MICROSOFT ARCHITECT

By Joydip Kanjilal, Columnist, InfoWorld

JUL 8, 2019 3:00 AM PDT

How to write efficient controllers in ASP.Net Core

Take advantage of these best practices to create lean, clean, and flexible controllers in ASP.Net Core

You can write better controllers by adhering to the best practices. So-called “thin” controllers—i.e. controllers with less code and fewer responsibilities—are easier to read and maintain. And once your controllers are thin, you might not need to test them much. Rather, you can focus on testing the business logic and data access code. Another advantage of a thin or lean controller is that it is easier to maintain multiple versions of the controller over time.

This article discusses bad habits that make controllers fat and then explores ways to make the controllers lean and manageable. My list of best practices in writing controllers may not be comprehensive, but I have discussed the most important ones with relevant source code wherever applicable. In the sections that follow, we’ll examine what a fat controller is, why it is a code smell, what a thin controller is, why it is beneficial, and how to make the controllers thin, simple, testable, and manageable.

[Microsoft .Net 5 unites the .Net Framework and .Net Core: Find out what the merger of .Net Standard and .Net Core means for developers. | Learn how to get the most from .Net Framework and .Net Core from InfoWorld’s Microsoft Architect blog. | Keep up with hot topics in programming with InfoWorld’s App Dev Report newsletter.]

Remove data access code from controllers

When writing a controller, you should adhere to the Single Responsibility Principle, meaning the controller should have “one responsibility” or “one and only one reason to change.” In other words, you want to keep the reasons for changing the controller code to a minimum. Consider the following piece of code that shows a typical controller with data access logic.

```
public class AuthorController : Controller
{
    private AuthorContext dataContext = new AuthorContext();
    public ActionResult Index(int authorId)
    {
        var authors = dataContext.Authors
            .OrderByDescending(x=>x.JoiningDate)
            .Where(x=>x.AuthorId == authorId)
            .ToList();
        return View(authors);
    }
    //Other action methods
}
```

Note the usage of the data context instance to read data inside the action method. This is a violation of the Single Responsibility Principle and makes your controller bloated with code that shouldn’t reside there. In this example, we are using a DataContext (assume that we’re using Entity Framework Core) to connect to and work with the data in the database.

Tomorrow if you decide to change your data access technology (for better performance or whatever reason it may be), you would have to change your controllers as well. As an example, what if I want to use Dapper to connect to the underlying database? A better approach would be to use a repository class to encapsulate the data access logic (though I'm not a big fan of the repository pattern). Let's update the AuthorController with the following code.

```
public class AuthorController : Controller
{
    private AuthorRepository authorRepository = new AuthorRepository();
    public ActionResult Index(int authorId)
    {
        var authors = authorRepository.GetAuthor(authorId);
        return View(authors);
    }
    //Other action methods
}
```

The controller looks leaner now. Is this then the best approach to writing this controller? Not really. If your controller is accessing the data access components, it is doing too many things and hence violating the Single Responsibility Principle. Your controller should never have data access logic or code that accesses the data access components directly. Here's the improved version of the AuthorController class.

```
public class AuthorController : Controller
{
    private AuthorService authorService = new AuthorService();
    public ActionResult Index(int authorId)
    {
        var authors = authorService.GetAuthor(authorId);
        return View(authors);
    }
    //Other action methods
}
```

The AuthorService class takes advantage of the AuthorRepository class to perform CRUD operations.

```
public class AuthorService
{
    private AuthorRepository authorRepository = new AuthorRepository();
    public Author GetAuthor (int authorId)
    {
        return authorRepository.GetAuthor(authorId);
    }
    //Other methods
}
```

Avoid writing boilerplate code to map objects

You often need to map data transfer objects (DTOs) with the domain objects and vice-versa. Refer to the code snippet given below that shows mapping logic inside a controller method.

```
public IActionResult GetAuthor(int authorId)
{
    var author = authorService.GetAuthor(authorId);
    var authorDTO = new AuthorDTO();
    authorDTO.AuthorId = author.AuthorId;
    authorDTO.FirstName = author.FirstName;
    authorDTO.LastName = author.LastName;
    authorDTO.JoiningDate = author.JoiningDate;
    //Other code
   ......
}
```

You shouldn't write such mapping logic inside your controller since it will bloat the controller and add an additional responsibility. If you are to write mapping logic you can take advantage of an object mapper tool like AutoMapper to avoid having to write a lot of boilerplate code.

