Lauren Finley

R11143740

CS 4000

Study Station

Contents

Introduction

Project Overview

Time management and organization are two essential skills that many college students struggle with. In order to achieve better performance in their classes, a large number of students turn to productivity applications to improve their studying efficiency. Although there are already tons of productivity applications currently on the market, I would like to introduce a new application that combines simple yet cutting edge techniques in to a single web-application. Study Station is an application that is designed to help college students plan and organize their study time while eliminating distractions. The techniques that Study Station combines to accomplish this goal are outlined below:

- The Kanban Technique. The Kanban technique is a technique that utilizes what is called a Kanban Board in order to create a visualization of an individual's or team's workflow. Typically, a Kanban Board is a white board that is divided into three categories: To Do, Doing, and Done. Using sticky notes, tasks are placed and moved from section to section

depending on the status of the task. Study Station will create a virtual Kanban Board that aligns with the course schedule and other needs of the student.

- The Pomodoro Technique. The Pomodoro technique is a time management method in which tasks are broken down into short bursts or intervals. Each interval is followed by a short break, and after successfully completing four intervals the individual is rewarded with an extended break. Typically, a Pomodoro interval is 25 minutes in length with the short breaks ranging from 3-5 minutes and the extended breaks ranging from 15-30 minutes after the completion of four consecutive intervals. Study Station will incorporate a timer that mirrors the Pomodoro technique. Additionally, users will have the option to list any distracting websites such as Twitter or Facebook and Study Station will temporarily block access to those sites while an interval is in progress.

Study Station will combine these two techniques by allowing students to assign individual tasks on their Virtual Kanban Board a designated number of Pomodoro intervals, and Study Station will track the number of intervals completed for each task. Study Station will also include charts and graphs so that users can monitor their progress, and a calendar that integrates with either Google or Apple calendars to help prevent users from missing deadlines.

Objectives

To develop a web application that creates the optimal online environment for college students to manage their study habits.

To further my knowledge of web application development- to include the construction of databases, implementing the Model View Controller framework, and integrating client-side rendering with server-side rendering to elevate the performance of Single Page Application techniques.

MVC

The Model View Controller (MVC) 5 framework is a web-development framework that falls under Microsoft's ASP.NET umbrella. The purpose of a web-application framework is to simplify the development process of web applications. Web application frameworks typically help developers to better organize their code while granting them easy access to common libraries that are frequently utilized by web developers.

MVC5 is the latest version of the MVC framework. As indicated by the name, the MVC framework makes web development easier by separating code into three different components: Models, Views, and Controllers.

Models are classes that are dedicated to organizing data that will be used by the Views and the Controllers. In MVC, Models are similar to header files in C++; they essentially create skeletons of objects that can be referenced every time a new instance of an object needs to be created.

Views within the MVC5 framework contain all the code that will generate the User Interfaces that the users will see. In MVC5, it is possible to create views that serve as Single Page

Applications (SPA's) that utilize both server-side and client-side rendering. On the initial page load, html/css and c# code can be utilized to quickly render the page from the server side, and each subsequent action can be rendered on the client-side using Java Script (to include Jquery and JSON files) in order to save time. The Views receive information from the Controllers, and then display that information on the screen using these methods.

Controllers contain the majority of the functionality of an MVC web-application, and determine what happens to the data that passes through. Controllers can use Data Access Objects (DAOs) to retrieve information from a database, and then organize and manipulate that information using Models. After manipulating and determining what happens to various information, that information can then be passed to the View to be displayed in the User Interfaces.

Method

The following steps outline the process in which I executed the Study Station project:

1. *Planning.* Determine the exact functionality of the application, how users can interact with the application, what information needs to be stored, and so forth. During the planning stage, every user interface/ web page needs to be outlined and organized, and how they will interact with the user, each other, and the database needs to be determined.

2. *Execution and Information Sharing.* Determine the best course of action/revision control software to utilize in order for you to monitor and track my progress. Most likely will end up using a Visual Studio shared repository. Also will look into using Lync (Skype for

business) which is available for download on Raiderlink in order to conduct live demos throughout the semester.

3. *Views.* Create the user interface for each page within the web application utilizing CSS and HTML.

4. *Database Integration.* Build and create database tables to store information.

5. *Model.* Create model classes to organize data from the database into objects for ease of use within the Views and the Controller

6. *Controllers.* Using C# or C++, create a controller that contains methods/functions that incorporate and determine what happens when the page is loaded and when users interact with the user interface.

7. *View Interactions.* Use C# or C++ for the initial server-side rendering of the page in the Views to display the saved information. Use Java Script and Java Script components to handle client-side rendering and manipulate data based on user interaction in the Views/user interfaces.

8. *Clean Up.* Go through and clean up code by fixing formatting, adding missing comments, and removing any unnecessary components.

9. *Test.* Test all aspects of the application. Ensure that there are no bugs or security loopholes within the application.

10. *Submit.* Submit the application for grading.

Resources

The following resources are applications that I utilized throughout the process:

GitHub – Even though this was an individual project, I utilized GitHub for source control in order to learn more about code collaboration practices such as, merging code, branching, pushing, pulling, and so forth to better prepare myself for team projects in the future. I also used GitHub as a way to easily make my code viewable for grading purposes.

Visual Studio – I decided to use Visual Studio for this project because it is the Integrated Development Environment that I am the most familiar with, and it goes hand and hand with the MVC5 framework. GitHub also has software that can be integrated with Visual Studio that made committing and sharing code a lot easier.

MySQL – Although unsuccessful, I downloaded and attempted to use MySQL to store and organize data and practice database integration and creating data tables.

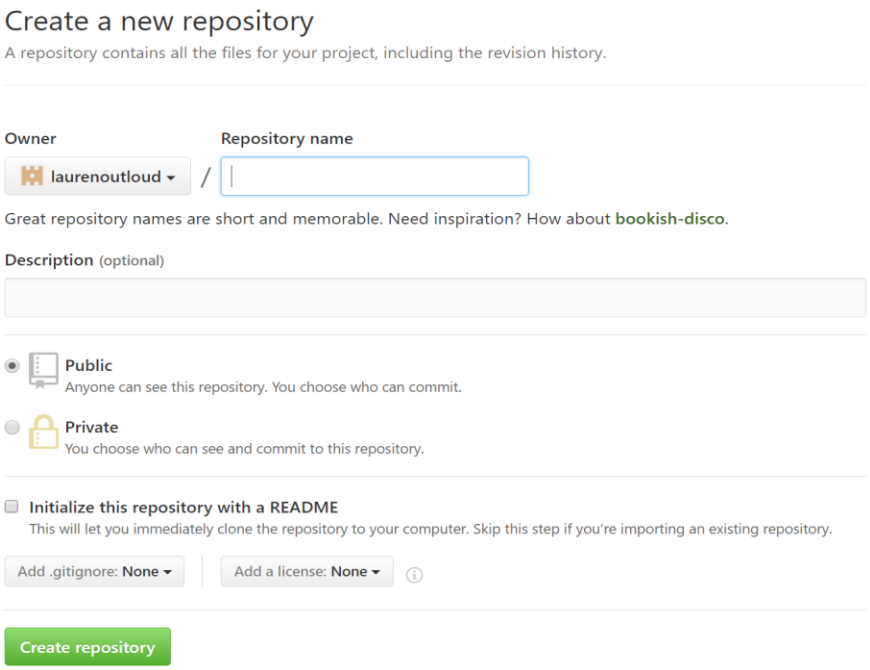
Google Chrome – All of the testing of Study Station was done using Google Chrome with a local host.

