Data Base 2 Project

Group 132

Gianluca Ruberto – 10607366

Andrea Prisciantelli - 10618568

Index

Specification Interpretation

- We assumed that the employees and the users are two distinct entities and therefore they need to access different pages for the restricted data that they need.
- A new employee cannot be created by using the application, instead, some employees have been created within the database.
- Since a service and a product are two different entities, we assumed that they have two different service activation schedule.

Functional analysis
Consumer application

Pages Components Action Events

The consumer application has a public Landing page with a form for login and a form for registration. Registration requires a username (which can be assumed as the unique identification parameter), 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 fixed phone service has no specific configuration parameters. 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.

Functional analysis: consumer application

Pages Components Action Events

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.

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.

Functional analysis
Employee application

Pages Components Action Events

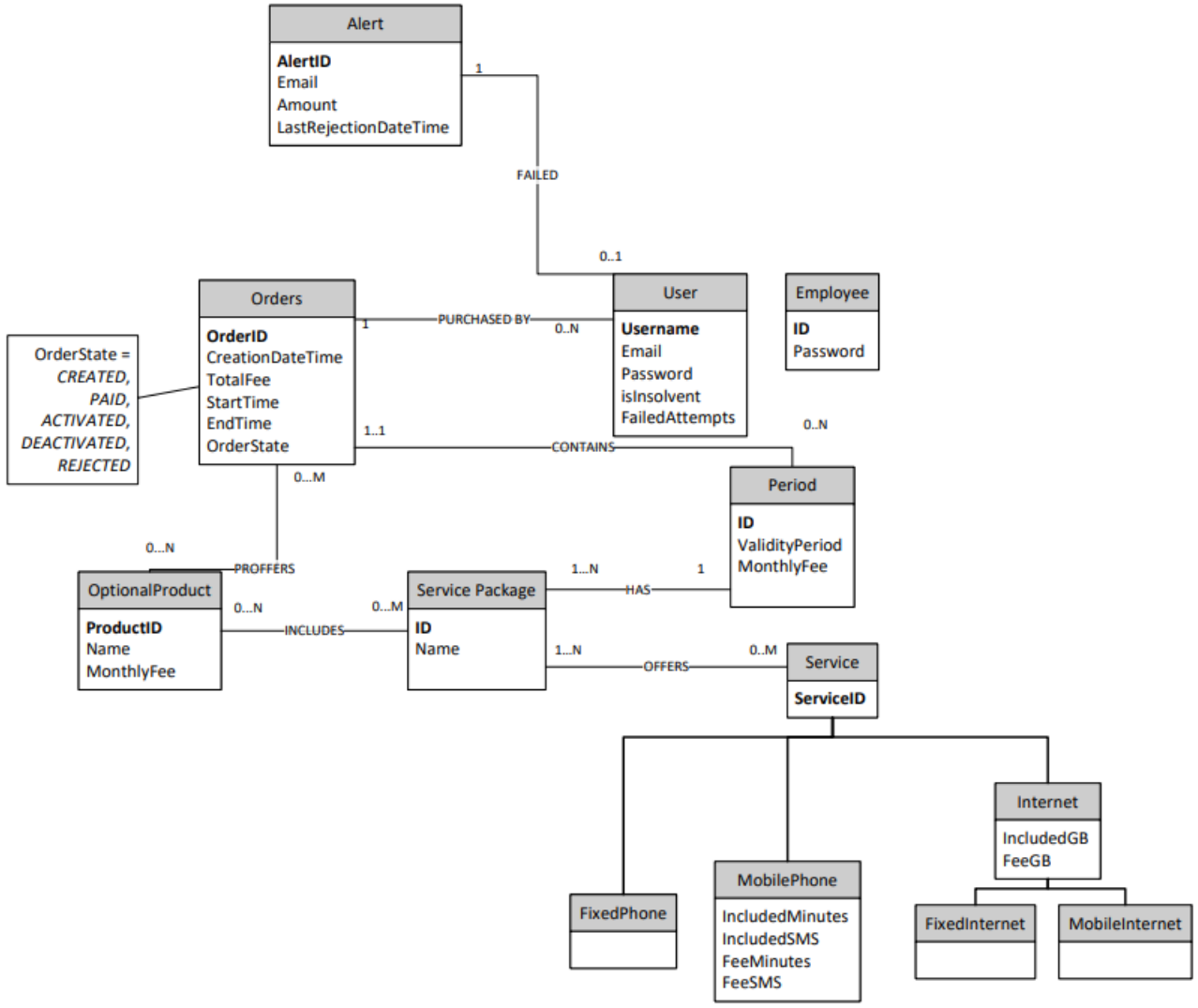
The employee application allows the authorized employees of the telco company to log in. In the Home page, a form allows the creation of service packages, with all the needed data and the possible optional products associated with them. The same page lets the employee create optional products as well.

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.

Conceptual (ER) and logical data models

Conceptual model



Explanation of the ER diagram

- We decided that an order can be made of one single service package, if the user wants more service packages, he can just place more orders;
- We decided to keep track if a user is insolvent and its failed payments in the users table. These fields are updated through triggers;
- The table Period has just one service package associated per record. Therefore, its primary key is an unique identifier of a certain service package with a certain time period and can be associated with an order;
- The only table of the ER diagram populated by triggers is the Alert table;
- Also, all the materialized view tables are populated by triggers and will be explained later;
- The state of an order is tracked on the database, and it follows the its entire life. For scalability purpose, it includes even the final stage that it is not subject of the project.

Logical model

- Orders(**OrderID**, CreationDateTime, TotalFee, StartTime, EndTime, OrderState,Username,PeriodID)
- User(**Username**, Email, Password, isInsolvent,FailedAttempts)
- Alert(**AlertID**,Username,Email, Amount,LastRejectionDateTime)
- Optional Product(**ProductID**, Name, MonthlyFee)
- Service Package(**PackageID,**Name)
- Period(**ID**, ValidityPeriod, MonthlyFee, PackageID)
- Service(**ServiceID**, ServiceType, IncludedMinutes,IncludedSMS, FeeMinutes, FeeSMS, IncludedGB, FeeGB)
- Employee(**ID**,Password)

Motivations of the logical design

- We decided to collapse up the Service classes. The field ServiceType is an Enum that can be: FixedPhone, MobilePhone, FixedInternet,MobileInternet;
- The many to many relationship are not displayed in the logical model and are mapped in bridge tables.

