Ecocion

Facility Management System

Alex Anderson
Niles Hacking
Ryan Shipp

June 16, 2015

Table of Contents

1. Introduction

1.1 Client Description

Ecocion is a company that serves to help companies comply with environmental, health, and safety regulations. To do this, Ecocion developed a software package called ACTS. ACTS stands for Asset and Compliance Tracking System. Originally this program was implemented as a desktop application where one could enter all of their company's information including facilities, areas, equipment, workers, etc. ACTS comes with many rules and regulations pre-installed in the database and the ability to add new regulations as needed. Tasks can be put into the system to help users comply with the regulations. ACTS will notify the user of their progress towards these tasks put into the system through e-mail and from within the app.

Ecocion is currently in the process of expanding and re-implementing ACTS as a web application called ACTS O&G (Asset and Compliance Tracking System: Oil and Gas). This new version is largely the same software operating on a new platform; however, some small improvements are being made to the system including a more flexible design and the ability to directly stream data from a piece of equipment into the system.

1.2 Product Vision

For our field Session project, we assisted Ecocion in moving ACTS Classic onto the web in the form of ACTS Oil and Gas (O&G). Specifically, we worked on the facilities page. The facilities page is where all of the information regarding facilities is put into the database. This information includes the facility name, staff, location, associations with other facilities, and other general information. We implemented all of these tabs in the same manner as the existing O&G pages.

2. Requirements

The page we made needed to be able to organize a company's facility data and make it easy to access and update. It should contain a series of tabs (Address, People, Phone, Regulations, Projects, etc.), each with various fields displaying information from the client's database. We implemented a complete web application as a series of layers, from the TypeScript presentation layer to the C# business logic and data access layers. Our project was to be as close in functionality to ACTS classic (as shown in figure 1 below) as possible.

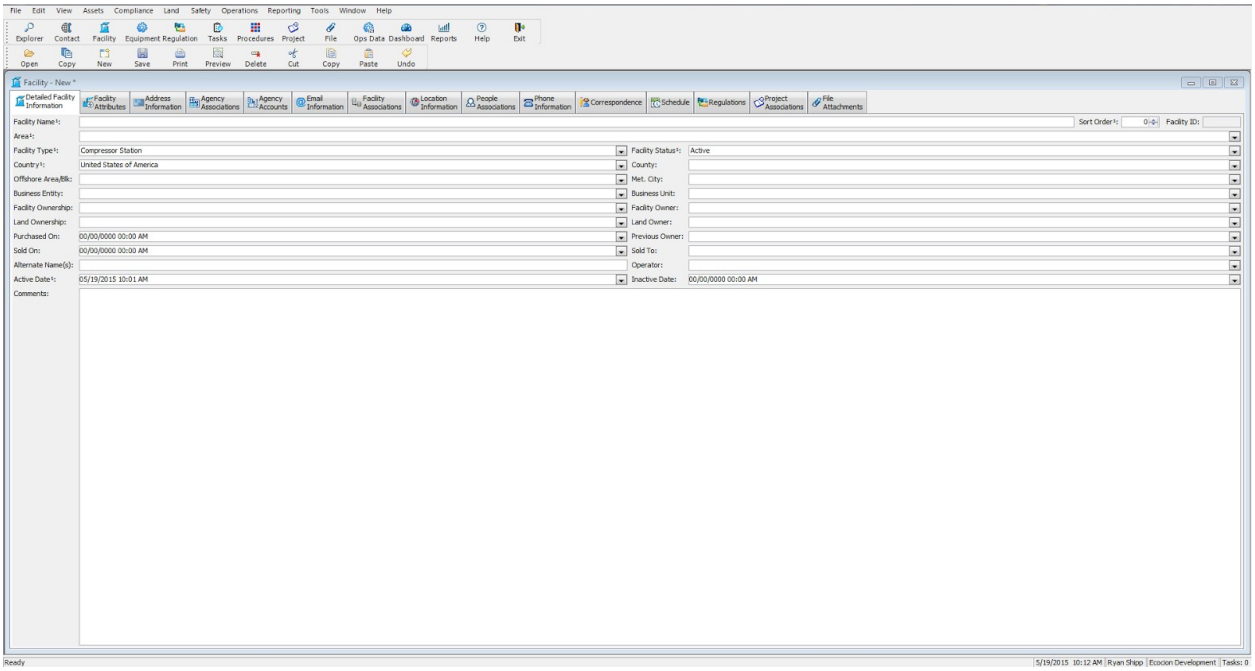

Figure 1: The Details tab on the Facilities page of ACTS Classic.

