Mediate

DementCore

Aug 04, 2022

INTRODUCTION

1 What attempts to provide this project? 3

2 Contact 5

2.1 Mediator pattern . . . 5
2.2 Message types . . . 6
2.3 Installation . . . 7
2.4 Configuration . . . 8
2.5 Queries usage . . . 9
2.6 Events usage . . . 10
2.7 Advanced configuration . . . 11
2.8 Event Dispatch Strategies . . . 12
2.9 Middlewares . . . 13
2.10 Providers . . . 17
2.11 Generic event handlers . . . 19
2.12 Generic middlewares . . . 20
2.13 Background Event Dispatch . . . 22
2.14 Configuration . . . 22
2.15 Advanced configuration . . . 23
2.16 Event Queue Exception Handler . . . 23

Mediate is another simple and little in-process communication system based in mediator pattern.

CHAPTER

ONE

WHAT ATTEMPTS TO PROVIDE THIS PROJECT?

This project is mostly developed for learn and fun, but also attempts to provide an easy communication mechanism to develop decoupled communication between code layers.

CHAPTER

TWO

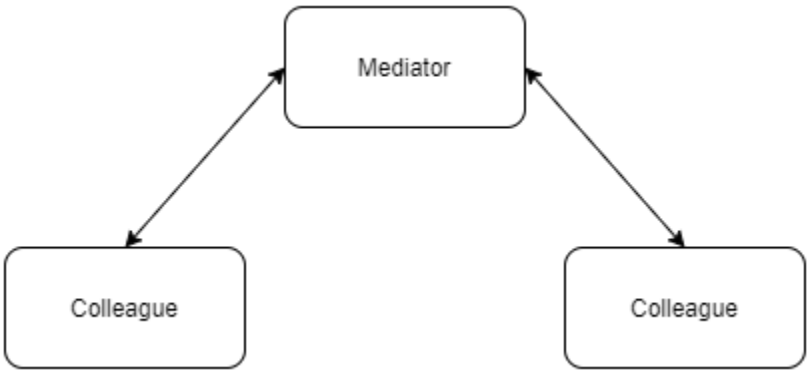
CONTACT

You can contact me in the project GitHub.

If you want request a feature, please create a Feature Request.

2.1 Mediator pattern

2.1.1 Pattern definition



The mediator pattern defines an object that encapsulates how different objects interacts.

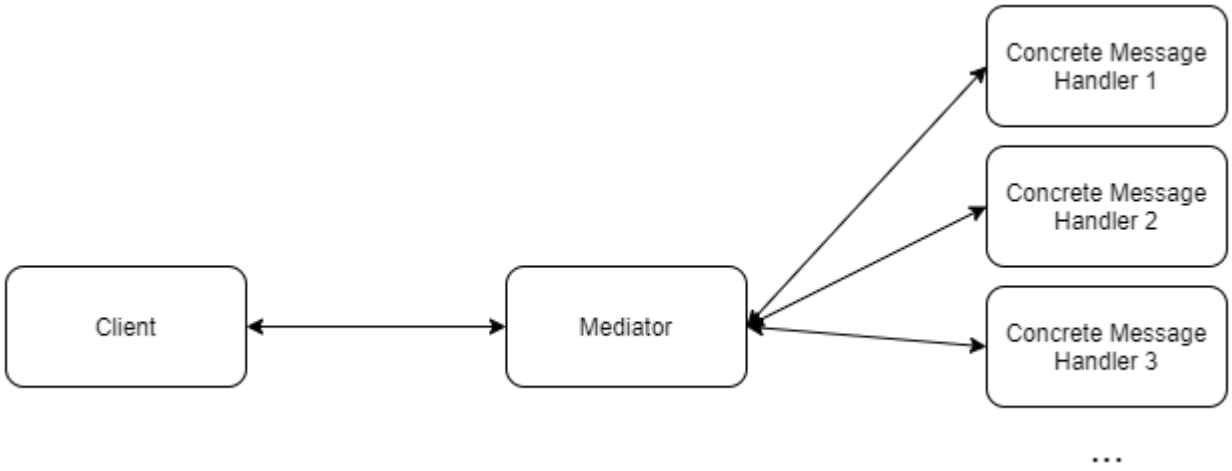
This ensures a low coupling between objects because none of them need to have a direct dependency on each other.

2.1.2 Participants in the pattern

Mediator
Is an intermediary that controls the communication between Colleagues.

Colleagues
Represents an object or a component in an application. Each Colleague communicates only with the Mediator.

