Advanced Internet Technologies

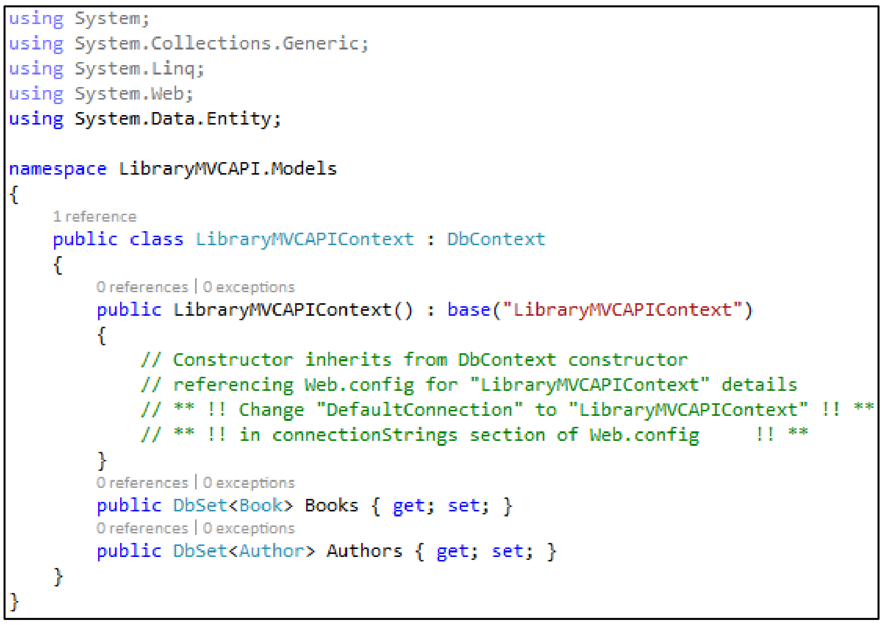
MVC Application Development Lab

This document contains a set of lab material that covers developing sample functionality for an ASP.Net MVC application in the following areas:

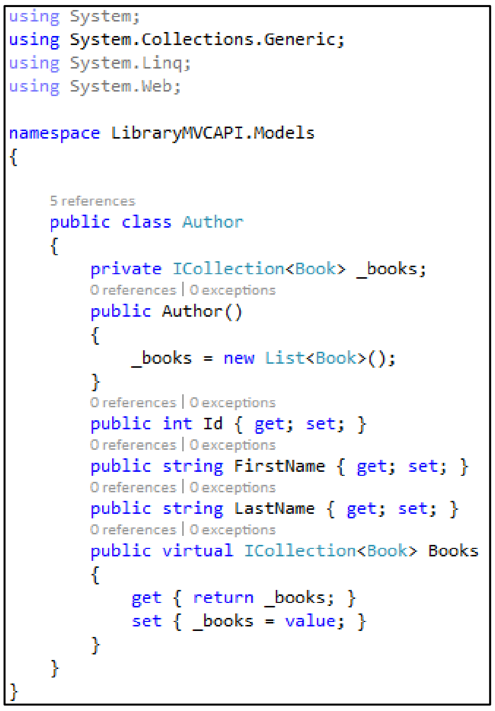
I) Implementing a *Seed* method to initialise a database with sample data.
II) Creating Web API Controllers in an ASP.Net MVC application.
III) Using Data Transfer Objects (DTOs).
IV) Using Ajax to interface with the Web API.
V) Modification of the *ApplicationUser* class to incorporate additional attributes into user accounts for the application.

I - Implementing a *Seed* method to initialise a database with sample data

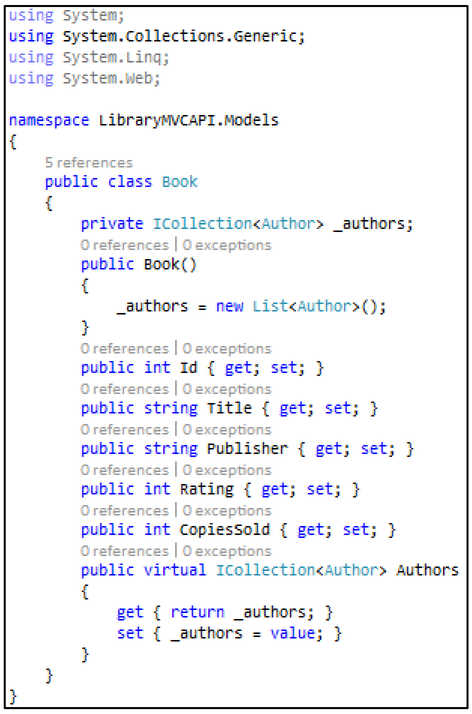
We assume that we have already created an MVC application called *LibraryMVCAPI* that has implemented Entity Framework Code-First development principles resulting in the creation of the following classes (in the Models directory):



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace LibraryMVCAPI.Models
{
    1 reference
    public class LibraryMVCAPIContext : DbContext
    {
        0 references | 0 exceptions
        public LibraryMVCAPIContext() : base("LibraryMVCAPIContext")
        {
            // Constructor inherits from DbContext constructor
            // referencing Web.config for "LibraryMVCAPIContext" details
            // ** !! Change "DefaultConnection" to "LibraryMVCAPIContext" !! **
            // ** !! in connectionStrings section of Web.config     !! **
        }
        0 references | 0 exceptions
        public DbSet<Book> Books { get; set; }
        0 references | 0 exceptions
        public DbSet<Author> Authors { get; set; }
    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace LibraryMVCAPI.Models
{
    5 references
    public class Author
    {
        private ICollection<Book> _books;
        0 references | 0 exceptions
        public Author()
        {
            _books = new List<Book>();
        }
        0 references | 0 exceptions
        public int Id { get; set; }
        0 references | 0 exceptions
        public string FirstName { get; set; }
        0 references | 0 exceptions
        public string LastName { get; set; }
        0 references | 0 exceptions
        public virtual ICollection<Book> Books
        {
            get { return _books; }
            set { _books = value; }
        }
    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace LibraryMVCAPI.Models
{
    5 references
    public class Book
    {
        private ICollection<Author> _authors;
        0 references | 0 exceptions
        public Book()
        {
            _authors = new List<Author>();
        }
        0 references | 0 exceptions
        public int Id { get; set; }
        0 references | 0 exceptions
        public string Title { get; set; }
        0 references | 0 exceptions
        public string Publisher { get; set; }
        0 references | 0 exceptions
        public int Rating { get; set; }
        0 references | 0 exceptions
        public int CopiesSold { get; set; }
        0 references | 0 exceptions
        public virtual ICollection<Author> Authors
        {
            get { return _authors; }
            set { _authors = value; }
        }
    }
}
```

As can be seen from the class definitions above, we have implemented a many-to-many relationship between the *Author* and *Book* entities. This will have an impact on the design of our Web API functionality when we implement the Web API Controller methods.

A set of MVC Controller classes (i.e., *AuthorsController* class and *BooksController* class) and the associated Views have also been scaffolded for this application.

Entity Framework Code-First migrations have been enabled for the application. An *InitialCreate* migration has also been added. The database has been updated.

