why event sourcing

the main concept behind the event sourcing is capture the state of the application.

event sourcing is the one of the best way to automatically update the state and publish an event.

creating event source entity

here we will crete account entity. basically this entity acts as the our use-case to demonstrate event sourcing.

```
@Aggregate
public class AccountAggregate {

    @AggregateIdentifier
    private String id;

    private double accountBalance;

    private String currency;

    private String status;
}
```

@Aggregate annotation tells Axon that this entity will be managed by axon basically similar to the @Entity annotation available with jpa.

@AggregateIdentifier annotation is used for the identifing a particular instance of aggregates.

modelling the commands and events

Axon works on the concept of the command and events. Commands are the user initiated actions that can change the state of the aggregate.

itmeans commands is used to change the state of the aggregate.

Events are the actual changing of state.

Commands:

1. Create Account

2. Credit Money
3. debit Money

According to the command event fired.

these fired eventsts are the

- Account Created Event
- Money Created Event
- Money Debited Event

Base Commands and Base Events

base command snippet

```
public class BaseCommand<T> {

    @TargetAggregateIdentifier
    public final T id;

    public BaseCommand(T id) {
        this.id = id;
    }
}
```

BaseEvent Snippet

```
public class BaseEvent<T> {

    public final T id;

    public BaseEvent(T id) {
        this.id = id;
    }
}
```

we have java classes Generics here Basically this makes our id field flexible across different classes that extend these classes.

however the most impo to note here is the @TargetAggregateIdentifier annotation.

Basically this is an axon specific requirement to identifies the aggregate instance in other words this annotation is required to determine the instance of the aggregate that should handle the command.

the annotation is placed on either the field or the gettter method. in this example we choose to put in the field.

create account command

```
public class CreateAccountCommand extends BaseCommand<String> {

    public final double accountBalance;

    public final String currency;

    public CreateAccountCommand(String id, double accountBalance, String currency) {
        super(id);
        this.accountBalance = accountBalance;
        this.currency = currency;
    }
}
```

Credit Money Command

```
public class CreditMoneyCommand extends BaseCommand<String> {

    public final double creditAmount;

    public final String currency;

    public CreditMoneyCommand(String id, double creditAmount, String currency) {
        super(id);
        this.creditAmount = creditAmount;
        this.currency = currency;
    }
}
```

Debit Money Command

```
public class DebitMoneyCommand extends BaseCommand<String> {

    public final double debitAmount;

    public final String currency;

    public DebitMoneyCommand(String id, double debitAmount, String currency) {
        super(id);
        this.debitAmount = debitAmount;
        this.currency = currency;
    }
}
```

Next step is to implements the events

Account Created Event

```
public class AccountCreatedEvent extends BaseEvent<String> {

    public final double accountBalance;

    public final String currency;

    public AccountCreatedEvent(String id, double accountBalance, String currency) {
        super(id);
        this.accountBalance = accountBalance;
        this.currency = currency;
    }
}
```

Money Credited Event

```
public class MoneyCreditedEvent extends BaseEvent<String> {

    public final double creditAmount;

    public final String currency;

    public MoneyCreditedEvent(String id, double creditAmount, String currency) {
        super(id);
        this.creditAmount = creditAmount;
        this.currency = currency;
    }
}
```

Money Debited Event

```
public class MoneyDebitedEvent extends BaseEvent<String> {

    public final double debitAmount;

    public final String currency;

    public MoneyDebitedEvent(String id, double debitAmount, String currency) {
        super(id);
        this.debitAmount = debitAmount;
        this.currency = currency;
    }
}
```

Account Activated Event

```
public class AccountActivatedEvent extends BaseEvent<String> {

    public final Status status;

    public AccountActivatedEvent(String id, Status status) {
        super(id);
        this.status = status;
    }
}
```

Account Held Event

```
public class AccountHeldEvent extends BaseEvent<String> {

    public final Status status;

    public AccountHeldEvent(String id, Status status) {
        super(id);
        this.status = status;
    }
}
```

we are successfully created base commands commands and events but then we have to implement the command handler and the event handler.

