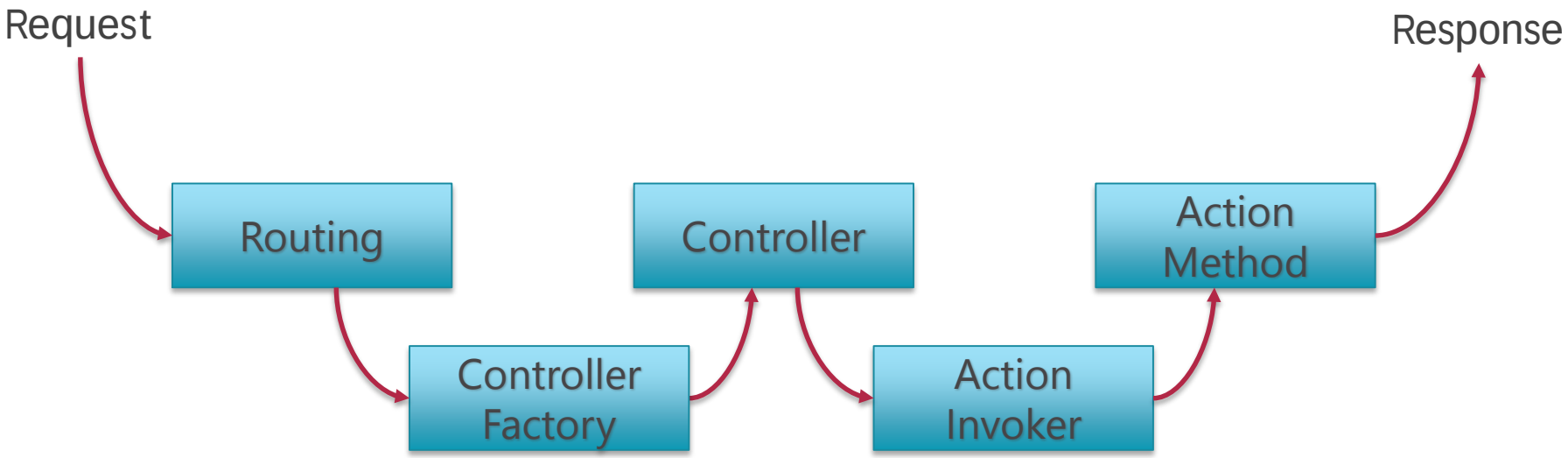
<epam>

Controllers

ASP.NET MVC – REVIEW

SEPT 2, 2016

MVC – Basic flow



Plan

- Controller Factory
  - Custom Controller Factory
  - Built-in Controller Factory
- Action Invoker
  - Custom Action Invoker
  - Built-in Action Invoker
  - Action Name
  - Action Selectors
- Specialized Controllers
  - Sessionless Controller
  - Asynchronous Controller

Controllers

- Controller Factory
- Action Invoker

Set up a project

1. Create an empty ASP.NET Web Application with MVC file structure
2. Add ‘Result.cs’ to Models with properties as below:

```
4 references | 0 changes | 0 authors, 0 changes
public string ControllerName { get; set; }
4 references | 0 changes | 0 authors, 0 changes
public string ActionName { get; set; }
```

3. Add ‘Result.cshtml’ as shared strongly typed view with ‘Result.cs’ model.

Set up a project

4. Add ‘Product’ and ‘Customer’ controllers.
5. Add ‘Index’ and ‘List’ views for each controller that will return ViewResult as below:

```
return View("Result", new Result
{
    ControllerName = "ControllerName",
    ActionName = "ActionName"
});
```

Custom Controller Factory

The best way to understand how something works is to disassemble it. In our case is to make it by our own.

1. Add ‘Infrastructure’ folder to your solution.
2. Add ‘CustomControllerFactory’ class.
3. Implement ‘IControllerFactory’ interface.

Custom Controller Factory

```
0 references | 0 changes | 0 authors, 0 changes
public SessionStateBehavior GetControllerSessionBehavior(
    RequestContext requestContext,
    string controllerName)
{
    return SessionStateBehavior.Default;
}

0 references | 0 changes | 0 authors, 0 changes
public void ReleaseController(IController controller)
{
    IDisposable disposable = controller as IDisposable;
    disposable?.Dispose();
}
```

Custom Controller Factory

```
public IController CreateController(
    RequestContext requestContext, string controllerName)
{
    Type targetType = null;
    switch (controllerName)
    {
        case "Product":
            targetType = typeof(ProductController);
            break;
        case "Customer":
            targetType = typeof(CustomerController);
            break;
        default:
            requestContext.RouteData.Values["controller"] =
            "Product";
            targetType = typeof(ProductController);
            break;
    }
    return targetType == null
        ? null
        : (IController) DependencyResolver.Current.GetService(targetType);
}
```

Custom Controller Factory - Request Context

- **RequestContext**
  - Member of System.Web.Routing
  - Encapsulates information about an HTTP request that matches a defined route.