Description of the materialized view tables and code of the materialization triggers

- MySQL doesn’t allow materialized view tables and it needs actual tables. Despite this, the materialized view tables have not been included in the ER schema;
- All the materialized view tables are updated and populated solely by the use of triggers, as requested by the specification;
- Since trigger are used to perform updates on different tables, they all are “AFTER” triggers.

Number of total purchases per package (1)

```
create table `total_purchases_per_package`(
`packageId` int NOT NULL AUTO_INCREMENT,
`total_purchases` int NOT NULL DEFAULT 0,
PRIMARY KEY (`packageId`),
CONSTRAINT `total_purchases_per_package_servicepackage` FOREIGN KEY (`packageId`) REFERENCES `servicepackage` (`packageId`)
)
```

It contains the total number of purchases for each service package.

```
DELIMITER $$
create trigger insert_new_total_purchases_per_package after insert ON servicepackage
for each row
begin
        insert into total_purchases_per_package(packageId, total_purchases) values (new.packageId, 0);
end$$

DELIMITER ;
```

It initialize the value of total number of purchases for a service package to 0.

Number of total purchases per package (2)

```
DELIMITER $$

create trigger update_total_purchases_per_package after update ON
orders

for each row

begin

        if not(old.orderState <=> new.orderstate) and new.orderstate
<=> "Paid" then

                update total_purchases_per_package

                set total_purchases = total_purchases + 1

                where packageId in (select packageId from
period where new.periodId = ID);

    end if;

end$$

DELIMITER ;
```

It updates the number of service packages sold when the associated order goes from the rejected state to the paid state.

```
DELIMITER $$

create trigger insert_paid_total_purchases_per_package after insert ON
orders

for each row

begin

        if new.orderstate <=> "Paid" then

                update total_purchases_per_package

                set total_purchases = total_purchases + 1

                where packageId in (select packageId from
period where new.periodId = ID);

    end if;

end$$


DELIMITER ;
```

It updates the number of service packages sold when the associated order is paid at the first try.

Number of total purchases per package and validity period (1)

```
create table `total_purchases_per_package_validityperiod`(
`periodID` int NOT NULL AUTO_INCREMENT,
`total_purchases` int NOT NULL DEFAULT 0,
PRIMARY KEY (`periodID`),
CONSTRAINT `total_purchases_per_package_validityperiod_period`
FOREIGN KEY (`periodID`) REFERENCES `period` (`ID`)
)
```

It contains the total number of purchases for each service package at the granularity of the validity period.

```
DELIMITER $$
create trigger insert_new_total_purchases_per_package_validityperiod
after insert ON period
for each row
begin
        insert into
total_purchases_per_package_validityperiod(periodID, total_purchases)
values (new.ID,0);
end$$

DELIMITER ;
```

It sets the initial value of purchases per validity period to 0 when a new validity period is created.

Number of total purchases per package and validity period (2)

```
DELIMITER $$

create trigger update_total_purchases_per_package_validityperiod after update ON orders

for each row

begin

        if not(old.orderState <=> new.orderstate) and new.orderstate <=> "Paid" then

                update total_purchases_per_package_validityperiod

                set total_purchases = total_purchases + 1

                where periodID = new.periodId;

    end if;

end$$


DELIMITER ;
```

It updates the number of total service packages per validity period sold when the associated order goes from the rejected state to the paid state.

```
DELIMITER $$

create trigger insert_paid_total_purchases_per_package_validityperiod after insert ON orders

for each row

begin

        if new.orderstate <=> "Paid" then

                update total_purchases_per_package_validityperiod

                set total_purchases = total_purchases + 1

                where periodID = new.periodId;

    end if;

end$$


DELIMITER ;
```

It updates the number of service packages per validity period sold when the associated order is paid at the first try.

Total sales per package with and without optional product (1)

```
create table `total_sales_per_package`(
`packageId` int NOT NULL AUTO_INCREMENT,
`totalSales` int NOT NULL DEFAULT 0,
`totalSalesWithOptionalProduct` int NOT NULL DEFAULT 0,
PRIMARY KEY (`packageId`),
CONSTRAINT `total_sales_per_package_servicepackage` FOREIGN KEY (`packageId`) REFERENCES `servicepackage` (`packageId`)
)
```

It contains the total sales (in euros) for each service package with and without the optional products.

```
DELIMITER $$
create trigger insert_new_total_sales_per_package after insert ON servicepackage
for each row
begin
        insert into total_sales_per_package(packageId, totalSales,totalSalesWithOptionalProduct) values (new.packageId, 0,0);
end$$

DELIMITER ;
```

It initializes the initial value of sales to 0 when a new service package is created.

Total sales per package with and without optional product (2)

```
DELIMITER $$
create trigger update_sales_purchases_per_package after update ON orders
for each row
begin
declare packageId2 int;
declare monthlyFee2 float;
declare validityPeriod2 int;
                                                    select
                                                                packageId,
                                                                monthlyFee,
                                                                validityPeriod
        into
        packageId2, monthlyFee2, validityPeriod2
                                                    from period where new.periodId = ID;
                                if not(old.orderState <=> new.orderstate) and new.orderstate <=> "Paid" then
                                                    update total_sales_per_package
                                                    set totalSales = totalSales + monthlyFee2*validityPeriod2,
                                                                totalSalesWithOptionalProduct = totalSalesWithOptionalProduct + new.totalFee
                                                    where packageId = packageId2;
    end if;
end$$

DELIMITER ;
```

It updates the value of total sales when the associated order goes from the rejected state to the paid state.

```
DELIMITER $$
create trigger insert_sales_purchases_per_package after insert ON orders
for each row
begin
declare packageId2 int;
declare monthlyFee2 float;
declare validityPeriod2 int;
                                                    select
                                                                packageId,
                                                                monthlyFee,
                                                                validityPeriod
        into
        packageId2, monthlyFee2, validityPeriod2
                                                    from period where new.periodId = ID;
                                if new.orderstate <=> "Paid" then
                                                    update total_sales_per_package
                                                    set totalSales = totalSales + monthlyFee2*validityPeriod2,
                                                                totalSalesWithOptionalProduct = totalSalesWithOptionalProduct + new.totalFee
                                                    where packageId = packageId2;
    end if;
end$$

DELIMITER ;
```

It updates the value of total sales when the associated order is paid at the first try.

