similar to what we experience with WM.

In order to compare popularity of frameworks there will need to be some key metrics that can indicate how well each community is doing. By analyzing source code history from GitHub we can gather accurate metrics on different aspects of the code. GitHub is the largest web based source code repository with over twelve million registered users, so I will use it to judge which frameworks are popular and have a thriving community to maintain them.[1]

3.3 Open Source Data Explained

In the open source community all code changes are tracked through commits, each commit shows which lines were added or removed, a brief message on the work done, who made the change and when they did it. A developer usually makes a commit any time they complete a portion of functionality that is ready to be added to the source code history, this functionality could be new development or bug fixes but either way it indicates an important unit of work that was committed to the history of the project. Issues are used to request changes to the source code, they are a list of task that need to be completed and can be viewed by anyone, these task could include development request, bug fixes, or documentation but each member of the community has a chance to view the issue and weigh in on it. Branches are complete separate copies of source code that can have their own unique histories, branches are created to work on bug fixes or development and not break the master branch of the code. Pull request, are request from contributors to merge their new code back into the main copy of the code, pull requests are

usually created in response to an issue and merge the branch full of new commits and history back into the master branch.

Metric	**Angular**	**Backbone**	**Ember**
Initial Release	2009	2010	2011
Commits	7,164	3,167	11,194
Issues Open/Closed	955/5,986	21/2,133	269/3,861
Branches	16	3	24
Pull Request Open/Closed	349/5,841	19/1,643	57/4,734
Contributors	1,331	276	536

Table 2: Angular, Backbone, and Ember GitHub Statistics, gathered October 21, 2015[1]

3.4 Chosen Frameworks

After examining the statistics on GitHub, I have chosen Angular, Backbone and Ember to research in the recommendation for a JavaScript framework. Each one has been around for a few years, has plenty of contributors and is still actively maintained through issues and pull request. Table 2 details the three popular frameworks and selected metrics.

A quick glance at the numbers makes it obvious Angular is the most popular and has been around the longest which isn't that surprising since Angular started as a Google side project.[4] It by far has the most contributors but it also has the largest amount of open issues which seems odd, all those contributors should be fixing the open issues. Angular was created by a Google employee, Misko Hevery, when he made a bet that he could re-write a 17,000 line program in two weeks, unfortunately he lost the bet but by the third week he had re-written the application in just 1,500 lines of code and had created Angular.[4]

Backbone is the least popular of the listed frameworks, and actually it isn't even a framework, it is a library that can be used to solve the same problems as other MVC frameworks

but with a little more flexibility.[6] Even though it is not a framework, it has potential as a recommendation for Quintify, the file size is the smallest of the three and it works well with existing web pages. Frameworks impose their architecture on developers to do things their way whether it is naming conventions or code architecture, where libraries offer a set of functions to be utilized when needed.

Ember is the second most popular of the three listed, it is the youngest, has the most branches, and is the only one with more pull request than issues.[8] This is interesting because typically a pull request will be a response to one or more issues, so this could indicate the community is creating their own pull request without issues requesting them.

Each of these frameworks tries to accomplish the same objectives: an MVC framework, data binding, template rendering, and routing to create a SPA. In order to make a recommendation for which is more suitable I will build a simple application in each framework so I can compare how the code implements each objective.

3.5 Storage Techniques

The final areas of my research are the new HTML5 API's supporting local web storage and application cache. In HTML5 developers realized they needed a better way to persist data on the client side so they created a way to store data and cache pages. Web storage currently has three different ways to store data client-side: local storage, web sql, and index DB but of the three only local storage seems mature enough to be production ready.[15]

- Web sql, which is now deprecated, was an attempt at implementing a SQLite database on the client side using SQL statements in JavaScript to store and retrieve data.

- Index DB is a flat-file database that supports key-value pairs and indexing.
- Local storage uses a JavaScript object to store key-value pairs on the DOM.

STORAGE	CHROME	FIREFOX	SAFARI
LOCAL STORAGE	4+	3.5+	4+
INDEXED DB	23+	10+	-
WEB SQL	4+	-	3.1+
LATEST RELEASE	46+	42+	9

Table 3: Browser implementations of local storage [12,13]

3.5.1 Local Storage

Local storage is the most stable and has been implemented in the most browser releases, so it is safe to say local storage is compatible with the most browsers. Table 3 list some major browsers and which version implemented each storage API. If a web page doesn't utilize client-side storage, data has to be loaded into JavaScript memory each time the DOM reloads. Local storage gives access to a JavaScript object accessible through the DOM object that can store data that persist on the client side.[15] It stores data in a key, value pair so many developers encode their data to store it as a single value and then decode it when needed. This provides a way to store large amounts of data locally that wasn't possible before.

3.5.2 Application Cache

Application cache is used to cache the files needed to run a web page. Application cache requires two things, an application cache file and a manifest attribute in the root HTML element of the application.[15] The application cache file lists which files to cache and what to do in case

there is no internet connection, besides editing the HTML element all other application cache events take place at the browser level, so the only difference in the implementations will be the files listed to cache, the frameworks themselves will not interact with the application cache in any way. Once a page is cached in the browser it can be used anytime the URL is visited, meaning the browser won't need to re-load any HTML or JavaScript from the server. It will just read the cached files list and load them from memory.

As long as the storage was initialized and the files are cached the new HTML5 APIs make it possible to cache entire web sites and store data and if they are implemented correctly it can eliminate some server calls and reduce the size of each call to just the necessary data. The API's themselves are very simple to use but the challenge will be how well they integrate with the chosen framework. I want to leverage each API with the JavaScript frameworks to create a SPA that runs seamlessly offline.

CHAPTER 4: RESEARCH GOALS

The purpose of this research is to compare JavaScript frameworks based on functionality and popularity to recommend which framework is suitable to be used by Quintify to build offline web applications. Based on statistics from GitHub the frameworks Angular, Backbone, and Ember each have a large community that should provide a promising future. Along with the popularity each one has clear documentation and local storage add-ons that should make the development process easier.

4.1 Application Description

In order to compare the three frameworks I will need to design the same application utilizing the latest versions of each framework. It will also leverage HTML5 technologies to run seamlessly offline when there is no internet connection. The application that will be used to compare the frameworks will be a customer storage program, the application will allow the user to interact with customer information while offline.

Figure 1 Application Main Page

The application will initially load over a normal web browser, cache itself and then load data from a server as soon as it is initialized. Figure 1 demonstrates the initial load screen. It will then store the data internally in JavaScript cache and in local storage; at this point it is offline ready. It will store simple customer information such as first name, last name, address city, phone number, email, and zip code as shown in Figure 2.

Figure 2 Customer list page with filter

The application should be able to navigate to a list page that displays a list of the stored customers. Each frameworks list page will contain a search filter that can search the list and dynamically filter out those that don't match the filter. Each individual customer will have the capability to be displayed, edited and deleted. The application will allow new customers to be added.