2.1 Functional Requirements

The functional requirements of our project are:

- Include tabs for: details, attributes, addresses, agencies, agency accounts, email, facilities, locations, people, phones, correspondence, schedules, regulations, projects, and files.
- Include a grid on each tab's page as required. This grid is similar in style to an Excel spreadsheet, and presents data from the database.
- Allow basic CRUD (create, read, update, and delete) operations on the database tables.
- Make sure the application is functionally and visually similar to the desktop app.

2.2 Non-functional Requirements

The non-functional, or technical, requirements of our project are:

- Use Ecocion's APIs (including TypeScript and Entity Framework APIs).
- Use C# for the data access layer and business logic layer, which will share the responsibility of interacting with the database.
- Our project will have a presentation layer with all of the tabs and text fields. This serves as the user interface for this project. It must be written in TypeScript.

3. System Architecture

3.1 Overview

Our project is a web application, and can be broken down into components in a few different ways. We will focus on the "layer" abstraction introduced earlier in the Requirements section (figure 2). We can split the application into four main layers: the presentation, business logic, data access, and data source layers. The presentation layer is the user interface (UI), which delegates CRUD (create, read, update, delete) operations to the layers below. Our presentation layer will be written in TypeScript. The next two layers, the business logic layer and the data access layer, will share the responsibility of moving information back and forth between the database below and the presentation layer above. The business logic layer will be written in C#, with LINQ queries and Microsoft Entity Framework serving as the data access layer. The final layer is our Oracle database, which stores information and is able to serve it up to the data access layer as requested. These layers are shown in Figure 2.

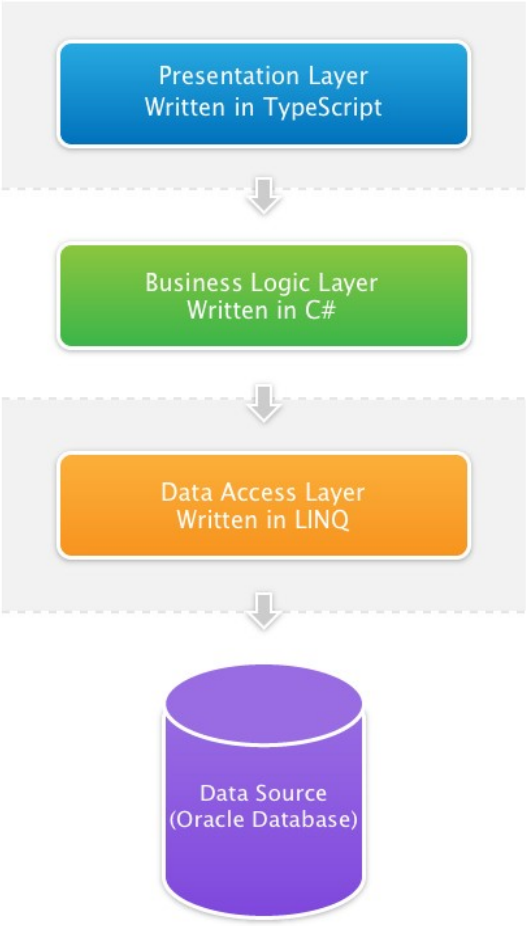

Figure 2: A diagram showing the layers of our web application. This diagram shows the traditional layers of a webpage, with specific details about each layer in relation to our project.

3.2 Presentation Layer

The UI of our web application is written in HTML, TypeScript, JavaScript, and CSS. Since the application must integrate into the rest of Ecocion's system, we used their API to automatically generate HTML pages with the correct CSS and JavaScript styling, allowing us to focus solely on the TypeScript portion that defined the main UI elements on the page, and how they interact with the business logic layer's web API.

3.3 Business Logic and Data Access Layers

The majority of our time on this project was spent in the business logic and data access layers. This is where all the database interaction is done, broken up into what we called "reads," "saves," and

“deletes.” For each text field or dropdown in the UI, we created a method in the Facilities “controller,” the part of the web application that decides what to do when the browser requests a certain API endpoint. For example, when the user opens the “Details” tab on the Facilities page, JavaScript running in the browser might make an AJAX call for the “Facility Owner” dropdown that looks something like this:

```
GET /acts/api/facilities/FacilityOwner
```

The request gets routed through the web server to the application, which sends it to the Facilities controller, then looks for a “FacilityOwner” method in that controller. The FacilityOwner method sets up an object with a list of options for the dropdown, and the currently selected option’s position in the list, and sends it back as a JSON object to be rendered into the actual dropdown by JavaScript back in the browser. These “read” methods allow us to populate fields in the UI when the user opens an existing facility record.