```
PM> Enable-Migrations -ContextTypeName LibraryMVCAPI.Models.LibraryMVCAPIContext
Checking if the context targets an existing database...
Code First Migrations enabled for project LibraryMVCAPI.
```

```
PM> Add-Migration InitialCreate
Scaffolding migration 'InitialCreate'.
The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is
used to calculate the changes to your model when you scaffold the next migration. If you make additional changes to
 your model that you want to include in this migration, then you can re-scaffold it by running 'Add-Migration
InitialCreate' again.
```

```
PM> Update-Database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201612032112264_InitialCreate].
Applying explicit migration: 201612032112264_InitialCreate.
Running Seed method.
```

Note that the last action performed by the Update-Database command was **Running Seed Method**.

The **Seed** method is used to initialise a database with some records. When we ran the **Enable-Migrations** command a folder called **Migrations** was added to the project. A file called **Configuration.cs** was also created in this folder. The **Seed** method that runs during the **Update-Database** command is contained in the **Configuration.cs** file. We will edit this method to add several Book and Author records. We will also establish many-to-many relationships between the Book and Author records. To do this:

i) Include the following *using* statements:

```
using System.Collections.Generic;
using LibraryMVCAPI.Models;
```

ii) Add code to the **Seed** method to create a *List* of *Book* objects and add/update these objects in the *context.Books* collection.

```
var books = new List<Book>
{
    new Book {Title="The Good Book", CopiesSold=2464, Publisher="Happy Swamp", Rating=2 },
    new Book {Title="The Bad Book", CopiesSold=666, Publisher="Wobbly Handle", Rating=4 },
    new Book {Title="The OK Book", CopiesSold=246, Publisher="Angry Frog", Rating=4 },
    new Book {Title="The Noisy Book", CopiesSold=24, Publisher="Ear Today", Rating=6 },
    new Book {Title="This is not a Book", CopiesSold=2464, Publisher="Dada Gone", Rating=1 }
};
books.ForEach(b => context.Books.AddOrUpdate(book => book.Title, b));
context.SaveChanges();
```

iii) Add code to the **Seed** method to create a *List* of *Author* objects and add/update these objects in the *context.Authors* collection. We add some of the *Book* objects to each of the *Author* object's *Books* collection properties.

```
var authors = new List<Author>
{
    new Author {FirstName = "AKA", LastName = "Noone",
                Books = books.Where(b => (b.Title == "The Good Book") || (b.Title == "The Bad Book")).ToList() },
    new Author {FirstName = "J.K.", LastName = "Growling",
                Books = books.Where(b => (b.Title == "The Good Book") || (b.Title == "The OK Book")).ToList() },
    new Author {FirstName = "A.N.", LastName = "Other",
                Books = books.Where(b => (b.Title == "The Noisy Book") || (b.Title == "This is not a Book")).ToList() },
    new Author {FirstName = "Pu", LastName = "Lp",
                Books = books.Where(b => (b.Title == "The OK Book") || (b.Title == "The Bad Book")).ToList() }
};
authors.ForEach(a => context.Authors.AddOrUpdate(author => author.LastName, a));
context.SaveChanges();
```

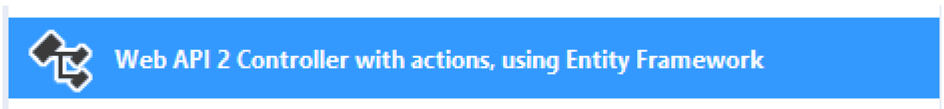
iv) Run the **Update-Database** command again.

