C868 – Software Capstone Project Summary

Task 2 – Section C

Capstone Proposal Project Name: **Time Cutter – Workforce Management**

Student Name: **Jesse Anderson**

3 AM Productions

Software Solution

Time Cutter: A Workforce Management System

Table of Contents

Table of Contents 3
A. Application Design 4
A.1 Design Document 4
A.1.1 Class Design 4
A.1.1.1 The Owner 4
A.1.1.2 The Supervisor 7
A.1.1.3 The Employee 8
A.1.2 Class MVC Model and ERD Design 9
A.1.3 UI Design 11
B. Testing – Unit Design Test Plan 15
B.1 Introduction 15
B.1.1 Purpose 15
B.1.2 Overview 15
B.2 Test 6Plan 15
B.2.1 Timeclock for a Task at a Job 15
B.2.1 Sending User Timeclock to Owner/Supervisor 16
B.3 Specifications 17
B.4 Procedures 21
B.4.1 Timeclock for a Task at a Job 21
B.4.2 Sending User Timeclock to Owner/Supervisor 22
B.5 Results 23
B.5.1 Timeclock for a Task at a Job 23
B.5.2 Timeclock for a Task at a Job 23
C. Source Code 24
A compressed version of the source code can be found in the file workschedulersolution.zip 24
I. Sources and References 25

A. Application Design

A.1 Design Document

A.1.1 Class Design

A.1.1.1 The Owner

The class diagram is representative of the early structure the application took during its planning phase. The structure has continued to morph some beyond this initial diagram to include necessary constructs not initially foreseen for project completion. Using these structures and relationships you can see how the classes came together to form the program. This section showing class design directly correlates to the early modeling of the high-fidelity wireframes. As some functions cross user types and some are user-specific, I will cover the generic functions with the owner pages.

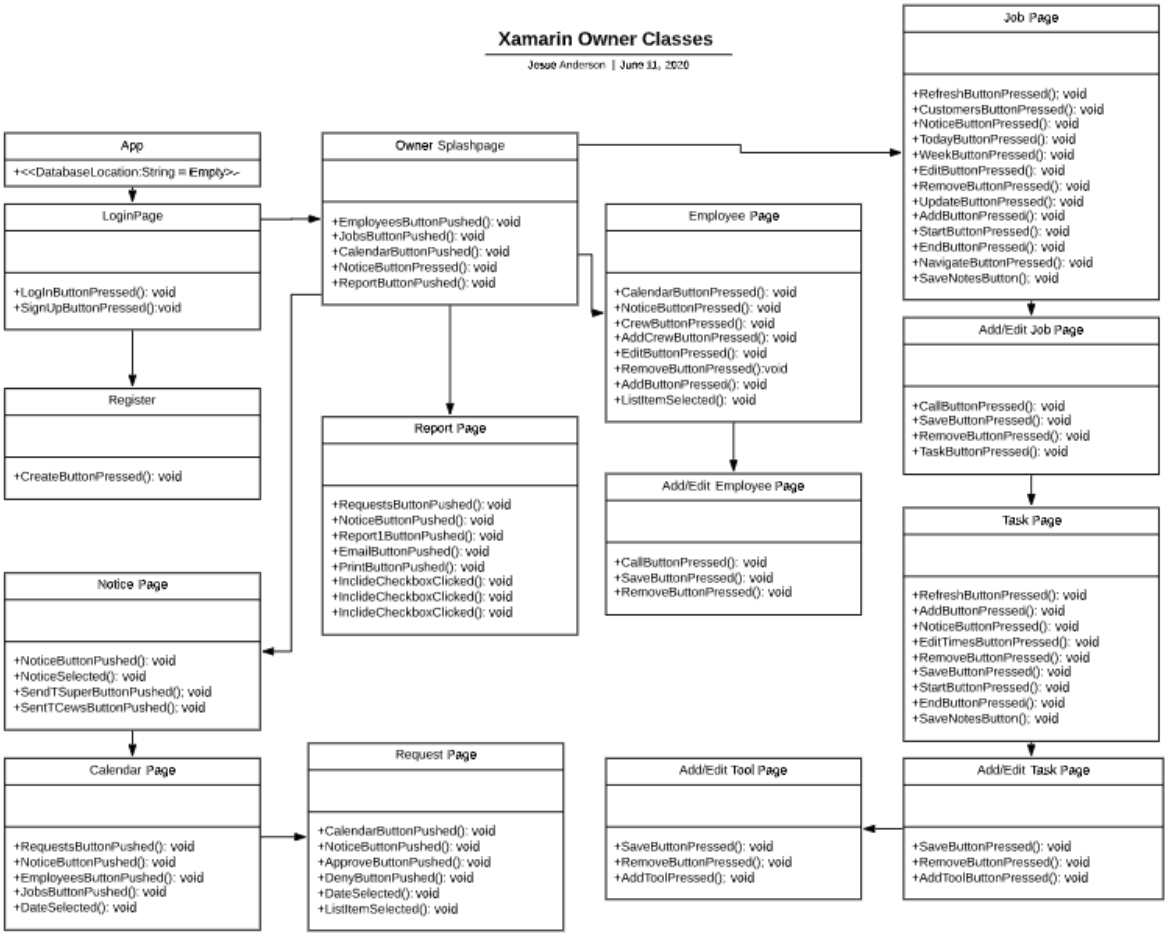


Figure 1 Planning Xamarin Owner Classes

The application has many classes and follows the MVC (Model View Controller) model of architecture. Under the View aspect of MVC, I developed out the classes to individually represent the types of users who would be accessing the Database. Further, many pages that shared functions were broken down to prevent pages from being overly complicated. For instance, the Task Page also includes scheduling the Task, or as we would consider it, scheduling the job. So as the Task Page was developed, it became secondary to the necessity of scheduling the job. The Task Page became part of the flow of

the scheduling that included a Scheduling Page. As the functionality changed, so then the View classes changed to reflect the underlying Model.

The project was developed in Xamarin Forms to allow cross-platform development for Android and Microsoft devices. The goal is to create an application that allows the users access to the database and work information at the palm of their hands. Each platform requires some necessary changes to provide platform-specific functionality, like navigation or calling features. IOS devices are included in the design for future platform development but aren't included in the scope of this project as the developer owns no IOS devices to test on.

The classes listed in the above graphic are for the 'owner' user, who has the greatest direct access to information from the database. As this means additional functionality it also means greater repeated access to the database. This created a lot of overhead from repeated database queries per page. Another example of the changes from these early models to the current model was the breaking up of each database query into a specific query type that would cover only the functionality included in the Class View. For instance, if a page was for notices/messaging, it would only query the aspects of the database necessary to provide the information for that function. Once the user switches to a different page, the FormModel class that utilizes the class types in the ERD would only query for those pages' needs. This way the database would be queried for all information upon login, but only update the iLists that are directly relative to the page content the user is interacting with.

