Amm1g13
Rb7e15

ASP.NET MVC Prototype Web Site

This work illustrates development and testing of a HTML coursework marking web application.

Required features, Technologies and Techniques

Application Structure

This application was structured around two user roles, namely Teacher and Student, as can be seen in **Figure 1**.

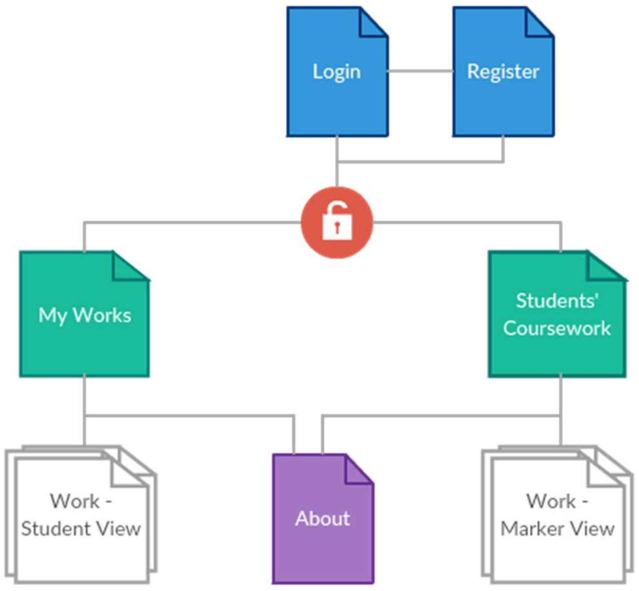


Figure 1– Site structure diagram

These required the following database model, as seen in the figure below.

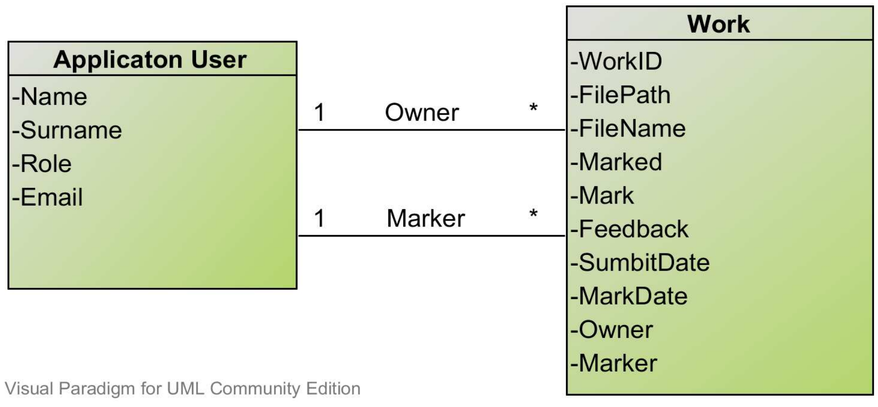


Figure 2 - Data structure diagram

The code was structured following the MVC pattern, including a separate testing project. Directories are marked in **bold**, Web root is omitted.

OnlineMarkerCW

- Program.cs
- Startup.cs
- **Views**
  - **Account**
    - Login.cshtml
    - Register.cshtml
  - **Home**
    - Index.cshtml
    - About.cshtml
    - MyMarkings.cshtml
    - MyWorks.cshtml
    - WorkView.cshtml
    - WorkViewMarker.cshtml
  - **Shared**
    - Layout.cshtml
    - Error.cshtml
  - _ViewImport.cshtml
  - _ViewStart.cshtml
- **ViewModels**
  - AccountViewModels.cs
  - HomeViewHodels.cs
- **Models**
  - ApplicationUser.cs (consider rename)
  - Models.cs
- **Controllers**
  - AccountController.cs
  - HomeController.cs
- **Data**
  - ApplicationDbContext.cs
- **Filters**
  - Filters.cs ([AnonymousOnly])
- **Interfaces**
  - Interfaces.cs
- **Services**
  - Services.cs
- [Configuration Files]

OnlineMarkerCw.Test

- AccontControllerTests.cs
- CustomFiltersTests.cs
- HomeControllerTests.cs
- ServicesTests.cs
- ViewModelTests.cs

Figure 3- Data structure diagram

Technologies and Techniques Used

ASP.NET MVC core

ASP.NET MVC core 1.0, used in this project, is a freshly franchised version of the ASP.NET (1). The biggest differences from the previous released include moving away from the proprietary development and making the platform open source and cross platform. The key idea behind the redesign of the platform was making it modular in its design and architecture. This separates the compiler, runtime and the libraries to be independent components, which closely relates to Dependency Injection, as discussed in one of the next sections.

ASP.NET MVC core provides a set of tools which eases the creation of the corresponding components in order to follow the MVC pattern. A controller is defined as a class, which includes a set of actions which handle incoming requests (8).Controllers inherit from the *Controller* base class, which contains predefined methods and properties necessary to process an incoming HTTP request. A code sample bellow shows a definition of an *Account Controller* and a *Logout Action Handler*, contained under */src/controllers/AccounConroller.cs*. The annotations defined in the square brackets before the *Logout() action task*, indicate the extra filters that request has to go through before reaching the request handler. This action sing outs and redirects the user to the login page.

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager; //object for creating and managins users
    private readonly SignInManager<ApplicationUser> _signInManager;//object for singing in the user and creating a session
    private readonly RoleManager<ApplicationUserRole> _roleManager; //role manager for managing student/teacher roles
    private readonly ILogger _logger;//logger for debuging.
    private readonly IDbServices _dbServices;//dbservice methods.

    //Inject the dependencies into the controller via the constructor
    public AccountController( UserManager<ApplicationUser> userManager,  SignInManager<ApplicationUser> signInManager,
    RoleManager<ApplicationUserRole> roleManager, ILoggerFactory loggerFactory,IDbServices dbServices )
    {
      _userManager = userManager;
      _signInManager = signInManager;
      _roleManager = roleManager;
      _logger = loggerFactory.CreateLogger<AccountController>();
      _dbServices = dbServices;
    }

    //POST: /Account/Logout
    [HttpPost]
    [Authorize]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Logout()
    {
        //sign out the user out of the system
        await _signInManager.SignOutAsync();
        //Redirect user to the login page
        return RedirectToAction(nameof(AccountController.Login), "Account");
    }
}
```

Sample 1 - Definition an Account controller

Data model entities within the framework are simply defined a class as shown in **Sample 2**. The properties of the entity are created as the properties within that class. These can include Data Annotations defined in square bracket.

```
// Define the applicaiton User Entity and its properties
public class ApplicationUser : IdentityUser
{
  [Required]
  public string Name    { get; set; }
  [Required]
  public string Surname { get; set; }
}
```

Sample 2 – Definition of the Application User entity. Available under /src/models/ApplicationUsers.cs

The *ViewResult* object is generated when an action handler returns a *View()* method as a result. The *ViewResult* object typically contains a model data that has been queried and *ViewData* that has been generated by the action handler to be passed to the Razor templating engine.