4.2 Functional Limitations of study

For the purposes of the comparison I am not concerned with an authentication or authorization, the study will compare the following functionality: modeling data, rendering templates, filtering data, and local storage implementation. The application will only talk to the server once; it is not responsible for sending updates or new data to the server. I will not implement any custom CSS since each application will look exactly the same. Each application

will communicate with the same server and data but data formatting might be different to accommodate the different frameworks.

4.3 Approach to comparing frameworks

By teaching myself each framework and building the exact same application, I can judge how difficult each language is to learn. Items to compare in the study will be:

- The file size of each application and load speed can be measured and indicate which is faster and lighter.
- Large data sets can help determine which framework handles data better. I can look at how fast the data loads, the template renders or a filter takes to load all the data.
- The size of the data in local storage is another concern, if a framework stores data more efficiently than it should be able to store more customers.
- Each framework uses similar terminology and programming constructs that can be compared as pros and cons to determine which is more effective for my recommendation.
- Each frameworks controller implements functions slightly differently or the templates use a slightly different syntax, I will evaluate naming conventions and architecture to determine which is more readable and explicit in what it is doing

I will need to find plugins that handle local storage implementations rather than build it myself, the ease of finding and using these plugins can also indicate how easy the framework is to use. A big problem with frameworks is they seem magical, when really there's just a function hiding somewhere in scope. When looking at the code it should be readable and clear with what

it is trying to do. Once I evaluate and compare these aspects of each application I should have an idea on which is suitable for Quintify.

CHAPTER **5**: APPLICATION DEVELOPMENT OVERVIEW

Each of these frameworks accomplish the same thing, so I started by looking for examples that were similar between them. I found a GitHub repository to do MVC, which is a project dedicated to making the same MVC to do list in popular JavaScript libraries and frameworks.[14] These examples provided a simple, easy to follow architecture that allowed me to see how each one utilizes the models, views and controllers to display and interact with a list of things to do. The to do examples demonstrated a lot but they did not explain some functionality for example how to store and pass data between controllers, use routers, filter data, load data from a server or run offline. By extending the application to meet this functionality and changing the data model from a to do list to a customer list, I was able plan out how the application should look and act consistently across the frameworks. Table 4 details the size of each finished application with 5,000 customers loaded in to local storage before the DOM renders.

Framework	*Application Size*	*Local Storage Size*	*DOM Load Speed*
Angular	104 KB	1.83 MB	894 ms
Backbone	59.7 KB	1.85 MB	645 ms
Ember	1.4 MB	-	4.26 s

Table 4: Comparison of Finished Applications

Backbone was by far the smallest framework which led to it being the smallest application with the quickest DOM load speed. DOM load speed, application size, and local storage size were measured using Chrome's developer tools. Special care was taken to make sure no files were cached while gathering statistics but DOM load speed comes from a single sample

of a page load time, so further results might vary based on the network and other factors outside of the browser.

Local storage was populated using sample customer data contained in a data base on the server running the application. Each framework was designed by loading 500 customers into local storage, which worked fine but when I changed it so that it loaded 5,000 customers, to compare how each framework does under stress and how much space is consumed, I noticed Ember could not handle it. Angular and Backbone had slightly noticeable differences in load time when loading 5,000 customers, Ember on the other hand would stall and then crash the browser while trying to load the data. In order to get Ember to work I had to limit the size of the data to 2,000 customers, which is why the local storage size is omitted from Table 4.

5.1 Framework Development Concerns

As the applications were developed it was difficult and confusing to determine which objects should be used and how they should interact with each other. One of the most difficult parts was determining how to put data in scope, if a controller has its own model, how does it access data from other models from within the controller. In Angular, a factory object has to be created and is responsible for holding all the data that could then be injected into every controller. In Backbone an object on the global application variable is used to store the data. Ember was the only framework of the three that had a built in data manager known as the data store. The store was accessible throughout the whole application and contained many utility functions that had to be manually written for Angular and Backbone such as querying for a record by id or filtering records.

When initially choosing these frameworks to research it wasn't obvious how much

additional code would need to be added to the core framework, some of this code was written for just the framework like the Ember data store or the Angular router while other code complemented and enhanced the framework from an outside community like jQuery, Handlebars or Mustache. Handlebars is a templating engine recommended by Ember and Mustache is one recommended by Backbone because they don't have their own templating abilities. Angular handles its template rendering internally so it does not need an extra file.

CHAPTER 6: RESULTS

There were noticeable differences in each framework as well as similarities. First I would like to discuss the overall implementation of each and some features that stood out, and then after that compare them to each other as a whole to determine which is superior.

6.1 Angular

Angular is the oldest of the three frameworks which means it's had longer to mature and amass knowledge on the subject. The Angular application required three JavaScript files (shown in Table 5) plus the application code to run. The Angular JavaScript I wrote for the application was 143 lines of code and the HTML template was 145 lines of code. It is important to note that the lines of code counted include very descriptive comments that boost the number of lines significantly.

PURPOSE	NAME	SIZE	VERSION
CORE FRAMEWORK	Angular.min.js	45.4 KB	v1.3.14
ROUTER	Angular-route.js	151 B	v1.4.3
LOCAL STORAGE	Angular-local-storage.js	151 B	v0.2.1

Table 5: Required Angular Files[2]

My Angular application starts by defining the router, inside the router it has a switch statement for matching the URL in the browser; inside each case of the switch statement it defines what controller and template to use. The controllers use a $scope variable, which is passed as an argument, to interact with the template. My first thought was each controller could just use the $scope variable to pass data but it's actually unique between each controller. It refers to the $scope of just the controller not the entire application.

In order to pass data between controllers I had to define a custom factory object to store the data during run time that could be injected into each controller. It was inside this factory that I loaded the customer information and interfaced with the local storage add-on. I wouldn't call any of the code written to do this as clever; the customer loading was more of a configuration parameter to point at the correct server and parse the data correctly. The local storage code just used an Angular service as a wrapper to interface with local storage so instead of the normal JavaScript function of localStorage.setItem("key", "value"); you would use localStorageService.set("key", "value"); which is essentially the same thing except it returns everything in the correct format Angular expects. Another interesting feature is it did not require me to define any of the fields that would be inside the model anywhere, so Angular doesn't actually know anything about the data in the model, just that it's a JavaScript object with attributes, meaning any attribute can be added to the model.

6.2 Backbone

I think Backbone was the most interesting to work with, as discussed earlier Backbone isn't considered a framework, so the application code will be responsible for more of the architecture of the application.

PURPOSE	NAME	SIZE	VERSION
CORE FRAMEWORK	Backbone-min.js	7.7 KB	v1.2.3
LOCAL STORAGE	Backbone.localStorage-min.js	1.6 KB	v1.1.16
TEMPLATE ENGINE	Underscore-min.js	6.1 KB	v1.8.3
JQUERY	Jquery-git2.min.js	33.9 KB	v3.0.0

Table 6: Required Backbone Files[5]

In Backbone templates are defined as jQuery objects and their controllers are called views, so a Backbone view is responsible for fetching the model and placing it on the template similar to how a controller in Angular or Ember works. My Backbone application required four JavaScript files plus the application code as shown in Table 6. The Backbone JavaScript code was 186 lines of code and the HTML was 135 lines of code.