All groups of class pages have a Login Page that accesses the database and checks for matching usernames and passwords. If the user hasn't registered yet they can register themselves as a new business owner or register themselves as a worker in another's business on different individual pages. As the user then logs in the database is accessed to return the information for that specific database. The first login also brings the user to the profile page where they will fill in additional information relevant to the application's use. This page is again accessible from the first page after login in the ellipsis at the top.

The splash page for each type of user varies slightly due to their permissions, but all splash pages show the jobs that are their responsibility that day. The owner will see all jobs that day, but a worker and supervisor logging in will only see jobs assigned to them that day. This page is updated every time they visit it, so if an owner wishes to change the order of the jobs, or add or remove jobs from the list, the employees can see the changes each visit.

The owner can access their employee's page from the splash page where they can add an employee or modify crew information. An employee can be on multiple crews and the crew information is saved on the owner's device in a SQLite database. This is because the crew information is a programmatic solution for navigating a rapidly changing workplace structure. Employees may move between crews throughout the day, and combinations of employees into crews may only exist when directly related to certain jobs. For example, all employees may be on hand for a huge job at the start of the day, before splitting into 2 groups for a large job and a medium job. Then, once the medium job is finished, the owner may reconcile some employees over to the large job, but send a small, 2-man crew of employees to a distance job an hour away that doesn't require a full crew. This ability to schedule those dynamic changes ahead of time allows the owner to foresee rapidly changing crew groups and isn't a necessary query to add to the overhead of database use.

From the splash page, users can access the jobs page, which will display all scheduled jobs. Supervisors and the owner will see all jobs scheduled and if they are assigned to any workers, this is where the workers will only be able to see jobs assigned to them. There are also notes that can be added to the database for any job to explain things relative to that job. The notes can be made by any user type but only deleted by the owner. The owner is also able to delete a scheduled job or to edit and add workers to it. This will open a page that shows the workers grouped by crews they are in. Employees can be added to a job individually or with the add crew button. This will include some repetition of worker names, as different crews may include the same workers, but it will not add an employee to a job twice. The page also allows for phone and navigation features for Android and Microsoft devices.

Specific to the owner is the ability to access the chain of pages following the scheduled jobs page where they can modify a job's and customer's information. This includes the ability to add, edit, or remove customers. As a customer is added, the owner can add a job to that customer and choose to import the customers' address or use a new address. This creates an unscheduled job, like a class that hasn't been instantiated. Instances of that job can then be created and scheduled on an individual or reoccurring basis. Scheduling the job also allows the owner to indicate what tasks and tools will be used that job and how much each task or tool will cost the customer for use per hour. The owner can return to the jobs page to see the job scheduled and add workers so they can access the job and record the hours worked.

As employees are not always available to work, and an owner has frequent requests for vacation days, a calendar is included to allow employees to request off days, and employers to block days scheduled to be high demand workdays. Adding this functionality with generic reasons allows employees to have privacy in their requests, and the employer to grant or reject requests without lengthy conversations and reminders. This calendar displays the jobs for that day and requests 'off' so employees know if that date has already been requested off by another employee.

At the bottom of the splash page, and on most pages is an ellipsis, is access to the notifications button. This allows the user to view messages sent to the crew by the owner or supervisors. This is included so that in future development users can be notified when calendar requests are approved, they are scheduled for a job, their work hours, a jobs' hours and completion, or job relevant information they'd like to pass on. Currently, the notices page only supports chat and passing work hours.

With all this information the owner also has a simple reporting function that can export to PDF on android for email or use with external high-end business reporting software. The reporting includes hourly income to date for specific jobs and information relating to employee attendance.

A.1.1.2 The Supervisor

The supervisor chart below is an intermediate step between a general employee and the owner. It carries less access than the owner, not having the ability to create, modify, or delete customers or jobs. They can access their individual profile information. It does have access to recording times for jobs and seeing notices intended for a supervisor and owner access. They also cannot block calendar dates to modify employees or delete scheduled jobs. The supervisor gains access to the time clock function for them to record their hours worked.