```
//GET: /Account/Login
[HttpGet]
[AnonymousOnly]
public IActionResult Login(string returnUrl = null)
{
    //save redirect data in the view, so that it can preserved for a post request
    ViewData["ReturnUrl"] = returnUrl;
    return View("Login");
}
```

Sample 3 - Login handler which returns a View() method which generated the ViewResult object. Available under /src/Controllers/HomeControllers.cs

Authentication and Authorisation

Authentication and Authorisation features are achieved using the Identify system (10). It provides a mechanism for creating User, SignIn and Role managers to address the security concerns of a web application. In order to use them, an AplicationUser entity extending the IdentityUser has to be created. The use of the Identity managers is demonstrated under the Registration action handle in **Sample 4.** It also demonstrates using User Claims as session variables, which is a common approach (11). It enables the authentication cookie to serve as a session cookie and makes user's related claims information available from the controller's context properties, without having to access the database.

```
//POST: /Account/Register
[HttpPost]
[AnonymousOnly]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
  if (ModelState.IsValid)//checks if the incomming values from the reuqest can be mapped on the model.
    {
        ApplicationUser user = new ApplicationUser() {user.UserName = model.Email, user.Email = model.Email, user.Name = model.Name, user.Surname = model.Surname };

        var result = await _userManager.CreateAsync(user, model.Password); //await in async for user to be created
        if (result.Succeeded) //if registration succeeded, singin the user and redirect home
        {
          string userType;
          userType = model.UserTypeID == 0 ? "Student" : "Teacher"; //check what value is passed from the request, if UserTypeID is 0 it is student, esle it is Teacher
          if(!_roleManager.RoleExistsAsync(userType).Result) //if role does not exists, create a new one.
             {
               ApplicationUserRole role = new ApplicationUserRole();
               role.Name = userType;
               role.Description = userType;
               var roleResult = _roleManager.CreateAsync(role).Result;
               if(!roleResult.Succeeded)
               {
                 AddErrors(roleResult);
                 return View("Register",model);
               }
             }
          //store claims about the suer, so that they can be accessed from the controllers ContextData without having to read the db (like a session)
          await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Email, model.Email));
          await _userManager.AddClaimAsync(user, new Claim("Name",  model.Name));
          await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Surname, model.Surname));
          await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Role, userType));

          await _userManager.AddToRoleAsync(user,userType); //add the user to the role

          await _signInManager.SignInAsync(user, isPersistent: false); //sing in the user

          return RedirectToAction(nameof(HomeController.Index), "Home"); //redirect to home page
        }
        //If creating user is unsuscefull, add errors to the ModelState
        AddErrors(result);
    }
```

Sample 4 – Registration action handler, Available under /src/Controllers/AccountControllers.cs

Entity Framework Core

The core, lightweight and cross platform version of the Entity Framework was used as an Object-Relational mapper to map the .NET model objects to the cross-platform SQLlite database (12). Entity Framework uses a predefined DB context (**Sample 5**) to establish a connection and query or modify the database.

```
//Define DB context for the applcicaiton, derive it from the Identity context so that one context is used for
the Indeity framework and the application itself.
  public class ApplicationDbContext : IdentityDbContext<ApplicationUser,ApplicationUserRole, string>
  {
      public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) {}

      //include works into the context
      public virtual DbSet<Work> Works { get; set; }

      protected override void OnModelCreating(ModelBuilder modelBuilder)
      {

          base.OnModelCreating(modelBuilder);
          // Customize the ASP.NET Identity model and override the defaults if needed.
          // For example, you can rename the ASP.NET Identity table names and more.
          // Add your customizations after calling base.OnModelCreating(builder);
      }
  }
```

Sample 5 – Definition of a database context. Available under /src/Data/ ApplicationDbContext.cs

Modifications and Queries are performed via the context on the selected entities, as shown in **Sample 6** for the Work entity. The ORM reads the DB and returns a usable .NET object which can be passed to the controller to perform necessary logic operations.

```
//Get a full list of works object including their owner objects
public async Task<List<Work>> GetWorksAndOwners()
{
  //Include Owner tells the framework to load the foreign key field Owner, otherwise it will be null.
    return await _context.Works.OrderBy(w => w.SubmitDate).Include(w => w.Owner).ToListAsync();
}

//Add a new work to the DB
public void AddWork(Work work){
    _context.Works.Add(work);
    _context.SaveChanges();
}
```

Sample 6 – Samples of the Database manipulation and queuing using the DBContext. Available under /src/Services/ Services.cs

Razor based Templating Engine

Razor based Templating engine provides a simple syntax which consist of Razor markup, C# and HTML for rendering HTML pages (13).

```
@{
  ViewData["Title"] = "My Markings";
}
@model  List<OnlineMarkerCW.Models.Work>


<h1>Students' Coursework</h1>
<br>

<div class="submissions_list">

    <p>Student</p>
    <p>Submission</p>
    <p>Mark</p>
    <ol>
        @foreach (var work in Model){
            <li>
                <p>@work.Owner.Name @work.Owner.Surname</p>

                <p class="middle-col">
                    <a asp-controller="Home" asp-route-id="@work.WorkID" title="@work.FileName"
                    asp-action="WorkViewForMarker"> @work.FileName</a>
                    <span> - @work.SubmitDate</span>
                </p>

                @if (@work.Marked){<p class="@( @work.Mark < 40 ? "generic-error" : "generic-message")">
                @work.Mark </p>}
                else { <p>Not marked</p> }
            </li>
        }
    </ol>
</div>
```

Sample 7 – My Markings view, available under /src/Views/Home/MyMarkings.cshtml

Symbol @ specifies all Razor syntax, followed by C# code, that includes loops and branching statements, etc. – as seen in the sample. Local variables and model can be declared to be used by the view.

A new construct of tag helpers has in introduced under ASP.NET core MVC. Tag helpers replaced big part of the HTML helpers' functionality, for generating forms and action links, as seen in **Sample 9** of the next section. They use html attribute like syntax, hence easier to read and manipulate to generate the HTML code (14).

Input Validation and Testing

User data input validation on both server and client side can be easily achieved with combination of ViewModel and tag helpers. **Sample 8** shows a definition of a ViewModel. The attributes which decorate the properties of the class indicate what validation is to be done on the server side. Many of them are transferred via the tag helpers to the client side (15).

```
public class LoginViewModel
{
    [Required]
    [EmailAddress(ErrorMessage = "Invalid Email Address Format.")]
    [Display(Name = "Email")]
    9 references
    public string Email { get; set; }
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
    MinimumLength = 6)]
    9 references
    public string Password { get; set; }
}
```

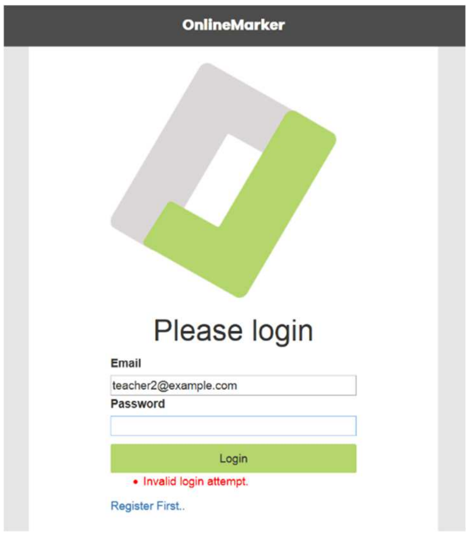
Sample 8 – Login ViewModel, available under /src/ViewModels/AccountViewModels.cs

Client side validation attributes such as *min, max length and required* are not generated by the tag helpers hence have to be defined manually (**Sample 9**).

