ASP.NET MVC

# Purpose of this module

This module will introduce the ASP.NET MVC framework. ASP.NET MVC gives developers a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and that offers full control over markup for enjoyable, agile development. After this module, the student should:

* understand how the ASP.NET MVC framework works
* know how to build a basic website using the MVC framework

# Module Requirements

You must have completed the Advanced C# module before you can start working on this module.

# Homework BEFORE the exercise session

This assignment guides you through most of the work you have to do, but it is wise to look up some information about ASP.NET MVC beforehand. In particular, we will be using the Razor view engine to build our webpages. It is **highly** recommended that you get to know the basics of this before you start the exercise session. A good place to start is [Scott Guthrie’s blog](http://weblogs.asp.net/scottgu/introducing-razor) or the Razor book that is included with this module.

# Exercises

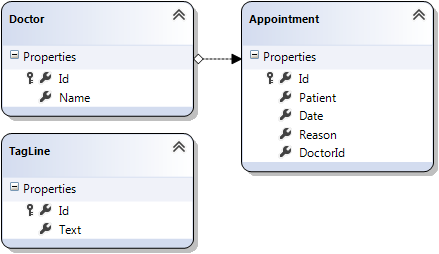
In this assignment, we are going to implement the website of a doctor’s office using the MVC 5 framework. (If you have completed the ASP.Net Webforms module, you will already be familiar with this website. This module creates the same website using a different framework.) The ASP.NET MVC framework is a powerful tool to build clean websites according to the Model-View-Controller paradigm. Models are *dumb* classes that only store data (also called *data access objects*). Controllers are classes where the business logic of the application resides. They process user interactions and change or retrieve the corresponding data using the model classes. Views are ASP.NET pages that visualize a given model.

This assignment describes what is expected of you and gives some pointers to extra information and an occasional helpful tip. For the most part, however, you are expected to look up information online if you’re stuck.

## Preliminaries

The MVC framework offers lots of control to programmers, but consequently also requires more effort to create a decent-looking website. In this exercise session, we will use the Bootstrap[[1]](#footnote-1) package to do some of the heavy lifting in the user interface. In addition to Bootstrap, we will also use jQuery[[2]](#footnote-2) in the project. The MVC framework uses jQuery for a variety of things, including AJAX calls and client-side validation.

Because this is not a module on how to access a database, you do not have to write all the code that deals with accessing the database. Extract the solution in the file MVCModule.zip and open it. You will see that it consists of a project `MVCModule.Models’ that contains all the necessary code to access the database. There are three classes that represent individual rows in the database:



These items can be accessed by first creating a *data manager* object and then querying that object. To allow for easy unit testing, the MVCModule.Models library is built as a service-oriented architecture (and uses a service location feature). This means that in order to create an Appointment object, for example, you will need to create the appointments data manager as follows:

IAppointmentManager manager = ServiceLocator.Resolve<IAppointmentManager>();

manager.Create(new Appointment() { Patient = "Tuur", ... });

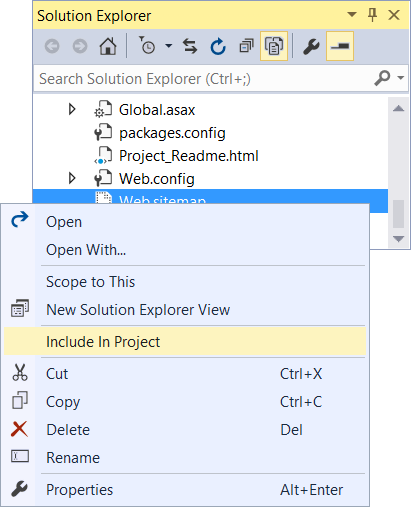
The Resolve method returns an object that implements the correct interface, but your code will never have a direct link to the actual class that implements this interface.

## Website creation

Add a new ASP.NET Web Application project to the solution and name it ‘MVCModule’. A new window will be shown where you can edit the project properties. Choose the MVC template and change the authentication type to ‘No authentication’.

Visual Studio will automatically generate a small website with three pages. Set the MVCModule project as the startup project and run the solution to verify that everything works. The website project contains a number of folders and some files. The folders of note are *Controllers*, *Models*, and *Views*. These folders will contain the classes that will respectively contain the business logic, the data layer, and the UI layer. There are also two Web.config files in the project. The file in the *Views* directory modifies some settings, and blocks access to the directory. This is necessary to make sure that the views can only be accessed through the controller (and not by surfing directly to them). This file will not be modified in this assignment. The Web.config in the root directory contains the project settings. When this document refers to ‘the Web.config’, it refers to the file in the root directory.

Copy the file *Web.sitemap* into the project root folder. After copying it to the right location, include it into the project. Do this by clicking the ‘Show all files’ icon, right-clicking the file you wish to include, and selecting the ‘Include In Project’ menu option.



Because we will use the classes in the MVCModule.Models project from the website project, add a reference to the Models project (right-click *References* and select *Add Reference*).

In order to learn the basics of MVC development, we will start from scratch. Delete the following files:

* Controllers\HomeController.cs
* Views\Home\About.cshtml
* Views\Home\Contact.cshtml
* Views\Home\Index.cshtml
* Views\Shared\\_Layout.cshtml
* Views\Shared\Error.cshtml
* Project\_Readme.html

Create the following directories:

* Views\Account
* Views\Doctor
* Views\Patient

We are now ready to start building our doctor’s practice website using the ASP.NET MVC framework.

## Models, Views and Controllers

The ASP.NET MVC framework is, as the name implies, an implementation of the MVC design pattern for ASP.NET. Programs code is broken into three different categories: models, views and controllers. Models are classes that contain data but do not offer advanced logic. They are typically very simple classes with many properties and potentially some constraints on these properties (e.g. “this property should be a string that contains a valid email address”, or “this integer should be in the range between 0 and 100”). Views are also very simple classes (or in the case of ASP.NET MVC: pages) that take a model object and use it to render a view (e.g. a page on a website). The controllers are the classes that contain the business logic of the application. They are responsible for filling the model classes and passing them to the correct views.