Custom Controller Factory - Request Context

- **HttpContext** (type HttpContextBase)
  - Provides information about the HTTP request
- **RouteData** (type RouteData)
  - Provides information about the route that matches the request

Custom Controller Factory

Contras:

- Logic to locate controllers dynamically is needed
- Logic should be able to locate controllers consistently in different namespaces

Custom Controller Factory

When you are creating a Custom Controller Factory a good practice is to define a special controller that renders an error message when the request you are processing does not target any of the controllers in your project.

Custom Controller Factory

- requestContext.RouteData.Values

```
requestContext.RouteData.Values["controller"] = "Product";
```

This change will cause the MVC Framework to search for views associated with the fallback controller and not the controller that the routing system has identified based on the URL that the user requested.

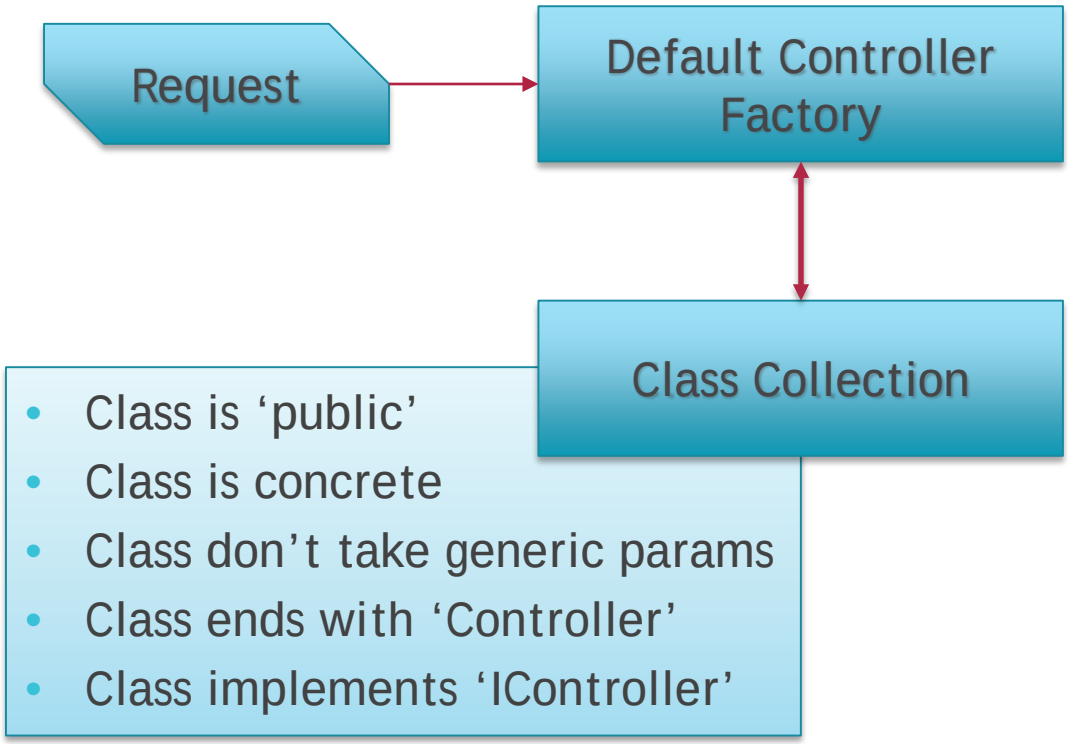
Custom Controller Factory

To start using Custom Controller Factory you need to register it using ‘ControllerBuilder’.

```
public class MvcApplication : System.Web.HttpApplication
{
    0 references | 0 changes | 0 authors, 0 changes
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        ControllerBuilder.Current.SetControllerFactory(
            new CustomControllerFactory());
    }
}
```

Default Controller Factory



Default Controller Factory

- Prioritizing Namespaces

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RouteConfig.RegisterRoutes(RouteTable.Routes);

