Data base 2

Project documentation

Canali Davide - 10674880
Cordioli Matteo - 10611332

Index

Specification
- Revision of the specifications

Conceptual (ER) and logical data models
- Explanation of the logical model

Trigger design and code

ORM relationship design

Entities code

Interface diagrams or functional analysis of the specifications

List of components
- Motivations of the design of the components

UML sequence diagrams

Specifications 1/2

TELCO SERVICE APPLICATIONS

A telco company offers pre-paid online services to web users. Two client applications using the same database need to be developed.

CONSUMER APPLICATION

The consumer application has a public Landing page with a form for login and a form for registration. Registration requires a username, a password and an email. Login leads to the Home page of the consumer application. Registration leads back to the landing page where the user can log in.

The user can log in before browsing the application or browse it without logging in. If the user has logged in, his/her username appears in the top right corner of all the application pages.

The Home page of the consumer application displays the service packages offered by the telco company.

A service package has an ID and a name (e.g., “Basic”, “Family”, “Business”, “All Inclusive”, etc). It comprises one or more services. Services are of four types: fixed phone, mobile phone, fixed internet, and mobile internet. The mobile phone service specifies the number of minutes and SMSs included in the package plus the fee for extra minutes and the fee for extra SMSs. The mobile and fixed internet services specify the number of Gigabytes included in the package and the fee for extra Gigabytes. A service package must be associated with one validity period. A validity period specifies the number of months (12, 24, or 36). Each validity period has a different monthly fee (e.g., 20€/month for 12 months, 18€/month for 24 months, and 15€ /month for 36 months). A package may be associated with one or more optional products (e.g., an SMS news feed, an internet TV channel, etc.). The validity period of an optional product is the same as the validity period that the user has chosen for the service package. An optional product has a name and a monthly fee independent of the validity period duration. The same optional product can be offered in different service packages.

From the Home page, the user can access a Buy Service page for purchasing a service package and thus creating a service subscription. The Buy Service page contains a form for purchasing a service package. The form allows the user to select one package from the list of available ones and choose the validity period duration and the optional products to buy together with the chosen service. The form also allows the user to select the start date of his/her subscription. After choosing the service packages, the validity period and (0 or more) optional products, the user can press a CONFIRM button. The application displays a CONFIRMATION page that summarizes the details of the chosen service package, the validity period, the optional products and the total price to be pre-paid: (monthly fee of service package * number of months) + (sum of monthly fees of options * number of months).

If the user has already logged in, the CONFIRMATION page displays a BUY button. If the user has not logged in, the CONFIRMATION page displays a link to the login page and a link to the REGISTRATION page. After either logging in or registering and immediately logging in, the CONFIRMATION page is redisplayed with all the confirmed details and the BUY button.

Specifications 2/2

When the user presses the BUY button, an order is created. The order has an ID and a date and hour of creation. It is associated with the user and with the service package, its validity period and the chosen optional products. It also contains the total value (as in the CONFIRMATION page) and the start date of the subscription. After creating the order, the application bills the customer by calling an

external service. If the external service accepts the billing, the order is marked as valid and a service activation schedule is created for the user. A service activation schedule is a record of the services and optional products to activate for the user with their date of activation and date of deactivation.

If the external service rejects the billing, the order is put in the rejected status and the user is flagged as insolvent. When an insolvent user logs in, the home page also contains the list of rejected orders. The user can select one of such orders, access the CONFIRMATION page, press the BUY button and attempt the payment again. When the same user causes three failed payments, an alert is created in a dedicated auditing table, with the user Id, username, email, and the amount, date and time of the last rejection.

EMPLOYEE APPLICATION

The employee application allows the authorized employees of the telco company to log in. In the Home page, a form allows the creation of service packages, with all the needed data and the possible optional products associated with them. The same page lets the employee create optional products as well.