Average number of sales per optional product with each service package(1)

```
create table `average_sales_optionalproduct_per_servicepackage`(

`packageId` int NOT NULL AUTO_INCREMENT,

`averageOptionalProducts` float NOT NULL DEFAULT 0,

PRIMARY KEY (`packageId`),

CONSTRAINT `average_sales_optionalproduct_per_servicepackage_servicepackage` FOREIGN KEY (`packageId`) REFERENCES `servicepackage` (`packageId`)

)
```

It contains the average number of optional products for each service package sold.

```
DELIMITER $$

create trigger insert_new_average_sales_optionalproduct_per_servicepackage after insert ON servicepackage

for each row

begin

                insert into average_sales_optionalproduct_per_servicepackage(packageId, averageOptionalProducts) values (new.packageId, 0);

end$$


DELIMITER ;
```

It initialize the number of average optional products sold with each service package to 0, when a new service package is created.

Average number of sales per optional product with each service package(2)

```
DELIMITER $$
create trigger update_average_sales_optionalproduct_per_servicepackage after update ON orders
for each row
begin

declare packageId2 int;
declare totalPackage int;
declare totalProducts int;
                select packageId into packageId2 from period where new.periodId = ID;
    select count(*) into totalPackage from orders where periodId in (select ID from period where packageId = packageId2) and orderState <=> "Paid";
    select count(*) into totalProducts from order_optionalproduct where order_Id in (select orderId from orders where periodId in (select ID from period where packageId = packageId2)  and orderState <=> "Paid");
                if  not(old.orderState <=> new.orderstate) and new.orderstate <=> "Paid" then
                        update average_sales_optionalproduct_per_servicepackage
                        set averageOptionalProducts = totalProducts/totalpackage
                        where packageId = packageId2;
    end if;
end$$

DELIMITER ;
```

It updates the average number of optional products sold with a package when the associated order state goes from the rejected state to the paid state.

```
DELIMITER $$

create trigger insert_average_sales_optionalproduct_per_servicepackage after insert ON orders
for each row
begin
declare packageId2 int;
declare totalPackage int;
declare totalProducts int;
                select packageId into packageId2 from period where new.periodId = ID;
    select count(*) into totalPackage from orders where periodId in (select ID from period where packageId = packageId2) and orderState <=> "Paid";
    select count(*) into totalProducts from order_optionalproduct where order_Id in (select orderId from orders where periodId in (select ID from period where packageId = packageId2)  and orderState <=> "Paid");
                if new.orderstate <=> "Paid" then
                        update average_sales_optionalproduct_per_servicepackage
                        set averageOptionalProducts = totalProducts/totalpackage
                        where packageId = packageId2;
    end if;
end$$

DELIMITER ;
```

It updates the average number of optional products sold with a service package when the associated order is paid at the first try (it works when no additional optional product is bought).

Average number of sales per optional product with each service package(3)

```
DELIMITER $$

create trigger insert_bridge_average_sales_optionalproduct_per_servicepackage after insert ON order_optionalproduct
for each row
begin
declare packageId2 int;
declare totalPackage int;
declare totalProducts int;
                select packageId into packageId2 from period where  ID = (select periodId from orders where orderId = new.order_id);
    select count(*) into totalPackage from orders where periodId in (select ID from period where packageId = packageId2) and orderState <=> "Paid";
    select count(*) into totalProducts from order_optionalproduct where order_Id in (select orderId from orders where periodId in (select ID from period
where packageId = packageId2)  and orderState <=> "Paid");
                if (select orderState from orders where orderId = new.order_id) <=> "Paid" then
                        update average_sales_optionalproduct_per_servicepackage
                        set averageOptionalProducts = totalProducts/totalpackage
                        where packageId = packageId2;
    end if;
end$$

DELIMITER ;
```

It updates the average number of optional products sold with a service package when the associated order is paid at the first try (it works when at least one additional optional product is bought).

List of insolvent users

```
create table `insolvent_users`(
`username` varchar(64) NOT NULL,
PRIMARY KEY (`username`),
CONSTRAINT `insolvent_users_users` FOREIGN KEY (`username`) REFERENCES `users` (`username`)
)
```

It contains the users that failed at least one payment.

```
DELIMITER $$
create trigger insert_insolvent_users after update ON users
for each row
begin
        if not(new.isInsolvent <=> old.isInsolvent) and new.isInsolvent <=> 1 then
        insert into insolvent_users(username) values (new.username);
    end if;
end$$

DELIMITER ;
```

It inserts a new user as insolvent if it is not already insolvent (it may happen when you fail a payment twice.

```
DELIMITER $$
create trigger delete_insolvent_users after update ON users
for each row
begin
        if not(new.isInsolvent <=> old.isInsolvent) and new.isInsolvent <=> 0 then
        delete from insolvent_users where username = new.username;
    end if;
end$$

DELIMITER ;
```

It deletes from the table an insolvent user when he is no more insolvent.

List of suspended orders

```
create table `suspended_orders`(
`orderId` int NOT NULL AUTO_INCREMENT,
PRIMARY KEY (`orderId`),
CONSTRAINT `suspended_orders_orders` FOREIGN KEY (`orderId`) REFERENCES `orders` (`orderId`)
)
```

It contains all the suspended orders.

```
DELIMITER $$
create trigger insert_suspended_orders after insert ON orders
for each row
begin
        if new.orderState <=> "Rejected" then
        insert into suspended_orders(orderId) values (new.orderId);
    end if;
end$$

DELIMITER ;
```

It inserts an order when its payment is rejected for the first time.

```
DELIMITER $$
create trigger delete_suspended_orders after update ON orders
for each row
begin
        if new.orderState <=> "Paid" and old.orderState <=>"Rejected" then
        delete from suspended_orders where orderId = new.orderId;
    end if;
end$$

DELIMITER ;
```

It removes an order when it goes from the rejected state to the paid state.

List of Alerts – User Payments (1)

```
CREATE TABLE `alert` (
  `alertId` int NOT NULL AUTO_INCREMENT,
  `username` varchar(64) NOT NULL,
  `amount` float NOT NULL,
  `lastRejectionDateTime` timestamp NOT NULL,
  `email` varchar(64) NOT NULL,
  PRIMARY KEY (`alertId`),
  CONSTRAINT `username` FOREIGN KEY (`username`) REFERENCES
`users` (`username`)
)
```

It contains the alerts as requested by the specification

```
DELIMITER $$
create trigger update_failed_attempt after update ON orders
for each row
begin
declare attempts int;
declare email2 varchar(64);
                    select FailedAttempts,email into attempts,email2 from users where username;
                    if new.orderState <=> "Rejected" and attempts <=> 2 then
                    insert into alert(username,amount,lastRejectionDateTime,email) values
(new.username,new.totalFee,current_timestamp(),email2);
   end if;
   if new.orderState <=>"Rejected" then
   update users
   set FailedAttempts = attempts + 1, isInsolvent = 1 where username = new.username;
   end if;
end$$

DELIMITER;
```

When a payment of an order fails from the second time on, it updates the number of failed attempts in the user table, it sets the user as insolvent and it creates an Alert if the user has 3 failed attempts (2 in the past + 1 now).

