



```

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+F) P RoutingExample
Program.cs | MonthCustomConstraint.cs
RoutingExample
24     {fileName} - {extension}");;
25 };
26 //Eg: employee/profile/john
27 //This is for application development.
28 endpoints.Map("employee/profile/
{EmployeeName:length(4,7):alpha=harsha}", async context =>
29 {
30     string? employeeName = Convert.ToString
31         (context.Request.RouteValues["employeename"]);
32     await context.Response.WriteAsync($"In
33         Employee profile - {employeeName}");
34 });
35 //Eg: products/details/

```



In the existing application, we created one endpoint for each URL and wrote the corresponding code in the same middleware.

But think about real-world projects—can you follow this approach in large-scale applications?

In real-world scenarios, there will be thousands of URLs with a significant amount of code. Writing all the endpoints in the same file is not good programming practice.

If the code grows in length and everything is in the same file without logical grouping, it becomes difficult to identify issues, debug errors, and maintain the codebase.

So, what's the solution? The solution is to group middlewares based on their logical purpose, and this is exactly what controllers provide.

Asp.Net Core | Harsha

Introduction to Controllers

Controller is a class that is used to group-up a set of actions (*or action methods*).

The diagram illustrates the flow of requests through the routing and controller layers. On the left, two arrows point to a blue box labeled 'Routing'. The top arrow is labeled 'Request to /url1' and the bottom arrow is labeled 'Request to /url2'. From the 'Routing' box, two arrows point to a white box labeled 'Controller'. The top arrow is labeled 'Request to /url1' and the bottom arrow is labeled 'Request to /url2'. Inside the 'Controller' box, there are two grey boxes: 'Action 1 [endpoint]' and 'Action 2 [endpoint]'. A small watermark for 'Harsha Web University' is visible in the bottom right corner of the slide.

A controller is a class that contains a set of action methods. Each action method acts as an endpoint that can be requested via a specific URL.

For example, if you send a request to URL1, Action 1 will execute, and similarly, URL2 will trigger Action 2. These actions are grouped into a controller class, meaning there is a logical connection between them.

For instance, Action 1 may represent the user registration process, and Action 2 may represent the sign-in process. Both are related to user account maintenance, which forms the logical connection.