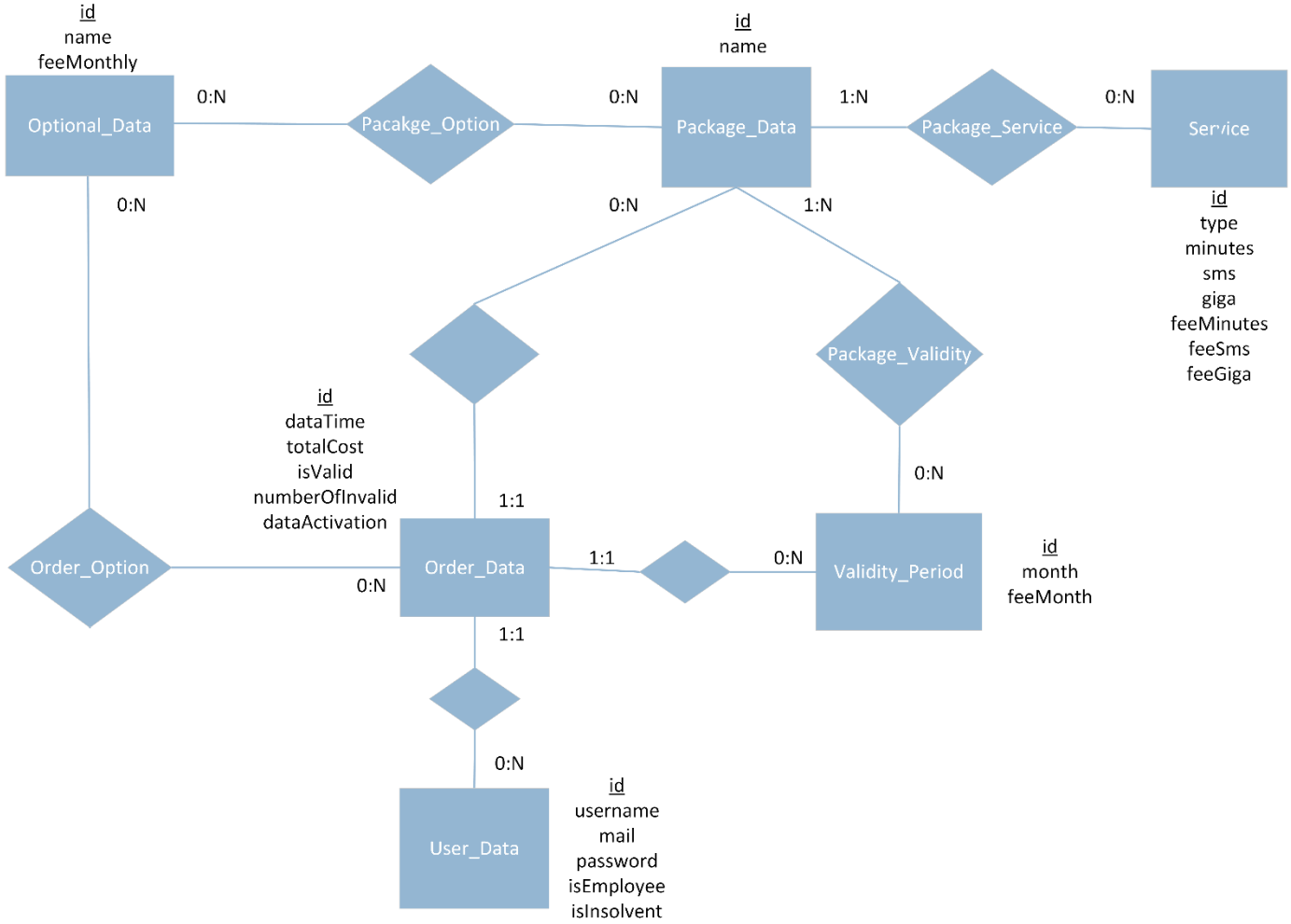
A Sales Report page allows the employee to inspect the essential data about the sales and about the users over the entire lifespan of the application:

- Number of total purchases per package.
- Number of total purchases per package and validity period.
- Total value of sales per package with and without the optional products.
- Average number of optional products sold together with each service package.
- List of insolvent users, suspended orders and alerts.
- Best seller optional product, i.e. the optional product with the greatest value of sales across all the sold service packages.

Specification interpretation

- Employee are already inserted in the DB.
- Services are already inserted in the DB
- There could be multiple validity period with the same duration
- There could be multiple optional product with same name
- These application is a single module of a bigger one which will handle all the other possible interactions with users and employees

Entity Relationship



Motivations of the ER design

- We don’t have a table where we save the info about each service’s start and end date per client since they’re easily computable. We decided to use a view table because all the data needed are already stored in our DB.
- Each package has multiple optionals, services and validity that represent the choices between which the user can pick.
- Order data is linked with optionals and validity to save the user choices among the available ones.
- To answer the required queries we save the data in 5 materialized views populated by triggers when a specific event occurs.

Logical Schema

- User_Data(id, username, mail, password, isEmployee, isInsolvent)
- Package_Data(id, name)
- Optional_Data(id, name, feeMonthly)
- Service(id, type, minutes, sms, giga, feeMinutes, feeSms, feeGiga)
- ValidityPeriod(id, month, feeMonth)
- Order_data(id, idUser, idPackage, idValidityperiod, dateTime, totalCost, isValid, numberOfInvalid, dateActivation)
- Package_Optional(idPackage, idOptional)
- Order_Option(idOrder, idOptional)
- Package_Service(idPackage, idService)
- Package_Validity(idPackage, idValidity)

Trigger: createSeller

- After an insert in the optional_data table
- Insert a new row in the seller_optional table of the new inserted optional

```
CREATE TRIGGER `createSeller`
AFTER INSERT ON `optional_data`
FOR EACH ROW
begin
        insert into seller_optional
        values(new.id, new.name, new.feeMonthly, 0);
end
```

Trigger: newPackageValidityOrder

- After an insert in the order_data table
- If the payment has been successful
- Update the count of purchases of that package with that validity period in the purchases_package_validity table

```
CREATE TRIGGER `newPackageValidityOrder`
AFTER INSERT ON `order_data`
FOR EACH ROW begin
    if(new.isValid=1) then
        update purchases_package_validity
        set numPurc=numPurc+1
        where new.idPackage=idPack and new.idValidityPeriod=idValidity;
    end if;
end
```

