## Filtri in ASP.NET Core

I *filtri* in ASP.NET Core consentono l'esecuzione del codice prima o dopo fasi specifiche della pipeline di elaborazione della richiesta.

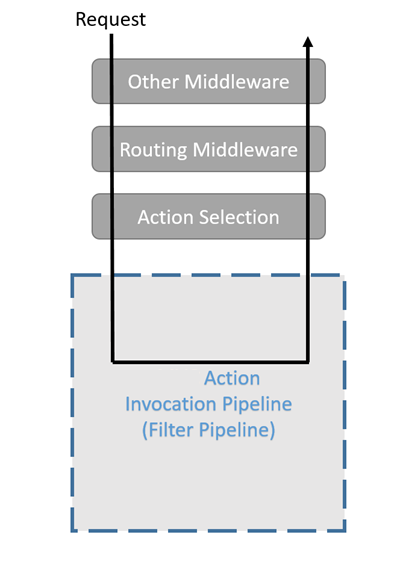
I filtri predefiniti gestiscono attività, ad esempio:

* Autorizzazione (impedire l'accesso alle risorse per cui un utente non è autorizzato).
* Memorizzazione nella cache delle risposte (blocco della pipeline delle richieste per restituire una risposta memorizzata nella cache).

I filtri personalizzati possono essere creati per gestire problemi relativi a più settori. Esempi di problematiche trasversali includono la gestione degli errori, la memorizzazione nella cache, la configurazione, l'autorizzazione e la registrazione. I filtri evitano la duplicazione del codice. Ad esempio, un filtro eccezioni per la gestione degli errori potrebbe consolidare la gestione degli errori.

### Funzionamento dei filtri

I filtri vengono eseguiti all'interno della pipeline di chiamata di azioni ASP.NET Core, talvolta definita pipeline di filtro. La pipeline di filtro viene eseguita dopo che ASP.NET Core ha selezionato l'azione da eseguire.

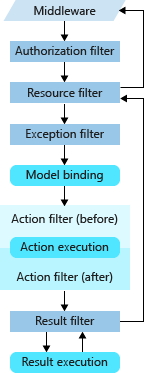


### Tipi di filtro

Ogni tipo di filtro viene eseguito in una fase diversa della pipeline di filtro:

* I [filtri di autorizzazione](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#authorization-filters) vengono eseguiti per primi e consentono di determinare se l'utente è autorizzato per la richiesta. I filtri di autorizzazione causano il corto circuito della pipeline se la richiesta non è autorizzata.
* [Filtri di risorse](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#resource-filters):
  + Vengono eseguiti dopo l'autorizzazione.
  + [OnResourceExecuting](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresourcefilter.onresourceexecuting) può eseguire codice prima del resto della pipeline di filtro. Ad esempio, OnResourceExecuting può eseguire codice prima dell'associazione di modelli.
  + [OnResourceExecuted](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresourcefilter.onresourceexecuted) può eseguire codice dopo che il resto della pipeline è stato completato.
* I [filtri azione](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#action-filters) possono eseguire codice immediatamente prima e dopo la chiamata di un metodo di azione singolo. Possono essere usati per modificare gli argomenti passati a un'azione e il risultato restituito dall'azione. I filtri di azione **non** sono supportati in Razor Pages.
* I [filtri eccezioni](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#exception-filters) vengono usati per applicare i criteri globali a eccezioni non gestite che si verificano prima che qualsiasi elemento sia stato scritto nel corpo della risposta.
* I [filtri risultato](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#result-filters) possono eseguire codice immediatamente prima e dopo l'esecuzione di risultati di azioni singole. Vengono eseguiti solo quando il metodo di azione è stato eseguito correttamente. Sono utili per la logica che deve racchiudere l'esecuzione del formattatore o di una vista.

Il diagramma seguente illustra come interagiscono i tipi di filtro nella pipeline di filtro.



## Implementazione

I filtri supportano sia implementazioni sincrone che asincrone tramite definizioni di interfaccia diverse.

I filtri sincroni possono eseguire codice prima (On-Stage-Executing) e dopo (On-Stage-Executed) la relativa fase della pipeline. La chiamata di OnActionExecuting, ad esempio, avviene prima della chiamata del metodo di azione. La chiamata di OnActionExecuted avviene dopo l'esecuzione del metodo di azione.

C#Copia

public class MySampleActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// Do something before the action executes.

}

public void OnActionExecuted(ActionExecutedContext context)

{

// Do something after the action executes.

}

}

I filtri asincroni definiscono un metodo On-Stage-ExecutionAsync:

C#Copia

public class SampleAsyncActionFilter : IAsyncActionFilter

{

public async Task OnActionExecutionAsync(

ActionExecutingContext context,

ActionExecutionDelegate next)

{

// Do something before the action executes.

// next() calls the action method.

var resultContext = await next();

// resultContext.Result is set.

// Do something after the action executes.

}

}

Nel codice precedente SampleAsyncActionFilter ha un oggetto [ActionExecutionDelegate](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutiondelegate) (next) che esegue il metodo di azione. Ognuno dei metodi On-Stage-ExecutionAsync accetta un oggetto FilterType-ExecutionDelegate che esegue la fase della pipeline del filtro.

### Fasi di filtro multiple

È possibile implementare interfacce per più fasi di filtro in una singola classe. Ad esempio, la classe [ActionFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionfilterattribute) implementa IActionFilter, IResultFilter e i relativi equivalenti asincroni.