Execution

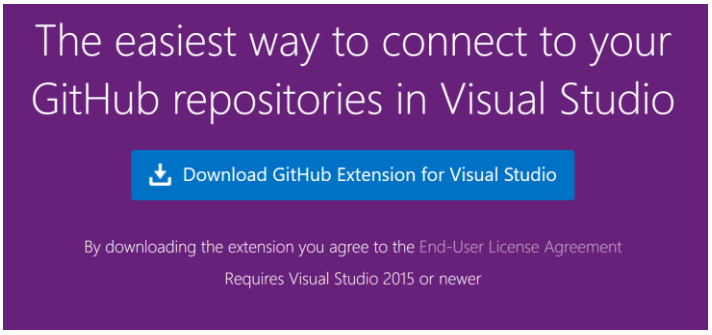
GitHub

https://github.com/laurenoutloud/StudyStation

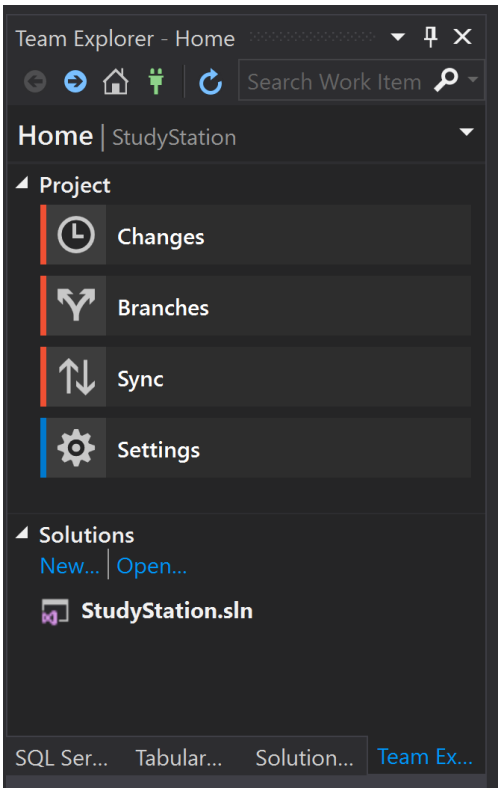
The first thing I did was log on to GitHub and create a Study Station repository to manage my application.

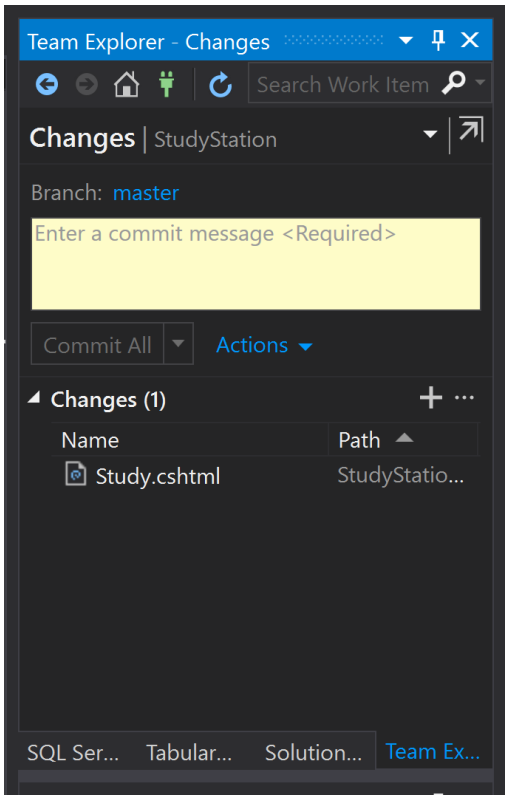


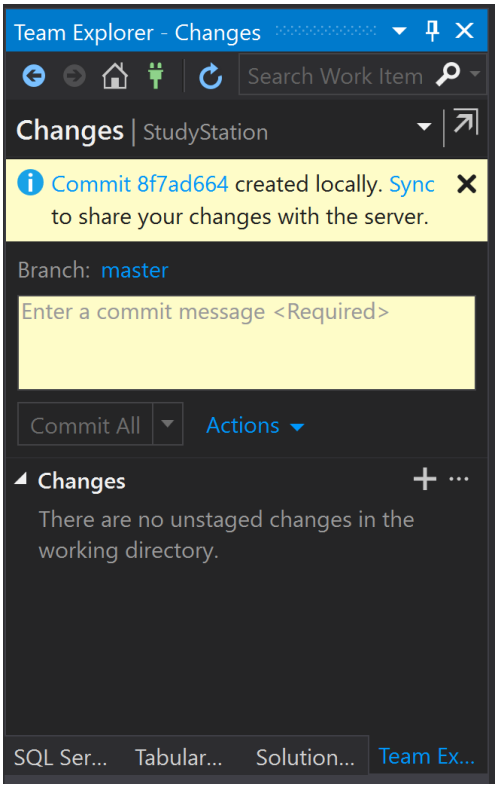
After creating the Study Station repository, I downloaded software from GitHub that allows for integration with Visual Studio's Team Explorer and cloned the empty repository that I had created on GitHub to Visual Studio. I later realized that it would have been easier to download the extension first and then create the repository directly from Visual Studio, but I was able to gain some valuable experience with cloning repositories.

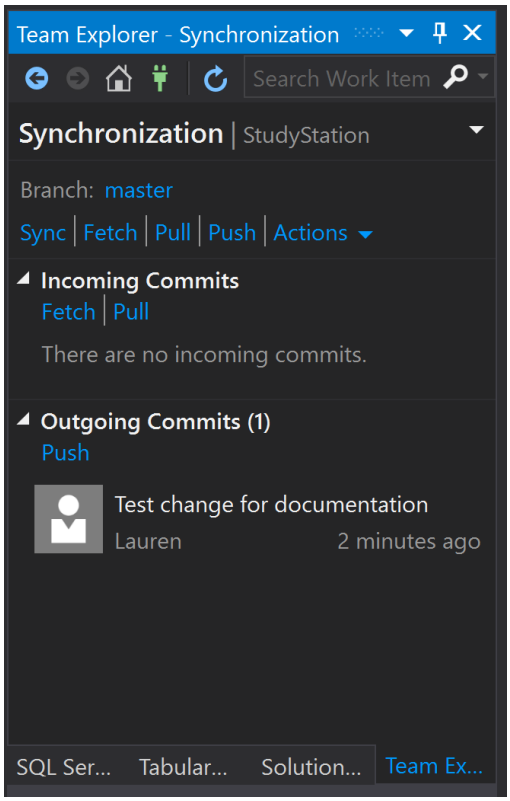


After setting up my Study Station repository in Visual Studio, I created a branch from the master branch (even though it had no code) for me to write my code. Although I did not commit code to GitHub that often, I was able to learn how to commit changes from the branch that I created and push them to the master branch using Visual Studio's Team Explorer.