Trigger: newPackageOrder

- After an insert in the order_data table
- If the payment has been successful
- Update the value with and without optional and the number of purchases of that specific package in the purchases_package table

```
CREATE TRIGGER `newPackageOrder`
AFTER INSERT ON `order_data`
FOR EACH ROW begin
        DECLARE feeM FLOAT;
        DECLARE mon INT;
        DECLARE numOpt INT;
        if(new.isValid=1) then
                select feeMonth into feeM from validityperiod as vp where vp.id=new.idValidityPeriod;
                select month into mon from validityperiod as vp where vp.id=new.idValidityPeriod;

                update purchases_package
                set valueOptional=valueOptional+new.totalCost,
                numPurc=numPurc+1, valueNoOptional=valueNoOptional+(feeM*mon)
                where idPack=new.idPackage;
        end if;
end
```

Trigger: newSuspended

- After an insert in the order_data table
- If the payment has been rejected
- Insert in the suspended_order table the rejected order

```
CREATE TRIGGER `newSuspended`
AFTER INSERT ON `order_data`
FOR EACH ROW begin
        DECLARE nameP VARCHAR(45);
        DECLARE userMail VARCHAR(45);
        if(new.isValid=0) then
                select mail into userMail from user_data where user_data.id= new.idUser;
                select name into nameP from package_data where package_data.id= new.idPackage;

                insert into suspended_order values(new.id, nameP, userMail);
        end if;
end
```

Trigger: addAlert

- After an insert in the order_data table
- If the payment has been rejected
- If the user had 3 failed payment
- Add a new alert in the alert table

```
CREATE TRIGGER `addAlert`
AFTER INSERT ON `order_data`
FOR EACH ROW begin
        declare numErr int;
        declare emailUsr varchar(45);
        declare usernameUsr varchar(45);
        if (new.isValid = 0) then
                select sum(numberOfInvalid) into numErr from order_data where idUser = new.idUser;
                if(numErr % 3 = 0) then
                        select mail into emailUsr from user_data where id = new.idUser;
                        select username into usernameUsr from user_data where id = new.idUser;
                        insert into alert(idUser,email,username,totalCost,lastReject)
                                values(new.idUser,emailUsr,usernameUsr,new.totalCost,NOW());
                end if;
        end if;
end
```

Trigger: updatePackageValidityOrder

- After an update on the order_data table
- If the payment has been accepted and before was a rejected one
- Update the purchases_package_validity table by incrementing the number of purchases of the package

```
CREATE TRIGGER `updatePackageValidityOrder`
AFTER UPDATE ON `order_data` FOR EACH ROW begin
        if(new.isValid=1 and old.isValid=0) then
                update purchases_package_validity
                set numPurc=numPurc+1
                where new.idPackage=idPack and new.idValidityPeriod=idValidity;
        end if;
end
```

Trigger: updateSuspended

- After an update on the order_data table
- If the payment has been accepted and before was a rejected one
- Remove that order from the suspended_order table

```
CREATE TRIGGER `updateSuspended`
AFTER UPDATE ON `order_data`
FOR EACH ROW begin
        if(new.isValid=1 and old.isValid=0) then
                delete from suspended_order
                where new.id=idOrder;
        end if;
end
```

Trigger: updateSellerOrder

- After an update on the order_data table
- If the payment has been accepted and before was a rejected one
- Update the total Earning of each optional in that order

```
CREATE TRIGGER `updateSellerOrder`
AFTER UPDATE ON `order_data`
FOR EACH ROW begin
        DECLARE mon INT;
        if(new.isValid=1 and old.isValid=0) then
                select month into mon from validityperiod where new.idValidityPeriod=validityperiod.id;

                update seller_optional
                set totEarn=totEarn+(mon*feeMonth)
                where idOptional in (select idOptional from order_option where idOrder=new.id);
        end if;
end
```

Trigger: updatePackageOrder

- After an update on the order_data table
- If the payment has been accepted and before was a rejected one
- Update the total earning of each optional in that order

```
CREATE TRIGGER `updatePackageOrder`
AFTER UPDATE ON `order_data`
FOR EACH ROW begin
        DECLARE feeM FLOAT;
        DECLARE mon INT;
        DECLARE amount INT;
        if(new.isValid=1 and old.isValid=0) then
                select count(*) into amount from order_option where idOrder=new.id;
                select feeMonth into feeM from validityperiod as vp where vp.id=new.idValidityPeriod;
                select month into mon from validityperiod as vp where vp.id=new.idValidityPeriod;

                update purchases_package
                set valueOptional=valueOptional+new.totalCost,
                numPurc=numPurc+1, averageOpt=(((numPurc-1)*averageOpt)+amount)/(numPurc),
                        valueNoOptional=valueNoOptional+(feeM*mon)
                where idPack=new.idPackage;
        end if;
end
```