2.1.3 Mediate implementation of the pattern



2.1.4 Participants in mediate implementation

Client
Is an application component or object that needs to communicate with other part of the application. The client sends a message to the Mediator.

Note: In the Mediator pattern the Client is a Colleague.

Concrete Message Handler
Represents an object or a component in an application that receives a concrete message from the Mediator and processes it.

Note: In the Mediator pattern the Concrete Message Handler is a Colleague.

2.2 Message types

In mediate we have two kinds of messages that represent different things:

2.2.1 Query

A query is a message that returns a response. You can think of a query like a question that receives an answer.

Note: A concrete query can only have a one concrete handler.

A query is defined by the `IQuery` generic interface

```
/// <summary>
/// Marker interface for defining a query with a response
/// </summary>
/// <typeparam name="TResult">Query response type</typeparam>
public interface IQuery<out TResult>
{
}
```

2.2.2 Event

An event is a message without response. You can think of an event like a notification that inform someone that something has happened.

Note: A concrete event can have multiple concrete handlers.

An event is defined by the `IEvent` interface

```
/// <summary>
/// Marker interface for defining an event
/// </summary>
public interface IEvent
{
}
```

2.3 Installation

To install Mediate you can use the NuGet package manager console, or the dotnet CLI.

Using the NuGet package manager console within Visual studio, run:

```
Install-Package Mediate
```

Or using the dotnet CLI:

```
dotnet add package Mediate
```

2.4 Configuration

Once installed, you will need to configure Mediate in your app's Startup class inside the `ConfigureServices` method. For this, Mediate provides some helper methods as `IServiceCollection` extension methods.

2.4.1 Mediate services configuration

First, you have to configure Mediate's basic services.

You can use one of the following methods:

- **`AddMediate`**
  Registers the mediator service with the default providers as scoped services. See *Providers* for details.
- **`AddMediateCore`**
  Registers the mediator service as a scoped service and returns a builder object that has a number of helper methods to configure Mediate. See *Advanced configuration* for details.

2.4.2 Event dispatching strategy configuration

Second, you have to configure the event dispatch strategy that you want to use.

You can use one of the following methods:

- **`AddMediateSequentialEventDispatchStrategy`**
  Registers the sequential event dispatch strategy as a scoped service. See *SequentialEventDispatchStrategy* for details.
- **`AddMediateParallelEventDispatchStrategy`**
  Registers the parallel event dispatch strategy as a scoped service. See *ParallelEventDispatchStrategy* for details.
- **`AddMediateCustomDispatchStrategy`**
  Registers a custom event dispatch strategy as a scoped service. You can also use `AddMediateCustomDispatchStrategy(ServiceLifetime)` to control the registration lifetime in the DI container. See *Custom Event Dispatch Strategy* for details.

2.4.3 Automatic handlers and middlewares registration

Third, you have to register your handlers and middlewares into the container.

You can do this manually or use the helper method `AddMediateClassesFromAssembly(Assembly)`.

This method scans the given assembly and registers the found classes as transient services. (open generics handlers and middlewares included).

2.5 Queries usage

Once configured, you will need to create the queries that you will send through the mediator.

2.5.1 Query creation

First, to create a query you have to implement the `IQuery<TResult>` generic interface.

For example:

```
public class MyQuery:IQuery<string>
{
    public string QueryData { get; set; }
}
```

Note: You can use any type that you want for the response.

2.5.2 Handler creation

Second, you have to create a handler for the above query. For this, you have to implement the `IQueryHandler<TQuery,TResponse>` interface.

```
/// <summary>
/// Interface for implement a query handler for a concrete query
/// </summary>
/// <typeparam name="TQuery">Query type</typeparam>
/// <typeparam name="TResult">Query response type</typeparam>
public interface IQueryHandler<in TQuery, TResult>
    where TQuery : IQuery<TResult>
{
    /// <summary>
    /// Handle the message
    /// </summary>
    /// <param name="message">Message data</param>
    /// <param name="cancellationToken"></param>
    /// <returns>Message response</returns>
    Task<TResult> Handle(TQuery message, CancellationToken cancellationToken);

}
```

For example:

```
public class MyQueryHandler : IQueryHandler<MyQuery, string>
{
   public Task<MyQueryResponse> Handle(MyQuery query, CancellationToken␣
↪cancellationToken)
   {
       //Example operation
       return Task.FromResult("Hello: " + query.QueryData);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.5.3 Sending through the mediator

Third, send the query through the mediator:

```
MyQuery query = new MyQuery() { QueryData = "Dementcore" };

string res = await _mediator.Send(query);
```

2.6 Events usage

To send events, is the same that you have already seen in the previous page.

2.6.1 Event creation

First, to create an event you have to implement the `IEvent` interface.

For example:

```
public class MyEvent:IEvent
{
   public string EventData { get; set; }
}
```

Note: In this class you can put any info that you need for your event.

2.6.2 Handler creation

Second, you have to create a handler for the above event. For this, you have to implement the `IEventHandler<TEvent>` generic interface.

