School E-marking Web Application

Table of Contents

1. Structure of the course work

1.1. Database Structure

First, we created a Model class **Homework** which defines the Homework entity as illustrated in figure 1. All homework file related data will be stored in **Homework** table. Regarding to user login information, we chose to use ASP.NET Identity. Therefore, all user login information will be stored inside ASP.NET Identity tables whose details will not be covered in this report.

In this coursework, we decided to implement database integrity through constraints, transactions and automated migrations. And we also used Entity Framework to map our model to database. Since ASP.NET already defined a Database Context called **ApplicationDbContext**. In order to make things simple, we decided to use ApplicationDbContext to provide access to **Homework** table as well.

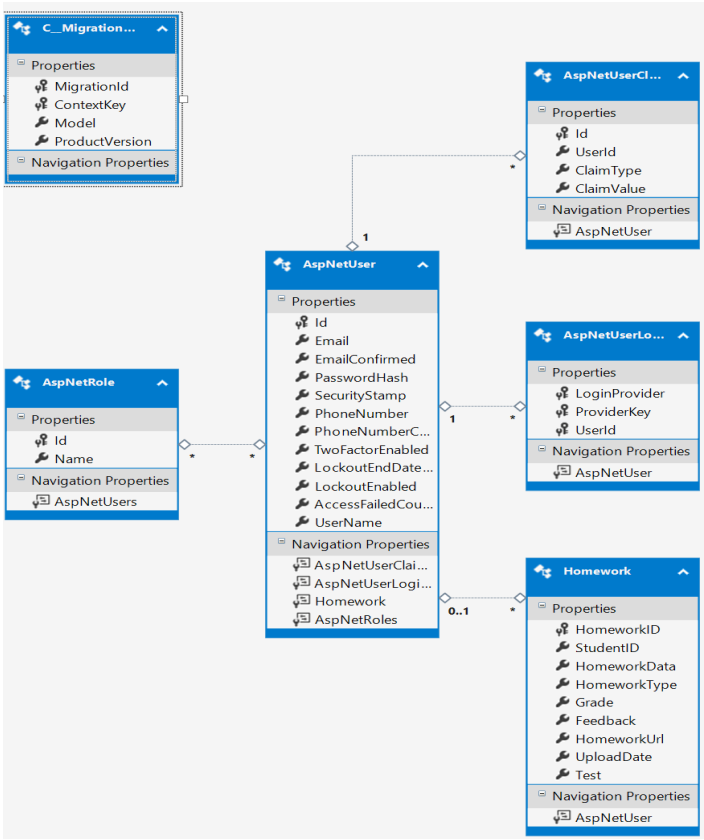


Figure 1 Database Diagram

1.2. Code Structure

In this grading system, there are two roles, **student** and **teacher**. So we decided to create two controllers for student and teacher respectively (1).

As illustrated in figure 2, Inside teacher controller, there are seven methods. Three of them including **CurrentUserID**, **CurrentUserRole** and **GetStudentID** are created to provide user related information for other methods. When teacher successfully login, he will be directed to **Index** method. When teacher tries to mark a student's homework, HttpGet method **Mark** will handle the request. Right after teacher submits the feedback and grade which are valid, HttpPost method **Mark** receives the posted data and write to database and then redirect to **MarkSuccess** method.

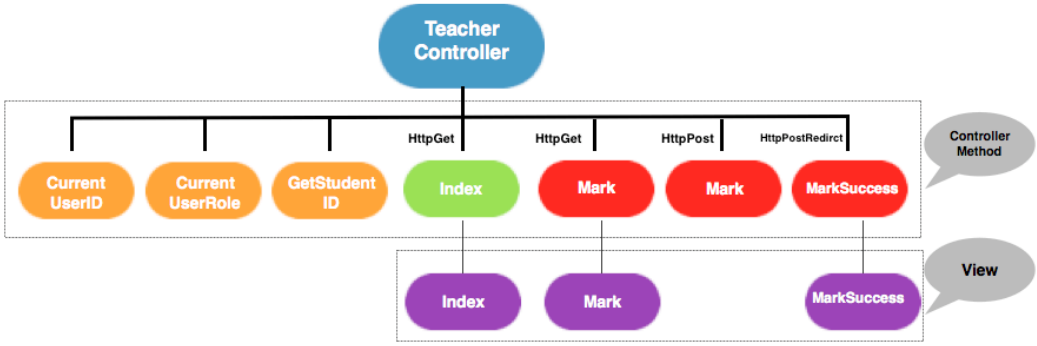


Figure 2 Teacher controller and relative views

Regrading to student controller, there are in total four methods showing in figure 3, namely **CurrentUserID**, **CurrentUserName**, HttpGet **Index** and HttpPost **Index**. Similar as teacher controller, **CurrentUserID** and **CurrentUserName** provides user related data to these two Index methods. The HttpGet **Index** reads student homework file data if he has already uploaded one and then pass it to View. Once a student uploads a file, HttpPost **Index** method receives the file and write to database.

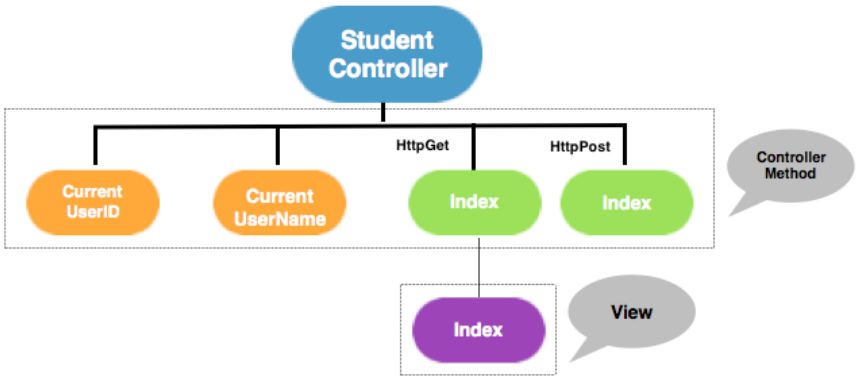


Figure 3 Student controller and relative views

For Teacher, there are three views per figure 2. **Index** View shows a table of all students while **Mark** View allows teacher to mark students' homework and give feedback. **MarkSuccess** View displays the grade and feedback afterwards.

