19

Lightweight Controllers

This chapter covers

* Using lightweight controllers to simplify programming
* Deriving action results to apply common behavior
* Managing common view data without filter attributes
* Using an application bus

Do you remember those swollen and unwieldy Page\_Load methods in Web Forms? Those methods can quickly grow out of control and stage a servant's revolt against your code base. Controller actions are dangerous too. Nestled snugly between the model and view, controllers are an easy place to put decision-making code, and they are often mistaken to be a good place to put that logic. And it is quite convenient, at first. It just takes two lines of code to build a select list in an action method. And adding a filter attribute to the controller is an easy way to manage global data for a master page. But these techniques don't scale with greater complexity. Orchestrating a process to find the specified order, authorize it, transmit it to the shipping service and email a receipt to the user, before redirecting the client to the confirmation page? That's too much for our controller to handle.

19.2 Why lightweight controllers

It's important to focus on keeping controllers lightweight because without intentional, continuous refactoring they will become bloated. In many contexts, and especially in software design, bloat is bad. One symptom of a bloated class is that it's hard to understand exactly, precisely, only what it's doing. It's hard to understand because it's doing many things.

19.2.1 Maintainability

As code becomes hard to understand it becomes hard to change. As code becomes hard to change it becomes a minefield of errors and rework and headaches. Deep technical analysis must be rendered for each seemingly simple enhancement or bug fix, because the developer is unsure what the ramifications of her change will be.

The Single Responsibility Principle (SRP)

The guiding principle behind this is the SRP. Basically, SRP states that a class should have one and only one responsibility. Another way to look at it is that a class should only have one reason to change. If you find that a class has potential to be changed for nonrelated reasons, the class is probably doing too much. A common violation of SRP is mixing data access with business logic. For example, a Customer probably shouldn’t have a Save() method.

SRP is a core concept of good object-oriented design, and its application can help your code become more maintainable. SRP is sometimes referred to as Separation of Concerns (SoC). You can read more about SRP/SoC from Bob Martin’s excellent article on the subject: http://www.objectmentor.com/resources/articles/srp.pdf

Not only that, but bloat makes understanding how to make a change difficult. Without clear responsibilities, a change could potentially happen anywhere. As developers we don't want building software to be a guessing game, where we blindly slap logic into action methods. We want to create a system where software design exists apart from controllers so that we don't struggle when working with our source code.

19.2.2 Testability

The best way to ensure it's easy to work with our source code is to practice test driven development (TDD). When we do TDD, we work with our source code before it exists. And hard to test classes, including controllers, are immediately suspect as flawed. Testing friction - problems writing tests or with test management - is a clear and convincing indicator that the software's design has room for improvement. Simple, lightweight controllers are easy to test. We'll discuss TDD in detail in Chapter 26.

19.2.3 Focusing on the Controller's Responsibility

A quick way to lighten the controller's load is to simply remove responsibilities from it. Consider the following burdened action, shown in listing 19.1.

Listing 19.1 A heavyweight controller

public RedirectToRouteResult Ship(int orderId)

{

User user = \_userSession.GetCurrentUser();

Order order = \_repository.GetById(orderId);

if (order.IsAuthorized) #1

{

ShippingStatus status = \_shippingService.Ship(order);

if (!string.IsNullOrEmpty(user.EmailAddress)) #2

{

Message message = \_messageBuilder

.BuildShippedMessage(order, user);

\_emailSender.Send(message);

}

if (status.Successful)

{

return RedirectToAction("Shipped", "Order", new {orderId});

}

}

return RedirectToAction("NotShipped", "Order", new {orderId});

}

This action is doing a lot of work - it's incomprehensible at first glance. You can almost count its jobs by the number of if statements. Beyond its appropriate role as director of the storyboard flow of the user interface, our action is deciding if the Order is appropriate for shipping (1) and determining whether or not to send the User a notification email (2). Not only is it doing those things, but it's deciding how to do them - it's determining what it means for an Order to be appropriate for shipping and how the User object communicates it should be sent the message.

