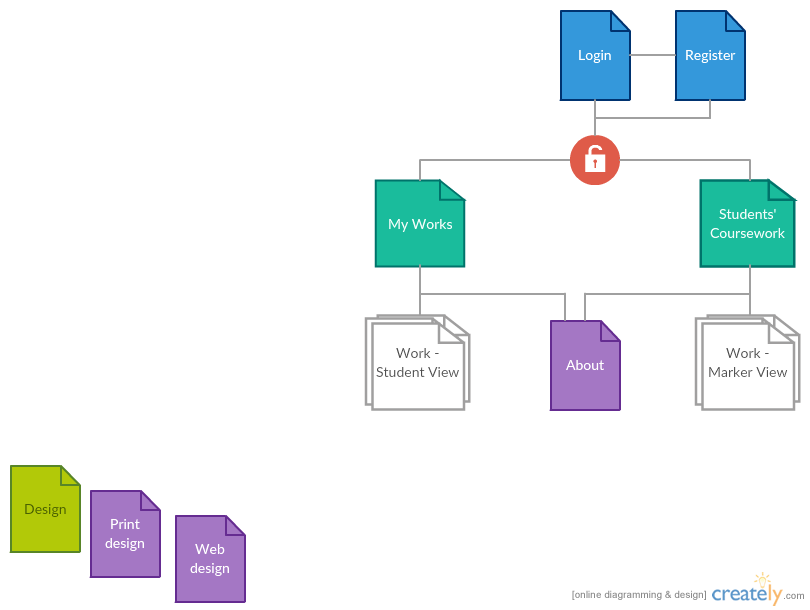
ASP.NET MVC Prototype Web Site

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

# Required features, Technologies and Techniques

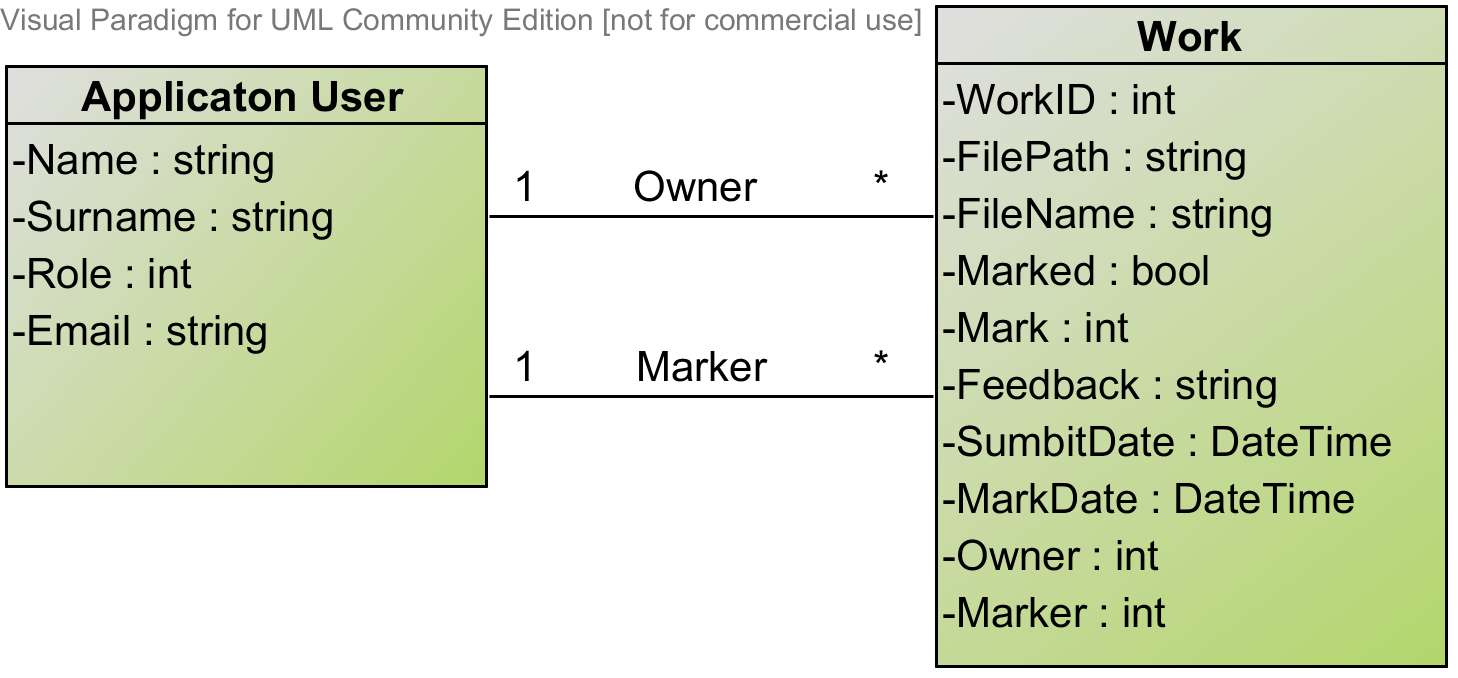
## Application Structure

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

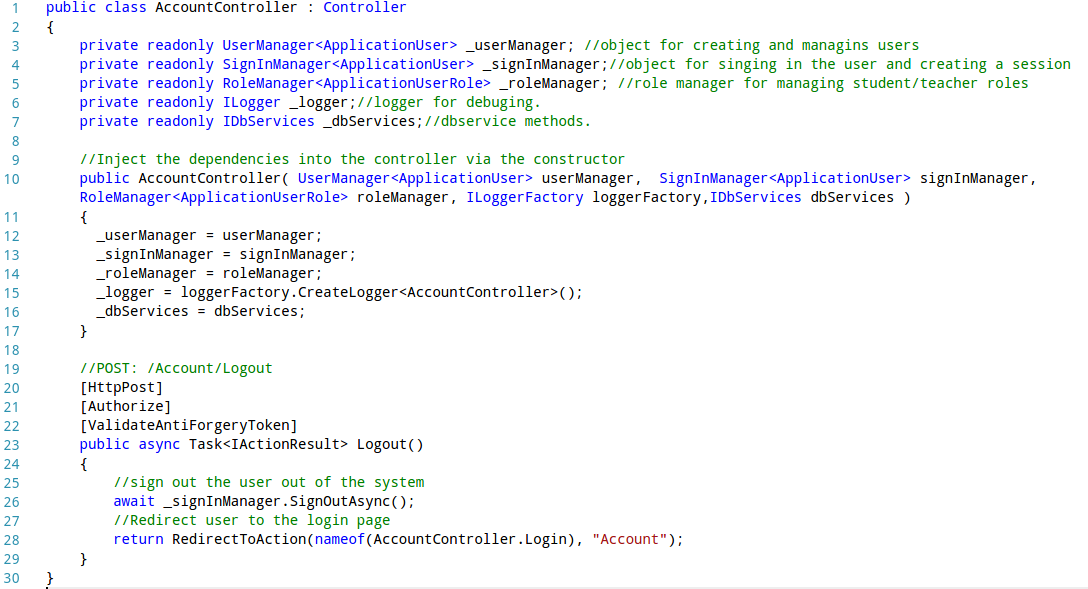
**Figure 2 –** Data structure diagram

## Technologies and Techniques Used

### ASP.NET MVC core

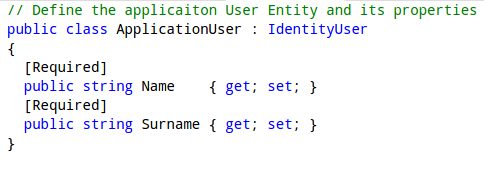
ASP.NET MVC core 1.0, used in this project, is a freshly franchised version of the ASP.NET (1). The biggest differences from the previous released include moving away from the proprietary development and making the platform open source and cross platform. The key idea behind the redesign of the platform 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 inh*e*rit 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*.

**Sample 1 –** Definition an Account controller.

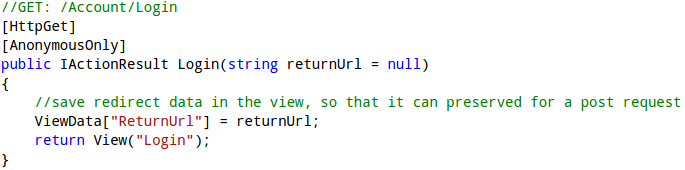
