

# Lesson 06

## Ajax in ASP.NET Core MVC apps



# ASP.NET Core

tongsreng TAL

ASP.NET Core

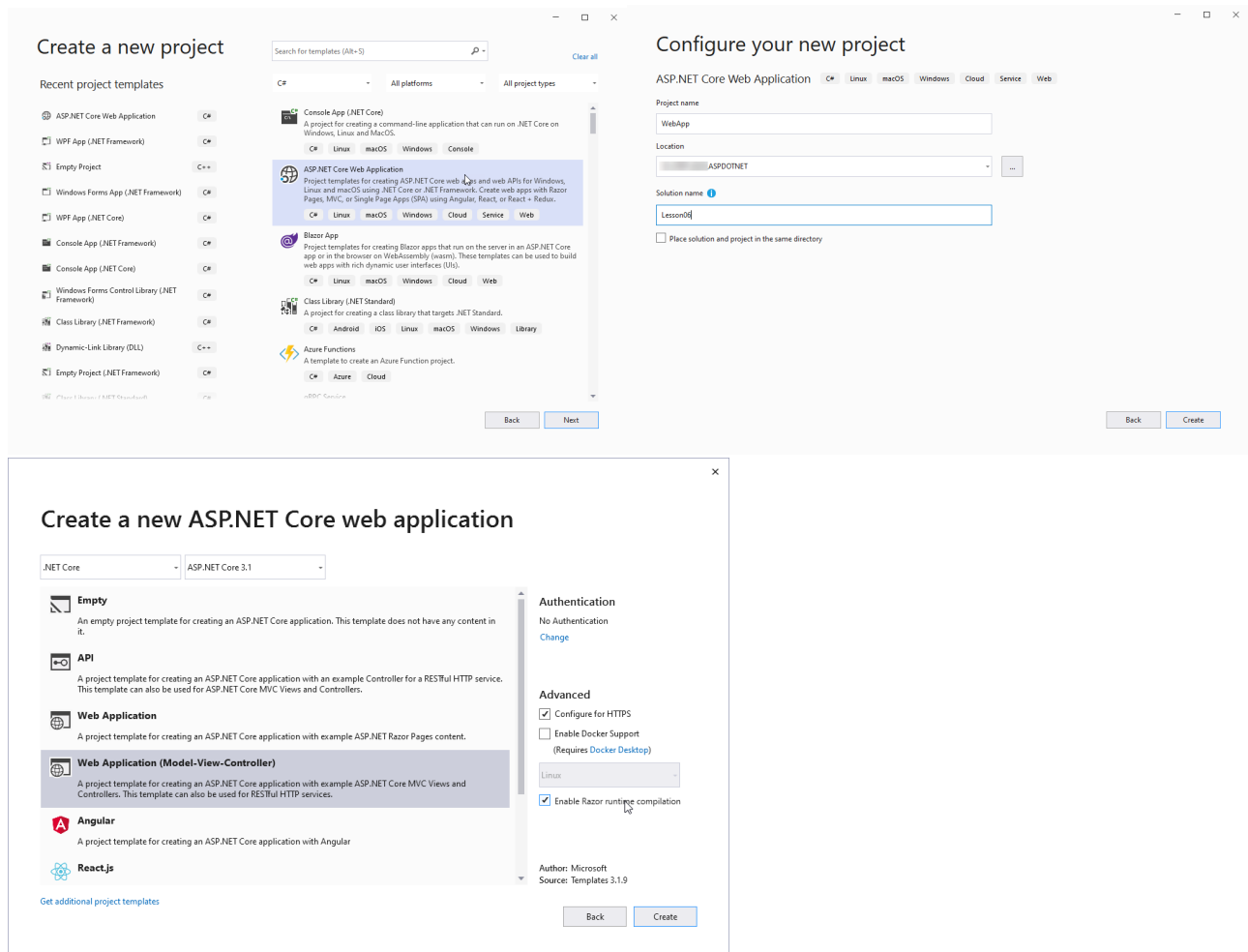
## Contents

1	Prepare for this lesson .....	2
1.1	Libraries used.....	2
1.2	Prepare Models.....	3
1.3	Seeding Data.....	4
1.4	Prepare Database .....	5
1.4.1	Prepare for model changes.....	9
1.4.2	Managing migrations.....	11
1.5	Check if data is inserted.....	11
1.6	Run the application .....	12
2	Ajax simple scenario .....	12
2.1	Excluding field from Json .....	15
2.2	Formatting DateTime.....	15
3	Response formatting .....	21
3.1	Format-specific Action Results.....	21
3.2	Content negotiation.....	22
3.2.1	The Accept header.....	23
3.2.2	Browsers and content negotiation .....	24
3.2.3	Configure formatters .....	24
3.2.4	Response format URL mappings.....	27
4	References .....	29

In this lesson we will discuss about using Ajax in View-Model-Controller Architecture in ASP.NET Core Web Application.

## 1 Prepare for this lesson

Create ASP.NET Core MVC project named "WebApp".









### 1.1 Libraries used

There are 5 more packages needed for this lesson:

1. Entity Framework Core packages:

- a. **Microsoft.EntityFrameworkCore**: for data manipulation with Database (s)
  - b. **Microsoft.EntityFrameworkCore.SqlServer**: for Data management with Database MS SQLServer
  - c. **Microsoft.EntityFrameworkCore.Tools**: for command line tools for migrations and other related Database management
2. EntityFrameworkCore.LazyLoadingProxies: for automatic loading related entities
  3. Microsoft.AspNetCore.Mvc.NewtonsoftJson: for conversion object to json and vice versa.

