# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

# Getting started

## Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

# Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

# Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

## Getting started

### Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

## Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

## Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# Namespace MicroPlumberd

## Classes

AggregateAttribute

AggregateBase<TState>

EventHandlerAttribute

EventStoreProjectionManagementClientExtensions

InvocationContext

InvocationScope

MetadataExtensions

OutputStreamAttribute

Plumber

## Structs

Metadata

## Interfaces

IAggregate

IAggregate<TSelf>

IConventions

IObjectSerializer

IPlumber

Root interface for plumber

IProjectionRegister

IReadModel

ISubscriptionRunner

ISubscriptionSet

ITypeRegister

## Enums

StandardMetadataEnricherTypes

## Delegates

EventIdConvention

EventNameConvention

GroupNameModelConvention

MetadataConvention

OutputStreamModelConvention

SteamNameConvention

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

## Getting started

### Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

## Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

## Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

# Getting started

## Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

# Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

# Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

## Getting started

### Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

## Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

## Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

## Getting started

### Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

## Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

## Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.

# micro-plumberd

Micro library for EventStore, CQRS and EventSourcing Just eXtreamly simple.

## Getting started

### Install nugets:

```
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGeneratiors
```

If you'd like to use direct dotnet-dotnet communication to execute command-handlers install MicroPlumberd.DirectConnect

```
dotnet add package MicroPlumberd.DirectConnect
```

## Configure plumber

```
/// change to your connection-string.
string connectionString = $"esdb://admin:changeit@localhost:2113?
tls=false&tlsVerifyCert=false";
var seetings = EventStoreClientSettings.Create(connectionString);

var plumber = new Plumber(settings) as IPlumber;
```

## Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    internal new FooState State => base.State;
    public record FooState { public string Name { get; set; } };
    private static FooState Given(FooState state, FooCreated ev) => state with { Name =
ev.Name };
    private static FooState Given(FooState state, FooUpdated ev) => state with { Name
=ev.Name };
    public void Open(string msg) => AppendPendingChange(new FooCreated() { Name = msg });
    public void Change(string msg) => AppendPendingChange(new FooUpdated() { Name = msg });
}
```

Comments:

- State is encapsulated in nested class FooState.
- Given methods, that are used when loading aggregate from the EventStoreDB are private and static. State is encouraged to be immutable.
- [Aggregate] attribute is used by SourceGenerator that will generate dispatching code and handy metadata.

2. Consume an aggregate.

If you want to create a new aggregate and save it to EventStoreDB:

```
FooAggregate aggregate = FooAggregate.New(Guid.NewGuid());
aggregate.Open("Hello");

await plumber.SaveNew(aggregate);
```

If you want to load aggregate from EventStoreDB, change it and save back to EventStoreDB

```
var aggregate = await plumber.Get<FooAggregate>("YOUR_ID");
aggregate.Change("World");
await plumber.SaveChanges(aggregate);
```

# Write a read-model/processor

1. Read-Models

```
[EventHandler]
public partial class FooModel
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
}
```

Comments:

- ReadModels have private async Given methods. Since they are async, you can invoke SQL here, or othere APIs to store your model.
- Metadata contains standard stuff (Created, CorrelationId, CausationId), but can be reconfigured.

```
var fooModel = new FooModel();
var sub= await plumber.SubscribeModel(fooModel);

// or if you want to persist progress of your subscription
var sub2= await plumber.SubscribeModelPersistently(fooModel);
```

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream.

2. Processors

```
[EventHandler]
public partial class FooProcessor(IPlumber plumber)
{
    private async Task Given(Metadata m, FooUpdated ev)
    {
        var agg = FooAggregate.New(Guid.NewGuid());
        agg.Open(ev.Name + " new");
        await plumber.SaveNew(agg);
    }
}
```

Implementing a processor is technically the same as implementing a read-model, but inside the Given method you would typically invoke a command or execute an aggregate.

# Features

## Conventions

- SteamNameConvention - from aggregate type, and aggregate id
- EventNameConvention - from aggregate? instance and event instance
- MetadataConvention - to enrich event with metadata based on aggregate instance and event instance
- EventIdConvention - from aggregate instance and event instance
- OutputStreamModelConvention - for output stream name from model-type
- GroupNameModelConvention - for group name from model-type

## Ultra development cycle for Read-Models (EF example)

Imagine this:

1. You create a read-model that subscribes persistently.
2. You subscribe it with plumber.
3. You changed something in the event and want to see the new model.
4. Instead of re-creating old read-model, you can easily create new one. Just change MODEL_VER to reflect new version.

*Please note that Sql schema create/drop auto-generation script will be covered in a different article. (For now we leave it for developers.)*

Comments:

- By creating a new read-model you can always compare the differences with the previous one.
- You can leverage canary-deployment strategy and have 2 versions of your system running in parallel.

```
[OutputStream(FooModel.MODEL_NAME)]
[EventHandler]
public partial class FooModel : DbContext
{
    internal const string MODEL_VER = "_v1";
    internal const string MODEL_NAME = $"FooModel{MODEL_VER}";
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
           .Entity<FooEntity>()
           .ToTable($"FooEntities{MODEL_VER}");
    }
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
        // your code
    }
}
```

# Subscription Sets - Models ultra-composition

- You can easily create a stream that joins events together by event-type.

# GRPC Direct communication

```
/// Let's configure server:
services.AddCommandHandler<FooCommandHandler>().AddServerDirectConnect();

/// Add mapping to direct-connect service
app.MapDirectConnect();
```

Here is an example of a command handler code:

```
[CommandHandler]
public partial class FooCommandHandler(IPlumber plumber)
{

    [ThrowsFaultException<BusinessFault>]
    public async Task Handle(Guid id, CreateFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = FooAggregate.New(id);
        agg.Open(cmd.Name);

        await plumber.SaveNew(agg);
    }

    [ThrowsFaultException<BusinessFault>]
    public async Task<HandlerOperationStatus> Handle(Guid id, ChangeFoo cmd)
    {
        if (cmd.Name == "error")
            throw new BusinessFaultException("Foo");

        var agg = await plumber.Get<FooAggregate>(id);
        agg.Change(cmd.Name);

        await plumber.SaveChanges(agg);
        return HandlerOperationStatus.Ok();
    }
}
```

And how on the client side:

```
service.AddClientDirectConnect().AddCommandInvokers();

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
```

```
var invoker = clientPool.Get("YOUR_GRPC_URL");
await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

## Aspects

You can easily inject aspects through decorator pattern.