The Backbone application starts by defining some global variables and then initializing the router. The router analyzes the browser's URL and calls a corresponding function; this function is responsible for rendering the view with the model. The global variables provided a way to share data between views, I just had to make sure when the application loaded it initialized the variables correctly. When loading the variables I was also able to check local storage, if the data wasn't there I could make the request from the server.

The template rendering was very confusing at first. In Backbone you are required to manually initialize the template and render it, which puts a lot of templating code inside the view, increasing the length and making it more difficult to comprehend. Mustache, the templating engine for Backbone, doesn't have control structures like loops, thus on the customer list page each individual table row was loaded from a separate model and view and then used in the list model and view. I don't like creating two views for a single page, especially if one view

is for creating a table and the other view creates the table rows, it seems like that should be handled in the same place.

In Backbone when you want to request data you just need to use the fetch function to talk to the server. The Backbone local storage module was very easy to use, it overrode the fetch function so that the default use was to fetch from local storage and by passing in some options it could still fetch from the server. In Backbone I had to define the model as a Backbone object but I didn't have to define the fields on the model. Backbone was the most difficult to learn but I don't think it's because it was overly complicated, it just doesn't do as much out of the box as Angular or Ember, once I was in the Backbone mindset everything made sense but the transition was painful.

6.3 Ember

Ember is the youngest of the three frameworks and in my opinion has the most functionality. My Ember application required six JavaScript files plus the application code as shown in Table 7. The Ember templating engine is currently in a transitional phase where it needs an additional file to compile the templates in a format Ember expects. I could not find a minified version of the templating engine or the compiler which might explain why Ember is so large compared to the other frameworks. The data store is another additional file used in Ember, it is kind of like a data layer that sits between the application and the data. The Ember application code was 161 lines of code and the HTML was 145 lines of code.

PURPOSE	NAME	SIZE	VERSION
CORE FRAMEWORK	Ember.min.js	363 KB	v2.1.0
DATA	Ember-data.min.js	104 KB	v2.2.0
LOCAL STORAGE	Localstorage_adapter.js	17.3 KB	v0.1.2
TEMPLATE ENGINE	Handlebars-latest.js	156 KB	v4.0.4
TEMPLATE COMPILER	Ember-template-compiler.js	668 KB	v2.1.0
JQUERY	Jquery-1.11.3.min.js	94 KB	v1.11.3

Table 7: Required Ember Files[8]

The Ember application begins just like Angular and Backbone, it defines a global variable and then defines the router. The Ember router is different from the other routers, it doesn't define the controller or model explicitly like Angular or Backbone and instead it relies on a strict naming convention. So if the route from the URL is "customers" then Ember would first check if the route is defined in the router and then automatically look for a controller name "CustomersController", a template named "customer" and a model name "customer". If the router can't find the controller or model it will actually generate a default one during runtime. It isn't necessary to define a controller for displaying a list of customers or a single customer, it just has to be defined in the router and have the appropriate templates.

The data store takes care of loading, saving, querying and sharing data between the controllers. The local storage adapter overrides part of the data store so that it uses local storage instead of a server. This solved many of the problems faced in Angular and Backbone, in Angular I had to define a custom object to manage the data and interface with the application, in Ember all this functionality comes built into the data store. Ember requires that all fields on the model are defined ahead of time, this allows for computed properties and observers. A computed property could be full name, which is a combination of first name and last name, so if a last

name changes it will re-compute the full name property again, updating that property on all templates. An observer is a similar property on a model except it is a function that observers another property and fires if the observed property changes.

The Ember framework has a thriving community and in my opinion has very good documentation. The community is also starting to push a CLI (command line interface) tool to enhance development.[9] I did not use the CLI for my research but it looks like all future documentation will be written with its use in mind. The CLI will run on a Node server and will provide tools such as node packet manager, unit testing, and file generation. In my performance comparison Ember could not load 5,000 customers from the server, initially this looked very bad, but after researching Ember more and reading the forums, it seems the data store is responsible for this slowness.[10] The data store is very ambitious in what it is trying to do and the performance suffers because of this. In an Ember application that needs to be connected to the internet pagination is the recommended way to solve the data store problem, only load the data when the user needs it, which obviously won't work for an offline web application.

6.4 Framework Comparison

After researching, designing, building prototypes and analyzing each framework, I've come to the conclusion that an Ember application would be the most suitable for Quintify. All three frameworks were able to accomplish the same goals in under 400 lines of JavaScript, HTML and comments. Each framework has a large community and is used in real world applications by large companies which indicates each framework is production ready. The MVC framework is an architecture that defines how a user interface should be broken up into functional pieces, because of this each sample application had the exact same logic flow the

application starts with the router, which analyzes the URL and then loads the appropriate model, view and controller.

In Angular and Backbone there is no strict naming convention so depending on the developer, the variable names might not be as intuitive as an Ember application. For example if the route in the URL is “customer”, in Ember I know that each variable holding the model, view and controller will begin with the singular form of customer, in Angular or Backbone I would need to look at the routing code to see which controller and template are being loaded. This naming convention simplifies following or reading the code because a developer will immediately know how objects are named and how they work together creating code that can be understood by anyone in the community rather than just the company or developer developing the code. Another benefit of the naming convention is Ember can actually generate missing objects at run time, for example if I didn’t create a controller for displaying a single customer, Ember would actually generate it at run time, and as long as the route is set up correctly it will even know to load the customer with the correct ID. This means you can skip declaring some objects all together and it will still work.

The Ember data store is another reason I chose Ember, in Angular and Backbone I felt like it was very routine writing the code to parse the list of models and load a single record based on ID, I declared a for loop and iterated through the list until I found the correct ID. This seems like something the framework should take care of, finding a single ID is simple enough but as the application expands writing utility code to do simple things will become very complicated. It is very disappointing that the data store can’t load 5,000 customer records up front but I think it is even more disappointing that Angular and Backbone don’t have a store type object that handles these functions.

The handlebars templating engine was another positive feature of Ember, because of the naming convention it was very easy to link the template to the route and know which controller and model were being loaded. Handlebars also included control structures like for loops and if statements inside the HTML making it very easy to understand how a template will render by looking at it. In Mustache, the Backbone templating engine, there are no control structures so more templating code is moved into JavaScript making the templates harder to follow. Handlebars and Angular had similar templating techniques except Angular defines everything as custom attributes on HTML elements and then uses the variable inside these elements. Handlebars uses custom symbols to declare control structures inside the HTML itself making it easier to see and understand. So if an angular template wants to make a list of models, it has to declare the loop inside the tags of the element containing it, in Handlebars you would see a traditional for each loop written inside the template.

The size and amount of files needed to build my Ember application is concerning, it required 1.4 MB of files which is more than ten times the amount of Angular and twenty times the amount of Backbone, I think the current development on the templating system is causing these numbers to be so high but even when I look at core framework size Ember is 363 KB which is still twice the size of the whole Angular application. Ember is definitely larger and has worse performance than Angular and Backbone but the overall architecture and naming conventions make it very easy to pick up and start building applications in.