For Student, Index View allows students to upload their HTML files and view the feedback and grade.

Figure 4 explains the authentication and authorization logic of this web application which is handled by **Asp.Net Identity**. When a user opens login portal, username and password will be checked to validate the legitimate of the user. A designated page will then be displayed based on the user's role.

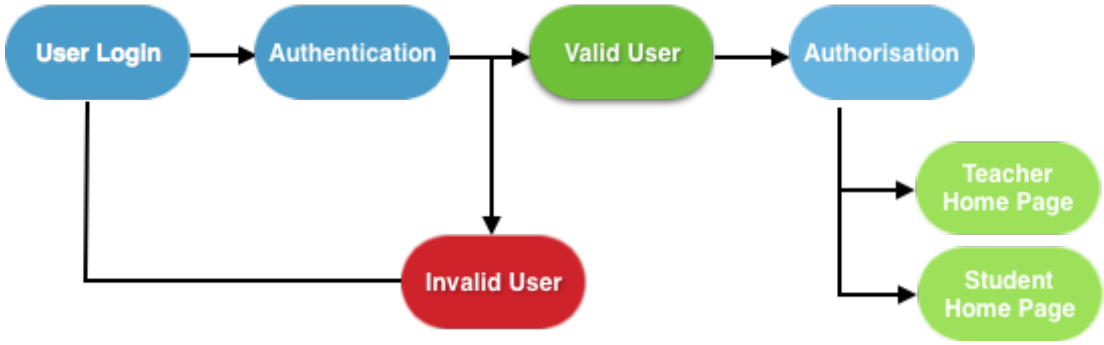


Figure 4 Authentication and Authorization

2. Implemented features

2.1. Required Technologies and Techniques

We used ASP.NET identity for authentication according to the requirement. In each role's controller, we just added a single line of code (screenshot 1).

Student Part

Students are allowed to upload a HTML file and view their grade and feedback. In **StudentControoller**, there is only one **Index** method to accomplish these two functions. In HttpGet **Index** method, the program first tries to find the homework data that matches the current student ID and use ViewBag.flag parameter to tell the view whether a student has homework data (screenshot 2).

HttpPost **Index** method receives the data student uploaded and writes it to database if it is valid.

```
namespace GradingSystem.Controllers
{
    [Authorize(Roles = "Teacher")]
    public class TeacherController : Controller
    {
```

Screenshot 1

```
public ActionResult Index()
{
    ViewBag.UserName = CurrentUserName();
    string sID = CurrentUserID();
    Homework homeworks = dbcontext.HomeworkDB.FindbyStudentID(sID);
    if (homeworks == null)
    {
        ViewBag.flag = null;
    }
    else
    {
        ViewBag.flag = "1";
    }
    return View(homeworks);
}
```

Screenshot 2

Meanwhile, ViewBad.flag will be set to true. The Index view will be re-displayed.

In Student **Index** View, by using ViewBag.flag, we can decide what content the page should present. If students haven't uploaded the HTML file, the page will show "no homework" and show buttons to let student submit their file. After the file being submitted, feedback and grade fields will be presented. All contents use the **well** class from Bootstrap to meet the fluid requirement. We used **accept** attribute in **input** button to validate the upload file to be html type.

```
[HttpPost]
public ActionResult Index(HttpPostedFileBase SelectedHomework)
{
    //Upload homework file
    ViewBag.UserName = CurrentUserName();
    if (SelectedHomework != null && ModelState.IsValid)
    {
        Homework hm = new Homework {
            StudentID = CurrentUserID(),
            HomeworkData = new byte[SelectedHomework.ContentLength],
            HomeworkType = SelectedHomework.ContentType,
            UploadDate = DateTime.Now
        };
        SelectedHomework.InputStream.Read(hm.HomeworkData, 0, hm.HomeworkData.Length);
        using (var dbContextTransaction = dbcontext.Database.BeginTransaction())
        {
            try
            {
                dbcontext.HomeworkDB.Add(hm);
                dbcontext.SaveChanges();
                dbContextTransaction.Commit();
            }
            catch(Exception)
            {
                dbContextTransaction.Rollback();
            }
        }
    }

    //if uploaded successfully, then read it from database
    Homework homeworks = dbcontext.HomeworkDB.FindbyStudentID(CurrentUserID());
    //var homeworks = context.HomeworkDB.FindbyStudentID(sID);
    if (homeworks == null)
    {
        ViewBag.flag = null;
    }
    else
    {
        ViewBag.flag = "1";
    }
    return View("Index", homeworks);
}
```

Screenshot 3 (HttpPost Index for StudentController)

```
@if (ViewBag.flag == null)
{
   <h3>Homework</h3><div class="well well-sm">Homework not submitted</div>
   <h3>Grade</h3><div class="well well-sm">No Grade</div>
   <h3>Feedback</h3><div class="well well-sm">No Feedback</div>
   <div class="container">
     <form enctype="multipart/form-data" method="post" action="Index">
        <input type="file" class="btn btn-primary" name="SelectedHomework" accept=".html"/
         <br />
        <input type="submit" class="btn btn-primary" value="submit" />
     </form>

   </div>
}
```

Screenshot 4 (the upload part in Index View)

