Documentation - Data Bases 2

Marco Fasanella
MsC Computer Science, Polimi
C.P. 10617541

Nicola Dean
MsC Computer Science, Polimi
C.P. 10674826

This document describes design and implementation of Data Base 2 Project

1. Specifications

A telco company offers pre-paid online services to web users. Two client applications using the same database have been developed: a costumer application and an employee application.

1.1 Extra hypotesis

A Service Package can have more than one Services of the same type (i.e. two fixed phones or different kind of mobile phones).

2. Diagrams and Schemas

2.1 SQL DDL

Here is the SQL DDL schema of the Database

```
Users(id,username,password,email,type,insolvent)
FailedPayment(userId,orderId,failDate)
Orders(id,creationDate,userId,packageId,rate Id,startDate,totalPayment,status)
Rate_costs(id,monthValidity,cost,packageId)
Packages(id,name)
Services(id,packageId,DTYPE)
Package_OptionalProducts(packageId,productId)
OptionalProducts(id,name,monthlyFee)
Orders_OptionalProducts(orderId,productId)
```

Figure 1. SQL DDL

Then Services references to three different tables depending on the type of offer:

```
mobile_internet_services(id,gigabyte,extraFee)
mobile_phone_services(id,minutes,sms,extraMinutesFee,extraSMSFee)
fixed_internet_services(id,gigabyte,extraFee)
```

Figure 2. SQL DDL Services Details

2.2 ER

The Entity-Relation Diagram of the Database

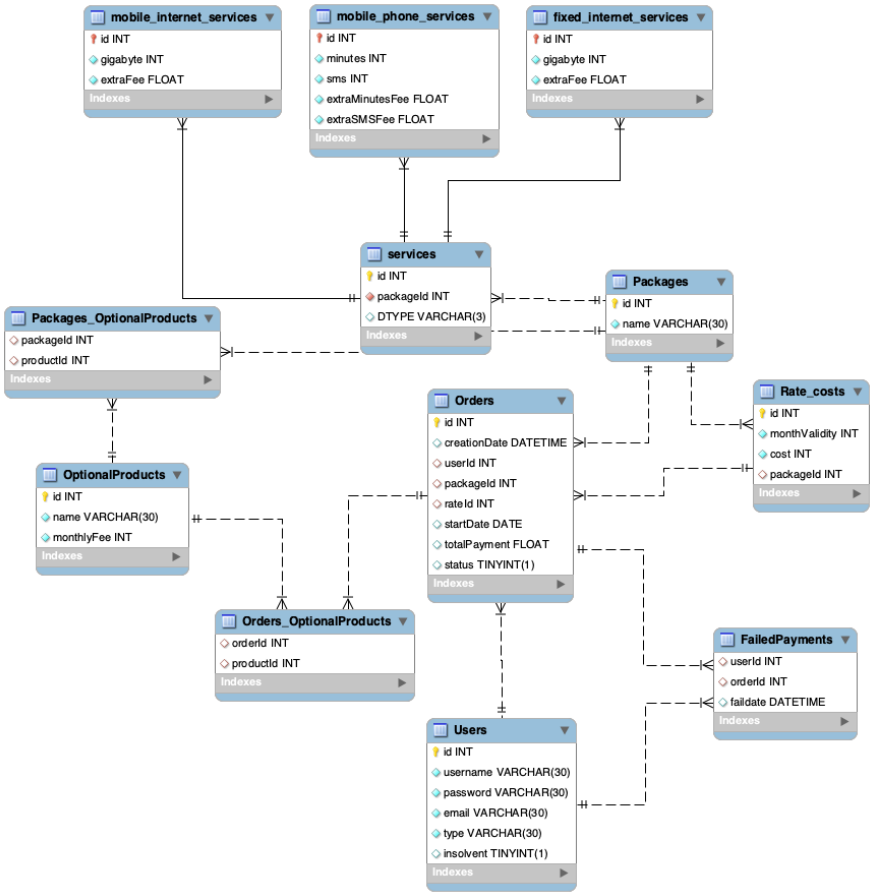

Figure 3. ER Diagram

3. SQL Description

Detailed description of SQL code used in the project.

3.1 Views

The following views are used for various Sales Reports.