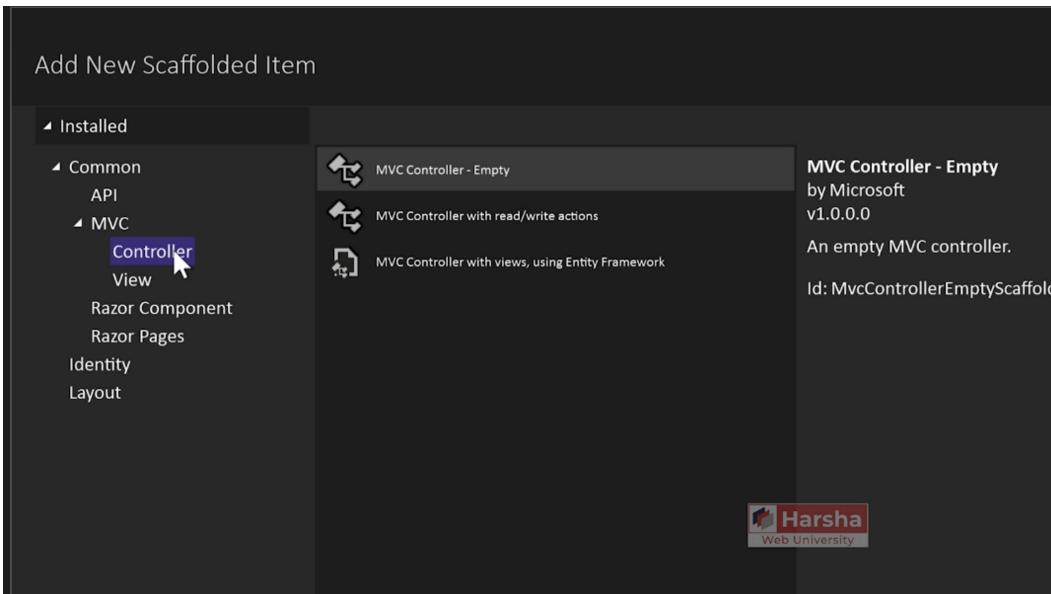
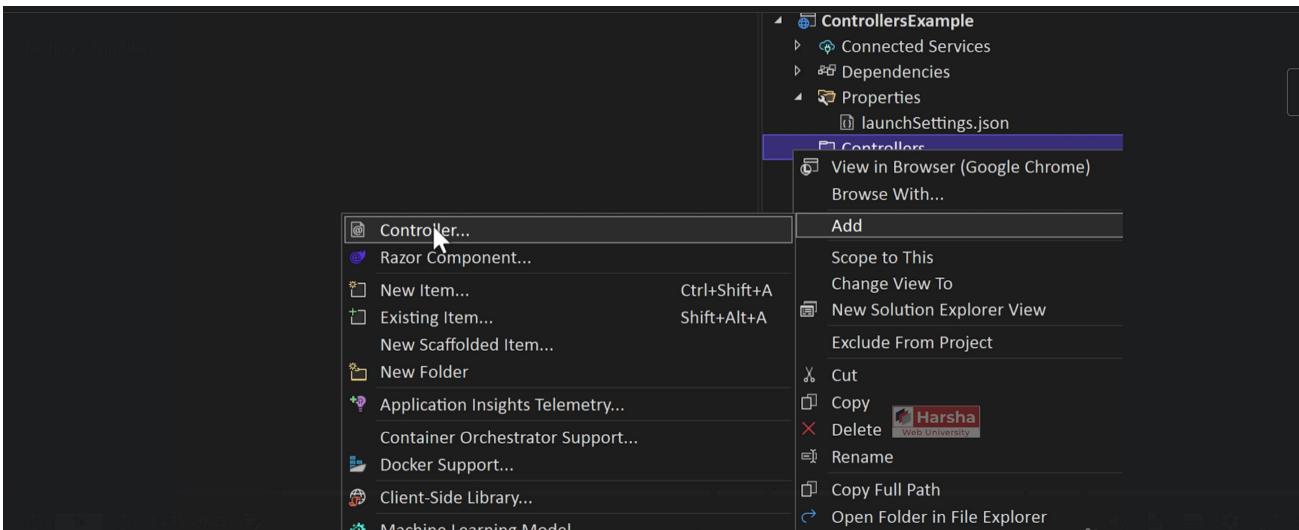
By definition, a controller is a collection of action methods (or actions for short), where each action performs a specific operation based on the inputs supplied.

For example:

- Supplying details like username, password, confirm password, and date of birth to Action 1 initiates the registration process.
- Supplying email and password to Action 2 handles the login process.

The primary reason to create controllers is to group actions based on their purpose. Now, how can you create and enable controllers? Let's try it practically!

create a folder called 'Controllers' (not mandatory) in visual studio.



just a normal class, not a controller class

A screenshot of the Visual Studio code editor and Solution Explorer. The code editor shows the 'Home.cs' file with the following content:

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace ControllersExample.Controllers
4  {
5      class Home
6      {
7          }
8      }
9  }
10
```

The Solution Explorer on the right shows the project structure: 'ControllersExample' (1 of 1 project) containing 'Connected Services', 'Dependencies', 'Properties', 'launchSettings.json', 'Controllers' (selected), 'HomeController.cs', 'appsettings.json', and 'Program.cs'.

In order to make that class as a Controller class, you have to suffix the same with 'Controller'. It's the fixed pre-defined name.



The screenshot shows the Visual Studio IDE. On the left is the code editor with the following C# code:

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace ControllersExample.Controllers
4  {
5      class HomeController
6      {
7      }
8  }
```

The 'HomeController' class is highlighted with a green background. On the right is the Solution Explorer window, which displays the project structure:

- Solution 'ControllersExample'
 - ControllersExample
 - Dependencies
 - Properties
 - launchSettings.json
 - Controllers
 - HomeController.cs
 - appsettings.json
 - Program.cs

Asp.Net Core

Multiple Action Methods

Asp .Net Core

Takeouts about Controllers

Asp .Net Core

Harsha

Creating Controllers

Should be either or both:

- The class name should be suffixed with "Controller". Eg: HomeController
- The [Controller] attribute is applied to the same class or to its base class.

Controller

```
[Controller]  
class ClassNameController  
{  
    //action methods here  
}
```

Optional:

- Is a public class.
- Inherited from Microsoft.AspNetCore.Mvc.Controller.