Trigger: updateAlert

- After an update on the order_data table
- If the number of invalids has changed
- If the user has failed a payment 3 times
- Insert a new alert in the Alert table

```
CREATE TRIGGER `updateAlert`
AFTER UPDATE ON `order_data`
FOR EACH ROW begin
        declare numErr int;
        declare emailUsr varchar(45);
        declare usernameUsr varchar(45);
        if (new.numberOfInvalid <> old.numberOfInvalid) then
                select sum(numberOfInvalid) into numErr from order_data where idUser = new.idUser;
                if(numErr % 3 = 0) then
                        select mail into emailUsr from user_data where id = new.idUser;
                        select username into usernameUsr from user_data where id = new.idUser;

                        insert into alert(idUser,email,username,totalCost,lastReject)
                                values(new.idUser,emailUsr,usernameUsr,new.totalCost,NOW());
                end if;
        end if;
end
```

Trigger: updateSeller

- After an insert on the order_option table
- If the order is valid
- Update the seller_optional table by increasing the total earning of the bought optional

```
CREATE TRIGGER `updateSeller`
AFTER INSERT ON `order_option`
FOR EACH ROW begin
        DECLARE valid BOOLEAN;
        DECLARE mon INT;
        select isValid into valid from order_data where new.idOrder=order_data.id;
        select month into mon from order_data join validityperiod on idValidityPeriod=validityperiod.id where new.idOrder=order_data.id;

        if(valid=1) then
                update seller_optional
                set totEarn=totEarn+(mon*feeMonth)
                where new.idOptional=idOptional;
        end if;
end
```

Trigger: newAverage

- After an insert on the order_option table
- If the order is valid
- Update the purchases_package table by updating the average number of optional of the bought package

```
CREATE TRIGGER `newAverage`
AFTER INSERT ON `order_option`
FOR EACH ROW begin
        DECLARE valid BOOLEAN;
        DECLARE idPacka INT;
        DECLARE numOfPack INT;

        select idPackage into idPacka from order_data where new.idOrder=order_data.id;
        select isValid into valid from order_data where new.idOrder=order_data.id;
        select count(*) into numOfPack from order_option as oo where oo.idOrder in (select id from order_data where idPackage=idPacka);

        if(valid=1) then
                update purchases_package
                set averageOpt=(numOfPack)/(numPurc)
                where idPack=idPacka;
        end if;
end
```

Trigger: newPackage

- After an insert on the package_data table
- Insert into Purchases_package the newly added package

```
CREATE TRIGGER `newPackage`
AFTER INSERT ON `package_data`
FOR EACH ROW begin
        insert into purchases_package values (new.id, new.name, 0, 0, 0, 0);

end
```

Trigger: newPackageValidity

- After an insert on the package_validity table
- Insert into Purchases_package_validity the newly added package validity pair

```
CREATE TRIGGER `newPackageValidity`
AFTER INSERT ON `package_validity`
FOR EACH ROW begin
        declare myname VARCHAR(45);

        select name into myname from package_data where id=new.idPackage;

        insert into purchases_package_validity values (new.idPackage, myname, 0, new.idValidity);
end
```

