# Action Filters in .NET Core Web API

Filters are great features of .NET which provide a way to hook into the MVC action invocation pipeline. Therefore, we can use filters to extract code which can be reused and make our actions cleaner and maintainable. There are some filters that are already provided by .NET like authorization filter, and there are the custom ones, which we create for the reusable code that we want to run before action methods in our controller.

There are different filter types:

* **Authorization filters** – They run first to determine whether a user is authorized for the current request
* **Resource filters** – They run right after the authorization filters and are very useful for caching and performance reasons.
* **Action filters** – They run right before and after the action method execution
* **Exception filters** – They are used to handle exceptions before the response body is populated
* **Result filters** – They run before and after the execution of the action methods result.

In this article, we are going to talk about Action filters and how to use them to create a cleaner and reusable code in our Web API’s.

## Implementation of Action Filters

When we implement action filters, we create a class that inherits either from the IActionFilter interface or IAsyncActionFilter interface or from the ActionFilterAttribute class which is the implementation of the IActionFilter, IAsyncActionFilter, and few different interfaces as well:

public abstract class ActionFilterAttribute : Attribute, IActionFilter, IFilterMetadata, IAsyncActionFilter, IResultFilter, IAsyncResultFilter, IOrderedFilter

In our examples, we are going to inherit from the IActionFIlter interface because it has all the method definitions we require.

If we create a synchronous filter by inheriting from the IActionFilter, we can implement it to run before and after action method execution by implementing OnActionExecuting and OnActionExecuted methods:

namespace ActionFilters.Filters

{

public class ActionFilterExample : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// our code before action executes

}

public void OnActionExecuted(ActionExecutedContext context)

{

// our code after action executes

}

}

}

We can do the same thing with an asynchronous filter by inheriting from the IAsyncActionFilter, but we only have one method to implement the OnActionExecutionAsync:

namespace ActionFilters.Filters

{

public class AsyncActionFilterExample : IAsyncActionFilter

{

public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)

{

// execute any code before the action executes

var result = await next();

// execute any code after the action executes

}

}

}

## Scope of Action Filters

As all the filters, the action filter can be added to different scope levels: Global, Action, Controller.

If we want to use our filter globally, we need to register it inside the AddMVC() method in the ConfigureServices method:

services.AddMvc(

config =>

{

config.Filters.Add(new GlobalFilterExample());

});

But if we want to use our filter as a service type on the Action or Controller level, we need to register it in the same ConfigureServices method but as a service in the IoC container:

services.AddScoped<ActionFilterExample>();

services.AddScoped<ControllerFilterExample>();

Finally, to use a filter registered on the Action or Controller level, we need to place it on top of the Controller or Action as a ServiceType:

namespace AspNetCore.Controllers

{

[ServiceFilter(typeof(ControllerFilterExample))]

[Route("api/[controller]")]

public class TestController : Controller

{

[HttpGet]

[ServiceFilter(typeof(ActionFilterExample))]

public IEnumerable<string> Get()

{

return new string[] { "example", "data" };

}

}

}

## Order of Invocation

The order in which our filters are executed is as follows:

* OnActionExecuting from the Global filter
  + OnActionExecuting from the Controller filter
    - OnActionExecuting from the Action filter
      * *Action method execution*
    - OnActionExecuted from the Action filter
  + OnActionExecuted from the Controller filter
* OnActionExecuted from the Global filter

Of course, we can change the order of invocation by adding additional property Order to the invocation statement:

namespace AspNetCore.Controllers

{

[ServiceFilter(typeof(ControllerFilterExample), Order=2)]

[Route("api/[controller]")]

public class TestController : Controller

{

[HttpGet]

[ServiceFilter(typeof(ActionFilterExample), Order=1)]

public IEnumerable<string> Get()

{

return new string[] { "example", "data" };

}

}

}

Or something like this on top of the same action:

[HttpGet]

[ServiceFilter(typeof(ActionFilterExample), Order=2)]

[ServiceFilter(typeof(ActionFilterExample2), Order=1)]

public IEnumerable<string> Get()

{

return new string[] { "example", "data" };

}

## Writing Better Code with ActionFilters

If we open the starting project from the AppStart folder from our repository, we are going to find the MoveController class in the Controllers folder. This controller has an implementation for all the CRUD operations. For the simplicity sake, we haven’t used any additional layers for our API. This project also implements global error handling so if you are not familiar with that topic, we suggest you to read Global Exception Handling in .NET Core Web API.