Separating code in this way greatly improves the maintainability of the application. The model layer can contain the mechanisms to read data from the database and to store the changes. If the database backend changes, the only part of the application you need to update is the way data model classes are retrieved from and stored to the database. Separating the views from the business logic also greatly increases the potential for code reuse. One controller action can return multiple views, and the same view can be reused from multiple controller actions.

We will start building our first MVC application and explain these concepts further as we move along.

## Layout pages

If you have completed the ASP.NET WebForms module, you will know that a web form (i.e. a page) can reference a master page in order to separate the common parts of the user interface from the specificities of different views. Master pages could reference other master pages (this technique is called ‘nesting master pages’) to reuse as much code as possible even in complex user interfaces.

In ASP.NET MVC, the distinction between the two types of pages is purely semantical and not technical. A master page is now called a layout page, whereas a web form is now called a view page, but technically they are the same thing. The only difference between a layout page and a view page is that a layout page calls the RenderBody method that renders the body of a child page.

Every (view or layout) page can reference another layout page as its parent. When a view is being rendered, the ASP.NET engine will render the view inside of the layout page that is referenced (essentially doing the same as an ASP.NET Web Form that references a Master page).

Right click on the Shared folder (in the Views folder), and click Add ⇢ MVC 5 Layout Page (Razor) and name it ‘\_Layout’. Open the page and modify it to your liking (in order to save time, you can use the style in the file *style.html* and add the CSS code in *extracss.txt* to the file */Content/Site.css*).

We can set up this layout page as the default layout for all view pages. When the MVC framework renders a view, it will look at the contents of the file Views\\_ViewStart.cshtml to determine which layout page is the default layout for all views. Open this file, and verify that it links the Layout property to the Views\Shared\\_Layout.cshtml page (that we just created).

## Razor

In ASP.NET Web Forms, every form had a separate code behind file where we could add the logic that is needed to render the view. This makes it easy (and thus also tempting) to place lots of business logic inside the view. This is of course a bad idea and you should always resist this temptation![[3]](#footnote-3)

In order to somewhat force developers to keep views simple, the ASP.NET MVC framework does not have code behind files. If a view requires some code to display the UI (e.g. render a list of items), that code must be included in the HTML of the view. Obviously, the code should be kept simple and as small as possible, because the HTML would otherwise quickly become unreadable.

Placing code in the HTML is also possible in an ASP.NET web form. Developers can use special delimiters to go from ‘HTML mode’ to ‘ASP.NET mode’ and back (much like PHP). For example, The following HTML contains a bit of C# code to create and fill a table.

<table>

<% foreach(StoreItem item in Model.Items) { %>

<tr>

<td><%= item.Name %></td>

<td>

<% if(item.Available) { %>

<p>Available!</p>

<% } else { %>

<p>Not available...</p>

<% } %>

</td>

</tr>

<% } %>

</table>

As you can see, the syntax is a bit cumbersome to work with. It is relatively bloated and we would like a deeper integration between the HTML and the C# code. Starting from MVC 3, Microsoft introduced a new way of embedding code in HTML views called *Razor*. MVC 3 and 4 supported both this new Razor syntax and the old ASPX syntax, but as of MVC 5 only Razor is supported. The example above looks like this in Razor:

<table>

@foreach(StoreItem item in Model.Items) {

<tr>

<td>@item.Name</td>

<td>

@if(item.Available) {

<p>Available!</p>

} else {

<p>Not available...</p>

}

</td>

</tr>

}

</table>

Razor makes it easy for us to switch from HTML mode to C# mode by simply using an @ symbol, and it automatically detects when we’re switching back to HTML mode. Furthermore, it understands the basic structure of C# commands, so it automatically recognizes the ‘else’ block for example as a continuation of the previous ‘if’ block. Take a moment to get to know the syntax of razor by studying the example above. It is advised to also quickly read a Razor tutorial (e.g. <http://weblogs.asp.net/scottgu/introducing-razor>).

## Bundles

HTML pages often link to one or multiple JavaScript and CSS files. The web browser requests each file individually, causing potential loading delays (web servers typically do not allow one client to download that many files in parallel). A second problem is that JavaScript files and CSS files can be greatly compressed by simply removing comments and whitespace (at the cost of readability, of course). The MVC framework offers the concept of *bundles* to overcome these problems[[4]](#footnote-4).

Developers can programmatically bundle files by creating instances of the ScriptBundle or StyleBundle classes and add (respectively JavaScript and CSS) files to them. Visual Studio had already created some bundles by default. Open the file BundleConfig.cs file (in the App\_Start folder) and verify that four ScriptBundles and one StyleBundle instances are created. Every bundle has a name (e.g. ‘~/bundles/jquery’) and a number of files that are associated with it.

In order to use these bundles, we must add a reference to them in our layout page. Open \_Layout.cshtml and add the following lines to the <head> section of the file:

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

The first line includes all the CSS files that are contained in the “~/Content/css” bundle. The second line includes the JavaScript files of the modernizr framework. Add the following lines at the end (but still *in*) the body tag:

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

@RenderSection("scripts", required: false)

The first two lines include the JavaScript files of jquery and bootstrap. The third line is an extension point for other views that use this layout page. It is essentially a placeholder with the name ‘scripts’, and can be overwritten by child views.

## /Home/Index

The default page of an ASP.NET MVC website is the /Home/Index action. Thus, if the user surfs to <http://mysite.com/>, the MVC framework will redirect the request to <http://mysite.com/Home/Index> and call the Index action of the Home controller. We will now add this index page to the project.

Every page needs at least two components to work: an action in a controller, and a view. In the /Views/Home project folder, add a new View by right-clicking the folder and selecting the Add → View menu option. The name of the view is *Index*. Also select the ‘Use a layout page’ checkbox, but leave the associated text field empty. The view opens automatically, and is almost empty. Add a welcome message to the page.

Right-click the *Controller* project folder, and add a controller class by selecting Add → Controller. Choose an ‘MVC 5 Controller – Empty’ and name it ‘HomeController’. By default, the controller contains a method ‘Index’, which will be called if the user surfs to either <http://mysite.com/>, <http://mysite.com/Home/>, or <http://mysite.com/Home/Index>. The content of the method returns the default view of the action (without performing any kind of business logic). Run the application, and verify that it correctly shows the /Home/Index view (using your layout page).