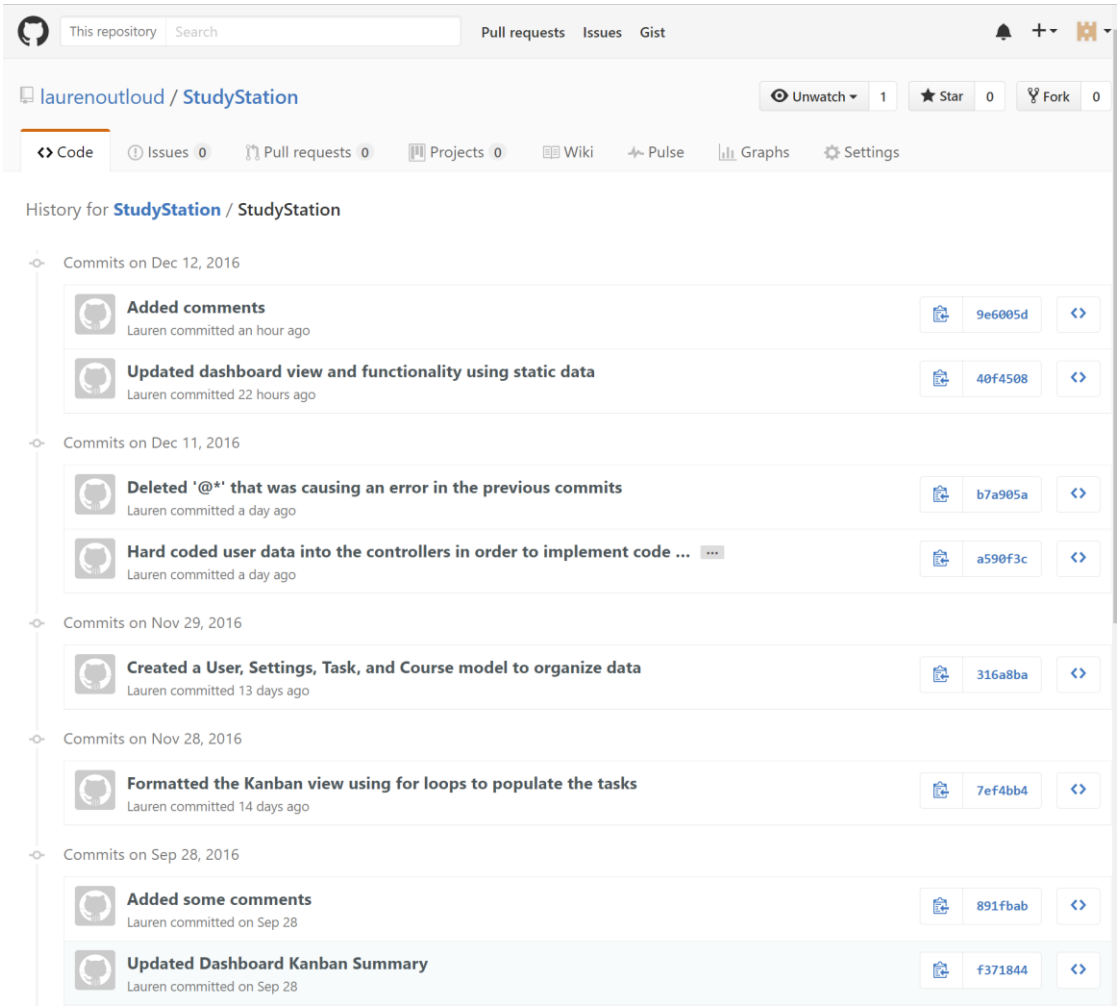

Team Explorer - Home
Search Work Item
Home | StudyStation
Project
Changes
Branches
Sync
Settings
Solutions
New... | Open...
StudyStation.sln
SQL Ser...
Tabular...
Solution...
Team Ex...


Team Explorer - Changes
Search Work Item
Changes | StudyStation
Branch: master
Enter a commit message <Required>
Commit All
Actions
Changes (1)
Name
Path
Study.cshtml
StudyStatio...
SQL Ser...
Tabular...
Solution...
Team Ex...


Team Explorer - Changes
Search Work Item
Changes | StudyStation
Commit 8f7ad664 created locally. Sync to share your changes with the server.
Branch: master
Enter a commit message <Required>
Commit All
Actions
Changes
There are no unstaged changes in the working directory.
SQL Ser...
Tabular...
Solution...
Team Ex...


Team Explorer - Synchronization
Search Work Item
Synchronization | StudyStation
Branch: master
Sync | Fetch | Pull | Push | Actions
Incoming Commits
Fetch | Pull
There are no incoming commits.
Outgoing Commits (1)
Push
Test change for documentation
Lauren
2 minutes ago
SQL Ser...
Tabular...
Solution...
Team Ex...

After pushing changes from Visual Studio, they became visible on GitHub.com under the Study Station repository.

