**Routing**public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

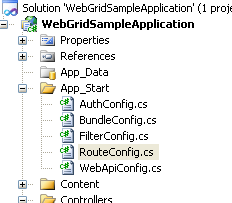
defaults: new { controller = "Inventory", action = "WebgridSample", id = UrlParameter.Optional }

);

}

}

1. **protected** **void** Application\_Start()
2. {
3. …
4. RouteConfig.RegisterRoutes(RouteTable.Routes);
5. }



**Global action filters** are mostly useful for these functionalities:

* Error handling
* Logging
* Debugging

**http://www.c-sharpcorner.com/UploadFile/56fb14/custom-authorization-in-mvc/**

**Dependency resolution**

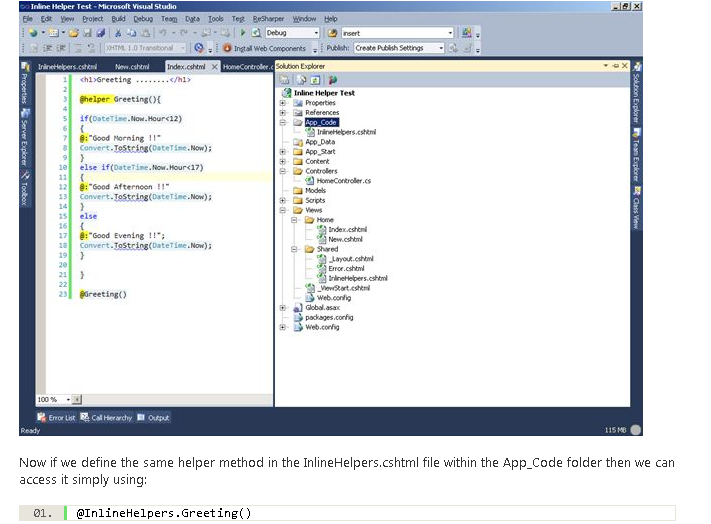
MVC 3 included a new concept called a dependency resolution that greatly simplified the use of dependency in your applications. This made it easier to decouple application components.  
  
Dependency resolution is added to MVC 3 for providing, or we can say for supporting, the following functionalities in your application:

* Controllers
* Action filters
* Views
* Value providers
* Metadata functionality
* Model validation
* Model binders

**Global Helper Method**

View Without Controller Action in MVC

To fix this issue, we can use a simple try-catch block and redirect the user on a PageNotFound view; here's how.

sss

**Implementing Custom Authorization**  
  
Now I will put it into practice. I have created a simple CustomAuthorizeAttribute.  
  
A class is derived from the AuthorizeAttribute class (because we need the common behavior of Authentication). I just overrode the 3 base methods CustomAuthorizeAttribute, AuthorizeCore and HandleUnauthorizedRequest.  
  
Here is the code:

1. **public** **class** CustomAuthorizeAttribute : AuthorizeAttribute
2. {
3. Entities context = **new** Entities(); // my entity
4. **private** **readonly** **string**[] allowedroles;
5. **public** CustomAuthorizeAttribute(**params** **string**[] roles)
6. {
7. **this**.allowedroles = roles;
8. }
9. **protected** **override** **bool** AuthorizeCore(HttpContextBase httpContext)
10. {
11. **bool** authorize = **false**;
12. **foreach** (var role **in** allowedroles)
13. {
14. var user = context.AppUser.Where(m => m.UserID == GetUser.CurrentUser/\* getting user form current context \*/ && m.Role == role &&
15. m.IsActive == **true**); // checking active users with allowed roles.
16. **if** (user.Count() > 0)
17. {
18. authorize = **true**; /\* return true if Entity has current user(active) with specific role \*/
19. }
20. }
21. **return** authorize;
22. }
23. **protected** **override** **void** HandleUnauthorizedRequest(AuthorizationContext filterContext)
24. {
25. filterContext.Result = **new** HttpUnauthorizedResult();
26. }
27. }

To fix this issue, we can use a simple try-catch block and redirect the user on a PageNotFound view; here's how.  
  