When surfing to the website, the MVC framework interprets the requested URL. The default URL is automatically mapped to /Home/Index. The framework then parses the URL and retrieves the name of the controller and the requested action. In this case, the name of the controller is ‘Home’, which corresponds to a class called HomeController. The action is ‘Index’, which corresponds to a method with the same name in the controller. The MVC framework creates an instance of the HomeController class and calls the Index method. This method returns an ActionResult object, which can represent a (partial) page, a redirect command, etc. By simply calling the View method and returning its result, the Index method asks the MVC framework to lookup the view that corresponds to the method (in this case the Index.cshtml page in the /Views/Home project folder) and render it.

## Partial views

If you have completed the ASP.NET Web Forms module, you may have grown accustomed to using web user controls, but using these controls in the MVC framework is frowned upon. Web Forms controls tend to generate ugly and large HTML output, which is something we’re trying to avoid.

The MVC framework’s equivalent of web user controls is *partial views*. A partial view works in the same way as a normal view, but only renders a fragment of a page instead of the entire view. As a first (simple) example, we’re going to implement a partial view that shows the taglines.

Add a new view to the /Views/Shared project folder. Choose *TagLine* as its name and check the ‘Create as a partial view’ check box. Create the page; it will be completely empty. Paste the following line into it:

@model MVCModule.Models.TagLine

This means that the view will expect a class of type TagLine (in the MVCModule.Models namespace) as ‘input’. This input is called the *model* and is available through the Model property. The type of the Model property will be the type you enter here. It can be anything.

Add some code to the view that writes the Text property of the TagLine to the page. By now you should know the Razor syntax to output the value of a specific variable; if not, take another look at a Razor tutorial!

We now need to add a controller that retrieves a tagline and renders the view. Add a new empty controller to the /Controllers project folder and name it SharedController. Rename the Index method to ‘TagLine’ and make sure it returns the result of the PartialView method instead of the View method. There are a number of overloads that we can use when calling the PartialView method. One overload accepts an object as the model that will be passed to the view, which is what we want to use.

var tagManager = ServiceLocator.Resolve<ITagLineManager>();

return PartialView(tagManager.GetRandom());

Open the layout page (\_Layout.cshtml) and scroll to the correct location where you want the tagline to show. We need to instruct ASP.NET to render the partial view we just created at this location. You can do this by calling the RenderAction method of the Html class.

@{ Html.RenderAction("TagLine", "Shared"); }

Run the application, and check that the partial view works as expected. It will print out a random tagline at the correct location.

## Adding a menu

The next step is to add a menu to the website. Go to the *SharedController* and add a method *Menu* to it. We will programmatically retrieve the contents of the sitemap file (Web.sitemap) in order to pass it to a partial view that renders the menu. You can find an example of how to parse a sitemap file in the online MSDN documentation (<http://msdn.microsoft.com/en-us/library/system.web.xmlsitemapprovider.initialize.aspx>). Modify this code to retrieve all the child nodes of the root node (using the GetChildNodes method), and let the *Menu* method in the *Shared* controller return a PartialView that is initialized with the collection of sitemap nodes (returned by GetChildNodes) as its model.

Add a *Menu* partial view to the *Shared* project directory. Make sure its model is of type System.Web.SiteMapNodeCollection (which is passed to the view from the controller). Write code in the partial view to loop through the sitemap nodes and write out the corresponding HTML code for each node. If you use bootstrap, the end result of this operation could look like this:

<ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu" style="margin-bottom: 5px; display: block; position: static;">

<li><a href="/Home/Index">Home</a></li>

<li class="dropdown-submenu"><a tabindex="-1" href="#">Patients</a>

<ul class="dropdown-menu">

<li><a href="/Patient/Appointments">Appointments</a></li>

<li><a href="/Patient/CreateAppointment">Create appointment</a></li>

</ul>

</li>

<li><a href="/Home/OpeningHours">Information</a></li>

<li><a href="/Doctor/Appointments">Administration</a></li>

</ul>

The above solution requires that you add submenu support to bootstrap (which was removed as of version 3), by pasting the following lines of CSS to the Content\Site.css file:

.dropdown-submenu{position:relative;}

.dropdown-submenu>.dropdown-menu{top:0;left:100%;margin-top:-6px;margin-left:-1px;-webkit-border-radius:0 6px 6px 6px;-moz-border-radius:0 6px 6px 6px;border-radius:0 6px 6px 6px;}

.dropdown-submenu:hover>.dropdown-menu{display:block;}

.dropdown-submenu>a:after{display:block;content:" ";float:right;width:0;height:0;border-color:transparent;border-style:solid;border-width:5px 0 5px 5px;border-left-color:#cccccc;margin-top:5px;margin-right:-10px;}

.dropdown-submenu:hover>a:after{border-left-color:#ffffff;}

.dropdown-submenu.pull-left{float:none;}.dropdown-submenu.pull-left>.dropdown-menu{left:-100%;margin-left:10px;-webkit-border-radius:6px 0 6px 6px;-moz-border-radius:6px 0 6px 6px;border-radius:6px 0 6px 6px;}

The only thing left to do now is to add the menu to the layout page. Open the \_Layout.cshtml page and scroll to the location where the menu should be generated. Use the Html.RenderAction method again to render the partial view at this location. Run the application and verify that the menu is shown correctly.

## Using Data Access Objects

To implement the opening hours page, we need to retrieve the relevant data from the XML file in the controller and send it to the view. We can open the XML file using the XMLDocument class, but it is not a good idea to send this XMLDocument instance to the view as the view model. The XMLDocument contains a lot of features that may be potentially harmful. For example, it contains a Save method that can be (mis)used to overwrite the existing file on disk (or potentially another file). Not only is this a security risk, but we do not want to give the developer who’s writing the view the wrong idea about what he is supposed to do with the model. By using a very simple model, the developer writing the view will not be able to do anything unexpected with it. Furthermore, we want the views to be completely oblivious as to which data storage technology is used. If at a later point we want to switch to a different technology (e.g. perhaps we would like to store this data in the database instead of an XML file), we simply have to update the controller to get the relevant data from the database instead of the XML file. If we would have passed an XMLDocument to the view, changing the backend technology from XML to SQL also requires a rewrite of the view.