Implementare la versione sincrona **oppure** la versione asincrona di un'interfaccia di filtro, **non** entrambe. Il runtime controlla per prima cosa se il filtro implementa l'interfaccia asincrona e, in tal caso, la chiama. In caso contrario, chiama i metodi dell'interfaccia sincrona. Se in una classe sono implementate entrambe le interfacce, sincrona e asincrona, viene chiamato solo il metodo asincrono. Quando si usano classi astratte come [ActionFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionfilterattribute) eseguire l'override solo dei metodi sincroni o del metodo asincrono per ogni tipo di filtro.

### Attributi filtro predefiniti

ASP.NET Core include filtri predefiniti basati su attributi che è possibile impostare come sottoclasse e personalizzare. Il filtro dei risultati seguente, ad esempio, aggiunge un'intestazione alla risposta:

C#Copia

public class AddHeaderAttribute : ResultFilterAttribute

{

private readonly string \_name;

private readonly string \_value;

public AddHeaderAttribute(string name, string value)

{

\_name = name;

\_value = value;

}

public override void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add( \_name, new string[] { \_value });

base.OnResultExecuting(context);

}

}

Gli attributi consentono ai filtri di accettare argomenti, come illustrato nell'esempio precedente. Applicare AddHeaderAttribute a un controller o a un metodo di azione e specificare il nome e il valore dell'intestazione HTTP:

C#Copia

[AddHeader("Author", "Joe Smith")]

public class SampleController : Controller

{

public IActionResult Index()

{

return Content("Examine the headers using the F12 developer tools.");

}

[ShortCircuitingResourceFilter]

public IActionResult SomeResource()

{

return Content("Successful access to resource - header is set.");

}

Molte delle interfacce del filtro presentano attributi corrispondenti che possono essere usati come classi di base per implementazioni personalizzate.

Attributi dei filtri:

* [ActionFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionfilterattribute)
* [ExceptionFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.exceptionfilterattribute)
* [ResultFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resultfilterattribute)
* [FormatFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.formatfilterattribute)
* [ServiceFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute)
* [TypeFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.typefilterattribute)

## Ambiti dei filtri e ordine di esecuzione

È possibile aggiungere un filtro alla pipeline in uno dei tre ambiti:

* Usando un attributo di un'azione.
* Usando un attributo di un controller.
* A livello globale per tutti i controller e le azioni come illustrato nel codice seguente:

C#Copia

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc(options =>

{

options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",

"Result filter added to MvcOptions.Filters")); // An instance

options.Filters.Add(typeof(MySampleActionFilter)); // By type

options.Filters.Add(new SampleGlobalActionFilter()); // An instance

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

}

Il codice precedente aggiunge tre filtri a livello globale tramite la raccolta [MvcOptions.Filters](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.filters#Microsoft_AspNetCore_Mvc_MvcOptions_Filters).

### Ordine di esecuzione predefinito

Quando sono presenti più filtri per una particolare fase della pipeline, l'ambito determina l'ordine di esecuzione predefinito dei filtri. I filtri globali racchiudono i filtri di classe, che a loro volta racchiudono i filtri dei metodi.

Come risultato dell'annidamento dei filtri, il codice after dei filtri viene eseguito in ordine inverso rispetto al codice before. Sequenza di filtro:

* Codice before dei filtri globali.
  + Codice before dei filtri del controller.
    - Codice before dei filtri del metodo di azione.
    - Codice after dei filtri del metodo di azione.
  + Codice after dei filtri del controller.
* Codice after dei filtri globali.

L'esempio seguente illustra l'ordine in cui i metodi dei filtri vengono chiamati per i filtri di azione sincroni.

| **Sequence** | **Ambito del filtro** | **Metodo del filtro** |
| --- | --- | --- |
| 1 | Global | OnActionExecuting |
| 2 | Controller | OnActionExecuting |
| 3 | Metodo | OnActionExecuting |
| 4 | Metodo | OnActionExecuted |
| 5 | Controller | OnActionExecuted |
| 6 | Global | OnActionExecuted |

Questa sequenza mostra che:

* Il filtro del metodo è annidato all'interno del filtro del controller.
* Il filtro del controller è annidato all'interno del filtro globale.

### Filtri a livello di controller e pagina Razor

Ogni controller che eredita dalla classe di base [Controller](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller) include i metodi [Controller.OnActionExecuting](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecuting), [Controller.OnActionExecutionAsync](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecutionasync) e [Controller.OnActionExecuted](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller.onactionexecuted) OnActionExecuted. Questi metodi:

* Eseguono il wrapping dei filtri che vengono eseguiti per una determinata azione.
* La chiamata di OnActionExecuting avviene prima di qualsiasi filtro dell'azione.
* La chiamata di OnActionExecuted avviene dopo tutti i filtri dell'azione.
* La chiamata di OnActionExecutionAsync avviene prima di qualsiasi filtro dell'azione. Il codice del filtro dopo next viene eseguito dopo il metodo di azione.

Ad esempio, l'applicazione di MySampleActionFilter nell'esempio scaricato avviene a livello globale all'avvio.

TestController:

* Applica SampleActionFilterAttribute ([SampleActionFilter]) all'azione FilterTest2:
* Esegue l'override di OnActionExecuting e OnActionExecuted.