```
@{
    Layout = "~/Views/Shared/_RegisterLayout.cshtml";
    ViewData["Title"] = "Login";
}


@model OnlineMarkerCW.ViewModels.LoginViewModel


<h1 class="login-heading"> <span> Please login </span></h1>
<form asp-controller="Account" asp-action="Login" asp-route-returnurl='@ViewData["ReturnUrl"]' method="post"
  <div class="center-block"><label asp-for="Email"></label></div>
  <div class="center-block"><input asp-for="Email" required/></div>
  <div class="center-block"><label asp-for="Password"></label></div>
  <div class="center-block"><input asp-for="Password" minlength="6" maxlength="100" pattern=".{6,100}"
  title="Should be between 6 and 100 characters" required/></div>
  <div class="center-block"><input type="submit" value="Login" class="reg" /></div>
</form>
  <div asp-validation-summary="All" class="generic-error"></div>
  <a asp-controller="Account" asp-action="Register">Register First..</a>
```

Sample 9 – Login page View, available under /src/Views/Account/Login.cshtml

It was confirmed that this technology preserved the input in non-password fields when reloading the page via manual testing, as seen in the figures below.



Please login

Email

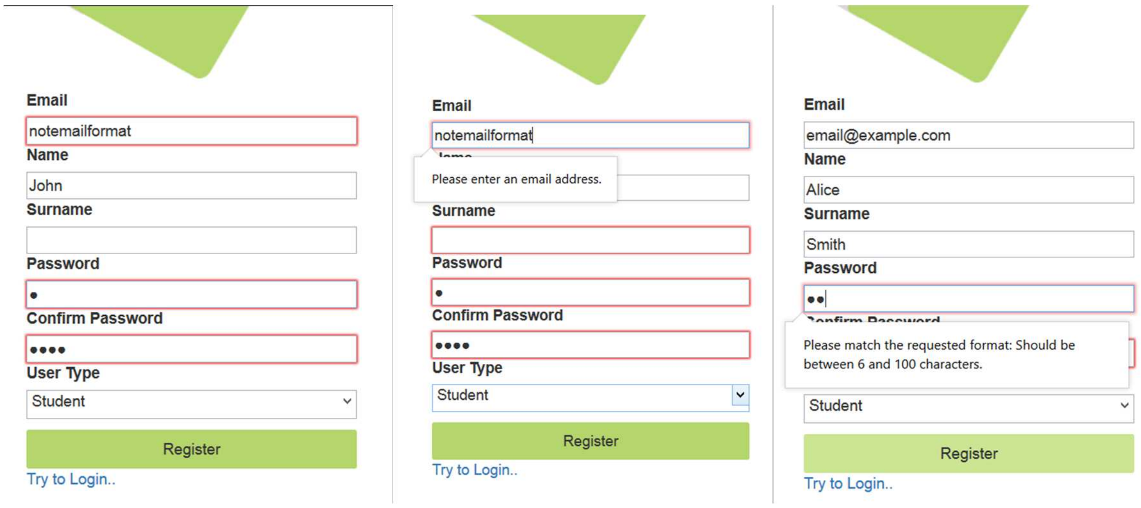
teacher2@example.com

Password

• Invalid login attempt.

Figure 4 - Login Screen, Error Reporting and Input Preservation

TagHelpers also took care of client-side validation dynamically. Even before the post request is made, they will highlight in red those fields that have failed their validation. After the post request with invalid data is made, they will also create a popup to indicate the error.



Email

notemailformat

Name

John

Surname

Password

•

Confirm Password

••••

User Type

Student

Register

Try to Login..

Email

notemailformat

Please enter an email address.

Surname

Password

•

Confirm Password

••••

User Type

Student

Register

Try to Login..

Email

email@example.com

Name

Alice

Surname

Smith

Password

••

Please match the requested format: Should be between 6 and 100 characters.

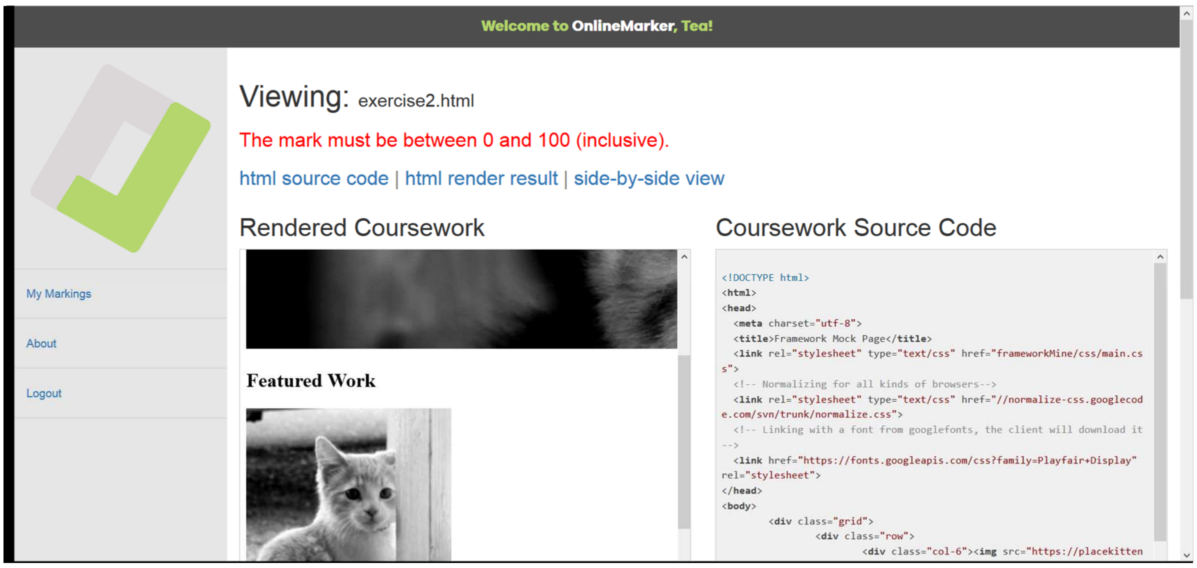
Student

Register

Try to Login..

Figure 5- Client Side Input Validation

Though already tested by unit tests, business logic was also manually tested, ensuring error reporting works as intended (example below).



Welcome to OnlineMarker, Tea!

Viewing: exercise2.html

The mark must be between 0 and 100 (inclusive).

html source code | html render result | side-by-side view

Rendered Coursework

Featured Work

Coursework Source Code

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Framework Mock Page</title>
  <link rel="stylesheet" type="text/css" href="frameworkMine/css/main.cs
s">
  <!-- Normalizing for all kinds of browsers-->
  <link rel="stylesheet" type="text/css" href="//normalize-css.googlecod
e.com/svn/trunk/normalize.css">
  <!-- Linking with a font from googlefonts, the client will download it
-->
  <link href="https://fonts.googleapis.com/css?family=Playfair+Display"
rel="stylesheet">
</head>
<body>
        <div class="grid">
                <div class="row">
                        <div class="col-6"><img src="https://placekitten
```