Every view has its own *view model class*, located in the Models directory. These classes are very simple (usually consisting of only properties) and hold all the data that the view requires. Create a new class file in the Models directory, and paste the following class definitions into it.

public class OpeningHoursViewModel {

public List<OpenDayViewModel> Days { get; set; }

}

public class OpenDayViewModel {

public string Name { get; set; }

public List<ConsultationViewModel> Consultations { get; set; }

}

public class ConsultationViewModel {

public string StartTime { get; set; }

public string EndTime { get; set; }

public string Description { get; set; }

}

The OpeningHoursViewModel is the class that represents the view model. It contains a list of OpenDayViewModel instances that represent a single day of the week. Every day also has a list of ConsultationViewModel instances. This class holds the information about a specific type of consultation on a specific day.

Create a view OpeningHours in the /Views/Home project directory. Choose ‘empty’ as the template, and choose the OpeningHoursViewModel class as the model type. It should also use the default layout page.

Open the Home controller and add a method *OpeningHours*. Add code to construct the required data model. Open the XML file[[5]](#footnote-5) and convert the contents into the respective OpeningHoursViewModel, OpenDayViewModel and ConsultationViewModel instances. At the end of the controller method, call the View method and pass it the OpenDayViewModel you created.

Open the OpeningHours view and write code that loops over the data model and writes out its information. Run the website, and verify that the OpeningHours view contains all the information from the XML file.

## HTTP verbs

Create a new empty controller class called *AccountController*. This class will implement the Login and Logout actions.

Conceptually, there are two different scenarios for the Login action. In the first scenario, the user simply surfs to /Account/Login. This generates a HTTP GET request that retrieves the HTML code of the login page. Then, when the user has entered his credentials and clicked the Login button on the page, the browser sends an HTTP POST request with the user’s data. This request should be handled by validating the credentials and, if successful, logging the user in. If unsuccessful, the user should get an error page (and the possibility to try again).

These two scenarios are handled by two different methods:

public ActionResult Login(string returnUrl) {

// ...

}

[HttpPost]

public ActionResult Login(LoginViewModel model, string returnUrl) {

// ...

}

The first method deals with the first scenario and processes the HTTP GET request. The second method deals with the HTTP POST request. The first method expects a parameter *returnUrl,* which can contain the URL where the user needs to be redirected to after logging in. This parameter is optional; if the browser does not send a value for the *returnUrl* parameter, it will be set to a null reference.

## Data models

The second variant has the same *returnUrl* parameter, but also an additional model parameter of type *LoginModel*. The *LoginModel* class is the data model type that is associated with the Login action. It is a simple class with some gettable and settable properties that represent all the information that is required to call a Login action.

public class LoginViewModel {

public string UserName { get; set; }

public string Password { get; set; }

public bool RememberMe { get; set; }

}

The MVC framework will automatically map any HTTP POST or GET parameters onto the fields of this class. For example, if you have a page that contains a text box with the id ‘UserName’, anything entered in this text box will automatically be mapped to the *UserName* property of the LoginModel class.

## Data annotations

The MVC framework can automatically validate models, both on client and server side. However, this means that the framework must somehow know which properties are required, which must fall within specific bounds, what error messages to generate, etc.

To tackle this problem, the MVC framework uses the concept of data annotations. Each property of the data model is decorated with one or more attributes that specify rules to which the property must comply (e.g. whether it is a required field) and other information. For this assignment session, we are only interested in three attributes: *Required*, *Display*, and *DataType*.

**Required** implies that the property is required. If the user does not enter a value for this field, the model will not be valid. The Required attribute also has an ErrorMessage property that holds a string that is shown to the user when he fails to enter a value for the required field.

**Display** is an attribute to hold a (localizable) string that describes the property and is displayed to the user. The Name property holds the (user friendly) name of the property.

**DataType** specifies extra information about the data type of a property. For example, it can be used to inform the system that a string property should actually contain a password, credit card number or email address (all of which are represented as string, but behave differently when displaying it to the user or validating the contents).

Other interesting attributes that you may want to take a look at are: StringLength, RegularExpression and Range.

One property can have multiple attributes. For example, the Password property can be decorated as follows:

[Required(ErrorMessage = "Please enter a password")]

[DataType(DataType.Password)]

[Display(Name = "Your password")]

public string Password { get; set; }

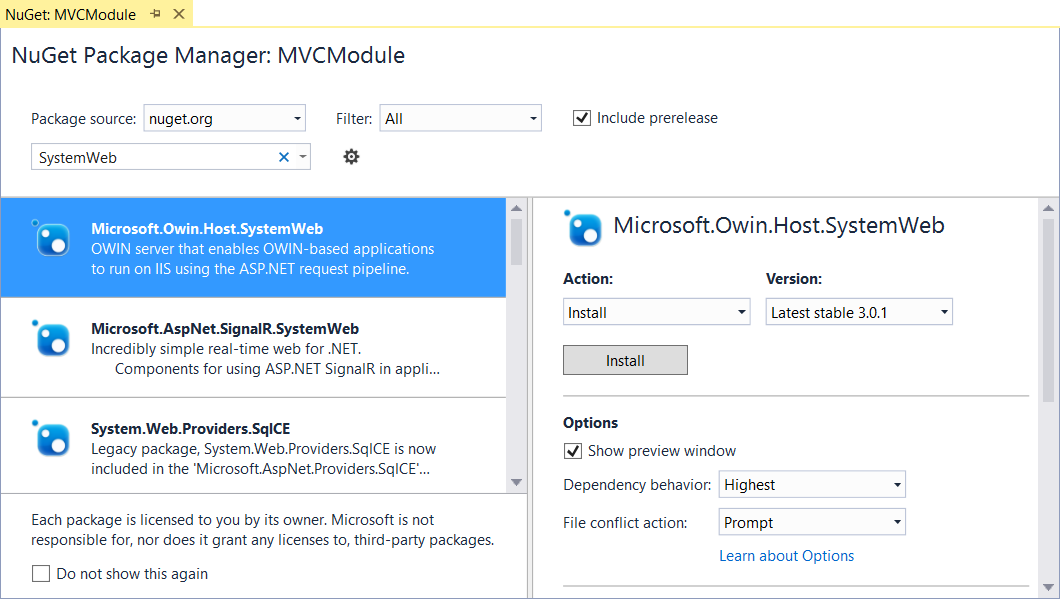
Add the LoginModel class to your project (if you haven’t done so already) to the Models directory. Decorate the properties using the attributes above.