Logic like this - domain logic, business logic - should generally not be in a user interface class like a controller. It violates the single responsibility principle, obfuscating both the true intention of the domain and actual duties of this controller: redirecting to the proper action. Testing and maintaining an application written like this is difficult.

Cyclomatic Complexity: Source Code Viscosity

Cyclomatic complexity is a metric we can use to analyze the complexity of code. The more logical paths a method or function presents, the higher its cyclomatic complexity is. In order to fully understand the implication of a particular procedure, each logical path must be evaluated. For example, each simple if statement presents two paths - one when the condition is true and another when it's false. Functions with high cyclomatic complexity are more difficult to test and to understand and have been correlated with increased defect rates.

http://mng.bz/R4D3A simple refactoring that can ease this is called Refactor Architecture By Tiers. It directs the software designer to move processing logic out of the presentation tier into the business tier (http://mng.bz/crHL). After we remove the bloat, our action is much simpler.

Listing 19.2 After Refactoring Architecture By Tiers

public RedirectToRouteResult Ship(int orderId)

{

var status = \_orderShippingService.Ship(orderId); #1

if (status.Successful)

{

return RedirectToAction("Shipped", "Order", new {orderId});

}

return RedirectToAction("NotShipped", "Order", new {orderId});

}

Everything having to do with actually shipping the order and sending the notification has been moved out of the controller into a new class (1). The controller is left with the single responsibility of deciding where to redirect the client. The new class can fetch the Order, get the User and do all the rest.

But it's more than just a move. It's a semantic break that puts the onus of managing this task in the right place. This step has resulted in a clean abstraction that our controller can use to represent what it was doing before. Other logical endpoints can reuse it - other controllers or services may participate in the order shipping process. This new abstraction is clear. And our new abstraction can change internally without affecting the presentation duties of the controller. Refactoring doesn't get much simpler than this, but a simple change can result in a significantly lower cyclomatic complexity and ease the testing effort and maintenance burden associated with a complex controller. In the next sections we'll look at other ways of simplifying controllers.

19.3 Managing common view data

Complexity can easily sneak in to our controllers by way of filter attributes. Those seemingly harmless attributes can encapsulate vast amounts of data access and processing logic. We often see filter attributes used to provide common view data, but there's another technique that can provide the same functionality without a reliance on attributes. Listing 19.3 shows a controller action using an action filter attribute to add a subtitle to ViewData.

Listing 19.3

[SubtitleData] #1

public ActionResult About()

{

return View();

}

#1 Applying the attribute

Whenever the action in listing 19.3 is invoked, the action filter attribute, shown in listing 19.4, will execute.

Listing 19.4

public class SubtitleDataAttribute : ActionFilterAttribute #1

{

public override void

OnActionExecuted(ActionExecutedContext filterContext)

{

var subtitle = new SubtitleBuilder(); #2

filterContext.Controller.ViewData["subtitle"]

= subtitle.Subtitle(); #3

}

}

#1 Derived from ActionFilterAttribute

#2 Instantiating a helper class

#3 Adding to ViewData

SubtitleDataAttribute uses SubtitleBuilder to retrieve the proper subtitle, and places the subtitle in ViewData. Attributes are special classes that do not afford the developer much control. We cannot instantiate them, nor can we supply them with non-constant parameters, so our action filter attribute must be responsible for instantiating any helper classes it needs (#2).

Dependencies

When a class we're writing needs help from another class, our class is dependent on that other class. We call those collaborators dependencies. Managing dependencies is a responsibility in and of itself. A class is doing too much (and violating the single responsibility principle) when it is responsible for managing its dependencies along with its own behavior. One common technique to remove this burden is constructor injection - providing the dependency to our class by passing (or injecting) it as a constructor argument. This way callers know exactly what our class depends on before they can instantiate it. We can also provide dummy implementations of the dependency during testing. The end result is a number of classes with single, focused responsibilities. When applied correctly, this technique transforms our application from a procedural uphill walk to a tightly choreographed ballet of objects.

Because SubtitleDataAttribute is responsible for instantiating its helpers, it has a compile-time coupling to SubtitleBuilder (evidenced by the new keyword). Another drawback to action filter attributes is the work involved in applying them. You must remember to apply them to each action on which they're needed. One solution to this could be to create a layer supertype controller (a base controller) and apply the filter attribute to that. Then all controllers that wanted the action filter's behavior could simply derive from that layer supertype.