Figure 6 - Error reporting after invalid mark (assuming client-side validation was circumvented)

Addition server side validation was required for the file upload functionality, that was included in the controller action for testing file size and extension (**Sample 10**)

```
//check that file is not empty and in the right extension
    if (file != null && file.Length > 0 &&  file.FileName.EndsWith(".html")) {
              using (var fileStream = new FileStream(Path.Combine(uploads,  timeNow + "_" +  file.FileName), FileMode.Create))
              {
                try  {
                  //save a db entry to the db
                  Work work = new Work ();
                  work.FileName = file.FileName;
                  work.FilePath = Path.Combine(uploads,  timeNow + "_" +  file.FileName);
                  work.SubmitDate = localDate;
                  work.Owner = user;
                  _dbServices.AddWork(work);
                  //save the file stream to the file
                  await file.CopyToAsync(fileStream);
                  ViewData["upload-message"] = "File upload sucessfull";
                } catch (Exception ex) {
                  //if there are any expection, catch the Error messages
                  ModelState.AddModelError("File", "Upload failed: " +ex.Message.ToString());
                }
            }
    }
```

Sample 10 - File Upload Server Side Validation

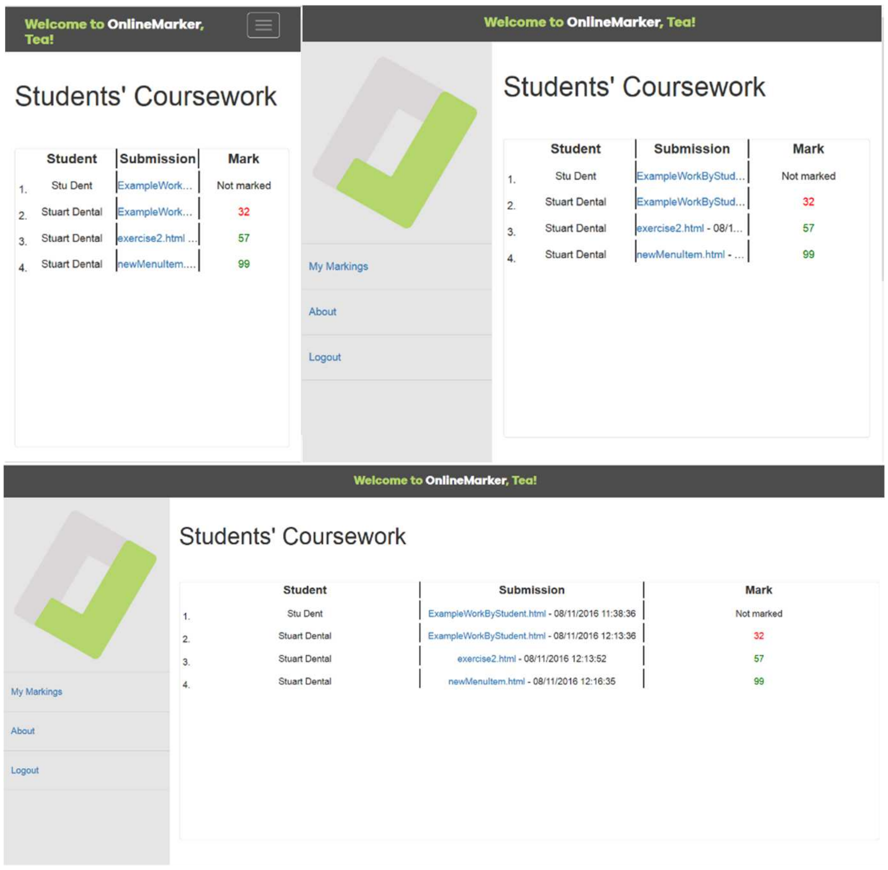
Professional Layout and Responsiveness, portability testing

The professional layout, and responsiveness of the application was achieved though combination of the CSS3 techniques and the Bootstrap Framework. The CSS3 *@media* rule was used extensively in order to achieve scalable design. The sample below shows that for screen sizes 768px wide and over the layout elements should have larger dimensions.

```css
/*For bigger screens only*/

 @media(min-width:768px) {
   .body-content {
      margin-left: 260px;
     min-height: 900px;
   }

   .login-content {
     min-width: 500px;
     padding-left: 100px;
     padding-right: 100px;
   }

  .sidebar {
      z-index: 1;
      position: absolute;
      width: 260px;
  }

}
```

Sample 11- Layout size parameter definitions for bigger screens, available under /src/wwwroot/css/site.css

Portability was tested by running the application in several browsers (Chrome, Firefox, Safari, Edge) and using caniuse.com (27), a site that offers HTML and CSS compatibility data. The application was run on differently sized browser windows to test responsiveness. Note how the left-hand-side menu disappears and is accessible through a button in mobile screens.



Students' Coursework

	Student	Submission	Mark
1.	Stu Dent	ExampleWork...	Not marked
2.	Stuart Dental	ExampleWork...	32
3.	Stuart Dental	exercise2.html ...	57
4.	Stuart Dental	newMenuItem....	99

Students' Coursework

	Student	Submission	Mark
1.	Stu Dent	ExampleWorkByStud...	Not marked
2.	Stuart Dental	ExampleWorkByStud...	32
3.	Stuart Dental	exercise2.html - 08/1...	57
4.	Stuart Dental	newMenuItem.html - ...	99

Students' Coursework

	Student	Submission	Mark
1.	Stu Dent	ExampleWorkByStudent.html - 08/11/2016 11:38:36	Not marked
2.	Stuart Dental	ExampleWorkByStudent.html - 08/11/2016 12:13:36	32
3.	Stuart Dental	exercise2.html - 08/11/2016 12:13:52	57
4.	Stuart Dental	newMenuItem.html - 08/11/2016 12:16:35	99