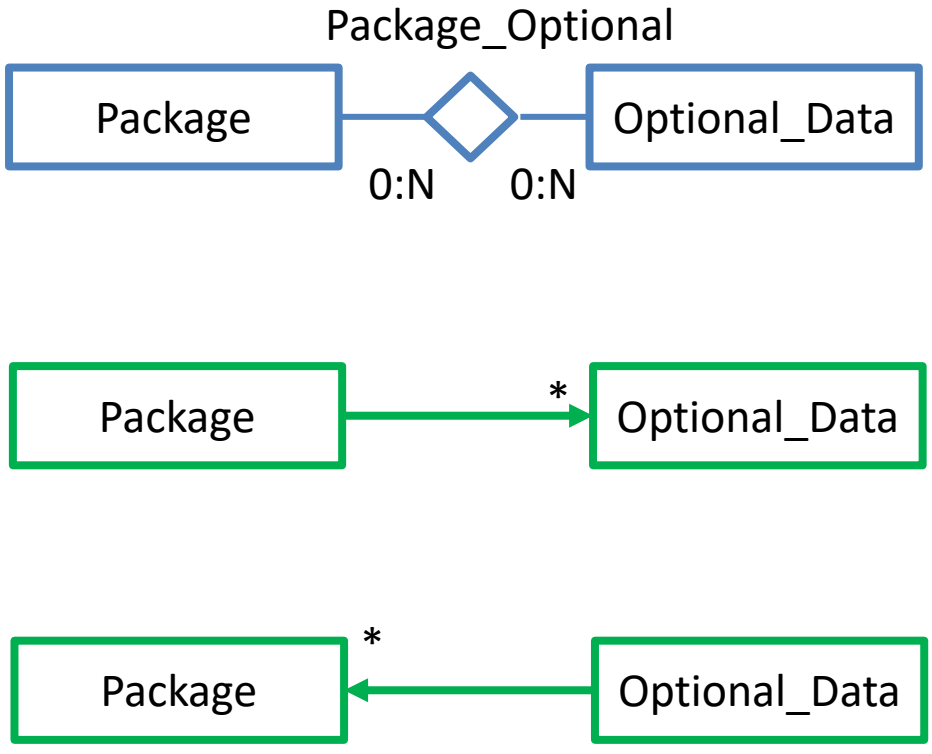
Trigger: newInsolventUser

- After an update on the user_data table
- If the user has become insolvent add the user to the insolvent_user table
- Else if the user bought all his suspended orders delete him from the insolvent_user table

```
CREATE TRIGGER `newInsolventUser`
AFTER UPDATE ON `user_data`
FOR EACH ROW begin
        if(new.isInsolvent=1 and old.isInsolvent=0) then
                insert into insolvent_user values(new.id, new.username, new.mail);
        else
                if(new.isInsolvent=0 and old.isInsolvent=1) then
                        delete from insolvent_user
                        where new.id=idUser;
                end if;
        end if;
end
```

ORM Design

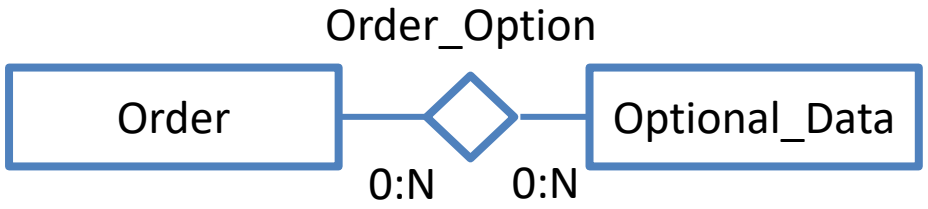
Relationship “Package_Optional”



Package → Optional_Data, a package could have more than one optional

Optional_Data → Package, an optional could be associated with N package

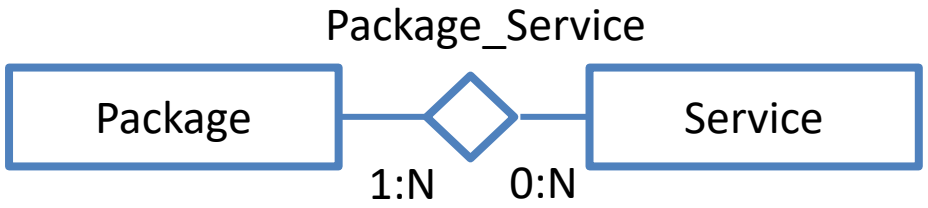
Relationship “Order_Option”



Order → Optional_Data, an order could be done with more N Optional

Optional_Data → Order, an optional could be ordered more than once

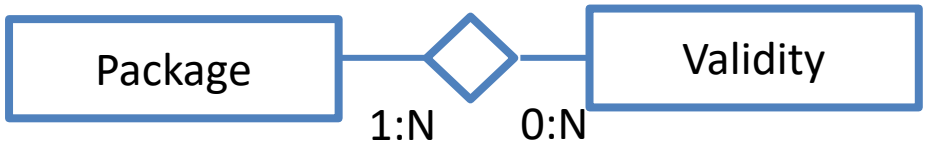
Relationship “Package_Service”



Package → Service, a package can contain N services

Service → Package, the same service can be used in more than one package

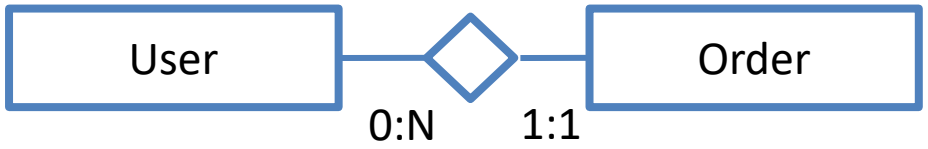
Relationship “Package_Validity”



Package → Validity, a package can be bought in more than one validity period

Validity → Package, the same validity period could be used in more than one package

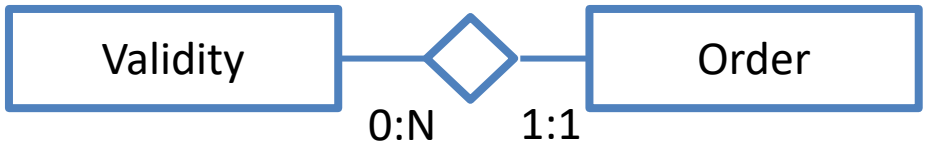
Relationship “OrderedUser”



User → Order, a user can order more than one time

Order→ User, an order is done by one user

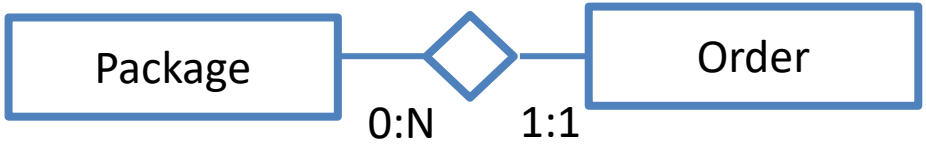
Relationship “OrderedValidity”



Validity → Order, a validity period could use in multiple order

Order → Validity, an order has exactly one validity period

Relationship “OrderedPack”