C#Copia

public class TestController : Controller

{

[SampleActionFilter]

public IActionResult FilterTest2()

{

return Content($"From FilterTest2");

}

public override void OnActionExecuting(ActionExecutingContext context)

{

// Do something before the action executes.

base.OnActionExecuting(context);

}

public override void OnActionExecuted(ActionExecutedContext context)

{

// Do something after the action executes.

base.OnActionExecuted(context);

}

}

Passando a https://localhost:5001/Test/FilterTest2, viene eseguito il codice seguente:

* TestController.OnActionExecuting
  + MySampleActionFilter.OnActionExecuting
    - SampleActionFilterAttribute.OnActionExecuting
      * TestController.FilterTest2
    - SampleActionFilterAttribute.OnActionExecuted
  + MySampleActionFilter.OnActionExecuted
* TestController.OnActionExecuted

Per Razor Pages, vedere [Implementare i filtri di Razor Pages eseguendo l'override dei metodi di filtro](https://docs.microsoft.com/it-it/aspnet/core/razor-pages/filter?view=aspnetcore-3.0#implement-razor-page-filters-by-overriding-filter-methods).

### Override dell'ordine predefinito

È possibile eseguire l'override della sequenza di esecuzione predefinita implementando [IOrderedFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iorderedfilter). IOrderedFilter espone la proprietà [Order](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iorderedfilter.order#Microsoft_AspNetCore_Mvc_Filters_IOrderedFilter_Order), che ha la precedenza sull'ambito per determinare l'ordine di esecuzione. Un filtro con un valore di Order inferiore:

* Esegue il codice before prima di quello di un filtro con un valore di Order più alto.
* Esegue il codice after dopo quello di un filtro con un valore di Order più alto.

La proprietà Order può essere impostata con un parametro del costruttore:

C#Copia

[MyFilter(Name = "Controller Level Attribute", Order=1)]

Prendere in considerazione gli stessi tre filtri di azione illustrati nell'esempio precedente. Se la proprietà Order del controller e dei filtri globali è impostata, rispettivamente, su 1 e su 2, l'ordine di esecuzione viene invertito.

| **Sequence** | **Ambito del filtro** | **ProprietàOrder** | **Metodo del filtro** |
| --- | --- | --- | --- |
| 1 | Metodo | 0 | OnActionExecuting |
| 2 | Controller | 1 | OnActionExecuting |
| 3 | Global | 2 | OnActionExecuting |
| 4 | Global | 2 | OnActionExecuted |
| 5 | Controller | 1 | OnActionExecuted |
| 6 | Metodo | 0 | OnActionExecuted |

La proprietà Order ha la precedenza sull'ambito nel determinare l'ordine di esecuzione dei filtri. I filtri vengono ordinati prima in base all'ordine, poi viene usato l'ambito per interrompere i collegamenti. Tutti i filtri predefiniti implementano IOrderedFilter e impostano il valore predefinito di Order su 0. Per i filtri predefiniti, l'ambito determina l'ordine, a meno che non si imposti Order su un valore diverso da zero.

## Annullamento e corto circuito

È possibile causare il corto circuito della pipeline di filtro impostando la proprietà [Result](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resourceexecutingcontext.result#Microsoft_AspNetCore_Mvc_Filters_ResourceExecutingContext_Result) sul parametro [ResourceExecutingContext](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resourceexecutingcontext) fornito al metodo di filtro. Ad esempio, il filtro di risorse seguente impedisce l'esecuzione del resto della pipeline:

C#Copia

public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter

{

public void OnResourceExecuting(ResourceExecutingContext context)

{

context.Result = new ContentResult()

{

Content = "Resource unavailable - header not set."

};

}

public void OnResourceExecuted(ResourceExecutedContext context)

{

}

}

Nel codice seguente sia il filtro ShortCircuitingResourceFilter che il filtro AddHeader hanno come destinazione il metodo di azione SomeResource. ShortCircuitingResourceFilter:

* Viene eseguito per primo perché è un filtro risorsa e AddHeader è un filtro azione.
* Blocca il resto della pipeline.

Pertanto il filtro AddHeader non viene mai eseguito per l'azione SomeResource. Questo comportamento sarebbe lo stesso se entrambi i filtri venissero applicati a livello di metodo di azione, a condizione che il filtro ShortCircuitingResourceFilter venga eseguito prima. ShortCircuitingResourceFilter viene eseguito per primo per il tipo di filtro o per l'uso esplicito della proprietà Order.

C#Copia

[AddHeader("Author", "Joe Smith")]

public class SampleController : Controller

{

public IActionResult Index()

{

return Content("Examine the headers using the F12 developer tools.");

}

[ShortCircuitingResourceFilter]

public IActionResult SomeResource()

{

return Content("Successful access to resource - header is set.");

}

## Inserimento di dipendenze

È possibile aggiungere filtri per tipo o per istanza. Se viene aggiunta un'istanza, tale istanza viene usata per ogni richiesta. Se viene aggiunto un tipo, il filtro è attivato dal tipo. Un filtro attivato dal tipo comporta:

* La creazione di un'istanza per ogni richiesta.
* Il popolamento di tutte le dipendenze del costruttore tramite [inserimento delle dipendenze](https://docs.microsoft.com/it-it/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.0).

I filtri implementati come attributi e aggiunti direttamente alle classi controller o ai metodi di azione non possono avere dipendenze costruttore specificate dall'[inserimento di dipendenze](https://docs.microsoft.com/it-it/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.0). Le dipendenze del costruttore non possono essere fornite tramite inserimento delle dipendenze in quanto:

* I parametri del costruttore devono essere forniti agli attributi nella posizione in cui vengono applicati.
* Questa è una limitazione del funzionamento degli attributi.