Figure 8 - Responsiveness/Device Portability Showcase

Advanced Technologies and Techniques

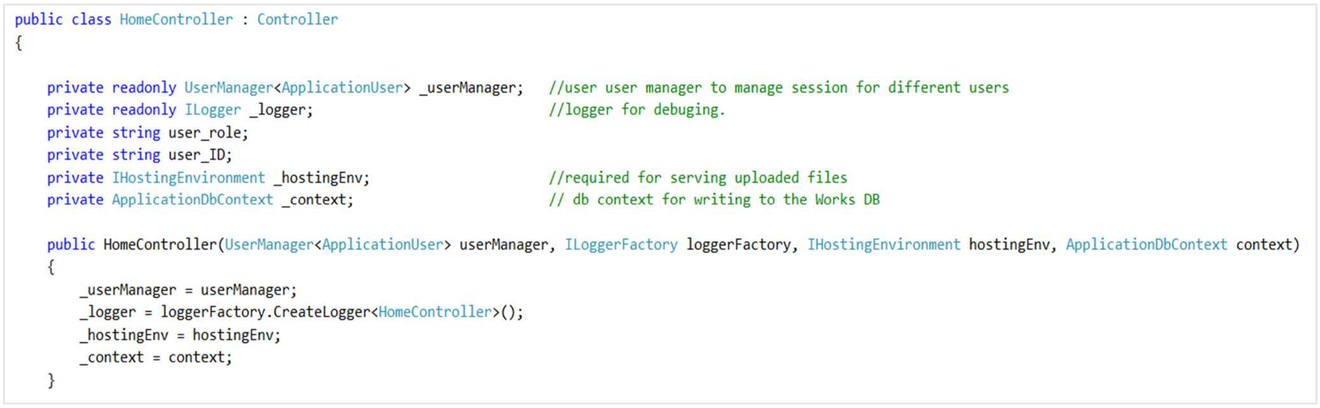
Dependency Injection

Dependency Injection (DI) is a widely recognised and applied design pattern in modern applications. It came about due to the spread of the Inversion of Control (IoC) design principle, which ASP.Net makes use of. Following this principle, the ASP.Net web application is driven primarily by the framework, which registers events such as user requests and then is the one which calls methods in the application's code (which are manually coded to handle these events) (4) (5). This is said to "invert" the placement of control because it is no longer the application code that directs the program flow and calls standard library functions, but the opposite. IoC allows for highly modular code, which is greatly desirable due to its increased maintainability, testability and reusability.

In a modular program, sections of code work together and thus may rely on each other, creating dependencies. These dependencies become problematic in the context of unit testing, where it is necessary to isolate code modules (in Object Oriented programming, these are often objects) in order to test their correctness without considering that of any other code, even if this code is a dependency of the module to be tested. Dependency Injection solves this problem by adding additional layers of abstraction between the module to be tested and any modules it depends on.

Fortunately, ASP.NET Core was built with DI in mind from the beginning. Soon after the entry point of the web application, Startup.cs calls ConfigureServices, and it is here that the DI container gets configured. Here "container" is a class which manages the creation of services and provides instances of them to whichever classes need them (in this case, our controllers). See **Sample 14**. By specifying dependencies here, the framework takes care of doing the injections for the programmer.

In practice, the code should also follow the Explicit Dependencies Principle (12), which means controllers should explicitly require their dependencies in their constructors.

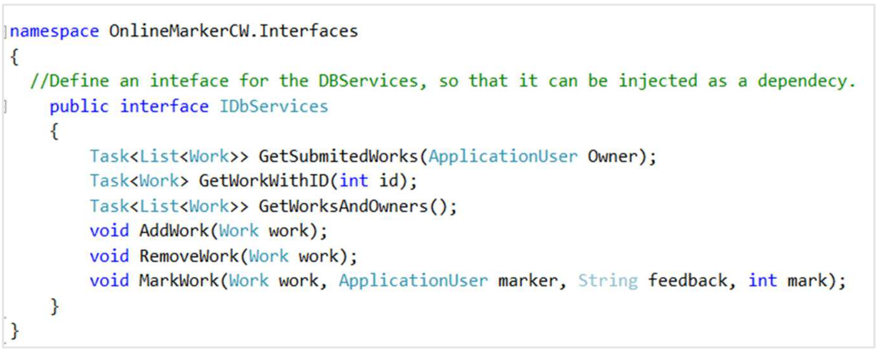


```
public class HomeController : Controller
{

    private readonly UserManager<ApplicationUser> _userManager;   //user user manager to manage session for different users
    private readonly ILogger _logger;                             //logger for debuging.
    private string user_role;
    private string user_ID;
    private IHostingEnvironment _hostingEnv;                      //required for serving uploaded files
    private ApplicationDbContext _context;                        // db context for writing to the Works DB

    public HomeController(UserManager<ApplicationUser> userManager, ILoggerFactory loggerFactory, IHostingEnvironment hostingEnv, ApplicationDbContext context)
    {
        _userManager = userManager;
        _logger = loggerFactory.CreateLogger<HomeController>();
        _hostingEnv = hostingEnv;
        _context = context;
    }
```

Sample 12- HomeController with explicit dependencies and before Dependency Inversion Principle

Additionally, DI best practice requires the code to adhere to the Dependency Inversion Principle. According to this pattern, the controller should not depend on concrete instantiations of classes, but on abstractions of them. In other words, these dependencies had to be transformed into interfaces. By doing this, controllers ask only for abstractions and the DI container has the flexibility to decide the implementation they receive.

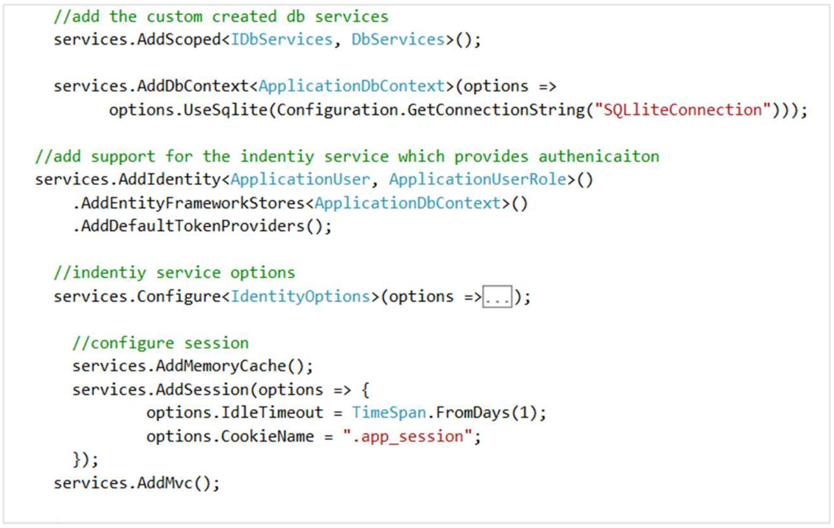
The main dependency in the HomeControlelr.cs was the ApplicationDbContext, a class wrapping the database connection. Thus, this class was extracted into an interface, IDbServices. Then it was necessary to create a class that implements said interface and move all the relevant code from the HomeController to this class (the code which used the ApplicationDbContext).



```
namespace OnlineMarkerCW.Interfaces
{
  //Define an inteface for the DBServices, so that it can be injected as a dependecy.
    public interface IDbServices
    {
        Task<List<Work>> GetSubmitedWorks(ApplicationUser Owner);
        Task<Work> GetWorkWithID(int id);
        Task<List<Work>> GetWorksAndOwners();
        void AddWork(Work work);
        void RemoveWork(Work work);
        void MarkWork(Work work, ApplicationUser marker, String feedback, int mark);
    }
}
```

Sample 13- IDbServices interface, available /src/interfaces/Interfaces.cs

As the final step, this service was added in the ConfigureServices, seen below.