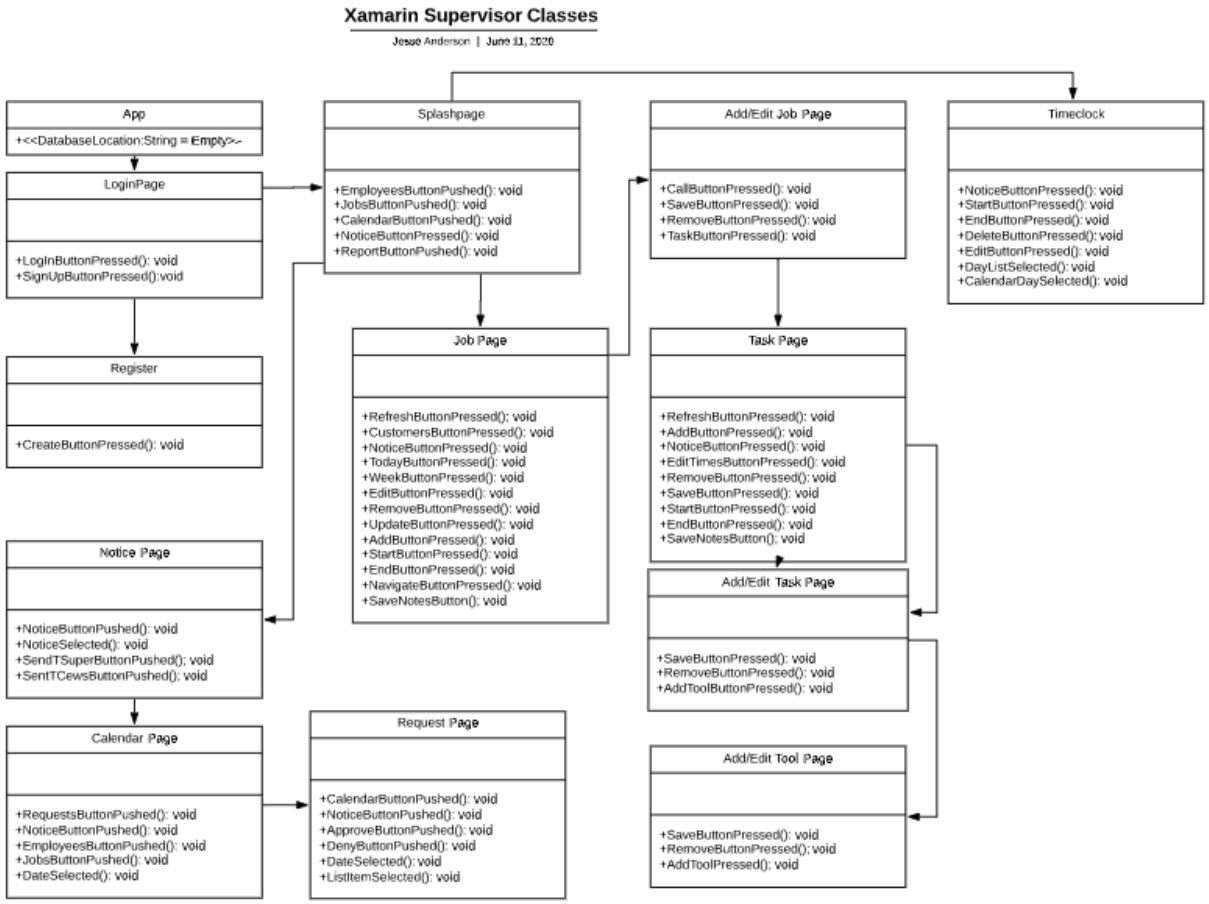


Figure 2 Planning Xamarin Supervisor Classes

A.1.1.3 The Employee

The worker pages have the least amount of access to the database. They have a splash page that shows jobs for the current day that have been assigned to them. The worker can access their individual profile information as mentioned above. They can also navigate to the 'Scheduled Jobs' page where they can add notes to a job. They have access to the calendar for requests and can view upcoming jobs that the company will be working on. The employee and the supervisor can record the hours worked per task per job. The employee can also create timeclock records and send them to the supervisors.

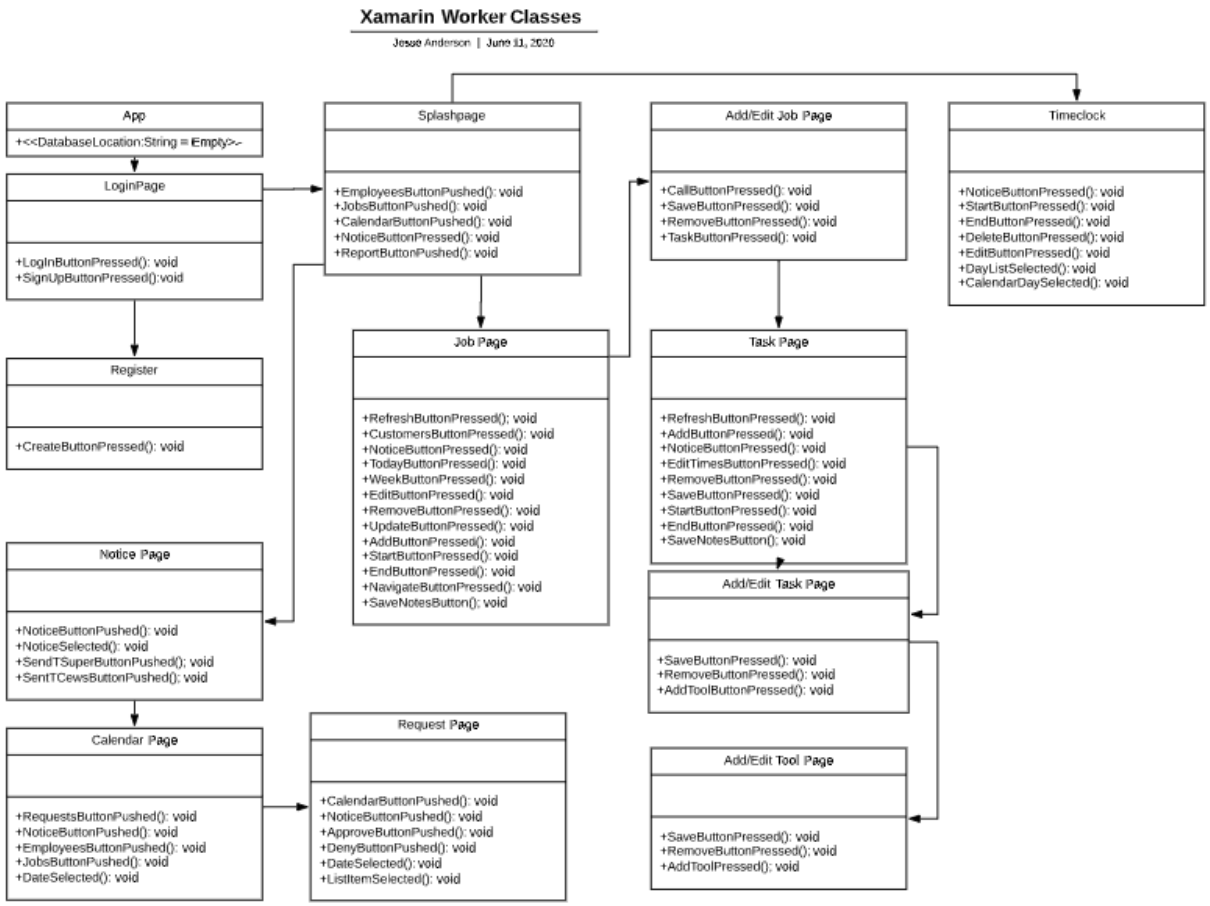


Figure 3 Planning Xamarin Worker Classes

A.1.2 Class MVC Model and ERD Design

The ERD design for the database displayed below is also an early design during the planning phase of the application. Each database entity (table) is directly reflective of the class Model of the MVC. The class was created to instantiate objects in the program and those objects are passed directly to the database. As the application expanded the form held to the architecture to create additional classes in the model for accessing the database. As the classes for accessing the database expanded, I opted to create subclasses that would contain more complete information to display the information in the views. While it Is possible to create an abstract object and use the objects array value to assign the combined database queries without created the subclasses, I found it simpler for future development to have properties that are specifically named and referenced to pass values to Listviews and other display formats.