List of Alerts – User Payments (2)

```
DELIMITER $$
create trigger delete_insolvence after delete ON suspended_orders
for each row
begin
declare username2 varchar(64);
declare unpaid int;
                select username into username2 from orders where orderId = old.orderId limit 1;
   select count(*) into unpaid from orders where username = username2 and orderState =
"Rejected";
                if unpaid <=> 0 then
   update users
   set isInsolvent = 0 , FailedAttempts = 0 where username = username2;
   end if;
end$$

DELIMITER ;
```

When an user pays all its suspended orders it resets is failed attempts counter and set the user as not insolvent.

```
DELIMITER $$
create trigger insert_failed_attempt after insert ON orders
for each row
begin
declare attempts int;
declare email2 varchar(64);
                select FailedAttempts,email into attempts,email2 from users where username = new.username limit 1;
                if new.orderState <=> "Rejected" and attempts <=> 2 then
                insert into alert(username,amount,lastRejectionDateTime,email) values
(new.username,new.totalFee,current_timestamp(),email2);
   end if;
   if new.orderState <=>"Rejected" then
   update users
   set FailedAttempts = attempts + 1, isInsolvent = 1 where username = new.username;
   end if;
end$$

DELIMITER ;
```

When a payment of an order fails from the first time, it updates the number of failed attempts in the user table, it sets the user as insolvent and it creates an Alert if the user has 3 failed attempts (2 in the past + 1 now).

Best seller optional product

```
create table `bestseller_optionalproduct`(
`ID` int NOT NULL AUTO_INCREMENT,
`productID` int,
`sales` int,
PRIMARY KEY (`ID`),
CONSTRAINT `bestseller_optionalproduct_optionalproduct` FOREIGN KEY (`productID`) REFERENCES `optionalproduct` (`productID`)
)
```

It is a table with just one row. It has the best seller among all the optional products.

```
DELIMITER $$
create trigger insert_bestseller_optionalproduct after insert ON order_optionalproduct
for each row
begin
declare optionalproduct_Id2 int;
declare total int;
SELECT optionalproduct_Id, count(*)
into optionalproduct_Id2, total
    FROM order_optionalproduct
    GROUP BY optionalproduct_Id
    ORDER BY count(*) DESC
    LIMIT 1;
                update bestseller_optionalproduct
                set productID = optionalproduct_Id2, sales = total where ID = 1;
end$$

DELIMITER ;
```

When the bridge table between order and optional product has a new row, and therefore when there is a new order with an optional product, it computes the best seller among the optional products.

Service activation scheduler service

```
CREATE TABLE `activation_scheduler_service` (
  `activationId` int NOT NULL AUTO_INCREMENT,
  `serviceId` int NOT NULL,
  `startTime` timestamp NOT NULL,
  `endTime` timestamp NOT NULL,
  `username` varchar(64) NOT NULL,
  PRIMARY KEY (`activationId`),
  KEY `activation_scheduler_service_serviceId` (`serviceId`),
  KEY `activation_scheduler_service_username` (`username`),
  CONSTRAINT `activation_scheduler_service_serviceId` FOREIGN KEY (`serviceId`) REFERENCES `service` (`serviceId`),
  CONSTRAINT `activation_scheduler_service_username` FOREIGN KEY (`username`) REFERENCES `users` (`username`)
)
```

It is the activation scheduler for the services as requested by the specification.

```
DELIMITER $$
create trigger insert_scheduler_service after insert ON orders
for each row
begin

                    insert into activation_scheduler_service(serviceId,startTime,endTime,username) select
    servicepackage_service.service_id, orders.startTime, orders.endTime, orders.username
                    from servicepackage_service join period on servicepackage_service.servicepackage_id = period.packageId
    join orders on orders.periodId = period.ID where orders.orderState = "Paid" and orders.orderId = new.orderId;
end$$

DELIMITER ;
```

It creates the service activation scheduler record for each service involved when the corresponding order is paid at the first try.

```
DELIMITER $$
create trigger update_scheduler_service after update ON orders
for each row
begin
                    insert into activation_scheduler_service(serviceId,startTime,endTime, username) select
    servicepackage_service.service_id, orders.startTime, orders.endTime, orders.username
                    from servicepackage_service join period on servicepackage_service.servicepackage_id = period.packageId
    join orders on orders.periodId = period.ID where orders.orderState = "Paid" and orders.orderId = new.orderId
    and old.orderState = "Rejected";
end$$

DELIMITER ;
```

It creates the service activation scheduler record for each service involved when the corresponding order is paid from the second try on.

Service activation scheduler optional product

```
CREATE TABLE `activation_scheduler_optionalproduct` (
  `activationId` int NOT NULL,
  `productId` int NOT NULL,
  `startTime` timestamp NOT NULL,
  `endTime` timestamp NOT NULL,
  `username` varchar(64) NOT NULL,
  PRIMARY KEY (`activationId`),
  CONSTRAINT `activation_scheduler_optionalproduct_productId` FOREIGN KEY (`productId`) REFERENCES `optionalproduct` (`productID`),
  CONSTRAINT `activation_scheduler_optionalproduct_username` FOREIGN KEY (`username`) REFERENCES `users` (`username`)
)
```

It is the activation scheduler for the optional products as requested by the specification.

```
DELIMITER $$
create trigger insert_scheduler_optionalproduct after insert ON orders
for each row
begin
                insert into activation_scheduler_optionalproduct(productId,startTime,endTime,username) select
    servicepackage_optionalproduct.optionalproduct_productID, orders.startTime, orders.endTime, orders.username
                from servicepackage_optionalproduct join period on servicepackage_id = period.packageId
                join orders on orders.periodId = period.ID where orders.orderState = "Paid" and orders.orderId = new.orderId;

end$$

DELIMITER ;
```

It creates the service activation scheduler record for each optional product involved when the corresponding order is paid at the first try.