Joining Orders and Rate_costs tables we performed a *selection* for each distinct package (depending on their month validity) and a *count* for their occurrences.

```
create view PurchasesCount as (
       Select Packages.name as name,Rate_costs.monthValidity as validity, count(*) as count
       from Orders as o join Packages  join Rate_costs
       on o.packageId=Packages.id and Rate_costs.packageId=o.packageId
       and o.rateId=Rate_costs.id
       group by o.packageId, Rate_costs.id
           );
```

Then for a less detailed view, from *PurchasesCount* , a *group by* on the name of the package gives a count of each one.

```
create view PurchasesCountGrouped as (
       select p.name as name, sum(p.count) as count
       from PurchasesCount as p
       group by p.name
           );
```

To count the Optional Products was necessary a *left join* because it could exist a Package without any Optional Product, and as a consequence it wouldn't be stored in *Orders_OptionalProducts* table.

```
create view OptionalProductsCount as(
       select o.packageId as packageId, count(opt.productId) as optcount
       from Orders as o left join Orders_OptionalProducts as opt
       on o.id=opt.orderId
       group by o.id
           );
```

Then to have the average count of them, we performed an *avg* on the count of previous view.

```
create view OptionalProductsAverage as(
       select p.name as name, avg(opc.optcount) as avg
       from OptionalProductsCount as opc join Packages as p
       where opc.packageId=p.id
       group by id
           );
```

Following view sums the *totalPayment* of each Order grouped by *packageId*

```
create view ValueOfTotalSales as(
       select p.name as name, sum(o.totalPayment) as totalPayment
       from Orders as o join Packages as p
       where p.id=o.packageId
       group by o.packageId
           );
```

Then in *OptionalProductsSales* the same method is used to get the totalCost of the Optional Products related to their corresponding Service Package in *Orderds*

```
create view OptionalProductsSales as(
     select p.name as name,sum(op.monthlyFee*r.monthValidity) as totalOptionalProductsSales
     from Orders_OptionalProducts as orderop join Orders as o
     join OptionalProducts as op join Packages as p join Rate_costs as r
     where o.id=orderop.orderId and orderop.productId=op.id
     and o.packageId=p.id and o.rateId=r.id
     group by p.id
          );
```

And finally these two views are *left joined* to have both *totalPayment* with and without Optional Products. A *left join* is required because as mentioned before it could exist a service package without Optional Product. In this case *totalPaymentWithoutOP* will return *null*.

```
create view ValueOfSalesDetailed as(
       select vot.name as name, vot.totalPayment as totalPayment,
               (vot.totalPayment-ops.totalOptionalProductsSales) as totalPaymentWithoutOP
       from ValueOfTotalSales as vot left join OptionalProductsSales as ops
       on vot.name=ops.name
         );
```

In *InsolventReport* are selected all Users with *having count(*)¿=3* of insolvent orders stored in *FailedPayments*.

```
create view InsolventReport as(
        select u.id as id,u.username as username,u.email as email,
              (
              select max(fp1.faildate)
              from FailedPayments as fp1
              where fp1.userId=u.id) as lastDate,
              (
              select o.totalPayment
              from Orders as o join FailedPayments as fp2
              where o.id=fp2.orderId and lastdate=fp2.faildate) as amount
        from FailedPayments as fp join Users as u
```

```
        where fp.userId=u.id
        group by u.id
        having count(*)>=3
           );
```

Last view *OptionalProductBestSeller* is used to have a count and the value of total sales of each Optional Product.

```
create view OptionalProductBestSeller as(
        select op.name as name,count(op.id) as amountSold,
               sum(op.monthlyFee*r.monthValidity) as value
        from Orders_OptionalProducts as ordop join OptionalProducts as op
        join Rate_costs as r join Orders as o
        where ordop.productId=op.id and o.rateId=r.id and o.id=ordop.orderId
        group by op.id
             );
```

The using two distinct queries we performed the *OptionalProductBestSeller-ForValue*

```
        select o
        from OptionalProductBestSeller o
        where o.value=(
        select max(o2.value)
        from OptionalProductBestSeller o2
        );
```

and *OptionalProductBestSellerForAmount*

```
          select o from OptionalProductBestSeller o
          where o.amountSold=(
          select max(o2.amountSold)
          from OptionalProductBestSeller o2
          );
```