frameworks services. The annotations defined in the square brackets before the *Logout() action task*,indicate the extra filters that request has to go through before reaching the request handler. This action sing outs and redirects the user to the login page.

Data model entities within the framework are simply defined a class as shown in **Sample 3**. The properties of the entity are created as the properties within that class. These can include Data Annotations defined in square bracket.



**Sample 3 –** Definition of the Application User entity. Available under /src*/models/ApplicationUsers.cs.*

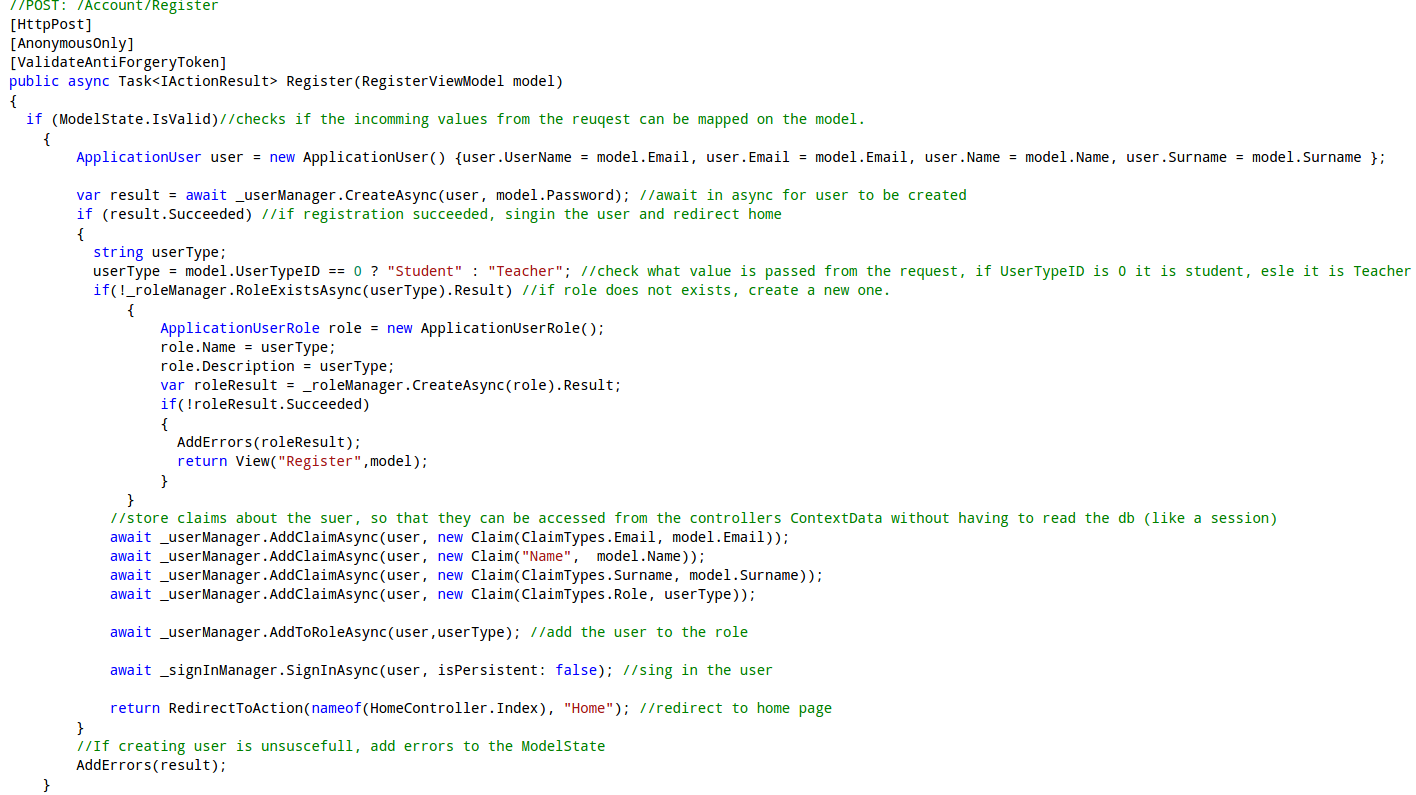
The *ViewResult* object is generated when an action handler returns a *View()* method as a result. The *ViewResult* object typically contains a model data that has been queried and *ViewData* that has been generated by the action handler to be passed to the Razor templating engine.