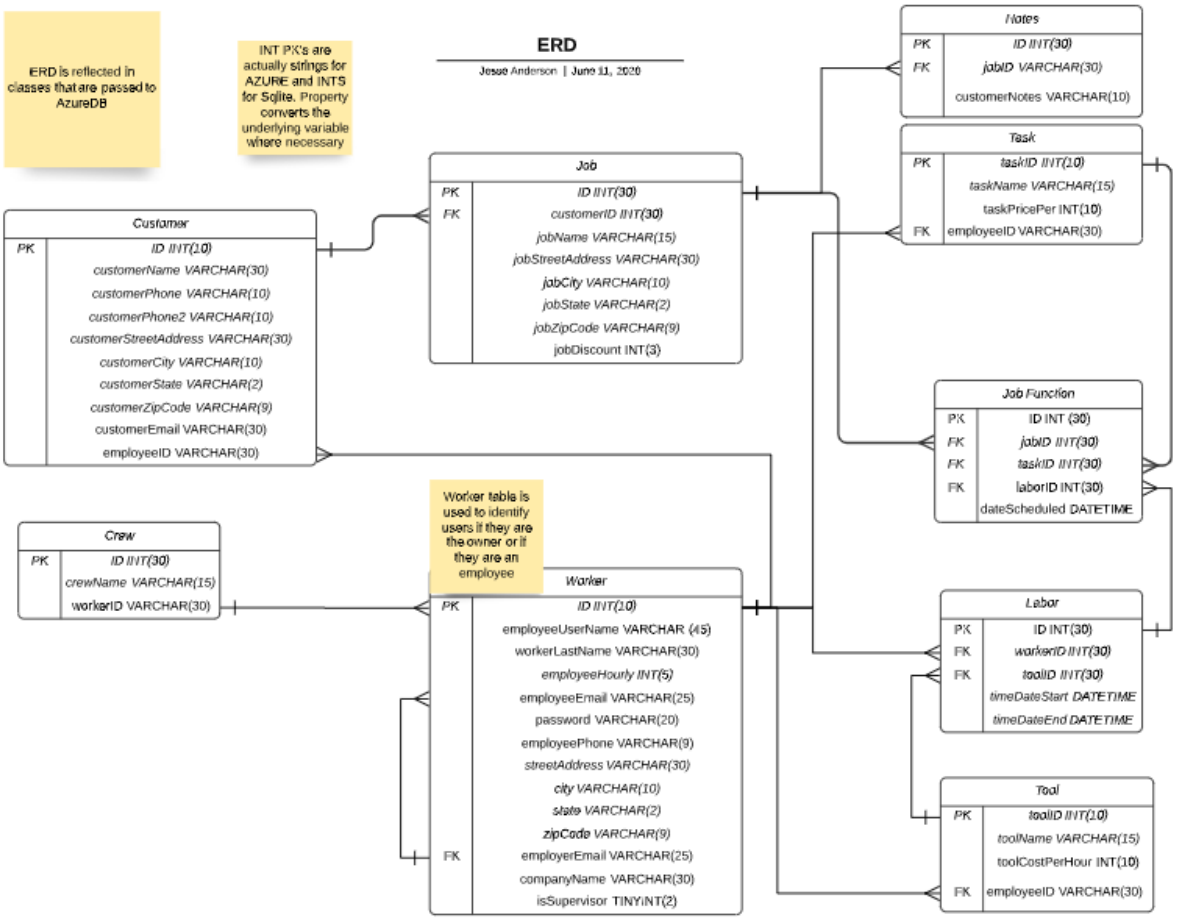


Figure 4 ERD (Entity Relationship Diagram)

The Employee table contains the login information for the users. It was renamed to the generic 'worker' since it includes the owner and employees. Using the worker table all tables are accessed based upon the login of the owner of the company. All employees are referenced by the owner's email. The

owner's ID is used to access the database tables where objects are created and inserted but not yet linked to their corresponding connections that would make an outside query possible.

For instance, a tool or task may be created but not yet used at any job. The owner has a task and has input how much the task costs but hasn't been required to do the task at any job yet. Normalization would require we access the task table through the connecting tables in-between since the relationship between the employer and the task seems to be an indirect association in terms of the reasons for the creation of the database. The purpose of the database is to define which task is used at the job. So, the relationship normally would pass from the worker, through their labor, to the specific task. But the argument is that there is a direct association between the owner and what services he can provide. It's the owner's skill set, and while not in a direct relationship to the job and customer, yet, it is a skill he can perform that others in his field may not. This also holds, that he owns a tool, if he chooses to use it or not. This still adheres to the first three rules of normalization.

A.1.3 UI Design

The completed design of the software held to the initial wireframing in the registration and splash pages with only minor changes. The following GUI pages show the high-fidelity wireframes made during the planning phase. The users are then broken out to different areas of responsibility upon login. Most of the responsibility of running the business remains with the owner, while the supervisors and workers track the times they work and work on jobs. The supervisors heightened responsibility allows them to view additional information. Messaging and calendars allow users to communicate upcoming important dates and customer needs. Notes for customers are also included as a reference for that job when it's next visited.

Each page was designed with a singular feature focus. The registration page was to display the registration function. The splash page displays the current job of the day. The jobs page changed some to only display the scheduled jobs, and hours worked moved to the reports page. These slight changes continued throughout development, to simplify each page, while not adding too much depth of views for the user to have to navigate to reach their intended information.

Though each page was wireframed to allow as much simplicity in development as possible, this goal didn't always hold. As development began and changes were made to accommodate the required features, complexity started developing on each page. As a single-person created project, the methodology occasionally devolved from agile into extreme programming. Overlays were used to allow objects to be modified or defined as additional user needs on features were discovered. The lack of pauses, and planning/review phases that agile promotes with its iterative cycles, allowed for some scope drift, not in features, but the simplicity of development. This isn't reflected in the UI design itself, as the user doesn't see the complexity in the code, but is apparent upon code review.

