Compensating actions in the state machine

Script start

We previously talked about how compensating transactions in sagas helps us rollback operations that could not be completed across multiple microservices. Let's see where in our current purchase saga we might need to introduce compensation.

1. Start Inventory, Identity and Trading services

In Postman

2. Confirm how many items the user has

3. Try to purchase more items than the user can afford

4. Get the state machine state

5. Notice the state remains in **ItemsGranted**

6. Notice the VS Code console error

In Trading repo

7. Update PurchaseStateMachine:

```
public class PurchaseStateMachine : MassTransitStateMachine<PurchaseState>
{
   …
   public Event<GilDebited> GilDebited { get; }
   public Event<Fault<GrantItems>> GrantItemsFaulted { get; }
   public Event<Fault<DebitGil>> DebitGilFaulted { get; }

   public PurchaseStateMachine()
   {
      InstanceState(state => state.CurrentState);
      ConfigureEvents();
      ConfigureInitialState();
      ConfigureAny();
      ConfigureAccepted();
      ConfigureItemsGranted();
      ConfigureCompleted();
```

```
        ConfigureFaulted();
    }

    private void ConfigureEvents()
    {
        …
        Event(() => GilDebited);
        Event(() => GrantItemsFaulted, x => x.CorrelateById(context =>
context.Message.Message.CorrelationId));
        Event(() => DebitGilFaulted, x => x.CorrelateById(context =>
context.Message.Message.CorrelationId));
    }

    …

    private void ConfigureAccepted()
    {
        During(Accepted,
            …
            When(InventoryItemsGranted)
                …
                .TransitionTo(ItemsGranted),
            When(GrantItemsFaulted)
                .Then(context =>
                {
                    context.Instance.ErrorMessage = context.Data.Exceptions[0].Message;
                    context.Instance.LastUpdated = DateTimeOffset.UtcNow;
                })
                .TransitionTo(Faulted)
        );
    }

    private void ConfigureItemsGranted()
    {
        During(ItemsGranted,
            …
            When(GilDebited)
                …
                .TransitionTo(Completed),
            When(DebitGilFaulted)
                .Send(context => new SubstractItems(
                    context.Instance.UserId,
                    context.Instance.ItemId,
                    context.Instance.Quantity,
```

```
            context.Instance.CorrelationId))
          .Then(context =>
          {
            context.Instance.ErrorMessage = context.Data.Exceptions[0].Message;
            context.Instance.LastUpdated = DateTimeOffset.UtcNow;
          })
          .TransitionTo(Faulted)
        );
  }

  …

  private void ConfigureFaulted()
  {
    During(Faulted,
      Ignore(PurchaseRequested),
      Ignore(InventoryItemsGranted),
      Ignore(GilDebited));
  }
}
```

8. Update appsettings.json:

```
 "QueueSettings": {
  …
  "DebitGilQueueAddress": "queue:identity-debit-gil",
  "SubstractItemsQueueAddress": "queue:inventory-substract-items"
 },
```

9. Update QueueSettings:

```
public class QueueSettings
{
  …
  public string DebitGilQueueAddress { get; init; }
  public string SubstractItemsQueueAddress { get; init; }
}
```

10. Update Startup:

```
private void AddMassTransit(IServiceCollection services)
{
  …
  EndpointConvention.Map<DebitGil>(new Uri(queueSettings.DebitGilQueueAddress));
```

```
    EndpointConvention.Map<SubstractItems>(new Uri(queueSettings.SubstractItemsQueueAddress));
    …
}
```

11. Start Trading service

In Postman

12. Confirm how many items the user has

13. Try the Purchase again

14. Get the state machine state

15. Confirm the state is **Faulted** and check the **reason**.

16. Confirm the user does not have the additional items

However, compensation is not the only issue we need to deal with in our purchase saga. In the next lesson we will talk about idempotency and why it is so important in the context of microservices.