**accounting:**

1-in entity \*->card relationship entityType Must be person/organization

2-bank defines fee for each card

3-account balance needs to be updated by database trigger each time new record inserted

4-invoice payee/payer entity need to be of entityType{person/org}

5-in entity/Card relation entity must be of entityType {person/org}

**Invoice and Services >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>**

1-all the services attached to invboice need to be in the same currency as invoice

2-all services added to the invoice need to the follow invoice\_payeeID =service\_giverID

***create Services***

entities including in service need to be of entityType{}

***create invoice***

invoiceAction

invoiceActionTransaction

***create service[s] and assign to invoice***

***payInvoice***

invoicePayment

invoiceAction

invoiceActionTransaction

**CARDS and BANKING >>>>>>>>>>>>>>>>>>>>>>>>>>>>>**

***create bank***

***assign card to the bank***

***create fee and assign to card***

**Test Scenario >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>**

**(1)Invoice and payments**

**1-reset database and seeds , 2-setup basic entities**

2-create initial lookups

3-create currency

4-create 2 persons

5-attach accounts to persons

6-Create 2 services

7-Creeat Invoice

8-assign Services to invoice

9-Finalize Invoice

10-Create Cards and assign to users

11-assign cards to persons

12-to partial payments for invoices

**(2)Payment Cancellation**

cancel payments w/o applying fees > everything will be reversed

Note: if fees need to be applied then transactions related to fees will be added.

**Scenario 1 plus cancel payments**

**Scenario[3]**

create card

create bank

create card Type

bank assign[Replace/New] fee to card Type

**OOP [programming structure]**

1-class implementation for each entity needs its

(1)inheritance Distinctor,

(2)properties [like DB table definition] and reference [ID or object reference] to other related objects

Note1: references can be fetched using Methods instead of saving for a long time

Note2:these references can be in form of list<EF objects> or data table using SP

(3)constructor(optional) for each inheritance level.

2-lower db connections can be EF or SP and each time need to reload the object

3-non-related sets of data can be fetched from static Methods

4-obviously there is a need for mapping between server-side class definition and data fetched from database using SP or EF

5-any Method call may change stat of object including new values for properties

concept : server side encapsulation is a shell

6-Method-return-type can be any kind of data structure EF objects list or dataset

7-define class props like database in hierarchical method and remove redundant props

8-use virtual/override technique to populate inherited tables

9-when using (ask yourself about polymorphism)virtual/override plus the fact that can call

BASE.method, if polymorphism happens overridden one will be effective and in any call to virtual method in base classes will bubble up.

10-by using protected/public let decide which inherited classes are allowed to have access to the base class. instead of opening that method from abase class [method access control in hierarchy].

11-sometimes just to need a simpler base class to get a shared info

**Design**

1-analysis and business rules

2-class and object diagram

3-database design and cascading flows

4-loop

development module by module and documentation in code

write test cases and update document

write SQL select result queries

build required views

end-loop

5-always put ways to raw object be loaded hierarchically by property-based attributes

Note:

1-payments can be done for any type of actions like deposit or invoice.

2-track of original payment transactions will be kept in paymentTransaction Table.

3-any status change happened to the payment of any kind, e\will be recorded into separate category like invoice/invoiceAction/invoiceActionTransaction or deposit/depositAction/depositActionTRansaction

4-payment cancellation happens thru invoice and invoice calls payment Cancellation and saves another log for recorded transactions and action happened.

5-in polymorphism if a method is defined virtual-override, even if method is called in base class, will bubble up to inherited class. which is fine but tricky

**lookup types:**

glType

catType

cardtype

ccCardType

paymentType

extPaymentType

invoiceStat

currencyType

entityType

contectType

officeType

userType

sysUserType

1-

service issuer = invoice issuer = payee

service receiver = invoice receiver = payer

2-

after any operation , total balance of account table remains ZERO

3-

payment-action & payment-Action-Transaction tables can also be effected by invoice or deposit operations.

*Restrictions*:

1-payer can pay more than what exists in wallet money

2-there is no fee for internal payment

3-only fee for [bank/db-card] and [bank/cc-Card-Type] and

4-if [bank/cc-Card-Type] is defined then overrides [bank/card-Type]

5-fee and cc-Fee need to be unique per bank-card-Type for now

**Enumeration:**

**entityType {**organization=1,office=2,person=3**}**

**officeType {**temporaryOffice=1,headOffice=2,bankBranch=3**}**

**userType {** appUser=1,sysUser=2**}**

**sysUserType{**normalsysUser=1,adminSysUser=2**}**

**paymentType{**external=1,internal=2**}**

**extPaymentType{**creditPayment=1,interacPayment=2**}**

**ccCardType{**MC=1,Visa=2**}**

**cardType{**DebitCard=1,CreditCard=2**}**

**invoiceStat{**Generated=1,Refunded=2,Voided=2**}**

**catType {}**

**glType {**Asset=1,OE=2,Lib=3**}**

**currencyType{** real=1,unReal=2**}**