```
PM> Update-Database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
No pending explicit migrations.
Running Seed method.
```

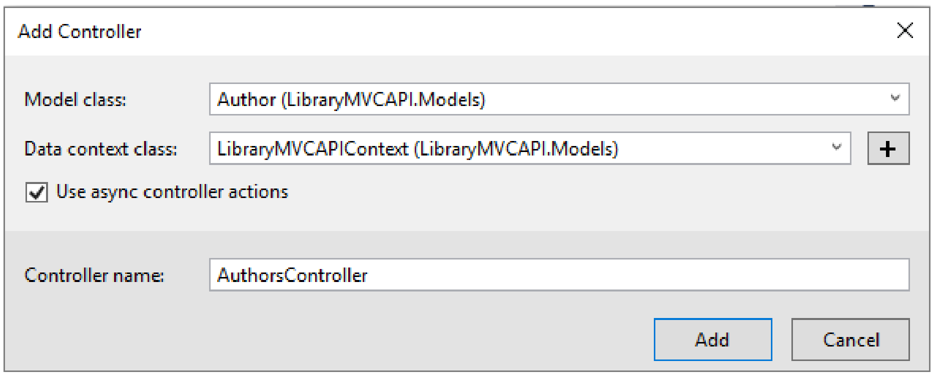
II - Creating Web API Controllers in an ASP.Net MVC application

We will next create Web API Controller classes for the *Author* and *Book* entity classes. To do this:

i) Add a *New Folder* to the project called **API** .
ii) Right-click on the **API** folder and choose to Add a new Controller.
iii) Choose the option **Web API 2 Controller with actions, using Entity Framework**



iv) Add the Controller as follows:



Add Controller

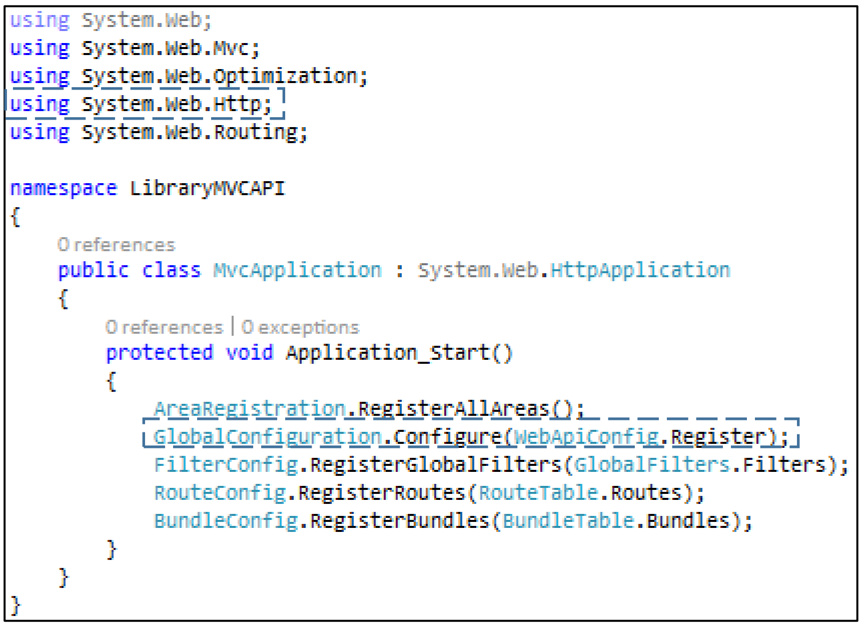
Model class: Author (LibraryMVCAPI.Models)

Data context class: LibraryMVCAPIContext (LibraryMVCAPI.Models)

☑ Use async controller actions

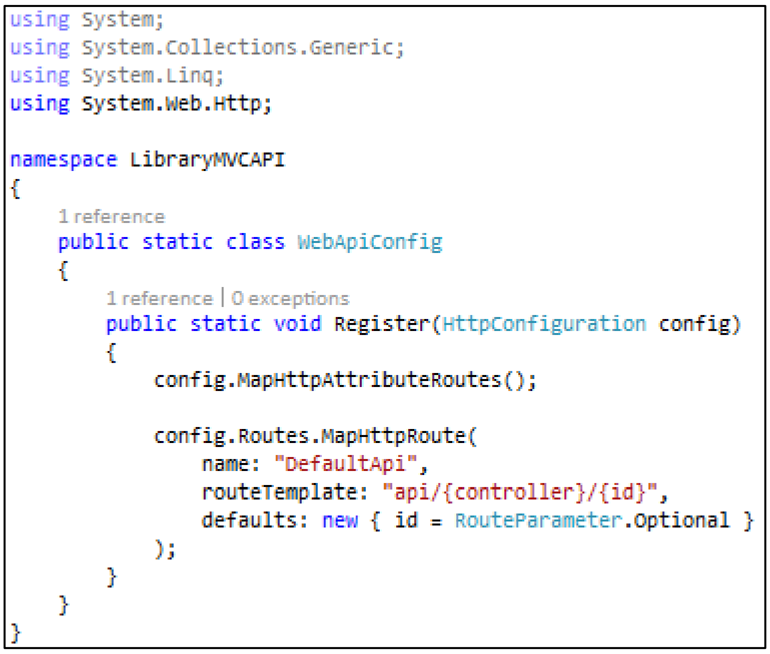
Controller name: AuthorsController

You will be presented with a **readme.txt** document informing you of certain edits that are required to the **Global.asax** file. Make these changes.



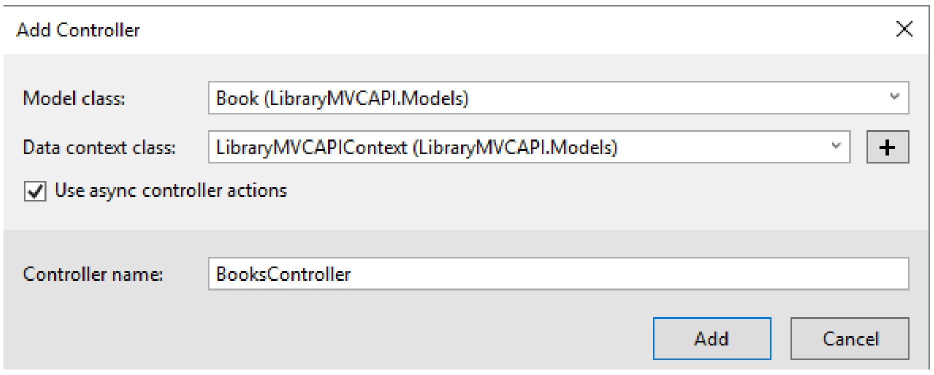
```
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Http;
using System.Web.Routing;

namespace LibraryMVCAPI
{
    0 references
    public class MvcApplication : System.Web.HttpApplication
    {
        0 references | 0 exceptions
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            GlobalConfiguration.Configure(WebApiConfig.Register);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

A **WebAPIConfig.cs** file will also have been added to your project in the **App_Start** folder. This file sets up the default URL routing for the Web API Controllers.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace LibraryMVCAPI
{
    1 reference
    public static class WebApiConfig
    {
        1 reference | 0 exceptions
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

v) Add another Web API Controller for the Book entity class.


Add Controller

Model class: Book (LibraryMVCAPI.Models)

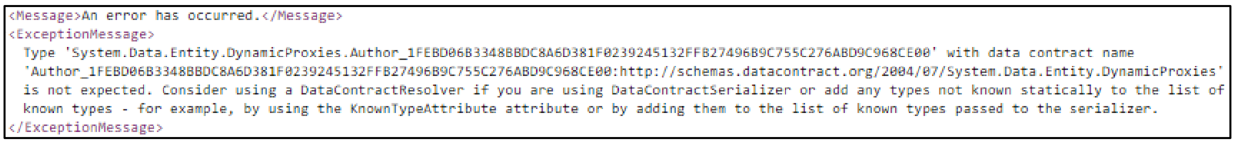
Data context class: LibraryMVCAPIContext (LibraryMVCAPI.Models)

✓ Use async controller actions

Controller name: BooksController

vi) Build and run the application and navigate to the **api/Authors** URL.

