Web API 2 Resource Toolkit

Contents

Getting Started

Introduction

Web API 2 is a standard approach to create a RESTful API. It has a lot of features that ease the development process compare to Web API. Web API 2 introduces new features which are:

1. Attribute Routing
   Routing configuration can be done by using attribute, which is more declarative and easy to maintain compare to conventional-based routing, where route configuration is done in one configuration method.
2. Cross Origin Resource Sharing – CORS
   Web API 2 support resource sharing through AJAX call that is requested from a different domain.
3. OWIN self hosting
   OWIN process can be hosted independently from IIS server so it's not tightly coupled. The recommendation of OWIN implementation is Katana.
4. IHttpActionResult Return Type
   In the first Web API version, we can only return an object type or void type, later web api will process that object and convert it to IHttpResponseMessage. Now in Web API 2, we can return IHttpAction, which is more flexible, because we can specify the http status code, and message as we want.

Create Project

To create a web API project, do the following steps:

1. In the menu bar, choose **File > New > Project** (Ctrl + Shift + N)
2. In new project dialog, choose **Templates > Visual C# > Web** in the left side. Then, choose **ASP.NET Web Application** template and rename the project to "**WebAPITutorial".** After that, click **Ok**.
3. Choose **Empty** template and then add **Web API** for folder and core references. Then, click **Ok**.

Routing

What is Routing?

Routing is a process to direct/route an incoming request to request handler. In Web API 2, a request is handle by a method called action method, that is located in a class called controller class. We need to specify a route (path in URL) to the right action method. This route configuration can be done in two ways, first in a conventional-based routing, or attribute routing. In conventional-based routing, a route configuration is defined in a route register method, in the other hand, attribute routing use attribute to define a route configuration.

How to Add Route

To add a controller, do the following steps:

1. In Solution Explorer, expand the **WebAPITutorial** project.
2. Right-click the controller folder, Choose **Add > Controller…**.
3. Choose **Web API 2 Controller – Empty**, and then click **Add**.
4. Name the Controller **MovieController**, and then click **Add**.
5. Write the following code, to add new action methods:

```
List<string> movies = new List<string>
{
    "Captain America - Civil War",
    "X-Men: Apocalypse",
    "Suicide Squad",
    "Me Before You"
};

public List<string> GetMovies()
{
    return movies;
}

public string GetMovie(int id)
{
    if(0 <= id && id < movies.Count)
        return movies[id];

    return "-";
}

public List<string> GetSearch(string movieName)
{
    return movies.Where(movie => movie.Contains(movieName)).ToList();
}
```

To access the prior action method, do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5)
2. After a browser has opened, add the following path in the URL respectively:
   http://localhost:xxx/api/movie

http://localhost:xxx/api/movie/0
http://localhost:xxx/api/movie/4
http://localhost:xxx/api/movie/Captain

3. It will show the following result respectively

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>Captain America - Civil War</string>
  <string>X-Men: Apocalypse</string>
  <string>Suicide Squad</string>
  <string>Me Before You</string>
</ArrayOfstring>
```

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Captain
America - Civil War</string>
```

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">-</string>
```

```
<Error>
  <Message>The request is invalid.</Message>
  <MessageDetail>
    The parameters dictionary contains a null entry for parameter 'id' of non-
nullable type 'System.Int32' for method 'System.String GetMovie(Int32)' in
'WebAPITutorial.Controllers.MovieController'. An optional parameter must be a
reference type, a nullable type, or be declared as an optional parameter.
  </MessageDetail>
</Error>
```

How to Configure Route

In the example before, the first three route works because there is a default route configuration that's located in WebApiConfig.cs. But the last path doesn't work because there is no matching route configuration suitable for it.

To add a route configuration, do the following steps:

1. In Solution Explorer, open **WebApiConfig.cs** file
2. Add the following code in **Register** method after this code: **`config.MapHttpAttributeRoutes();`**

```
config.Routes.MapHttpRoute(
    name: "SearchMovieApi",
    routeTemplate: "api/Movie/{movieName}",
    defaults: new { controller = "Movie", action = "GetSearch" }
);
```

In the above example, we add a route that map http://localhost:xxx/api/Movie/{movieName} to the MovieController class and GetSearch action method. To access the new route, do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5) to redeploy the application

2. After that a browser will be opened with our web site (e.g. http://localhost: 3640)
3. Add the following path in the URL: http://localhost:xxx/api/movie/Captain
4. It will show the following result

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-
instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Array
s">
<string>Captain America - Civil War</string>
</ArrayOfstring>
```

How to Add Attribute Routing

To add a new controller for attribute routing example, do the following steps:

1. To ensure the route attribute configuration has been enabled, in Solution Explorer open **WebApiConfig.cs**
2. Ensure the following line of code has been added to the Register method

```
public static void Register(HttpConfiguration config)
{
    ...
    config.MapHttpAttributeRoutes(); // <--ensure this line has been added
    ...
}
```