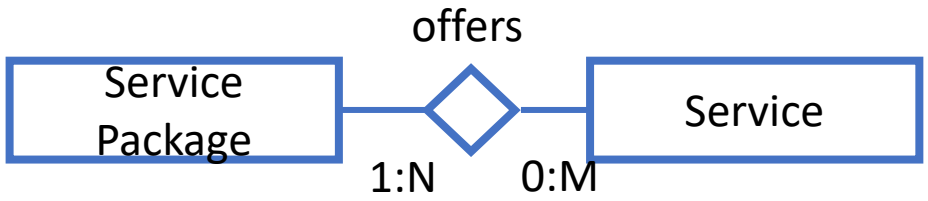
```
DELIMITER $$
create trigger update_scheduler_optionalproduct after update ON orders
for each row
begin
                insert into activation_scheduler_optionalproduct(productId,startTime,endTime,username) select
    servicepackage_optionalproduct.optionalproduct_productID, orders.startTime, orders.endTime, orders.username
                from servicepackage_optionalproduct join period on servicepackage_id = period.packageId
                join orders on orders.periodId = period.ID where orders.orderState = "Paid" and orders.orderId = new.orderId
    and old.orderState = "Rejected";
end$$

DELIMITER ;
```

It creates the service activation scheduler record for each optional product involved when the corresponding order is paid from the second try on.

ORM relationship design with explanations

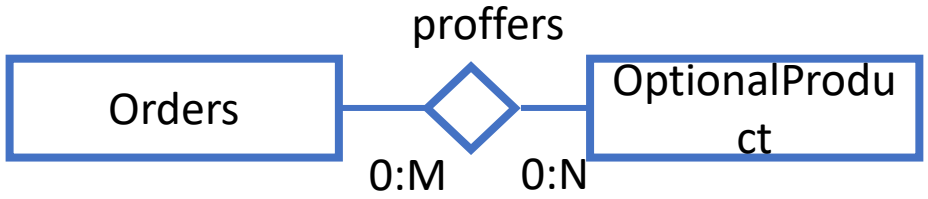
Relationship Service Package “offers” Service



ServicePackage -> Service
@ManyToMany (Fetch Eager, cascade None) is necessary to know what services are offered by a service package.

Service -> ServicePackage
@ManyToMany (Fetch Lazy, cascade Persist) is not requested by the specification, but it is mapped for simplicity and for potential future purpose.

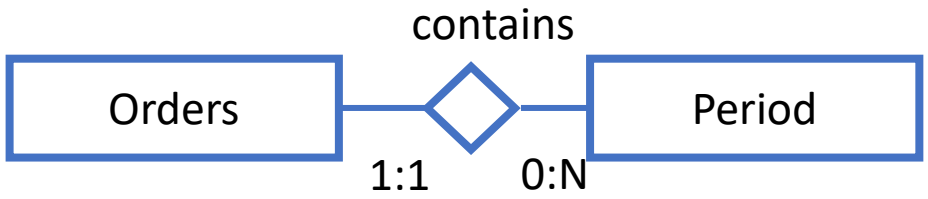
Relationship Orders “proffers” OptionalProduct



Order -> OptionalProduct
@ManyToMany (Fetch Eager, Cascade None) is necessary to know what optional product are part of the order.

OptionalProduct -> Order
@ManyToMany (Fetch Lazy, Cascade Persist, Merge, Refresh, Remove) is requested by the specification.

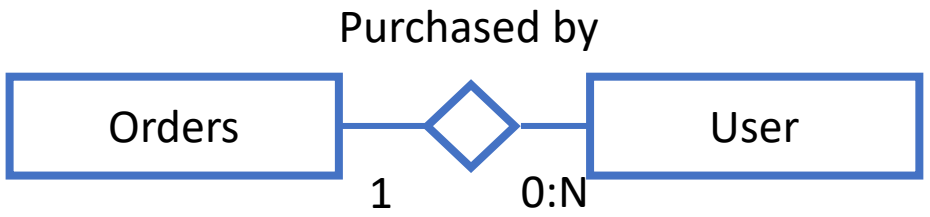
Relationship Orders “contains” Period



Order -> Period
@OneToMany (Fetch Eager, Cascade None) is required to know which is the service package offered by the order and with which validity period.

Period -> Order
@ManyToOne (Fetch Lazy, Cascade Persist) is requested by the specification.

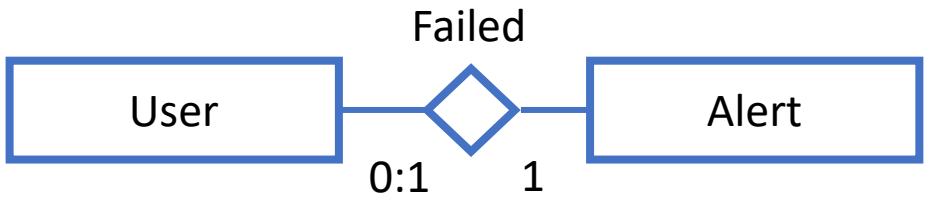
Relationship Orders “purchased by” User



User -> Order
@ManyToOne (Fetch Eager, Cascade Persist) is required to know which orders have been purchased by the user

Order -> User
@OneToMany (Fetch Eager, Cascade None) is not requested by the specification, but it is mapped for simplicity and for potential future purpose.

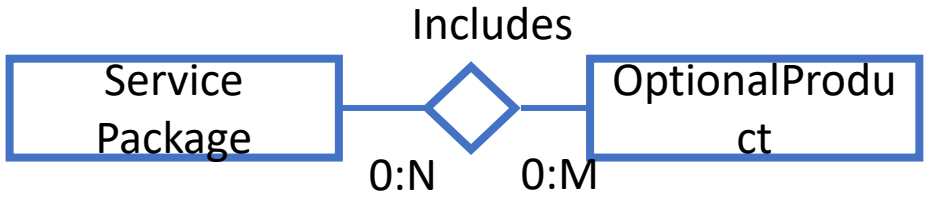
Relationship User “Failed” Alert



User -> Alert
@OneToOne (Fetch Eager, Cascade All) is requested by the specification.

Alert -> User
@OneToOne (Fetch Eager, Cascade None) is not requested by the specification, but it is mapped for simplicity and for potential future purpose.

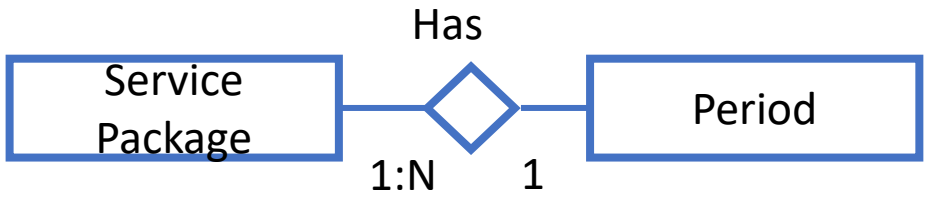
Relationship Service Package “Includes” OptionalProduct



ServicePackage -> OptionalProduct
@ManyToMany (Fetch Eager, Cascade None) is requested by the specification.