```
/// <summary>
/// Interface for implement an event handler for an event
/// </summary>
/// <typeparam name="TEvent">Event type</typeparam>
public interface IEventHandler<in TEvent> where TEvent : IEvent
{
    /// <summary>
    /// Handle the event
    /// </summary>
    /// <param name="event">Event data</param>
    /// <param name="cancellationToken"></param>
    /// <returns></returns>
```

(continues on next page)

(continued from previous page)

```
    Task Handle(TEvent @event, CancellationToken cancellationToken);
}
```

For example:

```
public class MyEventHandler : IEventHandler<MyEvent>
{
    public Task Handle(MyEvent @event, CancellationToken cancellationToken)
    {
        //Example operation
        Console.WriteLine("MyEvent Event handler " + @event.EventData );

        return Task.CompletedTask;
    }
}
```

Tip: Remember, you can have multiple handlers for an event.

2.6.3 Dispatching through the mediator

Third, dispatch the event through the mediator:

```
MyEvent @event = new MyEvent() { EventData = "SomeData" };

await _mediator.Dispatch(@event, cancellationToken);
```

2.7 Advanced configuration

If you want to manually configure each Mediate service you can use the `AddMediateCore` extension method. See *Configuration* for details.

This method returns a builder object that has the following methods:

- **`AddServiceProviderHandlerProvider`**
  Registers a handler provider as a scoped service for queries and events that are retrieved from the Service Provider. See *Providers* for details.
- **`AddServiceProviderMiddlewareProvider`**
  Registers a middleware provider as a scoped service for queries and events that are retrieved from the Service Provider. See *Providers* for details.
- **`AddCustomHandlerProvider`**
  Registers a custom handler provider as a scoped service for queries and events. See *Custom Handler Provider* for details.
- **`AddCustomMiddlewareProvider`**
  Registers a custom middleware provider as a scoped service for queries and events. See *Custom Middleware Provider* for details.

2.8 Event Dispatch Strategies

An event dispatch strategy defines how the event handlers are invoked.

2.8.1 Default event dispatch strategies

Mediate has two event dispatch strategies available out-of-the-box.

Note: The event handlers are invoked in the same registration order.

Sequential Event Dispatch Strategy

This strategy executes event handlers after one another, sequentially.

Important: In case of exception in one event handler the rest of the handlers will be executed and then an AggregateException will be thrown when all handlers finish.

Parallel Event Dispatch Strategy

This strategy executes event handlers in parallel.

Important: In case of exception in one event handler the rest of the handlers will be executed and then an AggregateException will be thrown when all handlers finish.

2.8.2 Custom event dispatch strategy

You can execute the event handlers in any form that you want. For this purpose you have to implement the `IEventDispatchStrategy` interface with your custom logic.

```
/// <summary>
/// Interface for implement an event dispatch strategy
/// </summary>
public interface IEventDispatchStrategy
{
    /// <summary>
    /// Executes this strategy to dispatch an event
    /// </summary>
    /// <typeparam name="TEvent">Event type</typeparam>
    /// <param name="event">Event data</param>
    /// <param name="handlers">Event handlers</param>
    /// <returns></returns>
    Task Dispatch<TEvent>(TEvent @event, IEnumerable<IEventHandler<TEvent>> handlers)␣
↪where TEvent : IEvent;