Teacher Part

Teacher should see a list of all students and can mark their homework. We get all students from the database in the **Index** method by using **RoleManager** (2) to filter users according to their roles. All students related information will be passed to the view by **ViewData** in array form.

```
public ActionResult Index()
{

    var roleManager = new RoleManager<IdentityRole>(new RoleStore<Identity
    string TeacherName = CurrentUserName();
    //get all student list
    var allUsers = dbcontext.Users.ToList();
    List<string> Studentlist = new List<string>();
    foreach (var u in allUsers)
    {
        var roleID = u.Roles;
        var role = roleManager.FindById(roleID.ToArray()[0].RoleId);
        if (role.Name == "Student")
        {
            Studentlist.Add(u.UserName);
        }

    }
    ViewData["passedArray"] = Studentlist.ToArray();
    ViewBag.TeacherName = TeacherName;
    return View();

}
```

Screenshot 5 (Index method in TeacherController)

```
using (@Html.BeginForm("Mark", "Teacher"))
{
    <input type="hidden" name="hmid" value="@ViewBag.HomeworkId" />
    <div class="form-group">
        @Html.ValidationMessageFor(Model => Model.Feedback)
        <label for="Feedback">Feedback:</label>
        @Html.EditorFor(Model => Model.Feedback, new { htmlAttributes = new { @class = "form-control" }, })
        @Html.ValidationMessageFor(Model => Model.Grade)
        <label for="Grade">Grade:(A,B,C,D,F)</label>
        @Html.EnumDropDownListFor(Model => Model.Grade, "Select a grade", new { @class = "form-control" })
    </div>

    <h2>Here are student's html file</h2>
    <div class="embed-responsive embed-responsive-16by9">
        <iframe class="embed-responsive-item"
                srcdoc="@System.Text.Encoding.Default.GetString(Model.HomeworkData)" ></iframe>
    </div>
    <br />
    <div class="container">
        <input type="submit" class="btn btn-primary" value="Mark" />
    </div>
}
```

Screenshot 6 (Part code in Mark View file)

Again, we used the **table** class from Bootstrap framework in the **Index** View to decorate the page and make it fluid. There is a link behind every student's name leading the teacher to mark page. The HttpGet **Mark** method will use student name to find their homework data and then pass the homework model to the **Mark** View. At **Mark** View, Html.EditorFor() and Html.EnumDropDownListFor() are used to edit feedback and let teacher select a grade from a list for this homework. Html.ValidationMessageFor() method is used to display error message for incorrect input data. We used **iframe** tag to display the content of students' uploaded HTML file, and in srcdoc attribute we convert the HTML data from byte form to text so iframe can display the HTML file. Bootstrap makes all contents responsive.
In HttpPost **Mark** method, the program gets the model from the view and validate it. The data will be saved to database if it is valid. After this process, the page will be redirected to **MarkSuccess** View which shows the feedback and grade the teacher posted.

```
public ActionResult Mark(int hmid, Homework hm)
{
    Homework current = dbcontext.HomeworkDB.FindbyHomeworkID(hmid);
    if (ModelState.IsValidField("Feedback") && (hm.Feedback.Length < 50 || hm.Feedback.Length > 200))
    {
        ModelState.AddModelError("Feedback", "Please input feedback s between 50 and 200 characters of text");
    }
    if (ModelState.IsValidField("Grade") && hm.Grade == null)
    {
        ModelState.AddModelError("Grade", "Please select correct grade from:A,B,C,D,F");
    }
    if (ModelState.IsValidField("Feedback") && ModelState.IsValidField("Grade"))
    {
        using (var dbContextTransaction = dbcontext.Database.BeginTransaction())
        {
            try
            {
                current.Feedback = hm.Feedback;
                current.Grade = hm.Grade;
                dbcontext.SaveChanges();
                dbContextTransaction.Commit();
            }
            catch (Exception)
            {
                dbContextTransaction.Rollback();
            }
        }
        return RedirectToAction("MarkSuccess", hm);//return redirection
    }
    else
    {
        return View(current);
    }
}
```

Screenshot 7(Save changes in HttpPost Mark method)

2.2. Additional Technologies and Techniques

We choose database integrity through constraints, transactions (3) and automated migrations (4) as our additional technologies and techniques.

Constraints

Constraints are used to add restriction to the data in database. As the database structure showed in figure 1. We specified rules for each element. For instance, the "Feedback" data is **mutilineText** data type and its length is restricted

and we build an **enum** for "Grade" to ensure that this web application store the correct grade. Details are showed at screenshot 8.

Transactions

Transactions symbolize treat operations of database as a unit of work and allow database to recovery from any system failure. It is very easy to achieve transactions by Entity Framework 6. We just need to add simple lines of sample code from the official document. There are two places related to database changing in this project. One occurs at the time when students upload their file and another happens when the teacher marks students' homework. We first declare the **begintransaction** then put those codes that modifies the data into **try** section. **Catch** section will be executed to recover the database when failure occurs.

```
public enum Grade
{
    A,B,C,D,F
}


public class Homework
{
    [Key]
    [HiddenInput(DisplayValue = false)]
    public int HomeworkID { get; set; }

    [Required]
    [HiddenInput(DisplayValue = false)]
    public string StudentID { get; set; }

    [Required]
    public byte[] HomeworkData { get; set; }

    [HiddenInput(DisplayValue = false)]
    public string HomeworkType { get; set; }

    public Grade? Grade { get; set; }

    [DataType(DataType.MultilineText)]
    [StringLength(200, MinimumLength = 50)]
    public string Feedback { get; set; }

    [HiddenInput(DisplayValue = false)]
    public string HomeworkUrl { get; set;}
    [HiddenInput(DisplayValue = false)]
    [DataType(DataType.Date)]
    public DateTime UploadDate { get; set; }
```