OptionalProduct -> ServicePackage
@ManyToMany (Fetch Lazy, Cascade Persist) is not requested by the specification, but it is mapped for simplicity and for potential future purpose.

Relationship Service Package “Has” Period



ServicePackage -> Period
@ManyToOne (Fetch Eager, Cascade Persist) is requested by the specification.

Period -> ServicePackage
@OneToMany (Fetch Eager, Cascade None is requested by the specification.

Entities code

Alert Entity

```
@Entity
@Table(name = "alert")
@NamedQueries({
	@NamedQuery(name = "AlertEntity.getAlerts", query = "SELECT a FROM AlertEntity a"),
})
public class AlertEntity {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "alertID", nullable = false)
   private int alertID;

   @Column(name = "amount", nullable = true)
   private float amount;

   @Column(name = "lastRejectionDateTime", nullable = true)
   private Timestamp lastRejectionDateTime;

   @Column(name = "email", nullable = false, length=64)
   private String email;

   @OneToOne
   @JoinColumn(name = "username")
   private UserEntity relatedUser;
```

Average Sales Optional Product per Service Package Entity

```
@Entity
@Table(name = "average_sales_optionalproduct_per_servicepackage")
@NamedQueries({
      @NamedQuery(name = "AverageSalesOptionalProductPerServicePackageEntity.getAverageSales", query = "SELECT s FROM AverageSalesOptionalProductPerServicePackageEntity s"),
})
public class AverageSalesOptionalProductPerServicePackageEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "packageId")
   private ServicePackageEntity associatedPackage;

   @Column(name = "averageOptionalProducts", nullable = true)
   private float averageOptionalProducts;
```

Best Seller Optional Product Entity

```
@Entity
@Table(name = "bestseller_optionalproduct")
@NamedQueries({
      @NamedQuery(name = "BestsellerOptionalProductEntity.getBestsellerProduct", query = "SELECT o FROM BestsellerOptionalProductEntity o"),
})
public class BestsellerOptionalProductEntity {

   @Id
   @Column(name = "id", nullable = false)
   private int ID;

   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "productID")
   private OptionalProductEntity optionalProduct;

   @Column(name = "sales", nullable = false)
   private int sales;
```

Employee Entity

```
@Entity
@Table(name = "employee")
@NamedQueries({
    @NamedQuery(name = "EmployeeEntity.checkCredentials", query = "SELECT e FROM EmployeeEntity e WHERE e.id = :id AND e.password = :password")
})

public class EmployeeEntity {

   @Id
   @Column(name = "id", nullable = false, length=64)
   private String id;

   @Column(name = "password", nullable = false, length=64)
   private String password;
```

Insolvent Users Entity

```
@Entity
@Table(name = "insolvent_users")
@NamedQueries({
      @NamedQuery(name = "InsolventUsersEntity.getInsolventUsers", query = "SELECT u FROM InsolventUsersEntity u"),
})
public class InsolventUsersEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "username")
   private UserEntity user;
```

Optional Product Entity

```
@Entity
@Table(name = "optionalproduct")


@NamedQueries({
      @NamedQuery(name = "OptionalProductsEntity.getAllOptionalProducts", query = "SELECT o FROM OptionalProductEntity o"),
      @NamedQuery(name = "OptionalProductsEntity.getOptionalProductsByIdAndMonthlyFee", query = "SELECT o FROM OptionalProductEntity o WHERE o.productId = :productId and o.monthlyFee = :monthlyFee"),
})
public class OptionalProductEntity {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "productID", nullable = false)
   private int productId;

   @Column(name = "name", nullable = false, length =25)
   private String name;

   @Column(name = "monthlyFee", nullable = false)
   private int monthlyFee;

   @ManyToMany(mappedBy = "optionalProducts", fetch = FetchType.LAZY, cascade = {CascadeType.REMOVE, CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REFRESH})
   List<OrderEntity> orderEntities ;

   @ManyToMany(mappedBy = "optionalProducts", fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST})
   List<ServicePackageEntity> servicePackageEntities ;
```

Orders Entity

```java
@Entity
@Table(name = "orders")
public class OrderEntity {

  // attributes
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "orderId", nullable = false)
  private int orderId;

  @Column(name = "creationDateTime", nullable = false)
  private Timestamp creationDateTime;

  @Column(name = "totalFee", nullable = false)
  private float totalFee;

  @Column(name = "startTime", nullable = false)
  private Timestamp startTime;

  @Column(name = "endTime", nullable = false)
  private Timestamp endTime;

  @Column(name = "orderState", nullable = false)
  @Enumerated(EnumType.STRING)
  private OrderState orderState;

  // foreign keys
  @ManyToOne
  @JoinColumn(name = "username")
  UserEntity user;

  @ManyToMany(fetch = FetchType.EAGER )
  @JoinTable(name = "order_optionalproduct",
        joinColumns = {
              @JoinColumn(name = "order_id", referencedColumnName = "orderId"),
        },
        inverseJoinColumns = {@JoinColumn(name = "optionalproduct_id", referencedColumnName = "productId")})
  List<OptionalProductEntity> optionalProducts ;

  @ManyToOne(fetch = FetchType.EAGER)
  @JoinColumn(name = "periodId")
  PeriodEntity associatedPeriod;
```

Period Entity

```
@Entity
@Table(name = "period")
public class PeriodEntity {

   // columns
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "ID")
   private int periodId;

   @Column(name = "validityPeriod")
   private int validityPeriod;

   @Column(name = "monthlyFee")
   private float monthlyFee;

   // foreign keys
   @ManyToOne()
   @JoinColumn(name = "packageId")
   private ServicePackageEntity servicePackage;    // on service package table

   @OneToMany(mappedBy = "associatedPeriod", fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST} )
   List<OrderEntity> orders;   // on order table
```

Service Entity

```
@Entity
@Table(name = "service")
@NamedQueries({
      @NamedQuery(name = "ServiceEntity.getAllServices", query = "SELECT s FROM ServiceEntity s"),
})
public class ServiceEntity {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "serviceId", nullable = false)
   private int serviceId;

   @Column(name = "serviceType", nullable = false)
   @Enumerated(EnumType.STRING)
   private ServiceType serviceType;

   @Column(name = "includedMinutes", nullable = true)
   private int IncludedMinutes;

   @Column(name = "feeMinutes", nullable = true)
   private int FeeMinutes;

   @Column(name = "includedSms", nullable = true)
   private int IncludedSMS;

   @Column(name = "feeSms", nullable = true)
   private int FeeSMS;

   @Column(name = "includedGb", nullable = true)
   private int IncludedGB;

   @Column(name = "feeGb", nullable = true)
   private int FeeGB;

   @ManyToMany(mappedBy = "services", fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST})
   List<ServicePackageEntity> servicePackageEntities ;
```