The problem with relying on inheritance to solve this problem is that it couples our controller to the base type. Inheritance is a compiled condition, which makes run time changes difficult. And even compile-time changes are hard: if the layer supertype changes, all derivations must change. In cases like these we favor composition over inheritance.

By extending the default ControllerActionInvoker (mentioned briefly in chapter 9) we can compose action filters at run time without using attributes on actions, controllers, or a layer supertype controller. In listing 19.5 we'll look at extending ControllerActionInvoker to allow us to apply action filters without attributes.

Listing 19.5 Extending ControllerActionInvoker to provide custom action filters

public class AutoActionInvoker : ControllerActionInvoker #1

{

private readonly IAutoActionFilter[] \_filters; |#2

|#2

public AutoActionInvoker(IAutoActionFilter[] filters) |#2

{ |#2

\_filters = filters; |#2

} |#2

protected override FilterInfo GetFilters

(ControllerContext controllerContext,   
 ActionDescriptor actionDescriptor)

{

FilterInfo filters = |#3  
 base.GetFilters(controllerContext, |#3

actionDescriptor); |#3

|#3

foreach (IActionFilter filter in \_filters) |#3

{ |#3

filters.ActionFilters.Add(filter); |#3

} |#3

return filters;

}

#1 Derive from ControllerActionInvoker

#2 Array of filters is injected

#3 Use custom and default filters

The controller action invoker we'll use will take an array of custom action filters as a constructor parameter (2) and apply each of them to the action when its invoked (3).

DefaultControllerFactory

Controllers are instantiated by a special class called DefaultControllerFactory. It's possible to derive from this class to create our own controller factory. A custom controller factory allows ASP.NET MVC 2 developers to customize the instantiation of controllers.

In listing 19.6 we'll set our new action invoker as the default for each controller as it's created in the controller factory.

Listing 19.6 Using our custom action invoker with a custom controller factory

public class ControllerFactory : DefaultControllerFactory

{

public static Func<Type, object> GetInstance = |#1

type => Activator.CreateInstance(type); |#1

protected override IController GetControllerInstance(

RequestContext requestContext, Type controllerType)

{

if (controllerType != null)

{

var controller = (Controller) GetInstance(controllerType);

controller.ActionInvoker = (IActionInvoker) |#2

GetInstance(typeof (AutoActionInvoker)); |#2

return controller;

}

return null;

}

}

#1 Factory function

#2 Setting our custom action invoker

We need a factory function to provide an instance for a given type (1). Because the specific controller type we need isn't known until runtime, we can't pass the controller as a dependency to the constructor of our controller factory. But we can set this function to a function that knows about all the controller types in our system.

Inversion Of Control

We've seen that a class's dependencies should be managed from outside and not by the dependent class itself. As an application grows, it's dependency graph - the tree of objects that depend on each other - can grow to a level of complexity that isn't reasonable for the developer to manually maintain. But utility libraries exist that use reflection, conventions and configuration to keep track of dependencies in our objects. We can use these libraries to instantiate classes with their entire dependency graphs in place. Doing this - relinquishing the responsibility of managing our dependencies - is inversion of control. There are several popular inversion of control libraries available to .NET developers: Microsoft Unity (http://mng.bz/DNro), StructureMap (http://structuremap.sourceforge.net) and Castle Windsor (http://mng.bz/CKKn) are recommended.

In order to leverage our inversion of control tool in our controller factory, we have to set the factory function to the tool's instantiating function. This should happen when the application is first started, and we do this in listing 19.7.

Listing 19.7 Setting the factory function to use the IOC tool

protected void Application\_Start()

{

// ...