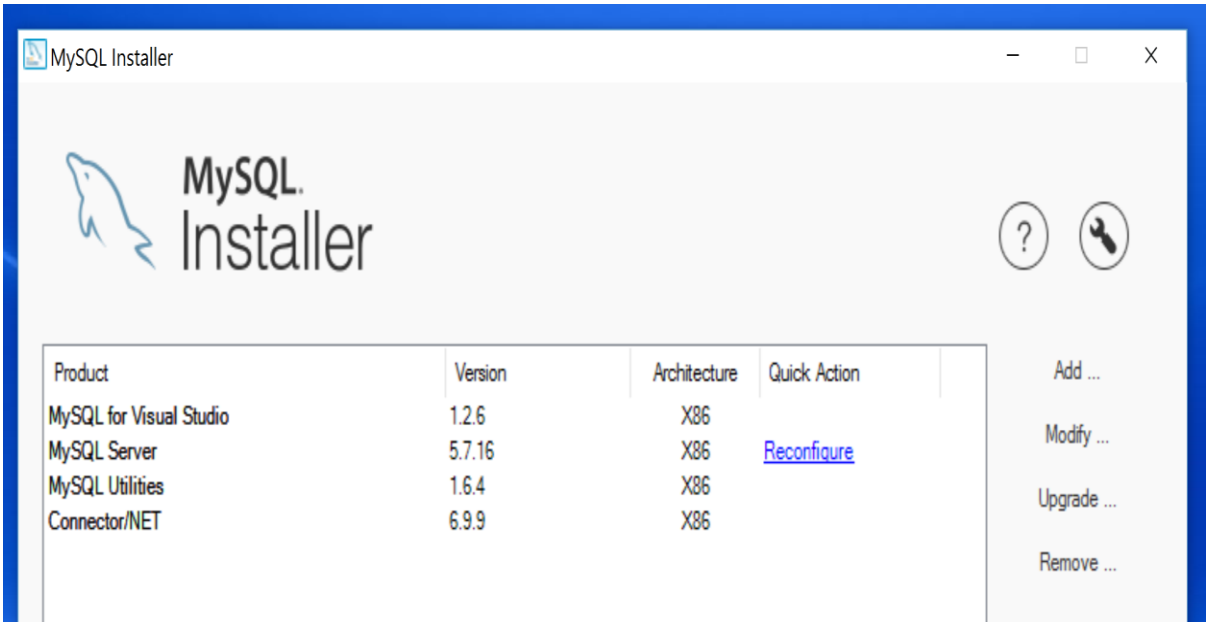


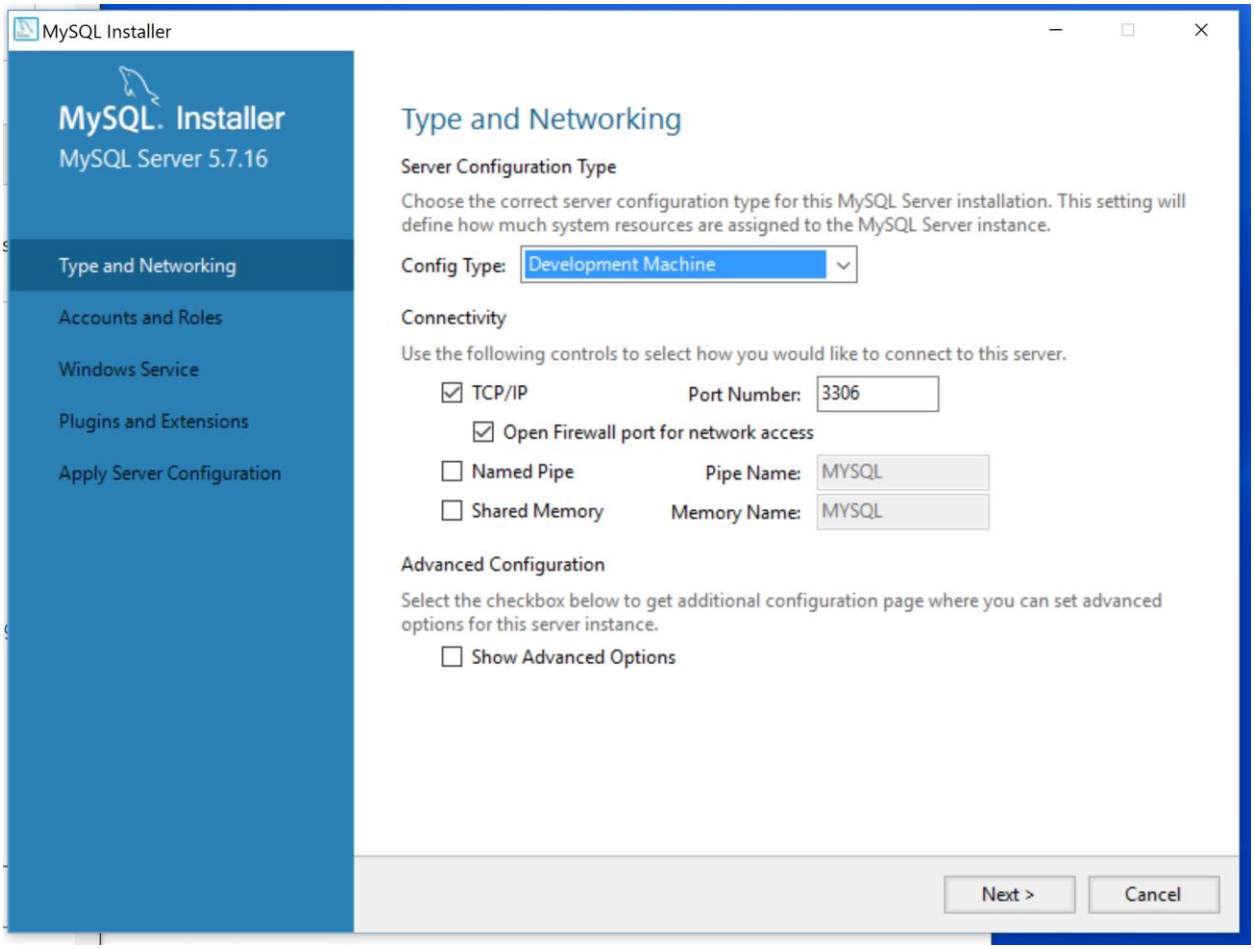
Throughout this semester, by trial and error I was able to learn how to effectively use GitHub as a tool for code management and version control.

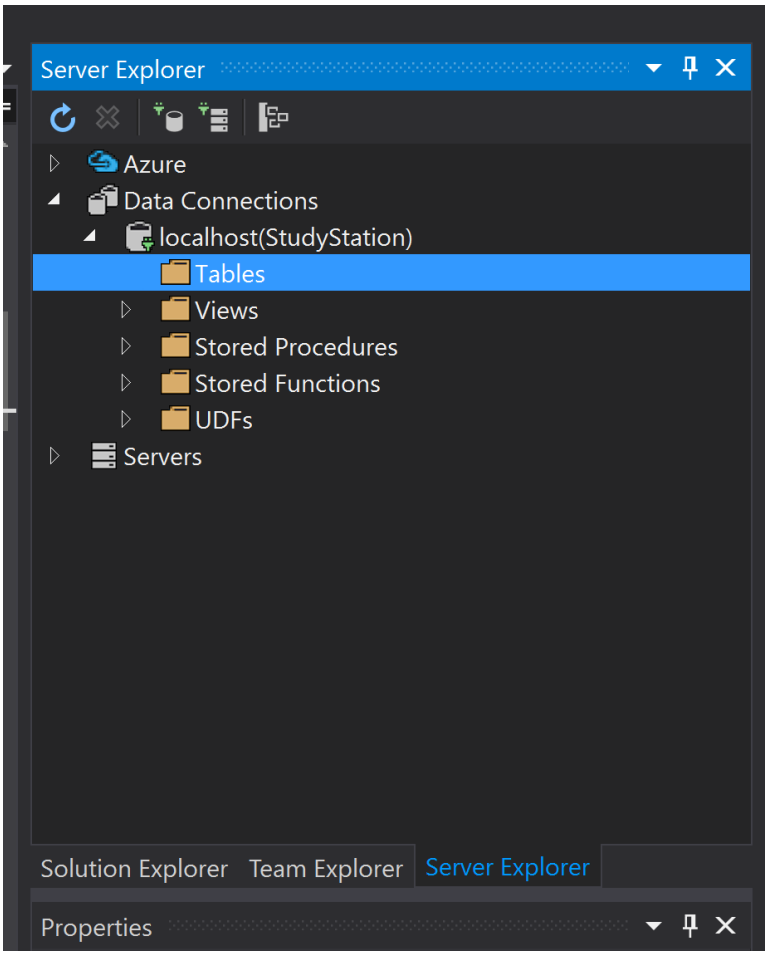
MySQL

I tried numerous times to install MySQL on my computer, but was unsuccessful. I was initially able to create a Study Station database, but when I tried to add any information to it I kept receiving an endless amount of errors. First I downloaded MySQL Installer to assist with

downloading a version of MySQL that was compatible with Visual Studio, then I created a Study Station database using Visual Studio's Server Explorer.



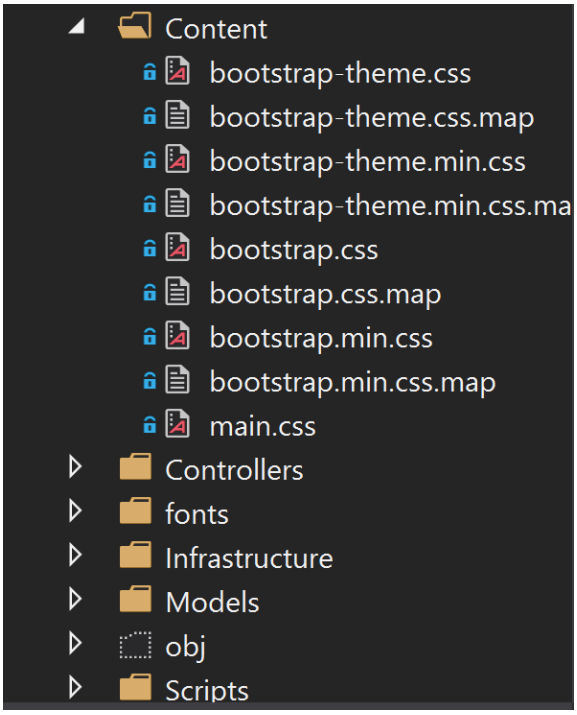

MySQL Installer
MySQL. Installer
MySQL Server 5.7.16
Type and Networking
Accounts and Roles
Windows Service
Plugins and Extensions
Apply Server Configuration
Type and Networking
Server Configuration Type
Choose the correct server configuration type for this MySQL Server installation. This setting will define how much system resources are assigned to the MySQL Server instance.
Config Type:
Development Machine
Connectivity
Use the following controls to select how you would like to connect to this server.
TCP/IP
Port Number:
3306
Open Firewall port for network access
Named Pipe
Pipe Name:
MYSQL
Shared Memory
Memory Name:
MYSQL
Advanced Configuration
Select the checkbox below to get additional configuration page where you can set advanced options for this server instance.
Show Advanced Options
Next >
Cancel