	<b>Microsoft.AspNetCore.Mvc.NewtonsoftJson</b> by Microsoft ASP.NET Core MVC features that use Newtonsoft.Json. Includes input and output formatters for JSON and JSON PATCH.	v3.1.8 v5.0.0-rc.1.20451.17
	<b>Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation</b> by Microsoft Runtime compilation support for Razor views and Razor Pages in ASP.NET Core MVC.	v3.1.8 v5.0.0-rc.1.20451.17
	<b>Microsoft.EntityFrameworkCore</b> by Microsoft Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.	v5.0.0-rc.1.20451.13
	<b>Microsoft.EntityFrameworkCore.Proxies</b> by Microsoft Lazy-loading proxies for Entity Framework Core.	v5.0.0-rc.1.20451.13
	<b>Microsoft.EntityFrameworkCore.SqlServer</b> by Microsoft Microsoft SQL Server database provider for Entity Framework Core.	v5.0.0-rc.1.20451.13
	<b>Microsoft.EntityFrameworkCore.Tools</b> by Microsoft Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	v5.0.0-rc.1.20451.13

## 1.2 Prepare Models

Add Classes Product.cs and Order.cs in folder Models:

```
//Product.cs
namespace WebApp.Models
{
    public class Product
    {
        public long ProductId { get; set; }
        public string Name { get; set; }
        public double Price { get; set; }
    }
}
```

```
//Order.cs:
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace WebApp.Models
{
    public class Order
    {
        public long OrderId { get; set; }
        public virtual ICollection<Product> Products { get; set; }
    }
}
```

```

        public long OrderNo { get; set; }
        public DateTime OrderDate { get; set; }
        [NotMapped]
        public double Total {
            get {
                double t = 0;
                if(Products != null)
                {
                    foreach (var p in Products) t += p.Price;
                }
                return t;
            }
        }
    }
}

```

Create Database context named "DataContext.cs" in folder "Models":

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Models
{
    public class DataContext:DbContext
    {
        public DataContext(DbContextOptions<DataContext> options) : base(options) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}

```

### 1.3 Seeding Data

We may want to insert some data when first creation of tables. To do this we override method OnModelCreating of DbContext class:

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Models
{
    public class DataContext:DbContext
    {
        public DataContext(DbContextOptions<DataContext> options) : base(options) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

```

```

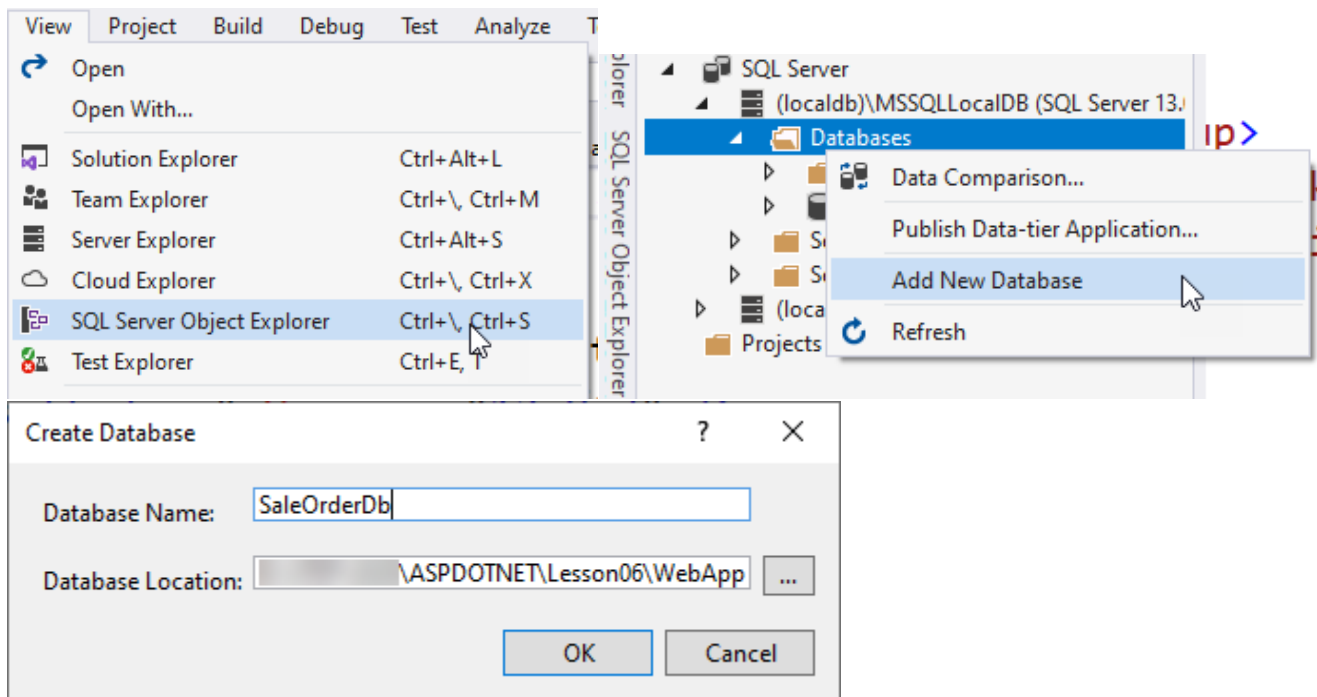
        Order order = new Order { OrderId = 1L, OrderDate = DateTime.Now, OrderNo = 1 }
;
        modelBuilder.Entity<Order>().HasData(
            order
        );
        modelBuilder.Entity<Product>().HasData(
            new { order.OrderId, ProductId = 1L, Name = "Watch", Price = 30.5D },
            new { order.OrderId, ProductId = 2L, Name = "Phone", Price = 270D },
            new { order.OrderId, ProductId = 3L, Name = "Shoes", Price = 55.75D }
        );
    }
}

