Quick intro to Eventuous
- Allows us to "easily" getting started with Event Sourcing

Main "code blocks" for wiring stuff together to be able to store an event in EventStoreDB with Eventuous.

Program.cs
- Register EventTypes (such ass OrderAdded, OrderBooked)
- Register Eventuous configuration : builder.Services.AddEventuous(builder.Config)

Register.cs AddEventuous()
- -- configures Eventstore Database and Client)
- -- registers Command Handlers ; AddApplicationService(OrderCommandService)

OrderCommandService : (Inherriting ApplicationService<Order, OrderState, OrderId>)
- is hooking up our command handlers to the incoming commands (coming from the API "CommandsApi" : inherriting CommandHttpApiBase<Order>)

The OrderCommandService then register Handlers for the commands given:
handler registration methods: OnNew (you are adding something new), OnExisting(you are altering something), OnAny(you are either adding or altering something) - handlers can be Async.

- The handler registration connects the implementation with the command, f ex "OrderCommands.AddOrder" > "order.AddOrder" : where order is the Order.cs from 'OrdersCommandService : ApplicationService<Order, OrderState, OrderId>'

Order.cs (inherriting Aggregate<OrderState, OrderId>
- .AddOrder(...) then does validation, on the aggregate version and your business validations if any, before creating the Event in "Aggregate.Apply"

API gets a HttpRequest with command

[hostaddress] /order/add

OrderCommands.AddOrder

API calls ApplicationService.Handle (via CommandHttpApiBase) where ApplicationService is our registered OrdersCommandService

OrdersCommandService

OnNewAsync<**OrderCommands.AddOrder**>

(order, cmd, _) => **order.AddOrder(...)**

ApplicationService.Handle

- validates a Handler exists (registered in OrderCommandService)

- loads the Aggregate with all events from Store (if not new)

ApplicationService.Handle

runs the registerd Handler registeredHandler.Handler(..)
- triggering the Apply in Order.AddOrder

ApplicationService.Handle

appends the event to OrderState
(stores the event to Store)

returns the aggregate State and changes

OrderState is always built up from all events in history (from the event stream of this Aggregates id)

CommandApi.cs

```
using Eventuous;
using Eventuous.AspNetCore.Web;
using Microsoft.AspNetCore.Mvc;
using Orders.Application;
using Orders.Domain.Orders;

namespace Orders.HttpApi.Orders
{
    /// <summary>
    /// this command api operates on a (one) Order,
    /// </summary>
    [Route("/order")]
    public class CommandApi : CommandHttpApiBase<Order>
    {
        public CommandApi(ApplicationService<Order> service)
        :base(service)
        { }

        [HttpPost]
        [Route("add")]
        public Task<ActionResult<Result>> AddOrder(
            [FromBody] OrderCommands.AddOrder cmd,
            CancellationToken cancellationToken)
        {
            try
            {
                return Handle(cmd, cancellationToken);

            }
            catch (Exception e)
            {
                Console.WriteLine(e);
                throw;
            }
        }


        [HttpPost]
        [Route("book")]
        public Task<ActionResult<Result>> BookOrder(
            [FromBody] OrderCommands.BookOrder cmd,
            CancellationToken cancellationToken)
            => Handle(cmd, cancellationToken);

    }
}
```