I filtri seguenti supportano le dipendenze del costruttore fornite dall'inserimento delle dipendenze:

* [ServiceFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute)
* [TypeFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.typefilterattribute)
* [IFilterFactory](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory) implementato nell'attributo.

I filtri precedenti possono essere applicati a un controller o a un metodo di azione:

I logger sono disponibili tramite l'inserimento delle dipendenze. Evitare tuttavia di creare e usare filtri esclusivamente per scopi di registrazione. La funzionalità di [registrazione del framework predefinita](https://docs.microsoft.com/it-it/aspnet/core/fundamentals/logging/index?view=aspnetcore-3.0) in genere fornisce il necessario per la registrazione. La registrazione aggiunta ai filtri:

* Deve concentrarsi su problemi del dominio di business o comportamenti specifici del filtro.
* **Non** deve registrare azioni o altri eventi del framework. I filtri predefiniti registrano azioni ed eventi del framework.

### ServiceFilterAttribute

I tipi di implementazione del filtro di servizi sono registrati in ConfigureServices. [ServiceFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute) recupera un'istanza del filtro dall'inserimento di dipendenze.

Il codice seguente illustra AddHeaderResultServiceFilter:

C#Copia

public class AddHeaderResultServiceFilter : IResultFilter

{

private ILogger \_logger;

public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();

}

public void OnResultExecuting(ResultExecutingContext context)

{

var headerName = "OnResultExecuting";

context.HttpContext.Response.Headers.Add(

headerName, new string[] { "ResultExecutingSuccessfully" });

\_logger.LogInformation("Header added: {HeaderName}", headerName);

}

public void OnResultExecuted(ResultExecutedContext context)

{

// Can't add to headers here because response has started.

}

}

Nel codice seguente l'oggetto AddHeaderResultServiceFilter viene aggiunto al contenitore di inserimento delle dipendenze:

C#Copia

public void ConfigureServices(IServiceCollection services)

{

// Add service filters.

services.AddScoped<AddHeaderResultServiceFilter>();

services.AddScoped<SampleActionFilterAttribute>();

services.AddMvc(options =>

{

options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",

"Result filter added to MvcOptions.Filters")); // An instance

options.Filters.Add(typeof(MySampleActionFilter)); // By type

options.Filters.Add(new SampleGlobalActionFilter()); // An instance

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

}

Nel codice seguente l'attributo ServiceFilter recupera un'istanza del filtro AddHeaderResultServiceFilter dall'inserimento delle dipendenze:

C#Copia

[ServiceFilter(typeof(AddHeaderResultServiceFilter))]

public IActionResult Index()

{

return View();

}

Quando si usa l'impostazione ServiceFilterAttribute, [ServiceFilterAttribute.IsReusable](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute.isreusable#Microsoft_AspNetCore_Mvc_ServiceFilterAttribute_IsReusable):

* Indica che l'istanza del filtro può essere riutilizzata all'esterno dell'ambito della richiesta in cui è stata creata. Il runtime di ASP.NET Core non garantisce:
  + Che venga creata una singola istanza del filtro.
  + Che il filtro non verrà richiesto di nuovo dal contenitore di inserimento delle dipendenze in un momento successivo.
* Non si deve usare con un filtro che dipende da servizi con una durata diversa da singleton.

[ServiceFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute) implementa [IFilterFactory](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory). IFilterFactory espone il metodo [CreateInstance](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory.createinstance) per creare un'istanza [IFilterMetadata](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifiltermetadata). CreateInstance carica il tipo specificato dall'inserimento delle dipendenze.

### TypeFilterAttribute

[TypeFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.typefilterattribute) è simile a [ServiceFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.servicefilterattribute), ma il relativo tipo non viene risolto direttamente dal contenitore dell'inserimento di dipendenze. Viene creata un'istanza del tipo tramite [Microsoft.Extensions.DependencyInjection.ObjectFactory](https://docs.microsoft.com/dotnet/api/microsoft.extensions.dependencyinjection.objectfactory).

Poiché i tipi TypeFilterAttribute non vengono risolti direttamente dal contenitore di inserimento delle dipendenze:

* Non è necessario che i tipi a cui viene fatto riferimento tramite TypeFilterAttribute siano registrati nel contenitore di inserimento delle dipendenze. Le loro dipendenze vengono soddisfatte dal contenitore di inserimento delle dipendenze.
* In via facoltativa, TypeFilterAttribute può anche accettare gli argomenti del costruttore per il tipo.

Quando si usa l'impostazione TypeFilterAttribute, [TypeFilterAttribute.IsReusable](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.typefilterattribute.isreusable#Microsoft_AspNetCore_Mvc_TypeFilterAttribute_IsReusable):

* Indica che l'istanza del filtro può essere riutilizzata all'esterno dell'ambito della richiesta in cui è stata creata. Il runtime di ASP.NET Core non fornisce alcuna garanzia che venga creata una singola istanza del filtro.
* Non si deve usare con un filtro che dipende da servizi con una durata diversa da singleton.