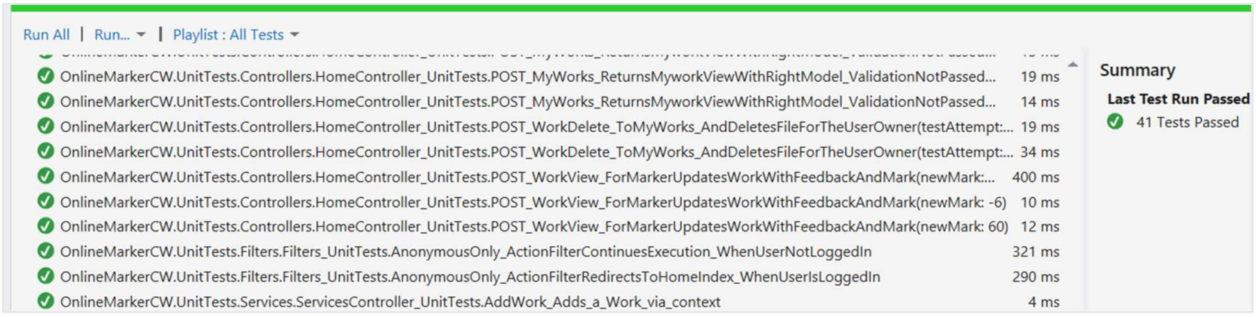
```
    //add the custom created db services
    services.AddScoped<IDbServices, DbServices>();

    services.AddDbContext<ApplicationDbContext>(options =>
          options.UseSqlite(Configuration.GetConnectionString("SQLliteConnection")));

  //add support for the indentiy service which provides authenicaiton
  services.AddIdentity<ApplicationUser, ApplicationUserRole>()
      .AddEntityFrameworkStores<ApplicationDbContext>()
      .AddDefaultTokenProviders();

    //indentiy service options
    services.Configure<IdentityOptions>(options =>...);

      //configure session
      services.AddMemoryCache();
      services.AddSession(options => {
              options.IdleTimeout = TimeSpan.FromDays(1);
              options.CookieName = ".app_session";
      });
    services.AddMvc();
```

Sample 14- .Fragment of Configure Services, /src/Startup.cs

The dependency on the UserManager, a class from within the ASP.Net Identity module, was easily mocked due to its design with "virtual" methods. This class was designed by Microsoft themselves to work within these design patterns and thus does not break the Dependency Inversion Principle. Consequently, it is within good practice to leave it as it is and not extract it as an interface (26). Lastly, it can be seen that the other two constructor parameters (ILoggerFactory, IHostingEnvironment) are already interfaces, as recommended.

Automated Unit Testing

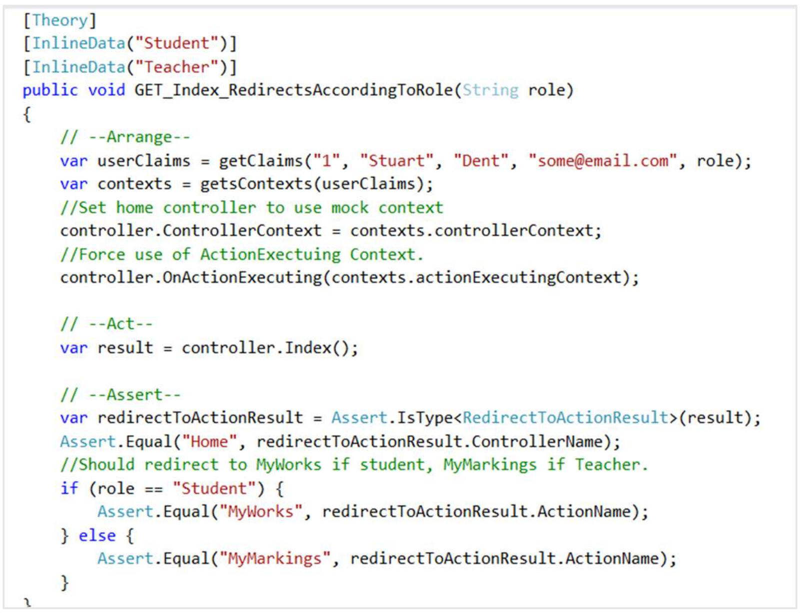
XUnit was chosen to test the model and business logic, as it can be used both with and without Visual Studio. (25) Using DI left the code made it possible to leverage the power of mock objects for automated unit tests and, for this purpose, we used the Moq mocking framework.



```
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_MyWorks_ReturnsMyworkViewWithRightModel_ValidationNotPassed...   19 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_MyWorks_ReturnsMyworkViewWithRightModel_ValidationNotPassed...   14 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_WorkDelete_ToMyWorks_AndDeletesFileForTheUserOwner(testAttempt:... 19 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_WorkDelete_ToMyWorks_AndDeletesFileForTheUserOwner(testAttempt:... 34 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_WorkView_ForMarkerUpdatesWorkWithFeedbackAndMark(newMark:...     400 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_WorkView_ForMarkerUpdatesWorkWithFeedbackAndMark(newMark: -6)   10 ms
OnlineMarkerCW.UnitTests.Controllers.HomeController_UnitTests.POST_WorkView_ForMarkerUpdatesWorkWithFeedbackAndMark(newMark: 60)   12 ms
OnlineMarkerCW.UnitTests.Filters.Filters_UnitTests.AnonymousOnly_ActionFilterContinuesExecution_WhenUserNotLoggedIn              321 ms
OnlineMarkerCW.UnitTests.Filters.Filters_UnitTests.AnonymousOnly_ActionFilterRedirectsToHomeIndex_WhenUserIsLoggedIn             290 ms
OnlineMarkerCW.UnitTests.Services.ServicesController_UnitTests.AddWork_Adds_a_Work_via_context                                     4 ms

Summary
Last Test Run Passed
41 Tests Passed
```

Figure 9 - A sample test run in Visual Studio

Each test is divided as per conventions in three sections: arrange, act and assert. The first section is the one which creates and configures the objects and mocks, readying them for the short "act" section, in which the behaviour to be tested happens. Finally, XUnit's asserts are used to determine if the resulting state after "act" is correct by comparing it with known expected values.