Saves and deletes are tied to their respective buttons in the UI. When the user clicks “Save” or “Delete,” an AJAX call is sent that ends up at the “SaveFacility” or “DeleteFacility” methods in the Facilities controller. The save method then calls individual functions, each of which saves a certain object (corresponding to one tab in the UI) from the facility record. This allows us to also include an auto-save feature in each tab, so when the user clicks off a certain UI element, the JavaScript can detect this, and send another AJAX request to save just the information in the tab the user is working on. The delete method simply deletes the record from the Facility table, clearing up any referenced objects from other tables along the way when appropriate.

3.4 Data Source Layer

The data source for this application is Ecocion’s Oracle database server. The schema for this database, which describes the layout of the tables and their columns, was written by the company’s founder for use with ACTS Classic. We had to become familiar with its layout during the course of our project because it played a large role in the way the business logic and data access layers had to be written. We learned a great deal about relational databases and how primary and foreign keys tie everything together. The Facilities model, which is the part of the database we worked with the most, is a collection of interrelated tables (figure 3).

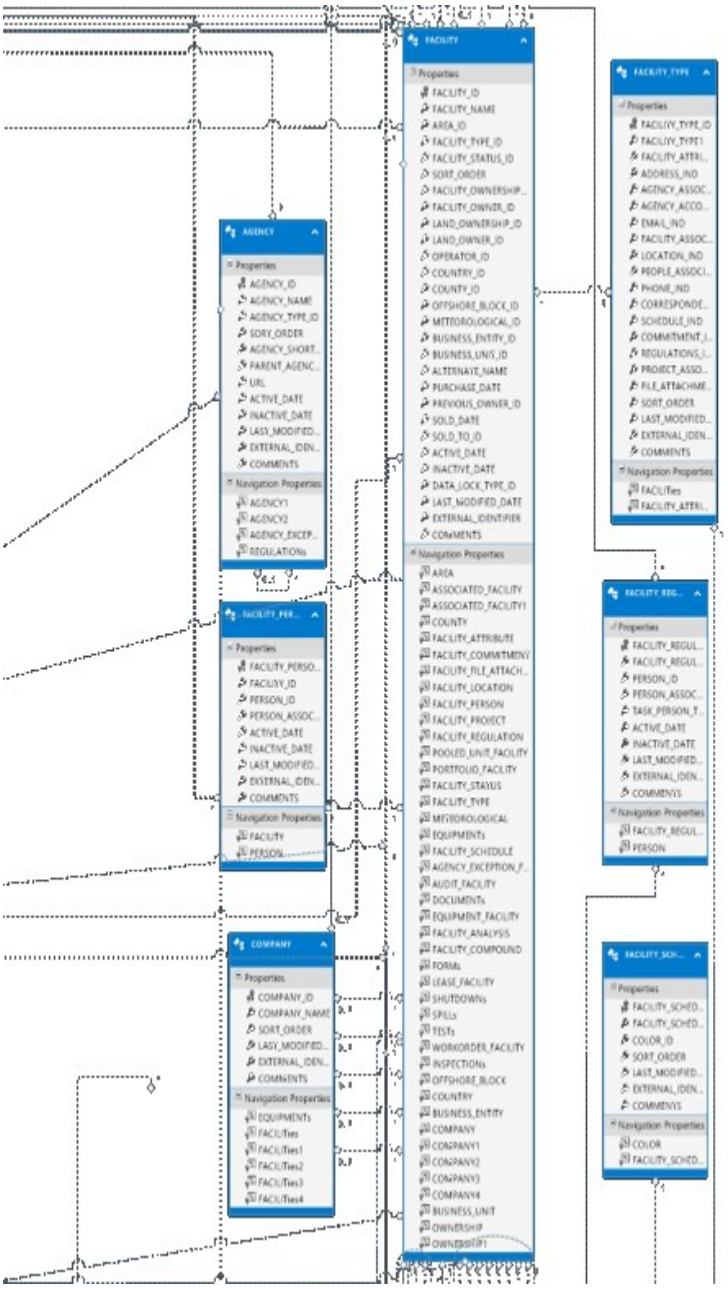

Figure 3: Part of the Facilities model in the Oracle database, including the FACILITY table itself.