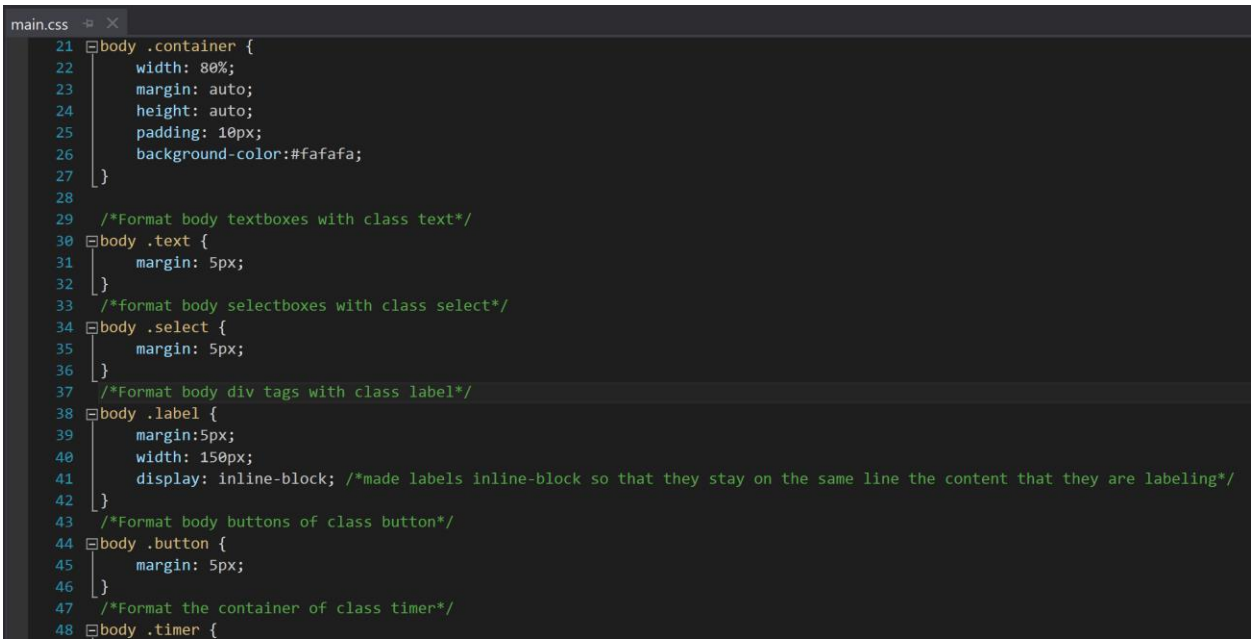


Because I was unable to successfully create a database for my application, I continued with static data that I hard coded into the controllers. As a result, I had to recreate that data in every single controller. Also, any changes made in the view are not saved, and cannot be accessed from other parts of the application.

Style Sheets

When I created my MVC5 project with Visual Studio, it was accompanied by various pre-written bootstrap code that contained different themes and designs. However, I did not utilize any of the bootstrap code in this project, and I created main.css to serve as the style sheet for this application.





```
main.css
21 body .container {
22     width: 80%;
23     margin: auto;
24     height: auto;
25     padding: 10px;
26     background-color:#fafafa;
27 }
28
29  /*Format body textboxes with class text*/
30 body .text {
31     margin: 5px;
32 }
33  /*format body selectboxes with class select*/
34 body .select {
35     margin: 5px;
36 }
37  /*Format body div tags with class label*/
38 body .label {
39     margin:5px;
40     width: 150px;
41     display: inline-block; /*made labels inline-block so that they stay on the same line the content that they are labeling*/
42 }
43  /*Format body buttons of class button*/
44 body .button {
45     margin: 5px;
46 }
47  /*Format the container of class timer*/
48 body .timer {
```

The main.css style sheet contained all of the css code related to the formatting and design of the application. This includes text color, background color, font family, margins, display options, and so forth.

While _Layout.cshtml is a view, I consider it as a type of style/formatting file because it serves as a skeleton for each page within the application. All of the code written on this page is used by every single view to avoid having to rewrite code such as the Navigation bar on every single page.

```
_Layout.cshtml
<!DOCTYPE html>
@*_Layout.cshtml is the skeleton for all of the views within the study station
    application. All of the views contain all of the information on this page*@
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/Content/main.css" /> @*Link the main.css style sheet to this file. main.css contains the majority
        of the code used for styling/formatting components of Study Station*@
</head>

@*Navigation Bar that appears at the top of every single view*@
<ul class="navbar">
    @*<li><a href="\Home\Index">Home</a></li>*@
    <li><a href="\Dashboard\Dashboard">Dashboard</a></li> @*Navigation bar link to dashboard view*@
    <li><a href="\Kanban\Kanban">Kanban</a></li> @*Navigation bar link to kanban view*@
    <li><a href="\Calendar\Calendar">Calendar</a></li> @*Navigation bar link to calendar view*@
    <li><a href="\Study\Study">Study</a></li> @*Navigation bar link to study view*@
    <li><a href="\Settings\Settings">Settings</a></li> @*Navigation bar link to settings view*@
    @*Note: There is no link to the home (index) view from the Navigation Bar because home is only used to sign in or create an account only.
        The user does not need access to the home view once they have logged in*@
</ul>

<body>

    <div class="container">
        @*Create a container for all of the content to go in inside the body so that there are some margins around the outside *@

        <h1>@ViewBag.Title</h1>@*Get the title of the view from the ViewVag and display it at top of page*@

    </div>

    <div class="container-fluid">
        @RenderBody()
    </div>


    <div class="copyright">Study Station &copy; 2016</div> @*Added copyright underneath body container*@
    @RenderSection("scripts", false)
</body>

</html>

82 %
```

I set the default layout to the _Layout.cshtml so that every time the application starts, _Layout.cshtml is the default layout.

```
_ViewStart.cshtml
@*Made _Layout.cshtml the default layout when the application starts*@
@{
    Layout = "Shared/_Layout.cshtml";
}
```

Models

I created three models for the Study Station application: User.cs, Course.cs, and Task.cs.

```
User.cs
StudyStation                    StudyStation.Models.User                    ID
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace StudyStation.Models
{
    public class User //User model serves as a template for storing are information pertaining to eac
    {
        public int ID { get; set; }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Birthday { get; set; }
        public string PhoneNumber { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }

        //information pertaining to a user's desired study settings
        public int IntervalLength { get; set; } //the length of each study interval
        public int ShortBreakLength { get; set; } //the length of each short break
        public int LongBreakLength { get; set; } //the length of each long break
        public int LongBreakAfter { get; set; } //number of intervals to be completed before a long b
        public int DailyIntervalGoal { get; set; } //the goal number of daily intervals the user woul

        public List<Course> Courses { get; set; } //a list of Course objects to allow a user to have
    }

}
```

User.cs is a class that is designed to represent a user. It contains user information such as first name, last name, etc., as well as the various user settings such as interval lengths. The User model also contains a list of Course objects that represent each course the student has created.

```
Course.cs
StudyStation                                StudyStation.Models.Course
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Web;
5
6   namespace StudyStation.Models
7   {
8       public class Course //the course model serves as a template for course data.
9       {
10          public string CourseName; //course name that will be used throughout the
11          public List<Task> Tasks; //a list of task objects so each individual cour
12      }
13  }
```

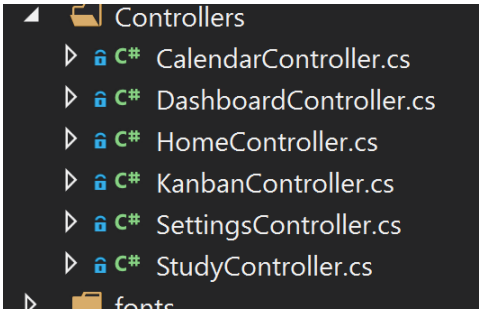
The Course model was very simple. It contains a CourseName variable that will be displayed on the Kanban wall and other various parts of the application as needed, and a list of Task objects for when a user assigns a task to a course.

```
Task.cs
StudyStation                                StudyStation.Models.Task
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Web;
5
6   namespace StudyStation.Models
7   {
8       public class Task //the task model serves as a templat
9       {
10          public int Status { get; set; } //each task will b
11          public string Text { get; set; } //the text allows
12          //public string DateCreated { get; set; } //date c
13          //public string DateCompleted { get; set; } //date
14
15      }
16  }
```

Although the Task model will eventually contain variables such as “number of completed intervals”, “date created”, and “date completed”, I was only able to implement two variables. The first variable is the Status variable, which indicates whether the task is classified as To Do, Doing, or Done (1= To Do, 2 = Doing, 3= Done). The second variable is Text, which allows users to write the details of the task.