Table A.1.3 GUI Pages

Figure 5 Planning Xamarin UI Login

Figure 6 Planning Xamarin UI Owner Pages 1

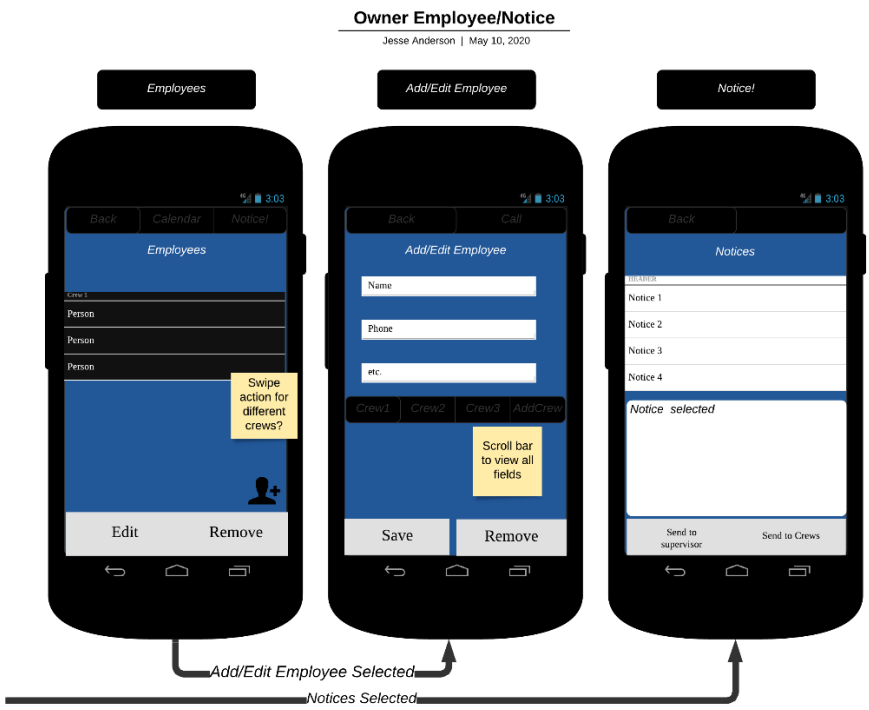


Figure 7 Planning Xamarin UI Owner Pages 2

Figure 8 Planning Xamarin UI Supervisor Pages 1

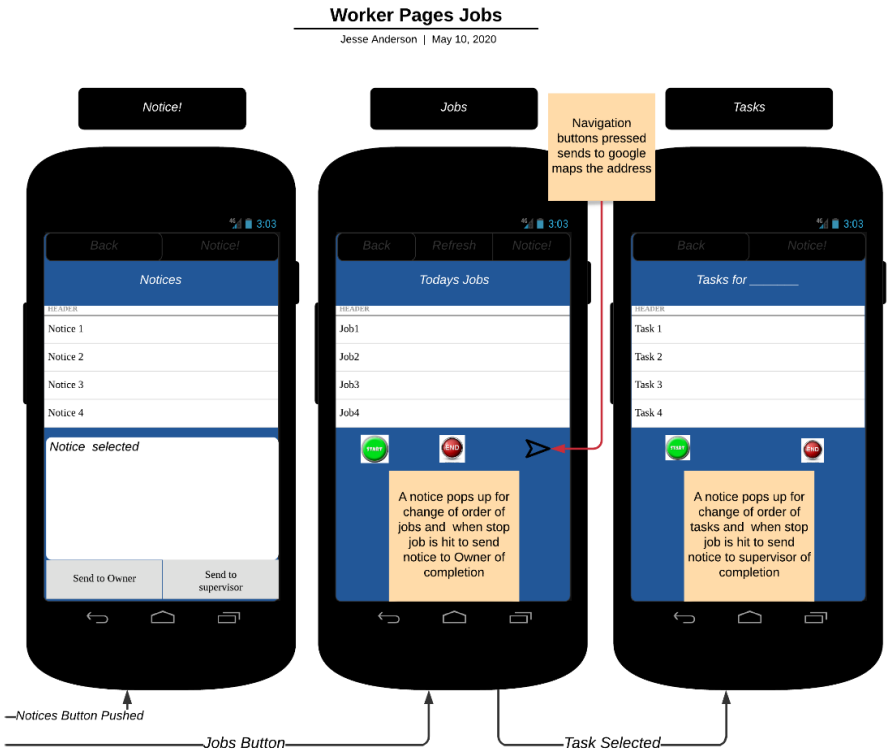


Figure 9 Planning Xamarin UI Worker Pages 1

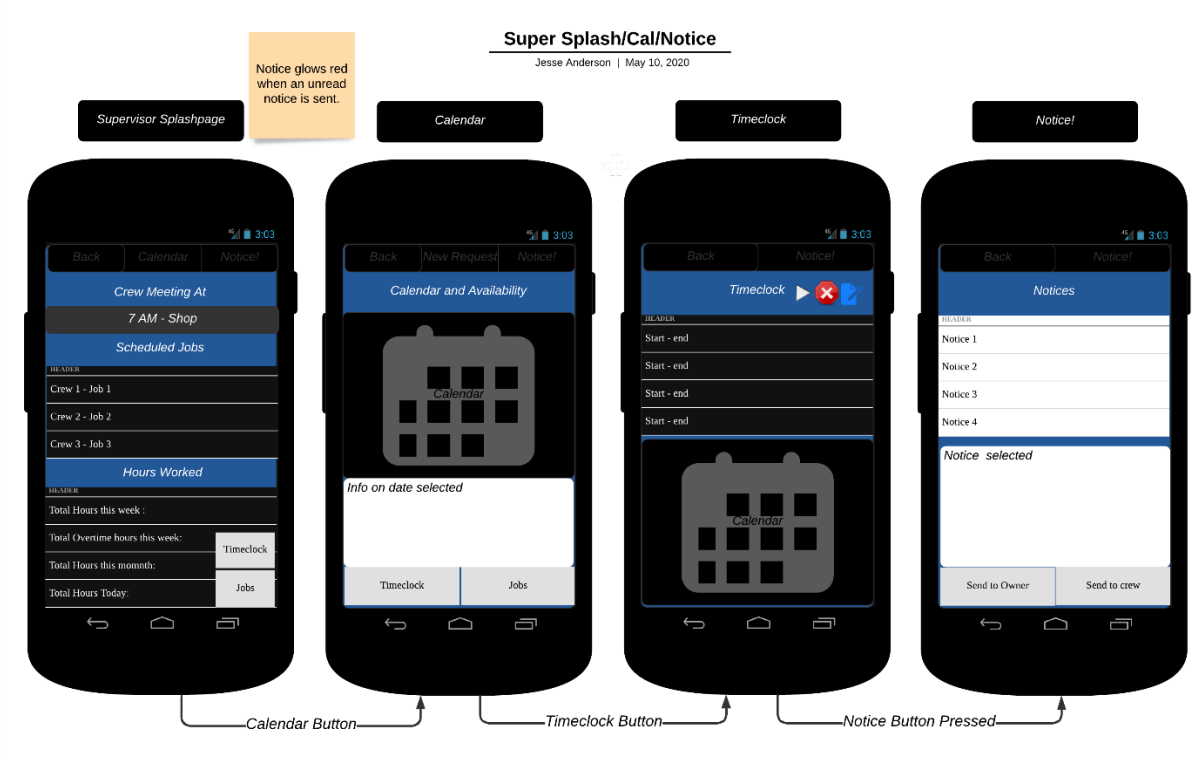


Figure 10 Planning Xamarin UI Supervisor Pages 2

Figure 11 Planning Xamarin UI Worker Pages 2

B. Testing – Unit Design Test Plan

B.1 Introduction

B.1.1 Purpose

As this application is comprised of many functional parts, the system testing of each component must be completed to demonstrate functionality before delivering it to the client. If the testing fails, the product has demonstrated it is not ready for delivery until the failures are resolved. As each function included in the application is necessary for the workflow, each aspect must be tested. Each test will automatically test the Azure SQL database and the API created for this application for its ability to insert, query, modify, and delete content. Specifically, each test will be designed to test the ability of the application to display and modify the aspect of the program. This could include, testing the ability to register a new employee, or create a new job, or a new record of hours for a job, etc. The scope of testing for this application is beyond the scope of this document, so only 2 tests are included that inclusively test the other aspects of the software.

B.1.2 Overview

The development of the application in Xamarin Forms requires the testing to include both the UWP (Microsoft Platforms) and Android platforms. The testing for Android will be done on Nougat 7.1 or better version and as a Windows Application for Windows 10, version 1903 (10,0; Build 18362). Each Unit Test will be performed on both devices and be a test of the platform tested. If the creation and modification of the database information are displayed correctly the test will be considered fulfilled. This testing is conducted manually, by the developer. Later, the developer is releasing the software on a limited scale to a team to perform guerilla systems testing, which is not included here. The manual testing of the application for alignment to the scope of the project is as follows.

B.2 Test 6Plan

B.2.1 Timeclock for a Task at a Job

Functions/Requirement Being Tested:
Use of the timeclock function in a specific job to clock in and out of the associated tasks and complete the job. This includes sending the job's timeclock to a supervisor.