Our actions are quite clean and readable without try-catch blocks due to global exception handling, but we can write them even better and cleaner as well.

The important thing to notice is that our Movie model inherits from the IEntity interface:

[Table("Movie")]

public class Movie: IEntity

{

[Key]

public Guid Id { get; set; }

[Required(ErrorMessage = "Name is required")]

public string Name { get; set; }

[Required(ErrorMessage = "Genre is required")]

public string Genre { get; set; }

[Required(ErrorMessage = "Director is required")]

public string Director { get; set; }

}

So let’s start with the validation code from the POST and PUT actions.

## Validation with ActionFilters

If we look in our POST and PUT actions, we will notice that we repeat the code in which we validate our Movie model:

if (movie == null)

{

return BadRequest("Movie object is null");

}

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

We can extract that code into a custom ActionFilter class, thus making this code reusable and action cleaner.

So let’s do that.

Let’s create a new folder in our solution explorer, and name it ActionFilters. Then inside that folder, we are going to create a new class ValidationFilterAttribute:

using Microsoft.AspNetCore.Mvc.Filters;

namespace ActionFilters.ActionFilters

{

public class ValidationFilterAttribute : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

}

public void OnActionExecuted(ActionExecutedContext context)

{

}

}

}

Now we are going to modify the OnActionExecuting method to validate our model:

using ActionFilters.Contracts;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

using System.Linq;

namespace ActionFilters.ActionFilters

{

public class ValidationFilterAttribute : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

var param = context.ActionArguments.SingleOrDefault(p => p.Value is IEntity);

if(param.Value == null)

{

context.Result = new BadRequestObjectResult("Object is null");

return;

}

if(!context.ModelState.IsValid)

{

context.Result = new BadRequestObjectResult(context.ModelState);

}

}

public void OnActionExecuted(ActionExecutedContext context)

{

}

}

}

Now let’s register this action filter in the ConfigureServices method:

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<MovieContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("sqlConString")));

services.AddScoped<ValidationFilterAttribute>();

services.AddMvc();

}

Finally, let’s remove that validation code from our actions and call this action filter as a service:

[HttpPost]

[ServiceFilter(typeof(ValidationFilterAttribute))]

public IActionResult Post([FromBody] Movie movie)

{

\_context.Movies.Add(movie);

\_context.SaveChanges();

return CreatedAtRoute("MovieById", new { id = movie.Id }, movie);

}

[HttpPut("{id}")]

[ServiceFilter(typeof(ValidationFilterAttribute))]

public IActionResult Put(Guid id, [FromBody]Movie movie)

{

var dbMovie = \_context.Movies.SingleOrDefault(x => x.Id.Equals(id));

if (dbMovie == null)

{

return NotFound();

}

dbMovie.Map(movie);

\_context.Movies.Update(dbMovie);