L'esempio seguente illustra come passare argomenti a un tipo usando TypeFilterAttribute:

C#Copia

[TypeFilter(typeof(LogConstantFilter),

Arguments = new object[] { "Method 'Hi' called" })]

public IActionResult Hi(string name)

{

return Content($"Hi {name}");

}

C#Copia

public class LogConstantFilter : IActionFilter

{

private readonly string \_value;

private readonly ILogger<LogConstantFilter> \_logger;

public LogConstantFilter(string value, ILogger<LogConstantFilter> logger)

{

\_logger = logger;

\_value = value;

}

public void OnActionExecuting(ActionExecutingContext context)

{

\_logger.LogInformation(\_value);

}

public void OnActionExecuted(ActionExecutedContext context)

{ }

}

## Filtri autorizzazione

I filtri di autorizzazione:

* Sono i primi filtri a essere eseguiti nella pipeline di filtro.
* Controllano l'accesso ai metodi di azione.
* Dispongono di un metodo precedente, ma non di un metodo successivo.

I filtri di autorizzazione personalizzati richiedono un framework di autorizzazione personalizzato. È preferibile configurare criteri di autorizzazione o scrivere criteri di autorizzazione personalizzati piuttosto che scrivere un filtro personalizzato. Il filtro di autorizzazione predefinito:

* Chiama il sistema di autorizzazione.
* Non autorizza le richieste.

**Non** generare eccezioni nei filtri di autorizzazione:

* L'eccezione non verrà gestita.
* I filtri di eccezione non gestiranno l'eccezione.

È consigliabile emettere una richiesta di verifica in caso di eccezione in un filtro di autorizzazione.

Altre informazioni sull'[autorizzazione](https://docs.microsoft.com/it-it/aspnet/core/security/authorization/introduction?view=aspnetcore-3.0).

## Filtri risorse

I filtri di risorse:

* Implementano l'interfaccia [IResourceFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresourcefilter) o [IAsyncResourceFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncresourcefilter).
* L'esecuzione racchiude la maggior parte della pipeline di filtro.
* Solo i [filtri di autorizzazione](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#authorization-filters) vengono eseguiti prima dei filtri di risorse.

I filtri di risorse sono utili per causare il corto circuito della maggior parte della pipeline. Un filtro di memorizzazione nella cache può, ad esempio, evitare il resto della pipeline in caso di riscontro nella cache.

Esempi di filtri di risorse:

* Il [filtro di risorse di corto circuito](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#short-circuiting-resource-filter) illustrato in precedenza.
* [DisableFormValueModelBindingAttribute](https://github.com/aspnet/Entropy/blob/rel/2.0.0-preview2/samples/Mvc.FileUpload/Filters/DisableFormValueModelBindingAttribute.cs):
  + Impedisce all'associazione di modelli di accedere ai dati del modulo.
  + Viene usato quando si caricano file di grandi dimensioni per impedire la lettura dei dati del modulo in memoria.

## Filtri azione

**Importante**

I filtri azione **non** si applicano a Razor Pages. Razor Pages supporta [**IPageFilter**](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ipagefilter) e [**IAsyncPageFilter**](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncpagefilter). Per altre informazioni, vedere [**Modalità di filtro per pagine Razor**](https://docs.microsoft.com/it-it/aspnet/core/razor-pages/filter?view=aspnetcore-3.0).

I filtri di azione:

* Implementano l'interfaccia [IActionFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iactionfilter) o [IAsyncActionFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncactionfilter).
* La loro esecuzione circonda l'esecuzione dei metodi di azione.

Il codice seguente mostra un filtro di azione di esempio:

C#Copia

public class MySampleActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// Do something before the action executes.

}

public void OnActionExecuted(ActionExecutedContext context)

{

// Do something after the action executes.

}

}

La classe [ActionExecutingContext](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutingcontext) specifica le proprietà seguenti:

* [ActionArguments](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutingcontext.actionarguments#Microsoft_AspNetCore_Mvc_Filters_ActionExecutingContext_ActionArguments): consente di leggere gli input per un metodo di azione.
* [Controller](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller): consente di modificare l'istanza del controller.
* [Result](https://docs.microsoft.com/dotnet/api/system.web.mvc.actionexecutingcontext.result#System_Web_Mvc_ActionExecutingContext_Result): l'impostazione di Result causa il corto circuito del metodo di azione e dei filtri di azione successivi.

La generazione di un'eccezione in un metodo di azione:

* Impedisce l'esecuzione dei filtri successivi.
* A differenza dell'impostazione di Result, è considerata un errore anziché un risultato positivo.

La classe [ActionExecutedContext](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutedcontext) specifica Controller e Result oltre alle proprietà seguenti:

* [Canceled](https://docs.microsoft.com/dotnet/api/system.web.mvc.actionexecutedcontext.canceled#System_Web_Mvc_ActionExecutedContext_Canceled): true se un altro file ha causato il corto circuito dell'azione.
* [Exception](https://docs.microsoft.com/dotnet/api/system.web.mvc.actionexecutedcontext.exception#System_Web_Mvc_ActionExecutedContext_Exception): non Null se l'azione o un filtro di azione eseguito in precedenza ha generato un'eccezione. Se si imposta questa proprietà su Null:
  + L'eccezione viene gestita in modo efficace.
  + L'oggetto Result viene eseguito come se fosse stato restituito dal metodo di azione.

Per un oggetto IAsyncActionFilter, una chiamata a [ActionExecutionDelegate](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutiondelegate):

* Esegue qualsiasi filtro azione successivo e il metodo di azione.
* Restituisce ActionExecutedContext.