Needs/Preconditions For The Test:
For the test to be conducted, the installation instructions must be followed to install the software on both testing devices. The software must be used to create Customers, Jobs, Employees, the job assigned to a test employee, and the jobs scheduled to be performed. Employees must be logged in to use the time clock function for the task.

Tasks/Steps Involved:

1. The User will log into the Employee account and select the job to be tested.
2. The job to test will be selected and the first task will be clocked into.
3. The task will be clocked out of after any duration the user wishes.
4. The next task follows the same steps as 2 and 3 until all tasks for the job are clocked out.
5. The user will then send the task information with the send button to the supervisors to inform them of task completion.

Deliverables/Expected Results:
The Owner account can navigate to the notice page and will see under the supervisor chat that the job information has been sent.

Pass Criteria
The test will be considered a success if the correct information as reported by the employee is listed in the messaging.
Fail Criteria
If the information passed to the messaging does not reflect what the employee sent or the information is not present at all, the test will be considered a failure.

B.2.1 Sending User Timeclock to Owner/Supervisor

Functions/Requirement Being Tested:
Use of the time clock function for an employee to clock in and out. This includes sending the timeclock to a supervisor.

Needs/Preconditions For The Test:
For the test to be conducted, the installation instructions must be followed to install the software on both testing devices. The software must be used to create Employees. Employees must be logged in to use the time clock function.

Tasks/Steps Involved:
1. The Employee will log in and navigate to the timeclock page.
2. The employee will use the clock in button.
3. The employee will wait an unspecified amount of time and use the clock out function.
4. The employee will click the send function to send their hours to a supervisor.

Deliverables/Expected Results:
The Owner account can navigate to the notice page and will see under the supervisor chat that the employee information has been sent.

Pass Criteria
The test will be considered a success if the correct information as reported by the employee is listed in the messaging.
Fail Criteria
If the information passed to the messaging does not reflect what the employee sent or the information is not present at all, the test will be considered a failure.

B.3 Specifications

The images included show the code that will be tested.

Test 1: Timeclock for a Task at a Job

```
0 references
private async void clockInButton_Clicked(object sender, EventArgs e)
{
    if (LaborClockListView.SelectedItem != null)
    {
        var selectedLaborView = LaborClockListView.SelectedItem as LaborView;
        if (selectedLaborView.TimeDateStart == null)
        {
            var selectedLabor = App.Labors.Where(u => u.ID == selectedLaborView.ID).FirstOrDefault();
            selectedLabor.TimeDateStart = DateTime.Now;

            try
            {
                await App.mobileService.GetTable<Labor>().UpdateAsync(selectedLabor);

                await FormModel.LaborAndTasksAzureDatabaseQuery();
                await ListViewPopulate();
            }
            catch (Exception ex)
            {
                await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "Ok.");
            }
        }
    }
    else
    {
        await Application.Current.MainPage.DisplayAlert("Alert", "Select a task to begin", "Ok.");
    }
}
```

```
0 references
private async void clockOutButton_Clicked(object sender, EventArgs e)
{
    if (LaborClockListView.SelectedItem != null)
    {
        var selectedLabor = LaborClockListView.SelectedItem as LaborView;
        var matchedLaborToUpdate = App.Labors.Where(u => u.ID == selectedLabor.ID).FirstOrDefault();

        matchedLaborToUpdate.TimeDateEnd = DateTime.Now;

        try
        {
            await App.mobileService.GetTable<Labor>().UpdateAsync(matchedLaborToUpdate);

            await ListViewPopulate();
        }
        catch (Exception ex)
        {
            await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "Ok.");
        }
    }
    else
    {
        await Application.Current.MainPage.DisplayAlert("Error", "Select the task to clock out.", "Ok.");
    }
}
```

```
0 references
private async void sendHoursButton_Clicked(object sender, EventArgs e)
{
    bool allRecordsClockedOut = true;

    foreach (LaborView item in laborViews)
    {
        if (item.TimeDateEnd == null && item.TimeDateStart != null)
        {
            allRecordsClockedOut = false;
        }
    }

    if (!allRecordsClockedOut)
    {
        await Application.Current.MainPage.DisplayAlert("Alert", "Please clock out of all active" +
            " tasks first.", "Ok.");
    }
    else
    {
        bool check = await Application.Current.MainPage.DisplayAlert("Query", "Send notice of completed job" +
            " to supervisors?", "Yes.", "Cancel.");
        if (check)
        {
            DateTime todaysDate = DateTime.Today;
            long min = Math.Abs(todaysDate.Ticks - laborViews[0].TimeDateStart.Value.Ticks);
            long max = Math.Abs(todaysDate.Ticks - laborViews[0].TimeDateStart.Value.Ticks);

            LaborView firstJob = laborViews[0];
            LaborView lastJob = laborViews[0];

            foreach (LaborView item in laborViews)
            {
                long diff = Math.Abs(todaysDate.Ticks - item.TimeDateStart.Value.Ticks);

                if (diff < min)
                {
                    min = diff;

                    firstJob = item;
                }

                if (diff > min)
                {
                    max = diff;

                    lastJob = item;
                }
            }

            var jobCompleted = App.Jobs.Where(u => u.ID == passedJobFunctionView.JobID).FirstOrDefault();
            string datesToPass = $"Hours for {jobCompleted.JobName}. Start Time:{firstJob.TimeDateStart.Value}" +
                $" End Time:{lastJob.TimeDateStart.Value}";

            Message message = new Message()
            {
                OwnerID = App.boss.ID,
                SenderID = App.user.ID,
                DateTimeSent = DateTime.Now,
                TextMessage = datesToPass,
                ToSupervisors = true
            };

            try
            {
                await App.mobileService.GetTable<Message>().InsertAsync(message);
                await Navigation.PopAsync();
            }
            catch (Exception ex)
            {
                await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "OK");
            }
```

Figure 12 CS. for tracking hours at a job

Task 2: Sending User Timeclock to Owner/Supervisor

```
0 references
private async void clockInButton_Clicked(object sender, EventArgs e)
{
    TimeClockRecord timeClockRecord = new TimeClockRecord()
    {
        OwnerID = App.boss.ID,
        EmployeeID = App.user.ID,
        ClockedIn = DateTime.Now
    };
    try
    {
        await App.mobileService.GetTable<TimeClockRecord>().InsertAsync(timeClockRecord);
        App.timeClockRecords.Add(timeClockRecord);

        await TimeClockListViewPopulate();

        timeClockInstanceActive = true;
    }
    catch (Exception ex)
    {
        await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "Ok.");
    }
}
```

```
0 references
private async void clockOutButton_Clicked(object sender, EventArgs e)
{
    if (TimeClockListView.SelectedItem != null)
    {
        var selectedRecord = TimeClockListView.SelectedItem as TimeClockRecord;
        var matchedRecordToUpdate = App.timeClockRecords.Where(u => u.ID == selectedRecord.ID).FirstOrDefault();

        matchedRecordToUpdate.ClockedOut = DateTime.Now;

        try
        {
            await App.mobileService.GetTable<TimeClockRecord>().UpdateAsync(matchedRecordToUpdate);

            await TimeClockListViewPopulate();
        }
        catch (Exception ex)
        {
            await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "Ok.");
        }

        timeClockInstanceActive = false;
    }
    else
    {
        await Application.Current.MainPage.DisplayAlert("Error", $"Select the record to clock out.", "Ok.");
    }
}
```