**Sample 4–**Login handler which returns a *View()* method which generated the *ViewResult* object. Available under /*src/Controllers/HomeControllers.cs.*

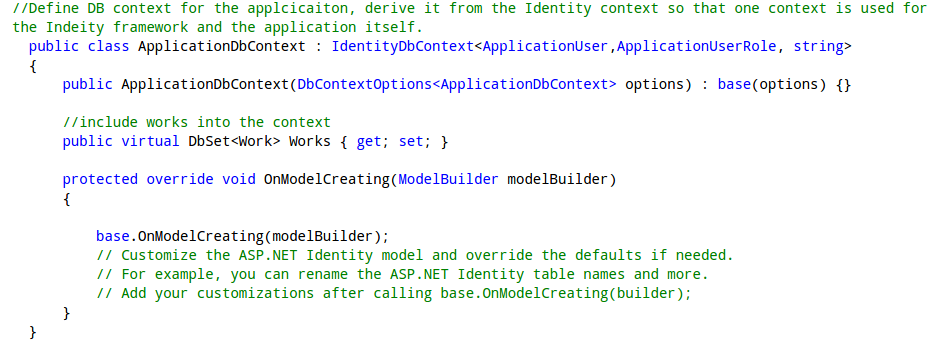
## Authentication and Authorisation

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 5.** Another design decision is demonstrated in this code snippet, which includes 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.

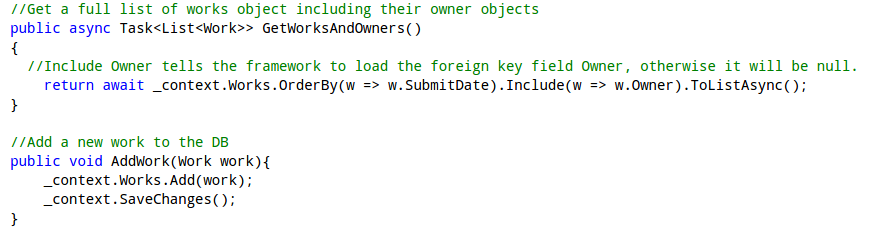
**Sample 5 *–*** Registration action handler, Available under /*src/Controllers/AccountControllers.cs.*

## Entity Framework Core

The core – lightweight and cross platform version of the Entity Framework was used as an Ojbect-Relational mapper to map the .NET model objects to the SQLlite database (12). SQLlite was used to ensure the cross platform development, as the MS SQLServer is not compatible with the Linux systems. Entity Framework uses a predefined DB context (**Sample 6**) to establish a connection and query or modify the database.

**Sample 6** – Definition of a database context. Available under /*src/Data/* *ApplicationDbContext.cs.*

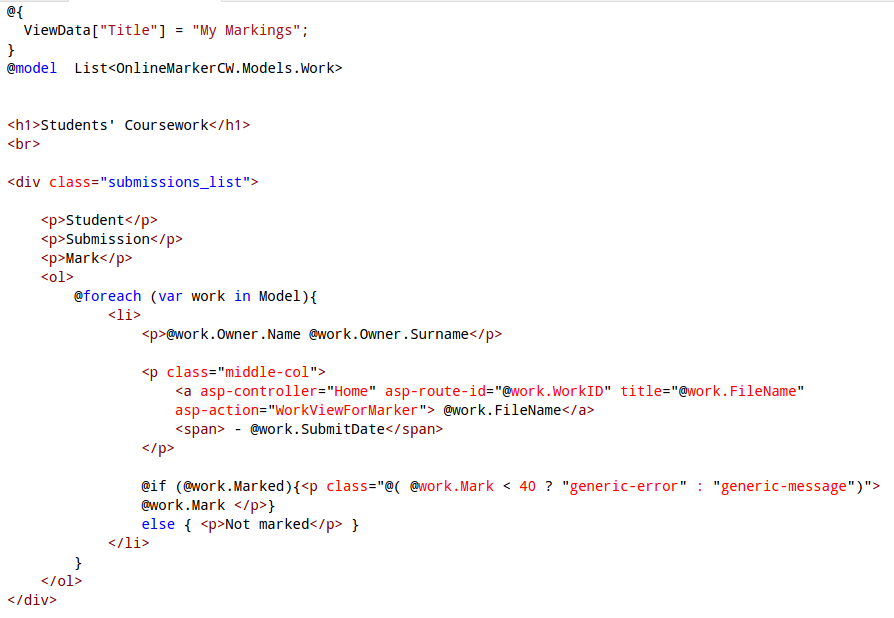
Modifications and Queries are performed via the context on the selected entities, as shown in the **Sample 7** for the Work entity. The ORM mechanism reads the DB and returns a usable .NET object which can be passed to the controller to perform necessary logic operations.



**Sample 7** – 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). It renders a default layout with a body content which corresponds to the view returned by the controlled, which is inserted into the layout via the *@RenderBody()* method call*.* **Sample 8** shows the view that will be rendered for the H*omeController/MyMarkings* route.



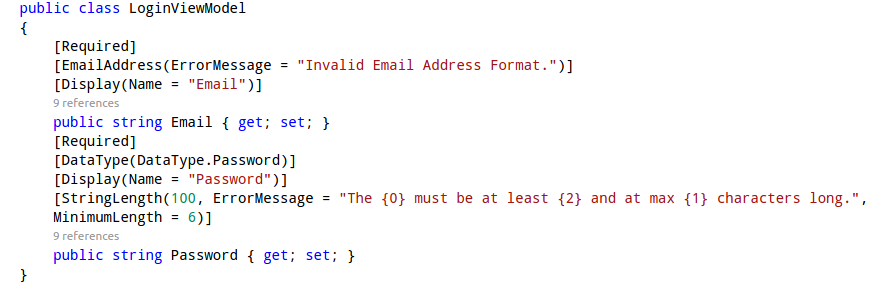
**Sample 8 –** My Markings view, available under */src/Views/Home/MyMarkings.cshtml*

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 10** 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 9** 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).