OrdersCommandService.cs

```
namespace Orders.Application
{
    /// <summary>
    /// OrdersCommandService is registered in <see cref="Registrations"/> <br/>
    /// All order commands are executed via CommandHttpApiBase.Handle in the <see cref="CommandApi"/>
    /// </summary>
    public class OrdersCommandService : ApplicationService<Order, OrderState, OrderId>
    {
        public OrdersCommandService(IAggregateStore store) : base(store)
        {
            /* A note on the params to Async methods.
             * the third param '_' is needed if we don't pass anything
             * since ActOnAggregateAsync takes 3 params
             * (cancellationToken is the last param)
             */

            OnNewAsync<OrderCommands.AddOrder>(
                (order, cmd, _) => order.AddOrder(
                    new OrderId(cmd.OrderId),
                    cmd.CustomerId,
                    DateTimeOffset.Now
                    )
                );

            /* OnExisting operates on the Id to get the existing order,
             * then the "change" uses the State.Id internally
             */

            OnExistingAsync<OrderCommands.BookOrder>(
                cmd => new OrderId(cmd.OrderId),
                (order, cmd, _) => order.BookOrder(
                    new Money(cmd.OrderPrice, cmd.Currency),
                    new Money(cmd.PrePaidAmount, cmd.Currency),
                    new Money(cmd.DiscountAmount, cmd.Currency),
                    DateTimeOffset.Now
                    )
                );

            //... //
        }
    }
```

Order.cs

```
namespace Orders.Domain.Orders
{

    /// <summary>
    /// The Aggregate class is used for validating the commands
    /// from the CommandService (i.e. CommandHandler) <br/>
    /// (CommandService is found in Orders.Application)
    /// <br />
    /// Apply then Applies a new event to the state (if succesfull; this means the event is also saved to EventStoreDB,
    /// adds the event to the list of pending changes,
    /// and increases the current version.
    /// </summary>
    public class Order : Aggregate<OrderState, OrderId>
    {
        public async Task AddOrder(OrderId orderId, string customerId, DateTimeOffset orderCreatedAt)
        {
            //from the Eventuous base class 'Aggregate'
            //checks that the CurrentVersion is not set (i.e. Aggregate exists)
            //will throw a 'DomainException' if it is
            EnsureDoesntExist();

            /* other business validations ... */

            //from the Eventuous base class 'Aggregate'
            // will AddChange to internal list of pending changes of this Aggregate,
            // bump the Current version, and return (previousState, newState)
            Apply(new OrderEvents.V1.OrderAdded(
                    orderId,
                    customerId,
                    orderCreatedAt
                )
            );
        }

        public async Task BookOrder(
            Money price,
            Money prepaid,
            Money discount,
            DateTimeOffset orderBookedAt
            )
        {
            //from the Eventuous base class 'Aggregate'
            //checks that the CurrentVersion is set (meaning the Order aggregate exists in store),
            //will throw a 'DomainException' if it does Not
            EnsureExists();

            //Note: since price and prepaid are records, the '-' operator is a method in Money,
            // containing business logic, i.e. validaion: (you can't subtract on different currencies)
            var outstanding = price - prepaid;


            if (State.Booked)
            {
                throw new DomainException("Order is already Booked");
            }

            /* other business validations ... */

            //from the Eventuous base class 'Aggregate'
            // will AddChange to internal list of pending changes,
            // bump the Current version, and return (previousState, newState)
            Apply(new OrderEvents.V1.OrderBooked(
                State.Id,
                orderBookedAt,
                price.Amount,
                prepaid.Amount,
                discount.Amount,
                outstanding.Amount,
                price.Currency
            ));

        }
    }
}
```