Package → Order, a pack can be bought N times

Order → Package, an order can be done with only one package

Entity Design

OptionalData Entity

```
@Entity
@Table(name="optional_data")
@NamedQueries({
   @NamedQuery(name="OptionalData.findAll",
         query="SELECT o FROM OptionalData o"),
   @NamedQuery(name="OptionalData.findByIds",
         query="SELECT o FROM OptionalData o where o.id in ?1")
})

public class OptionalData implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int id;
   private float feeMonthly;
   private String name;

   @ManyToMany(mappedBy="optionalData")
   private List<OrderData> orderData;

   @OneToMany(mappedBy="optionalData")
   private List<OrderOption> orderOptions;

   @ManyToMany(mappedBy="optionalData")
   private List<PackageData> packageData;

   @OneToMany(mappedBy="optionalData")
   private List<PackageOption> packageOptions;
}
```

OrderData Entity

```
@Entity
@Table(name = "order_data")
@NamedQueries({
   @NamedQuery(name = "OrderData.findAll",
         query = "SELECT o FROM OrderData o"),
   @NamedQuery(name = "OrderData.findAllSuspended",
         query = "SELECT o FROM OrderData o WHERE o.isValid = false")
})

public class OrderData implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;
   @Temporal(TemporalType.DATE)
   private Date dataActivation;
   private Timestamp dateTime;
   private boolean isValid;
   private int numberOfInvalid;
   private float totalCost;

   @OneToMany(mappedBy = "orderData")
   private List<OrderOption> orderOptions;

   @ManyToMany
      @JoinTable(name = "order_option"
        , joinColumns = { @JoinColumn(name = "idOrder") }
        , inverseJoinColumns = { @JoinColumn(name = "idOptional") })
      private List<OptionalData> optionalData;

      @ManyToOne
      @JoinColumn(name = "idPackage")
      private PackageData packageData;

      @ManyToOne
      @JoinColumn(name = "idUser")
      private UserData userData;

      @ManyToOne
      @JoinColumn(name = "idValidityPeriod")
       private Validityperiod validityperiod;
}
```

OrderOption Entity

```
@Entity
@Table(name="order_option")
@NamedQuery(name="OrderOption.findAll",
        query="SELECT o FROM OrderOption o")

public class OrderOption implements Serializable {
   private static final long serialVersionUID = 1L;
   @EmbeddedId
   private OrderOptionPK id;

   @ManyToOne
   @JoinColumn(name="idOptional")
   private OptionalData optionalData;

   @ManyToOne
   @JoinColumn(name="idOrder")
   private OrderData orderData;
}
```

PackageData Entity

```
@Entity
@Table(name="package_data")
@NamedQuery(name="PackageData.findAll",
        query="SELECT p FROM PackageData p")
public class PackageData implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int id;
   private String name;

   @OneToMany(mappedBy="packageData")
   private List<OrderData> orderData;

   @ManyToMany
   @JoinTable(  name="package_option"
     , joinColumns={  @JoinColumn(name="idPackage")}
     , inverseJoinColumns={ @JoinColumn(name="idOptional") }  )
   private List<OptionalData> optionalData;

   @OneToMany(mappedBy="packageData")
   private List<PackageService> packageServices;

   @ManyToMany
   @JoinTable(
      name="package_service"
     , joinColumns={
        @JoinColumn(name="idPackage")}
    , inverseJoinColumns={
        @JoinColumn(name="idService")})
   private List<Service> services;

   @OneToMany(mappedBy="packageData")
   private List<PackageOption> packageOptions;

   @OneToMany(mappedBy="packageData")
   private List<PackageValidity> packageValidities;

   @ManyToMany
   @JoinTable(
     name = "package_validity“
    , joinColumns= { @JoinColumn(name="idPackage")}
     , inverseJoinColumns= {@JoinColumn(name="idValidity")})
   private List<Validityperiod> validityPeriods;
}
```

PackageOption Entity

```
@Entity
@Table(name="package_option")
@NamedQueries({
@NamedQuery(name="PackageOption.findAll",
        query="SELECT p FROM PackageOption p")
})

public class PackageOption implements Serializable {
   private static final long serialVersionUID = 1L;
   @EmbeddedId
   private PackageOptionPK id;

   @ManyToOne
   @JoinColumn(name="idPackage")
   private PackageData packageData;

   @ManyToOne
   @JoinColumn(name="idOptional")
   private OptionalData optionalData;
}
```

PackageService Entity

```
@Entity
@Table(name="package_service")
@NamedQuery(name="PackageService.findAll",
        query="SELECT p FROM PackageService p")

public class PackageService implements Serializable {
   private static final long serialVersionUID = 1L;
   @EmbeddedId
   private PackageServicePK id;

   @ManyToOne
   @JoinColumn(name="idPackage")
   private PackageData packageData;

   @ManyToOne
   @JoinColumn(name="idService")
   private Service service;
}
```