You will be presented with an exception similar to the following exception message!!:

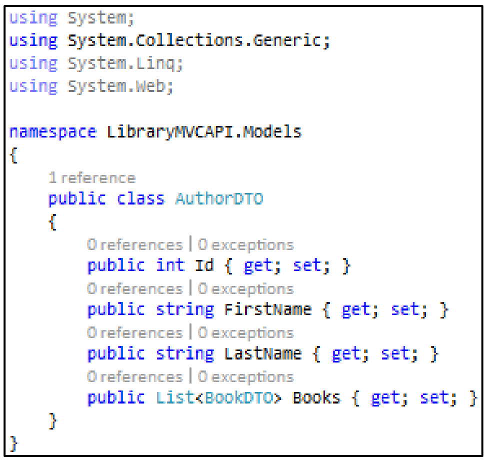

```
<Message>An error has occurred.</Message>
<ExceptionMessage>
  Type 'System.Data.Entity.DynamicProxies.Author_1FEBD06B3348BBDC8A6D381F0239245132FFB27496B9C755C276ABD9C968CE00' with data contract name
  'Author_1FEBD06B3348BBDC8A6D381F0239245132FFB27496B9C755C276ABD9C968CE00:http://schemas.datacontract.org/2004/07/System.Data.Entity.DynamicProxies'
  is not expected. Consider using a DataContractResolver if you are using DataContractSerializer or add any types not known statically to the list of
  known types - for example, by using the KnownTypeAttribute attribute or by adding them to the list of known types passed to the serializer.
</ExceptionMessage>
```

We are encountering this error due to the fact that we have implemented a many-to-many relationship between the *Author* and *Book* entity classes. When the application attempts to serialize the *Author* objects for transmission back to the client, it encounters a problem when it attempts to serialize each of the *Book* objects associated with the *Author* object. This is because each of the *Book* objects also has a property linking back to a set of *Author* objects. This creates a circular reference that prevents the *Author* objects being serialized. To overcome this problem we will create a set of **Data Transfer Object (DTO)** classes.

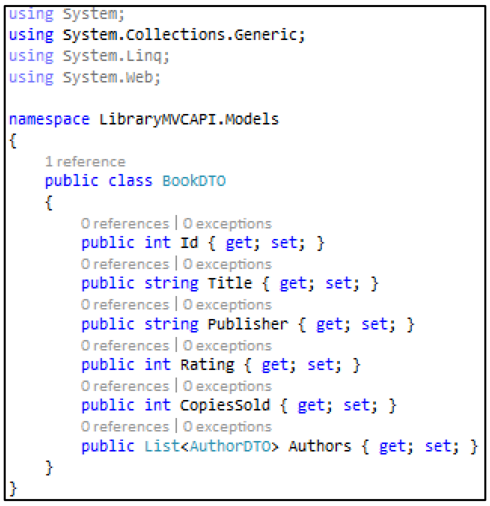
III - Using Data Transfer Objects (DTOs)

Data Transfer Objects (DTOs) are generally used to carry the data associated with objects between different processes. We can think of them as light-weight versions of objects that do not contain behaviour (i.e., methods) or other object characteristics. DTOs are like a reflection of the objects from which they are derived. They carry a representation of the original object's data member values. We will use some DTOs to get around the problem that was encountered whilst trying to serialize objects that contain circular references. Rather than directly using the *Author* and *Book* classes in the Web API controllers, we will use DTO classes instead. See https://www.asp.net/web-api/overview/data/using-web-api-with-entity-framework/part-5 .

i) Create *AuthorDTO* and *BookDTO* classes that reflect representations of the *Author* and *Book* classes' data members. To do this create **AuthorDTO.cs** and **BookDTO.cs** files in the **Models** folder as follows:

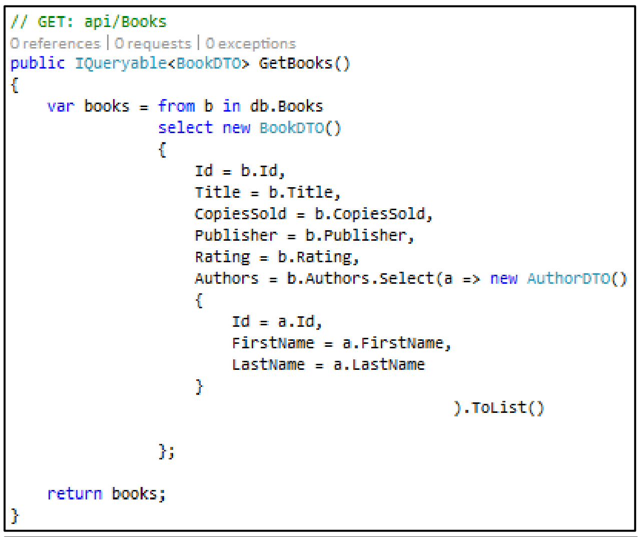


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace LibraryMVCAPI.Models
{
    1 reference
    public class AuthorDTO
    {
        0 references | 0 exceptions
        public int Id { get; set; }
        0 references | 0 exceptions
        public string FirstName { get; set; }
        0 references | 0 exceptions
        public string LastName { get; set; }
        0 references | 0 exceptions
        public List<BookDTO> Books { get; set; }
    }
}
```

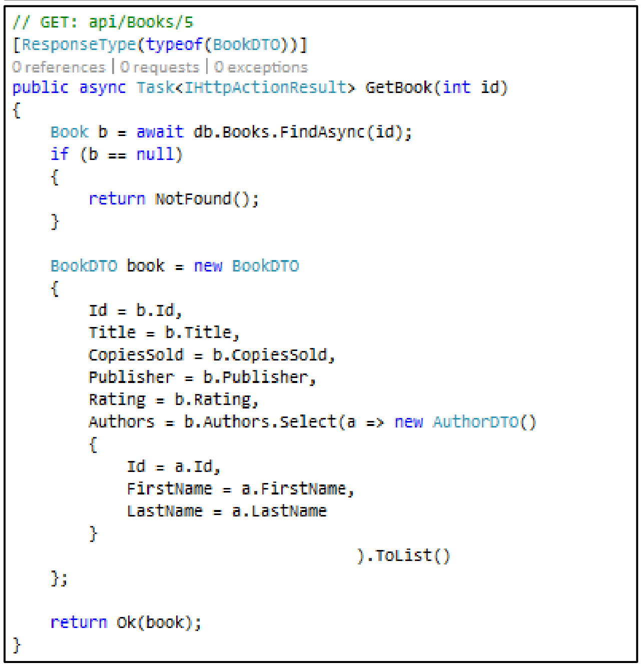


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace LibraryMVCAPI.Models
{
    1 reference
    public class BookDTO
    {
        0 references | 0 exceptions
        public int Id { get; set; }
        0 references | 0 exceptions
        public string Title { get; set; }
        0 references | 0 exceptions
        public string Publisher { get; set; }
        0 references | 0 exceptions
        public int Rating { get; set; }
        0 references | 0 exceptions
        public int CopiesSold { get; set; }
        0 references | 0 exceptions
        public List<AuthorDTO> Authors { get; set; }
    }
}
```

ii) Replace the code in the Web API *BooksController* class as follows:

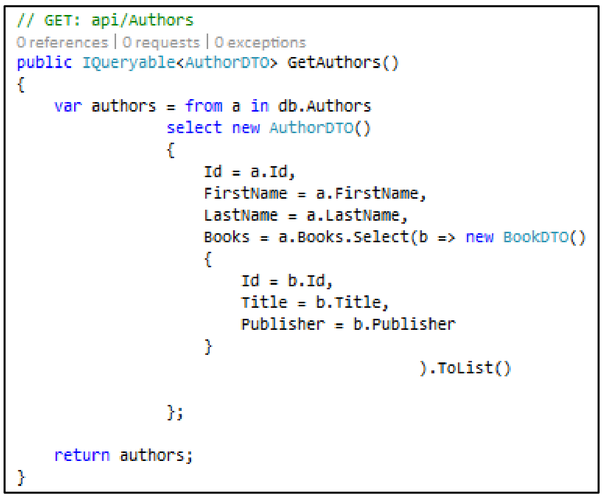


```
// GET: api/Books
0 references | 0 requests | 0 exceptions
public IQueryable<BookDTO> GetBooks()
{
    var books = from b in db.Books
                select new BookDTO()
                {
                    Id = b.Id,
                    Title = b.Title,
                    CopiesSold = b.CopiesSold,
                    Publisher = b.Publisher,
                    Rating = b.Rating,
                    Authors = b.Authors.Select(a => new AuthorDTO()
                    {
                        Id = a.Id,
                        FirstName = a.FirstName,
                        LastName = a.LastName
                    }
                                             ).ToList()

                };

    return books;
}
```

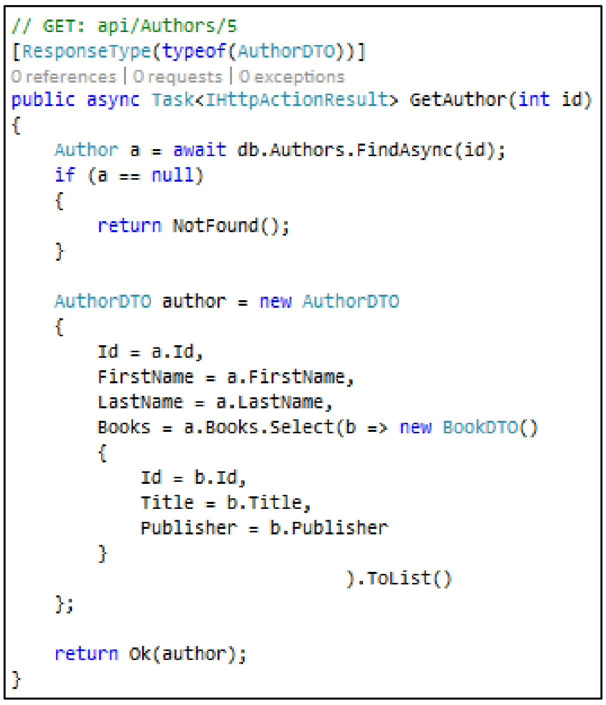


```
// GET: api/Books/5
[ResponseType(typeof(BookDTO))]
0 references | 0 requests | 0 exceptions
public async Task<IHttpActionResult> GetBook(int id)
{
    Book b = await db.Books.FindAsync(id);
    if (b == null)
    {
        return NotFound();
    }

    BookDTO book = new BookDTO
    {
        Id = b.Id,
        Title = b.Title,
        CopiesSold = b.CopiesSold,
        Publisher = b.Publisher,
        Rating = b.Rating,
        Authors = b.Authors.Select(a => new AuthorDTO()
        {
            Id = a.Id,
            FirstName = a.FirstName,
            LastName = a.LastName
        }
                                   ).ToList()
    };

    return Ok(book);
}
```

iii) Replace the code in the Web API *AuthorsController* class as follows:



```
// GET: api/Authors
0 references | 0 requests | 0 exceptions
public IQueryable<AuthorDTO> GetAuthors()
{
    var authors = from a in db.Authors
                  select new AuthorDTO()
                  {
                      Id = a.Id,
                      FirstName = a.FirstName,
                      LastName = a.LastName,
                      Books = a.Books.Select(b => new BookDTO()
                      {
                          Id = b.Id,
                          Title = b.Title,
                          Publisher = b.Publisher
                      }
                                              ).ToList()

                  };

    return authors;
}
```



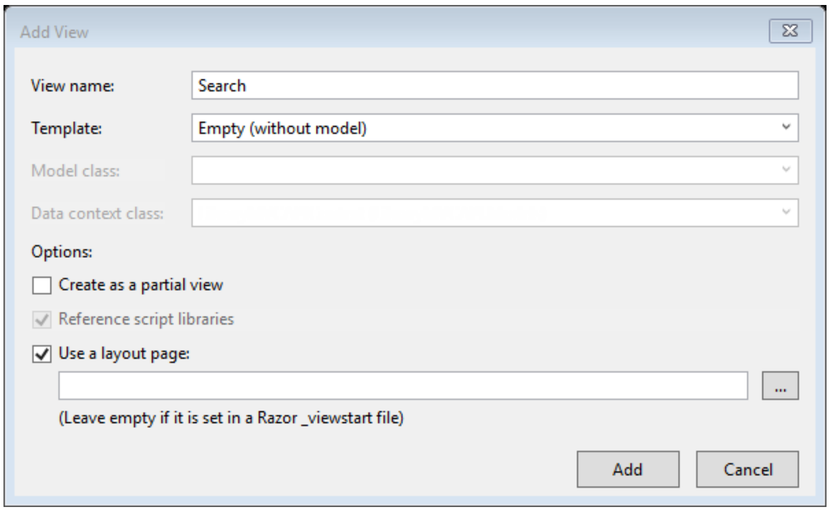
```
// GET: api/Authors/5
[ResponseType(typeof(AuthorDTO))]
0 references | 0 requests | 0 exceptions
public async Task<IHttpActionResult> GetAuthor(int id)
{
    Author a = await db.Authors.FindAsync(id);
    if (a == null)
    {
        return NotFound();
    }

    AuthorDTO author = new AuthorDTO
    {
        Id = a.Id,
        FirstName = a.FirstName,
        LastName = a.LastName,
        Books = a.Books.Select(b => new BookDTO()
        {
            Id = b.Id,
            Title = b.Title,
            Publisher = b.Publisher
        }
                              ).ToList()
    };

    return Ok(author);
}
```

vii) Build and run the application and navigate to the **api/Authors** and **api/Books** URLs.