OrderState.cs

```
using System.Collections.Immutable;
using Eventuous;

namespace Orders.Domain.Orders;


/// <summary>
/// OrderState is used to build the aggregate state from events <br/>
/// record, instead of class, ensures Imutability. it is always 'newed'. <br/>
/// Important note:
/// No business-logic-validations or altering of a state is "possible",<br/>
/// The current state is the truth from the events in history.<br/>
/// Business validations are made in the Aggregate (<see cref="Order"/>.cs)<br/>
/// Before Applying the event (creating/adding the changes of this states history)
/// </summary>
public record OrderState : AggregateState<OrderState, OrderId>
{

    /*
     * NOTE: setting an Id here will override the inherited AggregateState Id
     * and the following error will be thrown during Apply(...). In the Order class in this case. (inheriting Aggregate)
     * "errorMessage": "Aggregate id AggregateId cannot have an empty value",
     *
     * It is hard to find /catch the error (because it runs on another thread?)
     * I could not catch it with a try-catch block around
     *
     */
    //public OrderId Id { get; set; }
    public string CustomerId { get; set; }

    public Money Price { get; init; }
    public Money Outstanding { get; init; }
    public Money Discount { get; init; }
    public bool Paid { get; init; }


    /* a note on DateTimeOffset vs DateTime
     * The DateTimeOffset structure represents a date and time value,
     * together with an offset that indicates how much that value differs from UTC.
     * Thus, the value always unambiguously identifies a single point in time.
     * The DateTimeOffset type includes all of the functionality of the DateTime type
     * along with time zone awareness.
     */

    /// <summary>
    /// when initial order was created
    /// perhaps not needed? if using date from EventStore TimeStamp
    /// </summary>
    public DateTimeOffset OrderCreatedDate { get; set; }

    /// <summary>
    /// When order was submitted (as finished/book) from customer
    /// perhaps not needed? if using date from EventStore TimeStamp
    /// </summary>
    public DateTimeOffset OrderBookedDate { get; set; }
    /// <summary>
    /// if not booked, it is still open for changes (like a in a customer shopping-cart)
    /// </summary>
    public bool Booked { get; init; }

    public ImmutableList<PaymentRecord> PaymentRecords { get; init; } = ImmutableList<PaymentRecord>.Empty;

    /// <summary>
    /// ensure only unique payments.
    /// </summary>
    /// <param name="paymentId"></param>
    /// <returns></returns>
    internal bool HasPaymentBeenRecorded(string paymentId) => PaymentRecords.Any(x => x.PaymentId == paymentId);

//... _> continued on next "page"
```

```
// OrderState.cs
//... <-- continued from previous "page"

/// <summary>
    /// note: the state will run through all events
    /// when fetching the order from EventStoreDB,
    /// The events are stored by the aggregate Order, in the Apply methods
    /// </summary>
    public OrderState()
    {
        //from order.AddOrder
        On<OrderEvents.V1.OrderAdded>(HandleAdded);

        //from order.BookOrder
        On<OrderEvents.V1.OrderBooked>(HandleBooked);

        //from order.RecordPayment
        On<OrderEvents.V1.PaymentRecorded>(HandlePayment);

        //from order.RecordPayment / MarkFullyPaidIfNecessary
        On<OrderEvents.V1.OrderFullyPaid>((state, paid) =>
            state with { Paid = true });
    }


    /* a note on records "with"
     * 'record with' will "modify" the props set here,
     * and use the current props as is from current values
     * , in our case, our OrderState 'state'
     * and return a new record. (records are immutable)
     */

    private OrderState HandleAdded(OrderState state, OrderEvents.V1.OrderAdded added)
    {
        return state with
        {
            Id = new OrderId(added.OrderId),
            CustomerId = added.CustomerId,
            OrderCreatedDate = added.OrderCreatedDate
        };
    }

    private OrderState HandleBooked(OrderState state, OrderEvents.V1.OrderBooked booked)
    {
        return state with
        {
            Booked = true, //since this a Handler reading Events we know it is booked.
            Price = new Money(booked.OrderPrice, booked.Currency),
            Outstanding = new Money(booked.OutstandingAmount, booked.Currency),
            Discount = new Money(booked.DiscountAmount, booked.Currency),
            OrderBookedDate = booked.OrderBookedDate,
        };
    }

    private OrderState HandlePayment(OrderState state, OrderEvents.V1.PaymentRecorded payment)
    {
        return state with
        {
            Outstanding = new Money(payment.Outstanding, payment.Currency),
            PaymentRecords = state.PaymentRecords.Add(
                new PaymentRecord(payment.PaymentId, new Money(payment.PaidAmount, payment.Currency))
                )
        };
    }
}

public record PaymentRecord(string PaymentId, Money PaidAmount);
```