The following code snippet shows how you can configure AutoMapper.

```
public class AutoMapping
{
    public static void Initialize()
    {
        Mapper.Initialize(cfg =>
        {
            cfg.CreateMap<Author, AuthorDTO>();
            //Other code
        });
    }
}
```

Next, you can call the Initialize() method in the Global.asax as shown below.

```
protected void Application_Start()
{
    AutoMapping.Initialize();
}
```

Finally, you should move the mapping logic inside the service class we created earlier. Note how AutoMapper has been used to map the two incompatible types Author and AuthorDTO.

```
public class AuthorService
{
    private AuthorRepository authorRepository = new AuthorRepository();
    public AuthorDTO GetAuthor (int authorId)
    {
        var author = authorRepository.GetAuthor(authorId);
        return Mapper.Map<AuthorDTO>(author);
    }
    //Other methods
}
```

See my article about AutoMapper for more details.

Avoid writing business logic code in your controllers

You should not write business logic or validation logic in your controllers. The controller should only accept a request and then delegate the next action — nothing else. All business logic code should be moved into some other class (like the AuthorService class we created earlier). There are several ways in which you can set up validators in the request pipeline, but never write the validation logic inside your controller. That makes your controller unnecessarily fat and makes it responsible for tasks that it is not supposed to do.

Prefer dependency injection over composition

You should prefer using dependency injection in your controllers to manage the dependencies. Dependency injection is a subset of the Inversion of Control (IoC) principle. It is used to remove internal dependencies from the implementation by enabling these dependencies to be injected externally. You can read more about the dependency injection principle, using dependency injection in ASP.Net Core, and using dependency injection in ASP.Net Web Forms in my previous articles.

By taking advantage of dependency injection, you need not be concerned about object instantiation, initialization, etc. You can have a factory that returns an instance of the type you need and then you can use this instance by injecting it using constructor injection. The following code snippet illustrates how an instance of type IAuthorService can be injected to the AuthorController using constructor injection. (Assume that IAuthorService is an interface that the AuthorService class extends.)

```
public class AuthorController : Controller
{
    private IAuthorService authorService = new AuthorService();
    public AuthorController(IAuthorService authorService)
    {
       this.authorService = authorService;
    }
   // Action methods
}
```

Use action filters to eliminate duplicate code

You can use action filters in ASP.Net Core to execute custom code at specific points in the request pipeline. As an example, you can use action filters to execute custom code before and after the execution of an action method. Rather than writing validation logic inside your controllers and bloating them unnecessarily, you can remove the validation logic from the controller's action method and write it inside action filters. The following code snippet shows how this can be accomplished.

```
[ValidateModelState]
[HttpPost]
public ActionResult Create(AuthorRequest request)
{
    AuthorService authorService = new AuthorService();
    authorService.Save(request);
    return RedirectToAction("Home");
}
```

If you have several responsibilities assigned to a controller, then there will be several reasons for the controller to change as well. Hence this violates the Single Responsibility Principle, which states that a class should have one and only one reason to change. You can read more about the Single Responsibility Principle in my previous article.

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

- Stay up to date with InfoWorld's newsletters for software developers, analysts, database programmers, and data scientists.
- Get expert insights from our member-only Insider articles.

YOU MAY ALSO LIKE

Tim O'Reilly: the golden age of the programmer is over

How to work with Quartz.Net in C#

Why MongoDB is 'fundamentally better' for developers

What is a computational storage drive? Much-needed help for CPUs

Deno Company forms to back Node.js rival

LLVM 12 arrives with x86, AArch optimizations

3 enterprise AI success stories

Are industry clouds an opportunity or a

Red Hat OpenShift ramps up security and

When to use Task.WaitAll vs. Task.WhenAll in .NET

The best open source software of 2020