IV - Using Ajax to interface with the Web API

We will now implement some Ajax functionality into our application using jQuery. We will create a dedicated Search view for Books and use Ajax to make HTTP GET requests to the Web API interface we have created for Books.

i) Create a new **Search** View for Books. Right-click on the **Views/Books** folder and choose to add a new View as follows:



Add View

View name: Search

Template: Empty (without model)

Model class:

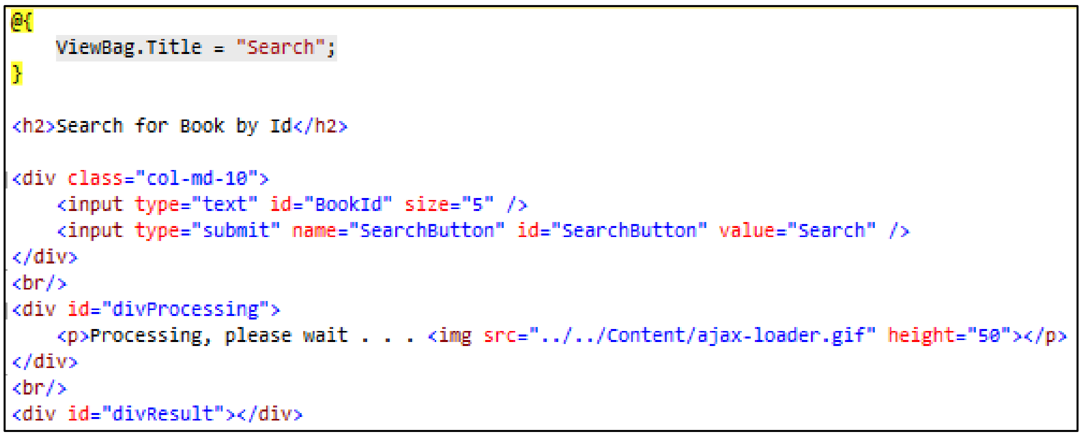
Data context class:

Options:

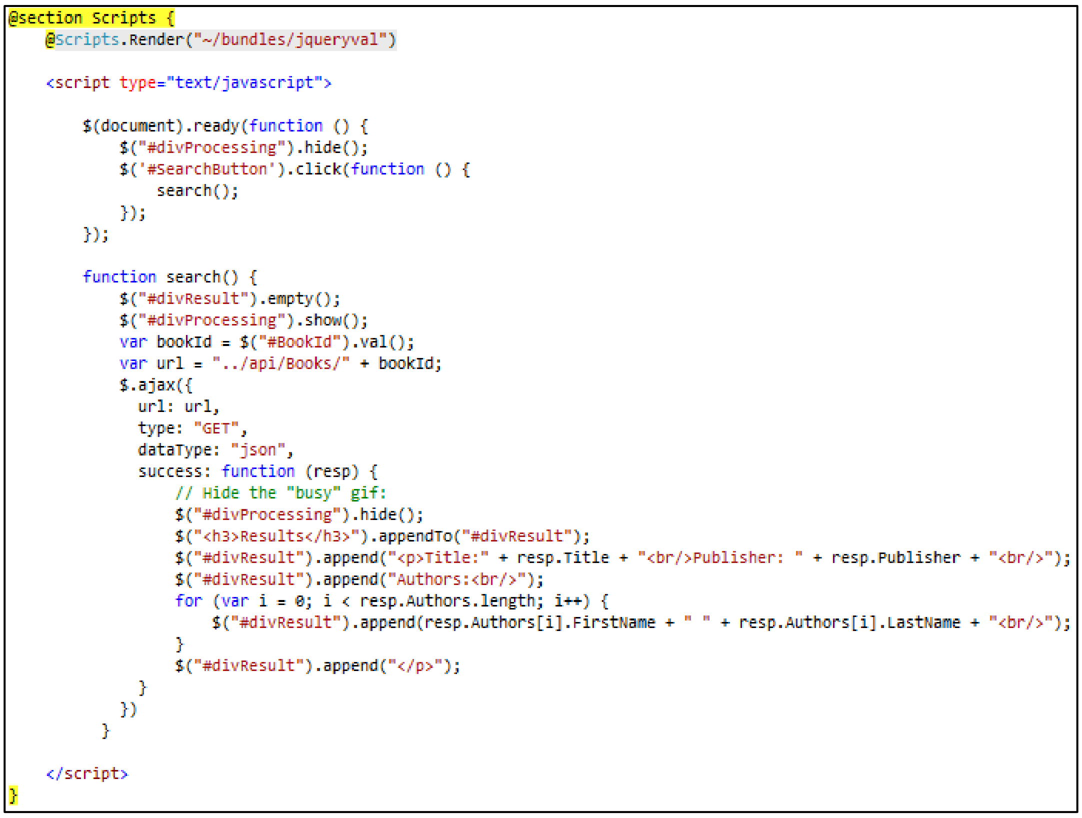
- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

ii) Modify the **Search.cshtml** code as follows:



```
@{
    ViewBag.Title = "Search";
}

<h2>Search for Book by Id</h2>

<div class="col-md-10">
    <input type="text" id="BookId" size="5" />
    <input type="submit" name="SearchButton" id="SearchButton" value="Search" />
</div>
<br/>
<div id="divProcessing">
    <p>Processing, please wait . . . <img src="../../Content/ajax-loader.gif" height="50"></p>
</div>
<br/>
<div id="divResult"></div>
```

