19

Lightweight Controllers

This chapter covers

How lightweight controllers can make programming easier

Decorating action results to apply common behavior

How to manage common view data without filter attributes

Using a hub and spoke architecture

Controllers are dangerous. Because, snugly nestled between the model and view, they are an easy place to put decision-making code, controllers are often mistaken to be a good place to put decision-making code. And it is quite convenient, at first. Building a select list? Do it in the action - it's two lines of code! Harnessing global data for a master page? Put that in an action filter attribute, it's right there! Orchestrating a process to find the specified order, authorize it, transimit it to the shipping service and email a receipt to the user, before redirecting the client to the confirmation page? In the controller! Wait... what?

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

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. 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. When we TDD, we work with our source code before it exists. And hard to test classes, including controllers, are immediately suspect as flawed. Testing friction is a clear and convincing indicator that the software's design has room for improvement. Simple, lightweight controllers are easy to test.

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)

{

ShippingStatus status = \_shippingService.Ship(order);

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

{

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. You can almost count its jobs by its 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 and determining whether or not to send the User a notification email. 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 some 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://en.wikipedia.org/wiki/Cyclomatic_complexity>

A 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://www.refactoring.com/catalog/refactorArchitectureByTiers.html>). 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);

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. 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 interally without affecting the presentation duties of the controller. Refactoring doesn't get much simpler than this, but a simple change can result in signficantly less cyclomatic complexity and ease the testing effort and maintenance burden associated with a complex controller.

19.2 Decorating action results

19.2.1 AutoMapActionResult

19.3 Managing common view data

19.4 Leveraging an application bus for a simple hub and spoke architecture

19.5 Summary