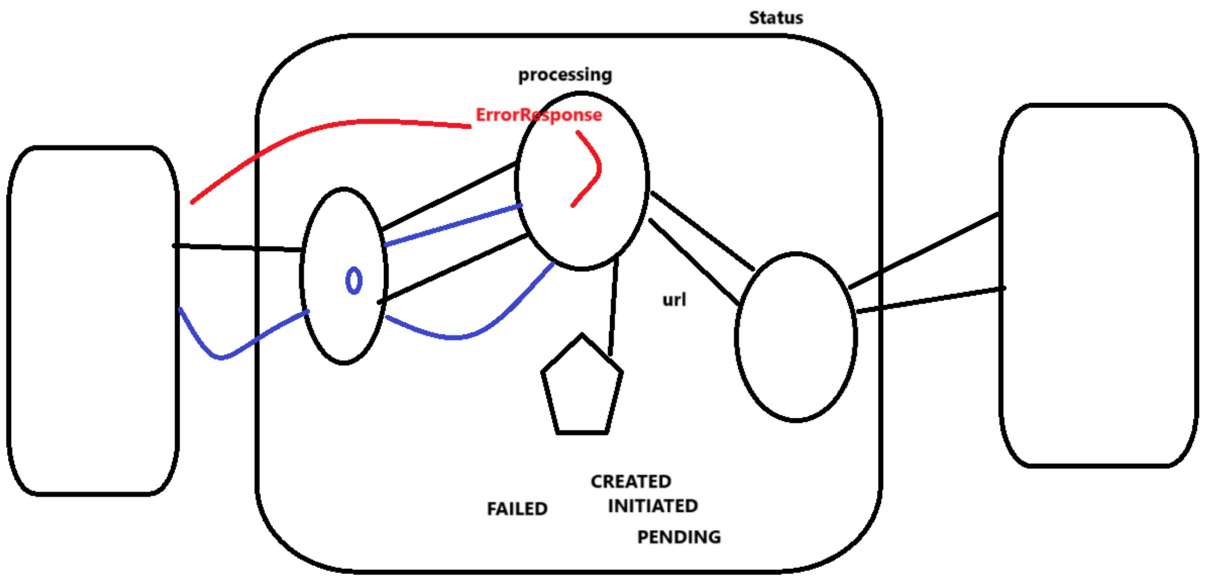
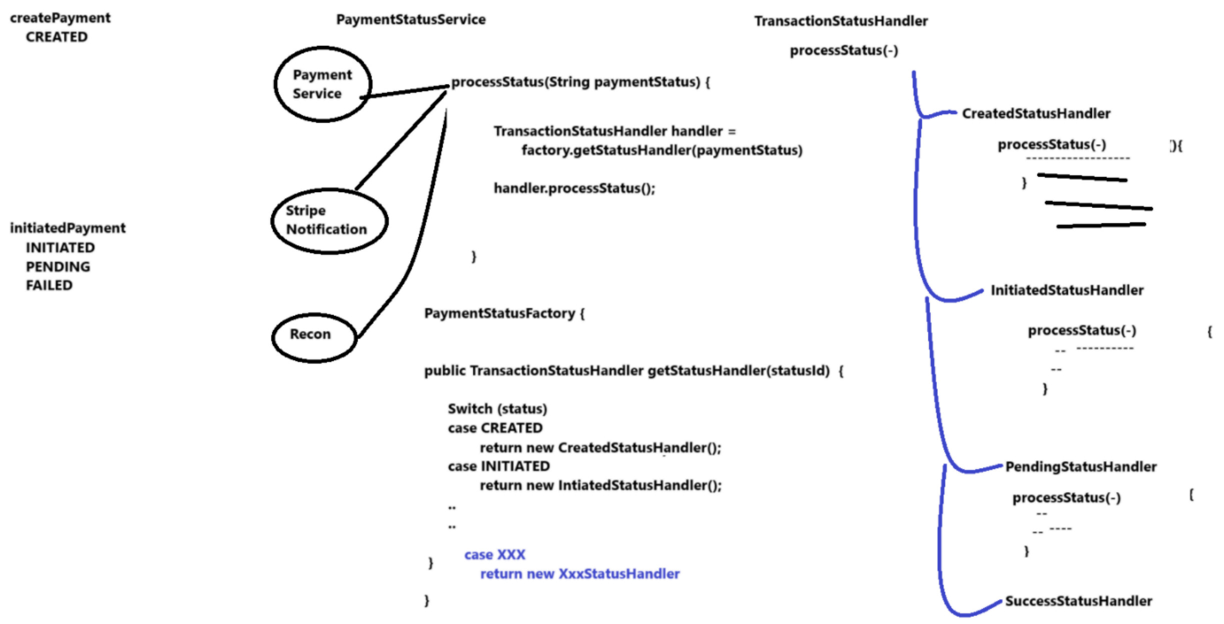
DIAGRAMS





```
processStatus(String paymentStatus) {

    TransactionStatusHandler handler =
        factory.getStatusHandler(paymentStatus)

    handler.processStatus();

}
```