3.2 Triggers

INSOLVENT_USER When a new order is created, if it's status is "false" then mark user as insolvent by changing a boolean into user table.

```
create trigger INSOLVENT_USER
    after insert on Orders
    for each row
begin
if ( new.status = false) then
update Users set Users.insolvent = true where Users.id = new.userId;
insert into FailedPayments (userId,orderId,faildate) values (new.userId,new.id,CURRENT_TIMESTA
end if;
END $$
```

INSOLVENT_USER_REMOVAL When an order is updated, if it's status is "true" then remove insolvance status from user.

```
create trigger INSOLVENT_USER_REMOVAL
    after update on Orders
    for each row
begin
if (new.status = true) AND -- user payed a suspended order i check if all his pending order ar
(select count(*) from Orders as o where o.userId=new.userId and o.status = false) = 0
then
update Users set Users.insolvent = false where Users.id = new.userId;
end if;
if (new.status = false AND old.status = new.status)  then
insert into FailedPayments (userId,orderId,faildate) values (new.userId,new.id,CURRENT_TIMESTA
end if;
END $$
```

NewPackage When a new package is created some view are initialized with a new row containing the package name and empty values for the statistic field (eg: AVG,totalPayments...)

```
    create trigger NewPackage
after insert on Packages
        for each row
        begin
-- Optional Average per package
insert into OptionalProductsAverage (name,avg) values (new.name ,0);
            -- Revenue per package
            insert into ValueOfSalesDetailed  (name,totalPayment,totalPaymentWithoutOP) values
        END $$
```

PurchaseCountInitialize When a new package is created are created also the rate cost, so as in the "NewPackage" trigger and for each new RateCost a new row with initial value is added to "PurchaseCount" view

```
create trigger PurchaseCountInitialize
after insert on Rate_costs
        for each row
        begin
declare pkgname Varchar(30);
set pkgname   := (select name from Packages where Packages.id=new.packageId);
insert into PurchasesCount (name,validity,count) values (pkgname,new.monthValidity,0);
        END$$
```

PurchasesCount_Population Whenever an Order is created and has a valid payment this table increase the number of time the package is payed

```
create trigger PurchasesCount_Population
after insert on Orders
    for each row
begin
declare pkgname Varchar(30);
    declare orderVal int;
    if new.status = true then
    set pkgname   := (select name from Packages where Packages.id=new.packageId);
    set orderVal  := (select monthValidity from Rate_costs where id=new.rateId);
 -- If this is the first order on package then set count = 1 and insert new line
            SET SQL_SAFE_UPDATES=0;
update PurchasesCount set PurchasesCount.count = PurchasesCount.count + 1
where name = pkgname and validity = orderVal;
            SET SQL_SAFE_UPDATES=1;
        end if;
END $$
```

PurchasesCount_Population_Update Same as PurchasesCount_Population but when Order is updated, check if payment is ok then update view.

```
create trigger  PurchasesCount_Population_Update
    after update on Orders
    for each row
begin
declare pkgname Varchar(30);
    declare orderVal int;
    if new.status = true then
set pkgname   := (select name from Packages where Packages.id=new.packageId);
set orderVal  := (select monthValidity from Rate_costs where id=new.rateId);
```

```
            SET SQL_SAFE_UPDATES=0;
update PurchasesCount set PurchasesCount.count = PurchasesCount.count + 1
where name = pkgname and validity = orderVal;
            SET SQL_SAFE_UPDATES=1;
        end if;
    END $$
```

AddNoOptionalOrderToAVG This trigger is a "support trigger" because is used to call another trigger. When an order is created add a "fake null optionalProduct" to it so that Order with no product are counted in the Average.(See next trigger)

```
create trigger AddNoOptionalOrderToAVG
after insert on Orders
    for each row
    begin
-- trigger OptionalProdAvg trigger
insert into Orders_OptionalProducts (orderId,productId) values (new.id,null);
delete from Orders_OptionalProducts where orderId = new.id;
    END $$
```

