

Format response data in ASP.NET Core Web API

01/28/2021 • 8 minutes to read •  +7

In this article

[Format-specific Action Results](#)

[Content negotiation](#)

[Response format URL mappings](#)

By [Rick Anderson](#) and [Steve Smith](#)

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.

[View or download sample code](#) ([how to download](#))

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 [ActionResult](#) types are serialized using the appropriate [OutputFormatter](#) 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:

C#


 Copy

```
// 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:

C#	 Copy
<pre>// GET api/authors/about [HttpGet("About")] public ContentResult About() { return Content("An API listing authors of docs.asp.net."); }</pre>	

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

C#	 Copy
<pre>// GET api/authors/version [HttpGet("version")] public string Version() { return "Version 1.0.0"; }</pre>	

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

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:

C#	 Copy
<pre>// 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); }</pre>	

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:

C#	 Copy
<pre>// GET api/authors/RickAndMSFT [HttpGet("{alias}")] public Author Get(string alias) { return _authors.GetByAlias(alias); }</pre>	

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.

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.ReturnHttpNotAcceptable` is set to `true`, 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`.

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`:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{
```

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

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](#).

Add XML format support

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

C#

 Copy

```
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.

Configure System.Text.Json based formatters

Features for the `System.Text.Json` based formatters can be configured using [Microsoft.AspNetCore.Mvc.JsonOptions.JsonSerializerOptions](#). The default formatting is `camelCase`. The following highlighted code sets `PascalCase` formatting:

C#

 Copy

```
public void ConfigureServices(IServiceCollection services)
{

```

```
services.AddControllers()
    .AddJsonOptions(options =>
        options.JsonSerializerOptions.PropertyNamingPolicy = null);
}
```

The following action method calls [ControllerBase.Problem](#) to create a [ProblemDetails](#) response:

C#

 Copy

```
[HttpGet("error")]
public IActionResult GetError()
{
    return Problem("Something went wrong!");
}
```

With the preceding code:

- <https://localhost:5001/WeatherForecast/temperature> returns PascalCase.
- <https://localhost:5001/WeatherForecast/error> returns camelCase. The error response is always camelCase, even when the app sets the format to PascalCase. [ProblemDetails](#) follows [RFC 7807](#) , which specifies lower case

The following code sets PascalCase and adds a custom converter:

C#

 Copy

```
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:

C#

 Copy

```
public IActionResult Get()
{
    return Json(model, new JsonSerializerOptions
    {
```

```
        WriteIndented = true,  
    });  
}
```

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`.

C#

 Copy

```
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`:

C#

 Copy

```
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:

C#

 Copy

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

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:

C#

 Copy

```
[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.

For more information, see [Filters](#).

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`.

C#

 Copy

```
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.

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:

C# 

```
[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
/api/products/5	The default output formatter
/api/products/5.json	The JSON formatter (if configured)
/api/products/5.xml	The XML formatter (if configured)

Is this page helpful?

 Yes  No