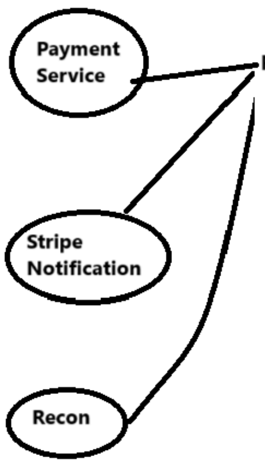
```
PaymentStatusFactory {

public TransactionStatusHandler getStatusHandler(statusId)  {

    Switch (status)
    case CREATED
        return new CreatedStatusHandler();
    case INITIATED
        return new IntiatedStatusHandler();
    ..
    ..
      case XXX
 }      return new XxxStatusHandler
}
```

```
CreatedStatusHandler
   processStatus(-)        ){
   -------------------
   }

InitiatedStatusHandler
   processStatus(-)        {
     --  ----------
     --
   }

PendingStatusHandler
   processStatus(-)        {
     --
     -- ----
   }

SuccessStatusHandler
```

PaymentService

```
createPayment
   CREATED
```

PaymentStatusService



```
processStatus(String paymentStatus) {

        TransactionStatusHandler handler =
             factory.getStatusHandler(paymentStatus)

        handler.processStatus();


    }
```

```
initiatedPayment
   INITIATED
   PENDING
   FAILED
```

```
PaymentStatusFactory {


public TransactionStatusHandler getStatusHandler(statusId)  {

     Switch (status)
     case CREATED
           return new CreatedStatusHandler();
     case INITIATED
           return new IntiatedStatusHandler();
     ..
     ..
          case XXX
 }           return new XxxStatusHandler

}
```

LIVE NOTES

Week5: Status Management system.

processing-service is responsible for core payment status management.
processing-service, exposes 2 APIs, 1 createPayment, 2nd to initiate payment.

CreatePayment:
    we save Payment in DB as CREATED status.

InitiatePayment
    we update status as INITIATED
    Make API call to stripe-provider-service to get response
    if we get valid url (success), then PENDING & return url to invoker.
    if we get failed response, then FAILED, throw exception with proper errorCode & errorMessage. & return standard error response to validation service.

RestAPI

resources & endpoint naming + HttpMethod
    => defines what functionality you will write.

1. create payment

POST /v1/payments

    create payment logic
    response: unique id

2. initiate payment

POST /v1/payments/{id}/initiate
    request body
    response
        url

txnReference - unique reference which is not predictable.

10
/v1/payments/10/initiate

/v1/payments/15/initiate

1. create post method to handle request
2. return ResponseEntity
3. CreatePaymentResponse should have txnReference
4. initiatePayment method also.
   @PostMapping("/{txnReference}/initiate")

5. Continue coding CreatePayment usecase.
   RequestStructure
   Controller => Service
   DAO save it in DB
   return response.

What should be request structure for both createPayment & initiatePayment

   - We should be able to successfully save Txn record in DB
   - For making stripe-provider api call we need some data.

in CreatePaymentRequest - expect fields needed to save txn in DB
In InitiatedPaymentRequest - expect fields needed to make API call to stripe-provider-service.

