micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

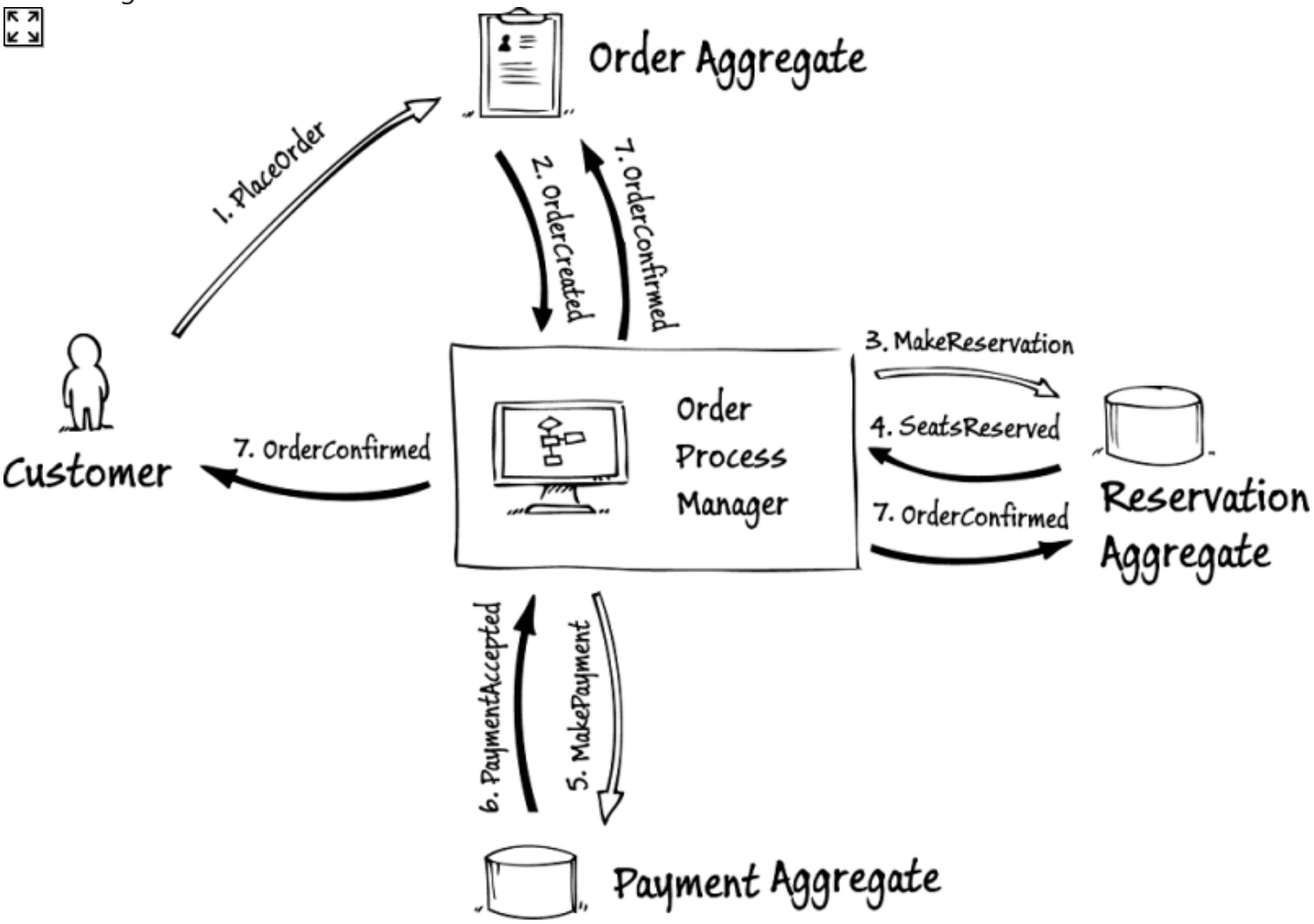
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