Asp .Net Core

Harsha

Enable 'routing' in controllers

AddControllers()

```
builder.Services.AddControllers();
```

Adds all controllers as services in the `IServiceCollection`.

So that, they can be accessed when a specific endpoint needs it.

MapControllers()

```
app.MapControllers();
```

Adds all action methods as endpoints.

So that, no need of using `UseEndpoints()` method for adding action methods as end points.

Responsibilities of Controllers

Reading requests

Extracting data values from request such as query string parameters, request body, request cookies, request headers etc.

Invoking models

Validation

Validate incoming request details (query string parameters, request body, request cookies, request headers etc.)

Preparing Response



either use Suffix 'Controller' . ex: HomeController

or

```

1  using Microsoft.AspNetCore.Mvc;
2
3  namespace ControllersExample.Controllers
4  {
5      [Controller]
6      public class Home
7      {
8          [Route("home")]
9          [Route("/")]
10         public string Index()
11         {
12             return "Hello from Index";
13         }
14     }
15 }
```



Asp.Net Core

ContentResult

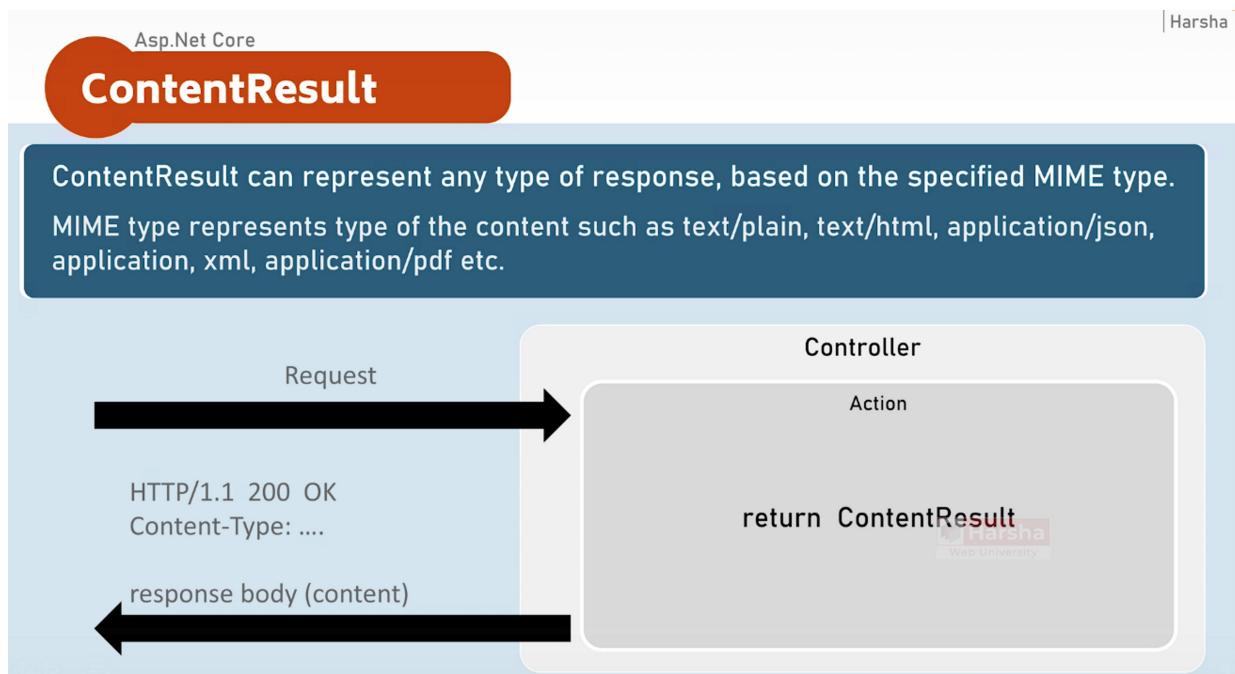
Asp.Net Core

| Harsha

ContentResult

ContentResult can represent any type of response, based on the specified MIME type.

MIME type represents type of the content such as text/plain, text/html, application/json, application/xml, application/pdf etc.



The ContentResult is a type of action result that can be returned from an action method. It can represent almost any type of content.

For example, you can return plain text, HTML, XML, JSON, PDF files, or nearly any type of content using ContentResult.