.pojo
   CreatePaymentRequest
.dto
   TransactionDTO
.entity

}

TransactionEntity

```
@Data
public class TransactionEntity {

    private int id;
    private int userId;
    private int paymentMethodId;
    private int providerId;
    private int paymentTypeId;
    private int txnStatusId;
    private double amount;
    private String currency;
    private String merchantTxnReference;
    private String txnReference;
    private String providerReference;
    private String errorCode;
    private String errorMessage;
    private int retryCount;

}
```

- we created 3 java objects, which we will need during Create Payment activity.
- finalize the request structure of CreatePaymentRequest
    do we need all the fields to be passed by validationservice to us, during createPayment API call.

```
@Data
public class CreatePaymentRequest {

    private int userId;
    private int paymentMethodId;
    private int providerId;
    private int paymentTypeId;

    private double amount;
    private String currency;
    private String merchantTxnReference;

}
```

=====
```
{
   "userId": 123,

   "paymentMethodId": 1,
   "providerId": 1,
   "paymentTypeId": 1,

   "amount": 100.50,
   "currency": "USD",

   "merchantTxnReference": "MERCHANT_ABC123"
}
```

---
to validation service, we are expecting the values as ID which is numeric paymentMethodId, providerid, paymentTypeId. all are int representing that value. We need to still understand which number means what & code accordingly in system.
=> so instead of having them is ID, if we make them as string
```
            "paymentMethod": "APM"
            "provider": "STRIPE"
```

we have these IDs in DTO and also Entity??
    we will not change for TransactionEntity

Yes, for DTOs have it as String.

    both POJO & DTO should have it as String
    Entity should continue to have it as int value.

```
   private String paymentMethod;
   private String provider;
   private String paymentType;
```

Request JSON
```
{
   "userId": 123,

   "paymentMethod": "APM",
   "provider": "STRIPE",
   "paymentType": "SALE",

   "amount": 100.50,
   "currency": "USD",

   "merchantTxnReference": "MERCHANT_ABC123"
}
```

TransactionDTO updated to hold strings.

in controller:

    convert incoming pojo to dto
    ModelMapper

//TODO

```
log.info("Controller||txnReference:" + createPaymentResponse.getTxnReference() +""
                                        + "|response:" + response);

log.info("Controller||txnReference:{}|response:{}",
createPaymentResponse.getTxnReference(),
                                        response);
```

for createPayment service layer.

    we need to add another DAO layer classes & save txn in DB
    call that DAO layer from service layer.

12:10

5 status

CREATED

    - when valid txns is submitted in payment.

INITIATED

    - before calling stripe-provider

PENDING

    - got url & returning that

SUCCESS

    - got stripe notification

FAILED
- createpayment call failed at stripe-provider
- notification
- recon, update

in different situations, we need to update different statuses.

1. anywere in code, where you feel the need to update status, you diretly write code there.

2. Write 1 method where all status handling logic will be.
    Anywhere in whole project, if you want to change status, then you call this method.

3. Where to write this central method
    1. PaymentService (you would need to expose other methods of Paymentservice even to StripeNotification & Recon)
    2. Status is core business logic of Payment processing.
        Payment Status Management.
        PaymentStatusService
            processStatus()

4. Instead of writing logic for each of the status, in this central method. We can refactor it into separate methods. ANd invoke that separate method form the core central method.

5. Instead of writing status-specific business logic in individual methods, how about we write it into separate java class. & create object of these java classes PaymentStatusService & invoke its business methods.

6. Instead of leaving all java classes independent, we can see a pattern, that each java classes is simply processing 1 specific STATUS. We can have a common parent, and all status handlers wil implement this common parent, and override the standard defined method(with specific parameters). Even the PaymentStatusService logic of invocation will improve.

7. Factory Design Pattern & invoking it via PaymentStatusService

```
TransactionStatusHandler handler = factory.getStatusHandler(paymentStatus);
handler.processStatus(txnDTO)
```