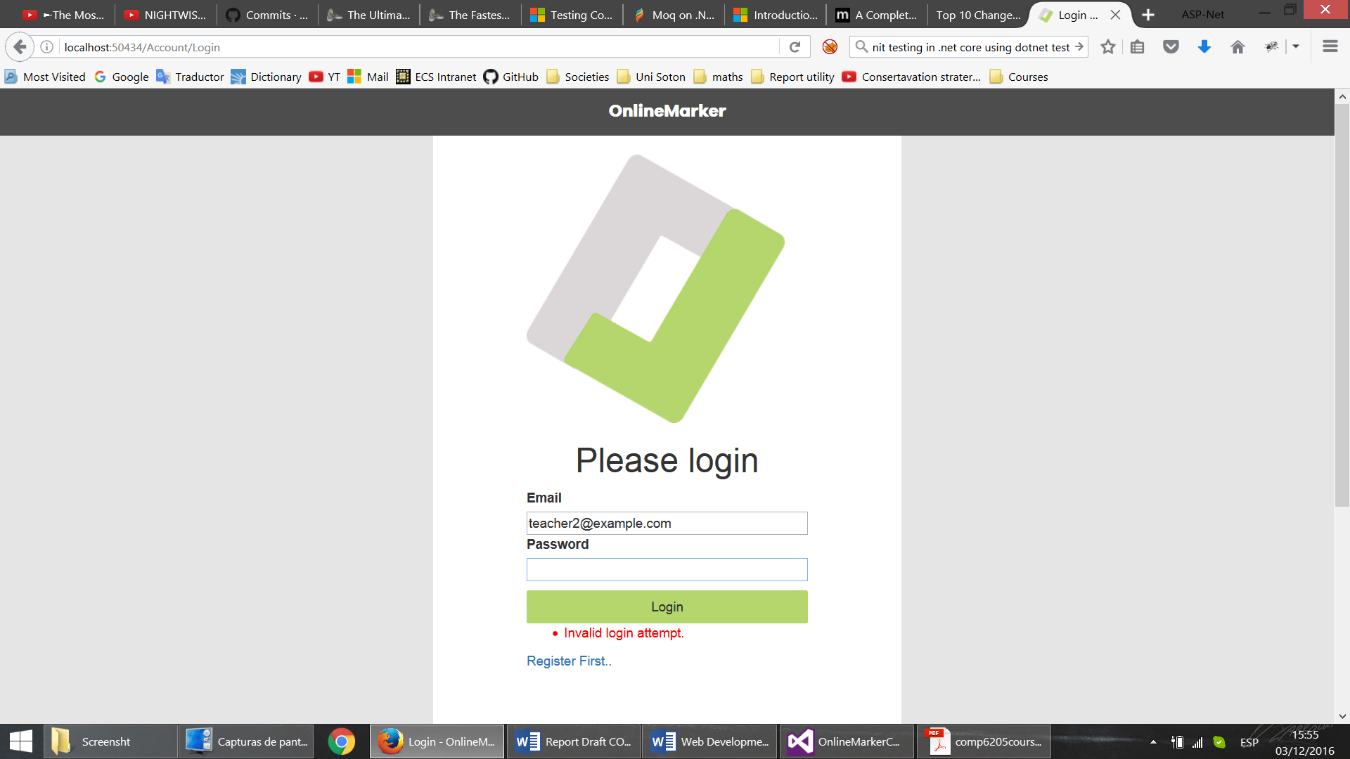


**Sample 9** – Login ViewModel, available under */src/ViewModels/AccountViewModels.cs*

Client side validation attributes such as *min, max length and required* are not generated by the tag helpers hence have to be defined manually as seen in the sample below.