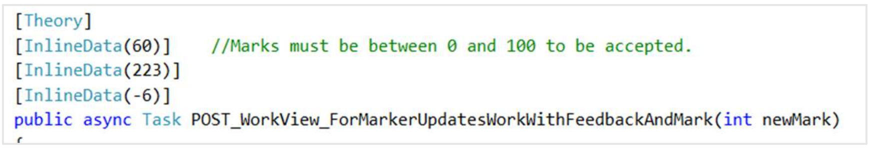


```
[Theory]
[InlineData("Student")]
[InlineData("Teacher")]
public void GET_Index_RedirectsAccordingToRole(String role)
{
    // --Arrange--
    var userClaims = getClaims("1", "Stuart", "Dent", "some@email.com", role);
    var contexts = getsContexts(userClaims);
    //Set home controller to use mock context
    controller.ControllerContext = contexts.controllerContext;
    //Force use of ActionExectuing Context.
    controller.OnActionExecuting(contexts.actionExecutingContext);

    // --Act--
    var result = controller.Index();

    // --Assert--
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    //Should redirect to MyWorks if student, MyMarkings if Teacher.
    if (role == "Student") {
        Assert.Equal("MyWorks", redirectToActionResult.ActionName);
    } else {
        Assert.Equal("MyMarkings", redirectToActionResult.ActionName);
    }
```

Sample 15- Example action controller test, in /test/HomeControllerTests.cs

A majority of the tests are for controller behaviour, and included asserting that the right views were being returned (for example when redirecting the user according to their role or lack thereof), asserting the content within these was correct and asserting that the model state because invalid whenever appropriate.

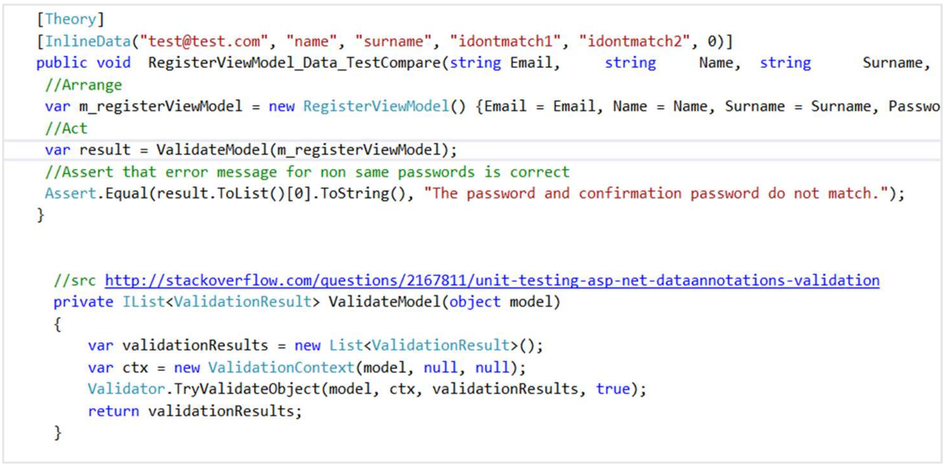
A variety of scenarios were tested within each testing method with the aid of XUnit's annotations, which make it simple to run a test method multiple times with different input.



```
[Theory]
[InlineData(60)]    //Marks must be between 0 and 100 to be accepted.
[InlineData(223)]
[InlineData(-6)]
public async Task POST_WorkView_ForMarkerUpdatesWorkWithFeedbackAndMark(int newMark)
```

Sample 16 - Xunit Annotations, in /test/HomeControllerTests.cs

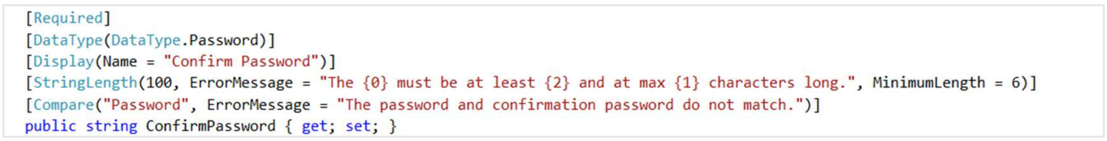
DbServices, the service class created for DI, was also tested by mocking the DbSet class (which acts as the "unit of work"). Within these tests, the "verify" method was used for those cases which did not test output, such as the tests of adding or updating "works" in the database. This mock object method reports whether or not a given method was called by the code being tested.



```
[Theory]
[InlineData("test@test.com", "name", "surname", "idontmatch1", "idontmatch2", 0)]
public void  RegisterViewModel_Data_TestCompare(string Email,     string     Name,  string     Surname,
 //Arrange
 var m_registerViewModel = new RegisterViewModel() {Email = Email, Name = Name, Surname = Surname, Passwo
 //Act
 var result = ValidateModel(m_registerViewModel);
 //Assert that error message for non same passwords is correct
 Assert.Equal(result.ToList()[0].ToString(), "The password and confirmation password do not match.");
}


  //src http://stackoverflow.com/questions/2167811/unit-testing-asp-net-dataannotations-validation
  private IList<ValidationResult> ValidateModel(object model)
  {
      var validationResults = new List<ValidationResult>();
      var ctx = new ValidationContext(model, null, null);
      Validator.TryValidateObject(model, ctx, validationResults, true);
      return validationResults;
  }
```

Input validation was performed through annotated view-model classes. Model binding validation happens outside and before the controller method call, and so this validation had to be tested separately and using the ValidationContext (which uses the ViewModel instance), as can be seen in ViewModelTest.cs.

Sample 17- code under /test/ViewModelTest.cs



```
[Required]
[DataType(DataType.Password)]
[Display(Name = "Confirm Password")]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]
[Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
public string ConfirmPassword { get; set; }
```

Sample 18 - RegisterViewModel class snippet, /src/ViewModels/AccountViewModels.cs

Additionally, our custom controller method filter (namely "AnonymousOnly") was tested in CustomFiltersTest.cs.

ASP.NET MVC core Evaluation and Conclusions

Features Comparison with python based Django framework

Both Django and ASP.NET MVC core follow the MVC development pattern, even though a View in Django is more of a call-back function, which is invoked by the router, when a certain route is accessed. Hence in a context of ASP.NET MVC it represents the functionality of a controller. As result Django can be considered a “MTV” framework, which stands for “model”, “template” and “view” (16).

Compared to Django, ASP.NET MVC core exhibits somewhat less sophisticated model definition capabilities. Django exercises use of predefined fields for the model definition (17), which can be described as instances of smaller models used as properties of a Model in ASP.NET. For example, in contrast with ASP.NET there exists a predefined File/Image field, which will automatically map the file path, image dimension to the model and will take care of the file streams, which simplifies the development greatly.

Django comes with out of a box ORM, whereas ASP.NET MVC uses the Entity ORM Framework. Both ORMs where designed to work with SQL databases only, hence alternative have to used to work with noSQL (12) (16). Both ORMs support use of their respective OOP style C# and Python like database expressions for modifying and querying the database, as well as SQL like ones native querying capabilities. For it .NET comes with the Language Integrated Query component, but Django with the RawSQL module (18) (19).

One aspect where ASP.NET MVC core excels is the user management. ASP.NET MVC core uses the Identity system, which offers more capabilities such as Roles and Claims based authentication, built in third party login and OAuth 2 support (10). On the other hand, Django comes with an automatically generated administration web interface, which extremely simplifies the management and overview of users (20).

Ease of Use, Efficiency and Productivity

ASP.NET MVC core is built on top of c# sharp language, which is a compiled, OOP proprietary language developed by Microsoft for the .NET stack (21). On the contrary, Django runs on top Python (16), which is an interpreted, open source, OOP language. Due to its straightforward syntax, lose typing and non compiled nature, Python is more intuitive and easy to use, as result Django applications are faster to write, to get up running, to manually validate and more importantly easier to learn to code. At the same time the less restrictive development approach opens room to creating more bugs and less robust web applications. In (22) Kurt Grandis produces a case study of his two development teams using the both frameworks and concluding that the team using Django was as twice as productive than one using ASP.NET. This closely coincides with the sentiment and experiences of authors of this report.