Controllers

While I have created six controllers for this application, only three of them contain functionality of substance at this point in time, so I will only elaborate on those three.



Because I was unable to set up a database, the KanbanController.cs, DashboardController.cs, and SettingsController.cs all currently share a similar functionality. All three controllers contain a function that creates a temporary user. When accessed, the controllers call that function to create a new user, adds the user to the Viewbag so that it can be accessed from the Views, and then open their respective Views.

KanbanController.cs | DashboardController.cs | SettingsController.cs

StudyStation | StudyStation.Controllers.KanbanController | Kanban()

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using StudyStation.Models; //Include models to organize generated data

namespace StudyStation.Controllers
{
    public class KanbanController : Controller
    {
        // GET: Kanban
        public ActionResult Kanban()
        {
            CreateUser(); //call CreateUser method to temporarily generate static data
            return View(); //open kanban view
        }

        public void CreateUser() //CreateUser function is a temporary function that creates static data for testing
                                 //TODO: learn how to integrate application with a data base so I no longer need this function
        {
            User currentUser = new User(); //create a User object using the User model
            currentUser.ID = 123456; //assign static fake id for testing
            currentUser.FirstName = "Adam"; //assign static fake first name for testing
            currentUser.LastName = "Clark"; //assign static fake last name for testing
            currentUser.Birthday = "01/01/1990"; //assign static fake birthday for testing
            currentUser.PhoneNumber = "555-555-5555"; //assign static fake phone number for testing
            currentUser.Email = "aclark@internet.com"; //assign static fake email for testing
            currentUser.Password = "password"; //assign static fake password for testing

            currentUser.Courses = new List<Course>() { //create list of Course objects to hold multiple courses
                new Course() { //create a course object using the Course model
                    CourseName ="English", //assign static fake course name for testing
                    Tasks = new List<Task>() //create a list of tasks so that each course can have multiple tasks
                    {
                        new Task() //create a new task object using the Task model
                        {
                            Text = "Write Essay", //assign task text that will be displayed
```

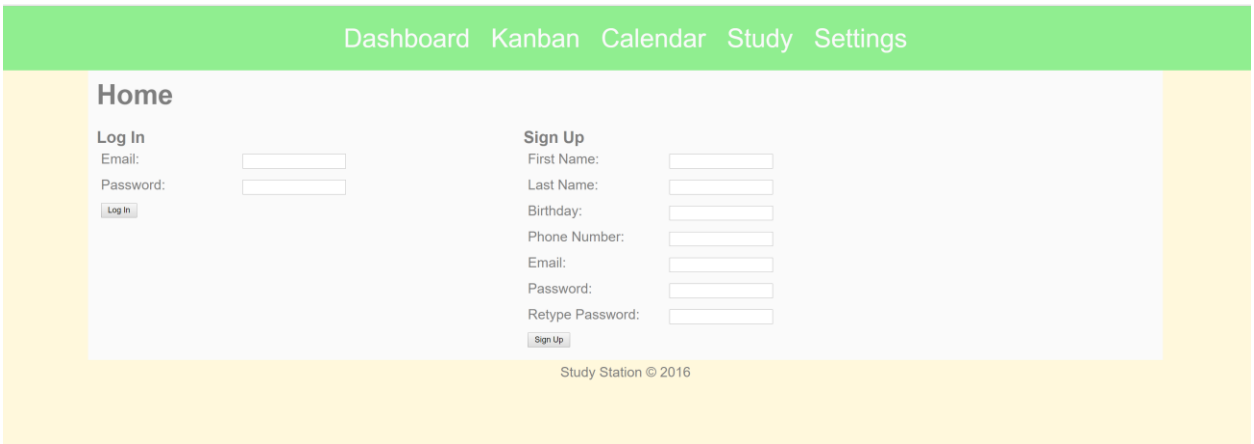
In the CreateUser() method, I use the models to create static data to be tested and manipulated in the views. More details on this method can be found in the comments for the code for the Kanban Controller, Dashboard Controller, or Settings Controller.

Views

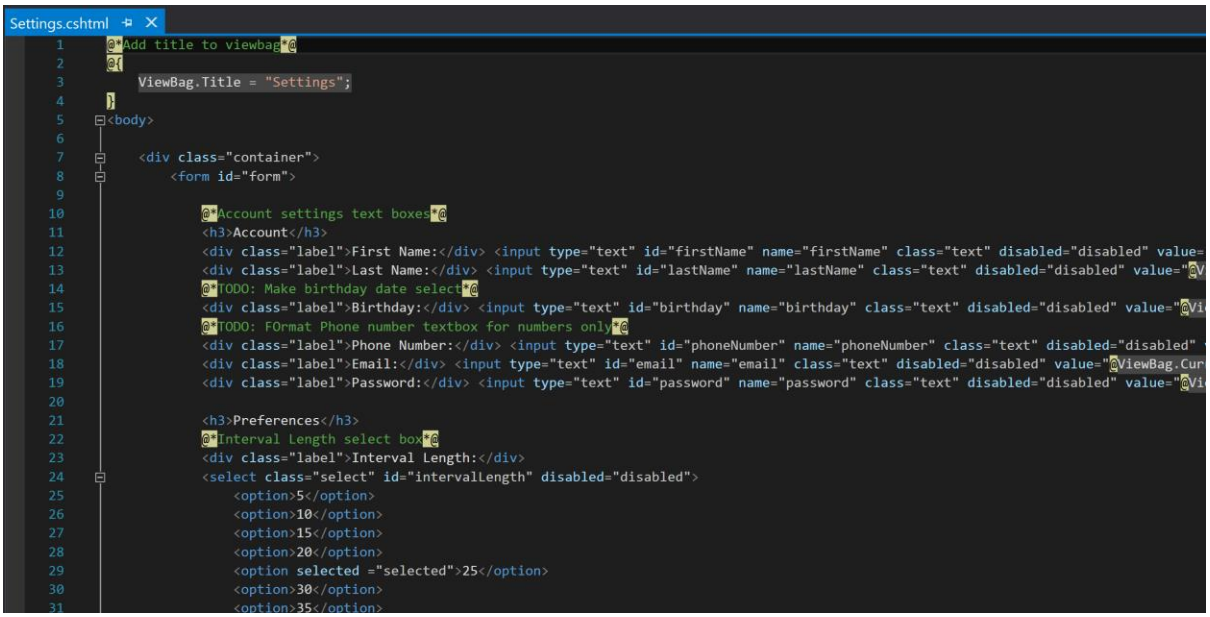
Home

```
Index.cshtml
@*Add title to viewbag*@
@{
    ViewBag.Title = "Home";
}

<body>
    @*TODO: Once application is integrated with database, figure out a way to verify a user's credentials as well a
        user by passing information entered in this view back to the controller*@