4. Technical Design

In order to make our project easier to code, we used LINQ queries rather than SQL queries. By using LINQ rather than SQL, we were able to write more maintainable and secure code at a faster rate than we would have if we were writing SQL. We also used Microsoft Entity Framework to allow us to use data models that represent database tables within our code, and this allowed us to use dot operations to

access different parts of the database rather than pesky SQL joins. By using these technologies together, queries made to the database became much shorter and more readable. Figure 4 below is a diagram outlining how an application communicates with a SQL server through LINQ. As shown in the diagram, the application uses LINQ queries that are then converted to SQL queries which are then given to the server.

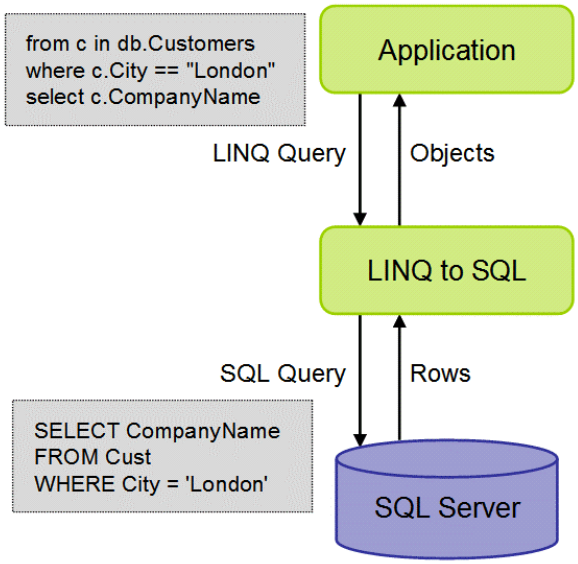

Figure 4: LINQ and SQL Diagram (from http://codesamplez.com/database/insert-update-delete-linq-to-sql)

Saving associations between two facilities was an interesting issue. Initially we just wanted to put an associated facility column into the existing facilities table, but many problems were found with this solution. One problem was that every facility would need to have some value for associations regardless of if it actually has an association. This value could just be set to null, but it would be easy for this to be accidentally changed creating a fake facility association. Another issue is that this structure would limit each facility to only having one association. A third issue is that there is no easy way to store any additional information about an association. This would either require even more columns added to the facilities table, or multiple different types of associated facility columns being added to the facilities table.

A better solution to this problem was to add an entirely new table to the database for facility associations such as the one shown in Figure 5. This new table has two columns for the primary and secondary facility ID, as well as columns for comments and other information about why the facilities are associated. This solution has none of the problems that would have been introduced with our initial solution. Associations are not directly in the facilities table, so there is no one column that could create

a fake association if it were changed. Facilities can have multiple associations by simply having two rows in the associations table with different secondary facility IDs. The associations table also provides a place where additional information about an association can be stored without forcing every facility to have these columns. Similar tables are used in our project for associations between facilities and other things such as phone numbers, emails, and equipment. These tables allow new associations between existing objects to be made without changing any of the objects.

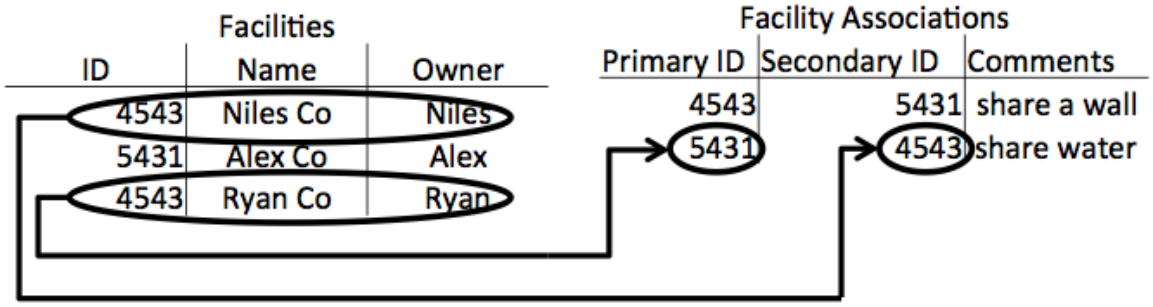

Figure 5: Niles Co and Ryan Co are associated in that they share the same water. Looking at the facility associations table tells us this because both Niles Co's and Ryan Co's ID numbers appear on the same row along with the comment "share water".