RegisterRoutes(RouteTable.Routes);

ControllerFactory.GetInstance = |#1

type => ObjectFactory.GetInstance(type); |#1

ControllerBuilder.Current. |#2

SetControllerFactory(new ControllerFactory()); |#2

}

#1 Setting the factory function

#2 Setting our controller factory as the default

First, we set the controller factory's static factory function (1) to the inversion of control tool's automatic factory method. In order to use our custom controller factory, we then call the SetControllerFactory method on the ControllerBuilder to replace the default controller factory with our own one (2). Now our controller factory will use our inversion of control tool to instantiate controllers, our custom invoker, and any action filters. Finally, we use a special interface and abstract base class to denote the action filters we want to apply in this fashion. This is shown in listing 19.8.

Listing 19.8 An interface to define our custom filter

public interface IAutoActionFilter : #1

IActionFilter

{

}

public abstract class BaseAutoActionFilter : #2

IAutoActionFilter

{

public virtual void OnActionExecuting

(ActionExecutingContext filterContext)

{

}

public virtual void OnActionExecuted

(ActionExecutedContext filterContext)

{

}

}

Our interface, IAutoActionFilter, implements IActionFilter. (1) BaseAutoActionFilter implements IAutoActionFilter and provides implementations of its methods that do nothing (2). These no-op methods will allow further derivations to only override the method they wish to use without having to implement the other method of IActionFilter. It's just a handy shortcut. In listing 19.9 we get to actually implement our custom filter, to replace the attribute-based one in listing 19.4.

Listing 19.9 Our custom, non-attribute-based, action filter

public class SubtitleData : BaseAutoActionFilter

{

readonly ISubtitleBuilder \_builder;

public SubtitleData(ISubtitleBuilder builder) #1

{

\_builder = builder;

}

public override void OnActionExecuted(

ActionExecutedContext filterContext)

{

filterContext.Controller.ViewData["subtitle"] =

\_builder.AutoSubtitle();

}

}

In this version of the action filter we can take the dependency as a constructor parameter (supplied automatically by our inversion of control tool) (1). Finally - a clean action filter: testable, lightweight, with managed dependencies and no clunky attributes.

This seems like a lot of work, but once you get the concept in place, adding filter attributes is simple: just derive from BaseAutoActionFilter. In the next section we'll eliminate another pesky attribute from our actions.

19.2 Derived action results

One possible use for action filter attributes is to perform post-processing on the ViewData provided by the controller to the view. In the example code for chapter 18 we used an action filter attribute that used AutoMapper to translate source types to destination types. This filter attribute is shown in listing 19.10.

Listing 19.11 AutoMapModelAttribute

public class AutoMapModelAttribute : ActionFilterAttribute #A

{

private readonly Type \_destType;

private readonly Type \_sourceType;

public AutoMapModelAttribute(Type sourceType, Type destType) #B

{

\_sourceType = sourceType;

\_destType = destType;

}

public override void

OnActionExecuted(ActionExecutedContext filterContext)

{

object model = filterContext.Controller.ViewData.Model;

object viewModel =

Mapper.Map(model, \_sourceType, \_destType); |#C

filterContext.Controller.ViewData.Model = viewModel; |#C

}

}

#A Derived from ActionFilterAttribute

#B Accepts type parameters

#C Uses AutoMapper to map ViewData.Model

By decorating an action method with this attribute we direct AutoMapper to transform ViewData.Model. This attribute is providing critical functionality — our views will not work if the attribute is improperly applied. One symptom of a bloated controller is that it contains processing logic that's hard to test and verify. Applying an attribute like the one we used to map ViewData is simply not testable.

Instead of using a filter attribute, what if we derived from ViewResult with the extra behavior of applying an AutoMapper map to ViewData.Model before regular execution? Then we could not only verify the correct model was initially set, but also verify that AutoMapper will map to the correct destination type. You can create many different action results like this - the key is to expose testable state, in this case, the destination type to which we'll map. Here's AutoMapViewResult in listing 19.11.

