Quick Tip: Move business logic out of your controllers

When you start building an ASP.NET application it seems perfectly reasonable to do this...

```
public class OrderController : Controller {

    // rest of code omitted

    public IActionResult ById(int id){
        var order = _dataContext.Orders.FirstOrDefault(x=>x.Id == id);
        if(order == null) {
            order = new Order();
        }
        return View(order);
    }
}
```

It works, your data is returned alongside your View, everyone's happy.

Over time, your controllers fill up with more and more of these actions, all doing roughly the same thing..

Get data, return data via a view.

But there's a problem.

A requirement comes in to return that same data but as JSON data so a client (e.g. javascript, Angular etc) can grab the data and do something with it.

What do you do?

We could...

1. Duplicate the method

So we end up with something like this.

```
public class OrderController : Controller {

    // rest of code omitted

    [HttpGet("byid")]
    public IActionResult ById(int id){
        var order = _dataContext.Orders.FirstOrDefault(x=>x.Id == id);
        if(order == null) {
            order = new Order();
        }
        return View(order);
    }

    [HttpGet("api/byid")]
    public IActionResult ApiById(int id){
```

```
        var order = _dataContext.Orders.FirstOrDefault(x=>x.Id == id);
        if(order == null) {
            return NotFound();
        }
        return Ok(order);
    }
}
```

The downside here comes when we inevitably need to extend or change this query.

Now we have two places to find and update our query code (or risk our API and MVC application's drifting apart in terms of behaviour).

2. Put in a conditional!

We could mash both together in one controller action...

```
public class OrderController : Controller {

    // rest of code omitted

    [HttpGet("byid")]
    public IActionResult ById(string id, bool api){
        var order = _dataContext.Orders.FirstOrDefault(x=>x.Id == id);
        if(api){
            if(order == null) {
                return NotFound();
            }
            return Ok(order);
        } else {
            if(order == null) {
                order = new Order();
            }
            return View(order);
        }
    }
}
```

But this is, well, ugly and already looks like a maintenance nightmare.

Spare a thought for the programmer who has to decipher this in every action in an MVC app!

Plus, we've lost the advantage of being able to have different routes. This approach gives us just one route and the need to pass a boolean (api) to control whether we get a page or data.

Now you've seen this, it can't be unseen, but please, don't try this at home :-)

3. Refactor (then duplicate the method)

Come back option 1, all is forgiven.

It seems option 1 is preferable in this case, so how can we avoid duplicating the data access code?

A refactor is in order.

If we can extract that code to somewhere else, we can call it from more than one controller action.

a) Extract Method

In Visual Studio, we can select the relevant lines and extract them into a method (CTRL + .). Alternatively you can "cut" the relevant line and move it yourself.

```
[HttpGet("byid")]
public IActionResult ById(int id)
{
    var order = OrderById(id);
    if (order == null)
    {
        order = new Order();
    }
    return View(order);
}

private Order OrderById(int id)
{
    return _dataContext.Orders.FirstOrDefault(x => x.Id == id);
}
```

Now we can easily call this method from two places.

```
[HttpGet("byid")]
public IActionResult ById(int id)
{
    var order = OrderById(id);
    if (order == null)
    {
        order = new Order();
    }
    return View(order);
}

[HttpGet("api/byid")]
public IActionResult ApiById(int id)
{
    var order = OrderById(id);
    if (order == null)
    {
        return NotFound();
    }
    return Ok(order);
}

private Order OrderById(int id)
{
    return _dataContext.Orders.FirstOrDefault(x => x.Id == id);
}
```

In this case, our query is pretty simple so this hasn't saved us much by way of lines of code in each controller action.

Crucially though, it's given us one place to change our query.

For example, if we needed to alter the query to include related data, we could now update just this method (and both controller actions would return the same thing).

```
private Order OrderById(int id)
{
    return _dataContext
        .Orders
        .Include(x=>x.Customer)
        .Include(x=>x.DeliveryAddress)
        .FirstOrDefault(x => x.Id == id);
}
```

b) Extract Class

Ideally, we'd want to move this method out of the controller entirely.

We can do this by moving our method to a dedicated class.

```
public class Orders {

    public Order OrderById(int id)
    {
        return _dataContext
            .Orders
            .Include(x=>x.Customer)
            .Include(x=>x.DeliveryAddress)
            .FirstOrDefault(x => x.Id == id);
    }

}
```

Now we're getting somewhere, we can call this from different controllers, or different Razor Pages (if you're using them instead of MVC).

```
public class SomeController {

    private Orders _orders;

    public SomeController(Orders orders){
        _orders = orders;
    }

    [HttpGet("byid")]
    public IActionResult ById(int id)
    {
        var order = _orders.OrderById(id);
        if (order == null)
        {
```

```
            order = new Order();
        }
        return View(order);
    }

}
```

Next Steps

So there you are, a quick way to avoid your controllers (or Razor pages) becoming bloated.

If you want to see this in action (in a real application), check out Practical ASP.NET Core MVC. There, you'll get to build an application from scratch, performing reactors like this as you go.

And if you want to take it up a notch I'm a big fan of the Mediator pattern and Jimmy Bogard's Mediatr library in particular.

You can see some examples of MediatR in action here.

Whichever approach you take, this is one of those situations where it's relatively painless to do early in a project and a much bigger job to go back and do when you have many controllers and a logic everywhere!

Often and early wins the day :-)

-- Jon