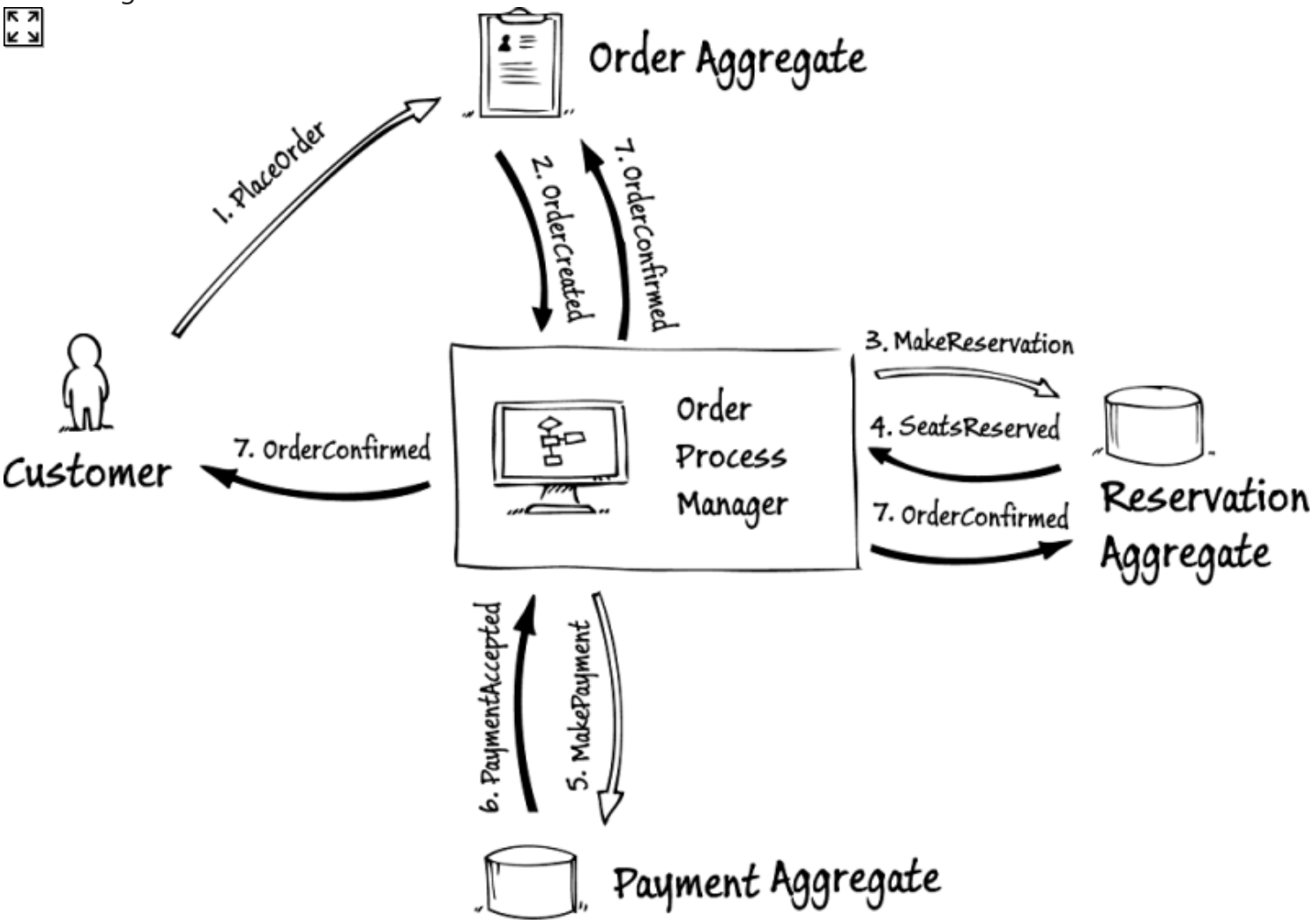
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

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:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

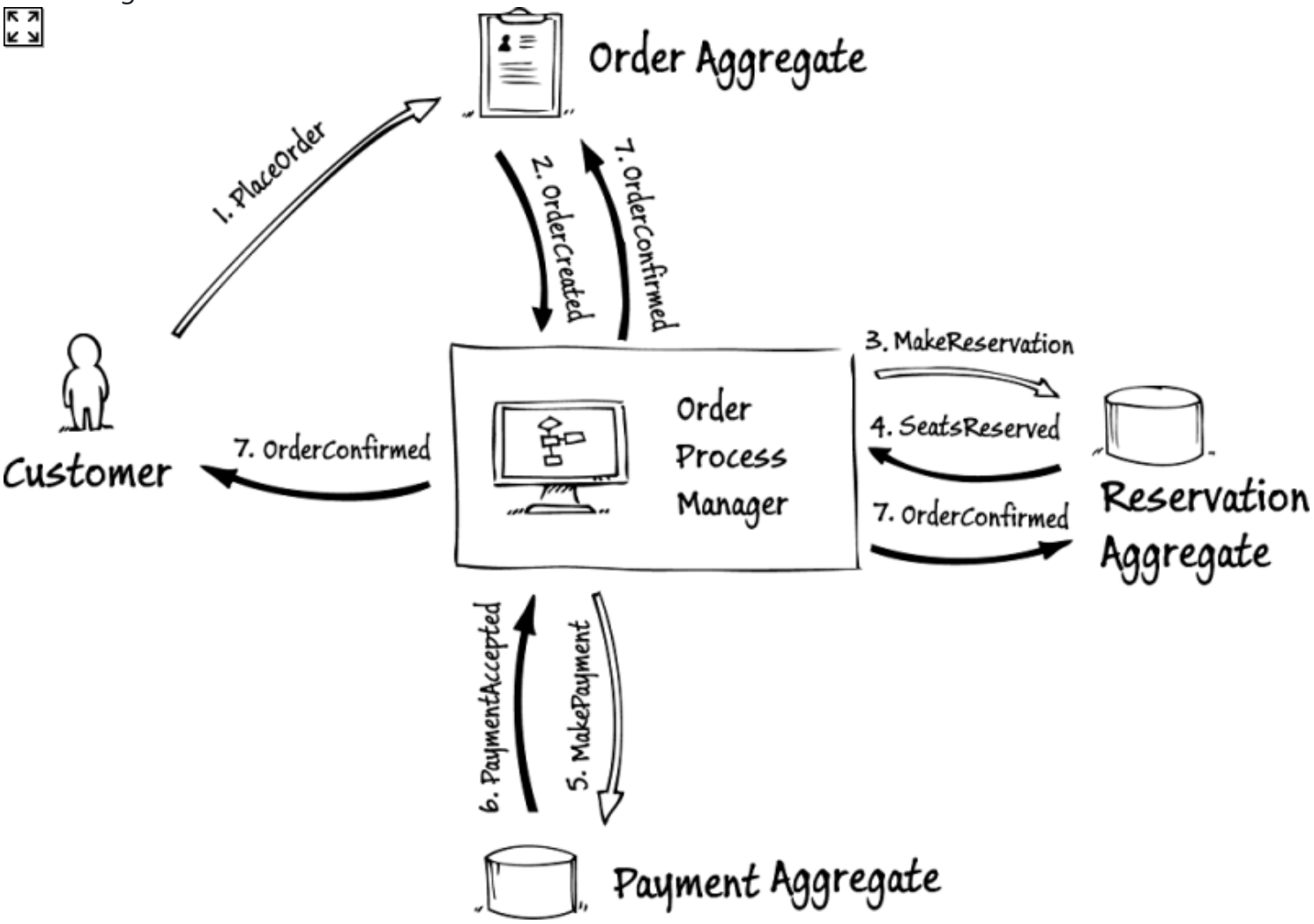
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