Screenshot 8 (The Model Constrains)

```
using (var dbContextTransaction = dbcontext.Database.BeginTransaction())
{
    try
    {
        current.Feedback = hm.Feedback;
        current.Grade = hm.Grade;
        dbcontext.SaveChanges();
        dbContextTransaction.Commit();
    }
    catch (Exception)
    {
        dbContextTransaction.Rollback();
    }
}
```

Screenshot 9 (Transaction code)

Automated Migration

We used Code First Migrations to accomplish this technique. It allows us to change the model without having a migration file every time. Then, we need to set a new database initializer in the context class (in our project is ApplicationDbContext located in IdentityModels.cs). The new initializer method shows as the screenshot 11. The commented code is the default initializer method.

```
namespace GradingSystem.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration : DbMigrationsConfiguration<GradingSystem.Models.ApplicationDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }
```

Screenshot 10 (two attributes in Configration.cs file after enabled automatic migrations in the console)

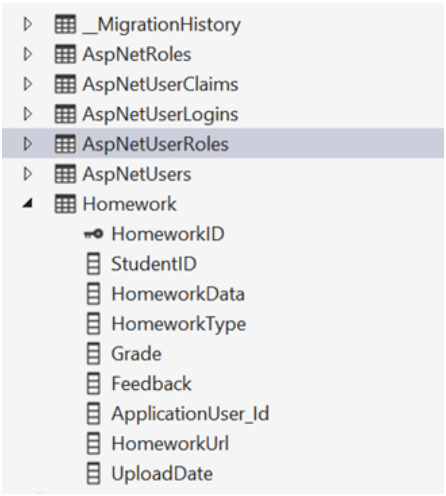
To do this, we run the **Enable-Migrations -EnableAutomaticMigrations:$true** command in the Package Manager Console. Then in the Configuration.cs file, the **AutomaticMigrationsEnabled = true** is in the constructor. We also need to set **AutomaticMigrationDataLossAllowed = true**, so we can delete the data in description column.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
        //Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ApplicationDbContext>());
        Database.SetInitializer(new MigrateDatabaseToLatestVersion<ApplicationDbContext, Migrations.Configuration>("DefaultConnection"));
    }
    public DbSet<Homework> HomeworkDB { get; set; }
    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }
}
```

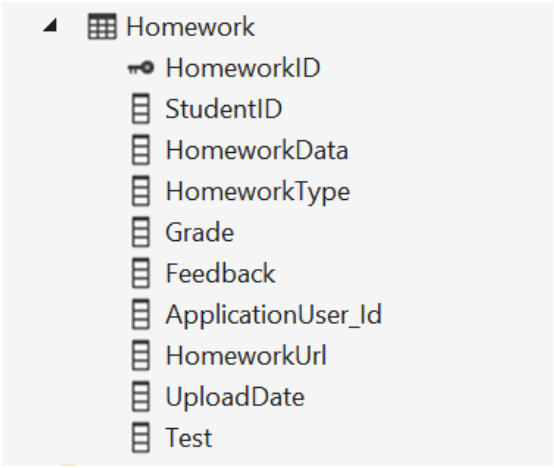
Screenshot 11 (new initializer method in the IdentityModels.cs file)

After we used Code First Migration there is a new _MigrationHistory table in the database which maintains the history of database changes of automated migration.