    <div class="container">
        @*Form Begins*@
        <form>
            @*Create div tag of class login*@
            <div class="login">
                <h3>Log In</h3>
                <div class="label">Email:</div> <input type="text" name="email" class="text" /><br />
                <div class="label">Password:</div> <input type="text" name="password" class="text" /><br />
                <button class="button">Log In</button>
            </div>
            @*Create div tag of class signup*@
            <div class="signup">
                <h3>Sign Up</h3>
                <div class="label">First Name:</div> <input type="text" name="firstName" class="text" /><br />
                <div class="label">Last Name:</div> <input type="text" name="lastName" class="text" /><br />
                <div class="label">Birthday:</div> <input type="text" name="birthday" class="text" /><br />
                <div class="label">Phone Number:</div> <input type="text" name="phoneNumber" class="text" /><br />
                <div class="label">Email:</div> <input type="text" name="email" class="text" /><br />
                <div class="label">Password:</div> <input type="text" name="lastName" class="text" /><br />
                <div class="label">Retype Password:</div> <input type="text" name="lastName" class="text" /><br />
                <button class="button">Sign Up</button>
            </div>
        </form>
    </div>
</body>
```

In the Home Page View, I created both a login and a sign up section with textboxes for users to input their information. This View (and all other Views) contain a <form></form> tag because once this application is integrated with a database, when a user clicks either the Log In button or the Sign Up button, the View will submit the information to the controller. The Controller will then either add the user to the database or verify that the username and password that were entered are correct.



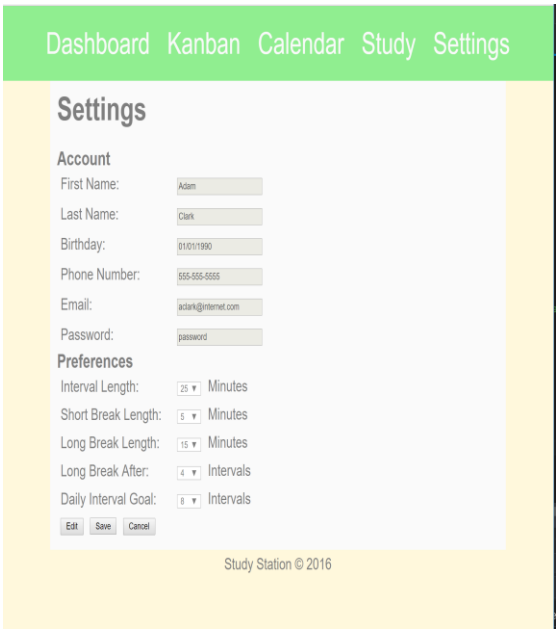
Settings

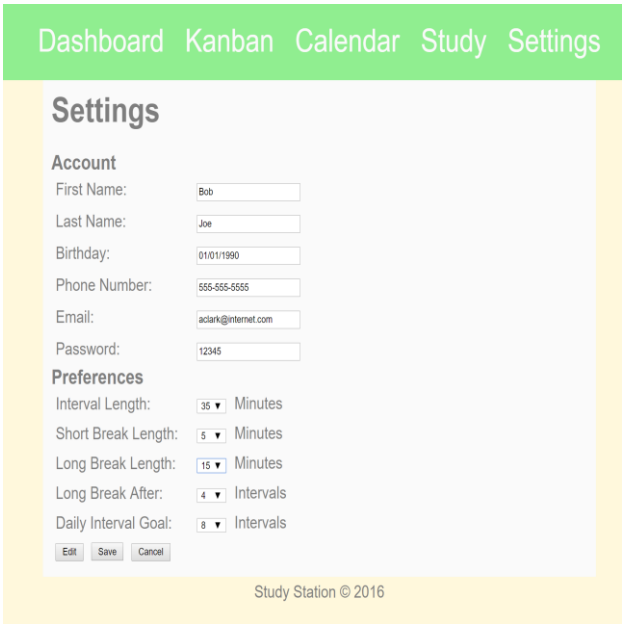


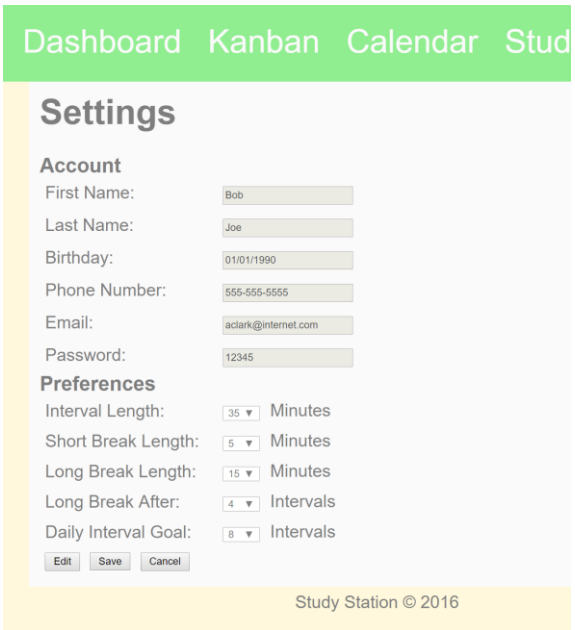
```
Settings.cshtml
@*Add title to viewbag*@
@{
    ViewBag.Title = "Settings";
}
<body>

    <div class="container">
        <form id="form">

            @*Account settings text boxes*@
            <h3>Account</h3>
            <div class="label">First Name:</div> <input type="text" id="firstName" name="firstName" class="text" disabled="disabled" value=
            <div class="label">Last Name:</div> <input type="text" id="lastName" name="lastName" class="text" disabled="disabled" value="@V
            @*TODO: Make birthday date select*@
            <div class="label">Birthday:</div> <input type="text" id="birthday" name="birthday" class="text" disabled="disabled" value="@Vi
            @*TODO: FOrmat Phone number textbox for numbers only*@
            <div class="label">Phone Number:</div> <input type="text" id="phoneNumber" name="phoneNumber" class="text" disabled="disabled"
            <div class="label">Email:</div> <input type="text" id="email" name="email" class="text" disabled="disabled" value="@ViewBag.Cur
            <div class="label">Password:</div> <input type="text" id="password" name="password" class="text" disabled="disabled" value="@Vi

            <h3>Preferences</h3>
            @*Interval Length select box*@
            <div class="label">Interval Length:</div>
            <select class="select" id="intervalLength" disabled="disabled">
                <option>5</option>
                <option>10</option>
                <option>15</option>
                <option>20</option>
                <option selected ="selected">25</option>
                <option>30</option>
                <option>35</option>
```

```
Settings.cshtml
<button class="buton" onclick="save(); return false;">Save </button>
<button class="button" onclick="cancel();">Cancel</button>

<script language="javascript">

    //Convert object in viewbag to Json object
    var userInfo = @Html.Raw(Json.Encode(ViewBag.CurrentUser));

    function edit(){ //function for edit button on click event
        enableSettings(); //when edit button is selected, enabledSettings function is created to allow the user to make changes
    }

    function save() { //function for save button on click event

        //when save button is selected, all of the information that was entered is stored in the JSON object which will
        //then repopulate the settings with the new information
        //TODO: once integrated with database, use Jquery to send data to controller to update database everytime save button is
        userInfo.FirstName = document.getElementById('firstName').value;
        userInfo.Birthday = document.getElementById('birthday').value;
        userInfo.PhoneNumber = document.getElementById('phoneNumber').value;
        userInfo.Email = document.getElementById('email').value;
        userInfo.Password = document.getElementById('password').value;
        userInfo.IntervalLength = document.getElementById('intervalLength').value;
        userInfo.ShortBreakLength = document.getElementById('shortBreakLength').value;
        userInfo.LongBreakLength = document.getElementById('longBreakLength').value;
        userInfo.LongBreakAfter = document.getElementById('longBreakAfter').value;
        userInfo.DailyIntervalGoal = document.getElementById('dailyIntervalGoal').value;