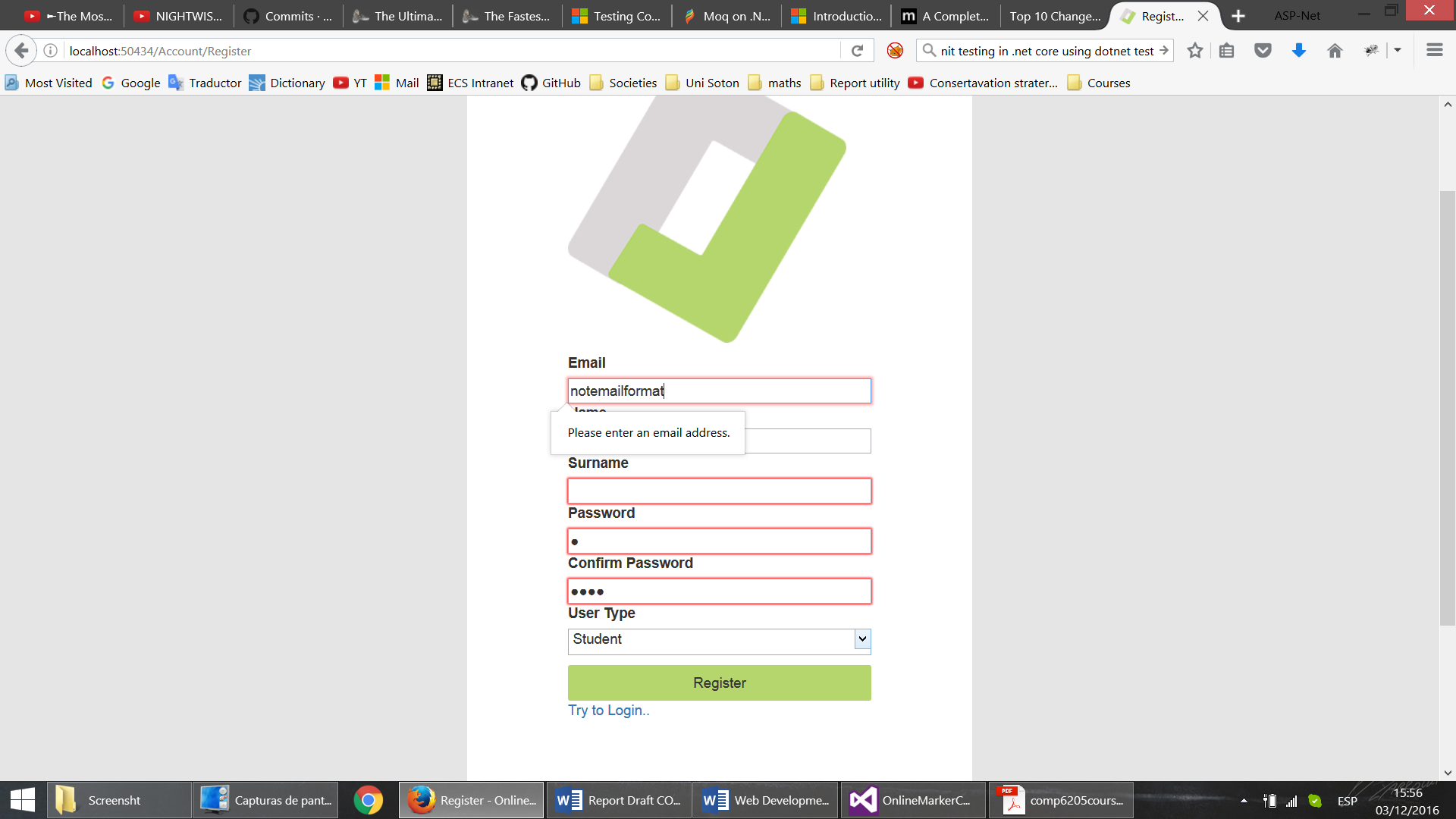
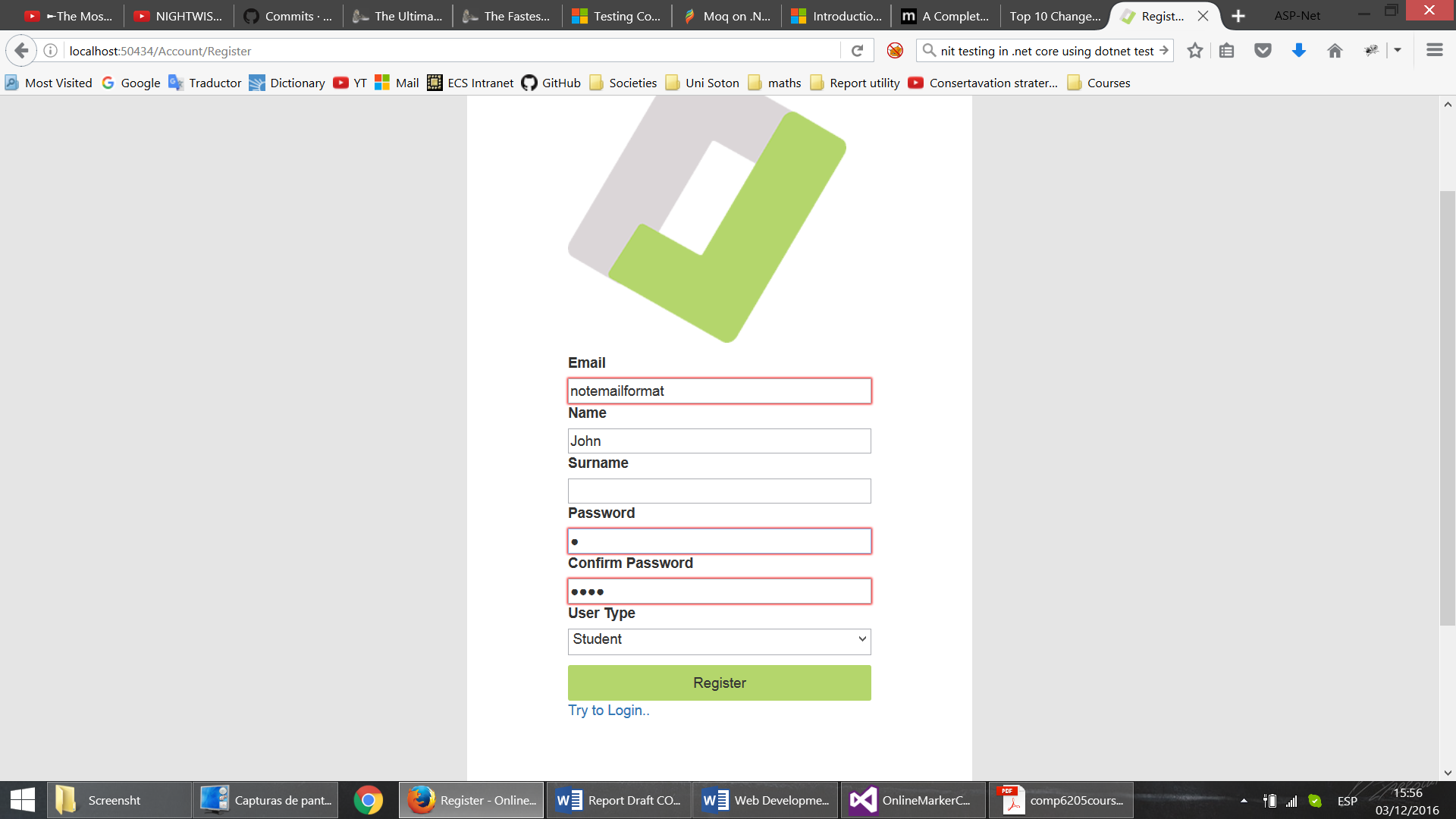
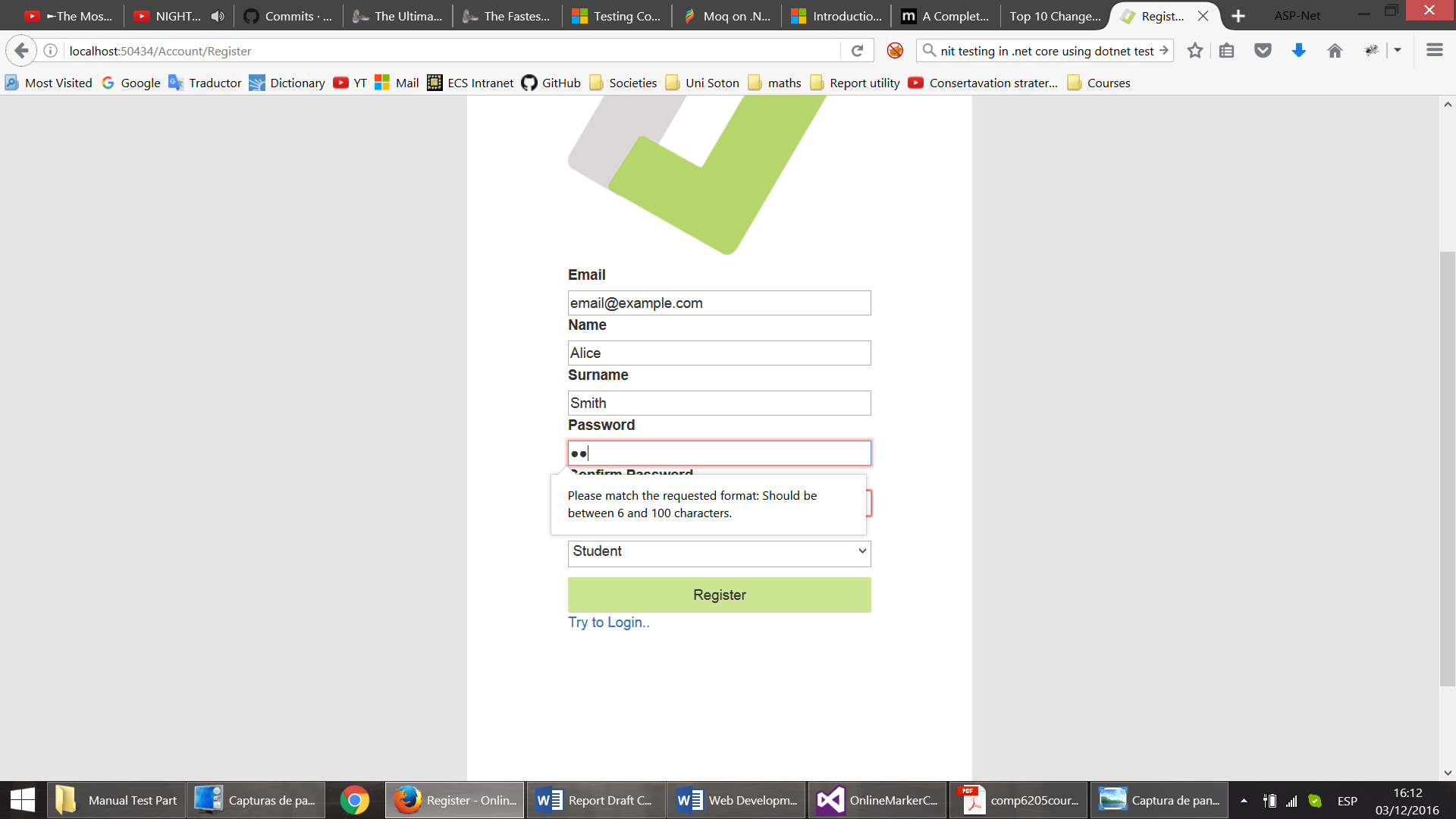
**Sample 10** – Login page View, available under */src/Views/Account/Login.cshtml*