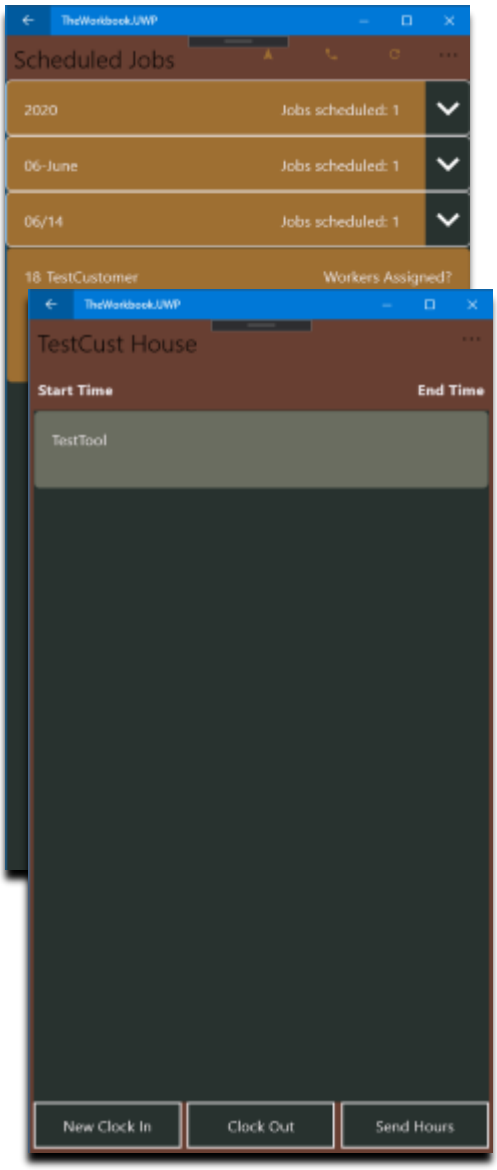
```
0 references
private async void sendHoursButton_Clicked(object sender, EventArgs e)
{
    if (TimeClockListView.SelectedItem != null)
    {
        if ((TimeClockListView.SelectedItem as TimeClockRecord).ClockedOut != null)
        {
            bool check = await Application.Current.MainPage.DisplayAlert("Query",
                "Send this selection to supervisors?", "Yes.", "Cancel.");
            if (check)
            {
                var selectedRecord = TimeClockListView.SelectedItem as TimeClockRecord;
                string startEndDateToPass = $"My hours. Start Time:{selectedRecord.ClockedIn}" +
                    $" End Time:{selectedRecord.ClockedOut}";

                Message message = new Message()
                {
                    OwnerID = App.boss.ID,
                    SenderID = App.user.ID,
                    DateTimeSent = DateTime.Now,
                    TextMessage = startEndDateToPass,
                    ToSupervisors = true
                };

                try
                {
                    await App.mobileService.GetTable<Message>().InsertAsync(message);
                }
                catch (Exception ex)
                {
                    await Application.Current.MainPage.DisplayAlert("Error", $"{ex}", "OK");
                }
            }
        }
    }
}
```

Figure 13 CS. for tracking hours of an employee

B.4 Procedures

Though the tests are similar, they use two different classes, model classes, and tables in the Azure Database. Being the main purpose of the application, to record times and share them via the cloud database. The first test requires the user to have set up at least one employee, at least one customer, and job and scheduled a job with at least one task. The second task only requires the creation of an employee.

The errors will be displayed in the application from try/catch statements, or if the information isn't logged in the application in the notice section of the application. After executing each test, the results can be easily and quickly recorded. Testing of these functions, in general, has occurred throughout development, but system testing of these individual functional units is required evidence of the functionality of the primary scope of the application.



B.4.1 Timeclock for a Task at a Job

From the ‘scheduled jobs’ page, the user will navigate to the job scheduling page by clicking on the ‘Job Times’ button. Once there they will view the tool(s) that was created for the scheduled job. The user can then select a task and click on the ‘New Clock In’ button.

Once the job is clocked in the job will then display a new clock in time. The same occurs with the clock out time on the task selected. Once all tasks are completed the ‘Send Hours’ button will open a confirmation. The approved confirmation will send the hours to the supervisors.

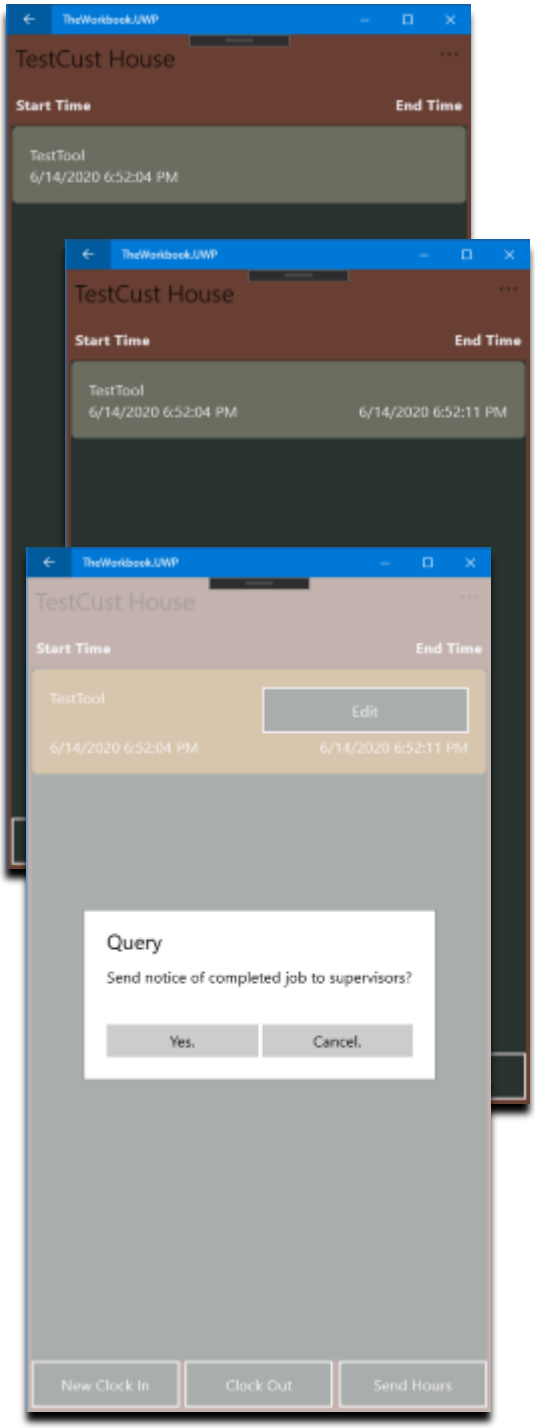


Figure 14 UI during test for task timekeeping

B.4.2 Sending User Timeclock to Owner/Supervisor

From the ‘splash page’, the user will navigate using the ‘Timeclock Button’ to the ‘Timeclock’ page. The user can then select ‘New Clock In’ to create a newly scheduled event. The user can select when to clock out of that event with the ‘Clock Out’ button at whatever is the desired workday interval. Once the user has selected which record they would like to send, and the user has been clocked out on that scheduled date, the user can click the ‘Send Hours’ button to open a query to send their hours.