3. In Solution Explorer, right-click the controller folder, Choose **Add > Controller…**.
4. Choose **Web API 2 Controller – Empty**, and then click **Add**.
5. Name the Controller **DirectorController**, and then click **Add**.
6. Write the following code inside DirectorController class:

```
private List<string> directors = new List<string>
{
    "Jack Kirby",
    "Joe Simon",
    "Jojo Moyes"
};

[HttpGet, Route("api/director")]
public List<string> FindAll()
{
    return directors;
}

[HttpGet, Route("api/ director /{directorName}")]
public List<string> Search(string directorName)
{
    return directors.Where(director => director.Contains(directorName)).ToList();
}
```

To access our service, we can navigate the browser to `/director` and `/director/xxx` where `xxx` is the name of director that we want to search. The code above use Route attribute to specify the access path to the action method. As you can notice, we annotate every action method with Route attribute and it has director prefix.

Instead of adding director prefix in every action method, we can specify the prefix with a RoutePrefix attribute as a class metadata. Below is the example of DirectorController class with RoutePrefix attribute.

```
[RoutePrefix("api/director")]
public class DirectorController : ApiController
{

    private List<string> directors = new List<string>
    {
        "Jack Kirby",
        "Joe Simon",
        "Jojo Moyes"
    };

    [HttpGet, Route("")]
    public List<string> FindAll()
    {
        return directors;
    }

    [HttpGet, Route("{directorName}")]
    public List<string> Search(string directorName)
    {
        return directors.Where(director => director.Contains(directorName)).ToList();
    }
}
```

As the name implied, route prefix will add the route prefix to all action method that is only declared in that class. In our example, we want to create an api service, so there will be another controller that have an “api” prefix for it. To add a global route prefix, perform the following steps:

1. Create a new class of type IDirectRouteProvider or its subclass.
2. In solution explorer, right click the project name and choose **Add > Class**
3. Name it GlobalRoutePrefixProvider
4. Write the following code inside **GlobalRoutePrefixProvider.cs** to override the GetRoutePrefix method to add a global prefix.

```
public class GlobalRoutePrefixProvider : DefaultDirectRouteProvider
{
    private string globalPrefix;

    public GlobalRoutePrefixProvider(string globalPrefix)
    {
        this.globalPrefix = globalPrefix;
    }

    protected override string GetRoutePrefix(HttpControllerDescriptor
controllerDescriptor)
    {
        var prefix = base.GetRoutePrefix(controllerDescriptor);
        if (prefix == null) return globalPrefix;
        return string.Format("{0}/{1}", globalPrefix, prefix);
    }
}
```

5. In solution explorer, open WebApiConfig.cs file inside App_Start folder
6. Use our GlobalRoutePrefixProvider by changing the following code, from

```
config.MapHttpAttributeRoutes();
```

To

```
config.MapHttpAttributeRoutes(new GlobalRoutePrefixProvider("api"));
```

7. Finally, open DirectorController.cs and remove the word api inside the RoutePrefix attribute.

How to Add Route Constraints

Route constraint is only used to limit the possible value of a route parameter. Below is the syntax how to write route constraint:

`{route-parameter:constraint(constraint-parameter?)}`

To add route constraint, do the following steps:

1. In Solution Explorer, open **DirectorController.cs** file
2. Write the following code to add minimum length constraint to search action method

```
[HttpGet, Route("{directorName:minlength(3)}")]
public List<string> Search(string directorName)
{
    return directors.Where(director => director.Contains(directorName)).ToList();
}
```

3. Write the following code to add new action method with integer route constraint:

```
[HttpGet, Route("{id:int}")]
public string FindById(int id)
{
    return directors[id];
}
```

To test our route constraint, do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5) to redeploy the application
2. After a browser has opened, add the following path in the URL respectively:
   http://localhost:xxx/api/Director/Jac
   http://localhost:xxx/api/Director/1
   http://localhost:xxx/api/Director/J
   Then, It will show the following result respectively

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>Jack Kirby</string>
</ArrayOfstring>
```

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Joe
Simon</string>
```

```
<Error>
  <Message>
    No HTTP resource was found that matches the request URI
'http://localhost:3640/api/director/J'.
  </Message>
  <MessageDetail>
    No action was found on the controller 'Director' that matches the request.
  </MessageDetail>
</Error>
```

The last example shows an error because the minimum character in the route parameter is 3 character

How to Add Custom Route Constraints

We can also specify our custom route constraint, for a specify purpose (e.g. SSN constraint). To add a custom constraint, do the following steps:

1. In Solution Explorer, right-click **WebAPITutorial** project, then choose **Add > New Folder**
2. Name it **Constraints**
3. Right-click **Constraints** Folder, then choose **Add > New Item…**.
4. In the side bar, Choose Visual C# > Code and then choose **Class**.
5. Name the class **SSNConstraint**, then click Ok.
6. Write the following code inside SSNConstraint class

```
public class SSNConstraint : IHttpRouteConstraint
{
    public bool Match(HttpRequestMessage request, IHttpRoute route, string
parameterName, IDictionary<string, object> values, HttpRouteDirection
routeDirection)
    {
        object value = null;
        var result = values.TryGetValue(parameterName, out value);

        if (result == false) return false;

        var ssn = Convert.ToString(value);
        var pattern = @"\d{3}-\d{2}-\d{4}";
        return Regex.Match(ssn, pattern).Success;
    }
}
```

7. To register ‘ssn’ as our new custom constraint class, replace the following code:
   before:

```
config.MapHttpAttributeRoutes(new GlobalRoutePrefixProvider("api"));
```

   after:

```
var constraintResolver = new DefaultInlineConstraintResolver();
var directRouteProvider = new GlobalRoutePrefixProvider("api");
```

```
constraintResolver.ConstraintMap["ssn"] = typeof(SSNConstraint);

config.MapHttpAttributeRoutes(constraintResolver, directRouteProvider);
```

To test our new custom constraint, we need to change our director data first so that it includes ssn information. To create a class that will encapsulate director data, do the following steps:

1. Ensure the **Models** folder has been created in the root project directory.
2. Right-click **Models** folder, then choose **Add > Class**
3. Name it **Director.cs**
4. Write the following code inside Director.cs

```
public class Director
{

    public int Id { get; set; }
    public string Name { get; set; }
    public string SSN { get; set; }

}
```

After that, we need to add an action route that get a director by his/her SSN. Do the following steps:

1. Add the following code to add field of Director list

```
private List<Director> directors2 = new List<Director>
{
    new Director { Id = 1, Name = "Jack Kirby", SSN = "123-12-1234" },
    new Director { Id = 2, Name = "Joe Simon", SSN = "234-23-2345" },
    new Director { Id = 3, Name = "Jojo Moyes", SSN = "345-34-3456" },
};
```

2. Add a new route action to get a director by his/her SSN with the following code:

```
[HttpGet, Route("ssn/{ssn:ssn}")]
public Director FindBySSN(string ssn)
{
    return directors2.FirstOrDefault(director => director.SSN == ssn);
}
```

To test our route constraint, do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5) to redeploy the application
2. After a browser has opened, add the following path in the URL respectively:
   http://localhost:xxx/api/Director/ssn/123-12-1234
   http://localhost:xxx/api/Director/ssn/abc-ab-abcd
   Then It will show the following result respectively

```
<Director xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebAPITutorial.Models">
  <Id>1</Id>
  <Name>Jack Kirby</Name>
  <SSN>123-12-1234</SSN>
</Director>
```

```
Http Error 404.0 – Not Found
```

The last example shows a http not found because the ssn number in url path doesn't match with the custom constraint rule that we made.

Integrating with Entity Framework

How to add Controller with actions, using Entity Framework

To create a controller with actions being generated based on entity framework model. This actions will cover create, read, update and delete action. To do so, do the following steps:

1. In Solution Explorer, right-click **Models** folder and choose **Add > Class**
2. Name the class **Movie**, then write the following code inside

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
}
```

3. In Solution Explorer, right-click **Controllers** folder and choose **Add > Controller**
4. Choose Web API 2 Controller with actions, using Entity Framework
5. In Model Class combo box, choose **Movie** class that we just created.
6. In Data Context class click **plus icon button** in the right to create new data context class
7. Rename the class to **TutorialDbContext**, then click Add button
8. Click Add button.

How to Seed Database using Migration

To seed the database with some data, do the following steps:

1. In Toolbar, choose Tools > NuGet Package Manager > **Package Manager Console**
2. Type **`Enable-Migrations`** in it.
3. To seed movie and director data, write the following code in **Migrations > Configurations**

```
var joeRusso = new Director { Id = 1, Name = "Joe Russo" };
var anthonyRusso = new Director { Id = 2, Name = "Anthony Russo" };
var bryanSinger = new Director { Id = 3, Name = "Bryan Singer" };

var captainAmerica = new Movie {
    Id = 1,
    Title = "Captain America: Civil War",
    ReleaseDate = new DateTime(2016, 4, 27)
};

var xmen = new Movie
{
    Id = 2,
    Title = "X-Men: Apocalypse",
    ReleaseDate = new DateTime(2016, 5, 19)
};

context.Directors.AddOrUpdate(director => director.Id, joeRusso, anthonyRusso);
context.Movies.AddOrUpdate(movie => movie.Id, captainAmerica, xmen);
```

4. In Package Manager Console, type "**Add-Migration InitialCreate"** command to create a table based on existing model.

5. In Package Manager Console, type "**Update-Database**" command to run the migration configuration and seed method.

To test our MovieController with the data that we provided do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5) to redeploy the application
2. After a browser has opened, Add the following path in the URL:
   http://localhost:xxx/api/movies
   http://localhost:xxx/api/movies/1
   Then, it will show the following result respectively:

```
<ArrayOfMovie xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebAPITutorial.Models">
  <Movie>
    <Id>1</Id>
    <ReleaseDate>2016-04-27T00:00:00</ReleaseDate>
    <Title>Captain America: Civil War</Title>
  </Movie>
  <Movie>
    <Id>2</Id>
    <ReleaseDate>2016-05-19T00:00:00</ReleaseDate>
    <Title>X-Men: Apocalypse</Title>
  </Movie>
</ArrayOfMovie>
```

```
<Movie xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebAPITutorial.Models">
  <Id>2</Id>
  <ReleaseDate>2016-05-19T00:00:00</ReleaseDate>
  <Title>X-Men: Apocalypse</Title>
</Movie>
```

How to Add Relation Between Classes

In current example, there is no relation between directors and movies. In this scenario, the relation between director and movie is many-to-many relationship, each director can have many movies, and each movie can have many directors. To add a many-to-many relationship between those two classes, do the following steps:

1. In Solution Explorer, open Movie.cs file
2. Add the following property inside Movie class

```
public ICollection<Director> Directors { get; set; }
```

3. In Solution Explorer, open Director.cs file
4. Add the following property inside Director class

```
public ICollection<Movie> Movies { get; set; }
```

5. In Package Manager Console, write "**Add-Migration MovieDirectorRelation**" command to add a new migration that will update our table structure base on updates that has been made.

6. In Solution Explorer, open **Configuration.cs** file under Migrations folder, and write the following code at the of Seed method

```
context.Movies.First(movie => movie.Id == 1).Directors = new List<Director> {
joeRusso, anthonyRusso };
context.Movies.First(movie => movie.Id == 2).Directors = new List<Director> {
bryanSinger };
```

7. In Package Manager Console, write "**Update-Database**" command to update our table structure

How to Use DTO Pattern

If we want to access all movie including their directors, we could end with serialization exception because of a cyclic reference from movie to director, and director to movie. One way to prevent that is by using Data Transfer Object pattern. Using that pattern, we can separate entity model from presentation model. To use DTO pattern, do the following steps:

1. In Solution Explorer, right-click project name, and choose Add > New Folder, and name it DTOs
2. Right-click DTOs folder, and choose Add > Class. Then name it MovieDTO.cs
3. Write the following code inside it:

```
public int Id { get; set; }
public string Title { get; set; }
public DateTime ReleaseDate { get; set; }
public List<string> DirectorNames { get; set; }
```

4. Open MoviesController.cs file under Controllers folder, and change GetMovies method from:

```
public IQueryable<Movie> GetMovies()
{
    return db.Movies;
}
```

To:

```
public IQueryable<MovieDTO> GetMovies()
{
    return db.Movies.Select(movie => new MovieDTO
    {
        Id = movie.Id,
        Title = movie.Title,
        ReleaseDate = movie.ReleaseDate,
        DirectorNames = movie.Directors.Select(director => director.Name).ToList()
    });
}
```

To access GetMovies action method do the following steps:

1. In Menu Bar, choose **Debug > Start Without Debugging** (Ctrl+F5) to redeploy the application
2. After a browser has opened, Open http://localhost:xxx/api/movies in the URL
   Then it will show the following results:

```
<ArrayOfMovieDTO xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebAPITutorial.DTOs">
```

```
  <MovieDTO>
    <DirectorNames
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
      <d3p1:string>Joe Russo</d3p1:string>
      <d3p1:string>Anthony Russo</d3p1:string>
    </DirectorNames>
    <Id>1</Id>
    <ReleaseDate>2016-04-27T00:00:00</ReleaseDate>
    <Title>Captain America: Civil War</Title>
  </MovieDTO>
  <MovieDTO>
    <DirectorNames
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
      <d3p1:string>Bryan Singer</d3p1:string>
    </DirectorNames>
    <Id>2</Id>
    <ReleaseDate>2016-05-19T00:00:00</ReleaseDate>
    <Title>X-Men: Apocalypse</Title>
  </MovieDTO>
</ArrayOfMovieDTO>
```

How to Use Attribute Validation

Attribute validation is a feature to validate the property of object based on attribute that is decorated in that property. Using attribute validation, we can validate payload object in our action method. To add an attribute validation, do the following steps:

1. In solution explorer, open Movie.cs file under Models folder
2. Write the following code inside:

```
public int Id { get; set; }
[Required, MinLength(5)]
public string Title { get; set; }
[Range(typeof(DateTime), "01/01/1970", "01/01/2020")]
public DateTime ReleaseDate { get; set; }

public ICollection<Director> Directors { get; set; }
```

To test our attribute validation, issue a POST HTTP request to http://localhost:xxx/api/movies with the following request body:

```
Title = Andi
RealeaseDate = 01/01/1900
```

Then it will result in the following response:

```
<Error>
  <Message>The request is invalid.</Message>
  <ModelState>
    <movie.Title>The field Title must be a string or array type with a minimum length of
'5'.</movie.Title>
    <movie.ReleaseDate>The field ReleaseDate must be between 1/1/1970 12:00:00 AM and
1/1/2020 12:00:00 AM.</movie.ReleaseDate>
  </ModelState>
</Error>
```

How to Add Custom Attribute Validation

To create a custom attribute validation, do the following steps:

1. In Solution Explorer, right-click the project name and choose **Add > New Folder**, name it Validations
2. Right-click Validations folder choose **Add > Class**, name it MovieTitleAttribute
3. Write the following code inside

```
public class MovieTitleAttribute : ValidationAttribute
{

    public override bool IsValid(object value)
    {
        //custom business logic
        return Convert.ToString(value).Contains(":");
    }

    public override string FormatErrorMessage(string name)
    {
        return "Movie title must contains colon character";
    }

}
```

4. Open **Movie.cs** file under Models folder and add MovieTitle attribute to Title property.

To test our custom attribute validation, issue a POST HTTP request to http://localhost:xxx/api/movies with the following request body:

```
Title = Andi the conqueror
ReleaseDate = 31/05/2016
```

Then it will result in the following response:

```
<Error>
  <Message>The request is invalid.</Message>
  <ModelState>
    <movie.Title>Movie title must contains colon character</movie.Title>
    <movie.ReleaseDate>The field ReleaseDate must be between 1/1/1970 12:00:00 AM and
1/1/2020 12:00:00 AM.</movie.ReleaseDate>
  </ModelState>
</Error>
```

Filter, Authorization, and Authentication

What Is Filters

Filter is class that act as an interceptor. It can intercept incoming request, transform the response, and handle exception. It is useful to separate the main business method with other cross cutting concern method, for example logging, security, etc.

How to Add Custom Action Filter Attribute

In this example, we will learn how to add a custom filter to log a duration that is needed for a method to run. To do so, do the following steps:

1. In Solution Explorer, right click project name and choose **Add > New Folder**. Name it **Filters**
2. Right-click Filters folder and choose Add > Class, name it ProfillingFilter
3. Write the following code inside ProfillngFilter

```
public class ProfillingFilter : ActionFilterAttribute
{
    private Stopwatch timer = new Stopwatch();
    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        timer.Restart();
    }

    public override void OnActionExecuted(HttpActionExecutedContext context)
    {
        Debug.WriteLine(string.Format("Method: {0} takes: {1}ms",
            context.ActionContext.ActionDescriptor.ActionName,
            timer.ElapsedMilliseconds));
    }
}
```

4. Open MoviesController.cs file and add ProfillingFilter Attribute to MovieController class

```
[ProfillingFilter]
public class MoviesController : ApiController
```

To test our filter, start our application in debug mode, and the access http://localhost:xxx/api/movies in the URL. After that in **Output panel**, it will print the elapsed time it took for a request to MovieController class.

How to Add Global Filter

In the example before, we already learn how to create a filter attribute and put it in every class to apply the filter. If we want to apply a filter to all web API controller, do the following steps:

1. In Solution Explorer, open **WebApiConfig.cs** file
2. Add the following code in **`Register`** method

```
config.Filters.Add(new ProfillingFilter());
```

How to Add Authorization Filter

Using filter, we can secure our web API from unauthorized access. This filter will check the credential for each request. If it's authorized then the request will continue, if not it will return unauthorized access. To use this filter, do the following steps:

1. In Solution Explorer, open MoviesController.cs file
2. Add Authorized attribute at MovieController class

```
[Authorize]
public class MoviesController : ApiController
```

If we access http://localhost:xxx/api/movies now, it will return the following error message

```
<Error>
  <Message>Authorization has been denied for this request.</Message>
</Error>
```

How to Add Basic Authentication Filter

To enable user to access an authorized resource, we need to provide a user to enter their credentials. In this example we will use basic authentication protocol for user to enter their username and password. To do so, we need to create a filter that checks authorization header for each incoming request. If it's not authorized, then we will ask user to enter their username and password. If they enter a valid credential, then the request is permitted.

To add basic authentication, do the following steps:

1. In Solution Explorer, right-click **Filters** folder and choose Add > Class
2. Name it BasicAuthenticationAttribute and then write the following code inside

```
public class BasicAuthenticationAttribute : ActionFilterAttribute,
IAuthenticationFilter
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string Realm { get; set; }

    public BasicAuthenticationAttribute(string username, string password, string
realm = "WebApiTutorial")
    {
        this.Username = username;
        this.Password = password;
        this.Realm = realm;
    }

    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        var user = GetBasicUser(actionContext);
        if(user == null || user.Username != this.Username || user.Password !=
this.Password)
        {
            actionContext.Response = new
HttpResponseMessage(HttpStatusCode.Unauthorized);
```