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.



8. Login Screen, Error Reporting and Input Preservation

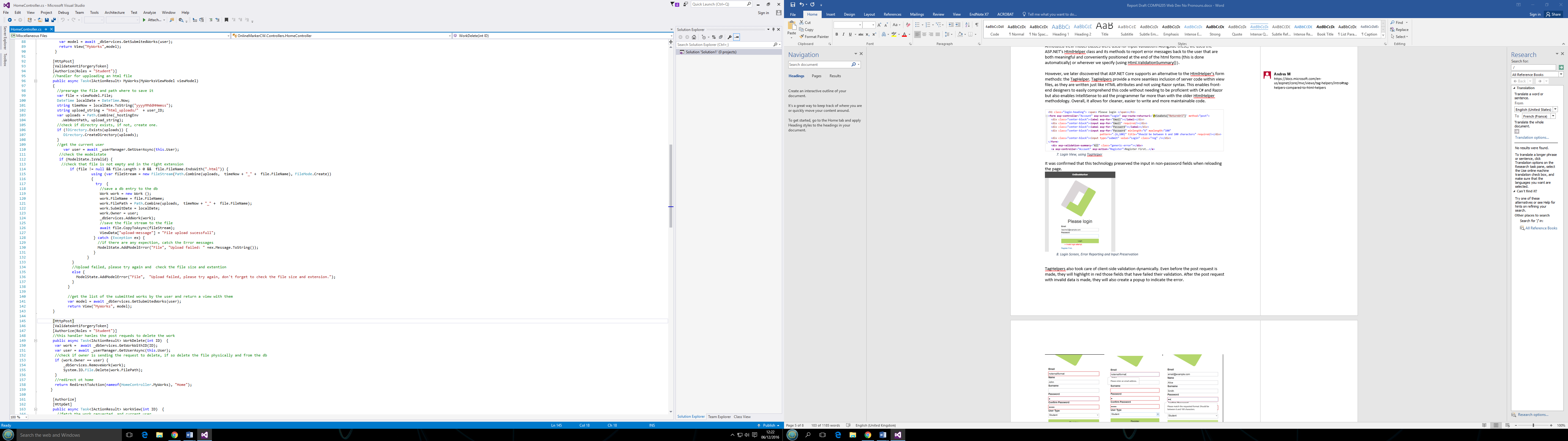
TagHelpers also took care of client-side validation dynamically. Even before the post request is made, they will highlight in red those fields that have failed their validation. After the post request with invalid data is made, they will also create a popup to indicate the error.

  fghfg

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

9. Error reporting after invalid mark (assuming client-side validation was circumvented)

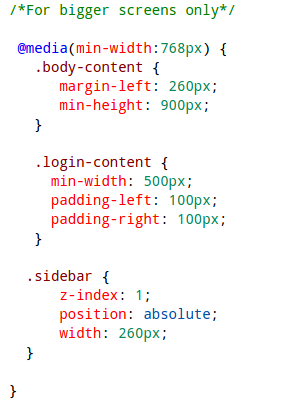
Addition server side validation was required for the file upload functionality, that was included in the controller action for testing file size and extension (Figure ??)