6.5 Framework Results

Based on my research Ember is the framework most suitable for Quintify. It provides a framework for building single page applications that are easy to understand by being very explicit in how the code is written. It has a large community which based on the development of

the templating system is very active and actively trying to make the framework better. The data store is either the best or worst quality of Ember, the slow loading almost disqualifies Ember as an acceptable framework but the utility functions provided by the store simplify the code and provide much needed functionality missing in other frameworks. As mentioned earlier the slow loading problem is known by the community so hopefully it will be optimized in the near future.

CHAPTER 7: CAPSTONE DELIVERABLES

The main goal of this research was to compare and choose a framework based on a very simple customer storage application, each application was built with the exact same user interface and data model to make them comparable, now that a framework has been chosen, I will develop a full scale customer storage application that could be used by businesses to store customer information offline and schedule appointments with the customer.

This application will be built using Ember and it will include all functionality from the sample application plus additional functionality for scheduling appointments for the new customers. In order to schedule appointments the application will need to connect to an API to download sample employees and appointments, to download the correct information the application will need to authenticate and authorize the user to connect to the API. Once the information is ready the application will enter an offline mode, in offline mode the user can add customers and schedule appointments using a grid like calendar that displays time slots and employees. If the customer does not want an appointment they can opt out for a phone call instead. Once the user is connected to the internet again, they will need to re authenticate and authorize in order to save the data back on the server.

This application will only be used in offline scenarios. To see the data after it is loaded to the server or to download existing customers is not in the scope of the application. This will build on the existing work by adding in relational data, an event scheduling system, user authentication and authorization, an API connection, responsive web design for any screen size, and a redesigned user interface with aesthetics for a nice look and feel. The data store performance will be important to monitor along the way but it shouldn't ruin the entire application.

CHAPTER 8: PROJECT TIMELINE

February 9, Tuesday - Establish Capstone Committee/Begin Proposal Work

February 16, Tuesday – Submit Proposal to Committee

March 4, Friday – Propose before this date, beginning of Spring Break

March 15, Tuesday – Application Diagrams

- Set up environment
- Set up source control
- Finalize data tables and data models
- Finalize HTML Templates

March 22, Tuesday – Application Progress Meeting

- Create routers and controllers
- Set up API calls to server
- Set up initial CRUD test cases

March 29, Tuesday – Meeting on Application Progress/TBD

- Create Calendar for scheduling
- Investigate scheduling conflicts
- Set up Calendar test cases

April 5, Tuesday – Meeting on Application Progress/Final Defense Preparation

- Finalize CSS
- Investigate mobile support and responsiveness
- Finalize Capstone research paper
- Schedule Capstone defense

April 19, Tuesday – Defend Capstone

- Final Capstone Oral Defense

CHAPTER 9: PROJECT REVIEW

When building the web application, it started as an empty project and didn't use the prototype built for research purposes. There were a few reasons for this but mainly because the project switched to the Ember-CLI for development which required all code to go in a project structure rather than a single HTML and application file. Due to the complexity of some items and length of time to complete, the project followed more of a bottom-up approach to building it. The focus was on completing each test case as it was developed rather than completing each module and then assembling at the end as described in the original project timeline. Below is a more appropriate timeline with each objective listed in order:

- Setup Environment
    - Build Database
    - Add Data
    - Download Node.js, NPM, Bower, Ember-CLI
- Setup Project
    - Create Ember-CLI Project
    - Open Ports for Development
    - GIT Repositories initialized
- Create Main Page
- Download Bootstrap, Ember Local Storage Adapter
- Create Customer Functionality
- Create Navigation Bar

- Create Login Form
- Create employee API Call in application
- Create employee API on server
- Create Login API Call in application
- Create Login API on server
- Create Employee and Appointment Functionality
- Create Scheduler
- Create Syncing Form
- Create Sync API Call in application
- Create Sync API on server
- Update Navigation bar
- Add Mobile Responsiveness
- Build Project with Application Cache

When building the application I took advantage of GIT and had many branches created for each large portion of functionality. Due to the bottom-up approach of building the project each time functionality was added in it caused tweaks to existing systems which could break it or introduce bugs into it and requires retesting of previous test cases. When rewriting code it is often better to restart rather than rewrite the first draft, having clean branches to always restart from ensured cleaner code and history yet it still maintained the original version.

Another issue during the project was the amount of spaghetti code and duplicate code that piled up on some of the complicated templates. One example of this is the scheduler, as different features were added in each one would reset and load the calendar template in a similar way

causing code to be duplicated in different parts of the program. After the scheduler was completed it needed to be re-written with all functionality in mind to reduce duplicate code.

CHAPTER 10: IMPLEMENTATION

The implementation of the final project is much different than the initial prototype. The final project utilizes the Ember-CLI (Command Line Interface) for development which runs on a node server. The Ember community is transitioning toward Ember-CLI and all documentation and plug-ins are written with it in mind. Ember-CLI is installed using Node.js, a cross platform runtime environment. The cross platform runtime environment means Ember-CLI runs consistently on all platforms. Ember-CLI and Node are open source projects. Node is responsible for managing the differences between platforms so in theory developers on Windows, MAC, and LINUX shouldn't need to worry about differences in the Ember-CLI behavior. Node Package Manager or NPM is a package manager used to manage the dependencies in an application, all the dependencies it manages are node modules. NPM is used to install Bower and Ember-CLI. Bower is a useful tool for managing front-end project dependencies. Bower manages client side libraries like jQuery or Bootstrap that aren't written specifically for node.

10.1 Ember-CLI

The Ember-CLI is completely different from the traditional Ember approach, it is easier but it requires a little more understanding to get started. When creating an Ember project it has its own set of commands that can be used to create the project and set up the initial directory structure. This project automatically comes with the latest libraries and frameworks like Ember, Ember-Data, jQuery, HTML Bars and the ability to download more with NPM or Bower

commands. Ember-CLI sets up the GIT repository and ignores the appropriate directories so that only essential project files are committed to the repository. The directory structure provided by the Ember-CLI helps keep code neat and organized as the project grows and more developers join the team.

When creating new objects in the application the CLI can generate the necessary files with skeleton code to begin coding faster. Using the Ember-CLI also makes it easier to get support for different CSS preprocessors like LESS, SASS or Compass or JavaScript compilers like CoffeeScript or EmberScript. Another feature of the Ember-CLI is built in jUnit testing, which means it will run each test before building the application letting you know if something is broken.

10.2 Development Features

The CLI also came with a JavaScript library called Watchman.js, Watchman can actually watch the directory structure for changes and then update the project when necessary. When developing web applications it often becomes strenuous refreshing the page every time the server code is updated, Watchman opens a web socket with the browser and when it senses the changes it can actually update the browser on the client side. It can also parse the code as it is updating the project, if the code fails the build on the server it will indicate where the error is with a very helpful debug message. If the build gets sent to the client browser and still has errors, the browser itself will display an error message letting the developer know which line of code is bad. This set-up allows the developer to easily pin point warnings, run-time errors, and syntax errors in JavaScript or HTML templates, which is very helpful typically if there is a JavaScript error the console is the only indication to the error.