Service Package Entity

```
@Entity
@Table(name = "servicepackage")

@NamedQueries({
    @NamedQuery(name = "ServicePackageEntity.getAllPackages", query = "SELECT p FROM ServicePackageEntity p")
})
public class ServicePackageEntity {

  // attributes
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "packageId")
  private int packageId;

  @Column(name = "Name", nullable = false, length =64)
  private String name;

  // foreign keys
  @OneToMany(mappedBy = "servicePackage", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
  List<PeriodEntity> periods;


  @ManyToMany(fetch = FetchType.EAGER )
  @JoinTable(name = "servicepackage_service",
  joinColumns = {
            @JoinColumn(name = "servicepackage_id", referencedColumnName = "packageId"),
  },
  inverseJoinColumns = {@JoinColumn(name = "service_id", referencedColumnName = "serviceId")})
  List<ServiceEntity> services;

  @ManyToMany(fetch = FetchType.EAGER )
  @JoinTable(name = "servicepackage_optionalproduct",
      joinColumns = {
            @JoinColumn(name = "servicepackage_id", referencedColumnName = "packageId"),
    },
      inverseJoinColumns = {@JoinColumn(name = "optionalproduct_productID", referencedColumnName = "productId")})
  List<OptionalProductEntity> optionalProducts;

  @OneToMany(mappedBy = "associatedPackage", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
  List<TotalPurchasesPerPackageEntity> associatedTotalPurchasesPerPackage;

  @OneToMany(mappedBy = "associatedPackage", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
  List<TotalSalesPerPackageEntity> associatedTotalSalesPerPackage;

  @OneToMany(mappedBy = "associatedPackage", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
  List<AverageSalesOptionalProductPerServicePackageEntity> associatedAverageSalesOptionalProduct;
```

Suspended Orders Entity

```
@Entity
@Table(name = "suspended_orders")
@NamedQueries({
      @NamedQuery(name = "SuspendedOrdersEntity.getSuspendedOrders", query = "SELECT o FROM SuspendedOrdersEntity o"),
})
public class SuspendedOrdersEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "orderId")
   private OrderEntity order;
```

Total Purchases per Package Entity

```
@Entity
@Table(name = "total_purchases_per_package")

@NamedQueries({
      @NamedQuery(name = "TotalPurchasesPerPackageEntity.getAllPurchasesPerPackage", query = "SELECT p FROM TotalPurchasesPerPackageEntity p"),
})
public class TotalPurchasesPerPackageEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "packageId")
   private ServicePackageEntity associatedPackage;

   @Column(name = "total_purchases", nullable = true)
   private int totalPurchases;
```

Total Purchases per Package Validity Period Entity

```
@Entity
@Table(name = "total_purchases_per_package_validityperiod")
@NamedQueries({
    @NamedQuery(name = "TotalPurchasesPerPackageValidityPeriodEntity.getAllPurchasesPerPackageValidityPeriod", query = "SELECT p FROM TotalPurchasesPerPackageValidityPeriodEntity p"),
})
public class TotalPurchasesPerPackageValidityPeriodEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "periodId")
   private PeriodEntity associatedPeriod;

   @Column(name = "total_purchases", nullable = true)
   private int totalPurchases;
```

Total Sales Per Package Entity

```
@Entity
@Table(name = "total_sales_per_package")
@NamedQueries({
    @NamedQuery(name = "TotalSalesPerPackageEntity.getTotalSalesPerPackage", query = "SELECT s FROM TotalSalesPerPackageEntity s"),
})
public class TotalSalesPerPackageEntity {

   @Id
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "packageId")
   private ServicePackageEntity associatedPackage;

   @Column(name = "totalsales", nullable = true)
   private int totalSales;


   @Column(name = "totalsalesWithOptionalProduct", nullable = true)
   private int totalSalesWithOptionalProduct;
```

User Entity

```
@Entity
@Table(name = "users")

@NamedQueries({
      @NamedQuery(name = "UserEntity.checkCredentials", query = "SELECT u FROM UserEntity u WHERE u.username = :username AND u.password = :password"),
      @NamedQuery(name = "UserEntity.findByEmail", query = "SELECT u FROM UserEntity u WHERE u.email = :email")
})
public class UserEntity {

   @Id
   @Column(name = "username", nullable = false, length=64)
   private String username;

   @Column(name = "password", nullable = false, length=64)
   private String password;

   @Column(name = "email", nullable = false, length=64)
   private String email;

   @Column(name = "isInsolvent", nullable = false)
   private boolean isInsolvent;

   @Column(name = "FailedAttempts", nullable = false)
   private int failedAttempts;

   @OneToOne(mappedBy = "relatedUser", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
   private AlertEntity alert;

   @OneToMany(mappedBy = "user", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
   List<OrderEntity> orderEntities ;

   @OneToMany(mappedBy = "user", fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST} )
   List<InsolventUsersEntity> associatedInsolventUser ;
```

List of components

User Views (Client Tier)

- buyservice.html
- confirmation.html
- home.html
- Index.html
- payorder.html

Employee Views (Client Tier)

- home.html
- index.html
- sales.html

Servlets (Web Tier)

- AdminCreateOptionalServlet
- AdminCreatePackageServlet
- AdminHomePageServlet
- AdminSalesServlet
- BuyServicePageServlet
- ConfirmationPageServlet
- HomePageServlet
- LoginEmployeeServlet
- LoginServlet
- LogoutServlet
- PayOrderPageServlet
- SignUpServlet

EJBs (Business tier)