Moreover ASP.NET and Entity core are rather new technologies and suffer from lack of a proper documentation and developer’s community. The full API reference for both was published only in end of October, early November, hence the development speed of the application suffered in the early stages from trying to guess how to implement certain features. Moreover couple of bugs present in Entity core prevented the model testing to be sufficiently implemented. In comparison Django is mature and has got excellent documentation and hefty community.

Cross Platform Development

Both frameworks support cross platform development and deployment. They both can be deployed on Linux-based and IIS servers, as both frameworks simply produce web applications which just require the language and libraries compatibility with the relevant system, although it is uncommon to deploy Django application on IIS (23) and the ASP.NET hosting framework became separated from IIS only under the core release (24).

This project was developed using Windows and Linux systems hand in hand. As results authors run into couple of issues which had influenced the design decisions. The biggest issue was lack of the MS SQL server support for the Linux systems, which lead to use of the rather limited SQLlite database. The Visual Studio IDE, upon which ASP.NET MVC heavily relies, is not available for the Linux systems. That leaves Linux users to purely using CLI and a text editor for development, which may seem to be unappealing for many developers, although Visual Studio like text editor Visual Studio Code is available for Linux. Due to lack of Visual Studio, .NET package management is somewhat frail, due to lack of good a overview outside of Visual Studio of the NuGet packages available and their compatibility with already installed components. In summary, regardless of all the drawbacks, the cross platform development is not impossible and leaves promising impression.

Bibliography

1. **Fritz, Jeffrey T.** ASP.NET 5 is dead – Introducing ASP.NET Core 1.0 and .NET Core 1.0. [Online] Microsoft, 19 January 2016. [Cited: 2 December 2016.] https://blogs.msdn.microsoft.com/webdev/2016/01/19/asp-net-5-is-dead-introducing-asp-net-core-1-0-and-net-core-1-0/.

2. **Phillip Carter, Zlatko Knezevic.** .NET Core - .NET Goes Cross-Platform with .NET Core. [Online] Microsoft, April 2016. [Cited: 2 December 2016.] https://msdn.microsoft.com/magazine/mt694084.

3. Inversion of Control. [Online] Microsoft. [Cited: 2 December 2015.] https://msdn.microsoft.com/en-us/library/ff921152.aspx.

4. Dependency Injection. [Online] Microsoft. [Cited: 2 December 2016.] https://msdn.microsoft.com/en-us/library/ff921152.aspx.

5. **Steve Smith, Andy Pasic.** Dependency Injection. [Online] Microsoft, 14 October 2016. [Cited: 2 December 2016.] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

6. —. Overview of ASP.NET Core MVC. [Online] 14 October 2016. [Cited: 2 December 2016.] https://docs.microsoft.com/en-us/aspnet/core/mvc/overview.

7. **Sukesh, Marla.** WebForms vs. MVC. [Online] 26 Sep 2014. [Cited: 2 December 2016.] https://www.codeproject.com/articles/528117/webforms-vs-mvc.

8. **Steve Smith, Rick Anderson.** Controllers, Actions, and Action Results. [Online] Microsoft, 14 10 2016. [Cited: 2 12 2016.] https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions.

9. **Anderson, Rick.** XSRF/CSRF Prevention in ASP.NET MVC and Web Pages. [Online] Microsoft, 14 March 2013. [Cited: 2 December 2016.] https://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages.

10. Introduction to Identity. [Online] Microsoft, 14 October 2016. [Cited: 2 December 2016.] https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity.

11. **rawel.** choose between asp identity claims and sessions data. *Stack OwerFlow.* [Online] 8 Feb 2016. [Cited: 2 December 2016.] http://stackoverflow.com/questions/35231107/choose-between-asp-identity-claims-and-sessions-data.

12. **Miller, Rowan.** Entity Framework Core. [Online] 27 10 2016. [Cited: 2 December 2016.] https://docs.microsoft.com/en-us/ef/core/index.

13. **Taylor Mullen, Rick Anderson.** Razor Syntax Reference. [Online] Microsoft, 14 10 2016. [Cited: 2016 December 2016.] https://docs.microsoft.com/en-gb/aspnet/core/mvc/views/razor.

14. **Anderson, Rick.** What are Tag Helpers? [Online] Microsoft, 14 10 2016. [Cited: 2 December 2016.] https://docs.microsoft.com/en-gb/aspnet/core/mvc/views/tag-helpers/intro.

15. **Appel, Rachel.** Model Validation. [Online] Microsoft, 2016 10 14. [Cited: 5 December 2016.] https://docs.microsoft.com/en-gb/aspnet/core/mvc/models/validation.

16. FAQ: General¶. *django.* [Online] [Cited: 5 December 2016.] https://docs.djangoproject.com/en/1.10/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names.

17. Model field reference¶. [Online] django. [Cited: 5 December 2016.] https://docs.djangoproject.com/en/1.10/ref/models/fields/.

18. Query Expressions¶. [Online] django. [Cited: 5 September 2016.] https://docs.djangoproject.com/en/1.10/ref/models/expressions/#django.db.models.expressions.RawSQL.

19. **Don Box, Anders Hejlsberg.** LINQ: .NET Language-Integrated Query. [Online] Microsoft, February 2007. [Cited: 5 December 2016.] https://msdn.microsoft.com/en-us/library/bb308959.aspx.

20. The Django admin site. [Online] django. https://docs.djangoproject.com/en/1.10/ref/contrib/admin/.

21. **Hasan, Noorul.** History of C# Programming. [Online] 27 09 2012. [Cited: 05 December 2016.] http://aboutcsharpprogramming.blogspot.co.uk/2012/09/history-of-c-programming.html.

22. **Grandis, Kurt.** Python + Django vs. C# + ASP.NET: Productivity Showdown. [Online] 24 02 2010. [Cited: 5 December 2016.] http://kurtgrandis.com/blog/2010/02/24/python-django-vs-c-asp-net-productivity-showdown/.

23. **VORAS, IVAN.** Installing Django on IIS: A Step-by-Step Tutorial. [Online] TOPAL. [Cited: 5 December 2016.] https://www.toptal.com/django/installing-django-on-iis-a-step-by-step-tutorial.

24. **Strahl, Rick.** ASP.NET vNext: The Next Generation. [Online] [Cited: 5 December 2016.] http://www.codemag.com/article/1501061.

25. **Smith, Steve.** Testing Controller Logic. [Online] Microsoft, 14 10 2016. [Cited: 5 December 2016.] https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing.

26. Explicit Dependencies Principle. [Online] [Cited: 5 December 2016.] http://deviq.com/explicit-dependencies-principle.

27. CanIUse. [Online] [Cited: 5 December 2016.] http://caniuse.com/.