The Ember inspector is another amazing feature for ember development, it's a chrome add-on for inspecting ember projects. The Ember inspector allows the user to inspect all aspects of the project, which lets the developer see how routes and controllers interact during runtime. The inspector can also echo any object to the console which allows access to functions and properties not easily seen without it. These new features make the switch from traditional Ember to Ember-CLI worth all the time, now when developing instead of refreshing and just seeing a blank screen, Ember will assist in every way possible whether it's generating directory structure, inspecting data during runtime or displaying syntax errors.

10.3 Building an API

The customer manager prototype built for research made building customer, appointment and employee forms trivial. The first major step in the development was building an API (Application Programming Interface). Ember-data is built to function with a RESTful JSON API so the API was modeled after one. The API works by using a randomly generated token to identify the application and which API actions they have access too. The token is embedded in the application itself, a different token means a completely different type of application. If the token isn't found on the server the API will return an error message, if the token is found then it will allow the user to request an action.

The CRM only has three API actions: authenticate, get employee information, and save customer information. Anytime the user wishes to connect with the API they must authenticate with their login information, once authenticated they can request to get information or save information. The server checks the actions, if the action is to get employee information, it will send back employee information for only employees related to the user who is currently logged

in. The save customer information action works the same way except it parses incoming information and saves it to the database. RESTful API's do not save state, so the API will not be responsible for knowing what the application is doing, the application is responsible for calling the API at the correct time and knowing what to do with errors.

10.4 New Database Features

In order to expand on the project new database tables had to be created for appointments and employees to store information on the server. The relationships were kept simple with customers being joined to employees only through appointments. This setup ensured a customer can only have an employee if they have an appointment, even though on the traditional web site a customer might be assigned an employee and have appointments with multiple other employees, this functionality was outside the scope of the application.

10.5 New Ember Features

Earlier the changes for transitioning from traditional Ember to the Ember-CLI were discussed, this section will talk about the new ember features used or discovered when building the ember application. Components and services really helped simplify certain parts of the project, a component is a custom HTML tag that has some JavaScript behind it, for example when selecting an employee from an ember select list it is made out of a component. Components can be passed data for internal use just like functions, the navigation bar is another component that uses certain parts of application state to know which navigation buttons to show. Components make it easy to reuse HTML templates that don't belong to a route but need to be used in many places.

Services are singleton objects that can be injected into all parts of the application to be used. The ember store is an example of a service used throughout the entire application. In order to make the login functionality and API calls, it was all put in a login service that handled all communication with the server. When the application initially loads, it all loads from a main application route that all other routes are nested inside, by injecting the login service into this route the login service became global across the entire application. Once the login variables were global the navigation bar could display appropriate links on every page. Another example of a service is the calendar service, building the HTML for a calendar and adding timeslots didn't really need to be in the scheduler's application code so a custom service was created that sets up the calendar and then just has the scheduler page interact with it. Using services and components really helped to clean up code and make it reusable.

10.6 Appointment Scheduler

The scheduling tool is the most complex portion of the application, it is used when scheduling an appointment for the customer. There are many different ways to make a scheduling tool and there are many open source options available to use. For this project a custom Ember scheduler was written, I have used schedulers before but they would be overkill for the type of application described in this paper. Figure 3 shows the scheduler in desktop view when adding an appointment for a new customer.

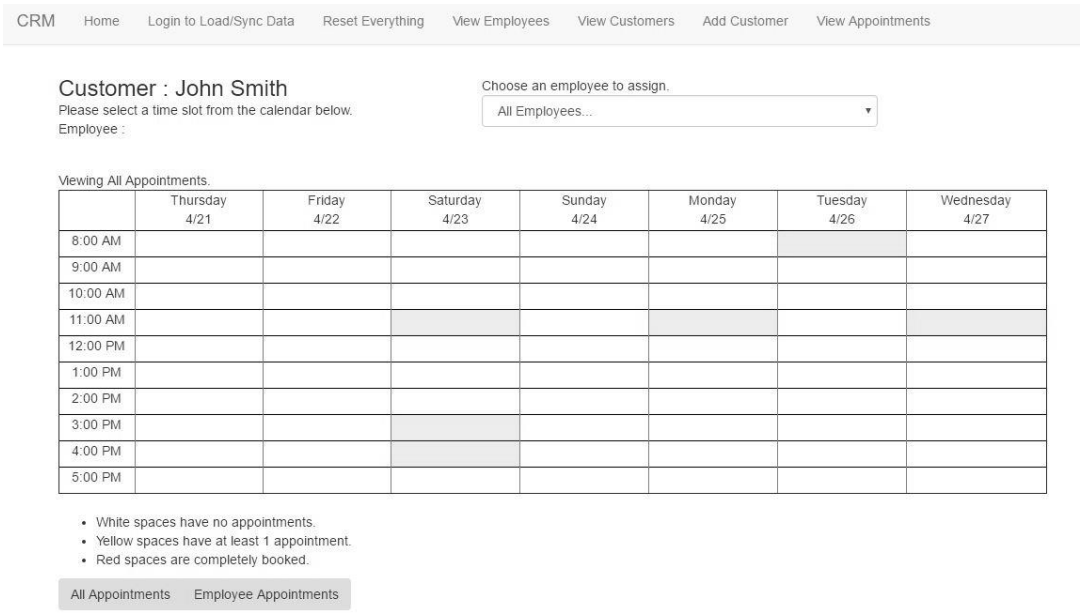

Figure 3 Scheduler adding an appointment

The scope of the application when it comes to scheduling is very basic and the only real concern is the size of the screen viewing it, most calendars aren't developed with all devices in mind. A tablet is very difficult to enter data on and uses half the screen for an on-screen keyboard which would cover half the scheduler, this alone rules out many tools that would use a date slider or some other type of plugin that isn't touch optimized. Due to the screen sizes and devices that might be used, the user should not have to enter any data on the page, just select between available options.

The user can only visit the scheduler if employee and customer information is loaded this ensures they only have to select an employee and a timeslot to create the appointment. In a more robust scheduling tool appointments can start and end at any time, this adds a lot more complexity to application code and makes it easier to make mistakes when scheduling appointments. By using 1 hour time slots that begin at the top of the hour it is easy for users to select a single time slot and it also made building the scheduler much simpler. When scheduling an appointment the scheduler only has to check for appointments that start at the same time as it

and because appointments can't be larger than one time slot it's very easy to find appointments that start at the same time.