        document.getElementById('firstName').value = userInfo.FirstName;
        document.getElementById('firstName').value = userInfo.FirstName;
        document.getElementById('birthday').value = userInfo.Birthday;
```

The Settings view contains code that creates textboxes for users to view/edit their personal information and select boxes for users to update their settings. This View also contains Java Script that enables all of the settings when the Edit button is selected, saves and disables all the settings when the Save button is selected, and cancels any changes when the cancel button is selected.

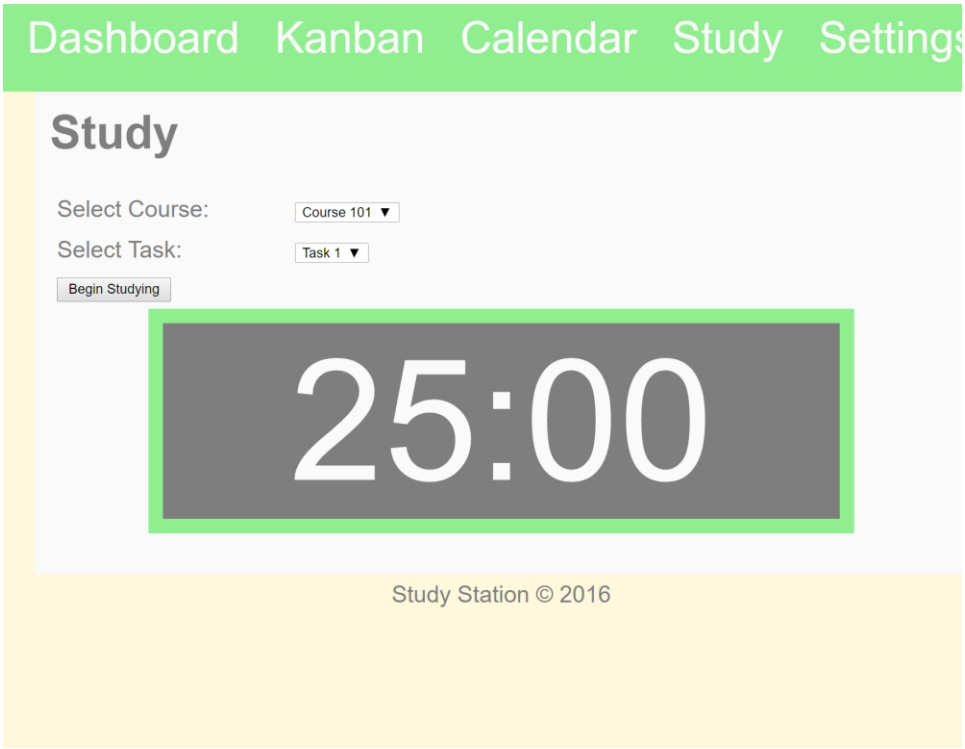






Study

The Study View is where the Pomodoro timer is located. I was unable to successfully use Java Script to create a countdown timer without additional plugins. When the timer is finally implemented, a “Intervals Completed” variable will be added to the task model to keep track of the number of intervals that have been completed for each task. The timer will countdown based on the interval/break settings selected by the user.



Calendar

I also was unable to integrate a calendar. When a calendar is implemented, the “date created” and “date completed” variables will be added to the task model to keep track of when a task was created and when a task was marked as “Done”.

Kanban

The Kanban view creates a table that has been designed to look like a Kanban wall. Using the temporary user that was created in the controller, the Kanban View loops through each task in each course and checks their status. If the status is 1, it adds the task to the To Do column, if the status is 2, it adds the task to the Doing column, and if the status is 3 it adds the task to the Done column.

Kanban.cshtml

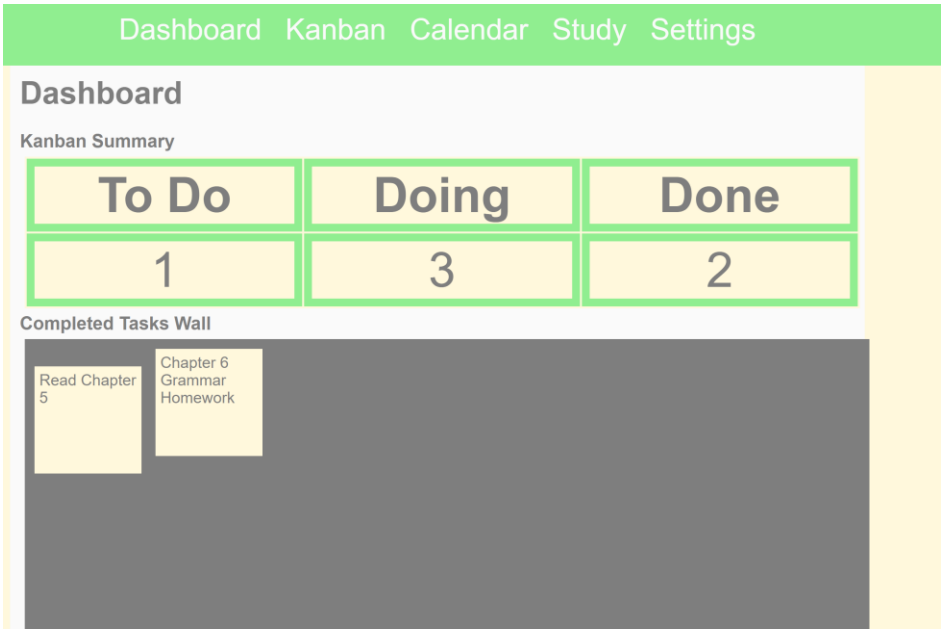
```
<div class="container"> @*container where body content goes*@
    <form>

        <table class="kanban"> @*Create Kanban wall in the form of a table*@
            <tr>
                @*Create table headers*@
                <th width="10%"></th>
                <th width="30%">To Do</th>
                <th width="30%">Doing</th>
                <th width="30%">Done</th>
            </tr>

            @foreach(var course in ViewBag.CurrentUser.Courses) //loop through all the courses created by a user
            {
                <tr>
                    <td>@course.CourseName</td> @*create a new row for each course name*@
                    <td>
                    @foreach (var task in course.Tasks) //loop through all of the tasks assigned to each course
                    {
                        if (task.Status == 1) //if task status is one, put in To Do column
                        {
                            <div class="task">@task.Text</div>
                        }
                    }
                    </td>

                    <td>
                        @foreach (var task in course.Tasks) //loop through all of the tasks assigned to each course
                        {
                            if (task.Status == 2) //if task status is 2, put in Doing column
```

Dashboard

The Dashboard View contains code that loops through all of a user's tasks and keeps track of the number of task in each status category. It uses a table to display the totals. The Dashboard View also loops through all of the tasks and displays all of the tasks that have been completed.

When the calendar and timer have been completed, this dashboard will also contain a date select to view how many tasks were completed during a specific time frame, and to show that information graphically.

```
Dashboard.cshtml
19          <option>This Quarter (3 Months)</option>
20          <option>This Year</option>
21      </select>
22  *@
23
24          <h3>Kanban Summary</h3> @*Header of summary section*@
25              @*Note: The code below used to determine the number of tasks in each category may be better if done i
26              @{
27                  int toDoTasks=0, doingTasks=0, doneTasks = 0; @*Create variables that count the number of each ta
28
29              foreach (var course in ViewBag.CurrentUser.Courses) //use a for loop to loop through all of the cours
30              {
31                  foreach (var task in course.Tasks) //use a for loop to loop through all of the tasks in each cour
32                  {
33                      if(task.Status == 1) //if the task status is 1, incremement the To Do Tasks counter
34                      {
35                          toDoTasks++;
36                      }
37                      if(task.Status == 2) //if the task status is 2, increment the Doing Tasks counter
38                      {
39                          doingTasks++;
40                      }
41                      if(task.Status == 3) //if the task status is 3, increment the Done Tasks counter
42                      {
43                          doneTasks++;
44
45                      }
46                  }
47              }
48              }
49      @*Table to provide a summary of all tasks that are currently on the virtual kanban wall*@
```



Conclusion

This semester I learned how important it is to do research and planning before starting a project. Because I jumped straight into coding, I ran into a lot of problems that were essential to my success with this application. The three main problems I came across were not being able to use MySQL, not being able to integrate a calendar, and not being able to figure out how to write code for a countdown timer.

While I had a lot of setbacks this semester, I did learn a lot of valuable skills as well. The Study Station project was the first project that I have implemented with GitHub, and I am now more comfortable with version control. As a result, I am better prepared for future classes in which we must collaborate with different students together on a project. I also learned a lot more about Visual Studio because this was my first time using Visual Studio as an individual to implement a project instead of just using it to compile and run code.

References

https://github.com/

https://www.visualstudio.com/