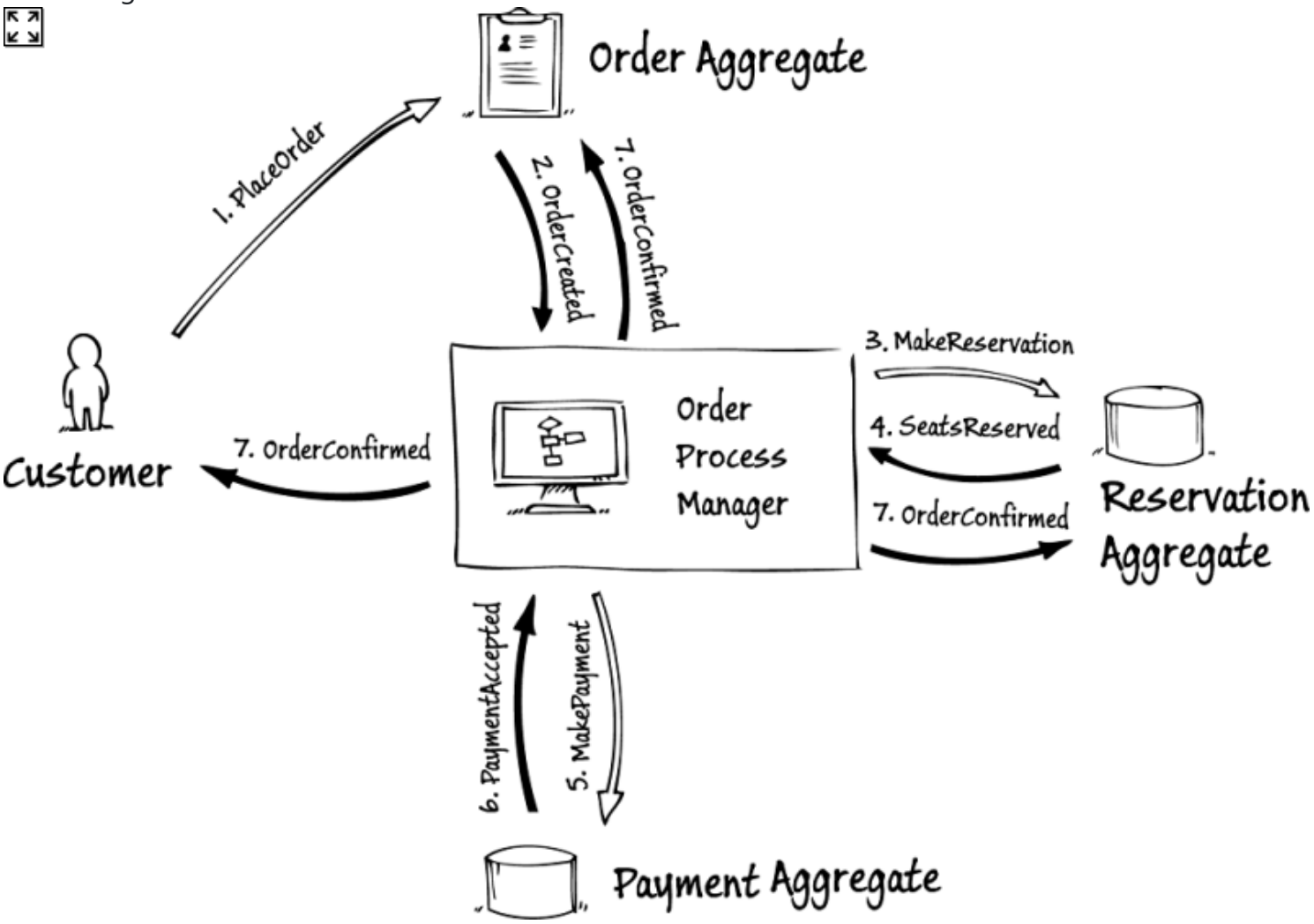
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