Listing 19.11 An action result that applies AutoMapper to the Model

public class AutoMappedViewResult : ViewResult #1

{

public static Func<object, Type, Type, object> Map = #2

(a, b, c) =>

{

throw new InvalidOperationException(

@"The Mapping function must be

set on the AutoMapperResult class");

};

public AutoMappedViewResult(Type type) #3

{

DesinationType = type; #4

}

public Type ViewModelType { get; set; }

public override void ExecuteResult

(ControllerContext context)

{

ViewData.Model = Map(ViewData.Model, #5

ViewData.Model.GetType(),

DestinationType);

base.ExecuteResult(context); #6

}

}

#1 Deriving from ViewResult

#2 Will set with AutoMapper function

#3 Takes a Type argument

#4 For later verification

#5 Applies mapping function

#6 Finally executes normal ViewResult processing

All this class really does is apply a mapping function (which we'll set to be AutoMapper's mapping function) to ViewData.Model before continuing on with the regular ViewResult work (6). But we make sure to expose the destination type (4) so that we can verify it in unit tests. Unlike when using the attribute, we can know for sure that the action is mapping to the correct destination type. Shown in listing 19.12, with a helper function, we can easily use this result in our actions.

Listing 19.12 Using AutoMappedViewResult in an action

public AutoMappedViewResult Index()

{

var customer = GetCustomer();

return AutoMappedView<CustomerInfo>(customer); #1

}

public AutoMappedViewResult AutoMappedView<TModel>(object Model) #2

{

ViewData.Model = Model;

return new AutoMappedViewResult(typeof (TModel))

{

ViewData = ViewData,

TempData = TempData

};

}

Returning the right result is straightforward - it's like the normal ViewResult but we do have to supply the destination type, CustomerInfo, our presentation model (1). Our helper function (2) does the heavy ViewData and TempData lifting.

In the next section we'll take lighten our controller even further using an application bus and a simple abstraction around a common controller theme: controlling storyboard flow for success and failure.

19.4 Using an application bus

In large distributed systems eliminating dependencies isn't just a good idea, it's required. Architects designing these systems have come to understand they must create a myriad of atomic services that can be reused and composed by several applications, just like application architects design classes to be reused and composed inside programs. But unlike classes inside programs, services cannot be coupled to physical network locations or to specific programming platforms. When a system is composed of services spread across a large network, rather than a shared memory space, extreme flexibility in deployment and configuration is necessary.

The metaphor that best describes the way many distributed systems work together is sending and receiving messages. One application will send a command message to a bus. The bus, in service-oriented terms, is responsible for, among other things, routing the message to ensure it's handled by the appropriate recipient. Services share a message schema, but their implementations can vary widely, even as far as being developed on different platforms. As long as the recipient understands the message the services can work together. They don't need to depend on each other, just the bus. These systems are loosely coupled.

This is an gross oversimplification of message-based, service oriented architectures, but these distributed systems can provide insight into better ways of designing in-process applications like the traditional web sites we've been discussing.

What if, instead of depending on an IOrderShippingService, our controller in listing 19.2 sent a message to a bus?

Listing 19.13 Sending a message on an application bus

public class ExampleOrderController : Controller

{

readonly IBus \_bus;

public ExampleOrderController(IBus bus) #1

{

\_bus = bus;

}

public ActionResult Ship(int orderId)

{

var message = new ShipOrderMessage |#2

{ |#2

OrderId = orderId |#2

}; |#2

var result = \_bus.Send(message); |#3

if (result.Successful) |#4

{ |#4

return RedirectToAction |#4

("Shipped", "Order", new {orderId}); |#4

} |#4

return RedirectToAction |#4

("NotShipped", "Order", new {orderId}); |#4

}

}

#1 Injecting the IBus dependency

#2 Creating the command message

#3 Sending the message on the bus

#4 Processing the result

The controller in listing 19.13 doesn't call a method on IOrderShipping service, but instead sends a ShipOrderMessage to an application bus (3). The user interface here is completely decoupled from the specific processor of the command. The entire order shipping process could change, the responsible interface could change, and our controller would continue working correctly without modification.

The bus, on the other hand, needs a way to associate messages with their specific handlers. A distributed system would need something pretty fancy to route messages to different networked endpoints, but in-process applications can harness the type system and use it as a registry. Consider a simple IHandler<T> interface in listing 19.14.

