**IgnoreRoute**

We have two controllers in our project, Student and Employee, and we have some issue with the Employee controller. Until the issue is solved, we need to stop the user accessing the Employee controller using the RegisterRoutes method in Routeconfg.cs,

routes.IgnoreRoute("Employee/");

*Routing* is how ASP.NET MVC matches a URI to an action. MVC 5 supports a new type of routing, called *attribute routing*. As the name implies, attribute routing uses attributes to define routes. Attribute routing gives you more control over the URIs in your web application.

The earlier style of routing, called convention-based routing, is still fully supported. In fact, you can combine both techniques in the same project.

This post will cover the basic features and options of Attribute Routing, in ASP.NET MVC 5.

* [Why Attribute Routing?](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#why-attribute-routing)
* [Enabling Attribute Routing](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#enabling-attribute-routing)
* [Optional URI Parameters and Default Values](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#optionals-and-defaults)
* [Route Prefixes](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#route-prefixes)
* [Default Route](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#default-route)
* [Route Constraints](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#route-constraints)
  + [Custom Route Constraints](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#custom-route-constraints)
* [Route Names](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#route-names)
* [Areas](https://devblogs.microsoft.com/aspnet/attribute-routing-in-asp-net-mvc-5/#route-areas)

**Why Attribute Routing?**

For example, a socially enhanced e-commerce website could have the following routes:

* *{productId:int}/{productTitle}*  
  Mapped to ProductsController.Show(int id)
* *{username}*  
  Mapped to ProfilesController.Show(string username)
* *{username}/catalogs/{catalogId:int}/{catalogTitle}*  
  Mapped to CatalogsController.Show(string username, int catalogId)

(Don’t mind the specific syntax right now, we will touch on this later.)

In previous version of ASP.NET MVC, the rules would be set in the RouteConfig.cs file, and point to the actual controller actions, as such:

routes.MapRoute(

    name: “ProductPage”,

    url: “{productId}/{productTitle}”,

    defaults: new { controller = “Products”, action = “Show” },

    constraints: new { productId = “\\d+” }

);

When the route definitions are co-located with the actions, within the same source file rather than being declared on an external configuration class, it can make it easier to reason about the mapping between URIs and actions. The previous route definition would be set using the following, simple attribute:

* [Route(“{productId:int}/{productTitle}”)]
* public ActionResult Show(int productId) { … }

**Enabling Attribute Routing**

To enable attribute routing, call MapMvcAttributeRoutes during configuration.

* public class RouteConfig
* {
* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* routes.MapMvcAttributeRoutes();
* }
* }

You can also combine attribute routing with convention-based routing.

* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* routes.MapMvcAttributeRoutes();
* routes.MapRoute(
* name: “Default”,
* url: “{controller}/{action}/{id}”,
* defaults: new { controller = “Home”, action = “Index”, id = UrlParameter.Optional }
* );
* }

**Optional URI Parameters and Default Values**

You can make a URI parameter optional by adding a question mark to the route parameter. You can also specify a default value by using the form parameter=value.

* public class BooksController : Controller
* {
* // eg: /books
* // eg: /books/1430210079
* [Route(“books/{isbn?}”)]
* public ActionResult View(string isbn)
* {
* if (!String.IsNullOrEmpty(isbn))
* {
* return View(“OneBook”, GetBook(isbn));
* }
* return View(“AllBooks”, GetBooks());
* }
* // eg: /books/lang
* // eg: /books/lang/en
* // eg: /books/lang/he
* [Route(“books/lang/{lang=en}”)]
* public ActionResult ViewByLanguage(string lang)
* {
* return View(“OneBook”, GetBooksByLanguage(lang));
* }
* }

In this example, both /books and /books/1430210079 will route to the “View” action, the former will result with listing all books, and the latter will list the specific book. Both /books/lang and /books/lang/en will be treated the same.

**Route Prefixes**

Often, the routes in a controller all start with the same prefix. For example:

* public class ReviewsController : Controller
* {
* // eg: /reviews
* [Route(“reviews”)]
* public ActionResult Index() { … }
* // eg: /reviews/5
* [Route(“reviews/{reviewId}”)]
* public ActionResult Show(int reviewId) { … }
* // eg: /reviews/5/edit
* [Route(“reviews/{reviewId}/edit”)]
* public ActionResult Edit(int reviewId) { … }
* }

You can set a common prefix for an entire controller by using the [RoutePrefix] attribute:

* [RoutePrefix(“reviews”)]
* public class ReviewsController : Controller
* {
* // eg.: /reviews
* [Route]
* public ActionResult Index() { … }
* // eg.: /reviews/5
* [Route(“{reviewId}”)]
* public ActionResult Show(int reviewId) { … }
* // eg.: /reviews/5/edit
* [Route(“{reviewId}/edit”)]
* public ActionResult Edit(int reviewId) { … }
* }

Use a tilde (~) on the method attribute to override the route prefix if needed:

* [RoutePrefix(“reviews”)]
* public class ReviewsController : Controller
* {
* // eg.: /spotlight-review
* [Route(“~/spotlight-review”)]
* public ActionResult ShowSpotlight() { … }
* …
* }

**Default Route**

You can also apply the [Route] attribute on the controller level, capturing the action as a parameter. That route would then be applied on all actions in the controller, unless a specific [Route] has been defined on a specific action, overriding the default set on the controller.