```

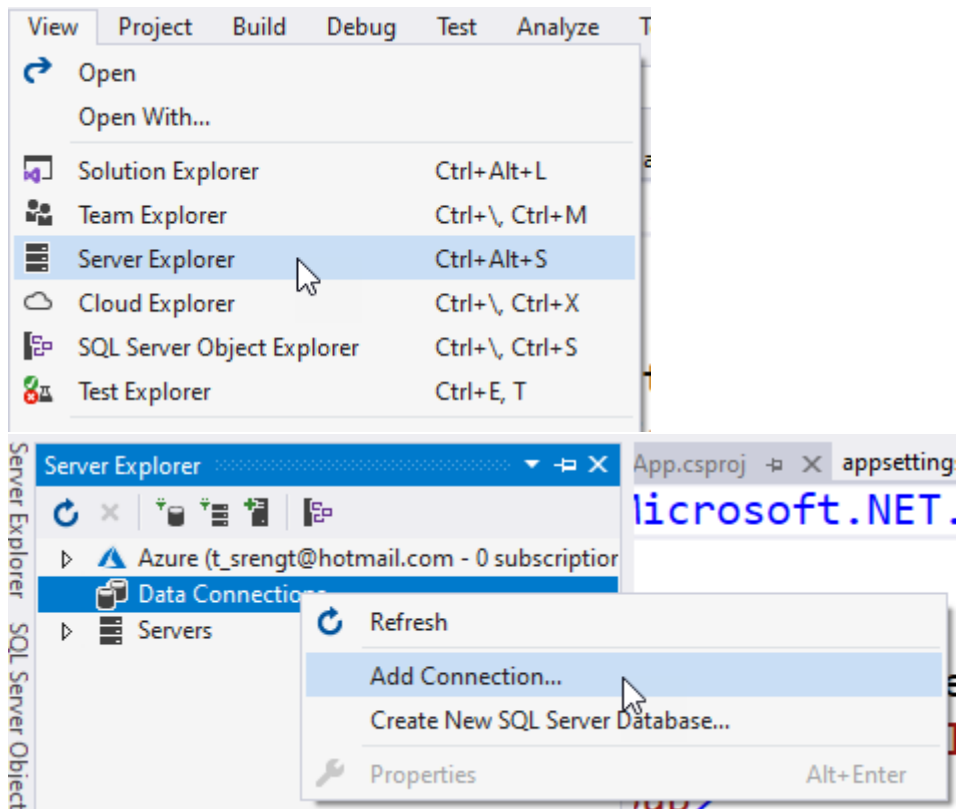
The listing above will add a new order into table Orders with OrderId=1, OrderDate = Now, OrderNo=1. And insert 3 products into this order.

## 1.4 Prepare Database

Create localDb named "SaleOrderDb.mdf" for development purpose only:



Click OK and you will see new file name "SaleOrderDb.mdf" in you selected folder, use Server Explorer to connect to it:



Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:  
 Microsoft SQL Server Database File (SqlClient) Change...

Database file name (new or existing):  
 PDOTNET\Lesson06\WebApp\SaleOrderDb.mdf Browse...

Log on to the server

☒ Use Windows Authentication  
☐ Use SQL Server Authentication

User name:   
 Password:   
☐ Save my password

Advanced...

Test Connection OK Cancel

Click test connection to see if it is possible to connect and then click OK.

After connection successfully, go Server Explorer and right click on connected SaleOrderDb and choose property and copy Connection String content to paste in appsettings.json below:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DevDb": "Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=<your_path>\\SaleOrderDb.mdf;Integrated Security=True;Connect Timeout=30"
  }
}
```

Where **<your\_path>** is your absolute location of database file.

Connect EntityFrameworkCore services to application in Startup.cs file:



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using WebApp.Models;

namespace WebApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<DataContext>(
                options=>options
                    .UseSqlServer(Configuration.GetConnectionString("DevDb"))
                    .UseLazyLoadingProxies()
            );
            services.AddControllersWithViews().AddNewtonsoftJson();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(

```

```

        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
}
}
}

```

In Listing above we also have added Newtonsoft.Json supports:

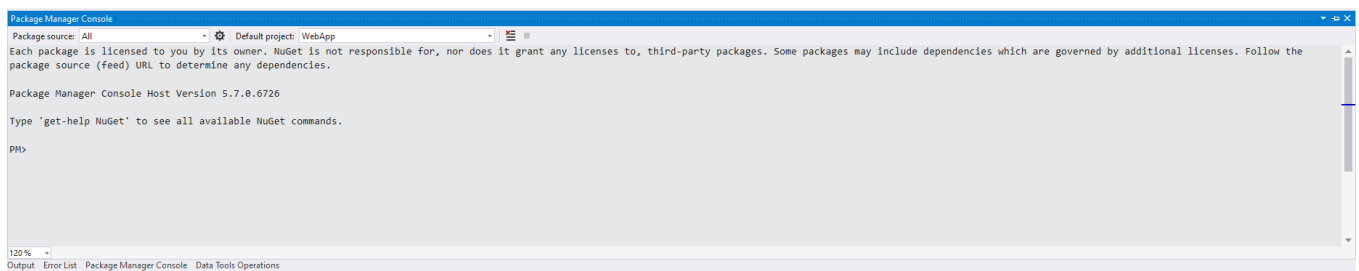
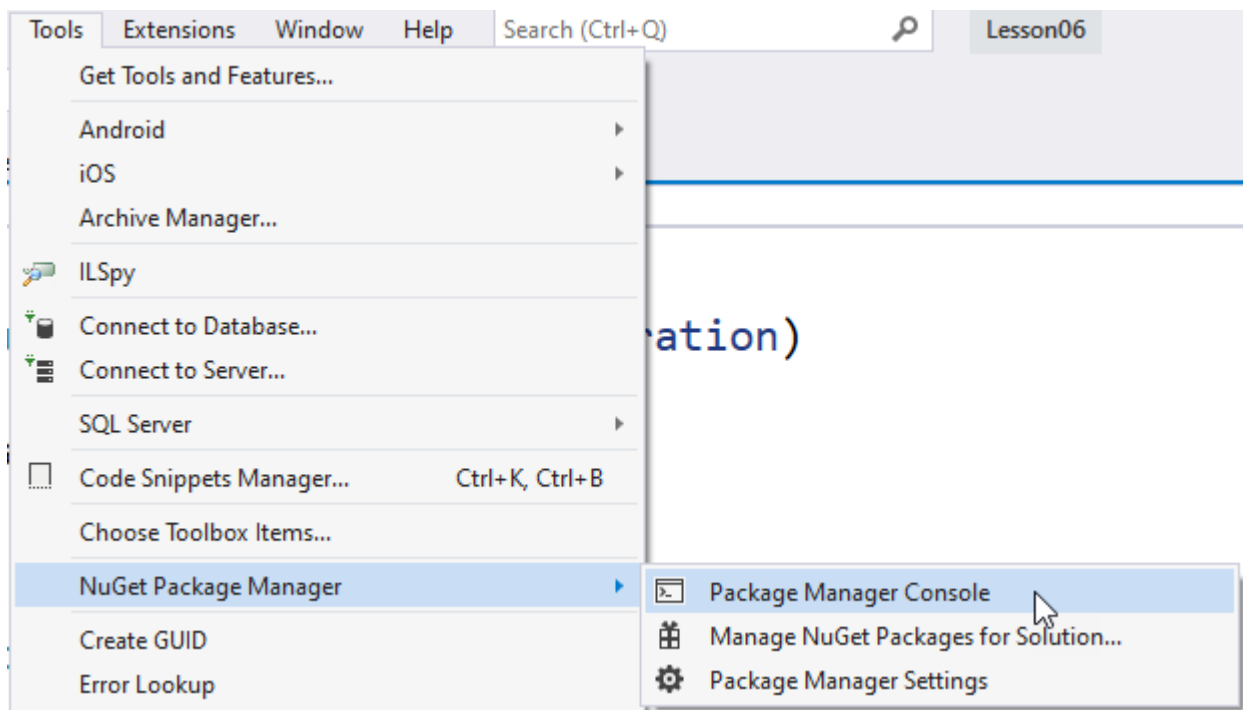
```

...
services.AddControllersWithViews().AddNewtonsoftJson();
...

```

#### 1.4.1 Prepare for model changes

Add-Migration to project using command by using Package Manager Console.



Change directory to Project folder (cd WebApp) if your solution have many projects.

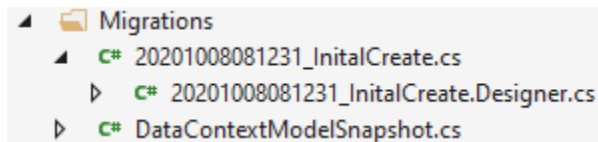
Available commands:

PMC Command	dotnet CLI command	Usage
<b>add-migration &lt;migration name&gt;</b>	Add <migration name>	Creates a migration by adding a migration snapshot.
<b>Remove-migration</b>	Remove	Removes the last migration snapshot.
<b>Update-database</b>	Update	Updates the database schema based on the last migration snapshot.
<b>Script-migration</b>	Script	Generates a SQL script using all the migration snapshots.

To make our first migration, run command:

Add-Migration InitialCreate

This command will create folder Migrations on success:



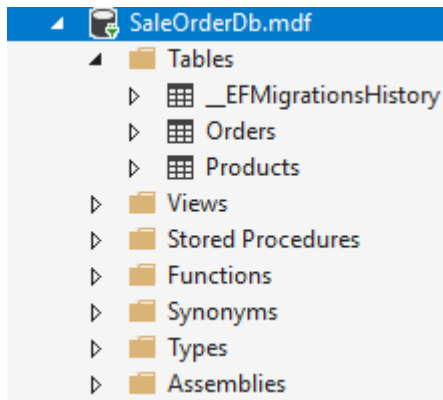
To generate schema we need to run migration. It can be done with command:

Update-Database

The output of the two commands above:

```
PM> Add-Migration InitialCreate
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Done.
```

We can verify by opening Server Explorer and refresh SaleOrderDb.mdf:



### 1.4.2 Managing migrations

Everytime your models changes you need to call:

Add-Migration <name>

Update-Database

If you think your **last migration** is wrong or missing fields/columns you can rollback:

Remove-Migration

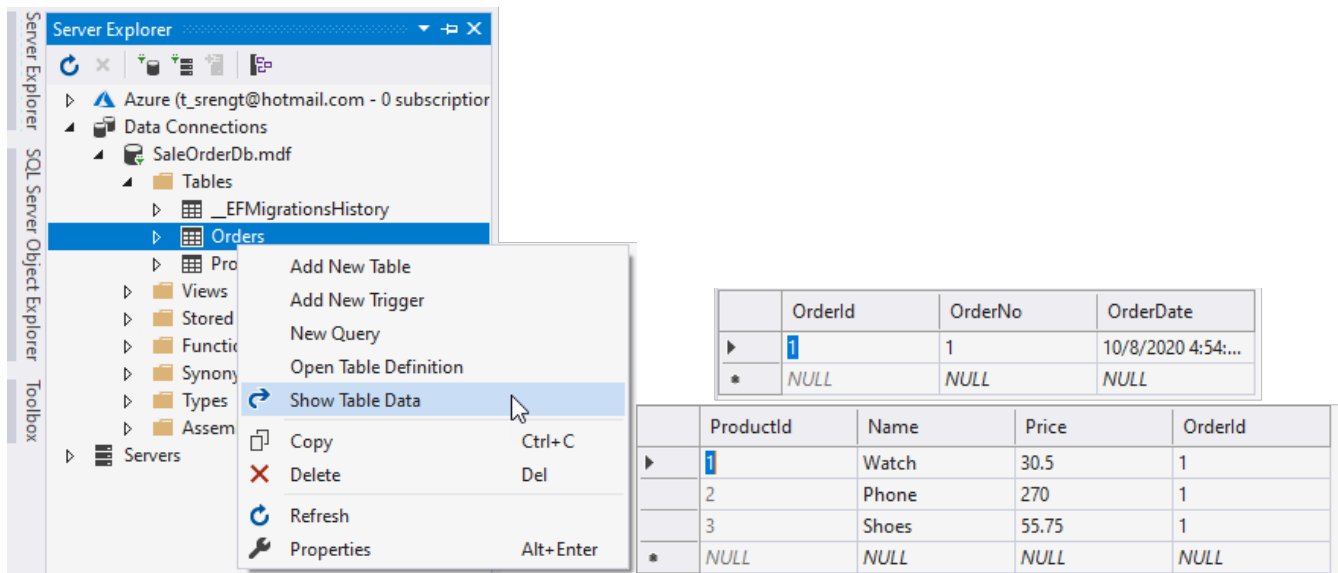
If you think the generated migration codes is not enough, you can edit it and add your queries.

There are sometimes you want to remove all migrations without losing your data, you can:

- Delete your **Migrations** folder
- Create a new migration and generate a SQL script for it
- In your database, delete all rows from the migrations history table
- Insert a single row into the migrations history, to record that the first migration has already been applied, since your tables are already there. The insert SEQL is the last operation in the SQL script generated above.

### 1.5 Check if data is inserted

You can use Server Explorer to check data of both tables:



## 1.6 Run the application

Build the project to see if everything is OK and run it.

## 2 Ajax simple scenario

Edit HomeController to use Dependency Injection to inject DataContext:

```
...
    private readonly DataContext dataContext;

    public HomeController(ILogger<HomeController> logger, DataContext context)
    {
        _logger = logger;
        dataContext = context;
    }
...

```

And response list of Orders in Index page:

```
...
    public IActionResult Index()
    {
        ViewData["Orders"] = new SelectList(dataContext.Orders, "OrderId", "OrderNo");
        return View();
    }
...

```

Add DropDownList in Home/Index.cshtml file:

```
...
    <p>
        <label for="Orders">Order: </label>
        @Html.DropDownList("Orders", "Select Order")
    </p>
...

```

Add Action ListProducts to list products in an order:

```
...
[HttpPost]
public async Task<IActionResult> ListProducts(long OrderId=0)
{
    Order order = await dbContext.Orders.FindAsync(OrderId);
    if(order==null) return Json(new object[] { });
    return Json(order.Products);
}
...
```

Edit Home/Index.cshtml to add scripts section for Ajax:

```
...
@section Scripts {
<script>
    $(function () {
        $("#Orders").change(function () {
            let orderId = $(this).val();
            $.post("@Url.Action("ListProducts")", { "OrderId": orderId },
                resp => {
                    console.log('resp', resp);
                    let tbl = $('<table class="table table-striped">'
                        + '<tr><th>Id</th><th>Name</th><th>Price</th></tr></table>');
                    for (let i = 0, p; i < resp.length; i++) {
                        p = resp[i];
                        tbl.append('<tr><td>' + p.productId + '</td><td>'
                            + p.name + '</td><td>'
                            + p.price + '</td></tr>');
                    }
                    $('#Products').empty().append(tbl);
                });
        });
    })
</script>
}
...
```

When you try to select an item in select box of Order numbers, it will do ajax request to /Home/ListProducts with OrderId=<selected item value>.

If you inspect the data sent and receive, you will see as listing below:

```
[
  {
    "productId": 1,
    "name": "Watch",
    "price": 30.5
  },{
    "productId": 2,
    "name": "Phone",
    "price": 270
  },{
```

```

        "productId": 3,
        "name": "Shoes",
        "price": 55.75
    }
]

```

You may notice the name of each property changed to camel Case naming conversion, we can change it. It supports 3 naming conversions:

- **CamelCaseNamingStrategy**: Change name case to camel case. Example *productId*
- **DefaultNamingStrategy**: keep name no change case. Example *ProductId*
- **SnakeCaseNamingStrategy**: Change name case to snake case. Example *product\_id*

To change naming strategy, update file Startup.cs as highlighted bold in listing below:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Newtonsoft.Json.Serialization;
using WebApp.Models;

namespace WebApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<DataContext>(
                options=>options
                    .UseSqlServer(Configuration.GetConnectionString("DevDb"))
                    .UseLazyLoadingProxies()
                    .EnableSensitiveDataLogging()
            );
            services.AddControllersWithViews().AddNewtonsoftJson(setupAction=>
            {
                setupAction.SerializerSettings.ContractResolver =
                    new DefaultContractResolver
                    { NamingStrategy = new DefaultNamingStrategy() };
            });
        }
    }
}

```

```

    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
}

```

## 2.1 Excluding field from Json

We can exclude some properties from generated Json string using `Newtonsoft.Json.JsonIgnoreAttribute`:

```

...
    [JsonIgnore]
    public virtual Order Order { get; set; }
...

```

## 2.2 Formatting DateTime

Add and action in HomeController called "ListOrders":

```

...
    [HttpPost]
    public async Task<IActionResult> ListOrders()
    {
        return Json(await DataContext.Orders.ToListAsync());
    }
...

```

Update code in Index.cshtml to load list of orders:

```

@{
    ViewData["Title"] = "Home Page";
}

```



```

}
@model ICollection<Order>
<div class="text-center">
  <h1 class="display-4">Welcome</h1>
  <p>
    <label for="Orders">Order: </label>
    @Html.DropDownList("Orders", "Select Order")
  </p>
  <p id="Products"></p>
  <button type="button" id="BtnListOrders">Load Orders</button>
  <p id="OrdersList"></p>
</div>

@section Scripts {
  <script>
    $(function () {
      $("#Orders").change(function () {
        let orderId = $(this).val();
        $.post("@Url.Action("ListProducts")", { "OrderId": orderId },
          resp => {
            console.log('resp', resp);
            let tbl = $('<table class="table table-striped">'
              + '<tr><th>Id</th><th>Name</th><th>Price</th></tr></table>');

            for (let i = 0, p; i < resp.length; i++) {
              p = resp[i];
              tbl.append('<tr><td>' + p.productId + '</td><td>'
                + p.name + '</td><td>' + p.price + '</td></tr>');
            }
            $('#Products').empty().append(tbl);
          });
    });

    $("#BtnListOrders").click(function () {
      $.post("@Url.Action("ListOrders")", resp => {
        console.log('resp', resp);
        let tbl = $('<table class="table table-striped">'
          + '<tr><th>Id</th><th>Order No.</th><th>Order Date</th></tr></table>');

        for (let i = 0, p; i < resp.length; i++) {
          p = resp[i];
          tbl.append('<tr><td>' + p.orderId + '</td><td>'
            + p.orderNo + '</td><td>' + p.orderDate + '</td></tr>');
        }
        $('#OrdersList').empty().append(tbl);
      });
    });
  </script>
}

```

When user click on Load Orders, user will see list of orders shown figure below:

# Welcome

Order:

Id	Order No.	Order Date
1	1	2020-10-08T16:54:00.7582188

You may notice the format of Order Date column, it is Newtonsoft.JSON DateTime format.

Let's see Default DateTime format without Newtonsoft.JSON by commenting out the AddNewtonSoftJson:

```
...
    services.AddControllersWithViews()/*.*.AddNewtonsoftJson(setupAction =>
    {
        setupAction.SerializerSettings.ContractResolver = new DefaultContractResolv
er
        { NamingStrategy = new CamelCaseNamingStrategy() };
        //{ NamingStrategy = new DefaultNamingStrategy() };
    })*;/
...
```

Restart the application and browse the page again.

You may see error something like this:

```
System.Text.Json.JsonException: A possible object cycle was detected which is not
supported. This can either be due to a cycle or if the object depth is larger than
the maximum allowed depth of 32.
    at
System.Text.Json.ThrowHelper.ThrowInvalidOperationException_SerializerCycleDetected(
Int32 maxDepth)
    at System.Text.Json.JsonSerializer.Write(Utf8JsonWriter writer, Int32
originalWriterDepth, Int32 flushThreshold, JsonSerializerOptions options,
WriteStack& state)
    at System.Text.Json.JsonSerializer.WriteAsyncCore(Stream utf8Json, Object value,
Type inputType, JsonSerializerOptions options, CancellationToken cancellationTok
en)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.SystemTextJsonResultExecutor.ExecuteAsync(Ac
tionContext context, JsonResult result)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.SystemTextJsonResultExecutor.ExecuteAsync(Ac
tionContext context, JsonResult result)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResultFilterAsync
```

```

>g__Awaited|29_0[TFilter,TFilterAsync](ResourceInvoker invoker, Task lastTask, State
next, Scope scope, Object state, Boolean isCompleted)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResultExecutedContex
tSealed context)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.ResultNext[TFilter,TFilterAs
ync](State& next, Scope& scope, Object& state, Boolean& isCompleted)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.InvokeResultFilters()
--- End of stack trace from previous location where exception was thrown ---
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>g__
_Awaited|24_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope,
Object state, Boolean isCompleted)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedCont
extSealed context)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Next(State& next,
Scope& scope, Object& state, Boolean& isCompleted)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeFilterPipelineAsync>g__
_Awaited|19_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope,
Object state, Boolean isCompleted)
    at
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Awaited|17_0
(ResourceInvoker invoker, Task task, IDisposable scope)
    at
Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|6_0(Endp
oint endpoint, Task requestTask, ILogger logger)
    at Microsoft.AspNetCore.Authorization.AuthorizationMiddleware.Invoke(HttpContext
context)
    at
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext
context)

```

#### HEADERS

=====

```

Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close
Content-Length: 0
Cookie: Phpstorm-e761a2b7=4723618a-6032-4140-8a47-d6bc7babb92d;
_ga=GA1.1.1697797399.1587198495; Phpstorm-d6b0d0b2=4723618a-6032-4140-8a47-
d6bc7babb92d
Host: localhost:44355
Referer: https://localhost:44355/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/86.0.4240.75 Safari/537.36 Edg/86.0.622.38
x-requested-with: XMLHttpRequest
origin: https://localhost:44355
sec-fetch-site: same-origin
sec-fetch-mode: cors
sec-fetch-dest: empty

```

...

To solve this problem, we need to break the cycle using `System.Text.Json.Serialization.JsonIgnoreAttribute`:

...