```
            actionContext.Response.Headers.Add("WWW-Authenticate",
string.Format("Basic realm=\"{0}\"", this.Realm));
        }
    }

    public async Task AuthenticateAsync(HttpAuthenticationContext context,
CancellationToken cancellationToken)
    {
        var user = GetBasicUser(context.ActionContext);
        if (user != null && user.Username == this.Username && user.Password ==
this.Password)
        {
            var claim = new Claim(ClaimTypes.Name, user.Username);
            var identity = new ClaimsIdentity(new List<Claim> { claim }, "basic");
            context.Principal = new ClaimsPrincipal(identity);
        }
    }

    public async Task ChallengeAsync(HttpAuthenticationChallengeContext context,
CancellationToken cancellationToken)
    {
        context.Result = new BasicChallengeResult(context.Result, this.Realm);
    }

    private BasicUser GetBasicUser(HttpActionContext context)
    {
        var auth = context.Request.Headers.Authorization;
        if (auth != null && auth.Scheme == "Basic")
        {
            var credential =
Encoding.ASCII.GetString(Convert.FromBase64String(auth.Parameter)).Split(':');

            return new BasicUser
            {
                Username = credential[0],
                Password = credential[1]
            };
        }
        return null;
    }

    private class BasicUser {
        public string Username { get; set; }
        public string Password { get; set; }
    }

    private class BasicChallengeResult : IHttpActionResult
    {
        private IHttpActionResult result;
        private string realm;

        public BasicChallengeResult(IHttpActionResult result, string realm)
        {
            this.result = result;
            this.realm = realm;
        }
```

```
        public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken
cancellationToken)
        {
            var response = await result.ExecuteAsync(cancellationToken);
            if(response.StatusCode == HttpStatusCode.Unauthorized)
                response.Headers.Add("WWW-Authenticate", string.Format("Basic
realm=\"{0}\"", this.realm));
            return response;
        }
    }
}
```

3. Open MoviesController.cs and add **BasicAutheticationAttribute** on MovieController class like the following code

```
[BasicAuthentication(username: "kenrick", password: "satrio")]
[Authorize]
public class MoviesController : ApiController
```

To test our basic authentication filter, run the application and access http://localhost:xxx/api/movies in the URL. At first, it will pop-up a windows to enter username and password. If we provide the right credentials, the window will not pop-up for the following request.

What is Owin

Owin is a standard middleware application that makes web server and web application loosely coupled. Using Owin we can authenticate incoming request using a standard OAuth 2.0 protocol.

How to Use Authentication with Owin

First we need to setup the identity database, that is used to store user data, roles, login and etc. To query those that, will need to create another repository class. For this example, will only need to Register the user and find the user by their credentials. To do so do the following steps:

1. In Package Manager Console, write the following command:

```
Install-Pacakge Microsoft.AspNet.Identity.EntityFramework –Version 2.2.1
```

2. In Solution Explorer, right click Models Folder and choose Add > Class
3. We will create a new db context context for authorization purpose. This class will extend from IdentityDbContext which will create user and role table if it's not exists. To do so, write the following code:

```
public class AuthDbContext : IdentityDbContext<IdentityUser>
{
    public AuthDbContext() : base("TutorialDbContext")
    { }
}
```

4. In Solution Explorer, right-click DTOs folder and choose Add > Class.
5. Name it UserDTO, then write the following code inside:

```
public class UserDTO
{
    [Required]
    public string Username { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Required]
    [Compare("Password")]
    public string ConfirmPassword { get; set; }
}
```

6. In Solution Explorer, right-click project name folder and choose **Add > New Folder**. Name it **Repositories**.
7. Right-click Repositories folder, choose Add > Class.
8. Name it UserRepository, then write the following code inside:

```
public class AuthRepository : IDisposable
{
    private AuthContext context;

    private UserManager<IdentityUser> userManager;

    public AuthRepository()
    {
        context = new AuthContext();
        userManager = new UserManager<IdentityUser>(new
UserStore<IdentityUser>(context));
    }

    public async Task<IdentityResult> RegisterUser(UserDTO userModel)
    {
        return await userManager.CreateAsync(new IdentityUser
        {
            UserName = userModel.Username
        }, userModel.Password);
    }

    public async Task<IdentityUser> FindUser(string username, string password)
    {
        return await userManager.FindAsync(username, password);
    }

    public void Dispose()
    {
        context.Dispose();
        userManager.Dispose();
    }
}
```

After that we need to create an oauth authorization provider class. This class is responsible to validate the client credential. If it is valid, then it will return a bearer token. This is token will be used to authorize each

request. After creating this class, we need to configure the owin to use this class as a middleware. To create that class, do the following steps:

1. In Package Manager Console, write the command to install Owin library

```
Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.2.3

Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.1

Install-Package Microsoft.Owin.Security.OAuth –Version 2.1.0
```

2. In Solution Explorer, right-click the project and choose **Add > New Folder**. Name it **Providers**
3. Right-click the Providers folder, and choose Add > Class. Name it **SimpleAuthorizationServerProvider**. Then write the following code inside:

```
public class SimpleAuthorizationServerProvider : OAuthAuthorizationServerProvider
{
    public async override Task
ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
    {
        context.Validated();
    }

    public async override Task
GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
    {
        using (var repo = new AuthRepository())
        {
            var user = await repo.FindUser(context.UserName, context.Password);
            if (user == null)
            {
                context.SetError("invalid grant", "user not found");
                return;
            }

            var identity = new ClaimsIdentity(context.Options.AuthenticationType);
            context.Validated(identity);
        }
    }
}
```

4. In Solution Explorer, right-click **App_Start** folder then choose Add > Class
5. Named it Startup, then write the following code inside

```
[assembly: OwinStartup(typeof(Startup))]
namespace OwinPractice.App_Start
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            ConfigureOAuth(app);
            HttpConfiguration configuration = new HttpConfiguration();
            WebApiConfig.Register(configuration);
            app.UseWebApi(configuration);
        }
```

```
        private void ConfigureOAuth(IAppBuilder app)
        {
            var option = new OAuthAuthorizationServerOptions
            {
                AllowInsecureHttp = true,
                TokenEndpointPath = new PathString("/api/Account/Token"),
                AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
                Provider = new SimpleAuthorizationServerProvider()
            };

            app.UseOAuthAuthorizationServer(option);
            app.UseOAuthBearerAuthentication(new
OAuthBearerAuthenticationOptions());
        }
    }
}
```

In the above steps, we have created two class. The first class, SimpleAuthorizationServerProvider, we override two methods. The first method, `ValidateClientAuthentication`, is used to validate the client id. Since this is tow legged authorization, then we can validate the request without client id. The second method, `GrantResourceOwnerCredentials`, is used to check request credential, whether the use exists or not. In the second class, we create a class to configure owin oauth middleware.

To enable user to register their account to our system, we need to create a new user resource controller, that have register action method. To do so, do the following steps:

1. In Solution Explorer, right-click Controllers folder and choose Add > Controller
2. Choose Web API 2 Controller – Empty, then click Add
3. Name it AccountController, then click Add
4. Write the following code inside:

```
[RoutePrefix("Account")]
public class AccountController : ApiController
{
    private AuthRepository repo;

    public AccountController()
    {
        repo = new AuthRepository();
    }

    [HttpPost, Route("")]
    public async Task<IHttpActionResult> Register(UserDTO user)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        return Ok(await repo.RegisterUser(user));
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
            repo.Dispose();
        base.Dispose(disposing);
```

```
    }
}
```

To register a user to system, issue a HTTP POST to http://localhost:xxx/api/Account request with the following body data:

```
username=admin&
password=SuperPass&
confirmPassword=SuperPass
```

Then it will result in the following response

```
{
  "succeeded": true,
  "errors": []
}
```

How to Authorize User

To authorize each http request using owin authorization, do the following steps:

1. Issue HTTP POST request to http://localhost:xxx/api/Account/Token with the following request body:
   ```
   username = admin
   password = SuperPass
   grant_type = password
   ```
2. Then it will result in the following response

   ```
   {
     "access_token":
   "iBJe0Ln6P3hdzBVJ6NV4BRb6l1hwgaHA7iASdpMCmWgFxAvb6wTNeB12mKe0o5h_fx97Aj2CXtuUGPyhm
   FJX4dJXHjwazX4-
   ocCxEJUYm1UrHlHSb1Ql88IsTKGGnPxf6xz3puKjcZxPi9iURdG2_JsTWtZsuGrjmYzupFY_GmB8WC2So8
   QrP3LMhlcHFGOIa1azQjfYMTSngFzZsASi2w",
     "token_type": "bearer",
     "expires_in": 86399
   }
   ```

   We will use this access_token as our header data in the future request.
3. In Solution Explorer, open MovieController file under Controller folder
4. Remove BasicAuthentication attribute from MovieController class
5. Issue HTTP GET request to http://localhost:xxx/api/Movies with the following request header:
   ```
   Authorization = bearer
   iBJe0Ln6P3hdzBVJ6NV4BRb6l1hwgaHA7iASdpMCmWgFxAvb6wTNeB12mKe0o5h_fx97Aj2CXt
   uUGPyhmFJX4dJXHjwazX4-
   ocCxEJUYm1UrHlHSb1Ql88IsTKGGnPxf6xz3puKjcZxPi9iURdG2_JsTWtZsuGrjmYzupFY_Gm
   B8WC2So8QrP3LMhlcHFGOIa1azQjfYMTSngFzZsASi2w
   ```

How to Authorize User with a Specific Role

In the previous example, we have already secure our resource from anonymous access. After authorization process succeed, the resource will be available to anyone. If we want to grant access to some resource for a specific user, we can use a role based authorization.

To do role based authorization, first we need to create a repository class that will manage role resource. Then we need to provide a method to add the role to a specific user. After that, whenever a user is authorized we

need to save their roles to identity object. Finally, we need to use role property in authorized attribute. To do so, do the following steps:

1. In Solution Explorer, right-click **Repositories** folder and choose Add > Class
2. Name it RoleRepository.cs, then write the following code inside:

```
public class RoleRepository : IDisposable
{
    private AuthDbContext context;
    private RoleManager<IdentityRole> roleManager;

    public RoleRepository()
    {
        context = new AuthDbContext();
        roleManager = new RoleManager<IdentityRole>(new
RoleStore<IdentityRole>(context));
    }

    public async Task<IdentityRole> FindById(string roleId)
    {
        return await roleManager.FindByIdAsync(roleId);
    }

    public async Task<IdentityRole> FindByRole(string roleName)
    {
        return await roleManager.FindByNameAsync(roleName);
    }

    public async Task<IdentityResult> Create(string roleName)
    {
        return await roleManager.CreateAsync(new IdentityRole
        {
            Name = roleName
        });
    }

    public void Dispose()
    {
        roleManager.Dispose();
        context.Dispose();
    }
}
```

3. Open AuthRepository.cs file under Repositories folder
4. Add "**AddRole"** method inside that class:

```
public async Task<IdentityResult> AddRole(string username, string role)
{
    var user = await userManager.FindByNameAsync(username);
    return await userManager.AddToRoleAsync(user.Id, role);
}
```

5. Open AccountController.cs file inside and then add the following filed

```
private RoleRepository roleRepository;
```

6. In AccountController.cs add a new action method to add a role for user. To do that add the following code inside AccountController class

```
[HttpPut, Route("{userName}/role")]
public async Task<IdentityResult> AddRole(string userName, [FromBody] string
roleName)
{
    var role = await roleRepository.FindByRole(roleName);
    if (role == null)
    {
        var result = await roleRepository.Create(roleName);
    }
    return await repo.AddRole(userName, roleName);
}
```

7. Open SimpleServerAuthorizationServerProvider.cs file and change GrantResourceOwnerCredentials method to the code below:

```
public async override Task
GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
{
    using (var repo = new AuthRepository())
    {
        var user = await repo.FindUser(context.UserName, context.Password);
        if (user == null)
        {
            context.SetError("invalid grant", "user not found");
            return;
        }

        var identity = new ClaimsIdentity(context.Options.AuthenticationType);

        using (var roleRepo = new RoleRepository())
        {
            foreach(var userRole in user.Roles)
            {
                var role = await roleRepo.FindById(userRole.RoleId);
                identity.AddClaim(new Claim(ClaimTypes.Role, role.Name));
            }
            context.Validated(identity);
        }
    }
}
```

8. Finally, we need add role property to authorized attribute in MovieController class

```
[Authorize(Roles = "User")]
public class MoviesController : ApiController
```

To test our role based authorization, there are several steps to follow:

1. We need to update user **admin** to have a specific role, by issuing HTTP PUT request to http://localhost:xxx/api/Account/admin/role,
   with the following request body:
   =User
   and the following request header:
   Content-Type=application/x-www-form-urlencoded

2. Authorize a user by issuing HTTP POST request to http://localhost:xxx/api/Account/token, with the following request body:
   username=admin&
   password=SuperPass&
   grant_type=password

   Then it will result in the following response:


```
{
  "access_token": "lZDgzUkeAIjea4KWfbS7q-HUZRaoktjb0ORmOsSpOLUlC_Kc8kX1Q-JMx7RuN-
ekiVHKcQSm1giDfaYS9rhXbuvd5zdDaeK9ohIKaTt-Z2FTEQ-6SsArh3u3VvYIPUVR31Z2i7biR3l_0-
NLp6T0pl1LpSJErwbFlaJfmvK1ciIECtHUkal0DjonUy02r84QXKtzUItesmQ2s9brF3lJ9JmOfVUN0gu
VT1DqYVPEhVy9ouPUMwdTEfl3sEiDDRi0IIVoj7R-9hm5kPnMSBE2mw",
  "token_type": "bearer",
  "expires_in": 86399
}
```


3. Using the given access_token, Issue HTTP GET request to http://localhost:xxx/api/Movies with the following request header:
   Accept=application/json
   Authorization=bearer lZDgzUkeAIjea4KWfbS7q-HUZRaoktjb0ORmOsSpOLUlC_Kc8kX1Q-JMx7RuN-ekiVHKcQSm1giDfaYS9rhXbuvd5zdDaeK9ohIKaTt-Z2FTEQ-6SsArh3u3VvYIPUVR31Z2i7biR3l_0-NLp6T0pl1LpSJErwbFlaJfmvK1ciIECtHUkal0DjonUy02r84QXKtzUItesmQ2s9brF3lJ9JmOfVUN0guVT1DqYVPEhVy9ouPUMwdTEfl3sEiDDRi0IIVoj7R-9hm5kPnMSBE2mw