## OWIN

The Open Web Interface for .NET (OWIN) defines a standard interface between .NET web servers and web applications. The goal of the OWIN interface is to decouple server and application, encourage the development of simple modules for .NET web development, and, by being an open standard, stimulate the open source ecosystem of .NET web development tools.

We will not go into the details of this framework in this module, but OWIN *is* important for our project because the authentication infrastructure that is used in the MVCModule.Models project uses the ASP.NET Identity framework which is an OWIN plug-in. Hence, in order to use authentication, we must first install OWIN and the Identity framework into our web project.

Right-click the web project and choose *Manage NuGet packages*. Search for the packages titled Microsoft.Owin.Host.SystemWeb and Microsoft.AspNet.Identity.Core, and install them.



## Logging in

The Login method that deals with the GET request is not very special. All it has to do is to render the view of the login page. We must pass the value of the *returnUrl* parameter, however the Login model does not support storing a return url. This is because the model only contains items that are conceptually related to the action (i.e. what the user enters as data), whereas the *returnUrl* is an implementation detail that has nothing to do with the act of logging in.

Fortunately, we have a mechanism that we can use to pass data (that is not part of the model) to the view. Both the controller and the view have a *ViewBag* property that returns an object where properties can be dynamically added (and read). The resulting code of the Login method becomes:

public ActionResult Login(string returnUrl) {

ViewBag.ReturnUrl = returnUrl;

return View();

}

Notice that the ViewBag object doesn’t actually implement the ReturnUrl property, but this property is added automatically by setting it[[6]](#footnote-6).

The Login method to handle the POST request is more interesting. It must validate the model, and the username/password combination. If the combination is correct, the user is logged in by setting an authentication cookie and the web browser is redirected to the return url (if any) or the default page of the website.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Login(LoginViewModel model, string returnUrl) {

if (ModelState.IsValid) {

var owinctx = Request.GetOwinContext();

using (var userManager = ServiceLocator.Resolve<IUserManager>()) {

var user = userManager.Find(model.UserName, model.Password);

if (user != null) {

var identity = userManager.CreateIdentity(user);

owinctx.Authentication.SignIn(identity);

if (Url.IsLocalUrl(returnUrl)) {

return Redirect(returnUrl);

} else {

return RedirectToAction("Index", "Home");

}

}

}

}

// If we got this far, something failed, redisplay form

ModelState.AddModelError("", "The username or password provided is incorrect.");

return View(model);

}

The above method has an additional attribute *ValidateAntiForgeryToken*. This will automatically validate a token that is generated by the login view in order to prevent certain replay attacks.

## The Login page

The controller code is only half of the Login story; we also need a Login view. Add a Login view to the /Views/Account folder. It is strongly typed and expects an object of type LoginViewModel. Select the LoginViewModel type from the dropdown list (it may only be present after you’ve built the solution).

Normally, you would expect to have to create an HTML <form> tag that embeds a number of <input> tags (text boxes for the username and password, a checkbox for the ‘remember me’ functionality, and a button to log in). Programming the page like this will indeed work perfectly with the MVC framework. You only need to make sure that the *name* and *id* attributes of the input elements correspond to the names of the properties in the model class. For example, if we wish to show a text box that maps to the UserName property of the model, we would need to add the following HTML code:

<input id="UserName" name="UserName" type="text" value="" />

However, we may also want to use automatic client-side validation and localization of the description, which are based on the data annotations we added to our model. Having to define these properties again would be a waste of time.

The MVC framework offers a number of methods in the HtmlHelper class (available through the Html property on a view page) that generate the correct HTML code automatically for you. These methods have built-in support for automatic validation and localization. In particular, we’re interested in the following methods:

* **LabelFor** – generates a descriptive text for a particular item (retrieves the data to display from the *Display* attribute on the data property)
* **TextBoxFor** – generates a text box for a specific data property
* **PasswordFor** – generates a password text box for a specific data property
* **ValidationMessagesFor** – shows validation errors for a specific data property (if any)

In addition to these methods, we will also use the following methods that are not related to data properties:

* **BeginForm** – begins a <form> tag (and automatically closes it when used in combination with the C# *using* keyword.
* **AntiForgeryToken** – generates a token that can be validated on the server side. This is used to avoid certain replay attacks.
* **ValidationSummary** – displays a summary of the validation errors in the model (if any)

Add the following contents to the Login view and investigate how it works.

<h2>Log in</h2>

<p>Enter your username and password.</p>

@using (Html.BeginForm(new { ReturnUrl = ViewBag.ReturnUrl })) {

@Html.AntiForgeryToken()

@Html.ValidationSummary(true)

<ol>

<li>

@Html.LabelFor(m => m.UserName)

@Html.TextBoxFor(m => m.UserName)

@Html.ValidationMessageFor(m => m.UserName)

</li>

<li>

@Html.LabelFor(m => m.Password)

@Html.PasswordFor(m => m.Password)

@Html.ValidationMessageFor(m => m.Password)

</li>

<li>

@Html.CheckBoxFor(m => m.RememberMe) @Html.LabelFor(m => m.RememberMe, new { @class = "checkbox" })

</li>

</ol>

<input type="submit" value="Inloggen" />

}

@section scripts {

@Scripts.Render("~/bundles/jqueryval")

}

The *Xxx*For methods use lambda function syntax to indicate which property of the model corresponds to the label/textbox/… in question. The variable *m* in the above examples refers to the model of the view (i.e. the LoginModel class).