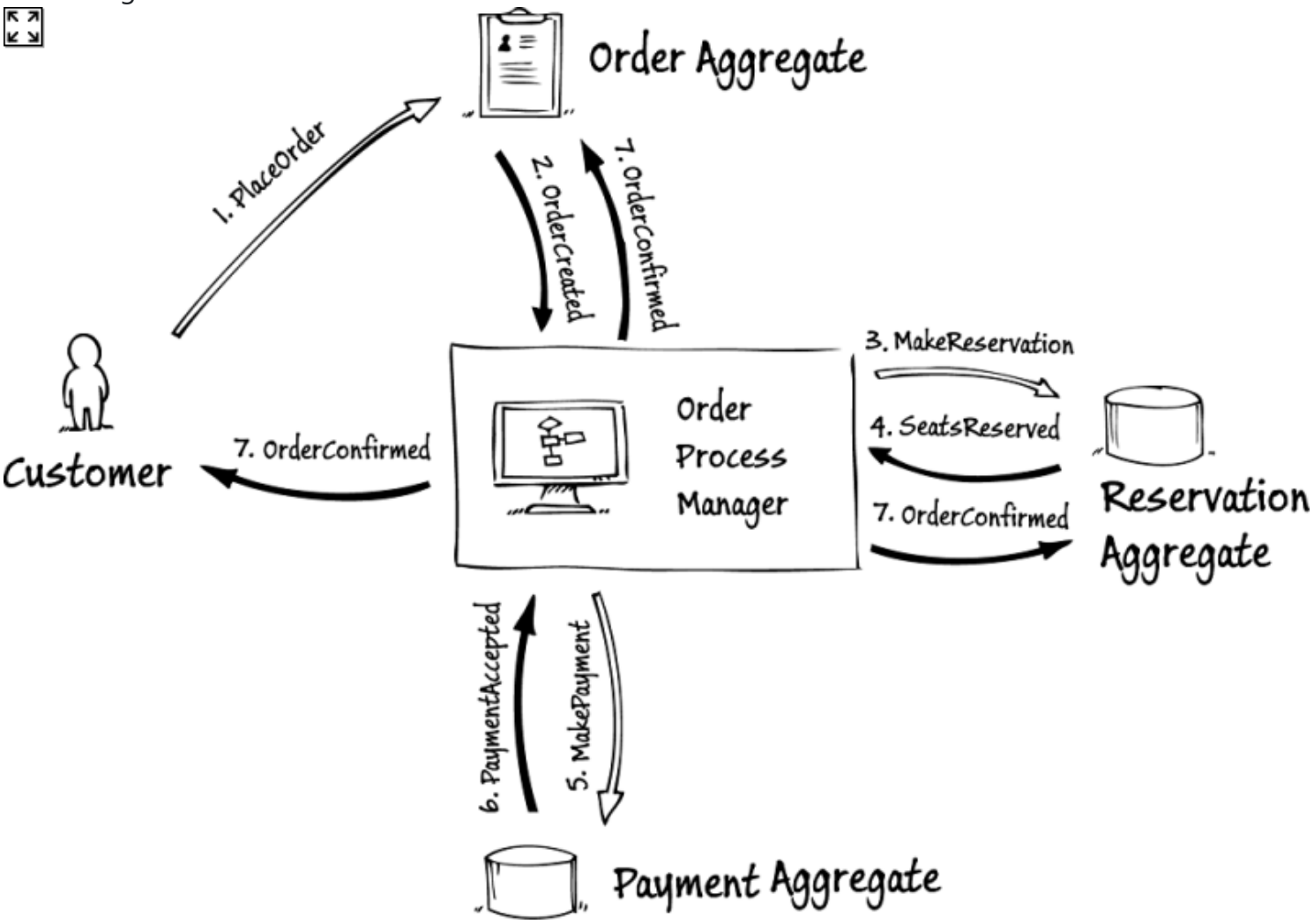
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

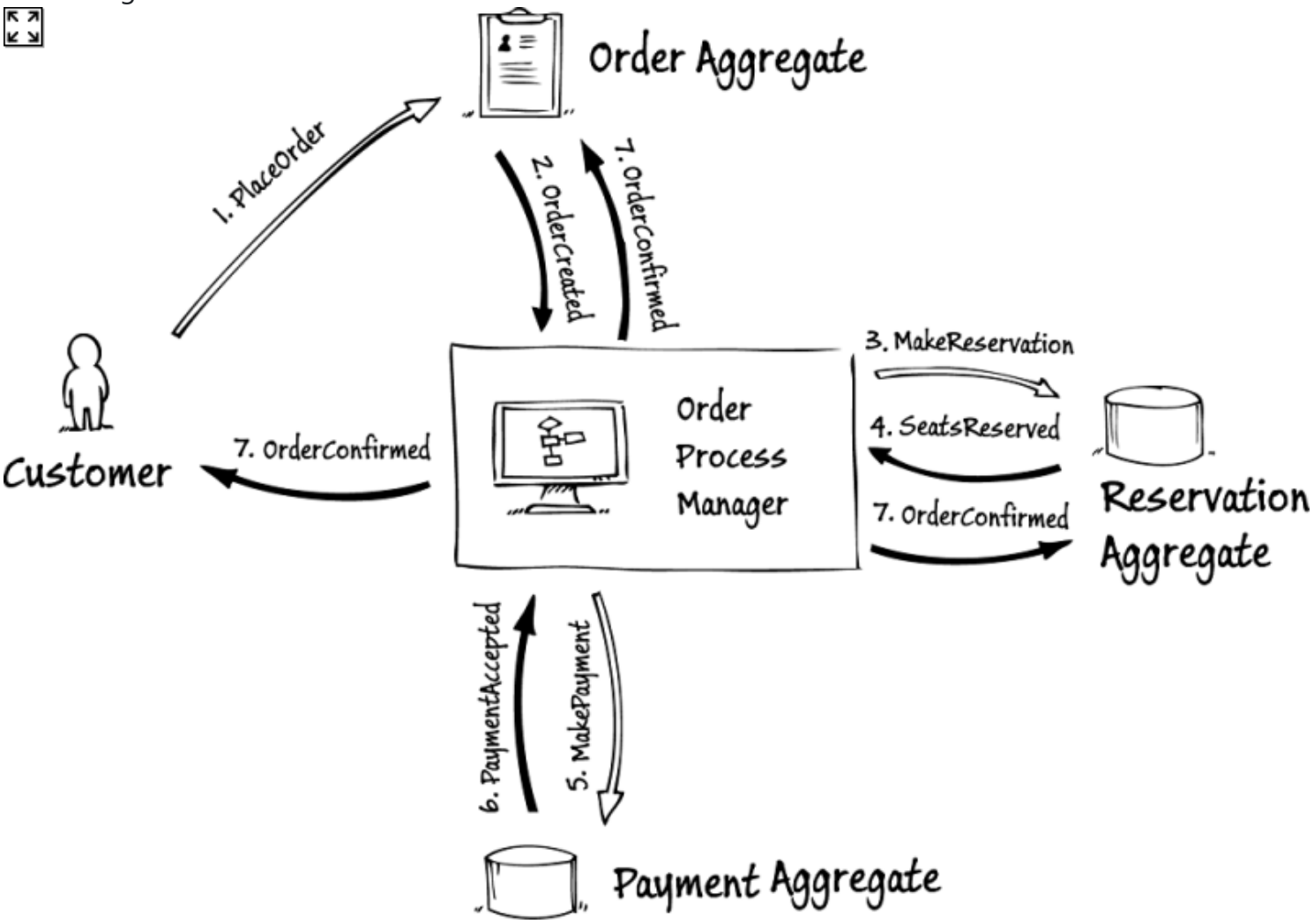
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