OptionalProdAvg When a new product is associated to a new order, after checking if payment is ok update the average of the specific product that is been added.

```
create trigger OptionalProdAvg
after insert on Orders_OptionalProducts
for each row
    begin
    declare pkgcount INT;
    declare pkgname Varchar(30);
    declare stat  bool;
    set stat    := (select status from Orders where id=new.orderId);

    if stat = 1 then
    set pkgname   := (select name from Packages where Packages.id=(select packageId from Order
set pkgcount  := (select count(*) from PurchasesCountGrouped group by name having name = pkgna
SET SQL_SAFE_UPDATES=0;
            if(new.productId is null) then
update OptionalProductsAverage set avg = ((avg*pkgcount))/(pkgcount+1) where name=pkgname;
            else
update OptionalProductsAverage set avg = ((avg*pkgcount)+1)/(pkgcount+1) where name=pkgname;
            end if;
            SET SQL_SAFE_UPDATES=1;
        end if;
    END $$
```

OptionalProdAvgOnUpdate When an order is updated, if payment is ok, this trigger calculate how many optional is associated with it and update the AVG of the package in consequence.

```
create trigger OptionalProdAvgOnUpdate
after update on Orders
    for each row
    begin
declare pkgcount INT;
    declare pkgname Varchar(30);
    declare prodCount INT;
    if new.status then
    set pkgname   := (select name from Packages where Packages.id=(select packageId from Orders
set pkgcount  := (select count(*) from PurchasesCountGrouped group by name having name = pkgname
    set prodCount := (select count(*) from Orders_OptionalProducts group by orderId having  ord
SET SQL_SAFE_UPDATES=0;
            if(prodCount is null) then
update OptionalProductsAverage set avg = ((avg*pkgcount))/(pkgcount+1) where name=pkgname;
            else
update OptionalProductsAverage set avg = ((avg*pkgcount)+prodCount)/(pkgcount+1) where name=pkgn
            end if;
            SET SQL_SAFE_UPDATES=1;
        end if;
    END $$
```

TotalPackageRevenue When a new order is created, if payment is ok update this view with totalRevenue of this package (see trigger PackageRevenueNoOptional to see how the OptionalRevenue is removed from totalRevenue).

```
create trigger TotalPackageRevenue
after insert on Orders
    for each row
    begin
    declare pkgname Varchar(20);
    declare validity int;
declare totalOfProducts int;
    if new.status = 1 then
set pkgname   := (select name from Packages where Packages.id=new.packageId);
set validity  := (select monthValidity from Rate_costs where id=new.rateId);
SET SQL_SAFE_UPDATES=0;
update ValueOfSalesDetailed set totalPayment =  totalPayment + new.totalPayment ,
                                                    totalPaymentWithoutOP = totalPaymentWithout
where name=pkgname;
SET SQL_SAFE_UPDATES=1;
        end if;
```

```
    END $$
```

PackageRevenueNoOptional For each product associated with order it query the revenue of it and remove it from the total revenue of the package.

```
-- remove optional revenue for each optional
create trigger PackageRevenueNoOptional
after insert on Orders_OptionalProducts
    for each row
    begin
    declare pkgId     INT;
declare pkgname Varchar(20);
declare price     INT;
    declare validity  INT;
declare stat  bool;
    set stat    := (select status from Orders where id=new.orderId);

    if stat = 1 then

         set pkgId     := (select packageId from Orders where id=new.orderId);
 set pkgname   := (select name from Packages where Packages.id=pkgId);
         set validity  := (select monthValidity from Rate_costs where id=(select rateId from O
 set price     := (select monthlyFee from OptionalProducts where id=new.productId)*validity;

         if(price is null)then
set price := 0;
         end if;
        SET SQL_SAFE_UPDATES=0;
        update ValueOfSalesDetailed set
totalPaymentWithoutOP = totalPaymentWithoutOP - price
where name=pkgname;
SET SQL_SAFE_UPDATES=1;

end if;
    END $$
```

TotalPackageRevenue_Update When Order is updated, check if payment is ok and as before it calculate optional prod revenue and remove it from the totalRevenue of this package.

```
create trigger TotalPackageRevenue_Update
after update on Orders
for each row
    begin
    declare pkgname Varchar(30);
```

```
declare revenueOptional   INT;
    declare validity  INT;
if new.status = 1 then
set pkgname   := (select name from Packages where Packages.id=new.packageId);
set validity  := (select monthValidity from Rate_costs where id=(select rateId from Orders where
set revenueOptional := (select sum(monthlyFee) from Orders_OptionalProducts join OptionalProduct

if(revenueOptional is null) then
set revenueOptional := 0;
end if;

SET SQL_SAFE_UPDATES=0;
update ValueOfSalesDetailed set totalPayment =  totalPayment + new.totalPayment ,
                                                    totalPaymentWithoutOP = totalPaymentWithout
where name=pkgname;
SET SQL_SAFE_UPDATES=1;
        end if;
    END$$
```