In the default layout page, we have rendered the jQuery bundle, however this bundle does not include the jQuery validation framework by default. In order to enable client-side validations, we must include this jQuery validation framework in our view. Remember that we defined a new (optional) section ‘scripts’ in the default layout file (i.e. ~/Shared/\_Layout.cshtml) that could be overwritten in child views. The above code does just that: it replaces the default (empty) content of the scripts section with a call to the Render method that emits the JavaScript calls to the jQuery validation framework.

Run the application, surf to /Account/Login and verify that the login page works as expected. Valid usernames are Pieter, Frederic, Tuur, Lore, Lena, Ilse, Wout and Karel (and the corresponding passwords are p, f, t, l, l, i, w and k). Leaving the username or password text boxes empty will give specific (client-side) validation errors. Entering a wrong username/password combination will show an error in the validation summary. Using a correct combination will log the user in and redirect the browser to the index page. Also note that if the client-side validation is not enabled (i.e. the scripts section is removed), the server will still check the constraints that are defined on the data model and issue the corresponding errors if the model is not valid. However, client side validation gives immediate feedback and avoids having to connect to the server in order to verify the input.

Open the HTML source of the Login page and take a look at the HTML code that was generated. As you will see, the HTML code is very clean. It uses standard HTML tags and jQuery data validation.

## Logging out

Logging out is much easier than logging in. All we need is a Logout action in the AccountController class that retrieves an OWIN context and calls the SignOut method. Afterwards, it must redirect to the index page by returning the result of a call to the Redirect or RedirectToAction methods (instead of the View method we usually call). The first method takes a string parameter and can redirect to any page. The second method will redirect the user to a different action on the same website. In this case, we want to redirect the user to the Index action of the Home controller. Implement the Logout method.

The Logout method should only be called if a user is authenticated (i.e. has called the Login action first). We can request the MVC framework to make sure that only logged in users can call the actions of the Account controller.

Decorate the AccountController class with the [Authorize] attribute. As a consequence, the MVC framework will make sure that only authenticated users can access the actions of the controller. This is what we want for the Logout method, but obviously users do not have to be authenticated to call the Login method. We can mark the Login methods as exceptions to the rule by decorating them with [AllowAnonymous] attribute.

## User-specific views

Sometimes we want the information on our view pages to depend on whether a user is logged in or not (or which role he has). We can make two useful additions to this website.

We want to see the name of the user who is currently logged in. In addition to the name, we want a logout link as well. If no user is logged in, a login link should be shown. Modify the layout page to include this functionality (you can use a partial view, but that is not required).

The index page of the website should welcome the user by name and offer him the option to either view his appointments or create a new appointment. If no user is logged in, a link should be present to allow the user to log in. Modify the view page of the Index action on the Home controller to include this functionality.

Some interesting properties that you may want to use are the Request.IsAuthenticated property and the User.Identity property. These properties also exist for regular ASP.NET pages. In addition to these properties, also take a look at the ActionLink method of the HtmlHelper class. This method generates a HTML <a> tag that links to a given MVC action.

## Listing appointments model and controller

One of the features of the website is that patients can get an overview of all their appointments. We will need a new model, controller and view. Let’s start with the model and the controller.

Add the following AppointmentsViewModel class to the Model directory. It stores a list of appointments for the currently logged in patient and a list of doctors. Notice that we do not use data annotations for this model, because it is a read-only model from the point of view of the user (i.e. he cannot modify values).

public class AppointmentsViewModel {

public IEnumerable<Appointment> Appointments { get; set; }

public IDictionary<int, Doctor> Doctors { get; set; }

}

Add a new empty controller named *PatientController* and remove the Index method. Decorate the controller with the [Authorize] attribute. Add an *Appointments* action and write code that retrieves the appointments of the currently logged on user (the GetAppointmentsForPatient method of an IAppointmentManager instance) and store this in an instance of the AppointmentsViewModel class. Also retrieve the list of doctors as a dictionary where the Id column is the key of the dictionary and store this in the AppointmentsViewModel instance as well[[7]](#footnote-7). Now return the default view of the action with the model you initialized.

## Listing appointments view

Create an Appointments view in the /Views/Patient directory. Its model is an instance of the AppointmentsModel class. Enter the full name of the class (including the namespace). Add code to the view page to list all the appointments (or display a message if no appointments are found). To convert the DoctorId property to the name of a doctor, you can use the *Doctors* dictionary in the model. Run the application and verify that the page works as expected.

## Adding extensions

Before we can continue with the development of our website, we need to add some extra features to it. In particular, we would like to add support for jQuery UI – a library built on top of jQuery to add support for advanced user interface elements (e.g. datepickers, progress bars, sliders, …) – and AJAX.

Open the NuGet package manager by right-clicking the web project and selecting *Manage NuGet packages*. Search for the JQuery.UI.Combined package and install it.

Installing this package will add a number of files to the project. Two jquery-ui-\* files are added to the ~/Scripts folder and a large number of files are added to ~/Content/themes/base. In order to be able to use them in the site, we add them to the Bundle configuration. Open file BundleConfig.cs file in the App\_Start folder, and add a new ScriptBundle with the name “~/bundles/jqueryui” that includes the files “~/Scripts/jquery-ui-{version}.js”. Also add the file “~/Content/themes/base/all.css” to the “~/Content/css” style bundle. jQuery UI is now ready to be used.

After installing jQuery UI, open the NuGet package manager again and install the Microsoft unobtrusive AJAX package (Microsoft.jQuery.Unobtrusive.Ajax). Create a new ScriptBundle (“~/bundles/jqueryajax”) that includes the file “~/Scripts/jquery.unobtrusive-ajax.js”.

## Create appointment model

When creating an appointment, the user must enter the date and reason of the appointment, and which doctor he wants to see. These things will have to be represented as properties in the data model of the CreateAppointment action. In addition, the view will also need a list of all the doctors. This gives us the following model:

public class CreateAppointmentViewModel {

public DateTime Date { get; set; }

public int DoctorId { get; set; }

public string Reason { get; set; }

public IEnumerable<Doctor> Doctors { get; set; }

}

Add this class to the Models folder, and annotate the properties using the data annotations attributes. Keep in mind that you only need to decorate properties that can be changed by the user.