Currently the only way to view the scheduler is from the customer page, once a customer's information is saved the user has the option to schedule a phone call or schedule an appointment. If the user chooses to schedule the appointment they are brought to the scheduler which initially shows all appointments in the system, from there the user can choose an available time slot and assign an employee, if some employees already have appointments in that time slot the slot will be yellow. Customers can only have one appointment so there is no risk of appointment overlap for them but employees can have many appointments so if the user selects a yellow time slot then only employees who don't have an appointment at that time will be available. The scheduler will also show red time slots which mean all employees are currently booked at that time and will not allow the user to schedule during it. A customer is assigned an employee only through the appointment, and if a customer needs a new employee they cannot edit the appointment directly and select a new one, they have to delete it completely and then return to the scheduler to schedule it.

The scheduler has two modes, an all employee mode and a single employee mode, when viewing the scheduler the user can choose to see all appointments or just the appointments for a single employee, this is for when the user already knows which employee they are going to assign, so they only have to load that employee's calendar. If the user makes a mistake when scheduling the appointment they do have the option to edit it, if the user views the customer display page, it will give them a button to edit the appointment, this button brings them to the scheduler in edit mode. In edit mode the scheduler looks at the appointment and loads all appointments for the employee listed on the appointment being edited, the time slot to be edited

will be green and all others red, the scheduler works the same way but it updates the appointment rather than creating a new one.

There were many challenges to implementing the scheduler from building the HTML itself to loading appointments into time slots. When working with time it is often difficult to display it in a way that makes sense or make decisions based off of what times are selected. A new service had to be created to be used by the scheduler service, this was a date service, and it was responsible for formatting all dates in the application to keep them consistent. The scheduler was built using JavaScript objects that contain information in them, like what time slot it is, and what employees currently have appointments inside of it. At first standard JavaScript objects were used but later on it was realized that they were missing the Ember extensions that make life easier like two-way data binding. The problem was when a user selected a time slot, it was able to access the information inside of it but it was not bound to the calendar object or template, so if the scheduler were to load the calendar and then the user deleted an appointment the scheduler wouldn't update because that time slot wasn't bound to the calendar. Once all object were changed to some extension of an Ember object it immediately got that functionality, Ember would monitor a time slot and if employee information changed in it, it would update it simultaneously on the template. It was interesting figuring out when Ember objects had to be used and when they didn't there were a lot of issues like this, simple JavaScript would be written and then cause weird bugs later on.

Another issue was syncing customers and appointments back to the database, the API itself was routine in designing but there were a few difficult questions to answer like when saving appointments back to the server what happens when appointments are overlapped? When the user syncs an appointment in theory it's done later after the customer is gone and there is an

internet connection. If there is an overlapped appointment it would not matter where the user fixes it because they can't communicate with the customer in person anymore, they just need to be aware of it. Due to this and the scope of the application, the ability to resolve conflicts was not built into the application, the application will alert the user to the overlap and direct them to the traditional web site to fix it. The last concern with syncing the information is what to do after it is synced, in theory if the user is done and synced the customers to the database they can just login to view the information on the traditional web site but what should be done about the customer information in local storage? What happens when they load the web application again and all the customers information is already saved to the server?

In order to limit the complexity of managing multiple uses and editing customer information that is already on the server, the application will not let the user edit any information that has already been synced. If the user loads the application and there is existing employee information, they can't download it again, the application will not let them make that API call to get employee information. Once the employee information is set, the application just looks for customers and appointments that aren't synced, if there are any that aren't synced the user will have the ability to save those customers and appointments back to the server. If there are any customers or appointments in the system that are synced the user will be alerted and not be allowed to edit them just view them.

10.7 Responsive Design

The last piece to the application was styling it in a responsive way, in order to make the application responsive Bootstrap was used. Bootstrap uses a row and column grid system that can easily expand with screen size and compensate for smaller screen size. Pages using only one

table for display information like list pages were easy to develop, Bootstrap naturally knows how to expand or contract a single table to make it fit a screen. The display pages were slightly more difficult to manage, the customer and employee display pages have two tables in them, one for displaying their information and another displaying appointment information. A customer can only have one appointment but an employee can have as many as possible, the display pages needed to take advantage of wide screens by displaying the tables next to each other, and take advantage of narrow screens by stacking them on top of each other. Figure 4 shows the differences in the mobile and desktop views. In order to accomplish this in a straightforward way, the tables were styled in columns next to each other that way when the screen becomes to small it automatically stacks the columns, meaning the user just needs to scroll down slightly to view appointment information.

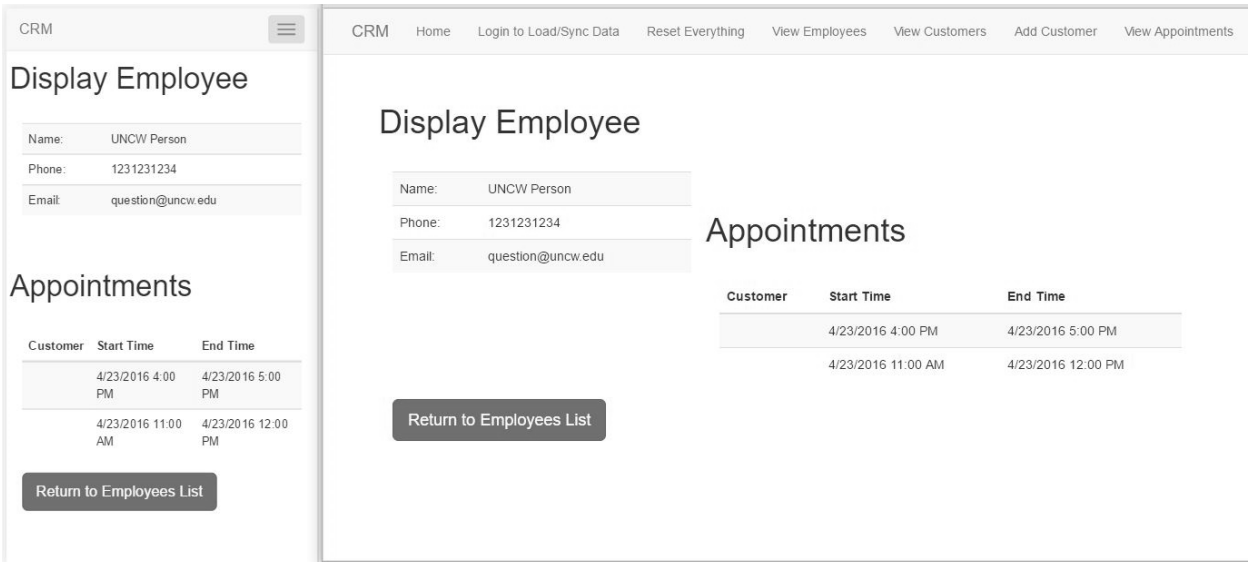

Figure 4 Side-by-side comparison of Mobile and Desktop Views

The Navigation bar is another piece of the puzzle that required some styling to make it responsive. When viewing the navigation bar on a desktop it list all the links across the top of the page but as the screen gets smaller the navigation bar takes up more and more room to display all the links. The navigation bar is on every page so it needs to be responsive and know how to fit

the page appropriately. In order to fit the page the navigation bar utilizes the bootstrap collapsible class, if it detects the screen size becoming too small it will take the horizontal links and hide them vertically on the page, then the user just needs to click a small icon at the top of the page to display the links. Figure 4 demonstrates the navigation bar collapsing as the screen shrinks and figure 5 demonstrates the vertical links used in the mobile view. This collapsible feature really enhances the navigation on a mobile device and has a lot of potential for making the application look good and easy to use.