InitializeBestOptional Initialize the BestOptional view whenever a new optional product occur

```
create trigger InitializeBestOptional
after insert on OptionalProducts
    for each row
    begin
insert into OptionalProductBestSeller (name,amountSold,value) values (new.name,0,0);
    END$$
```

UpdateBestOptional Whenever a product is associated with an order this trigger update the view that contain statistic about best optional. It calculate the revenue for this new product and sum it to the total revenue of it It increase the counter of how many time this product have been bought.

```
create trigger UpdateBestOptional
    after insert on Orders_OptionalProducts
for each row
    begin
 declare prodName Varchar(30);
         declare price     INT;
         declare validity  INT;

         declare stat  bool;
set stat    := (select status from Orders where id=new.orderId);
```

```
if stat = 1 then
 set prodName  := (select name from OptionalProducts where id=new.productId);
         set validity  := (select monthValidity from Rate_costs where id=(select rateId from O
 set price     := (select monthlyFee from OptionalProducts where id=new.productId)*validity;

if(prodName is not null) then
 SET SQL_SAFE_UPDATES=0;
update OptionalProductBestSeller set value = value + price,
 amountSold = amountSold + 1
 where name=prodName;
SET SQL_SAFE_UPDATES=1;
 end if;
        end if;
    END$$
```

UpdateBestOptional_OnUpdate Same as UpdateBestOptional but with a unique query for all product associated with this order (more expensive but only way if the first payment fail)

```
    create trigger UpdateBestOptional_OnUpdate
after update on Orders
    for each row
begin
         declare revenue     INT;
         declare validity  INT;
 declare count  INT;
if new.status = 1 then
 SET SQL_SAFE_UPDATES=0;
update OptionalProductBestSeller as best join OptionalProducts as op
on best.name = op.name
set amountSold = amountSold + 1,
value = amountSold * op.monthlyFee
where op.name in (select op1.name from OptionalProducts as op1 join Orders_OptionalProducts as
on id = productId where ord.orderId =new.id);
SET SQL_SAFE_UPDATES=1;
end if;
END $$
```

UpdateInsolventUser When a user fail a payment this trigger count how many time is happened and if it happen more then 3 times it update (or insert) a line with the last failed payment (date,amount).

```
create trigger UpdateInsolventUser
```

```
    after insert on FailedPayments
for each row
    begin
declare failedCount INT;
        declare failedImport INT;
        declare usr Varchar(30);
        declare mail  Varchar(30);

        set failedCount := (select count(*) from FailedPayments  group by userId having userId =
        set failedImport := (select totalPayment from Orders where id=new.orderId);
        set usr := (select username from Users where id=new.userId);
        set mail    := (select email from Users where id=new.userId);

         -- SIGNAL SQLSTATE '02000' SET MESSAGE_TEXT =username ;
        if(failedCount >=3) then
if( (select id from InsolventReport where id=new.userId) is  null) then
insert into InsolventReport (id,username,email,lastDate,amount) values (new.userId,usr ,mail ,n
else
update InsolventReport set lastDate = new.faildate, amount = failedImport where id=new.userId;
end if;
        end if;
    END$$
```

4. ORM Description

4.1 Entities

Optional Product Entity related to all type of optional products with id and related name.

```
@Entity
@Table(name = "OptionalProducts", schema = "test")
public class OptionalProduct{

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    int    id;
    String name;
    int    monthlyFee;
```

Order The Order entity refers to Table *Orders*, with a boolean *status* that is true if the payment has been accepted from bank. The *bank* pseudo application works at follows: the first payment is completely pseudo-random (using *Random* java library). Then if the order is insolvent the purchase process is always accepted for demo purposes.

```
@Entity
@Table(name = "Orders", schema = "test")
@NamedQuery(name="Orders.Id" , query="select o from Order o where o.Id
    = :orderId")
@NamedQuery(name="Orders.All" , query="select o from Order o")
    @NamedQuery(name="Orders.Suspended" , query="select o from Order
    o where o.status=false")
@NamedQuery(name="Orders.RemoveSuspend" , query="update Order o set
    o.status = true where o.Id=:orderId")
@NamedQuery(name="PurchasesByPackages" , query = "select count
    (distinct o) from Order o group by o.pack")
@NamedQuery(name="PurchasesByPackagesID" , query = "select
    o.pack.name, count(o.pack) from Order o where o.status=true group
    by o.pack ")
@NamedQuery(name="Orders.UserInsolvances" , query = "select o from
    Order o where o.status=false and o.user.id = :userId")

public class Order {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    int Id;
    Date startDate;
    Date creationDate;
```

```
    float totalPayment;
    Boolean status =null;
```