* [RoutePrefix(“promotions”)]
* [Route(“{action=index}”)]
* public class ReviewsController : Controller
* {
* // eg.: /promotions
* public ActionResult Index() { … }
* // eg.: /promotions/archive
* public ActionResult Archive() { … }
* // eg.: /promotions/new
* public ActionResult New() { … }
* // eg.: /promotions/edit/5
* [Route(“edit/{promoId:int}”)]
* public ActionResult Edit(int promoId) { … }
* }

**Route Constraints**

Route constraints let you restrict how the parameters in the route template are matched. The general syntax is {parameter:constraint}. For example:

* // eg: /users/5
* [Route(“users/{id:int}”]
* public ActionResult GetUserById(int id) { … }
* // eg: users/ken
* [Route(“users/{name}”]
* public ActionResult GetUserByName(string name) { … }

Here, the first route will only be selected if the “id” segment of the URI is an integer. Otherwise, the second route will be chosen.

The following table lists the constraints that are supported.

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| alpha | Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z) | {x:alpha} |
| bool | Matches a Boolean value. | {x:bool} |
| datetime | Matches a **DateTime** value. | {x:datetime} |
| decimal | Matches a decimal value. | {x:decimal} |
| double | Matches a 64-bit floating-point value. | {x:double} |
| float | Matches a 32-bit floating-point value. | {x:float} |
| guid | Matches a GUID value. | {x:guid} |
| int | Matches a 32-bit integer value. | {x:int} |
| length | Matches a string with the specified length or within a specified range of lengths. | {x:length(6)} {x:length(1,20)} |
| long | Matches a 64-bit integer value. | {x:long} |
| max | Matches an integer with a maximum value. | {x:max(10)} |
| maxlength | Matches a string with a maximum length. | {x:maxlength(10)} |
| min | Matches an integer with a minimum value. | {x:min(10)} |
| minlength | Matches a string with a minimum length. | {x:minlength(10)} |
| range | Matches an integer within a range of values. | {x:range(10,50)} |
| regex | Matches a regular expression. | {x:regex(^\d{3}-\d{3}-\d{4}$)} |

Notice that some of the constraints, such as “min”, take arguments in parentheses.

You can apply multiple constraints to a parameter, separated by a colon, for example:

* // eg: /users/5  
  // but not /users/10000000000 because it is larger than int.MaxValue,  
  // and not /users/0 because of the min(1) constraint.
* [Route(“users/{id:int:min(1)}”)]
* public ActionResult GetUserById(int id) { … }

Specifying that a parameter is Optional (via the ‘?’ modifier) should be done after inline constraints:

* // eg: /greetings/bye  
  // and /greetings because of the Optional modifier,  
  // but not /greetings/see-you-tomorrow because of the maxlength(3) constraint.  
  [Route(“greetings/{message:maxlength(3)?}”)]
* public ActionResult Greet(string message) { … }

**Custom Route Constraints**

You can create custom route constraints by implementing the IRouteConstraint interface. For example, the following constraint restricts a parameter to set of valid values:

* public class ValuesConstraint : IRouteConstraint
* {
* private readonly string[] validOptions;
* public ValuesConstraint(string options)
* {
* validOptions = options.Split(‘|’);
* }
* public bool Match(HttpContextBase httpContext, Route route, string parameterName, RouteValueDictionary values, RouteDirection routeDirection)
* {
* object value;
* if (values.TryGetValue(parameterName, out value) && value != null)
* {
* return validOptions.Contains(value.ToString(), StringComparer.OrdinalIgnoreCase);
* }
* return false;
* }
* }

The following code shows how to register the constraint:

* public class RouteConfig
* {
* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* var constraintsResolver = new DefaultInlineConstraintResolver();
* constraintsResolver.ConstraintMap.Add(“values”, typeof(ValuesConstraint));
* routes.MapMvcAttributeRoutes(constraintsResolver);
* }
* }

Now you can apply the constraint in your routes:

* public class TemperatureController : Controller
* {
* // eg: temp/celsius and /temp/fahrenheit but not /temp/kelvin
* [Route(“temp/{scale:values(celsius|fahrenheit)}”)]
* public ActionResult Show(string scale)
* {
* return Content(“scale is “ + scale);
* }
* }

**Route Names**

You can specify a name for a route, in order to easily allow URI generation for it. For example, for the following route:

* [Route(“menu”, Name = “mainmenu”)]
* public ActionResult MainMenu() { … }

you could generate a link using Url.RouteUrl:

* <a href=”@Url.RouteUrl(“mainmenu”)“>Main menu</a>

**Areas**

You can define that a controller belongs to an area by using the [RouteArea] attribute. When doing so, you can safely remove the AreaRegistration class for that area.

* [RouteArea(“Admin”)]
* [RoutePrefix(“menu”)]
* [Route(“{action}”)]
* public class MenuController : Controller
* {
* // eg: /admin/menu/login
* public ActionResult Login() { … }
* // eg: /admin/menu/show-options
* [Route(“show-options”)]
* public ActionResult Options() { … }
* // eg: /stats
* [Route(“~/stats”)]
* public ActionResult Stats() { … }
* }

With this controller, the following link generation call will result with the string “*/Admin/menu/show-options*“

* Url.Action(“Options”, “Menu”, new { Area = “Admin” })

You can set up a custom prefix for the area that defer from the area name, by using the AreaPrefix named parameter, for example:

* [RouteArea(“BackOffice”, AreaPrefix = “back-office”)]

If you are using both Areas with route attributes, and areas with convention based routes (set by an AreaRegistration class), then you need to make sure that area registration happen after MVC attribute routes are configured, however before the default convention-based route is set. The reason is that route registration should be ordered from the most specific (attributes) through more general (area registration) to the mist generic (the default route) to avoid generic routes from “hiding” more specific routes by matching incoming requests too early in the pipeline.

Example:

* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* routes.MapMvcAttributeRoutes();
* AreaRegistration.RegisterAllAreas();
* routes.MapRoute(
* name: “Default”,
* url: “{controller}/{action}/{id}”,
* defaults: new { controller = “Home”, action = “Index”, id = UrlParameter.Optional }
* );
* }