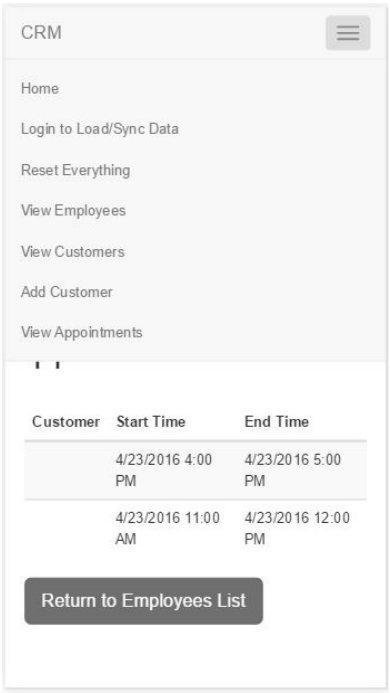

Figure 5 Mobile view of expanded navigation bar

The scheduler was the hardest thing to make responsive, if there are 8 columns, one for each day of the week plus the column displaying time, and 10 rows, one for each hour plus the header row, it means there are 80 table cells that all need to fit on a screen and 63 of them need to be easy to touch with a finger. On top of that, it needs to display some basic customer information, a navigation bar, an employee select list and instructional text just to schedule the appointment. There is no perfect scenario for responsive design while scheduling but by breaking

the calendar into three rows and limiting some text on the screen, it's functional. Bootstrap wasn't the only key to being responsive, Ember is designed to be consistent across devices so whether it is a touch screen or mouse, a Firefox or a Chrome browser all click events are handled the same way.

10.8 Review of technology used

The project required several different platforms, plug-ins and programming languages to be used in order to be developed and run. In order to develop with Ember-CLI, Node web server was download, this was only used for development. When the application is built using the Ember-CLI all files are minified and combined into 5 different files and can be served by an Apache web server. The Ember-CLI was very easy to use after downloading it just required a project to be created and some configuration options to be set. All Ember-CLI code files are created by the generator using premade blueprints. These blueprints contain just the structure and nomenclature needed to begin coding, there is no application code included inside the file just enough to not cause an error. There was no IDE used for development, instead a basic text editor and file transfer tool were used to design the code and a chrome browser was used to test. The ember application cache and local storage plugin provided access to the HTML5 API's and required very little configuration to include in the project.

JavaScript is the only programming language used to build and deploy the ember application, in order to communicate with the server a custom API was designed in PHP and communicates with a MySQL database. The API was implemented on a Linux web server in the cloud.

CHAPTER 11: COMPLETED PROJECT

11.1 Challenges/Issues

The first major challenge in the project was switching to the Ember-CLI and getting started. The Ember-CLI is very robust and has a lot of configuration options to enhance development but to install it Node had to be installed first and then the Ember-cli package could be downloaded. As a first time user it was difficult finding the correct set up for the development environment, it involved opening ports on computers, defining directory structure, and locating configuration files for set up. Once the environment was set up, it created the best web development environment I have ever used. Another Ember-CLI issues was downloading and installing packages, there were times that packages did not have up-to-date documentation or they simply didn't work or worse broke the compiler that assembled everything together. There were a few times in the beginning that a new project had to be created due to a bad or outdated package causing fatal errors in the existing project.

The scheduler was another area that was very difficult, the complexity of the functionality was underestimated. Scheduling is no simple task and to place all of that information onto a template for the user to interact with doesn't make it any easier. The deliverables for the scheduler were very limited, which thankfully meant it did not have to get too complex. As mentioned above, there is an Ember mind-set that the developer must be in when writing the code, the developer must distinguish between regular JavaScript objects that don't need to extend Ember and some that do. There were a few times there were bugs in templates causing them to not update due to a JavaScript object being used, there are no errors and no indication in the compiler, it's only obvious when templates aren't updating correctly because the data isn't bound and being updated for changes.

The last major hurdle in the development of the project was building the production code and then caching it to be used offline. In the development process the Ember-CLI watches the files and updates the developer's browser as changes occur. This means the project is broken up in to many different files across the project structure. An application cache file needs to explicitly list all the files, so it wouldn't make sense to do this until all the files are minified and compiled into a single file at the end. The first approach was to manually create the application cache file and just list the files inside of it, this didn't work. The files cached perfectly but the API calls started failing with network errors, this was very confusing, the chrome inspector would show the failed API call which allows the developer to copy the API URL and test it in a browser. The API worked fine and the Ember application worked fine when it wasn't cached but for some reason it did not work when it was cached, the user couldn't login to the application and download employee information and they also couldn't save customer information. I ended up finding an Ember Manifest Cache extension that could actually build the cache file for the project. I was very nervous downloading a plugin with the project so close to completion, I ended up cloning the GIT repository to be safe and then downloading it to the new repository. It worked perfectly, the application cache file actually came out the exact same which leads me to believe there was some kind of configuration option for caching files and still making API calls within the Ember-CLI build.

The last notable issue is the security of the application, all data is going to be stored in local storage and be viewable through the ember chrome inspector, also due to the environment not having a server certificate nothing will be encrypted. This poses a few questions about how to handle login information and sessions. There is a login service that handles all login functionality, this service stores the API token and connects to the server to authenticate the user.

If the service were to save the username and password of the user then it would be stored in local storage and then loaded each time the application starts, the local storage is local to only the machine it is on so if it's saved to a public machine then a user with bad intentions could locate the username and password easily in memory and then use it to login to the existing traditional web site which holds all company information. In order to mitigate this risk, the application does not maintain a logged in state, so when it initially loads it is not logged in, if the user wants employee information or to save customer information they must login again, this way the user can only be logged in as long as the page is open, if they close the page it forgets all login information. The user only needs to be logged in to access the API so it does not require a login to add customers or use the scheduler.

11.2 Moving to production

If this application were to be sold to a client there are a few changes that would need to take place to ensure it is ready for the average person to use. The first issue would be the server certificate, anytime an application sends or receives sensitive information over the internet it should be encrypted to protect its privacy, so switching from HTTP to HTTPS would be the first step in encrypting information. Along, with encryption the login functionality would need to be more realistic, currently if the user wants to connect to the server they have to authenticate every time meaning each API call is like a new session, the server knows nothing about previous connections and the application has no way to identify itself without a username and password. A better option would be if when the employee logs in for the first time the server saves the sessions and sends back a key to access the session. That way a username and password doesn't have to be stored but there is a link to the session on the server, then the application can just store

the session and if one is available know that the user is logged in. If the session expires, the server can check before an API call and issue an error forcing the user to login again which creates a completely new session to be stored in the browser eliminating the need to authenticate with every API connection.