    /// <summary>
```

(continues on next page)

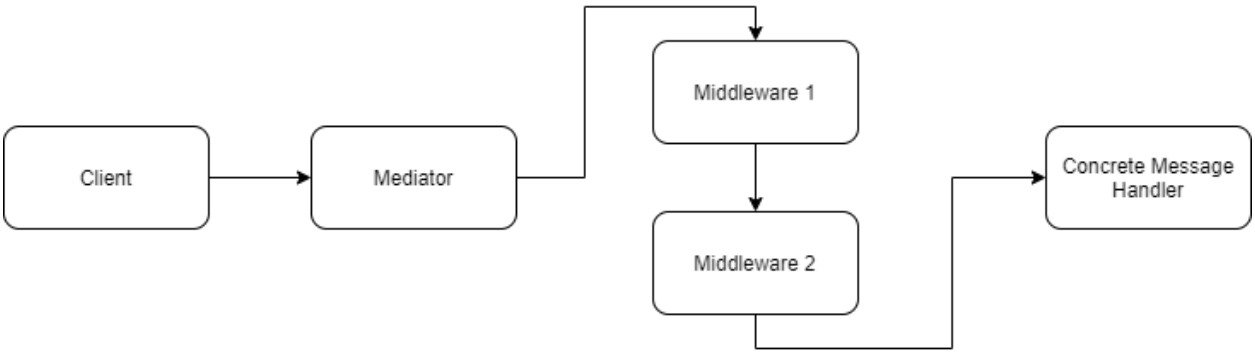
(continued from previous page)

```
    /// Executes this strategy to dispatch an event
    /// </summary>
    /// <typeparam name="TEvent">Event type</typeparam>
    /// <param name="event">Event data</param>
    /// <param name="handlers">Event handlers</param>
    /// <returns></returns>
    Task Dispatch<TEvent>(TEvent @event, IEnumerable<IEventHandler<TEvent>> handlers,␣
↪CancellationToken cancellationToken) where TEvent : IEvent;
}
```

To register your custom implementation you can use the `AddMediateCustomDispatchStrategy` extension method. See *Configuration*

2.9 Middlewares

A middleware is a piece of logic that is executed between the mediator and the message handler. It has been designed to be very similar to Asp.Net Core middlewares. You can use middlewares to create any pipeline that you need for your queries and events.



Each middleware can do the following things:

- Choose to invoke the next element in the pipeline.
- Do logic before and after calling the next element in the pipeline.

Note: The middlewares are invoked in the same registration order.

2.9.1 Query Middleware

A query middleware allows you to create a specific pipeline for a query.

Middleware creation

To create a query middleware you have to implement the `IQueryMiddleware<TQuery,TResult>` generic interface.

```
/// <summary>
/// Interface to implement a middleware to process a query before it reaches it's handler.
/// <typeparam name="TQuery">Query type</typeparam>
/// <typeparam name="TResult">Query response type</typeparam>
/// </summary>
public interface IQueryMiddleware<in TQuery, TResult> where TQuery : IQuery<TResult>
{
       /// <summary>
       /// Invoke the middleware logic
       /// </summary>
       /// <param name="query">Query object</param>
       /// <param name="cancellationToken"></param>
       /// <param name="next">Delegate that encapsulates a call to the next element in␣
↪the pipeline</param>
       /// <returns></returns>
       Task<TResult> Invoke(TQuery query, CancellationToken cancellationToken,␣
↪NextMiddlewareDelegate<TResult> next);
}
```

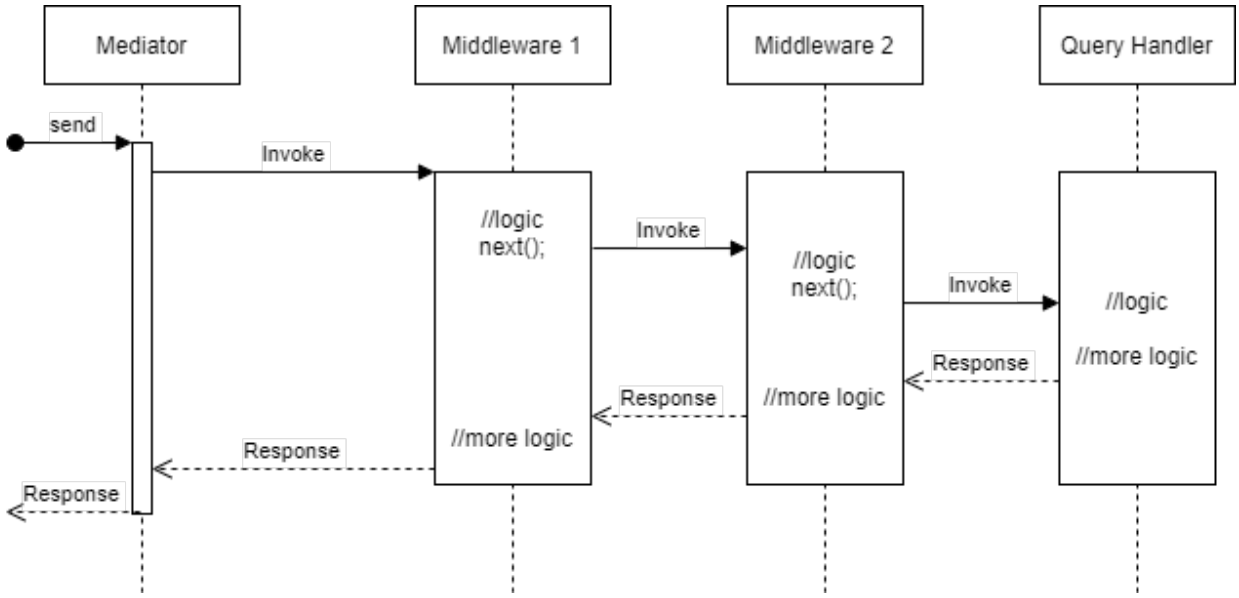
For example:

```
public class SampleQueryLoggerMiddleware : IQueryMiddleware<MyQuery, string>
{
     private readonly ILogger<SampleQueryLoggerMiddleware> _logger;

     public SampleQueryLoggerMiddleware(ILogger<SampleQueryLoggerMiddleware> logger)
     {
           _logger = logger;
     }

     public async Task<string> Invoke(MyQuery query, CancellationToken cancellationToken,
↪ NextMiddlewareDelegate<string> next)
     {
          _logger.LogDebug("Query log: ", query);

           //invoke the next middleware in the pipeline
           return await next();
     }
}
```

Note: The `NextMiddlewareDelegate<TResult> next` parameter is a delegate that encapsulates the call to the next element in the pipeline.

Query middleware pipeline

The following diagram demonstrates how middlewares are invoked under the hood.



Important: You have to invoke `next` to execute the next element in the pipeline. You can short circuit the pipeline simply don't invoking `next`.

2.9.2 Event Middleware

An event middleware allows you to create a specific pipeline for an event.

Middleware creation

To create an event middleware you have to implement the `IEventMiddleware<TEvent>` generic interface.

```
/// <summary>
/// Interface to implement a middleware to process an event before it reaches it's␣
↪handlers.
/// <typeparamref name="TEvent">Event type</typeparam>
/// </summary>
public interface IEventMiddleware<in TEvent> where TEvent : IEvent
{
    /// <summary>
    /// Invoke the middleware logic
    /// </summary>
    /// <param name="event">Event object</param>
    /// <param name="cancellationToken"></param>
    /// <param name="next">Delegate that encapsulates a call to the next element in the␣
↪pipeline</param>
    /// <returns></returns>
    Task Invoke(TEvent @event, CancellationToken cancellationToken,␣
```

(continues on next page)

(continued from previous page)

```
↪NextMiddlewareDelegate next);
}
```

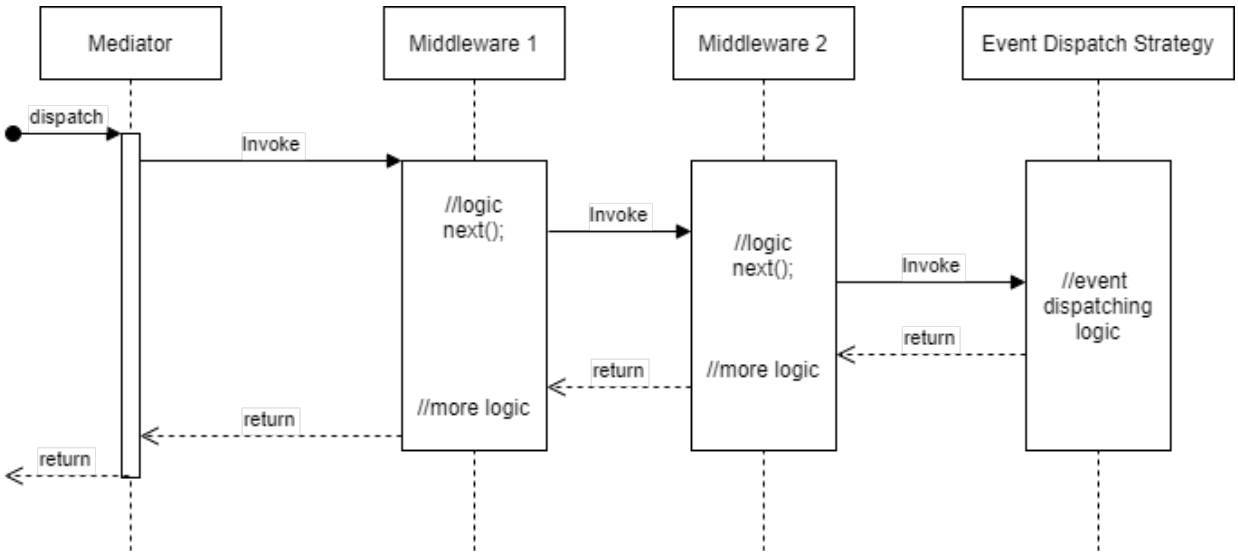
For example:

```
public class SampleEventLoggerMiddleware : IEventMiddleware<MyEvent>
{
     private readonly ILogger<SampleEventLoggerMiddleware> _logger;

     public SampleEventLoggerMiddleware(ILogger<SampleEventLoggerMiddleware> logger)
     {
           _logger = logger;
     }

     public async Task Invoke(MyEvent @event, CancellationToken cancellationToken,␣
↪NextMiddlewareDelegate next)
     {
          _logger.LogDebug("Event log: ", @event);

           //invoke the next middleware in the pipeline
           await next();
     }
}
```

Note: The `NextMiddlewareDelegate next` parameter is a delegate that encapsulates the call to the next element in the pipeline.

Event middleware pipeline

The following diagram demonstrates how middlewares are invoked under the hood.



Important: You have to invoke `next` to execute the next element in the pipeline. You can short circuit the pipeline

simply don't invoking `next`.

2.10 Providers

In Mediate a provider is the responsible for provide the message handlers and the middlewares.

2.10.1 Default Providers

Mediate includes the following providers out-of-the-box.

Service Provider Handler Provider

Provides the events and queries handlers from the DI Container.

Note: The handlers are retrieved in the same registration order.

Service Provider Middleware Provider

Provides the events and queries middlewares from the DI Container.

Note: The middlewares are retrieved in the same registration order.

2.10.2 Custom Handler Provider

You can create a custom provider for the handlers to provide handlers in any form you want. For this purpose you have to implement the `IHandlerProvider` interface with your custom logic.

```
/// <summary>
/// Defines a provider that encapsulates event and query handlers provider
/// </summary>
public interface IHandlerProvider : IEventHandlerProvider, IQueryHandlerProvider
{
}
```

The above interface is segregated in `IEventHandlerProvider` and `IQueryHandlerProvider` for more flexibility.

```
/// <summary>
/// Interface for implement an event handler provider
/// </summary>
public interface IEventHandlerProvider
{
        /// <summary>
        /// Gets all event handlers from an event
        /// </summary>
        /// <typeparam name="TEvent">Event type</typeparam>
```

(continues on next page)

(continued from previous page)

```
        /// <returns>All registered handlers for that event</returns>
        Task<IEnumerable<IEventHandler<TEvent>>> GetHandlers<TEvent>() where TEvent :␣
↪IEvent;
}

/// <summary>
/// Interface for implement a query handler provider
/// </summary>
public interface IQueryHandlerProvider
{
        /// <summary>
        /// Gets a query handler from a concrete query
        /// </summary>
        /// <typeparam name="TQuery">Query type</typeparam>
        /// <typeparam name="TResult">Query response type</typeparam>
        /// <returns>Registered handler for that query</returns>
        Task<IQueryHandler<TQuery, TResult>> GetHandler<TQuery, TResult>() where TQuery :␣
↪IQuery<TResult>;
}
```

To register your custom implementation you can use the `AddCustomHandlerProvider` advanced configuration method. See *Advanced configuration*

2.10.3 Custom Middleware Provider

You can create a custom provider for the middlewares to provide middlewares in any form you want. For this purpose you have to implement the `IMiddlewareProvider` interface with your custom logic.

```
/// <summary>
/// Defines a provider that encapsulates event and query middlewares provider
/// </summary>
public interface IMiddlewareProvider : IEventMiddlewareProvider, IQueryMiddlewareProvider
{

}
```

The above interface is segregated in `IEventMiddlewareProvider` and `IQueryMiddlewareProvider` for more flexibility.