Every time we change the model like adding a new column like "public string Test { get; set; }", then use Update-Database command in the Package Manage Console to automatically save all the changes. Screenshot 13 shows the change made after we added a new line of code in the Homework.cs file.



Screenshot 12(_MigrationHistory table created)



Screenshot 13

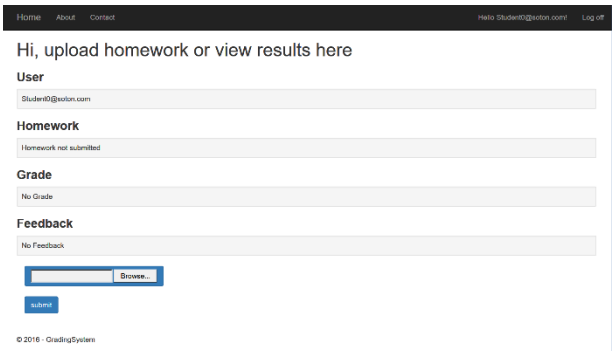
(new string type "test" showed in the database)

3. Test

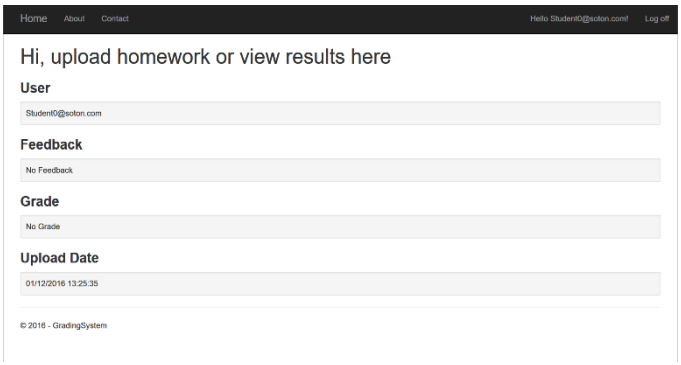
We basically follow the assignment instructions and test all the functions that we achieved.

3.1. Test for Students part

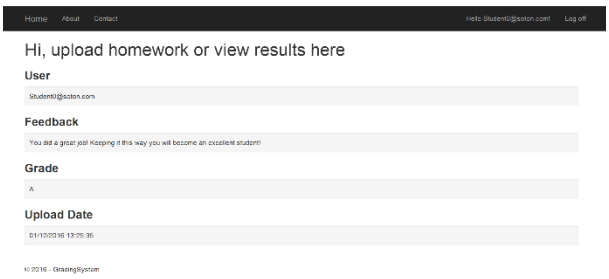
Students log in the website. The page will present the **Index** view. If students haven't submitted their homework, the page will look like screenshot 14. After students uploaded the HTML file, they can view their grade and feedback at the page (screenshot 16). Students is allowed to upload only HTML file. So, when students click the button to browse files in the file explore, only html file is shown. After the file being uploaded, data are showed at the database.



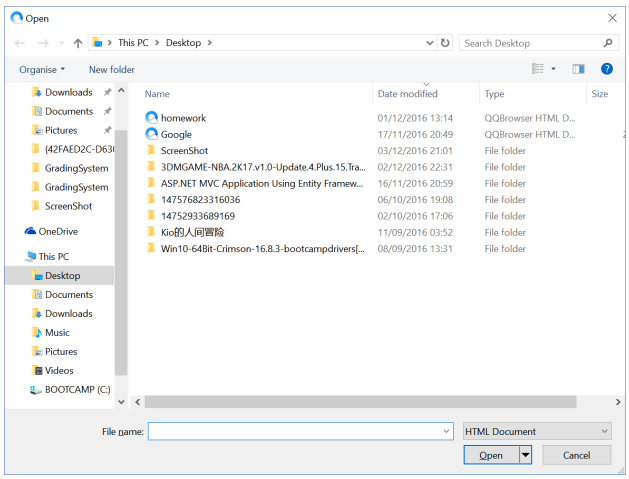
Screenshot 14



Screenshot 15 (Student index view without marked)



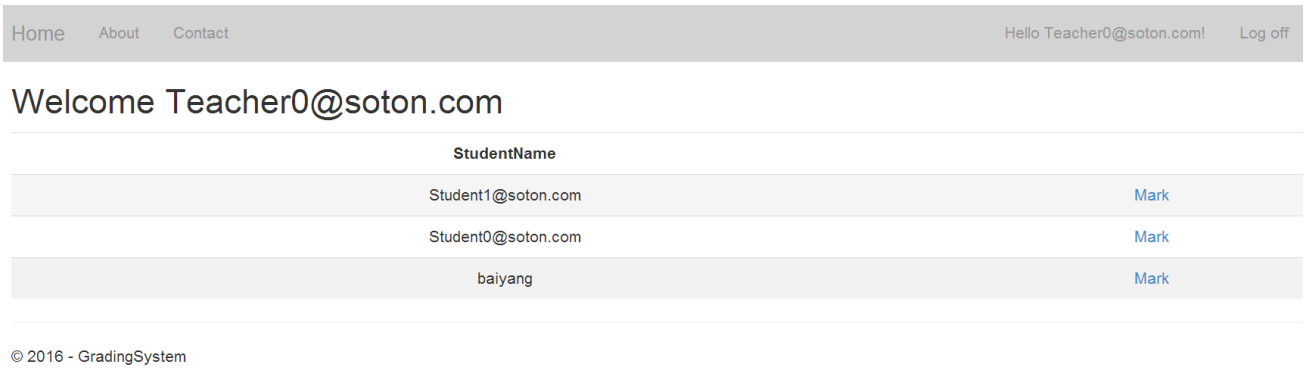
Screenshot 16



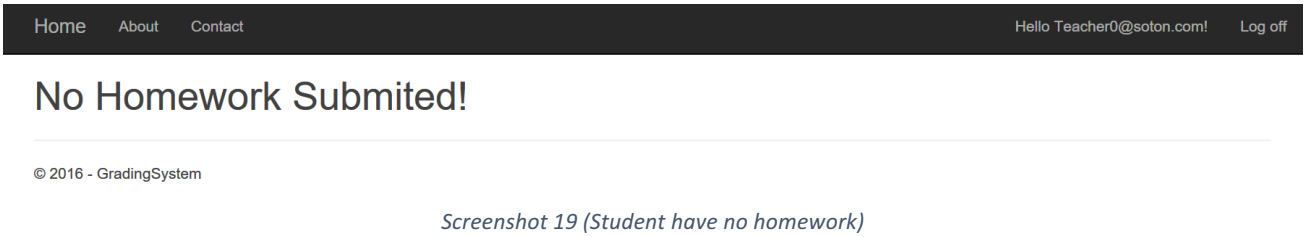
Screenshot 17 (only can upload html file)

3.2. Test for Teacher Part

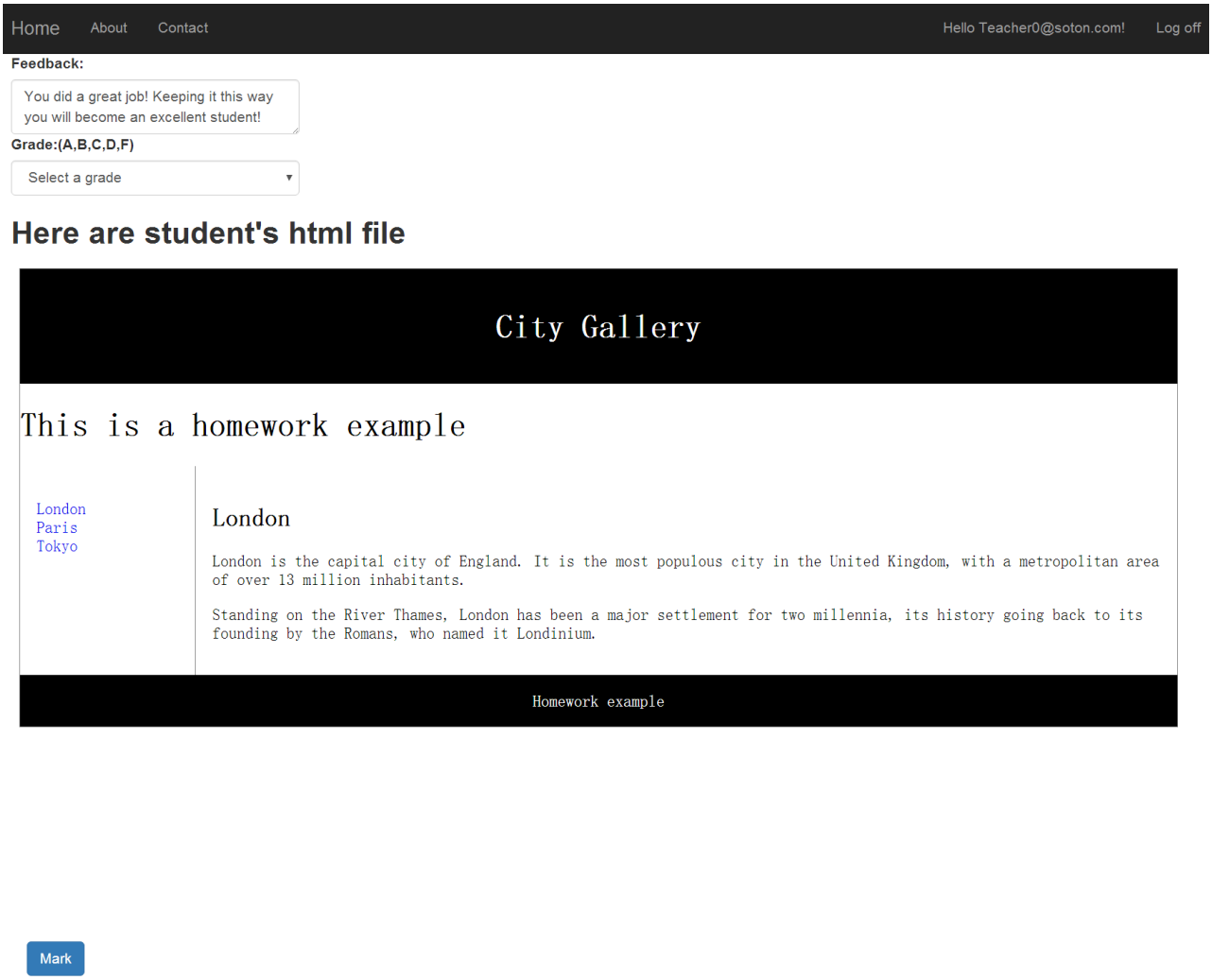
Using teacher account to log in, the page will show a table of all students (screenshot 18). Clicking **Mark**, mark view will be displayed. Teacher will know whether a student has submitted the homework or not. If teacher submit wrong format of feedback or grade, the error message will appear (screenshot 22). After teacher marks, the page will be redirected to a view which shows the mark detail. We have checked that invalid data won't be saved to database and error message will pop up if the input data is invalid. All functions meet the requirements.



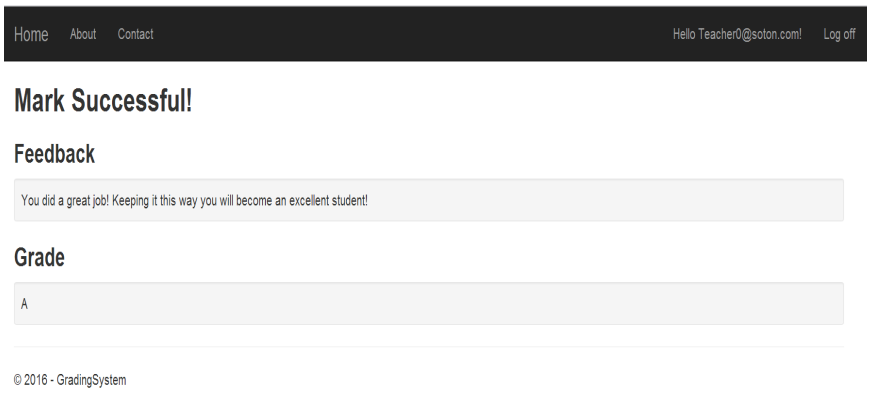
Screenshot 18 (Index View)



Screenshot 19 (Student have no homework)



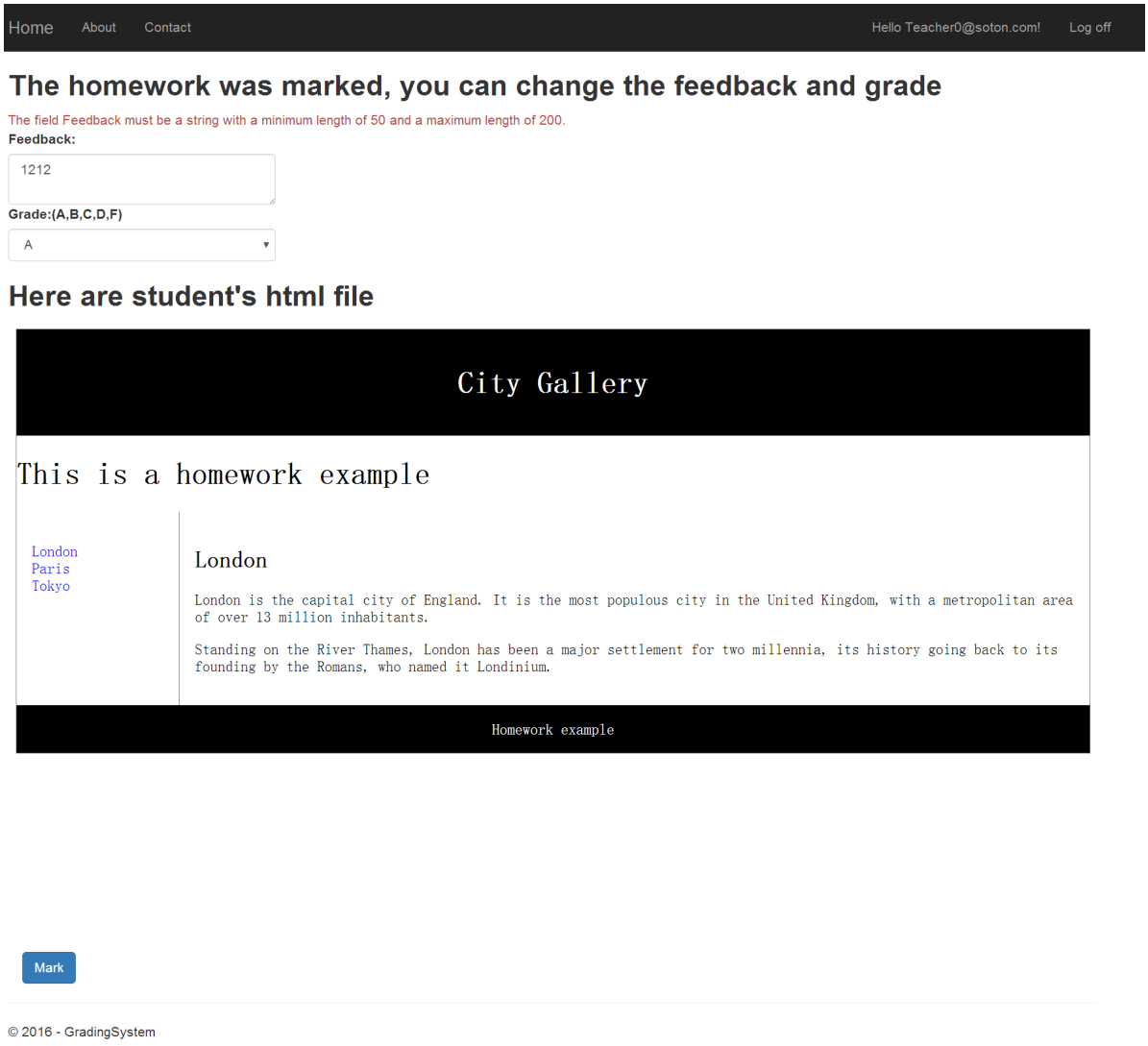
Screenshot 20 (Marking)



Screenshot 21

3.3. Test Portability