When creating an object for the ContentResult class, you need to specify two pieces of information:

Content: The actual response body you want to return, such as a message.

ContentType: The MIME type of the content, which is added to the response headers.

For example:

- If the response body is plain text, the ContentType would be text/plain.

We have already seen an overview of the ContentType while learning about request and response

objects earlier.

Now, let's see how to use it in this existing application.

Asp.Net Core JsonResult



JsonResult

```
return new JsonResult(your_object);  
[or]  
return Json(your_object);
```

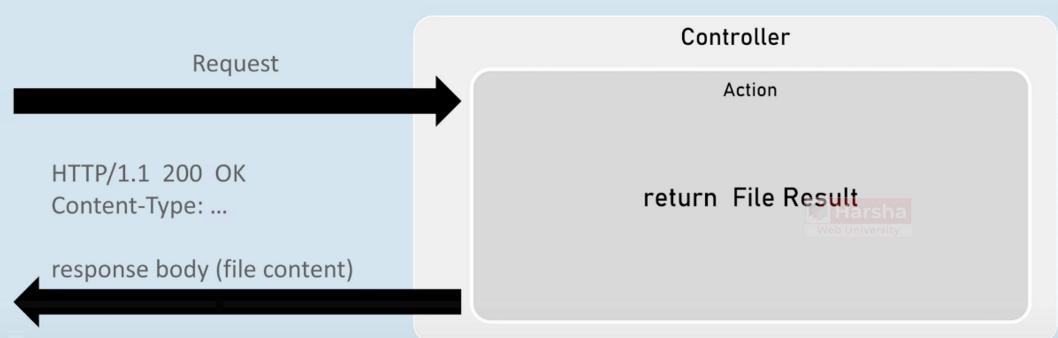
Asp.Net Core

File Results

File Results

File result sends the content of a file as response.

Eg: pdf file, txt file, exe file, zip file etc.



Did you notice sometimes when you open some webpages It gives you a file as download.

Asp.Net Core

Harsha

File Results

VirtualFileResult

```
return new VirtualFileResult("file  
relative path", "content type");
```

PhysicalFileResult

```
return new PhysicalFileResult("file  
absolute path", "content type");
```

FileContentResult

```
return new FileContentResult(byte_array, "content type");
```



Asp.Net Core

Harsha

File Results

VirtualFileResult

- Represents a file within the WebRoot ('wwwroot' by default) folder.
- Used when the file is present in the WebRoot folder.

```
return new VirtualFileResult("file  
relative path", "content type");  
//or  
return File("file relative path",  
"content type");
```

PhysicalFileResult

- Represents a file that is not necessarily part of the project folder.
- Used when the file is present outside the WebRoot folder.

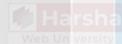
```
return new PhysicalFileResult("file  
absolute path", "content type");  
//or  
return PhysicalFile("file absolute  
path", "content type");
```

File Results

FileContentResult

- Represents a file from the byte[].
- Used when a part of the file or byte[] from other data source has to be sent as response.

```
return new FileContentResult(byte_array, "content type");  
//or  
return File(byte_array, "content type");
```