```
/// <summary>
/// Interface for implement an event middleware provider
/// </summary>
public interface IEventMiddlewareProvider
{
        /// <summary>
        /// Gets all event middlewares from an event
        /// </summary>
        /// <typeparam name="TEvent">Event type</typeparam>
        /// <returns>All registered middlewares for that event</returns>
        Task<IEnumerable<IEventMiddleware<TEvent>>> GetMiddlewares<TEvent>() where TEvent␣
↪: IEvent;
```

(continues on next page)

(continued from previous page)

```
}

/// <summary>
/// Interface for implement a query middleware provider
/// </summary>
public interface IQueryMiddlewareProvider
{
        /// <summary>
        /// Gets all query middlewares from a query
        /// </summary>
        /// <typeparam name="TQuery">Query type</typeparam>
        /// <typeparam name="TResult">Query response type</typeparam>
        /// <returns>All registered middlewares for that query</returns>
        Task<IEnumerable<IQueryMiddleware<TQuery, TResult>>> GetMiddlewares<TQuery,␣
↪TResult>() where TQuery : IQuery<TResult>;
}
```

To register your custom implementation you can use the `AddCustomMiddlewareProvider` advanced configuration method. See *Advanced configuration*

2.11 Generic event handlers

You can create handlers that are invoked for all events.

To do that you have to create an open generic class implementing the `IEventHandler<TEvent>` interface. See *Events usage* for details.

For example this event handler will be invoked for all events:

```
/// <summary>
/// This class catchs all events
/// </summary>
public class GenericEventHandler<T> : IEventHandler<T> where T : IEvent
{
    private readonly ILogger<GenericEventHandler<T>> _logger;

    public GenericEventHandler(ILogger<GenericEventHandler<T>> logger)
    {
        _logger = logger;
    }

    public Task Handle(T @event, CancellationToken cancellationToken)
    {
        _logger.LogDebug("Received event: ", @event);

        return Task.CompletedTask;
    }
}
```

Another example. Imagine that you have an abstract class called `BaseEvent` for some events in your app, and you want to create a handler that is only invoked for all events that are derived from BaseEvent.

This will do the trick:

```
public abstract class BaseEvent : IEvent
{
    public Guid EventId { get; }

    public BaseEvent()
    {
        EventId = Guid.NewGuid();
    }
}

/// <summary>
/// This class catchs all BaseEvent derived events
/// </summary>
public class BaseEventGenericHandler<T> : IEventHandler<T> where T : BaseEvent
{
    private readonly ILogger<BaseEventGenericHandler<T>> _logger;

    public BaseEventGenericHandler(ILogger<BaseEventGenericHandler<T>> logger)
    {
        _logger = logger;
    }

    public Task Handle(T @event, CancellationToken cancellationToken)
    {
        _logger.LogDebug("Received base event derived event: ", @event);

        return Task.CompletedTask;
    }
}
```

Important: This depends on the DI container support for generic variance.

2.12 Generic middlewares

You can create middlewares that are invoked for all events or queries. To do that you have to create an open generic class implementing the middleware interfaces. See *Middlewares* for details about middlewares.

For example this middleware will be invoked for all events:

```
public class EventGenericMiddleware<TEvent> : IEventMiddleware<TEvent> where TEvent :␣
↪IEvent
{
   public async Task Invoke(TEvent @event, CancellationToken cancellationToken,␣
↪NextMiddlewareDelegate next)
   {

           //example validation
           if (@event == null)
           {
               //example exception for this example
```

(continues on next page)

(continued from previous page)

```
                throw new InvalidOperationException("The event must be not null");
            }

            await next();
    }
}
```

Another example. Imagine that you have an abstract class called `BaseEvent` for some events in your app, and you want to create a middleware that is only invoked for all events that are derived from BaseEvent.

This will do the trick:

```
public abstract class BaseEvent : IEvent
{
    public Guid EventId { get; }

    public BaseEvent()
    {
            EventId = Guid.NewGuid();
    }
}

//our middleware for all BaseEvent derived events
public class BaseEventGenericMiddleware<TEvent> : IEventMiddleware<TEvent> where TEvent␣
↪: BaseEvent
{
    public async Task Invoke(TEvent @event, CancellationToken cancellationToken,␣
↪NextMiddlewareDelegate next)
    {

        //example validation
        if (@event.EventId == Guid.Empty)
        {
                //example exception for this example
                throw new InvalidOperationException("The event id must be not null");
        }

        await next();
    }
}
```

Important: This depends on the DI container support for generic variance.

Note: For queries is the same concept.