A new query will appear to confirm sending the hours to the supervisors or owners, and upon confirmation will send the hours.

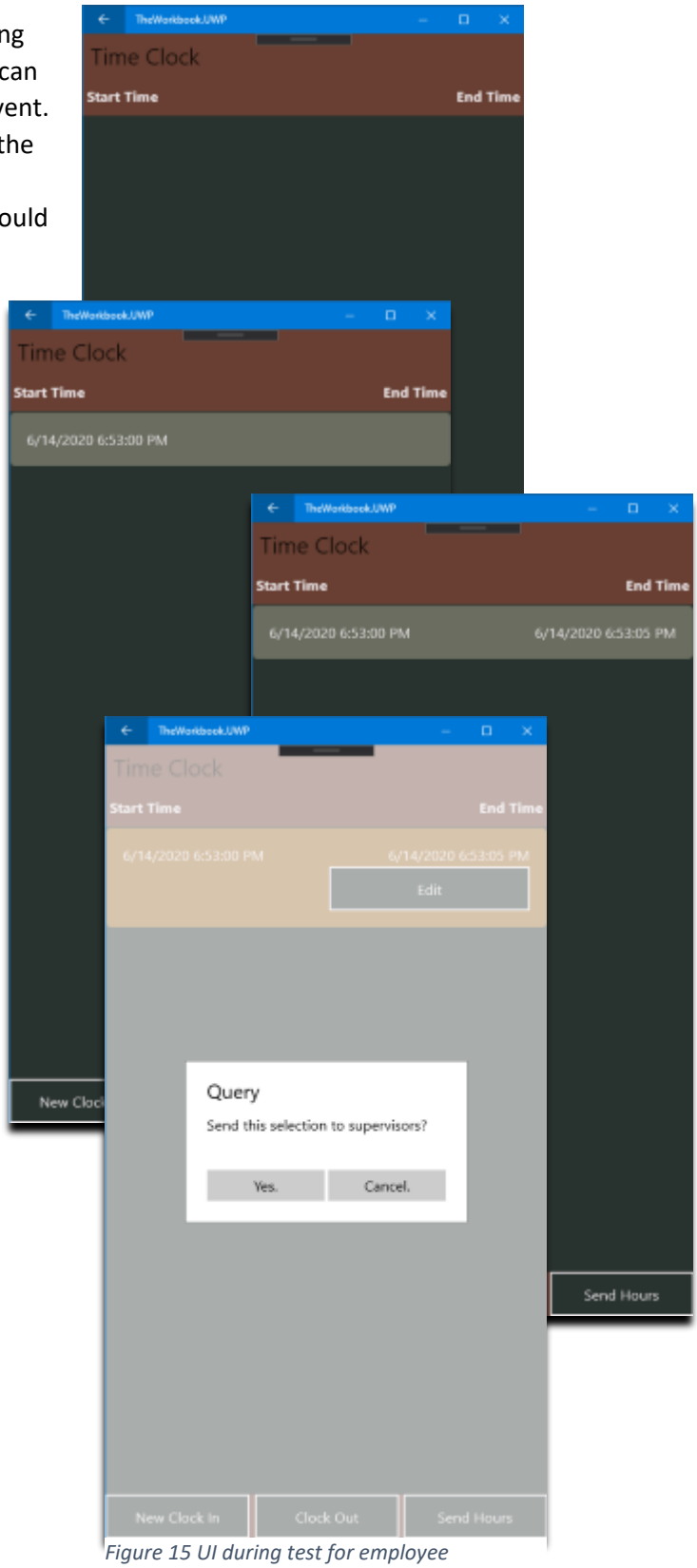


Figure 15 UI during test for employee timekeeping

B.5 Results

B.5.1 Timeclock for a Task at a Job

The results for the first test for recording the hours in a task and sending those hours to the supervisor or owner is recorded here as a success. The hours correspond to the hours recorded in for the job and are displayed under the supervisor's tab on the 'Notice' page.

B.5.2 Timeclock for a Task at a Job

The results for the second test for recording the hours of an employee and sending them to the supervisors or owners are recorded here as a success. The hours correspond to the hours recorded for the employee on the 'Timeclock' page and is displayed here on the 'Notice' page under the supervisor's tab.

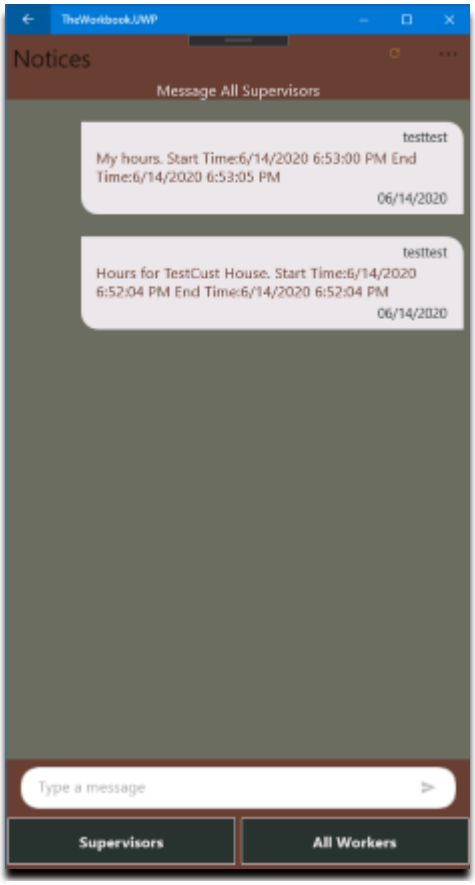


Figure 16 Successful test images

C. Source Code

A compressed version of the source code can be found in the file workschedulersolution.zip

I. Sources and References

Wingen, Malte (Photographer). (2018, November 18). Retrieved from https://unsplash.com/photos/85XLV4Po2mk

Borba, Jonathan (Photographer) (2020, April 28) Retrieved from https://unsplash.com/photos/vLnmmRq6bMY

Maria, Orlova (Photographer) (2019, February 5) Retrieved from https://unsplash.com/photos/LuRFzqHGiA4

Birkett, Adam (Photographer) (2017, August 15) Retrieved from https://unsplash.com/photos/TLomZTHsIqg