PackageValidity Entity

```
@Entity
@Table(name="package_validity")
public class PackageValidity implements Serializable {
   @EmbeddedId
   private PackageValidityPK id;

   @ManyToOne
   @JoinColumn(name="idPackage")
   private PackageData packageData;

   @ManyToOne
   @JoinColumn(name="idValidity")
   private Validityperiod validityPeriod;
}
```

Service Entity

```
@Entity
@NamedQueries({
   @NamedQuery(name = "Service.findAll",
         query = "SELECT s FROM Service s"),
   @NamedQuery(name = "Service.findByIds",
         query = "SELECT s FROM Service s where s.id in ?1")
})

public class Service implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;
   private float feeGiga;
   private float feeMinutes;
   private float feeSms;
   private float giga;
   private int minutes;
   private int sms;
   private String type;

   @OneToMany(mappedBy = "service")
   private List<PackageService> packageServices;

   @ManyToMany
   @JoinTable(name = "package_service“
     , joinColumns = { @JoinColumn(name = "idService")}
     , inverseJoinColumns = {@JoinColumn(name = "idPackage") })
   private List<PackageData> packageData;
}
```

UserData Entity

```
@Entity
@Table(name="user_data")
@NamedQueries({
   @NamedQuery(name="UserData.findAll",
         query="SELECT u FROM UserData u"),
   @NamedQuery(name = "UserData.checkCredentials",
         query = "SELECT u FROM UserData u WHERE u.mail = ?1 and u.password = ?2"),
   @NamedQuery(name= "UserData.findAllInsolvent",
          query= "SELECT u FROM UserData u WHERE u.isInsolvent = true")
})
public class UserData implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int id;
   private boolean isEmployee;
   private boolean isInsolvent;
   private String mail;
   private String password;
   private String username;

   @OneToMany(mappedBy="userData")
   private List<OrderData> orderData;
}
```

ValidityPeriod Entity

```
@Entity
@NamedQueries({ @NamedQuery(name =
"Validityperiod.findAll",
        query = "SELECT v FROM Validityperiod v"),
    @NamedQuery(name = "Validityperiod.findByIds",
        query = "SELECT v FROM Validityperiod v where v.id in ?1")
})
public class Validityperiod implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;
   private float feeMonth;
   private int month;

   @OneToMany(mappedBy = "validityperiod")
   private List<OrderData> orderData;

   @OneToMany(mappedBy = "validityPeriod")
   private List<PackageValidity> packageValidities;

   @ManyToMany
   @JoinTable(name = "package_validity“
     , joinColumns = { @JoinColumn(name = "idValidity")}
     , inverseJoinColumns = {@JoinColumn(name = "idPackage") })
   private List<PackageData> packageData;
}
```

Functional analysis of the interaction

A Web application allows users to buy new telecommunication services. On the **Landing Page,** a user can continue as a guest, **Login** or **Register** a new account the last two via **forms**. If the user decides to **continue as a guest**, he/she can navigate the website until does an action that requires some credential. After **logging** in the user is **redirected** to his **HomePageClient** in which the user can see a **list of all the available packages** and his/her suspended order, the user can **start buying a package** via a **button** that it will **redirect** to a **Buy Package** page in which it's possible to **select** an available package, and with that, the user must **select** a validity period associated with that and if available some options **can be chosen** and the starting date of the service need to be **selected**, after that the user is **redirected** to the **Confirmation** page in which a summary of what has been chosen is **showed** and the total cost is **calculated** and the user can log in if he/she **has entered as a guest** in the web site or buy the chosen package. After that he/she will be **redirected** to the **home page**.

Pages, view components, events, actions

Functional analysis of the interaction

A Web application allows the employee to manage the package of the company. After **logging in** the employee is **redirected** to his **home page** in which it's possible to **create** new optional and new packages via a **form** or **check the sales report** with a **button** that **redirects** the employee to the **sales report** page in which he/she can **choose** between **6 queries**:

- Number of total purchases per package.
- Number of total purchases per package and validity period.
- Total value of sales per package with and without the optional products.
- Average number of optional products sold together with each service package.
- List of insolvent users, suspended orders and alerts.
- Best seller optional product

Components

Servlets:

Login

Logout

Error

CreateUser

HomePageClient

BuyService

BuyOrder

LoadConfirm

Confirmation

HomePageEmployee

CreatePackage

CreateOptional

AverageOptionals

InsolventSuspendedAlert

MostValueOptional

PackageValue

PurchasePerPackage

PurchasePerPackageValidity

Business Components

OptionalSrv -> Stateless:

- CreateOptional: handles the creation of an optional.
- FindByIds: return a list of optionals based on a set of ids.
- FindAll: return all the optionals.

OrderSrv -> Stateless:

- CreateOrder: handles the creation of an order.
- FindAllRejectedWithUserId: return all the invalid orders of a specific user.
- FindRejectedOrderOfuser: return a specific order of a specific user.
- BuyInsolvent: based on a Boolean set the order valid or increase the number of attempts to buy the order.

PackageSrv -> Stateless:

- CreatePackage: handle the creation of a new package.
- FinAllPackage: return all packages.
- FindPackageWithId: return the package with a specific id.
- TotalCostForPackage: compute the total cost of a package.