```
public abstract class BaseQuery<TResult>:IQuery<TResult>
{
        public Guid QueryId { get; }
```

(continues on next page)

(continued from previous page)

```
        public BaseQuery()
        {
            QueryId = Guid.NewGuid();
        }
}

//middleware for all BaseQuery derived queries
public class BaseQueryGenericMiddleware<TQuery, TResult> : IQueryMiddleware<TQuery,␣
↪TResult> where TQuery : BaseQuery<TResult>
{
        public async Task<TResult> Invoke(TQuery query, CancellationToken␣
↪cancellationToken, NextMiddlewareDelegate<TResult> next)
        {
            //example validation
            if (query.QueryId != Guid.Empty)
            {
                return await next();
            }

            //example exception for this example
            throw new InvalidOperationException("The query id must be not null");
        }
}
```

2.13 Background Event Dispatch

Mediate also implements an event dispath strategy called `EventQueueDispatchStrategy`. This strategy enqueues event handlers to be executed in background by an Asp.Net Core hosted service.

To use this strategy you will need to install Mediate.BackgroundEventDispatch NuGet package.

Using the NuGet package manager console within Visual studio, run:

```
Install-Package Mediate.BackgroundEventDispatch
```

Or using the dotnet CLI:

```
dotnet add package Mediate.BackgroundEventDispatch
```

2.14 Configuration

Once installed, you will need to configure Mediate to use the `EventQueueDispatchStrategy`. For this, Mediate.BackgroundEventDispatch provides helper methods as `IServiceCollection` extension methods.

You can use one of the following methods:

- **`AddMediateEventQueueDispatchStrategy`**
  Registers `EventQueueDispatchStrategy` as a scoped service. This method also registers a hosted service called `EventDispatcherService` a singleton queue called `EventQueue` and a singleton exception handler called `DefaultEventQueueExceptionHandler`. See *EventQueueExceptionHandler* for details.

- **AddMediateEventQueueDispatchStrategyCore**
  Registers EventQueueDispatchStrategy as a scoped service and returns a builder object that has some helper methods to configure the strategy. See *Advanced configuration* for details.

  This method also registers a hosted service called EventDispatcherService and a singleton queue called EventQueue.

2.15 Advanced configuration

If you want to manually configure the strategy you can use the AddMediateEventQueueDispatchStrategyCore extension method. See *Configuration* for details.

This method returns a builder object that has the following methods:

- **AddDefaultExceptionHandler**
  Registers the DefaultEventQueueExceptionHandler that logs the error using the ILogger based logging system. . See *EventQueueExceptionHandler*
- **AddCustomExceptionHandler**
  Registers a custom EventQueueExceptionHandler See *EventQueueExceptionHandler*

2.16 Event Queue Exception Handler

A Event Queue Exception Handler is a piece of logic that handles the errors produced by event handlers when are executed in background by the event queue. You can use it to handle the exceptions in any form you want.

Important: In case of exception in one event handler the rest of the handlers will be executed and then an AggregateException will be generated.

Tip: As good practices, try to control the exceptions in the event handler.

2.16.1 Default Event Queue Exception Handler

This exception handler logs the errors using the ILogger based logging system.

```
/// <summary>
/// Default event dispatch exception handler that logs the exceptions
/// </summary>
public sealed class DefaultEventQueueExceptionHandler : IEventQueueExceptionHandler
{
    private readonly ILogger<DefaultEventQueueExceptionHandler> _logger;

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="logger"></param>
    public DefaultEventQueueExceptionHandler(ILogger<DefaultEventQueueExceptionHandler>␣
↪logger)
```

(continues on next page)

(continued from previous page)

```
    {
        _logger = logger;
    }

    /// <summary>
    /// Handles event dispatch exception
    /// </summary>
    /// <param name="aggregateException">Aggregate exception with all handlers errors</
↪param>
    /// <param name="eventName">Name of the event</param>
    /// <returns></returns>
    public Task Handle(AggregateException aggregateException, string eventName)
    {
        _logger.LogError(aggregateException, $"Errors occurred executing event
↪{eventName}");

        return Task.CompletedTask;
    }
}
```

2.16.2 Custom Event Queue Exception Handler

To create a Custom Event Queue Exception Handler you have to implement the `IEventQueueExceptionHandler` interface and then register it using `AddCustomExceptionHandler` method. See *Advanced configuration* for details.

```
/// <summary>
/// Interface for implement an exception handler for the event queue dispatch strategy
/// </summary>
public interface IEventQueueExceptionHandler
{
    /// <summary>
    /// Handles event queue dispatch strategy
    /// </summary>
    /// <param name="aggregateException">Aggregate exception with all handlers errors</
↪param>
    /// <param name="eventName">Name of the event</param>
    /// <returns></returns>
    Task Handle(AggregateException aggregateException, string eventName);
}
```