There are many areas of the web application that are limited just because they were defined in the scope or it made things easier. The user would need a configuration panel so they can configure the application to fit their business rules better some examples of this would be the number of days shown on the calendar, what time the calendar starts and stops, the length of appointments, or how the information should be represented when loaded back into the existing system. Currently there aren't any configuration options but it would be necessary for setting some things up like hours of operation. The application would also need to tie into existing systems to utilize all functionality, for example on the tradition scheduling form when an appointment is scheduled an email goes out to the customer and employee to let them know about it, small features like this would need to be incorporated into the application.

Currently the application only loads employees and their appointments, if the scheduler were to be used multiple times there's a risk of entering a person that is already saved to the server but there is no indication because the application doesn't load existing customers. The application would need to load old customer information, that way if an existing customer is found while offline they can be handled appropriately. The scheduler would also need to be worked on slightly to make it more user friendly, when designing the scheduler only six employees were used with around two or three appointments each, if the scheduler had twenty or thirty employees each with full schedules it would become difficult to see which employees were already scheduled in which time slots and who they were scheduled with. It would need some

sort of template for displaying who is in a time slot. Another concern when moving the application to production is the size limitation. While developing this application there haven't been any size concerns, in my research I was able to load thousands of customers into it without problems, as the scope of the application increases the size of the data and load speed will need to be monitored.

11.3 What would make this perfect?

There is a difference between making something production ready and making something perfect, we should always strive to write perfect code but this is the real world. If there were more resources to use on this project these are things that would be added to it. The changes in the sections above would obviously be added but so would a few more features that enhance the application. The application was built to only communicate with the server if the user tries to load employee information or save customer information. Once it loads employee information it can't load it again and once it syncs a customer it can't edit that customer again, for the application to work perfectly with the existing system it would need to work in online and offline scenarios. If the application is online it could use live data whenever possible and when offline limit the user to just the offline portions of the application. If the application can detect when it is connected it can make the appropriate server calls to send or receive data, when it's not connected it can just use cached information. The cached information could then be compared with live information the next time the user is online.

Due to the time and different scenarios of scheduling, the application only works if customer and employee information is available. A customer can't have an appointment scheduled if there are no employees and an employee can't view the scheduler unless they have a

customer ready. In order to make the scheduler more robust, it would need to be able to add appointments without employees and also view an employee's schedule without a customer loaded. This way if a user forgot to load employee information before going offline it would still function. Also, if the scheduler works without a customer, it could be used by an employee to view their schedule for the day, and if it's cached yet always updated while online it should always be up to date. The responsive design could also use some work, when going from a large screen to a small screen it helps to hide some information and leave only what is necessary. The navigation bar could become much smaller and easier to use if recognizable icons are used for navigation on a small screen.

CHAPTER 12: CONCLUSION

Overall the Customer Resource Manager turned out great, it meets all functionality and works on most devices. The HTML5 API's and Ember framework created a decent offline web application that feels native to the browser and runs perfectly in an offline scenario. The scope of the application and the amount of time available limited the functionality to only what was listed in the deliverables, with more time and specifications the application could be used in place of a main system in WM for scheduling appointments in any scenario not just offline. Modeling and displaying relational data is very straight forward, once CRUD pages are built for one entity it's easy to understand how to build CRUD pages for all entities.

12.1 Future Work

Now that I understand Ember development and what it takes to create a responsive offline web application, I can think about my future work and how ember can assist me. At Quintify, we use the WM generator to generate all CRUD pages for our web sites, WM generates very trivial CRUD pages for accessing and manipulating data. If WM is extended to produce not only PHP but also Ember-CLI project files it could be an easy way to transition into a hybrid application that is part Ember. This would not be for offline use, this would help make the existing traditional web application faster and seem more native by loading all templates at once and routing to them as needed. Obviously there would be a lot of trial and error with designing a code generator that generates ember but it would be worth researching.

When developing this application I relied on an offline mindset, I only have access to data loaded before going offline, if I remove the offline restriction then my application could incorporate more technologies one example would be on the scheduler, when a customer is added it is required to gather all their address information, if the application made an API call to google and had every customer's longitude and latitude it could then make suggestions about which customers should be scheduled near each other. The google API could even let the employee know how long it takes to get from one customer to another based on time of day and traffic patterns.

This research was very interesting to me and I actually think web applications of the future will be engineered with a hybrid mindset. By utilizing a client side framework an entire web application can be loaded on the initial connection and then only needs to worry about loading the data and rendering the templates as needed. This means a web application would use less data than a traditional one, and it doesn't need to reload the DOM or JavaScript libraries

after each new page. If the application is cached then the browser never needs to load the template or code, it just loads the latest data. These types of applications have potential to change the approach toward web development, by utilizing cache to reduce the amount of data sent between the browser and client, and it provides a more native feel that performs faster than traditional web sites. As this technology evolves it will be interesting to see how far it goes.

CHAPTER 13: RESOURCES

1. Build software better, together. (n.d.). Retrieved December 9, 2015, from https://github.com/about

2. Angular/angular.js. (n.d.). Retrieved December 9, 2015, from https://github.com/angular/angular.js

3. HTML enhanced for web apps! (n.d.). Retrieved December 9, 2015, from https://angularjs.org/

4. MisÌŒko Hevery and Brad Green - Keynote - NG-Conf 2014. (n.d.). Retrieved December 9, 2015, from https://www.youtube.com/watch?v=r1A1VR0ibIQ

5. Jashkenas/backbone. (n.d.). Retrieved December 9, 2015, from https://github.com/jashkenas/backbone/

6. Backbone.js. (n.d.). Retrieved December 9, 2015, from http://backbonejs.org/

7. Ashkenas, J. (n.d.). Jeremy Ashkenas - Taking JavaScript Seriously with Backbone.js. Retrieved December 9, 2015, from https://www.youtube.com/watch?v=4udR30JYenA

8. Emberjs/ember.js. (n.d.). Retrieved December 9, 2015, from https://github.com/emberjs/ember.js

9. Ember.js: A framework for creating ambitious web applications. (n.d.). Retrieved December 9, 2015, from http://emberjs.com/

10. Best way to handle loading thousands of records. (n.d.). Retrieved December 14, 2015, from http://discuss.emberjs.com/t/best-way-to-handle-loading-thousands-of-records/6087

11. All Things Open 2014 | Yehuda Katz | The Ember.js Framework - Everything You Need To Know. (n.d.). Retrieved December 9, 2015, from https://www.youtube.com/watch?v=f-6Qd3nuv4w

12. Can I use... Support tables for HTML5, CSS3, etc. (n.d.). Retrieved December 9, 2015, from http://caniuse.com/

13. The HTML5 test - How well does your browser support HTML5? (n.d.). Retrieved December 9, 2015, from https://html5test.com

14. Tastejs/todomvc. (n.d.). Retrieved December 9, 2015, from https://github.com/tastejs/todomvc

15. Web Storage (Second Edition). (n.d.). Retrieved December 14, 2015, from http://www.w3.org/TR/webstorage/