## Create appointment views

There are two views associated with the create appointment action: one view where the user can enter the information of the appointment, and another view that is shown when the appointment has been successfully created.

Let’s start with the easiest one of the two: the confirmation view. Add a new ‘AppointmentCreated’ view in the /Views/Patient folder, with a model of type CreateAppointmentViewModel. Add a message informing the user that the appointment has been created successfully and remind him of the date and doctor he has selected.

Add a second ‘CreateAppointment’ view with a model of type CreateAppointmentViewModel. We need to add a form to the page that has a dropdown list, a text box (for the date), and a second text box for the reason. The resulting code could look like this:

<h2>Create appointment</h2>

<p>Please enter the appointment data.</p>

@using (Html.BeginForm(new { ReturnUrl = ViewBag.ReturnUrl })) {

@Html.ValidationSummary(true)

<p>

@Html.LabelFor(m => m.DoctorId): @Html.DropDownListFor(m => m.DoctorId,

new SelectList(Model.Doctors, "ID", "Name"))

@Html.ValidationMessageFor(m => m.DoctorId)

</p>

<p>

@Html.LabelFor(m => m.Date) <input type="date" id="Date" name="Date"

class="date" value='@Model.Date.ToString("dd/MM/yyyy")' />

@Html.ValidationMessageFor(m => m.Date)

</p>

<p>

@Html.LabelFor(m => m.Reason): @Html.TextBoxFor(m => m.Reason)

@Html.ValidationMessageFor(m => m.Reason)

</p>

<p><input type="submit" value="Create appointment" /></p>

}

@section scripts {

@Scripts.Render("~/bundles/jqueryui")

<script type="text/javascript">

$(".date").datepicker({

defaultDate: $("#Date").val(), dateFormat: "dd/m/yy",

minDate: "@DateTime.Now.ToString("dd/MM/yyyy")"

});

</script>

}

Take a closer look at how the DropDownListFor method works. The first parameter identifies which element in the model should be updated when the user selects a specific value. The second parameter is of the type SelectList and generates the key/value pairs from which the user can select. The SelectList constructor expects three parameters: the first that holds the items from which to select, the second parameter that holds the name of the property that should be used as the key for the elements, and the third parameter that identifies the property that should be used as the description for each element.

The implementation of the date picker is also of note. We use a <input> tag of type ‘date’. The name and the id of the tag correspond to the Date property in the model. The class of the tag is set to the *date* style. Finally, we overwrite the ‘scripts’ section to render the jQuery UI scripts bundle and to initialize the datepicker.

One last problem remains to be solved. When the user selects a date, it will be in the format dd/mm/yyyy. However, it is possible that the website runs by default in a culture that parses dates in a different way. For example, if you run the website on an English edition of Windows, the date format will by default be mm/dd/yyyy. This may result in problems when the MVC framework automatically parses the date the user entered (using the wrong date format). To solve this, we need to set the culture manually. The fix is simple; add the following node to the Web.config as a child of the system.web node:

<globalization culture="nl-BE" uiCulture="nl-BE" />

## Create appointment controller

The CreateAppointment page works in a similar way as the Login page. We differentiate between two scenarios: one where the user performs a GET of the CreateAppointment page and another where the users POSTs data to CreateAppointment. Hence, we will again define two actions, each handling one of the scenarios.

The first CreateAppointment action handles the GET request. It has no parameters and simply initializes a CreateAppointmentViewModel instance and returns the default view. Make sure the Doctors property of the model is set to a list of available doctors. You may also set the Date property to the date of today. Run the application and verify that this works correctly. The CreateAppointment action should return a page where the user can enter his information and select a doctor.

The second CreateAppointment action handles the POST request. It has one parameter of the CreateAppointmentViewModel type. The method should verify that the model is valid. If the model is valid, the appointment should be created and return the ‘AppointmentCreated’ view should be returned (with the correct model). If the model was not valid, add an error message to the model state and return the default view. Note that in both cases, we need to repopulate the Doctors property of the model. This property will be a null reference when the CreateAppointment method is called, because the browser does not include a list of possible values for the dropdown list in the POST request. Run the application and verify that the CreateAppointment action is working as expected.

## Doctor appointments with AJAX

The last page we need to implement is the page where doctors can view their appointments. To spice things up a bit, we will use AJAX to refresh the page when a new doctor is selected from the dropdown list.

In the view of the Appointments page, we will have a drop down list that contains all the doctors in the database, as well as a partial view that contains the list of appointments for the selected doctor. Whenever the user selects a different doctor, we will use AJAX to refresh the partial view. To this end, we need to decouple the contents of the partial view with the appointments list and the contents of the full view. We get the following two model classes:

public class DoctorAppointmentsViewModel {

public IEnumerable<Doctor> Doctors { get; set; }

public int DoctorId { get; set; }

public AppointmentListViewModel AppointmentList { get; set; }

}

public class AppointmentListViewModel {

public IEnumerable<Appointment> Appointments { get; set; }

}

The DoctorAppointmentsViewModel is the model for the full view, and the AppointmentListViewModel is the model for the partial view with the appointment list. When we need to refresh the partial view, only the AppointmentListViewModel is required. These model classes do not need data annotations, because they are not user-modifiable.

## Doctor appointments controller

Add an empty Doctor controller and remove the Index method. Make sure only authenticated users that are members of the ‘Doctors’ role have access to the actions in this controller. You can specify these constraints using the [Authorize] attribute.

Add an Appointments action that initializes an instance of the AppointmentListViewModel class and returns the default view. The method could look like this:

public ActionResult Appointments() {

using (ThePracticeDataContext db = new ThePracticeDataContext()) {

var doc = db.Doctors.First();

var apps = db.Appointments.Where(c => c.DoctorId == doc.Id).OrderBy(c => c.Date);

return View(new DoctorAppointmentsModel {

Doctors = db.Doctors.ToList(), DoctorId = doc.Id,

AppointmentList = new AppointmentListModel { Appointments = apps.ToList() }

});

}

}

## Doctor appointments view

Create a view *Appointments* in the /Views/Doctor folder, with the DoctorAppointmentsViewModel class as its type of model. The view must perform an AJAX call whenever the user selects a new doctor from the dropdown list.