RateCost Entity of a package rate cost, depending on the *monthValidity* period.

```
@Entity
@Table(name="Rate_costs", schema = "test")
public class RateCost {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    int id;
    int monthValidity;
    int cost;
    int packageId;
```

Service Entity of services, used as a connection between *Packages* and various services as mentioned in figure 2

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "services", schema = "test")
public class Service {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    int id;
    int packageId;
    @Column(name = "DTYPE")
    String type;
```

User Entity with data of users, and boolean *Insolvent* to determine if there are three or more purchases not completed.

```
@Entity
@NamedQuery(name = "User.authentication", query = "select usr from
    User usr WHERE usr.username = :username and usr.password =
    :password")
@NamedQuery(name = "User.insolvent" , query = "select usr from User
    usr WHERE usr.insolvent = true")
@Table(name="Users", schema = "test")
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    int id;
```

```
    String username;
    String password;
    String email;
    String type;
    boolean insolvent;
```

Package Entity of purchased packages with its rates, services and optional products.

```
@Entity
@NamedQuery(name="Packages.All",query="select p from Package p")
@Table(name = "Packages", schema = "test")
public class Package {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    int    id;
    String name;


    @OneToMany
    @JoinColumn(name = "packageId")
    List<Service> services;

    @OneToMany
    @JoinColumn(name = "packageId")
    List<RateCost> rates;