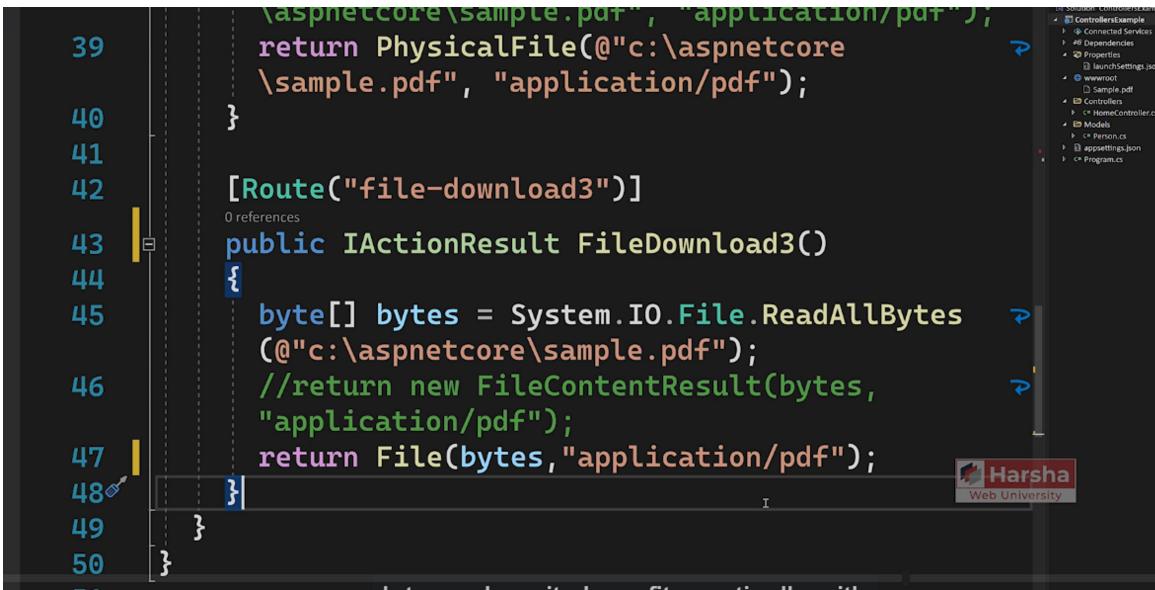
It is useful when you are reading images form databases

Asp.Net Core IActionResult

In ASP.NET Core, it is required to mention the return type of all the action methods as **IActionResult**, like this:

In this case, you can return any type of action result such as ContentResult, JsonResult, FileResult, etc.

But here, my question is: how is that possible?



```
39     \aspnetcore\sample.pdf", "application/pdf");
40     return PhysicalFile(@"c:\aspnetcore
41     \sample.pdf", "application/pdf");
42 }
43 [Route("file-download3")]
44 public IActionResult FileDownload3()
45 {
46     byte[] bytes = System.IO.File.ReadAllBytes
47     (@@"c:\aspnetcore\sample.pdf");
48     //return new FileContentResult(bytes,
49     //"application/pdf");
50     return File(bytes,"application/pdf");
51 }
```

Asp.Net Core

Harsha

IActionResult

It is the parent interface for all action result classes such as ContentResult, JsonResult, RedirectResult, StatusCodeResult, ViewResult etc.

By mentioning the return type as IActionResult, you can return either of the subtypes of IActionResult.

But why is the IActionResult even necessary?

suppose my requirement is to pass 'isLoggedIn' and 'bookid' value.

010/book?isloggedin=true&bookid=1

Body Pre-request Script Tests Settings

VALUE

true

1



010/book?isloggedin=true

Body Pre-request Script Tests Settings

VALUE

Asp.Net Core

Harsha

IActionResult

interface IActionResult

class ActionResult

class ContentResult

class JsonResult

class ViewResult

class EmptyResult

class StatusCodeResult

class FileResult

class ViewComponentResult

class RedirectResult

class LocalRedirectResult

class PartialViewResult

class ObjectResult

class RedirectToRouteResult

class RedirectToActionResult

Asp.Net Core

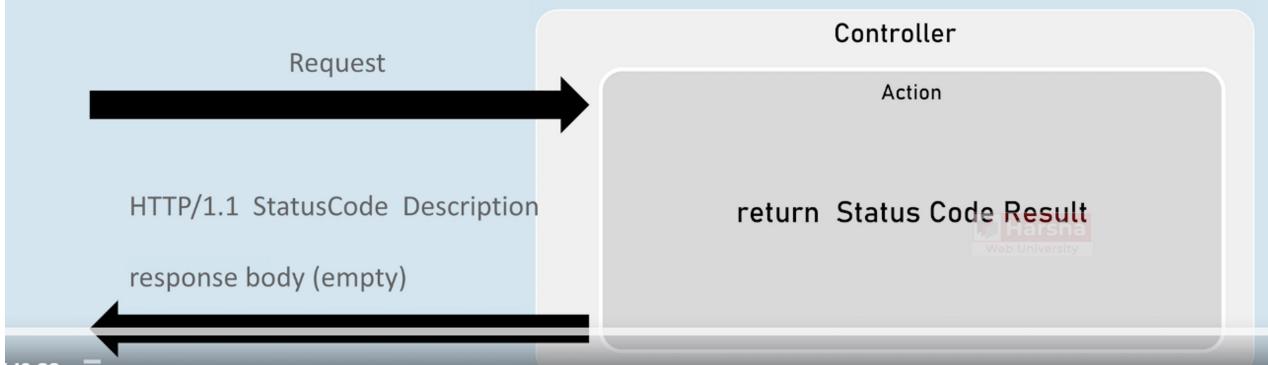
Status Code Results

Asp.Net Core

Harsha

Status Code Results

Status code result sends an empty response with specified status code.