Listing 19.14 IHandler<T> indicates a type can handle a message type

public interface IHandler<T> #1

{

Result Handle(T message); #2

}

#1 T is type of message

#2 Handles received messages

Implementers of this interface declare they can handle a specific message type. When the bus receives a ShipOrderMessage, it can look for an implementation of IHandler<ShipOrderMessage> and, using an inversion of control tool, instantiate the implementation and call Handle on it, passing in the message (an example of this is in the sample code for this chapter, but isn't necessary for the text). Listing 19.15 shows the concrete message handler.

Listing 19.15 Concrete message handler

public class ShipOrderHandler : IHandler<ShipOrderMessage> #1

{

readonly IRepository \_repository;

public ShipOrderHandler(IRepository repository)

{

\_repository = repository;

}

protected override Result Execute(ShipOrderMessage msg)

{

var order = \_repository.GetById<Order>(msg.OrderId);

order.Ship(); #2

\_repository.Save(order);

return Result.Success();

}

}

#1 Implement handler interface

#2 Order shipping logic

OrderShippingService has added itself to the bus' registry as the handler for messages of type ShipOrderMessage just by implementing IHandler<ShipOrderMessage> (1). The actual work is done in the Handle method, where it can use its own dependencies as needed.

While it's very useful (and cool) to decouple our business logic code from our user interface, this technique hasn't necessarily simplified our controller. Our cyclomatic complexity remains - we would still need to test what happens should the result succeed and should it fail. There's another abstraction to be extracted: the concept of success or failure can be baked in to our bus architecture. The idea is to set up an action result to handle sending the message. The action result can also check the result of the message dispatch and execute a nested action result function upon success or failure. The controller is still responsible for choosing the action result for success and the action result for failure, continuing in its role as the storyboard director. The complete action result is used in the sample code for this chapter, but we see a simplified CommandResult in listing 19.16.

Listing 19.16 A command executing action result

public class CommandResult : ActionResult

{

// ...

public override void Execute(ControllerContext context)

{

var bus = ObjectFactory.GetInstance<IBus>(); #1

var result = bus.Send(\_message); #2

if (result.Successful) #3

{

Success.ExecuteResult(context); #4

return;

}

Failure.ExecuteResult(context); #5

}

}

#1 IoC tool gets application bus

#2 Sending the message

#3 Checking the result

#4 Executing success action result

#5 If unsuccessful, execute failure action result

What's not shown in this listing is the constructor that takes functions that return action results for the success and failure case. These action results end up as the Success (4) and Failure (6) properties. Otherwise the semantics look the same as our controller in listing 19.13, except armed with this abstraction we can avoid repetitive code in each controller. Let's take a final look at our order shipping action, now using a special helper method to craft the CommandResult, in listing 19.17.

Listing 19.17 Using CommandResult in an action

public CommandResult Ship(int orderId)

{

var message = new ShipOrderMessage {OrderId = orderId};

return Command(message, #1

() => RedirectToAction("Shipped", new {orderId}), #2

() => RedirectToAction("NotShipped", new {orderId}));#3

}

#1 The ShipOrderMessage to be sent

#2 Defining the success action result

#4 Defining the failure action result

In our new Ship action, we call a helper method with arguments for the message (1), the success result (2), and the failure result (3). Because we are writing declarative code to define the message and action results, writing and testing controllers built with these techniques is dead simple. To test them, all we need to do is check the CommandResult's message and success and failure action results, verifying that the declared results are as expected. The test for this action is included in the sample code for this chapter.

Lastly, as a side benefit to sending commands through an application bus, we've established a very tiny logical pathway through which all business transactions move. We can take advantage of this pathway to set up a gate for stronger validation, auditing and other cross-cutting concerns.

19.5 Summary

In this chapter we applied a simple refactoring to remove business logic from the controller and into a useful abstraction. By properly managing our dependencies and adhering to object-oriented principles we are better equipped to craft well-designed software, functionality that can be easily verified to work. We extended ControllerActionInvoker and DefaultControllerFactory to manage action filters. Deriving from ActionResult allowed us to avoid repetitive code while not relying on filter attributes. Finally, we leveraged an application bus to write simple, declarative controller actions.

In the next chapter we'll explore how to organize large projects by using areas, a new ASP.NET MVC 2 feature.