KnightBus Documentation

Release 1

BookBeat

Oct 21, 2021

Contents:

KnightBus is a fast, lightweight and extensible messaging framework that supports multiple active messaging transports

When building BookBeat we soon discovered that there was no silver bullet messaging technology, each one had its own pros and cons. Reliability, performance, latency, scalability, pricing and capabilites made us build KnightBus so that we could choose transport on a per message basis.

Features:

- **Multiple Transports**, active simultaneously on per message basis
- **Middleware**, write your own middleware to implement custom features
- **Attachments**, attach large files to your messages, transport independent
- **Singleton Processing**, make sure only one message is processed at a time regardless of number of instances running
- **Throttling**, both global and per message
- **IoC**, bring you own or use the default SimpleInjector

CHAPTER 1

Quick Start

The goal with Knightbus was to build a fast and simple messaging library that supports having multiple active messaging transports at the same time. There are many messaging frameworks available, but most of them are very complex and only support one active message transport, forcing developers to choose a fit-all messaging stack.

1.1 Installation

Find the official KnightBus packages at NuGet.org : https://www.nuget.org/profiles/BookBeat

1.2 Processing Messages

```
public class CommandProcessor : IProcessCommand<SampleCommand, SampleSettings>,
{
    public CommandProcessor(ISomeDependency dependency)
    {
        //You can use bring your own container for dependency injection
    }

    public Task ProcessAsync(SampleCommand message, CancellationToken␣
↪cancellationToken)
    {
        //Your code goes here
        return Task.CompletedTask;
    }
}

public class SampleCommand : IServiceBusCommand
{
    public string Message { get; set; }
}
```

(continues on next page)

(continued from previous page)

```
public class SampleCommandMapping : IMessageMapping<SampleCommand>
{
    public string QueueName => "your-queue-name";
}

public class SampleSettings : IProcessingSettings
{
    //These settings can be re-used by multiple message processors
    public int MaxConcurrentCalls => 100; //How many concurrent messages do you want␣
↪to process
    public TimeSpan MessageLockTimeout => TimeSpan.FromMinutes(1); //How long should␣
↪you process before considering the message hung
    public int DeadLetterDeliveryLimit => 1; //How many retries before dead lettering
    public int PrefetchCount => 50; //Increase perfomance by fetching more messages␣
↪at once
}
```

1.3 Starting as a Hosted Service (Recommended)

Using a hosted service will allow graceful(ish) shutdown of running instance and message processors.

```
class Program
{
    static void Main(string[] args)
    {
            var host = Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
                .UseKnightBus(new KnightBusHost()
                    //Multiple active transports
                    .UseTransport(new ServiceBusTransport("sb-connection"))
                    .UseTransport(new StorageBusTransport("storage-connection"))
                        .Configure(configuration => configuration
                            //Register our message processors without IoC using the␣
↪standard provider
                            //Register our message processors without IoC using the␣
↪standard provider
                            .UseDependencyInjection(new StandardDependecyInjection()
                                .RegisterProcessor(new␣
↪SampleServiceBusMessageProcessor())
                                .RegisterProcessor(new␣
↪SampleServiceBusEventProcessor())))
                )).Build();

            try
            {
                host.Run();
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
    }
}
```

1.4 Starting the KnightBus Host Standalone

```
class Program
{
    static async Task Main(string[] args)
    {
        var serviceBusConnection = "your-connection-string";

        var knightBusHost = new KnightBusHost()
            //Enable the ServiceBus Transport
            .UseTransport(new ServiceBusTransport(serviceBusConnection))
            .Configure(configuration => configuration
                //Register our message processors without IoC using the standard␣
↪provider
                //Register our message processors without IoC using the standard␣
↪provider
                .UseDependencyInjection(new StandardDependecyInjection()
                    .RegisterProcessor(new SampleServiceBusMessageProcessor())
                    .RegisterProcessor(new SampleServiceBusEventProcessor()))
            );

        await knightBusHost.StartAndBlockAsync(CancellationToken.None);
    }
}
```

1.5 Sending Messages

```
var client = new ServiceBus(new ServiceBusConfiguration(serviceBusConnection));
//Send some Commands
for (var i = 0; i < 10; i++)
{
    await client.SendAsync(new SampleCommand { Message = "Hello from message " + i.
↪ToString() });
}
```

1.6 Examples

You can find all current examples at our GitHub repository https://github.com/BookBeat/knightbus/tree/master/knightbus/examples

CHAPTER 2

KnightBus Host

The KnightBus host is the part of KnightBus responsible for invoking your MessageProcessors with the message sent on the transport. All configuration options for the host is exposed through the IHostConfiguration property from Configure()

2.1 Middlewares

Middlewares enable you to easily control what happens when processing messages. Much of KnightBus's internal mechanisms are implemented as Middlewares. Middlewares can be attached at either host or transport level. A host attached Middleware will be invoked for all transports where as a transport attached middleware will be specific for the transport only.