    @JoinTable(
           name = "Packages_OptionalProducts",
           schema = "test",
           joinColumns = @JoinColumn(name = "packageId"),
           inverseJoinColumns = @JoinColumn(name = "productId"))
    @ManyToMany
    List<OptionalProduct> products;
```

Custom Furthermore to manage the views, some custom classes have been created to store query views correctly. These classes reflect exactly the type of data queried in Views from page 3. A list of them:

- InsolventReport
- OptionalProductBestSeller
- OptionalProductsAverage
- PurchasesCoun

- PurchasesCountGrouped
- ValueOfSalesDetailed

5. Application Components

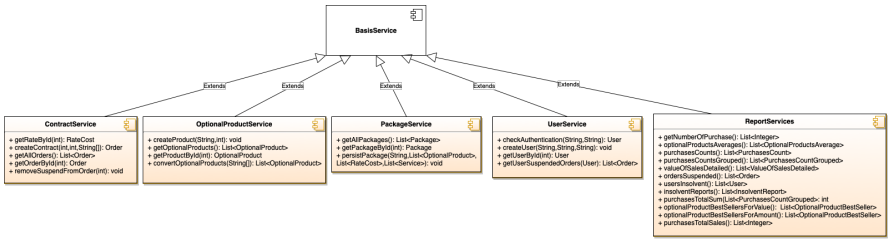

Figure 4. Services

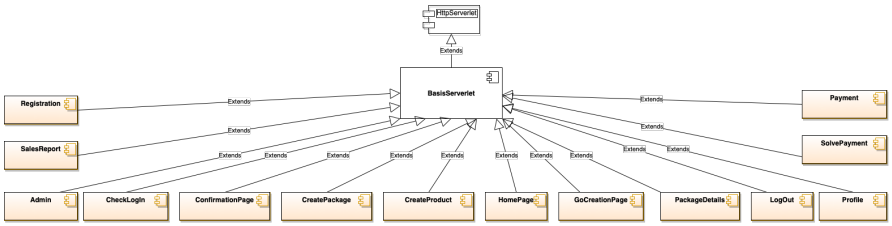

Figure 5. Controllers

6. UML sequence diagrams

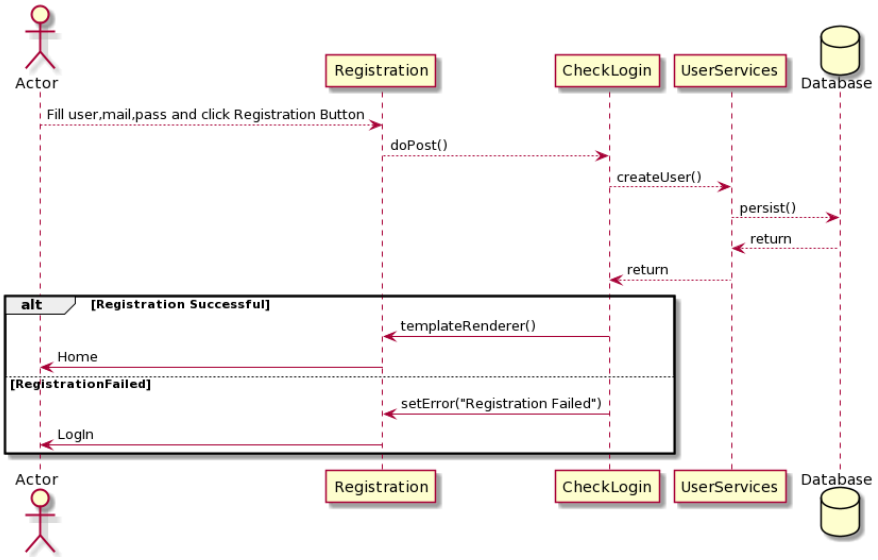

Figure 6. Registration

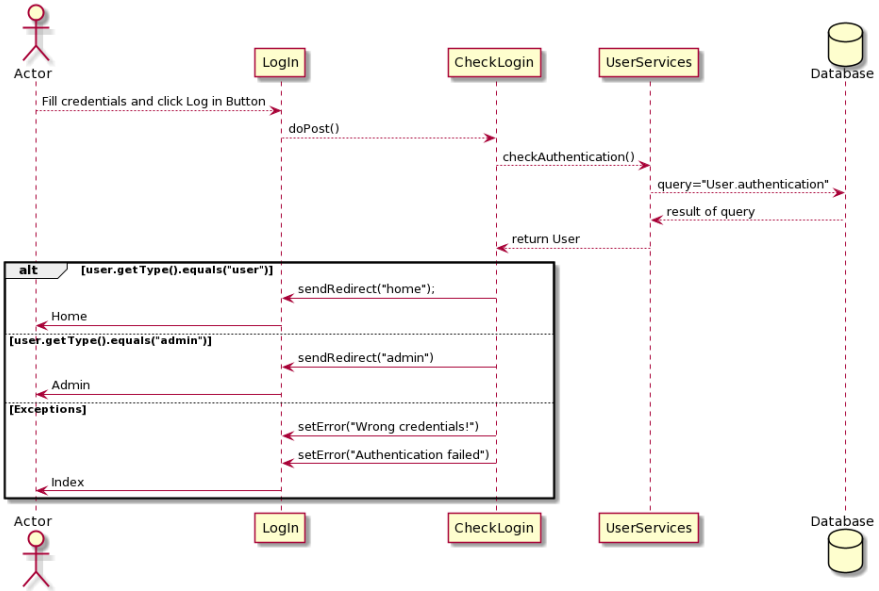

Figure 7. Log In

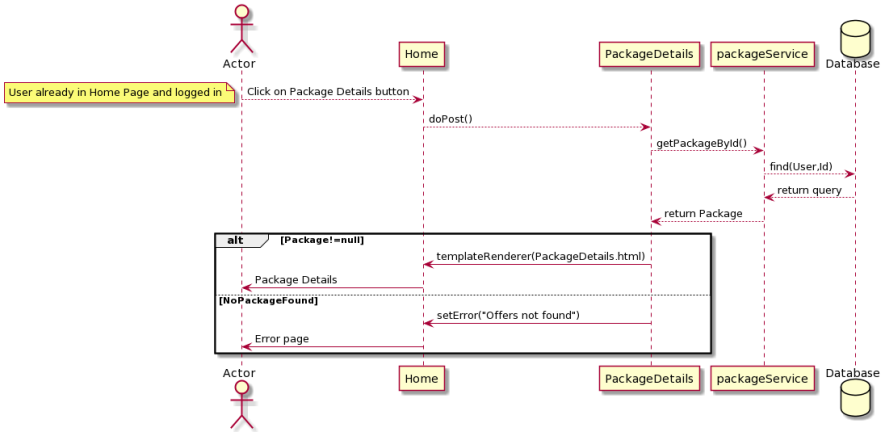

Figure 8. Package Details

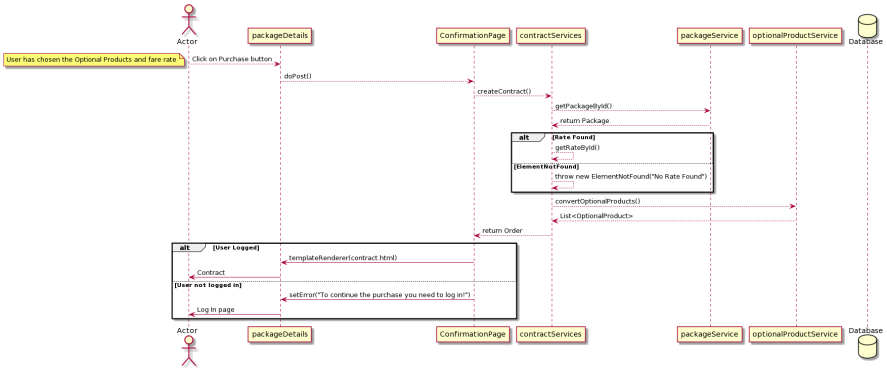

Figure 9. Purchase

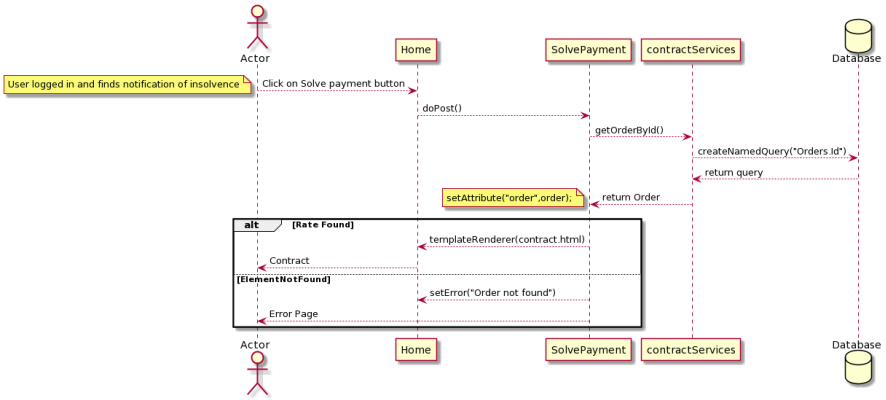

Figure 10. Insolvent User

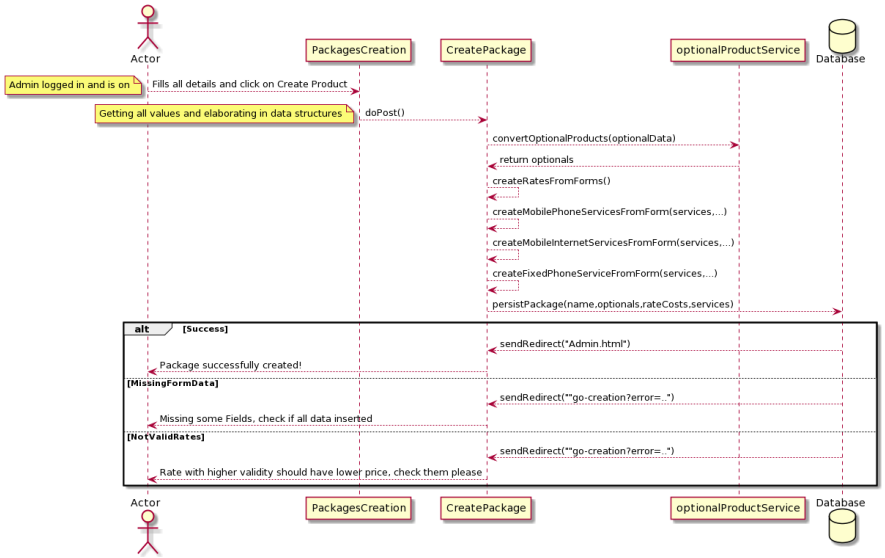

Figure 11. Package Creation