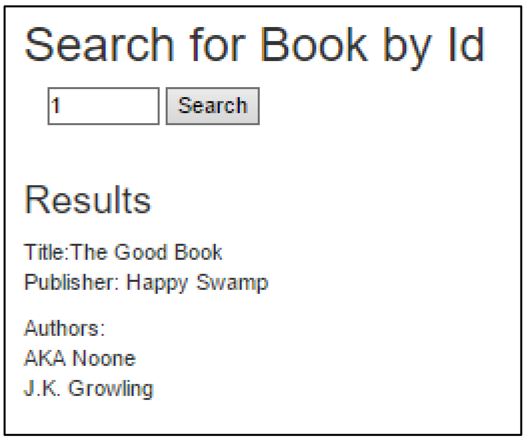


```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")

    <script type="text/javascript">

        $(document).ready(function () {
            $("#divProcessing").hide();
            $('#SearchButton').click(function () {
                search();
            });
        });

        function search() {
            $("#divResult").empty();
            $("#divProcessing").show();
            var bookId = $("#BookId").val();
            var url = "../api/Books/" + bookId;
            $.ajax({
              url: url,
              type: "GET",
              dataType: "json",
              success: function (resp) {
                  // Hide the "busy" gif:
                  $("#divProcessing").hide();
                  $("<h3>Results</h3>").appendTo("#divResult");
                  $("#divResult").append("<p>Title:" + resp.Title + "<br/>Publisher: " + resp.Publisher + "<br/>");
                  $("#divResult").append("Authors:<br/>");
                  for (var i = 0; i < resp.Authors.length; i++) {
                      $("#divResult").append(resp.Authors[i].FirstName + " " + resp.Authors[i].LastName + "<br/>");
                  }
                  $("#divResult").append("</p>");
              }
            })
          }

    </script>
}
```

iii) Modify the MVC BooksController class to include the following method:

```
// GET: Books/Search
public ActionResult Search()
{
    return View();
}
```

iv) Build and run the application. Navigate to the **Books/Search** URL and test.



Search for Book by Id

1 Search

Results

Title:The Good Book
Publisher: Happy Swamp

Authors:
AKA Noone
J.K. Growling

V - Modification of the *ApplicationUser* class to incorporate additional attributes into user accounts for the application

The *ApplicationUser* class definition is found in the **Models/IdentityModels.cs** file. We can modify this class using a Code-First approach to add new properties to the user model. The associated database tables associated with the *ApplicationUser* class are accessed via the *ApplicationDbContext* class (which is also defined in the **Models/IdentityModels.cs** file).

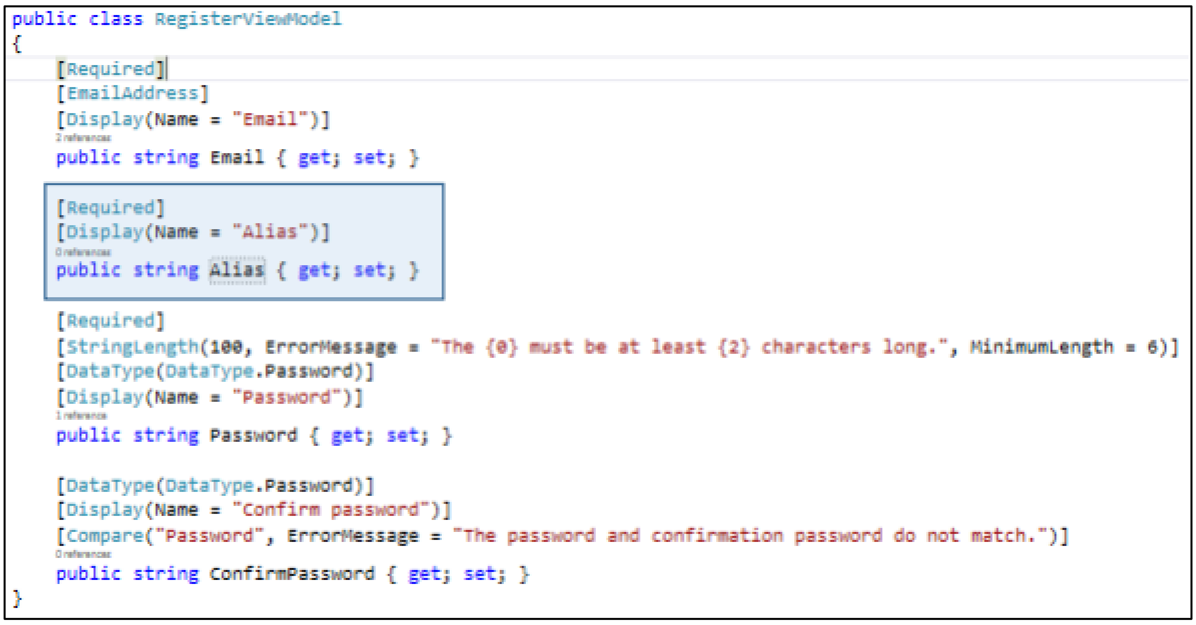
We have created our own database context (i.e., *LibraryMVCAPIContext*) class to access the database associated with the *Author* and *Book* entity classes that we have created. We have also previously run the **Enable-Migrations** command targeting the *LibraryMVCAPIContext*.

We will now need to enable migrations on the *ApplicationDbContext* class. We do not, however, want to overwrite the migration files that have already been created for the *LibraryMVCAPIContext* class. In order to keep the migration files associated with the *ApplicationDbContext* separate from the migration files associated with the *LibraryMVCAPIContext*, we will pass an extra parameter to the **Enable-Migrations** command to specify that the migration files should be created in a new folder.

i) Modify the *ApplicationUser* class to contain a new property called *Alias* as follows:

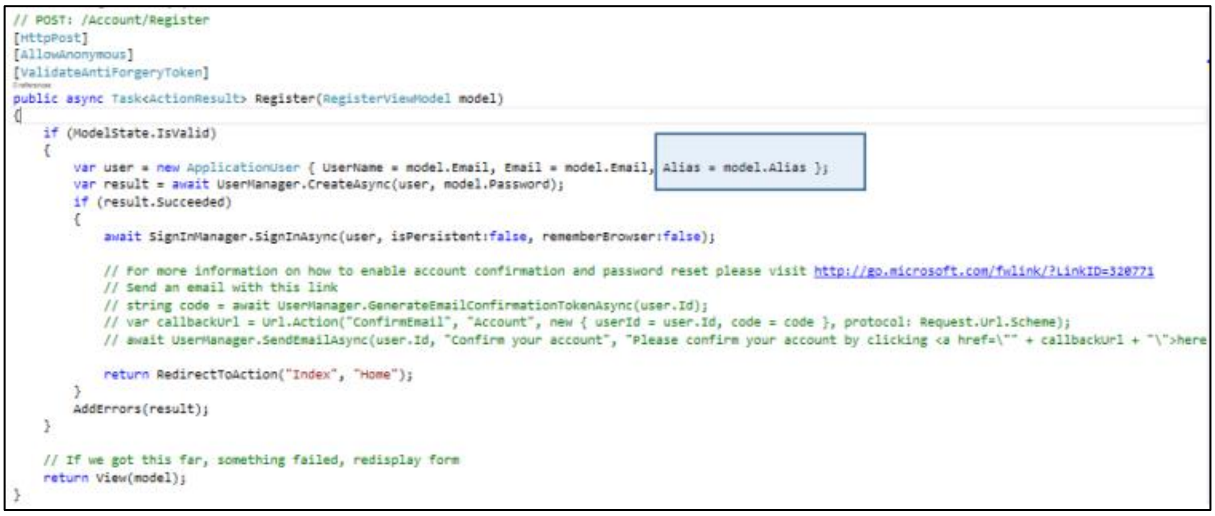
```
public class ApplicationUser : IdentityUser
{
    public string Alias { get; set; }
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenticationTypes.ApplicationCookie);
        // Add custom user claims here
        return userIdentity;
    }
}
```

ii) In the **AccountViewModels.cs** code file, modify the *RegisterViewModel* class so that it contains a property called *Alias*.



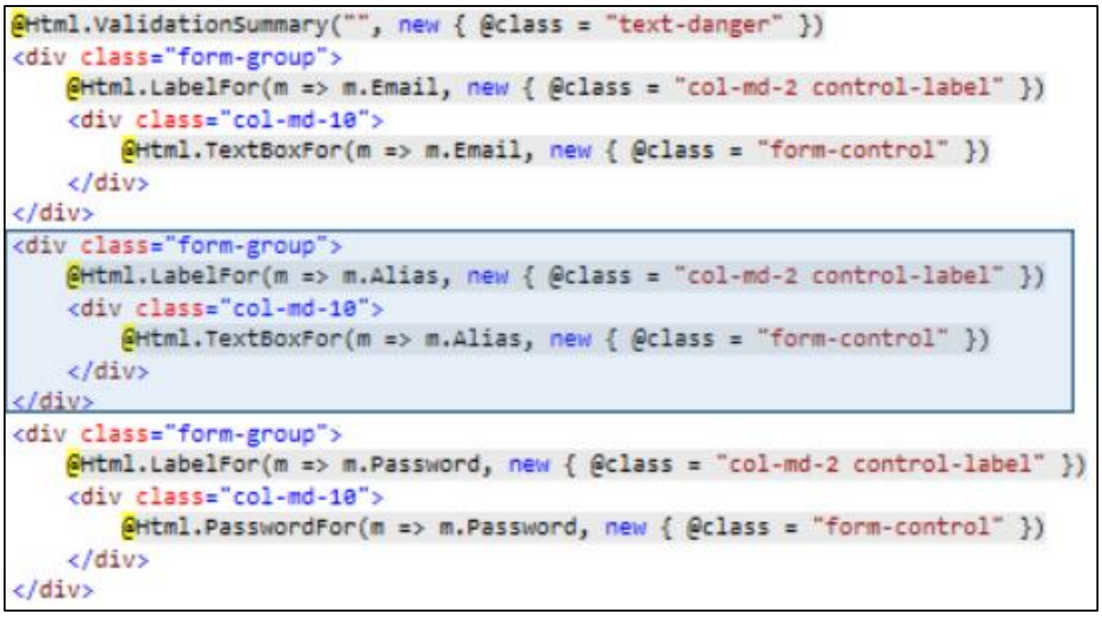
```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    2 references
    public string Email { get; set; }

    [Required]
    [Display(Name = "Alias")]
    0 references
    public string Alias { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    1 reference
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    0 references
    public string ConfirmPassword { get; set; }
}
```

iii) Modify the *AccountController Register* method to include the new Alias information when creating an *ApplicationUser*.



```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email, Alias = model.Alias };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

            // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkID=320771
            // Send an email with this link
            // string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            // var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
            // await UserManager.SendEmailAsync(user.Id, "Confirm your account", "Please confirm your account by clicking <a href=\"" + callbackUrl + "\">here

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

iv) Modify the **Views/Account/Register.cshtml** view to include an input box for the Alias text.



```
@Html.ValidationSummary("", new { @class = "text-danger" })
<div class="form-group">
    @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.Alias, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Alias, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
    </div>
</div>
```

v) Run the **Enable-Migrations** command for the *ApplicationDBContext* passing a parameter to specify that migration files should be created in the **Migrations\ApplicationDbContext** folder:

```
PM> Enable-Migrations -ContextTypeName ApplicationDbContext -MigrationsDirectory Migrations\ApplicationDbContext
Checking if the context targets an existing database...
Code First Migrations enabled for project LibraryMVCAPI.
```

vi) Add a new migration for the *ApplicationDbContext* by running the **Add-Migration** command passing a parameter that specifies the *Configuration* type *LibraryMVCAPI.Migrations.ApplicationDbContext.Configuration*:

```
PM> Add-Migration AddAliasField -ConfigurationTypeName LibraryMVCAPI.Migrations.ApplicationDbContext.Configuration
Scaffolding migration 'AddAliasField'.
The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to
calculate the changes to your model when you scaffold the next migration. If you make additional changes to your model that
 you want to include in this migration, then you can re-scaffold it by running 'Add-Migration AddAliasField' again.
```

vii) Run the **Update-Database** command passing a parameter that specifies the *Configuration* type *LibraryMVCAPI.Migrations.ApplicationDbContext.Configuration*:

```
PM> Update-Database -ConfigurationTypeName LibraryMVCAPI.Migrations.ApplicationDbContext.Configuration
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201612040359174_AddAliasField].
Applying explicit migration: 201612040359174_AddAliasField.
Running Seed method.
```

viii) Run the application and Register a new user.

ix) View the **AspNetUsers** table data and verify that the Alias field data is being successfully stored.

x) Modify the default Identity for your own project.