```
[JsonIgnore]
[System.Text.Json.Serialization.JsonIgnore]
public virtual Order Order { get; set; }
```

...

Restart the server and then click on button Load Orders.

# Welcome

Order:

Id	Order No.	Order Date
1	1	2020-10-08T16:54:00.7582188

The list is the same as Newtonsoft.Json library.

What if we want only to display the time or only the date?

We can by using custom DateTime format. Now create new class that inherit from abstract class `JsonConverter<T>`:

```
internal class DateJsonConverter : JsonConverter<DateTime>
{
    public override DateTime ReadJson(JsonReader reader, Type objectType, DateTime existingValue, bool hasExistingValue, JsonSerializer serializer)
    {
        return DateTime.ParseExact(reader.ReadAsString(),
            "dd/MM/yyyy", CultureInfo.InvariantCulture);
    }

    public override void WriteJson(JsonWriter writer, DateTime value, JsonSerializer serializer)
    {
        writer.WriteValue(value.ToString(
            "dd/MM/yyyy", CultureInfo.InvariantCulture));
    }
}
```

Then, uncomment the AddNewtonsoftJson:

```
...
    services.AddControllersWithViews().AddNewtonsoftJson(setupAction =>
    {
        setupAction.SerializerSettings.ContractResolver = new DefaultContractResolv
er
        { NamingStrategy = new CamelCaseNamingStrategy() };
        setupAction.SerializerSettings.Converters.Add(new DateJsonConverter());
    });
...
```

Restart application and click Load Orders:

# Welcome

Order:

Id	Order No.	Order Date
1	1	08/10/2020

It is also possible to apply format on a specific field for example:

In Startup.cs comment out the line:

```
...
        //setupAction.SerializerSettings.Converters.Add(new DateJsonConverter());
...
```

Add attribute JsonConverter in Order.cs:

```
using Microsoft.VisualBasic;
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace WebApp.Models
{
    public class Order
    {
        public long OrderId { get; set; }
        public virtual ICollection<Product> Products { get; set; }
        public long OrderNo { get; set; }
    }
}
```

```

[Newtonsoft.Json.JsonConverter(typeof(DateJsonConverter))]
public DateTime OrderDate { get; set; }
[NotMapped]
public double Total {
    get {
        double t = 0;
        if(Products != null)
        {
            foreach (var p in Products) t += p.Price;
        }
        return t;
    }
}
}
}

```

### 3 Response formatting

ASP.NET Core MVC has support for formatting response data. Response data can be formatted using specific formats or in response to client requested format.

#### 3.1 Format-specific Action Results

Some action result types are specific to a particular format, such as [JsonResult](#) and [ContentResult](#). Actions can return results that are formatted in a particular format, regardless of client preferences. For example, returning `JsonResult` returns JSON-formatted data. Returning `ContentResult` or a string returns plain-text-formatted string data.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that are not [IActionResult](#) types are serialized using the appropriate [IOutputFormatter](#) implementation. For more information, see [Controller action return types in ASP.NET Core web API](#).

The built-in helper method [Ok](#) returns JSON-formatted data:

```

// GET: api/authors
[HttpGet]
public ActionResult Get()
{
    return Ok(_authors.List());
}

```

The sample download returns the list of authors. Using the F12 browser developer tools or [Postman](#) with the previous code:

- The response header containing **content-type:** application/json; charset=utf-8 is displayed.

- The request headers are displayed. For example, the Accept header. The Accept header is ignored by the preceding code.

To return plain text formatted data, use [ContentResult](#) and the [Content](#) helper:

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

In the preceding code, the Content-Type returned is text/plain. Returning a string delivers Content-Type of text/plain:

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

For actions with multiple return types, return IActionResult. For example, returning different HTTP status codes based on the result of operations performed.

## 3.2 Content negotiation

Content negotiation occurs when the client specifies an [Accept header](#). The default format used by ASP.NET Core is [JSON](#). Content negotiation is:

- Implemented by [ObjectResult](#).
- Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on ObjectResult.

When a model type is returned, the return type is ObjectResult.

The following action method uses the ok and NotFound helper methods:

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authors.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

By default, ASP.NET Core supports `application/json`, `text/json`, and `text/plain` media types. Tools such as [Fiddler](#) or [Postman](#) can set the `Accept` request header to specify the return format. When the `Accept` header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an `ObjectResult` that wraps the object. The client gets the formatted serialized object. If the object being returned is `null`, a `204 No Content` response is returned.

Returning an object type:

```
// GET api/authors/RickAndMSFT
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authors.GetByAlias(alias);
}
```

In the preceding code, a request for a valid author alias returns a `200 OK` response with the author's data. A request for an invalid alias returns a `204 No Content` response.

### 3.2.1 The `Accept` header

Content *negotiation* takes place when an `Accept` header appears in the request. When a request contains an `accept` header, ASP.NET Core:

- Enumerates the media types in the `accept` header in preference order.
- Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

- Returns `406 Not Acceptable` if [MvcOptions](#) has been set, or -
- Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no `Accept` header appears in the request:

- The first formatter that can handle the object is used to serialize the response.
- There isn't any negotiation taking place. The server is determining what format to return.

If the `Accept` header contains `*/*`, the Header is ignored unless `RespectBrowserAcceptHeader` is set to `true` on [MvcOptions](#).



### 3.2.2 Browsers and content negotiation

Unlike typical API clients, web browsers supply Accept headers. Web browser specify many formats, including wildcards. By default, when the framework detects that the request is coming from a browser:

- The Accept header is ignored.
- The content is returned in JSON, unless otherwise configured.

This provides a more consistent experience across browsers when consuming APIs.

To configure an app to honor browser accept headers, set [RespectBrowserAcceptHeader](#) to true:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        options.RespectBrowserAcceptHeader = true; // false by default
    });
}
```

### 3.2.3 Configure formatters

Apps that need to support additional formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by [Model Binding](#). Output formatters are used to format responses. For information on creating a custom formatter, see [Custom Formatters](#).