Per causare il corto circuito, assegnare [Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutingcontext.result#Microsoft_AspNetCore_Mvc_Filters_ActionExecutingContext_Result) a un'istanza del risultato e non chiamare next (ActionExecutionDelegate).

Il framework fornisce un oggetto [ActionFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionfilterattribute) astratto che è possibile impostare come sottoclasse.

Il filtro di azione OnActionExecuting può essere usato per:

* Convalidare lo stato del modello.
* Restituire un errore se lo stato non è valido.

C#Copia

public class ValidateModelAttribute : ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext context)

{

if (!context.ModelState.IsValid)

{

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

}

}

Il metodo OnActionExecuted viene eseguito dopo il metodo di azione:

* È possibile visualizzare e modificare i risultati dell'azione tramite la proprietà [Result](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutedcontext.result#Microsoft_AspNetCore_Mvc_Filters_ActionExecutedContext_Result).
* L'impostazione di [Canceled](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutedcontext.canceled#Microsoft_AspNetCore_Mvc_Filters_ActionExecutedContext_Canceled) è true se un altro filtro ha causato il corto circuito dell'esecuzione dell'azione.
* L'impostazione di [Exception](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.actionexecutedcontext.exception#Microsoft_AspNetCore_Mvc_Filters_ActionExecutedContext_Exception) è un valore non Null se l'azione o un filtro di azione successivo ha generato un'eccezione. Impostazione di Exception su null:
  + Gestisce un'eccezione in modo efficace.
  + L'oggetto ActionExecutedContext.Result viene eseguito come se fosse stato restituito normalmente dal metodo di azione.

C#Copia

public class ValidateModelAttribute : ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext context)

{

if (!context.ModelState.IsValid)

{

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

}

}

public override void OnActionExecuted(ActionExecutedContext context)

{

var result = context.Result;

// Do something with Result.

if (context.Canceled == true)

{

// Action execution was short-circuited by another filter.

}

if(context.Exception != null)

{

// Exception thrown by action or action filter.

// Set to null to handle the exception.

context.Exception = null;

}

base.OnActionExecuted(context);

}

}

## Filtri eccezioni

Filtri eccezioni:

* Implementano [IExceptionFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iexceptionfilter) o [IAsyncExceptionFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncexceptionfilter).
* Possono essere usati per implementare criteri di gestione degli errori comuni.

Il filtro di eccezione di esempio seguente usa una visualizzazione degli errori personalizzata per mostrare i dettagli sulle eccezioni che si verificano quando l'app è in fase di sviluppo:

C#Copia

public class CustomExceptionFilterAttribute : ExceptionFilterAttribute

{

private readonly IHostingEnvironment \_hostingEnvironment;

private readonly IModelMetadataProvider \_modelMetadataProvider;

public CustomExceptionFilterAttribute(

IHostingEnvironment hostingEnvironment,

IModelMetadataProvider modelMetadataProvider)

{

\_hostingEnvironment = hostingEnvironment;

\_modelMetadataProvider = modelMetadataProvider;

}

public override void OnException(ExceptionContext context)

{

if (!\_hostingEnvironment.IsDevelopment())

{

return;

}

var result = new ViewResult {ViewName = "CustomError"};

result.ViewData = new ViewDataDictionary(\_modelMetadataProvider,

context.ModelState);

result.ViewData.Add("Exception", context.Exception);

// TODO: Pass additional detailed data via ViewData

context.Result = result;

}

}

Filtri eccezioni:

* Non dispone di eventi precedenti o successivi.
* Implementano [OnException](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iexceptionfilter.onexception) o [OnExceptionAsync](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncexceptionfilter.onexceptionasync).
* Gestiscono le eccezioni non gestite che si verificano nella creazione di un controller o una pagina Razor, nell'[associazione di modelli](https://docs.microsoft.com/it-it/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.0), nei filtri di azione o nei metodi di azione.
* **Non** rilevano le eccezioni che si verificano nei filtri di risorse, nei filtri dei risultati o nell'esecuzione dei risultati MVC.

Per gestire un'eccezione, impostare la proprietà [ExceptionHandled](https://docs.microsoft.com/dotnet/api/system.web.mvc.exceptioncontext.exceptionhandled#System_Web_Mvc_ExceptionContext_ExceptionHandled) su true o scrivere una risposta. In questo modo si arresta la propagazione dell'eccezione. Un filtro di eccezione non può modificare un'eccezione in una "operazione riuscita". Questa operazione può essere eseguita solo tramite un filtro di azione.

Filtri eccezioni:

* Sono ideali per intercettare le eccezioni che si verificano nelle azioni.
* Non sono flessibili quanto il middleware di gestione degli errori.

Scegliere il middleware per la gestione delle eccezioni. Usare i filtri di eccezione solo quando la gestione degli errori varia in base al metodo di azione chiamato. Un'app può ad esempio avere metodi di azione sia per gli endpoint API che per le visualizzazioni/HTML. Gli endpoint dell'API possono restituire informazioni sull'errore come JSON, mentre le azioni basate sulla visualizzazione possono restituire una pagina di errore in formato HTML.

## Filtri risultato

I filtri dei risultati:

* Implementare un'interfaccia:
  + [IResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresultfilter) o [IAsyncResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncresultfilter)
  + [IAlwaysRunResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ialwaysrunresultfilter) o [IAsyncAlwaysRunResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncalwaysrunresultfilter)
* La loro esecuzione circonda l'esecuzione dei risultati dell'azione.