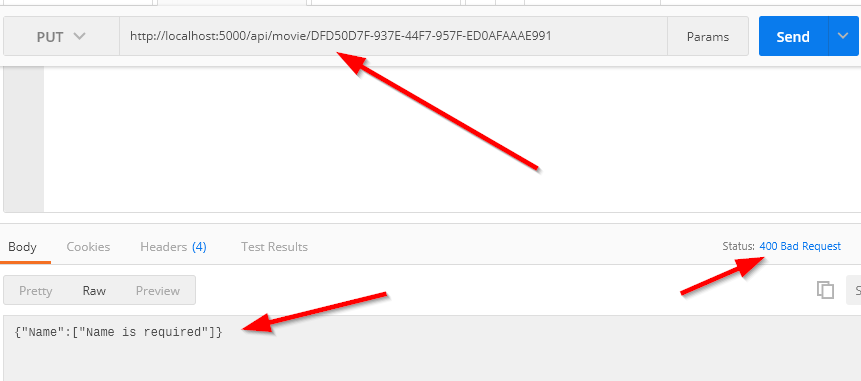
\_context.SaveChanges();

return NoContent();

}

Excellent. This code is much better and readable now without our validation part. And furthermore, the validation part is now reusable as long as our model classes inherit from the IEntity interface, which is quite common behavior.

If we send a POST request for example with the invalid model we will get an error response:



## Dependency Injection in Action Filters

If we take a look at our GetById, POST and PUT actions, we are going to see the code where we fetch the move by id from the database and check if it exists:

var dbMovie = \_context.Movies.SingleOrDefault(x => x.Id.Equals(id));

if (dbMovie == null)

{

return NotFound();

}

We can extract it to the ActionFilter class as well, thus making it reusable for all the actions.

Of course, we need to inject our context in a new ActionFilter class by using dependency injection.

So, let’s create another ActionFilter class ValidateEntityExistsAttribute in the ActionFilters folder and modify it:

using System.Linq;

namespace ActionFilters.ActionFilters

{

public class ValidateEntityExistsAttribute<T> : IActionFilter where T: class, IEntity

{

private readonly MovieContext \_context;

public ValidateEntityExistsAttribute(MovieContext context)

{

\_context = context;

}

public void OnActionExecuting(ActionExecutingContext context)

{

Guid id = Guid.Empty;

if (context.ActionArguments.ContainsKey("id"))

{

id = (Guid)context.ActionArguments["id"];

}

else

{

context.Result = new BadRequestObjectResult("Bad id parameter");

return;

}

var entity = \_context.Set<T>().SingleOrDefault(x => x.Id.Equals(id));

if(entity == null)

{

context.Result = new NotFoundResult();

}

else

{

context.HttpContext.Items.Add("entity", entity);

}

}

public void OnActionExecuted(ActionExecutedContext context)

{

}

}

}

We create this ActionFilter to be generic, so we could reuse it for any model in our project. Furthermore, if we find the entity in the database, we store it in HttpContext because we need that entity in our action methods and we don’t want to double our fetch action towards the database (we would lose more than we gain if we double that action)

Now let’s register it:

services.AddScoped<ValidateEntityExistsAttribute<Movie>>();

And let’s modify our actions:

[HttpGet("{id}", Name = "MovieById")]

[ServiceFilter(typeof(ValidateEntityExistsAttribute<Movie>))]

public IActionResult Get(Guid id)

{

var dbMovie = HttpContext.Items["entity"] as Movie;

return Ok(dbMovie);

}

[HttpPut("{id}")]

[ServiceFilter(typeof(ValidationFilterAttribute))]

[ServiceFilter(typeof(ValidateEntityExistsAttribute<Movie>))]

public IActionResult Put(Guid id, [FromBody]Movie movie)

{

var dbMovie = HttpContext.Items["entity"] as Movie;

dbMovie.Map(movie);

\_context.Movies.Update(dbMovie);

\_context.SaveChanges();

return NoContent();

}

[HttpDelete("{id}")]

[ServiceFilter(typeof(ValidateEntityExistsAttribute<Movie>))]

public IActionResult Delete(Guid id)

{

var dbMovie = HttpContext.Items["entity"] as Movie;

\_context.Movies.Remove(dbMovie);

\_context.SaveChanges();

return NoContent();

}

Awesome.

Now our actions look great without code repetition, try-catch blocks or additional fetch request towards the database.