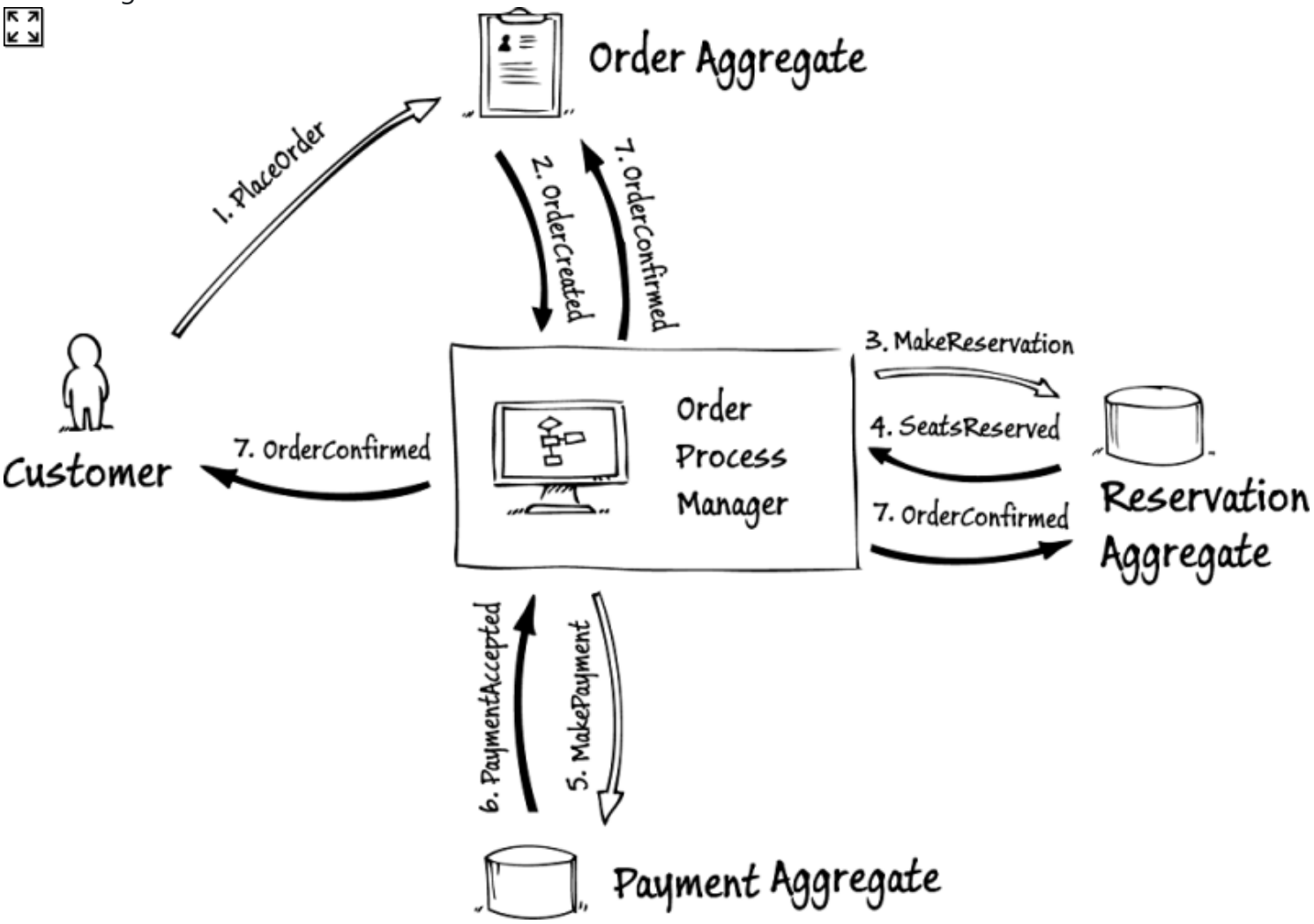
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