<h2>Appointments (by doctor)</h2>

@using (Ajax.BeginForm("GetAppointmentList", "Doctor", new AjaxOptions() {

UpdateTargetId = "applist", InsertionMode = InsertionMode.Replace },

new { id = "patientlist" })) {

<p>

@Html.DropDownListFor(m => m.DoctorId, new SelectList(Model.Doctors,

"ID", "Name", Model.DoctorId), new { onchange = "$('#patientlist').submit()" })

</p>

}

<div id="applist">

@{ Html.RenderPartial("~/Views/Doctor/AppointmentList.cshtml", Model.AppointmentList); }

</div>

@section scripts {

@Scripts.Render("~/bundles/jqueryajax")

}

Instead of using the Html.BeginForm method, the code calls the Ajax.BeginForm method. This generates a <form> tag that makes an AJAX call to the webserver. The first two parameters are the names of the action and the controller that should be called. The third parameter contains AJAX options that specify what should happen when the ajax call completes. In this case, it specifies that the result of the AJAX call should replace the contents of the HTML object with id *applist*. The fourth parameter contains optional HTML attributes. We manually set the name of the form to *patientlist*, because we will need this name later on.

The dropdown list is generated much like on the CreateAppointments page. However, in this case we specify an additional parameter to add a short script to the onchange attribute of the dropdown list. This script will make sure that every time the user selects a new doctor, the form is submitted to the server (which refreshes the appointments list).

Finally, there’s a <div> that will contain the results of the AJAX calls. When the page is loaded, it must of course also contain data. A call to the RenderPartial method is used to render the initial list of appointments.

In order to make sure that AJAX support is enabled on this view, we overwrite the scripts section to make sure it renders the AJAX bundle we defined earlier.

## The appointment list

Add a new partial view ‘AppointmentList’ to the /Views/Doctor folder. Its model should be an instance of the AppointmentListViewModel class.

If you completed the ASP.NET Web Forms module, you know we used the GridView control to display the appointment data. The ASP.NET MVC framework does not offer a tool out of the box that offers a comparable functionality, so we will need to use our own. Download the DataGridHelper.cs file and add it to your project. This class contains an implementation of a DataGrid control that is built for MVC applications. We can now use this control in the AppointmentList view:

@using UCLL.Helpers

@(Html.DataGrid<MVCModule.Models.Appointment>(Model.Appointments, new string[] { "Date", "Patient", "Reason" }))

By importing the UCLL.Helpers namespace into the page, we make the DataGrid extension method available. The method takes two parameters: the enumerable list of items to show and the properties from which to retrieve data. Each property will be rendered as a column. Run the application and verify that the appointments of the initially selected doctor are shown. (Make sure you log in as Pieter or Frederic; these are the only two user accounts that are in the Doctors role.)

## Implementing the AJAX call

If you select a different doctor from the dropdown list, you will notice that the page is not refreshed with the appointments of the newly selected doctor. The problem is that even though the page makes the AJAX call to the GetAppointmentList method already, the server doesn’t implement the called method yet.

Open the Doctor controller, and add a new method GetAppointmentList. The method should only support POST calls (hence it must be decorated with the [HttpPost] attribute) and has one parameter of type DoctorAppointmentsViewModel. It returns the result of a call to the PartialView method. The PartialView method expects two parameters: the name of the partial view to render (in our case ‘~/Views/Doctor/AppointmentList.cshtml’) and an initialized AppointmentListModel instance.

Run the application and verify that the page now works as expected. Selecting a different doctor results in a (partial) page refresh.

## Adding delete functionality

The website in the ASP.NET Web Forms module also supported deleting appointments. To support this, we will need to implement an action that deletes an appointment and also add a column with a delete button to the DataGrid.

Add a DeleteAppointment method to the Doctor controller that has one parameter with name *id* of type *int*. The method must delete the appointment with the given id. Then, it must redirect to the /Doctor/Appointments action.

Finally, we need to show a Delete link in the DataGrid. To accomplish this, we will need to modify the code that generates the DataGrid as follows:

@(Html.DataGrid<MVCModule.Models.Appointment>(Model.Appointments,new MvcDataGridColumn[] {

new MvcDataGridColumn("Date", "Datum", "dd/MM/yyyy"),

new MvcDataGridColumn("Patient", "Patiënt"),

new MvcDataGridColumn("Reason", "Reden"),

new MvcDataGridActionColumn("[delete]", "DeleteAppointment", "Doctor", "Id") }))

Instead of giving the DataGrid an array of strings that contain column names, we now give the DataGrid an array of MvcDataGridColumn instances. These objects allow us to better describe the layout of the columns. We can also add an MvcDataGridActionColumn instance that represents a column with an action link. In the above code, we specify that the link should call the DeleteAppointment method of the Doctor controller. We also specify that the *Id* property of the Appointment class will be used as the parameter to send to the DeleteAppointment method.

Run the application and verify that the website is now fully functional.

1. http://getbootstrap.com/ [↑](#footnote-ref-1)
2. http://jquery.com/ [↑](#footnote-ref-2)
3. If you need to write and ASP.NET Web Forms application for the final examination project, make sure you keep your pages as clean as possible. Placing business logic in your ASP.NET Web Forms will have a negative impact on your grade! [↑](#footnote-ref-3)
4. A detailed explanation can be found at <http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification> [↑](#footnote-ref-4)
5. Using an XMLDocument instance is easier than using an XmlReader instance. Also take a look at the Server.MapPath method; this can be useful if you want to retrieve the full path of the XML file on the hard disk given its relative path in the website. [↑](#footnote-ref-5)
6. This is called a [dynamic object](https://msdn.microsoft.com/en-us/library/dd264736.aspx). Properties and methods can be added on the fly at runtime. [↑](#footnote-ref-6)
7. The IDoctorManager class returns a simple list of Doctor objects instead of a Dictionary. However, if you have completed the LINQ Module, you know you can use the ToDictionary method for this: docMgr.Doctors.ToDictionary(c => c.Id) [↑](#footnote-ref-7)