    ControllerBuilder.Current.DefaultNamespaces
        .Add("EnchancedControllers.*");
    ControllerBuilder.Current.DefaultNamespaces
        .Add("SomeProject.Web.*");
}
```

Note. ‘EnchancedControllers’ and ‘SomeProject.Web’ will have same priority!

Default Controller Factory

Customization:

- Using the Dependency Resolver
- Using a Controller Activator
- Overriding Default Controller Factory Methods

Default Controller Factory

Using a Controller Activator

```
public class CustomControllerActivator : IControllerActivator
{
    0 references | 0 changes | 0 authors, 0 changes
    public IController Create(RequestContext requestContext,
        Type controllerType)
    {
        if (controllerType == typeof (ProductController))
        {
            controllerType = typeof (CustomerController);
        }
        return (IController)
            DependencyResolver.Current.GetService(controllerType);
    }
}
```

Default Controller Factory

Using a Controller Activator

```
ControllerBuilder.Current.SetControllerFactory(
    new DefaultControllerFactory(
        new CustomControllerActivator()));
```

Default Controller Factory

Overriding Default Controller Factory Methods

Method	Result	Description
CreateController	IController	Determine which type should be instantiated, and then gets a controller object by passing the result to the 'GetControllerInstance' method.
GetControllerType	Type	Maps requests to controller types.
GetControllerInstance	IController	Creates an instance of a specified type.

Custom Action Invoker

- Controller factory is responsible for creation of an instance of a class.
- Action Invoker is responsible for invoking and action of created instance.

Note. If you create a controller directly from the IController interface, then you are responsible for executing the action yourself. Action invokers are part of the functionality included in the Controller class.

Custom Action Invoker

```
public class CustomActionInvoker : IActionInvoker
{
    0 references | 0 changes | 0 authors, 0 changes
    public bool InvokeAction(ControllerContext controllerContext,
        string actionName)
    {
        if (actionName != "Index") return false;

        controllerContext.HttpContext.
            Response.Write("This is output from the Indexaction");
        return true;
    }
}
```

Custom Action Invoker

- **InvokeAction**, return type is bool
  - **True** indicates that the action was found and invoked
  - **False** indicates that the controller has no matching action. (404—Not found)

Note. Action invoker doesn't care about the methods in the controller class. In fact, it deals with actions itself.

Custom Action Invoker

```
1 reference | 0 changes | 0 authors, 0 changes
public class ActionInvokerController : Controller
{
    0 references | 0 changes | 0 authors, 0 changes
    public ActionInvokerController()
    {
        this.ActionInvoker =
            new CustomActionInvoker();
    }
}
```

Built-in Action Invoker

public ControllerActionInvoker()

- Member of System.Web.Mvc
- Built-in class that is responsible for invoking the action methods of a controller.

Built-in Action Invoker

Restrictions

- The method must be public.
- The method must not be static.
- The method must not be present in System.Web.Mvc.Controller or any of its base classes.
- The method must not have a special name. (constructors, property and event accessors are excluded)

Note. Methods that have generic parameters meet all of the criteria, but will throw an exception if you try to invoke such a method to process a request

Custom Action Name

Custom Action Name is an attribute that can be applied to an action to override it's name:

```
[ActionName("Enumerate")]
0 references | 0 changes | 0 authors, 0 changes
public ViewResult List()
{
    return View("Result", new Result
    {
        ControllerName = "Customer",
        ActionName = "List"
    });
}
```

Custom Action Name

Reasons why you might want to override a method name:

- To get an action name that wouldn’t be legal as a C# method name. ([ActionName("User-Registration")])
- To get actions with the same name and parameters but in response to different HTTP request types.

Action Method Selection

Action method selection

Mechanism that helps MVC Framework selecting the appropriate action with which to process a request if a controller contains several actions with the same name.

HttpPost, **HttpGet** attributes are the examples of action method selection.

Action Method Selection

NonAction attribute

```
[NonAction]
0 references | 0 changes | 0 authors, 0 changes
public ActionResult MyAction()
{
    return View("Result");
}
```

Custom Action Method Selector

public abstract class **ActionMethodSelectorAttribute**

- Derived from System.Attribure.
- Member of System.Web.Mvc.
- Represents an attribute that is used to influence the selection of an action method.

Custom Action Method Selector

```
0 references | 0 changes | 0 authors, 0 changes
public ActionResult Index()
{
    return View("Result",
        new Result { ControllerName = "Home", ActionName = "Index" });
}

[ActionName("Index")]
0 references | 0 changes | 0 authors, 0 changes
public ActionResult LocalIndex()
{
    return View("Result",
        new Result { ControllerName = "Home", ActionName = "LocalIndex"});
}
```

Custom Action Method Selector