### IResultFilter e IAsyncResultFilter

Il codice seguente illustra un filtro dei risultati che aggiunge un'intestazione HTTP:

C#Copia

public class AddHeaderResultServiceFilter : IResultFilter

{

private ILogger \_logger;

public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();

}

public void OnResultExecuting(ResultExecutingContext context)

{

var headerName = "OnResultExecuting";

context.HttpContext.Response.Headers.Add(

headerName, new string[] { "ResultExecutingSuccessfully" });

\_logger.LogInformation("Header added: {HeaderName}", headerName);

}

public void OnResultExecuted(ResultExecutedContext context)

{

// Can't add to headers here because response has started.

}

}

Il tipo di risultato eseguito dipende dall'azione. Un'azione che restituisce una visualizzazione include tutte le elaborazioni Razor come parte dell'elemento [ViewResult](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.viewresult) eseguito. Un metodo API può eseguire la serializzazione in quanto parte dell'esecuzione del risultato. Altre informazioni sui [risultati dell'azione](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.0)

I filtri risultato vengono eseguiti solo per i risultati corretti, ovvero quando l'azione o i filtri azione producono un risultato dell'azione. I filtri risultato non vengono eseguiti quando i filtri eccezioni gestiscono un'eccezione.

Il metodo [Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresultfilter.onresultexecuting) può causare il corto circuito dell'esecuzione del risultato dell'azione e dei filtri dei risultati successivi se si imposta [Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resultexecutingcontext.cancel#Microsoft_AspNetCore_Mvc_Filters_ResultExecutingContext_Cancel) su true. In caso di corto circuito, scrivere nell'oggetto risposta per evitare di generare una risposta vuota. La generazione di un'eccezione in IResultFilter.OnResultExecuting:

* Impedisce l'esecuzione del risultato dell'azione e dei filtri successivi.
* È considerata un errore anziché un risultato positivo.

Quando il metodo [Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresultfilter.onresultexecuted) viene eseguito:

* È probabile che la risposta sia stata inviata al client e non possa essere più modificata.
* Se è stata generata un'eccezione, il corpo della risposta non viene inviato.

L'impostazione di ResultExecutedContext.Canceled è true se un altro filtro ha causato il corto circuito dell'esecuzione del risultato dell'azione.

L'impostazione di ResultExecutedContext.Exception è un valore non Null se il risultato dell'azione o un filtro dei risultati successivo ha generato un'eccezione. Se si imposta Exception su Null l'eccezione viene gestita in modo efficace e si evita che venga generata nuovamente da ASP.NET Core in un punto successivo della pipeline. Non c'è alcun modo affidabile per scrivere i dati in una risposta quando si gestisce un'eccezione in un filtro dei risultati. Se le intestazioni sono state scaricate nel client quando il risultato di un'azione genera un'eccezione, non c'è alcun meccanismo affidabile per inviare un codice di errore.

Per un oggetto [IAsyncResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncresultfilter), una chiamata a await next in [ResultExecutionDelegate](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resultexecutiondelegate) esegue tutti i filtri dei risultati successivi e il risultato dell'azione. Per causare il corto circuito, impostare [ResultExecutingContext.Cancel](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.resultexecutingcontext.cancel#Microsoft_AspNetCore_Mvc_Filters_ResultExecutingContext_Cancel) su true e non chiamare ResultExecutionDelegate:

C#Copia

public class MyAsyncResponseFilter : IAsyncResultFilter

{

public async Task OnResultExecutionAsync(ResultExecutingContext context,

ResultExecutionDelegate next)

{

if (!(context.Result is EmptyResult))

{

await next();

}

else

{

context.Cancel = true;

}

}

}

Il framework fornisce un oggetto ResultFilterAttribute astratto che è possibile impostare come sottoclasse. La classe [AddHeaderAttribute](https://docs.microsoft.com/it-it/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.0#add-header-attribute) illustrata in precedenza è un esempio di un attributo di filtro dei risultati.

### IAlwaysRunResultFilter e IAsyncAlwaysRunResultFilter

Le interfacce [IAlwaysRunResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ialwaysrunresultfilter) e [IAsyncAlwaysRunResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncalwaysrunresultfilter) dichiarano un'implementazione di [IResultFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iresultfilter) eseguita per tutti i risultati dell'azione. Il filtro viene applicato a tutti i risultati dell'azione, a meno che:

* Non venga applicato un oggetto [IExceptionFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iexceptionfilter) o [IAuthorizationFilter](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.iauthorizationfilter) che causa il corto circuito della risposta.
* Un filtro di eccezione non gestisca un'eccezione producendo un risultato dell'azione.

I filtri diversi da IExceptionFilter e IAuthorizationFilter non causano il corto circuito di IAlwaysRunResultFilter e IAsyncAlwaysRunResultFilter.

Ad esempio, il filtro seguente viene eseguito sempre e imposta un risultato dell'azione ([ObjectResult](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.objectresult)) con un codice di stato 422 Entità non elaborabile quando la negoziazione del contenuto ha esito negativo:

C#Copia

public class UnprocessableResultFilter : Attribute, IAlwaysRunResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

if (context.Result is StatusCodeResult statusCodeResult &&

statusCodeResult.StatusCode == 415)

{

context.Result = new ObjectResult("Can't process this!")