We test our web site on different browsers like Chrome Edge and Firefox and it worked very well. All our views are responsive. We also used Chrome to simulate mobile device screen, the view changed dynamically.



Screenshot 22 (Marking with error message)

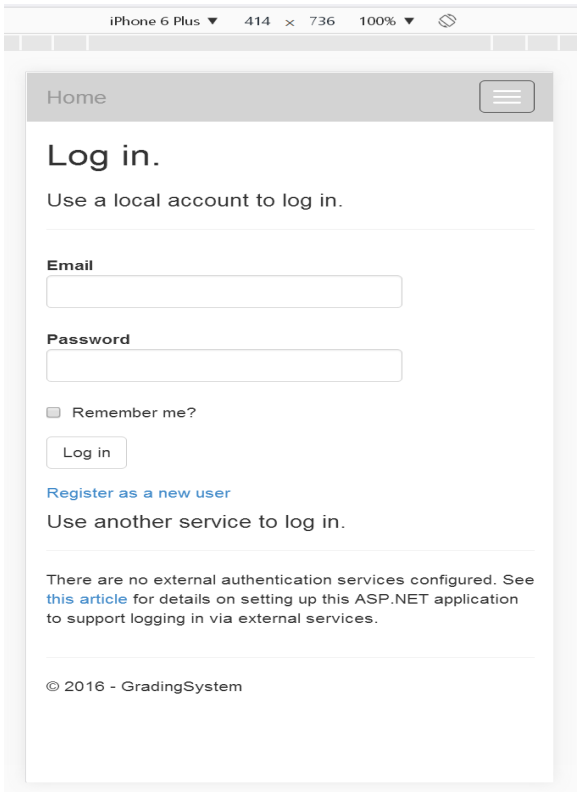
4. Evaluation

The whole structure of ASP.NET MVC is easy to understand, because it uses MVC pattern which helps people with some experiences in object-oriented develop techniques and MVC develop pattern.

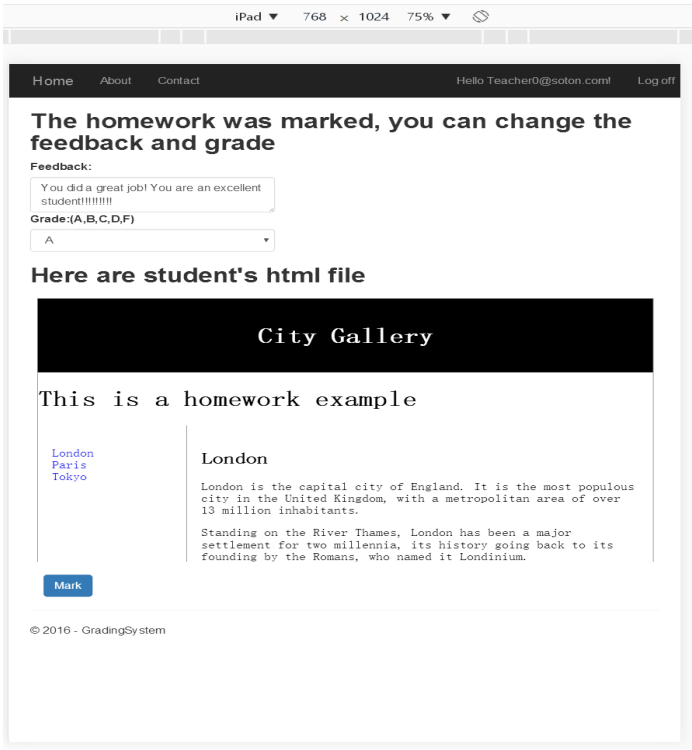
When we design the model, a .NET Framework called Entity Framework is a very useful tool to help us access the database from our program file. Instead of writing lines of complex SQL sentences, we create a new model by writing a simple class because the Code-First approach provided by Entity Framework. After creating a Dbcontext in the model, it becomes very convenient to modify the data since the Dbcontext will handle those operations that related to the database. LINQ syntax also allows us to query the Dbset of the model. In this course work, we need to find the file that students uploaded that matched the student ID and find the file by its own ID. It can be easily done by using LINQ. The use of Code-First also allows us to finish the additional technology smoothly with the detailed document from MSDN. Adding few instructions in the Package Manager Console, features like automatic migrations can be accomplished in few minutes. However, we spent lots of time to fix the bug when we are trying to implement automated database migration. The error message from Package Manager Console didn't give any help and we finally figure out it is the previous migration files interfered the process of setting up automatic migration. Sometimes it may not a good thing that these frameworks hide some fundamental details for developers while these frameworks provide convenience.