]

Fgireu

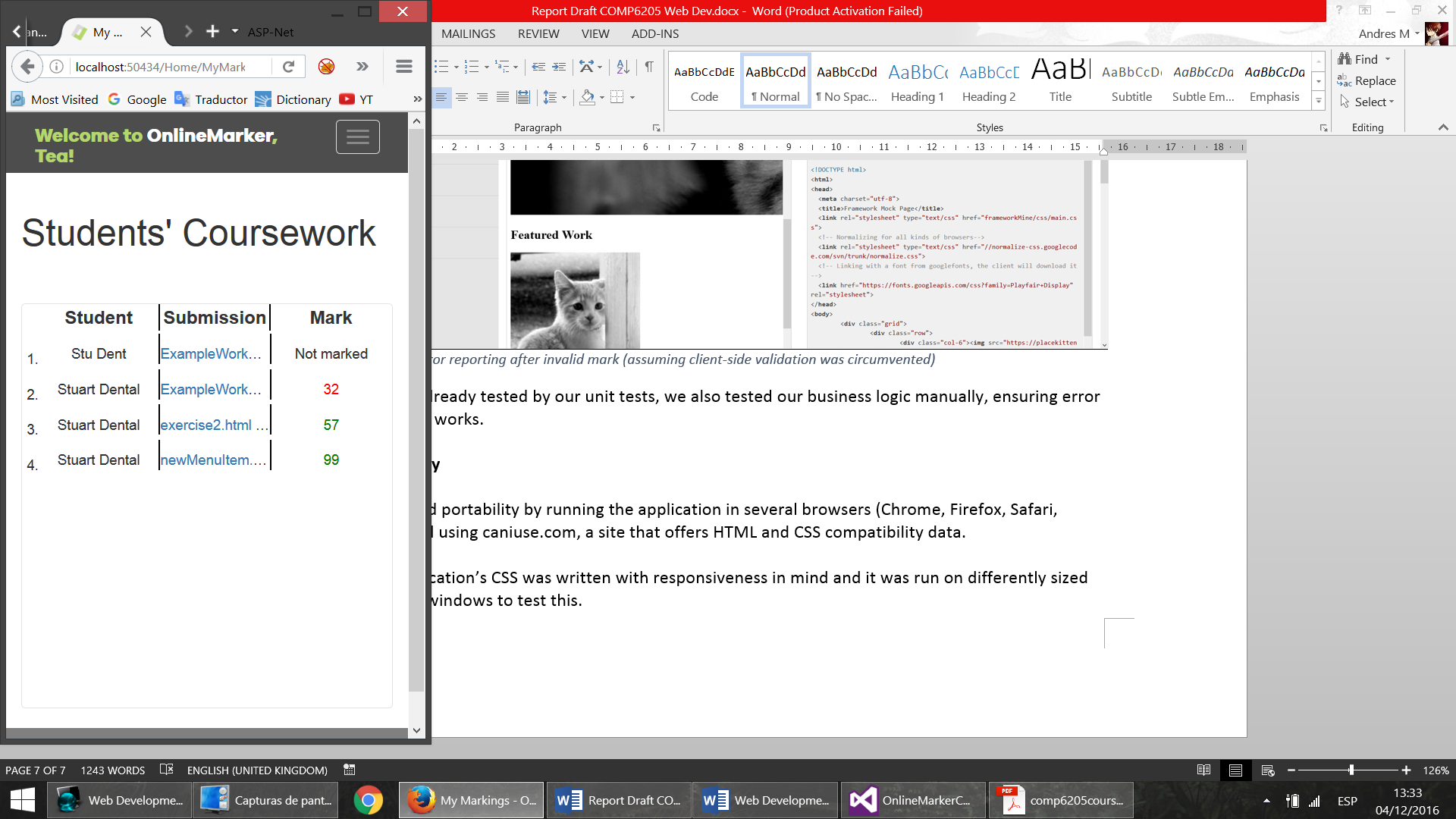
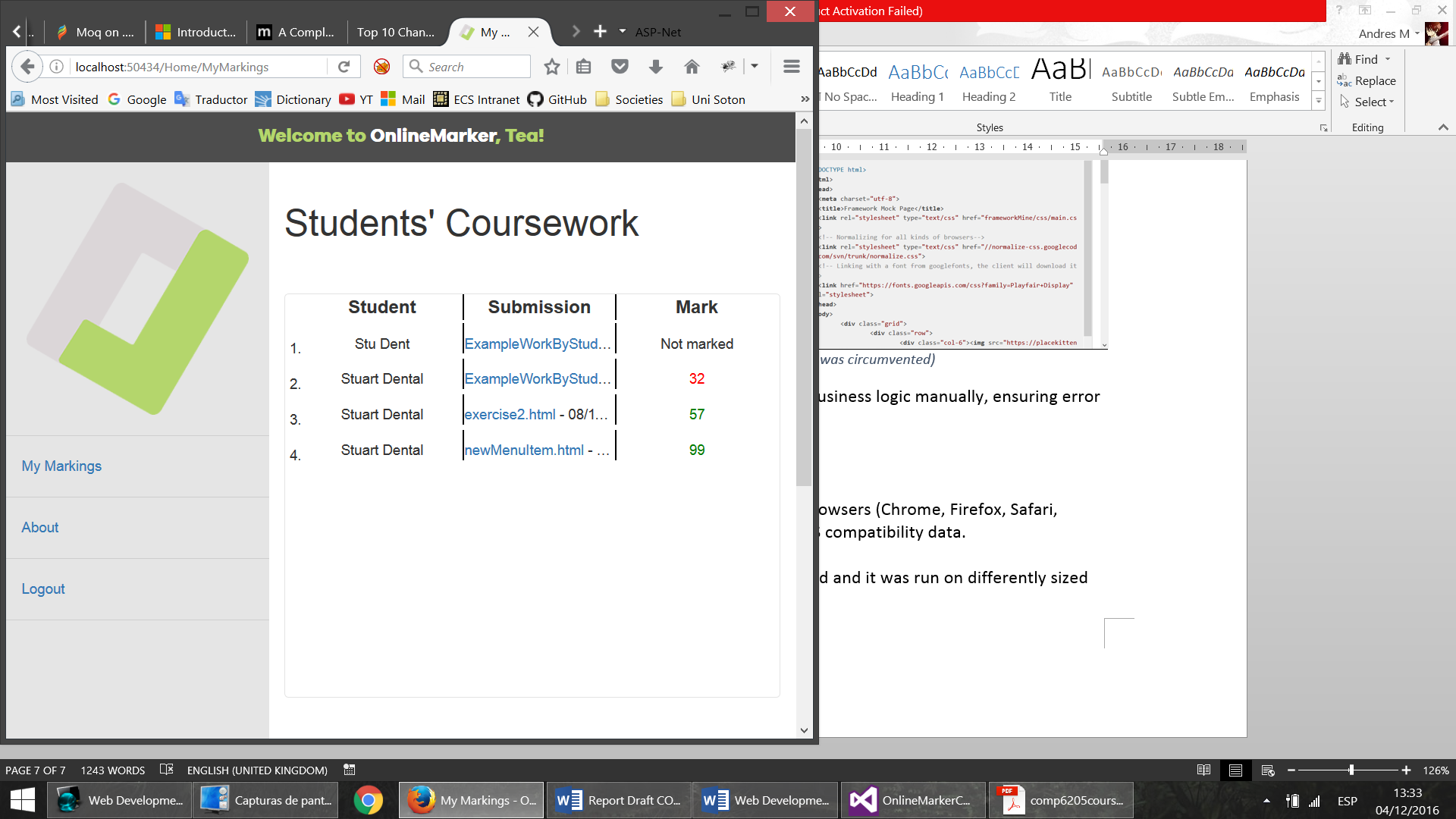
## Professional Layout and Responsiveness, portability testing