{

StatusCode = 422,

};

}

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

### IFilterFactory

[IFilterFactory](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory) implementa [IFilterMetadata](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifiltermetadata). Pertanto, un'istanza di IFilterFactory può essere usata come un'istanza di IFilterMetadata in un punto qualsiasi della pipeline filtro. Quando il runtime si prepara per richiamare il filtro, cerca di eseguirne il cast a un oggetto IFilterFactory. Se l'esecuzione del cast ha esito positivo, viene chiamato il metodo [CreateInstance](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory.createinstance) per creare l'istanza di IFilterMetadata richiamata. In questo modo viene specificata una struttura flessibile, poiché non è necessario impostare in modo esplicito la pipeline filtro all'avvio dell'app.

Un altro approccio alla creazione di filtri consiste nell'implementare IFilterFactory usando implementazioni dell'attributo personalizzate:

C#Copia

public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory

{

// Implement IFilterFactory

public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)

{

return new InternalAddHeaderFilter();

}

private class InternalAddHeaderFilter : IResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add(

"Internal", new string[] { "My header" });

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

public bool IsReusable

{

get

{

return false;

}

}

}

Il codice precedente può essere testato eseguendo l'[esempio scaricato](https://github.com/aspnet/AspNetCore.Docs/tree/master/aspnetcore/mvc/controllers/filters/sample):

* Richiamare gli strumenti di sviluppo F12.
* Passare a https://localhost:5001/Sample/HeaderWithFactory

Gli strumenti di sviluppo F12 visualizzano le intestazioni di risposta seguenti aggiunte dal codice di esempio:

* **author:** Joe Smith
* **globaladdheader:** Result filter added to MvcOptions.Filters
* **internal:** My header

Il codice precedente crea l'intestazione di risposta **internal:** My header.

### Implementazione di IFilterFactory in un attributo

I filtri che implementano IFilterFactory sono utili per i filtri che:

* Non richiedono il passaggio di parametri.
* Hanno dipendenze del costruttore che devono essere soddisfatte dall'inserimento delle dipendenze.

[TypeFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.typefilterattribute) implementa [IFilterFactory](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory). IFilterFactory espone il metodo [CreateInstance](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifilterfactory.createinstance) per creare un'istanza [IFilterMetadata](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.filters.ifiltermetadata). CreateInstance carica il tipo specificato dal contenitore di inserimento delle dipendenze.

C#Copia

public class SampleActionFilterAttribute : TypeFilterAttribute

{

public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))

{

}

private class SampleActionFilterImpl : IActionFilter

{

private readonly ILogger \_logger;

public SampleActionFilterImpl(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();

}

public void OnActionExecuting(ActionExecutingContext context)

{

\_logger.LogInformation("Business action starting...");

// perform some business logic work

}

public void OnActionExecuted(ActionExecutedContext context)

{

// perform some business logic work

\_logger.LogInformation("Business action completed.");

}

}

}

Il codice seguente illustra tre approcci all'applicazione di [SampleActionFilter]:

C#Copia

[SampleActionFilter]

public IActionResult FilterTest()

{

return Content($"From FilterTest");

}

[TypeFilter(typeof(SampleActionFilterAttribute))]

public IActionResult TypeFilterTest()

{

return Content($"From ServiceFilterTest");

}

// ServiceFilter must be registered in ConfigureServices or

// System.InvalidOperationException: No service for type '<filter>' has been registered.

// Is thrown.

[ServiceFilter(typeof(SampleActionFilterAttribute))]

public IActionResult ServiceFilterTest()

{

return Content($"From ServiceFilterTest");

}

Nel codice precedente la decorazione del metodo con [SampleActionFilter] è l'approccio da preferire per applicare SampleActionFilter.

## Uso di middleware nella pipeline filtro

I filtri risorse funzionano come [middleware](https://docs.microsoft.com/it-it/aspnet/core/fundamentals/middleware/index?view=aspnetcore-3.0) in quanto racchiudono l'esecuzione di tutto ciò che viene dopo nella pipeline. I filtri sono diversi dal middleware in quanto fanno parte del runtime di ASP.NET Core, quindi hanno accesso al contesto e ai costrutti di ASP.NET Core.

Per usare il middleware come filtro, creare un tipo con un metodo Configure che specifica il middleware da inserire nella pipeline di filtro. L'esempio seguente usa il middleware di localizzazione per stabilire le impostazioni cultura correnti per una richiesta:

C#Copia

public class LocalizationPipeline

{

public void Configure(IApplicationBuilder applicationBuilder)

{

var supportedCultures = new[]

{

new CultureInfo("en-US"),

new CultureInfo("fr")

};

var options = new RequestLocalizationOptions

{

DefaultRequestCulture = new RequestCulture(culture: "en-US",

uiCulture: "en-US"),

SupportedCultures = supportedCultures,

SupportedUICultures = supportedCultures

};

options.RequestCultureProviders = new[]

{ new RouteDataRequestCultureProvider() { Options = options } };

applicationBuilder.UseRequestLocalization(options);

}

}

Usare [MiddlewareFilterAttribute](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.middlewarefilterattribute) per eseguire il middleware:

C#Copia

[Route("{culture}/[controller]/[action]")]

[MiddlewareFilter(typeof(LocalizationPipeline))]

public IActionResult CultureFromRouteData()

{

return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"

+ $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");

}

I filtri middleware vengono eseguiti nella stessa fase della pipeline filtro come filtri risorse, prima dell'associazione di modelli e dopo il resto della pipeline.