5. Design and Implementation Decisions

Based on the UI text generated from our variable names, Ecocion's existing conventions, and the web framework's own conventions, we decided on naming conventions for all variables, database tables, and methods in our page. Methods are given descriptive names using CamelCase, variables with babyCamelCase, and references to database tables in ALL_CAPS. We also agreed on specific pluralization rules to keep everything consistent.

We decided to include name, ID, and owner fields in the search modal, so users can search by any of these pieces of information. Name was an obvious choice, and we found out that the short numeric ID was often easy enough to remember that people would use that to find the facility they were looking for, without having to scroll through several facilities with similar names. The owner field is also a major advantage, as it allows users to filter by the company that owns the specific facility they are searching for, perhaps if they have forgotten the name and ID.

The “grid” component of some of the tabs needs to change based on the number of columns in the specific table. Based on the tables system used in the app, we found that the page was broken into 10 units, regardless of window size. We then made individual size decisions for each column in the grid based on the average length of the data contained in that column, making sure the total size of the columns on a single tab was *at least* 10. We were able to optimize the sizing of these UI elements to make the best use of screen real estate, and not overly crowd the interface.

Several of the fields in our grids needed to contain multiple pieces of data. For example, a “name” cell might need to be structured as “LAST_NAME, FIRST_NAME MIDDLE_INITIAL.” (eg, “Doe, John H.”) While this is a relatively simple example, some of the other cells were a bit more complex. We made decisions for each of these on what information would be most useful to present to the user. “Regulations” was one of these more complex fields, which we ended up presenting as “REGULATION_NAME (REGULATION_TYPE, REGULATION_STATUS).” “Regulations” contained several other fields (comment, description, agency information…), but by looking at the information stored in each of these fields and thinking about the application from the user’s perspective, we were able to pick the fields that were used by all regulations, and contained information that would help describe the regulation to the user in the context of the grid.

6. Lessons Learned

We learned several valuable lessons while working on this project:

- The way tasks are divided between team members makes a significant difference in the quality of work and the speed at which it is completed. If tasks are not divided equally, some team members are left waiting for the others to finish. If tasks are not divided intelligently, a team member might waste time by completing the same task that another team member has already completed. Communication and intelligent divisions are vital at the start of a task in order to maximize efficiency.

- Communication is key on a team task. If the team works silently on a task and only reports to each other when a task is completed, one team member might not have understood the task as well as the others and will have errors in their code that would not have been written if more communication would have occurred while working on the task. Communication is key to be sure that everyone is on the same page.
- Pair programming is extremely useful. Developers end up writing better code at a faster pace just by having another developer looking over their shoulder. This can be great for a developer that is less experienced as well; having a more experienced developer look over your shoulder can be a great way to learn and can make you feel more confident about your code.
- Don't be afraid to ask questions. If developers are afraid to ask questions, massive amounts of time can be wasted trying to figure out a problem that could be solved in seconds by a more experienced developer. By asking plenty of questions, developers spend more time on difficult problems and less time on trivial ones, and their overall efficiency is increased tremendously.

7. Results

7.1 Overview

Overall, our project was a success. We finished on time and completed all of the features required by the client, along with a few extras. The page we created has all of the functionality of the original "ACTS Classic" facilities page, in a much cleaner and more accessible interface. It allows users to create, edit, and delete facilities, as well as search through and view existing facilities. Where Classic used simple dropdowns, we have autocomplete dropdowns that update dynamically via AJAX as the user types. All data loads and saves also go through AJAX calls to the endpoints we created using Ecocion's existing RESTful API. All forms are validated and have clear error messages when the user enters invalid information or leaves a required field blank.

7.2 Testing

To test our project, we used a variety of different browsers including Microsoft Internet Explorer, Google Chrome, and Mozilla Firefox. We did not encounter any differences in the functionality of our page between these browsers. We also tested our project by creating, reading, updating, and deleting a variety of facilities. We tested whether facilities were actually being created, updated, and deleted from

the database, and not just from our interface. We also made sure to test edge cases, such as entering data for required fields only, entering data for all fields, and leaving required fields blank to check form validation. We found a number of bugs in our testing, and were able to fix all of them.

Figure 6 below shows the completed Details tab of our project.