In the course work, MVC template and ASP.NET identity let us put less effort in web page structure and authentication. ASP.NET identity was evolved from ASP.NET membership, so it integrated numbers of helpful features. From those parts that we use in this project, it has role provider which enables us to create student role and teacher role without extra effort. Users' basic attribute are created in the database by default, and it is convenient to get users' information by calling **UserManager** and **RoleManager** class. In the controller, authorizing certain role to access pages is simply done by adding"[Authorize(Roles="Somerole")]".

The get-redirection-post pattern gives us a clear view to implementing the required functions. Razor syntax allows us to make the View become a dynamic page. We use Razor to decide what specific content should present in the view, like the upload button only shows when students have not submitted their homework. The C# statement makes the view more powerful than a simple HTML file. HTML helper saves a lot of work for us when we create a link or editing the model in view.



Screenshot 23



Screenshot 24

C# is a general-purpose, object-oriented programming language. We found C# is easy to get started benefit from our previous object-oriented programming experience, though we are both new in .NET area. The official documentation and tutorial are extremely valuable. ASP.NET provide many templates and frameworks to keep us focus on the core logic of this course work, which shows its power in web application construction. There are many developers who are using ASP.NET technology, therefore, tons of questions and answers can be found online and these lead us to go smoothly during the whole process. The strong typing character and the most powerful IDE help us to discover some mistakes that are difficult to detect with our eyes.

5. Comparison between ASP.NET MVC and AngularJS.

Both ASP.Net MVC and AngularJS are cross-platform open source web application framework. The former is a server side framework which uses the standard navigation in between pages powered by Microsoft, whereas the latter is a client-side single page application framework mainly maintained by Google.