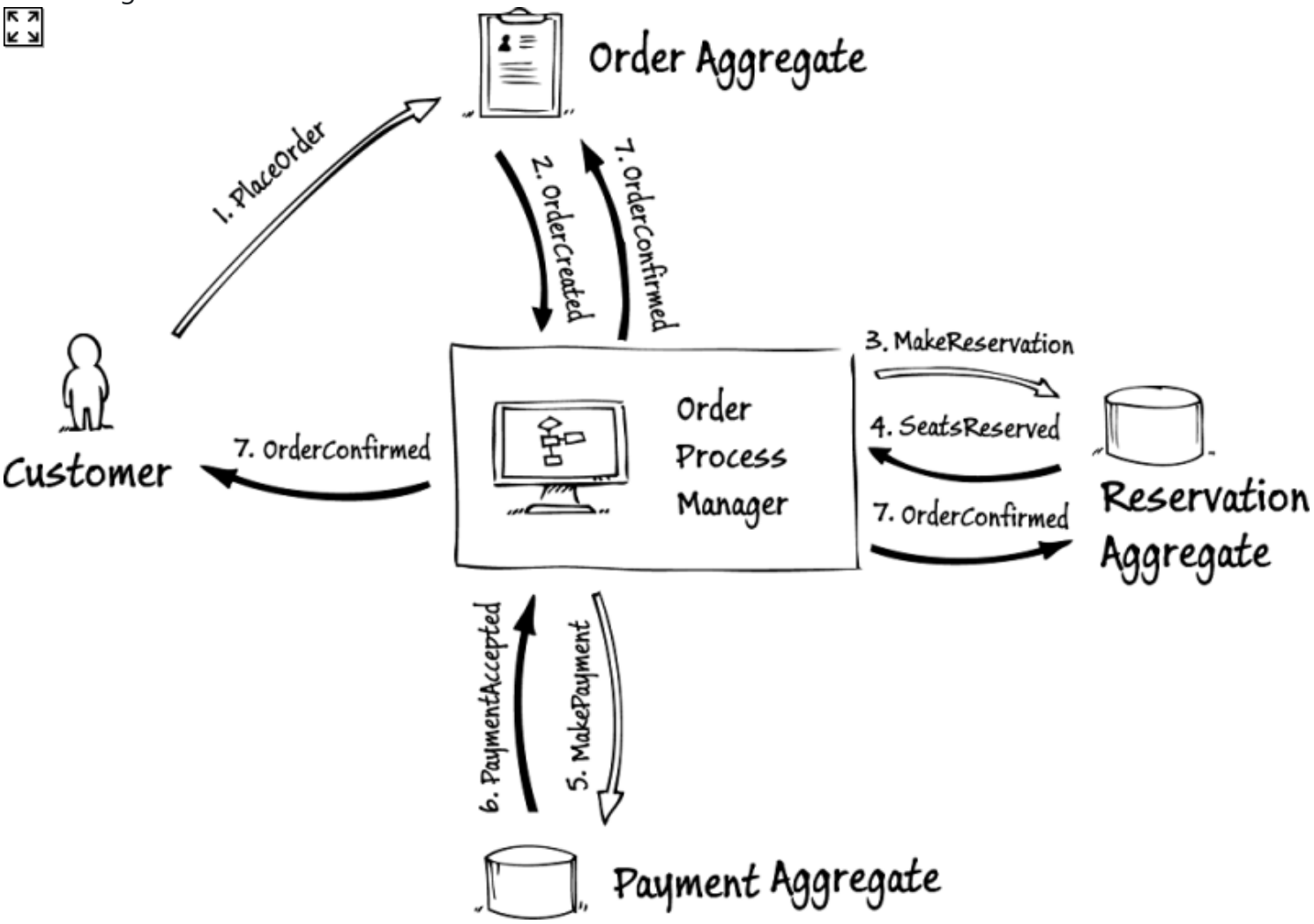
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

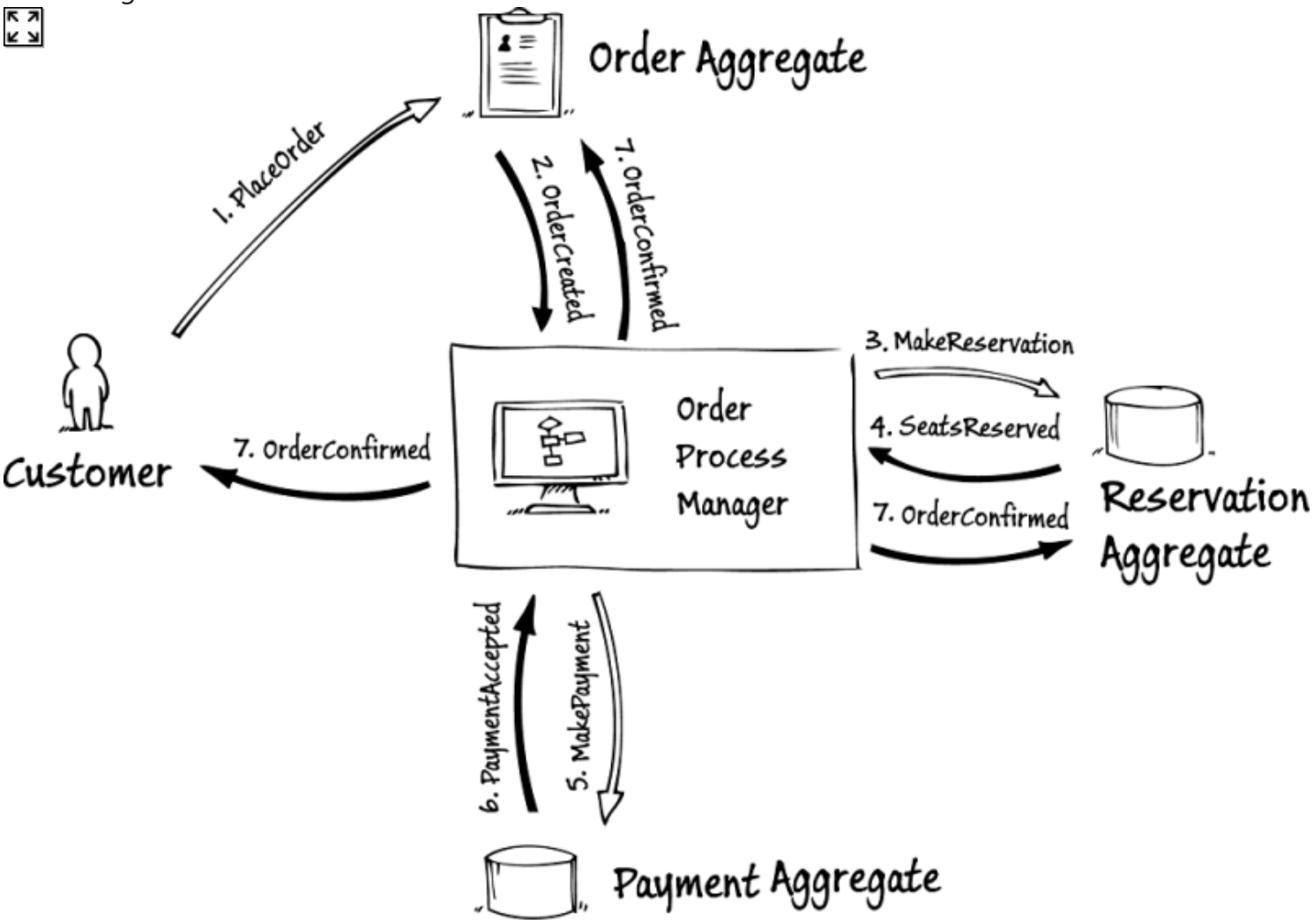
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```

micro-plumberd

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

Getting started

Install nugets:

```
# For your domain
dotnet add package MicroPlumberd
dotnet add package MicroPlumberd.SourceGenerators
```