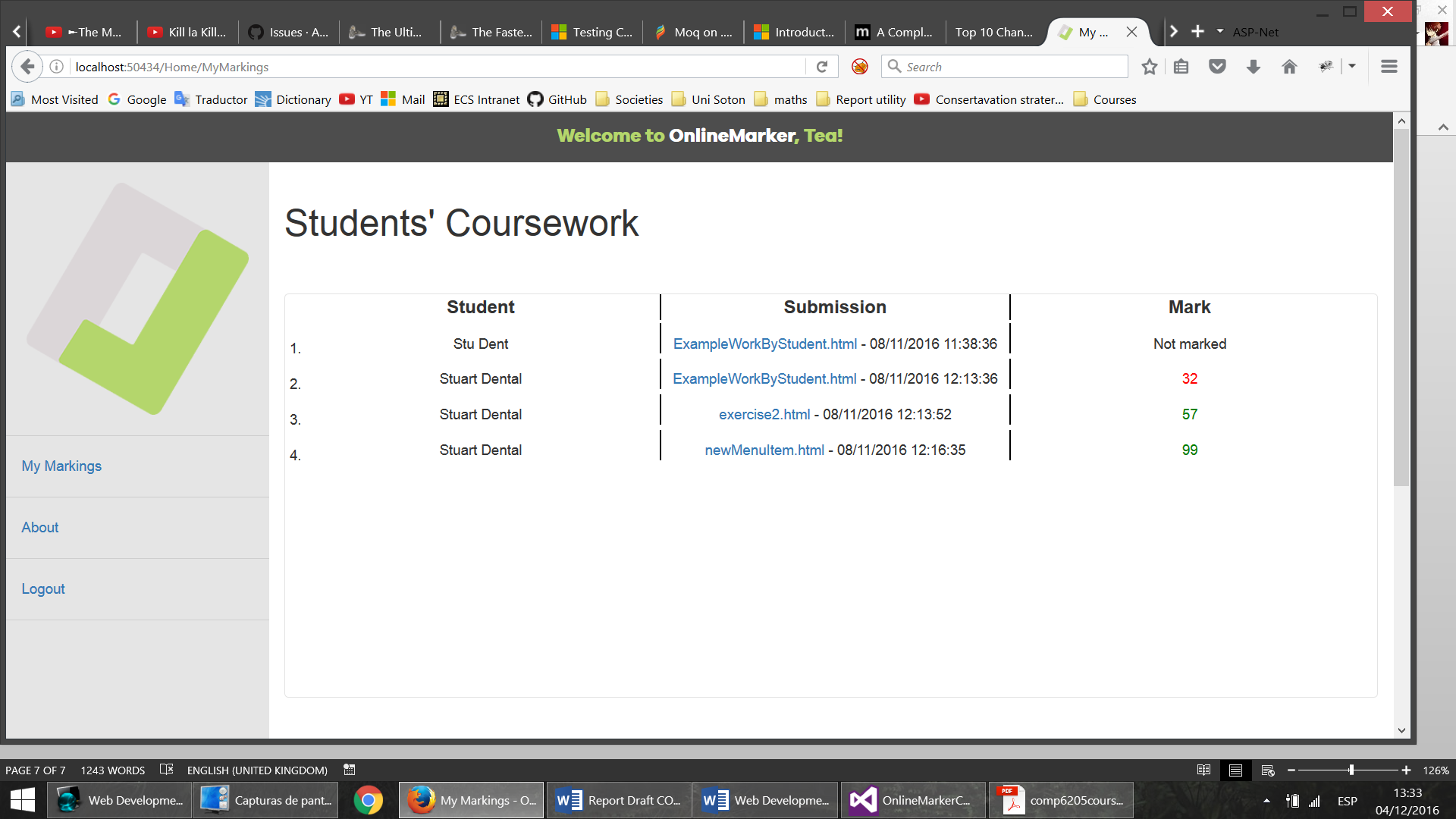
The professional layout, and responsiveness of the application was achieved though combination of the CSS3 techniques and the Bootstrap Framework. The CSS3 *@media* rule was used extensively in order to achieve scalable design. For instance, the code sample below shows that only for screen sizes of width of 768px over the sidebar should have certain width, login container should have bigger padding and body content should have bigger height.



**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, 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.



**Figure 10. Responsiveness/Device Portability Showcase**