PeriodSrv -> Stateless:

- FindAllPeriods: return all the validity periods.
- FindValityWithId: return a validity period with a specific id.
- FindbyIds: return a list of validity periods based on a set of ids.

SalesReportSrv -> Stateless:

- TotalPurchasePerPackage: return the number of purchases for each package.
- TotalPurchasePerPackageAndValidity: return the number of purchases for each package, validity period pair.
- PackageValue: return the total revenue for each package with and without optional.
- AvrOptionalsPerPackage: return the average number of optionals bought for each package.
- MostValueOptional: return the optional that created the most revenue.
- FindAllSuspended: return every suspended order.
- FindAllAlert: return all alerts.
- FindAllInsovent: return every insolvent user.

ServiceSrv-> Stateless:

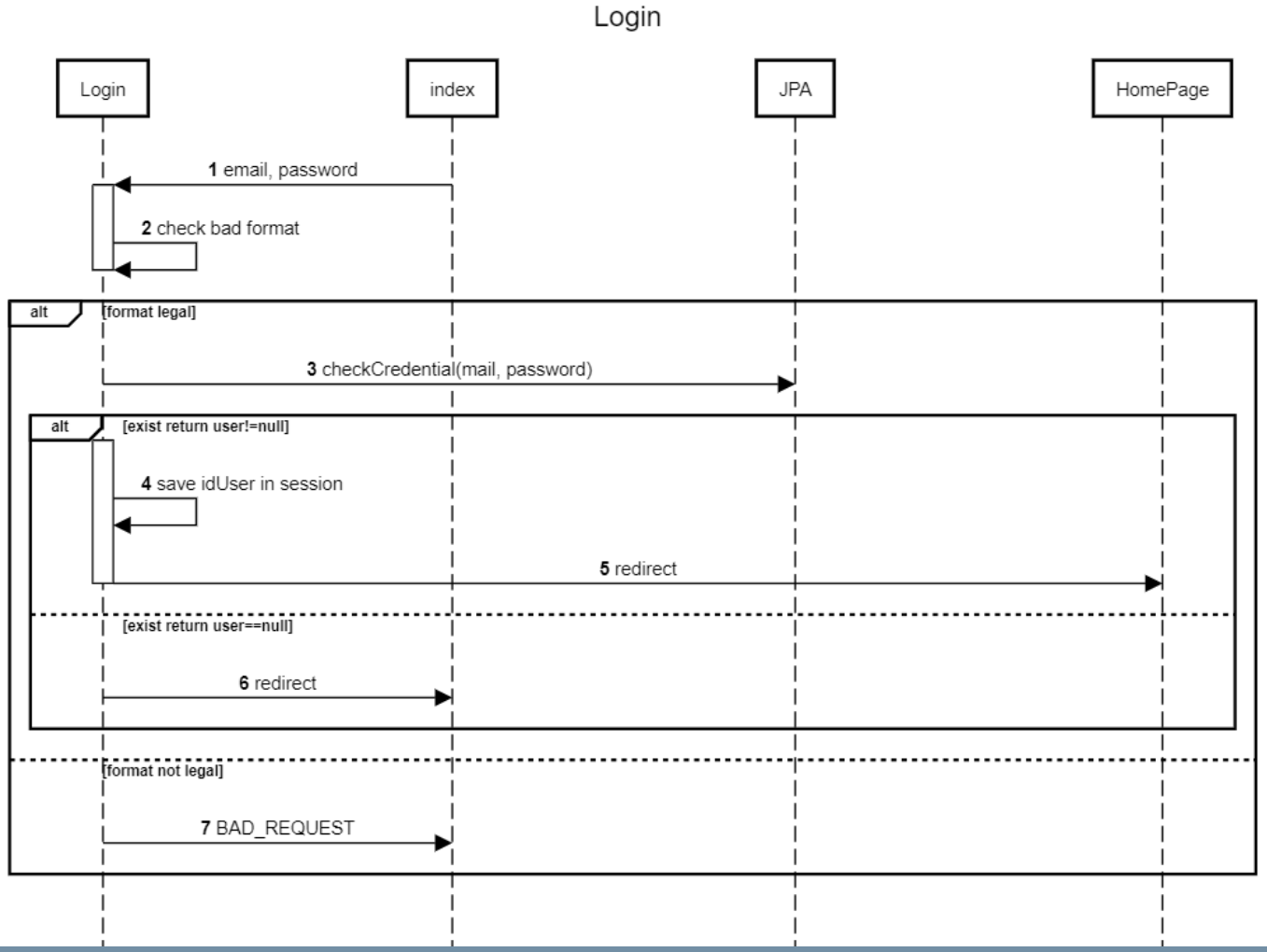
- FindByids: return a list of services based on a set of ids.
- FindAll: return all the services.

UserSrv -> Stateless:

- CreateUser: handle the creation of a user.
- FindUser: return a user with a specific id.
- IsEmployee: return true if the user is an employee.
- FindAllUser: return all the users.
- CheckCredentials: return a user based on his mail and password.

UML Sequence Diagram

Login diagram



Buy order diagram