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

```
# For application-layer using EventStore as message-bus.
dotnet add package MicroPlumberd.Services

# For application-layer communicating (dotnet-2-dotnet) using GRPC:
dotnet add package MicroPlumberd.Services.Grpc.DirectConnect

# EXPERIMENTAL ProcessManager support can be found here:
dotnet add package MicroPlumberd.Services.ProcessManagers
```

Configure plumber

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

If you'd want to do it at service-level with DI:

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

services.AddPlumberd(settings);
```

Aggregates

1. Write an aggregate.

```
[Aggregate]
public partial class FooAggregate(Guid id) : AggregateBase<FooAggregate.FooState>(id)
{
    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 });
}
// And events:
public record FooCreated { public string? Name { get; set; } }
public record FooUpdated { public string? Name { get; set; } }
```

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.SubscribeEventHandler(fooModel);

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

With **SubscribeModel** you can subscribe from start, from certain moment or from the end of the stream. If you want to use DI and have your model as a scoped one, you can configure plumber at the startup and don't need to invoke SubscribeEventHandler manually. Here you have an example with EF Core.

```
// Program.cs
services
    .AddPlumberd()
    .AddEventHandler<FooModel>();

// FooModel.cs
[EventHandler]
public partial class FooModel : DbContext
{
    private async Task Given(Metadata m, FooCreated ev)
    {
        // your code
    }
    private async Task Given(Metadata m, FooUpdated ev)
    {
         // your code
    }
    // other stuff, DbSet... etc...
}
```

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, and subscribe many read-models at once. Here it is named 'MasterStream', which is created out of events used to create DimentionLookupModel and MasterModel.
- In this way, you can easily manage the composition and decoupling of read-models. You can nicely composite your read-models. And if you don't wish to decouple read-models, you can reuse your existing one.

```
/// Given simple models, where master-model has foreign-key used to obtain value
from dimentionLookupModel

var dimentionTable = new DimentionLookupModel();
var factTable = new MasterModel(dimentionTable);

await plumber.SubscribeSet()
    .With(dimentionTable)
    .With(factTable)
    .SubscribeAsync("MasterStream", FromStream.Start);
```

EventStoreDB as message-bus

If you want to start as quickly as possible, you can start with EventStoreDB as command-message-bus.

```
services.AddPlumberd()
        .AddCommandHandler<FooCommandHandler>()

// on the client side:
ICommandBus bus; // from DI
bus.SendAsync(Guid.NewGuid(), new CreateFoo() { Name = "Hello" });
```

If you are running many replicas of your service, you need to switch command-execution to persistent mode:

```
services.AddPlumberd(configure: c =>
c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true)
        .AddCommandHandler<FooCommandHandler>()
```

This means, that once your microservice subscribes to commands, it will execute all. So if your service is down, and commands are saved, once your service is up, they will be executed. To skip old commands, you can configure a filter.

```
services.AddPlumberd(configure: c => {
    c.Conventions.ServicesConventions().AreHandlersExecutedPersistently = () => true;
    c.Conventions.ServicesConventions().CommandHandlerSkipFilter = (m,ev) =>
DateTimeOffset.Now.Substract(m.Created()) > TimeSpan.FromSeconds(60);
    })
    .AddCommandHandler<FooCommandHandler>()
```

GRPC Direct communication

If you prefer direct communication (like REST-API, but without the hassle for contract generation/etc.) you can use direct communication where client invokes command handle using grpc. Command is not stored in EventStore.