#### 3.2.3.1 Add XML format support

XML formatters implemented using [XmlSerializer](#) are configured by calling [AddXmlSerializerFormatters](#):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers()
        .AddXmlSerializerFormatters();
}
```

The preceding code serializes results using XmlSerializer.

When using the preceding code, controller methods return the appropriate format based on the request's Accept header.

#### 3.2.3.2 Configure System.Text.Json-based formatters

Features for the System.Text.Json-based formatters can be configured using Microsoft.AspNetCore.Mvc.JsonOptions.SerializerOptions.

```
services.AddControllers().AddJsonOptions(options =>
```

```

{
    // Use the default property (Pascal) casing.
    options.JsonSerializerOptions.PropertyNamingPolicy = null;

    // Configure a custom converter.
    options.JsonSerializerOptions.Converters.Add(new MyCustomJsonConverter());
});

```

Output serialization options, on a per-action basis, can be configured using JsonResult. For example:

```

public IActionResult Get()
{
    return Json(model, new JsonSerializerOptions
    {
        WriteIndented = true,
    });
}

```

### 3.2.3.3 Add Newtonsoft.Json-based JSON format support

Prior to ASP.NET Core 3.0, the default used JSON formatters implemented using the Newtonsoft.Json package. In ASP.NET Core 3.0 or later, the default JSON formatters are based on System.Text.Json. Support for Newtonsoft.Json based formatters and features is available by installing the [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#) NuGet package and configuring it in Startup.ConfigureServices.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers()
        .AddNewtonsoftJson();
}

```

In the preceding code, the call to AddNewtonsoftJson configures the following Web API, MVC, and Razor Pages features to use Newtonsoft.Json:

- Input and output formatters that read and write JSON
- [JsonResult](#)
- [JSON Patch](#)
- [IJsonHelper](#)
- [TempData](#)

Some features may not work well with System.Text.Json-based formatters and require a reference to the Newtonsoft.Json-based formatters. Continue using the Newtonsoft.Json-based formatters if the app:

- Uses Newtonsoft.Json attributes. For example, [JsonProperty] or [JsonIgnore].

- Customizes the serialization settings.
- Relies on features that Newtonsoft.Json provides.
- Configures Microsoft.AspNetCore.Mvc.JsonResult.SerializerSettings. Prior to ASP.NET Core 3.0, JsonResult.SerializerSettings accepts an instance of JsonSerializerSettings that is specific to Newtonsoft.Json.
- Generates [OpenAPI](#) documentation.

Features for the Newtonsoft.Json-based formatters can be configured using Microsoft.AspNetCore.Mvc.MvcNewtonsoftJsonOptions.SerializerSettings:

```
services.AddControllers().AddNewtonsoftJson(options =>
{
    // Use the default property (Pascal) casing
    options.SerializerSettings.ContractResolver = new DefaultContractResolver();

    // Configure a custom converter
    options.SerializerSettings.Converters.Add(new MyCustomJsonConverter());
});
```

Output serialization options, on a per-action basis, can be configured using JsonResult. For example:

```
public IActionResult Get()
{
    return Json(model, new JsonSerializerSettings
    {
        Formatting = Formatting.Indented,
    });
}
```

### 3.2.3.4 Specify a format

To restrict the response formats, apply the [\[Produces\]](#) filter. Like most [Filters](#), [Produces] can be applied at the action, controller, or global scope:

```
[ApiController]
[Route("[controller]")]
[Produces("application/json")]
public class WeatherForecastController : ControllerBase
{
```

The preceding [\[Produces\]](#) filter:

- Forces all actions within the controller to return JSON-formatted responses.
- If other formatters are configured and the client specifies a different format, JSON is returned.

### 3.2.3.5 Special case formatters

Some special cases are implemented using built-in formatters. By default, string return types are formatted as text/plain (text/html if requested via the Accept header). This behavior can be deleted by removing the `StringOutputFormatter`. Formatters are removed in the `ConfigureServices` method. Actions that have a model object return type return 204 No Content when returning null. This behavior can be deleted by removing the `HttpNoContentOutputFormatter`. The following code removes the `StringOutputFormatter` and `HttpNoContentOutputFormatter`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        // requires using Microsoft.AspNetCore.Mvc.Formatters;
        options.OutputFormatters.RemoveType<StringOutputFormatter>();
        options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
    });
}
```

Without the `StringOutputFormatter`, the built-in JSON formatter formats string return types. If the built-in JSON formatter is removed and an XML formatter is available, the XML formatter formats string return types. Otherwise, string return types return 406 Not Acceptable.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example:

- The JSON formatter returns a response with a body of null.
- The XML formatter returns an empty XML element with the attribute `xsi:nil="true"` set.

### 3.2.4 Response format URL mappings

Clients can request a particular format as part of the URL, for example:

- In the query string or part of the path.
- By using a format-specific file extension such as `.xml` or `.json`.

The mapping from request path should be specified in the route the API is using. For example:

```
[Route("api/[controller]")]
[ApiController]
[FormatFilter]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}.{format?}")]
    public Product Get(int id){
```

The preceding route allows the requested format to be specified as an optional file extension. The [FormatFilter] attribute checks for the existence of the format value in the RouteData and maps the response format to the appropriate formatter when the response is created.

Route	Formatter
<b>/api/products/5</b>	The default output formatter
<b>/api/products/5.json</b>	The JSON formatter (if configured)
<b>/api/products/5.xml</b>	The XML formatter (if configured)

For api, we will discuss in lesson 07.

## 4 References

1. <https://www.aspsnippets.com/Articles/ASPNet-Core-jQuery-AJAX-and-JSON-Example-in-ASPNet-Core-MVC.aspx>
2. <http://aspsolution.net/Code/5/5087/How-to-Ajax-call-in-ASPNET-Core-MVC/>
3. <https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-3.1>