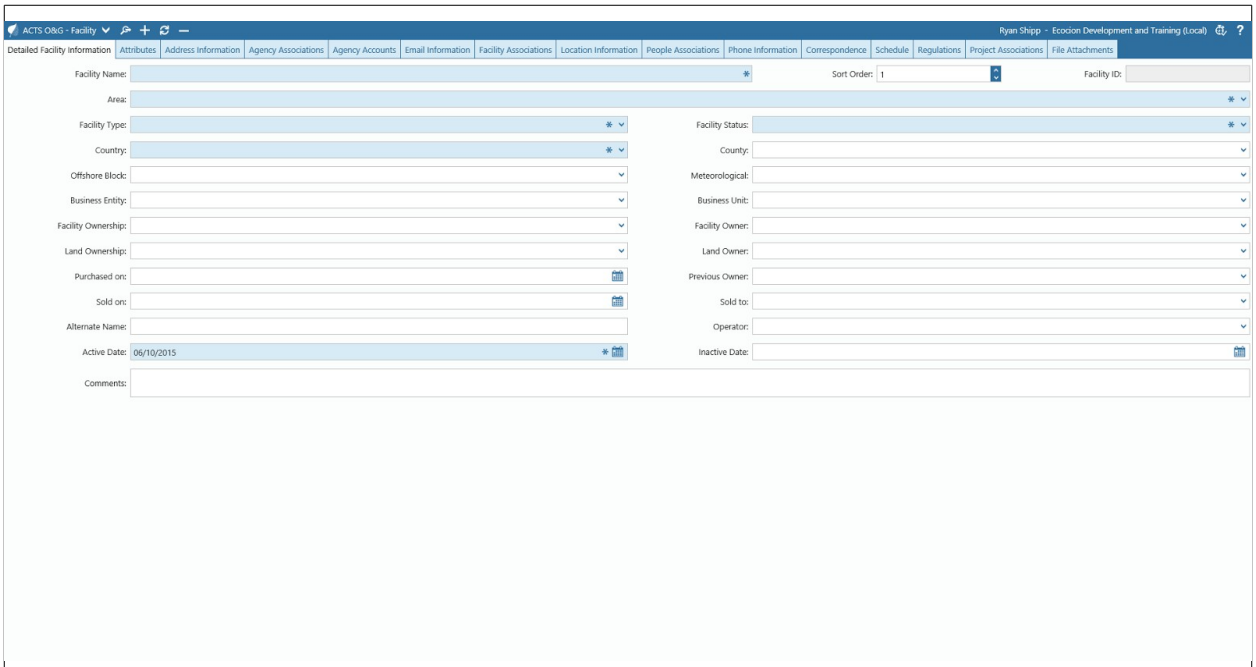

Figure 6: The details tab of our ACTS O&G Facilities page.

7.3 Features We Did Not Have Time to Implement

There were a few optional features that we did not have time to implement. One of these was adding custom latitude/longitude fields to the form, which could have used a special UI element that would have made it easier for users to enter these values. We left these as plain text fields, which still allowed users to enter valid data. We also could have dynamically added extra information to the "person" dropdown for the person records that contained the extra data. Instead, we left it just showing last and first name and middle initial. These features were low priorities for our client, so we made sure to focus on more important features first.

Appendices

Appendix A: Testing

In order to test our project, we created a variety of facilities and made many updates to them. After creating/updating facilities, we would check the database entries for these facilities to ensure that they were properly created/updated on the database side. We also spent time deleting facilities and ensuring that they were properly deleted from the database. We also heavily tested the interface to ensure the proper fields or dropdowns were required before a facility would save and that the user would not be allowed to save entries before these were filled/selected. We also spent time testing edge cases by entering large values into various fields to ensure that the database would accept them successfully.

We also tested our project in a variety of browsers. The browsers we tested on were Microsoft Internet Explorer, Mozilla Firefox, and Google Chrome; together, these three browsers cover nearly 95% of internet users. Our application ran exactly the same in all of these browsers, which ensured that most users would not experience any problems or inconsistencies in the experience of our web application depending on the browser they chose.

Appendix B: Development Environment Description

The development environment we used for this project was end-to-end Microsoft. The computers we used ran Windows 8 and the IDE we used was Microsoft Visual Studio 2013. The version control we used was also from Microsoft and was integrated into Visual Studio. We primarily used Microsoft languages and frameworks to create our project as well, including TypeScript, C#, Microsoft Entity Framework and LINQ.

Appendix C: Further Reading

- For further reading on Microsoft Entity Framework https://msdn.microsoft.com/en-us/data/ef.aspx
- For further reading on LINQ https://msdn.microsoft.com/en-us/library/bb397926.aspx
- For further reading on Oracle Databases https://www.oracle.com/database/index.html