AngularJS SPA is use to build web applications which provide smooth user experience close to desktop application. When a user interacts with the app, HTML code, JavaScript code, CSS code and related data are dynamically fetched while loading just a single page. SPA uses AJAX and Html5 to build responsive and fluid web page without frequent page reloads. Its goal is to make development and test easy. When the first page is loaded, communications between client and server are built through AJAX calls which returns data in JSON format. And these JSON format data is used to update the page dynamically as illustrated in figure 5. However, in ASP.NET MVC (5), every time the server receives a request, it renders a new HTML page and return it to client which involves a page refresh showed in figure 5.

By comparison, ASP (6) is more fluid and responsive without re-rendering and reloading the page. AngularJS SPA implements MVC pattern as well in order to separate display, data and logic parts. Views containing the HTML code add data driven dynamism to the web app. Angular first interprets the tag attributes embedded in the html code as directives to bind either output or input data to a model represented by JavaScript variables. Empowered by

dependency injection, Angular drives server-side services to client-side web applications, thus significantly reducing the burden of the server.

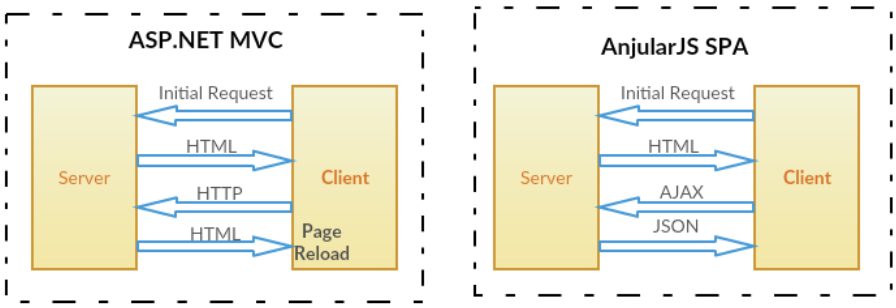


Figure 5

6. Summary

After two months of developing, we have gained a better understanding about ASP.NET MVC framework. Hopefully in the future, we could bring what we have learnt to industry.

7. Bibliography

1. MSDN. ASP.NET MVC 5 Security And Creating User Role. *Microsoft Developer Network.* [Online] https://code.msdn.microsoft.com/ASPNET-MVC-5-Security-And-44cbdb97.

2. —. Customizing ASP.NET Authentication with Identity. *Channel 9.* [Online] 05 12 2014. https://channel9.msdn.com/Series/Customizing-ASPNET-Authentication-with-Identity.

3. Entity Framework Working with Transactions (EF6 Onwards). *Data Developer Center.* [Online] 23 10 2016. https://msdn.microsoft.com/en-us/data/dn456843.aspx.

4. Entity Framework Tutorial. Automated Migration. *Entity Framework Tutorial.* [Online] http://www.entityframeworktutorial.net/code-first/automated-migration-in-code-first.aspx.

5. Microsoft. Get Started with ASP.NET. *ASP.NET.* [Online] https://www.asp.net/get-started.

6. Wikipedia. AngularJS. *Wikipedia.* [Online] https://en.wikipedia.org/wiki/AngularJS.