2.1.1 Default Middlewares

- ErrorHandlingMiddleware - Makes sure all exceptions are caught, logged and that the message is marked as failed. This Middleware is always added as the first Middleware.
- DeadLetterMiddleware - Dead letters messages that have failed to many times.

2.1.2 Optional Included Middlewares

- ThrottlingMiddleware - Allows throttling number of concurrent messages.
- AttachmentMiddleware - Enables the use of message attachments.
- ExtendMessageLockDurationMiddleware - Enables automatic message lock extension. Useful for very long running messages where you don't want to take an extremly long lock directly on the transport. Currently only works with Azure Storage Queues. Enable by having your ProcessingSetting implement *IExtendMessageLockTimeout*.

2.1.3 Pipeline

All Middlewares are executed in a Pipeline where the first component always is the ErrorHandlingMiddleware and the last is your actual implementation of the MessageProcessor

Since the Middlewares are executed in order, it is important to supply them in the order you want to execute them.

2.1.4 Writing your own Middleware

It's really easy to write your own custom Middleware, just implement the IMessageProcessorMiddleware and add the Middleware to the Pipeline on either host or transport level. Custom logging and performance monitoring are obvious use-cases.

```
public class MyCustomMiddleware : IMessageProcessorMiddleware
{
    public async Task ProcessAsync<T>(IMessageStateHandler<T> messageStateHandler,␣
↪IPipelineInformation pipelineInformation, IMessageProcessor next, CancellationToken␣
↪cancellationToken) where T : class, IMessage
    {
        Console.WriteLine("This is before the next step in the Pipeline");
        await next.ProcessAsync(messageStateHandler, cancellationToken).
↪ConfigureAwait(false); //Call the next Middleware in the Pipeline
        Console.WriteLine("This is after all later Middlewares have finished");
    }
}
```

2.2 Logging

You can bring your own logging framwork or use the supplied Serilog implementation in nuget KnightBus.Serilog. To use another framework simply implement the ILog interface from KnightBus.Core

2.3 Dependency Injection

KnightBus supports using your own IoC container or you can use the supplied SimpleInjector implementation in KnightBus.SimpleInjector.

The SimpleInjectorMessageProcessorProvider works together with the SimpleInjectorScopedLifeStyleMiddleware and gives you scoped access to resolving your MessageProcessors and injecting them with dependencies. The SimpleInjectorScopedLifeStyleMiddleware creates a per-message scoped lifestyle.

2.4 Singleton Locks

MessageProcessors can be marked as Singleton. This is done using the marker interface ISingletonProcessor.

A MessageProcessor marked with ISingletonProcessor will only run on one instance regardless of how you scale and will only process messages one at a time.

```
public class SingletonCommandProcessor : IProcessCommand<SingletonCommand, Settings>,␣
↪ISingletonProcessor
{
```

(continues on next page)

(continued from previous page)

```
    public Task ProcessAsync(SingletonCommand message, CancellationToken␣
↪cancellationToken)
    {
        //This code will never run concurrently
        return Task.CompletedTask;
    }
}
```

Since the SingletonLock is held globally a distributed locking mechanism must be supplied. By default KnightBus comes with an Azure implementation using Blob Storage leases.

To enable Singleton MessageProcessors simple supply the host with an ISingletonLockManager.

2.5 Hosting

The KnightBus Host can be hosted anywhere where you can run Console Applications. Currently all of our KnightBus Hosts are deployed using Kubernetes Pods, Azure WebJobs and TopShelf Windows Services.

CHAPTER 3

Messages

Everything processed using KnightBus is a considered a message and implements the IMessage interface. Messages are transport dependent, and need to implement the proper interface for the transport. All message haves a 1:1 relationship with a specific queue or topic.

3.1 Commands

Commands are messages that have a single recipient and is commanding the recipient to perform a task.