```
public class LocalAttribute : ActionMethodSelectorAttribute
{
    0 references | 0 changes | 0 authors, 0 changes
    public override bool IsValidForRequest(
        ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return controllerContext.HttpContext.Request.IsLocal;
    }
}
```

Custom Action Method Selector

```
0 references | 0 changes | 0 authors, 0 changes
public ActionResult Index()
{
    return View("Result",
        new Result { ControllerName = "Home", ActionName = "Index" });
}

[Local]
[ActionName("Index")]
0 references | 0 changes | 0 authors, 0 changes
public ActionResult LocalIndex()
{
    return View("Result",
        new Result { ControllerName = "Home", ActionName = "LocalIndex"});
}
```

Built-in Action Invoker

Action invoker process:

- The invoker discards any method based on name.
- The invoker discards any method that has an action method selector attribute that returns false for the current request.
- The invoker checks methods with selectors
    - If one left then it is invoked
    - If multiple are left then an error message is thrown
- The invoker checks methods without selectors
    - If one left the it is invoked
    - If multiple are left the an error message is thrown

Built-in Action Invoker

If no suitable method is found ‘InvokeAction’ method will return ‘false’ and ‘Controller’ class will call ‘HandleUnknownAction’ method. ‘HandleUnknownAction’ will return ‘404 – Not Found’ response.

```
0 references | 0 changes | 0 authors, 0 changes
protected override void HandleUnknownAction(string actionName)
{
    Response.Write(string.Format("You requested the {0} action",
    actionName));
}
```

Specialized Controllers

Improving Performance with Specialized Controllers

- Using Sessionless Controllers
- Using Asynchronous Controllers

Sessionless Controllers

```
0 references | 0 changes | 0 authors, 0 changes
public SessionStateBehavior GetControllerSessionBehavior(
    RequestContext requestContext,
    string controllerName)
{
    return SessionStateBehavior.Default;
}
```

Sessionless Controllers

enum *SessionStateBehavior*

- Default – Use the default ASP.NET behavior, which is to determine the session state configuration from HttpContext.
- Required – Full read-write session state is enabled.
- ReadOnly – Read-only session state is enabled.
- Disabled – Session state is disabled entirely.

Sessionless Controllers

Managing session state using Custom Controller Factory:

```
0 references | 0 changes | 0 authors, 0 changes
public SessionStateBehavior GetControllerSessionBehavior(RequestContext
    requestContext, string controllerName)
{
    switch (controllerName)
    {
        case "Home":
            return SessionStateBehavior.ReadOnly;
        case "Product":
            return SessionStateBehavior.Required;
        default:
            return SessionStateBehavior.Default;
    }
}
```

Sessionless Controllers

Managing session state using Default Controller Factory:

```
[SessionState(SessionStateBehavior.Disabled)]
0 references | 0 changes | 0 authors, 0 changes
public class FastController : Controller
{
    // GET: Fast
    0 references | 0 changes | 0 authors, 0 changes
    public ActionResult Index()
    {
        return View("Result", new Result
        {
            ControllerName = "Fast ",
            ActionName = "Index"
        });
    }
}
```

Sessionless Controllers