<http://www.dotnet-tricks.com/Tutorial/mvc/3WDE140613-Different-ways-of-rendering-layouts-in-Asp.Net-MVC.htmlhttp://www.dotnet-tricks.com/Tutorial/mvc/3WDE140613-Different-ways-of-rendering-layouts-in-Asp.Net-MVC.htmlhttp://www.dotnet-tricks.com/Tutorial/mvc/3WDE140613-Different-ways-of-rendering-layouts-in-Asp.Net-MVC.html>

http://www.dotnet-tricks.com/Tutorial/mvc/3WDE140613-Different-ways-of-rendering-layouts-in-Asp.Net-MVC.html

Different ways of rendering layouts in Asp.Net MVC

1. **@{**
2. **var controller = HttpContext.Current.Request.RequestContext.RouteData.Values["Controller"].ToString();**
4. **string layout = "";**
5. **if (controller == "Admin")**
6. **{**
7. **layout = "~/Views/Shared/\_AdminLayout.cshtml";**
8. **}**
9. **else**
10. **{**
11. **layout = "~/Views/Shared/\_Layout.cshtml";**
12. **}**
13. **Layout = layout;**
14. **}**

## Method 2 : Return Layout from ActionResult

1. **public ActionResult Index()**
2. **{**
3. **RegisterModel model = new RegisterModel();**
4. ***//TO DO:***
5. **return View("Index", "\_AdminLayout", model);**
6. **}**

## Method 3 : Define Layout with in each view on the top

We can also override the default layout rendering by defining the layout on the view by using the below code:

1. **@{**
2. **Layout = "~/Views/Shared/\_AdminLayout.cshtml";**
3. **}**

## Routing

Routing is the first step in ASP.NET MVC pipeline. typically, it is a pattern matching system that matches the incoming request to the registered URL patterns in the Route Table.

The UrlRoutingModule(System.Web.Routing.UrlRoutingModule) is a class which matches an incoming HTTP request to a registered route pattern in the RouteTable(System.Web.Routing.RouteTable).

When ASP.NET MVC application starts at first time, it registers one or more patterns to the RouteTable to tell the routing system what to do with any requests that match these patterns. An application has only one RouteTable and this is setup in the Application\_Start event of Global.asax of the application.

1. **public class RouteConfig**
2. **{**
3. **public static void RegisterRoutes(RouteCollection routes)**
4. **{**
5. **routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");**
7. **routes.MapRoute(**
8. **name: "Default",**
9. **url: "{controller}/{action}/{id}",**
10. **defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }**
11. **);**
12. **}**
13. **}**
14. **protected void Application\_Start()**
15. **{**
16. ***//Other code is removed for clarity***
17. **RouteConfig.RegisterRoutes(RouteTable.Routes);**
18. **}**

CRUD Operations Using the Repository Pattern in MVC

<http://www.codeproject.com/Articles/644605/CRUD-Operations-Using-the-Repository-Pattern-in-MV>

#### Overview of the Repository Pattern

The Repository pattern is intended to create an abstraction layer between the data access layer and the business logic layer of an application. It is a data access pattern that prompts a more loosely coupled approach to data access. We create the data access logic in a separate class, or set of classes, called a repository, with the responsibility of persisting the application's business model.

In this article we will implement a "One-per business model" approach to design a repository in which there is a repository class for each entity type. For the Book entity type we'll create a repository interface and a repository class. When we instantiate the repository in our controller, we'll use the interface so that the controller will accept a reference to any object that implements the repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework.

MVC controllers interact with repositories to load and persist an application business model. By taking advantage of dependency injection (DI), repositories can be injected into a controller's constructor. The following diagram shows the relationship between the repository and the Entity Framework data context, in which MVC controllers interact with the repository rather than directly with Entity Framework.