All commands implement ICommand through the specific transport implementation, for instance to send a command using the Azure Service Bus, you must implement IServiceBusCommand

3.2 Events

Events are messages that have a single dispatcher and multiple recievers and is telling the recievers that something has happened.

All events implement IEvent through the specific transport implementation, for instance to publish an event using the Azure Service Bus, you must implement IServiceBusEvent

3.3 Message Mappings

All Messages must also have a corresponding MessageMapping. KnightBus uses this to determine where to send and recieve the message. You implement this simply by adding your own implementation of IMessageMapping<MessageToMap>. When sending receiving messages KnightBus automatically finds these mappings.

```
public class MyMessage : IServiceBusCommand
{
    public string Message { get; set; }
```

(continues on next page)

(continued from previous page)

```
}

public class MyMessageMapping : IMessageMapping<MyMessage>
{
    public string QueueName => "some-queue-name";
}
```

The mapping class must reside in the same assembly as the message that is's mapping.

3.4 Sending Messages

To send a message you need to use the client for the specific message transport.

```
var serviceBusClient = new ServiceBus(new ServiceBusConfiguration("connectionString
↪"));
var storageBusClient = new StorageBus(new StorageBusConfiguration("connectionString
↪"));

await serviceBusClient.SendAsync(new MyServiceBusMessage());
await storageBusClient.SendAsync(new MyStorageBusMessage());
```

3.5 Custom Message Serialization

By default KnightBus uses Microsoft.Text.Json for message serialization. To override you can either change it per transport or for a specific message. To override for a specific Message set it on the message mapping.

```
public class CommandMapping : IMessageMapping<SomeCommand>, ICustomMessageSerializer
{
    public string QueueName => "testcommand";
    public IMessageSerializer MessageSerializer => new ProtobufNetSerializer();
}
```

3.6 Message Attachments

Regardless of the transport you can attach large files as attachments. Simply implement the ICommandWithAttachment interface on your command. The default implementation is for Azure Blob Storage using the BlobStorageMessageAttachmentProvider class which needs to be configured on the ITransportConfiguration supplied to the client and the host.

You can write your own implementation by implementing the IMessageAttachmentProvider interface. IMessageAttachmentProvider is separated by transport so you can use different attachment providers at the same time.

```
public class MyMessage : IServiceBusCommand, ICommandWithAttachment
{
    public string Message { get; set; }
    public IMessageAttachment Attachment { get; set; } //Here you can access the␣
↪attached file
}
```

3.7 Using Azure ServiceBus Creation Options Overrides For Queue/Topic

To tell the Azure ServiceBus queue/topic to override default creation options, add IServiceBusCreationOptions to IMessageMapping implementation.

```
public class MyMessage : IServiceBusCommand
{
    public string Message { get; set; }
}

public class MyMessageMapping : IMessageMapping<MyMessage>, IServiceBusCreationOptions
{
    public string QueueName => "your-queue";

    public bool EnablePartitioning => true;
    public bool SupportOrdering => false;
    public bool EnableBatchedOperations => true;
}
```

CHAPTER 4

Transports

Transports determine how your message is transported from Client to Processor. All current transports are listed here, but it's also fairly easy to extend KnightBus with other transports. A central concept is that each message is tagged with the transport it should use, for instance if you want to send a Command over Azure ServiceBus, implement the interface IServiceBusCommand.

4.1 Transport Configuration

TODO

4.2 Azure Service Bus

https://azure.microsoft.com/en-us/services/service-bus/

4.3 Azure Storage Bus

- commands
- attachments
- saga store
- singleton locks

https://azure.microsoft.com/en-us/services/storage/queues/

4.4 Redis

- commands

- events
- attachments
- saga store

The Redis transport supports both commands and events and is a very high performance transport.

KnightBus implements the Circular list pattern where all messages sent are stored on a list and when processed they are moved to another list making sure that no messages are lost in transit.

Pub/Sub is enabled by that the clients will find out what subscribers exists and publish to specific queue. An event that has three listeners will be published to three separed queues.

CHAPTER 5

Sagas

Sagas add state to your messagaging and allows you to handle long running processes within the same code. A Saga is a long lived transaction that is started by one or more specific messages and is finished by one or more messages. The messages can be either events or commands.