```
[SessionState(SessionStateBehavior.Disabled)]
```

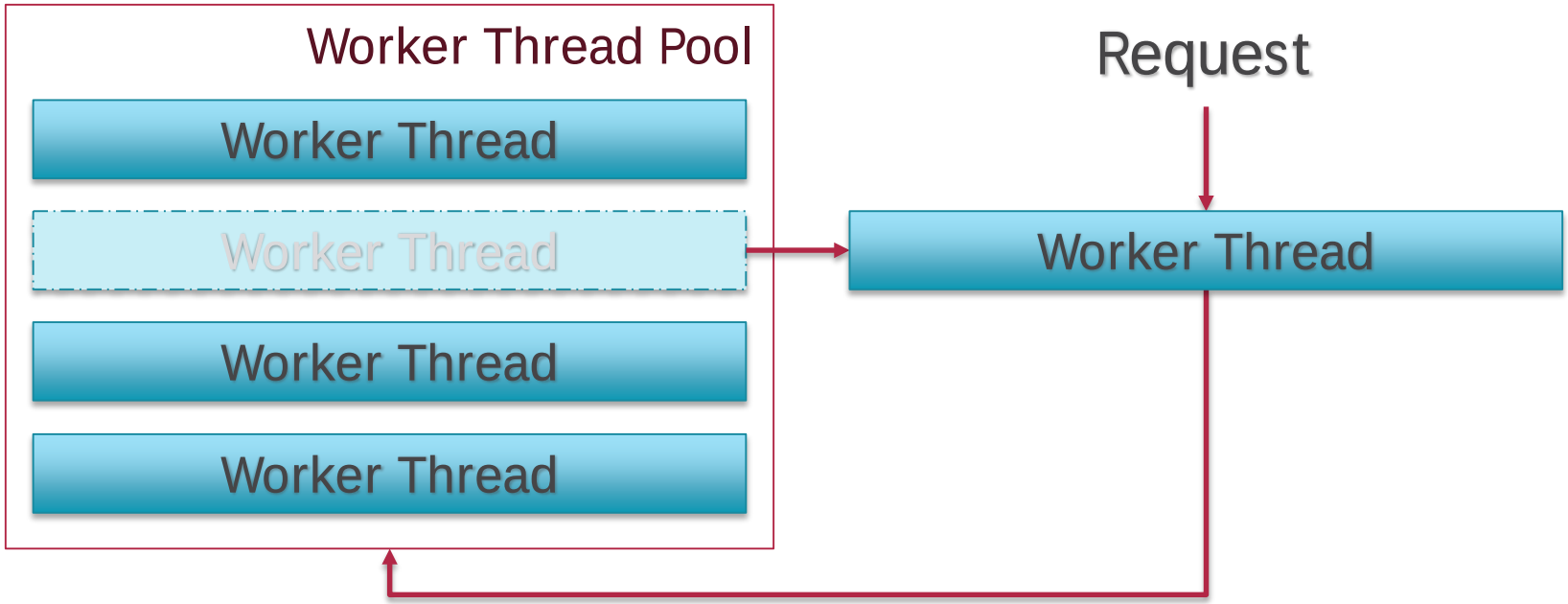
Session state will be applied to all the controller action.

```
Session["Message"] = "Some message";
Message: @Session["Message"]
```

Note. When session state is Disabled, the Session property returns null.

Note. To simply pass data from the controller to the view, consider using the View Bag feature.

Asynchronous Controllers



Asynchronous Controllers

Two key benefits of using thread pools:

- By reusing worker threads, you avoid the overhead of creating a new one each time you process a request.
- By having a fixed number of worker threads available, you avoid the situation where you are processing more simultaneous requests than your server can handle.

Note. Asynchronous controllers are useful only for actions that are I/O- or network-bound and not CPU-intensive.

Asynchronous Controllers

```
public class RemoteService
{
    1 reference | 0 changes | 0 authors, 0 changes
    public string GetRemoteData()
    {
        Thread.Sleep(2000);
        return "Hello from the other side of the world";
    }
}
```

Asynchronous Controllers

```
public class RemoteDataController : Controller
{
    0 references | 0 changes | 0 authors, 0 changes
    public ActionResult Data()
    {
        RemoteService service = new RemoteService();
        string data = service.GetRemoteData();
        return View((object)data);
    }
}
```

Asynchronous Controllers

Creating Asynchronous Controllers:

- Implement the ‘System.Web.Mvc.Async.IAsyncController’ interface.
- Use of ‘await’ and ‘async’ keywords in a regular controller.

Asynchronous Controllers

```
public async Task<ActionResult> Data()
{
    var data = await Task<string>.Factory
        .StartNew(() => new RemoteService().GetRemoteData());
    return View((object)data);
}
```

Asynchronous Controllers

```
public class RemoteService
{
    1 reference | 0 changes | 0 authors, 0 changes
    public string GetRemoteData()...
    1 reference | 0 changes | 0 authors, 0 changes
    public async Task<string> GetRemoteDataAsync()
    {
        return await Task<string>.Factory.StartNew(() => {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        });
    }
}
```

Asynchronous Controllers

```
public async Task<ActionResult> ConsumeAsyncMethod()
{
    var data = await new RemoteService().GetRemoteDataAsync();
    return View("Data", (object)data);
}
```

Task

Create Web app with features as below:

- HomeController – Display some info data (sessionless)
- Customer/User Controller – Can be referenced either as Customer or User. Should have actions:
    - Add-User – HttpPost is async and returns User-List with new user.
    - User-List – HttpGet returns view, HttpPost returns json
- AdminController – Actions can be accessed only locally. Holds an ability to manage user list.
- BaseController – It should process HandleUnknownAction and deliver custom 404 page.

Note. Implement 2 version of app. Home and Admin controllers features should be implemented using attributes and custom controller factory.