```
/// 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();
    }
}
```

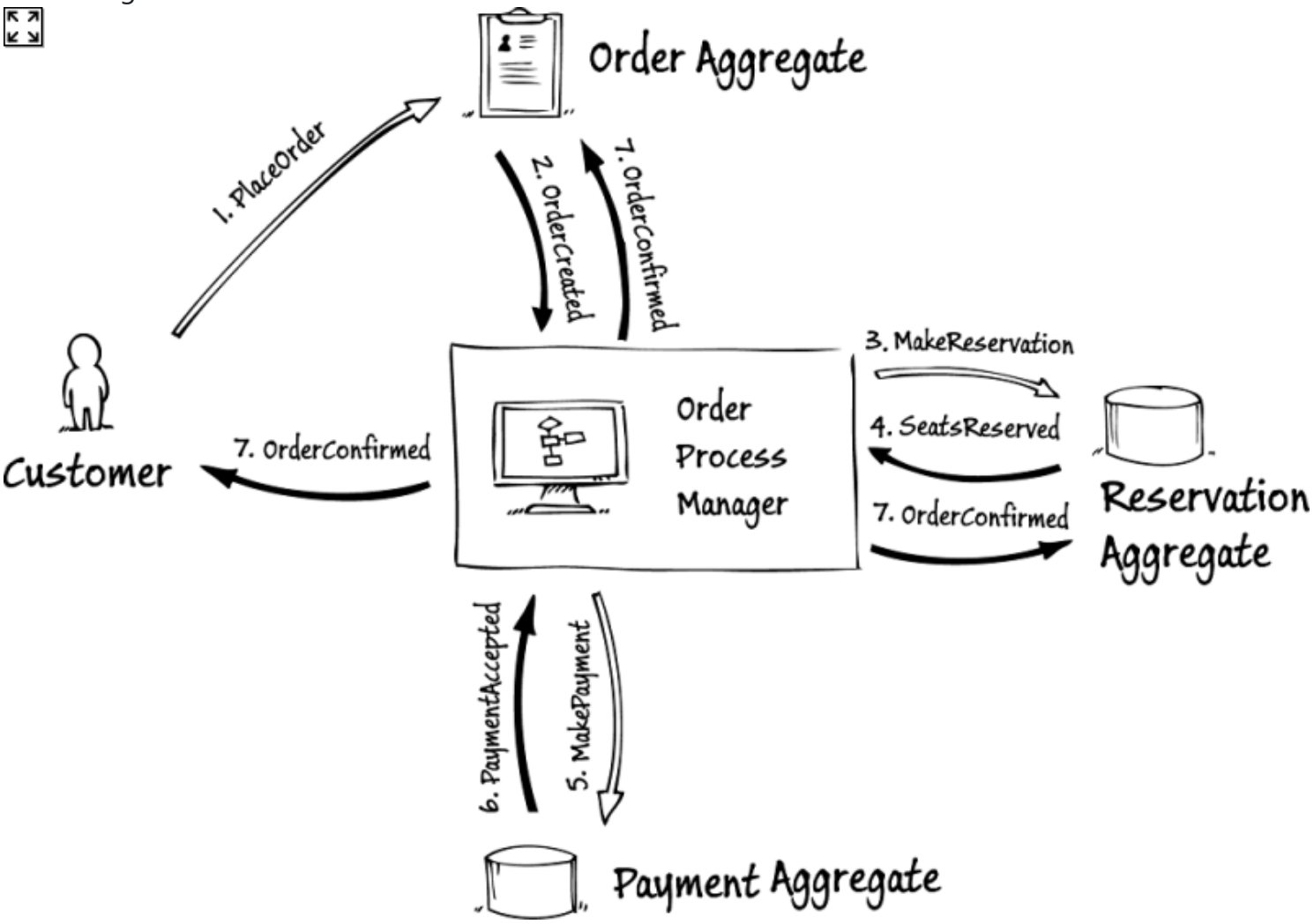
And how on the client side:

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

// And invocation
 var clientPool = sp.GetRequiredService<IRequestInvokerPool>();
 var invoker = clientPool.Get("YOUR_GRPC_URL");
 await invoker.Execute(Guid.NewId(), new CreateFoo(){});
```

EXPERIMENTAL Process-Manager

Given diagram:



The code of Order Process Manager looks like this:

```
// Let's configure stuff beforehand
services.AddPlumberd(eventStoreConfig)
    .AddCommandHandler<OrderCommandHandler>() // handles PlaceOrder command.
    .AddProcessManager<OrderProcessManager>();

// And process manager.
[ProcessManager]
public class OrderProcessManager(IPlumberd plumberd)
{
    public async Task<ICommandRequest<MakeReservation>> StartWhen(Metadata m,
OrderCreated e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakeReservation());
    }
    public async Task<ICommandRequest<MakePayment>> When(Metadata m, SeatsReserved e)
    {
        return CommandRequest.Create(Guid.NewId(), new MakePayment());
```

```
    }
    public async Task When(Metadata m, PaymentAccepted e)
    {
        var order = await plumberd.Get<Order>(this.Id);
        order.Confirm();
        await plumberd.SaveChanges(order);
    }
    // Optional
    private async Task Given(Metadata m, OrderCreated v){
        // this will be used to rehydrate state of process-manager
        // So that when(SeatsReserved) you can adjust the response.
    }
    // Optional 2
    private async Task Given(Metadata m, CommandEnqueued<MakeReservation> e){
        // same here.
    }
}
```

EXPERIMENTAL Uniqueness support

Uniqueness support in EventSourcing is not out-of-the-box, especially in regards to EventStoreDB. You can use some "hacks" but at the end of the day, you want uniqueness to be enforced by some kind of database. EventStoreDB is not designed for that purpose.

However, you can leverage typical reservation patterns. At the moment the library supports only the first option:

- At domain-layer, a domain-service usually would enforce uniqueness. This commonly requires a round-trip to a database. So just before actual event(s) are saved in a stream, a check against uniqueness constraints should be evaluated - thus reservation is made. When the event is appended to the stream, a confirmation is done automatically (on db).
- At a app-layer, command-handler would typically reserve a name. And when aggregate, which is being executed by the handler, saves its events successfully, then the reservation is confirmed. If the handler fails, then the reservation is deleted. Seems simple? Under the hood, it is not that simple, because what if the process is terminated while the command-handler is executing? We need to make sure, that we can recover successfully from this situation.

Let's see the API proposal:

```
// Let's define unique-category name
record FooCategory;
```

```
public class FooCreated
    // and apply it to one fo the columns.
    [Unique<FooCategory>]
    public string? Name { get; set; }

    // other stuff
}
```

For complex types, we need more flexibility.

```
// Let's define unique-category name, this will be mapped to columns in db
// If you'd opt for domain-layer enforcment, you need to change commands to events.
record BooCategory(string Name, string OtherName) : IUniqueFrom<BooCategory, BooCreated>,
IUniqueFrom<BooCategory, BooChanged>
{
    public static BooCategory From(BooCreated x) => new(x.InitialName, x.OtherName);
    public static BooCategory From(BooChanged x) => new(x.NewName, x.OtherName);
}

[Unique<BooCategory>]
public record BooCreated(string InitialName, string OtherName);

[Unique<BooCategory>]
public record BooChanged(string NewName, string OtherName);
```