5.1 Setup

Setup KnightBus for Sagas by registering it in the host:

```
var knightBusHost = new KnightBusHost()
//Enable the StorageBus Transport
.UseTransport(new StorageTransport(storageConnection)
.Configure(configuration => configuration
    //Register our message processors without IoC using the standard provider
    .UseMessageProcessorProvider(new StandardMessageProcessorProvider()
        .RegisterProcessor(new SampleSagaMessageProcessor(client))
    )
    //Enable Saga support using the table storage Saga store
    .EnableSagas(new StorageTableSagaStore(storageConnection))
);

await knightBusHost.StartAsync();
```

5.2 Sample Implementation

```
class SampleSagaMessageProcessor: Saga<MySagaData>,
        IProcessCommand<SampleSagaMessage, SomeProcessingSetting>,
        IProcessCommand<SampleSagaStartMessage, SomeProcessingSetting>,
        ISagaDuplicateDetected<SampleSagaMessage>
    {
```

(continues on next page)

(continued from previous page)

```
        private readonly IStorageBus _storageBus;

        public SampleSagaMessageProcessor(IStorageBus storageBus)
        {
            _storageBus = storageBus;
            //Map the messages to the Saga
            MessageMapper.MapStartMessage<SampleSagaStartMessage>(m=> m.Id);
            MessageMapper.MapMessage<SampleSagaMessage>(m=> m.Id);
        }
        public override string PartitionKey => "sample-saga";
        public async Task ProcessAsync(SampleSagaStartMessage message,␣
↪CancellationToken cancellationToken)
        {
            Console.WriteLine(message.Message);
            await _storageBus.SendAsync(new SampleSagaMessage());
        }

        public async Task ProcessAsync(SampleSagaMessage message, CancellationToken␣
↪cancellationToken)
        {
            Console.WriteLine("Counter is {0}", Data.Counter);
            if (Data.Counter == 5)
            {
                Console.WriteLine("Finishing Saga");
                await CompleteAsync();
                return;
            }

            Data.Counter++;
            await UpdateAsync();
            await _storageBus.SendAsync(new SampleSagaMessage());
        }

        public Task ProcessDuplicateAsync(SampleSagaMessage message,␣
↪CancellationToken cancellationToken)
        {
            // Saga duplicate found

            return Task.CompletedTask;
        }
    }

    // This is exposed as `Data` property to classes that implements Saga<MySagaData>
    class MySagaData
    {
        public int Counter { get; set; }
    }

    class SomeProcessingSetting : IProcessingSettings
    {
        public int MaxConcurrentCalls => 1;
        public int PrefetchCount => 1;
        public TimeSpan MessageLockTimeout => TimeSpan.FromMinutes(5);
        public int DeadLetterDeliveryLimit => 2;
    }
```

CHAPTER 6

Versioning

KnightBus uses Semantic Versioning to make sure that it's easy for developers to understand the impact of new versions.

Read more about it here: Semantic Versioning

CHAPTER 7

Monitoring

Using the middleware pattern KnightBus can monitor message processing with any tool you'd like.

7.1 Already available monitoring middlewares

- New Relic
- Application Insights

7.1.1 New Relic

Install the package *KnightBus.NewRelic* and configure *KnightBusHost* to use New Relic.

```
var knightBus = new KnightBusHost()
    .UseTransport(...)
    .Configure(configuration => configuration
        .UseNewRelic()
        ...
    );
```

7.1.2 Liveness

TcpAliveListenerPlugin offers monitoring for liveness using TCP that can be used for services that don't serve http.

```
var knightBus = new KnightBusHost()
    .UseTransport(...)
    .Configure(configuration => configuration
        .UseTcpAliveListener(port: 13000)
        ...
    );
```

Example using Kubernetes:

```
livenessProbe:
 tcpSocket:
  port: {{  .Values.ports.liveness  }}
 initialDelaySeconds: 10
 periodSeconds: 10
 timeoutSeconds: 3
 successThreshold: 1
 failureThreshold: 5
```

See https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-a-tcp-liveness-probe for more information on how to configure liveness with tcp.

CHAPTER 8

Licence

KnightBus is licenced under the MIT licence.

Copyright (c) 2018 BookBeat

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 9

Credits

9.1 Main Author

Niklas Arbin, Systems Architect @ BookBeat

9.2 Contributors

Unordered list of persons contributing their time and intellect to KnightBus

- Tobias Johansson
- Albin Carnstam
- Viktor Hartenberger
- Magnus Baneryd
- Olov Siktröm
- Simon Aunér
- André Virdarson
- Tobias Balzano