Status Code Results

StatusCodeResult

```
return new StatusCodeResult(status_code);
```

UnauthorizedResult

```
return new UnauthorizedResult();
```

BadRequestResult

```
return new BadRequestResult();
```

NotFoundResult

```
return new NotFoundResult();
```

Status Code Results

StatusCodeResult

- Represents response with the specified status code.
- Used when you would like to send a specific HTTP status code as response.

```
return new  
StatusCodeResult(status_code);  
//or  
return StatusCode(status_code);
```

UnauthorizedResult

- Represents response with HTTP status code '401 Unauthorized'.
- Used when the user is unauthorized (not signed in).

```
return new UnauthorizedResult();  
//or  
return Unauthorized();
```

Status Code Results

BadRequestResult

- Represents response with HTTP status code '400 Bad Request'.
- Used when the request values are invalid (validation error).

```
return new BadRequestResult();
//or
return BadRequest();
```

NotFoundResult

- Represents response with HTTP status code '404 Not Found'.
- Used when the requested information is not available at server.

```
return new NotFoundResult();
//or
return NotFound();
```

Asp.Net Core Redirect Results Part 1

Redirect:
/bookstore
to
/store/books

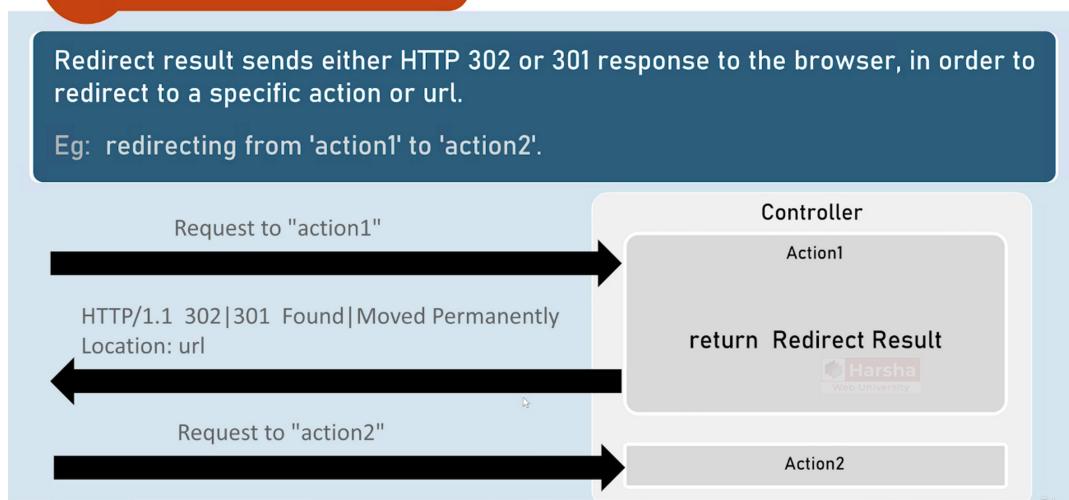
Asp.Net Core

Harsha

Redirect Results

Redirect result sends either HTTP 302 or 301 response to the browser, in order to redirect to a specific action or url.

Eg: redirecting from 'action1' to 'action2'.



Asp.Net Core
Redirect Results
Part 2

Redirect Results

RedirectToActionResult

```
return new RedirectToActionResult("action", "controller", new { route_values }, permanent);
```

LocalRedirectResult

```
return new LocalRedirectResult("local_url", permanent);
```

RedirectResult

```
return new RedirectResult("url", permanent);
```



RedirectToActionResult

- Represents response for redirecting from the current action method to another action method, based on action name and controller name.

302 - Found

```
return new RedirectToActionResult("action", "controller", new { route_values });
//or
return RedirectToAction("action", "controller", new { route_values });
```

301 - Moved Permanently

```
return new RedirectToActionResult("action", "controller", new { route_values }, true);
//or
return RedirectToActionPermanent("action", "controller", new { route_values });
```