Q. what is the command and event handlers ?
handlers are the methods on aggregate that should be implemented on invoking particular commands and event.

Due to their relation in to the aggregate it is recommended to define the handlers to the in the Aggregate class itself also the command habdlers often
to access the state of aggregate.

in our case we will define them in the accountaggregate class see below to Account Aggregate class implementation.

```
Aggregate
public class AccountAggregate {

    @AggregateIdentifier
    private String id;

    private double accountBalance;

    private String currency;

    private String status;

    public AccountAggregate() {
    }

    @CommandHandler
    public AccountAggregate(CreateAccountCommand createAccountCommand){
        AggregateLifecycle.apply(new AccountCreatedEvent(createAccountCommand.id, create
    }

    @EventSourcingHandler
    protected void on(AccountCreatedEvent accountCreatedEvent){
        this.id = accountCreatedEvent.id;
        this.accountBalance = accountCreatedEvent.accountBalance;
        this.currency = accountCreatedEvent.currency;
        this.status = String.valueOf(Status.CREATED);

        AggregateLifecycle.apply(new AccountActivatedEvent(this.id, Status.ACTIVATED));
    }

    @EventSourcingHandler
    protected void on(AccountActivatedEvent accountActivatedEvent){
        this.status = String.valueOf(accountActivatedEvent.status);
    }

    @CommandHandler
    protected void on(CreditMoneyCommand creditMoneyCommand){
        AggregateLifecycle.apply(new MoneyCreditedEvent(creditMoneyCommand.id, creditMor
    }

    @EventSourcingHandler
    protected void on(MoneyCreditedEvent moneyCreditedEvent){

        if (this.accountBalance < 0 & (this.accountBalance + moneyCreditedEvent.creditAm
            AggregateLifecycle.apply(new AccountActivatedEvent(this.id, Status.ACTIVATED
        }

        this.accountBalance += moneyCreditedEvent.creditAmount;
    }

    @CommandHandler
```

```
    protected void on(DebitMoneyCommand debitMoneyCommand){
        AggregateLifecycle.apply(new MoneyDebitedEvent(debitMoneyCommand.id, debitMoneyC
    }

    @EventSourcingHandler
    protected void on(MoneyDebitedEvent moneyDebitedEvent){

        if (this.accountBalance >= 0 & (this.accountBalance - moneyDebitedEvent.debitAmc
            AggregateLifecycle.apply(new AccountHeldEvent(this.id, Status.HOLD));
        }

        this.accountBalance -= moneyDebitedEvent.debitAmount;

    }

    @EventSourcingHandler
    protected void on(AccountHeldEvent accountHeldEvent){
        this.status = String.valueOf(accountHeldEvent.status);
    }
}
```

as you can see we are handling the three commands in their own handler methods these handler methods should be annotated with the @CommandHandler Annotation.
there are the three command handler methods because there are three commands we want to handle.

NOTE:
handler methods use AggregateLifecyle.apply() method to register the events.

these events in turns are methods are handled by methods
annotated with @EventSourcingHandler Annotation.

also it is imperetative that all state changes in the event sourced aggregate should be performed in these methods.

another important point to keep in the mind is aggregate identifier must be set in the first method annotated with the Event sourcing handler

NOTE:

@EventSourcingHandler annotation is the responsible for the sorcing the event.

in our example this is the evident in the below method.

```
@EventSourcingHandler
protected void on(AccountCreatedEvent accountCreatedEvent){
        this.id = accountCreatedEvent.id;
        this.accountBalance = accountCreatedEvent.accountBalance;
        this.currency = accountCreatedEvent.currency;
        this.status = String.valueOf(Status.CREATED);

        AggregateLifecycle.apply(new AccountActivatedEvent(this.id, Status.ACTIVATED));
}
```