- AlertService (@Stateless)
  - List<AlertEntity> getAlerts
- EmployeeService (@Stateless)
  - EmployeeEntity checkCredentials(String id, String password)
- OptionalProductService (@Stateless)
  - ArrayList<OptionalProductEntity> getAllOptionalProducts()
  - OptionalProductEntity getOptionalProduct(String productId)
  - List<OptionalProductEntity> getListOptionalProducts(List<String> productIdList)
  - void persistOptionalProduct(OptionalProductEntity optionalProductEntity)
- OrderService (@Stateless)
  - OrderEntity findOrderById(int orderId)
  - void persistOrder(OrderEntity order)
  - void updateOrderOnState(OrderEntity order)
- PeriodService (@Stateless)
  - PeriodEntity getPeriodById(int periodId)
  - void persistPeriod(PeriodEntity period)
  - void persistPeriods(ArrayList<PeriodEntity> validityPeriods)
- ServicePackageService (@Stateless)
  - ArrayList<ServicePackageEntity> getAllPackages()
  - ServicePackageEntity getPackageById(int packageId)
  - void persistServicePackage(ServicePackageEntity p)
- ServiceService (@Stateless)
  - ArrayList<ServiceEntity> getAllServices()
  - ServiceEntity getService(String serviceId)
  - List<ServiceEntity> getListServices(List<String> servicesList)
- UserService (@Stateless)
  - UserEntity checkCredentials(String username, String password)
  - UserEntity findUserByUsername(String username)
  - UserEntity findUserByEmail(String email)
  - UserEntity addNewUser(String username, String password, String email)

EJBs (Materialized Views)

- AverageSalesOptionalProductPerServicePackageService (@Stateless)
  - List<AverageSalesOptionalProductPerServicePackageEntity> getAverageSales()
- BestSellerOptionalProductService (@Stateless)
  - BestsellerOptionalProductEntity getBestsellerProduct()
- InsolventUsersService (@Stateless)
  - List<InsolventUsersEntity> getInsolventUsers()
- SuspendedOrdersService (@Stateless)
  - List<SuspendedOrdersEntity> getSuspendedOrders()
- TotalPurchasesPerPackageService (@Stateless)
  - List<TotalPurchasesPerPackageEntity> getTotalPurchasesPerPackage()
- TotalPurchasesPerPackageValidityPeriodService (@Stateless)
  - List<TotalPurchasesPerPackageValidityPeriodEntity> getTotalPurchasesPerPackageValidityPeriod()
- TotalSalesPerPackageService (@Stateless)
  - List<TotalSalesPerPackageEntity> getTotalSalesPerPackage()

Entities (Data Tier)

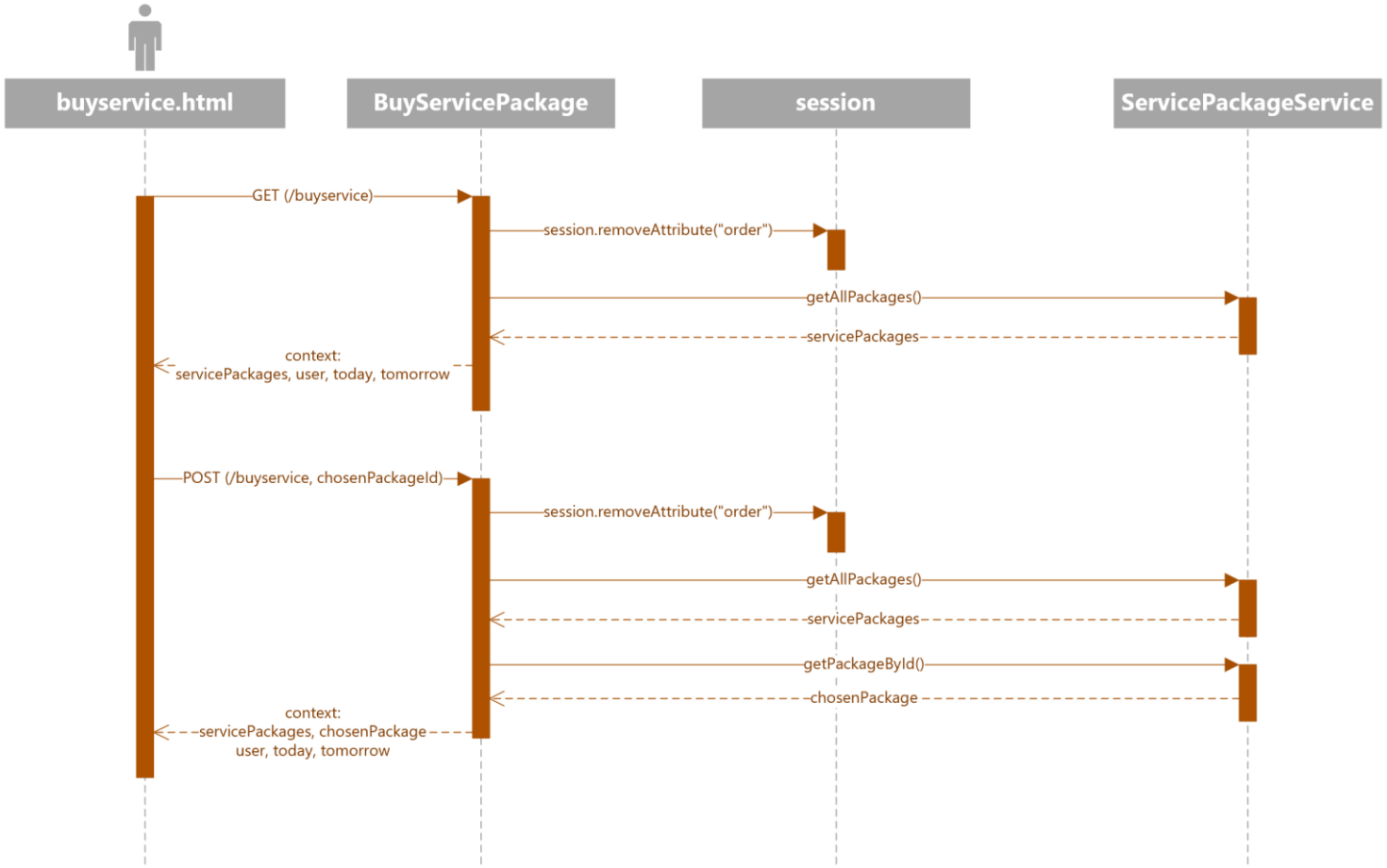
- AlertEntity
- EmployeeEntity
- OptionalProductEntity
- OrderEntity
- PeriodEntity
- ServiceEntity
- ServicePackageEntity
- UserEntity

Entities (Materialized Views)

- AverageSalesOptionalProductPerServicePackageEntity
- BestsellerOptionalProductEntity
- InsolventUsersEntity
- SuspendedOrdersEntity
- TotalPurchasesPerPackageEntity
- TotalPurchasesPerPackageValidityPeriodEntity
- TotalSalesPerPackageEntity

ULM Sequence diagrams

Customer application: purchase of a service package (1)



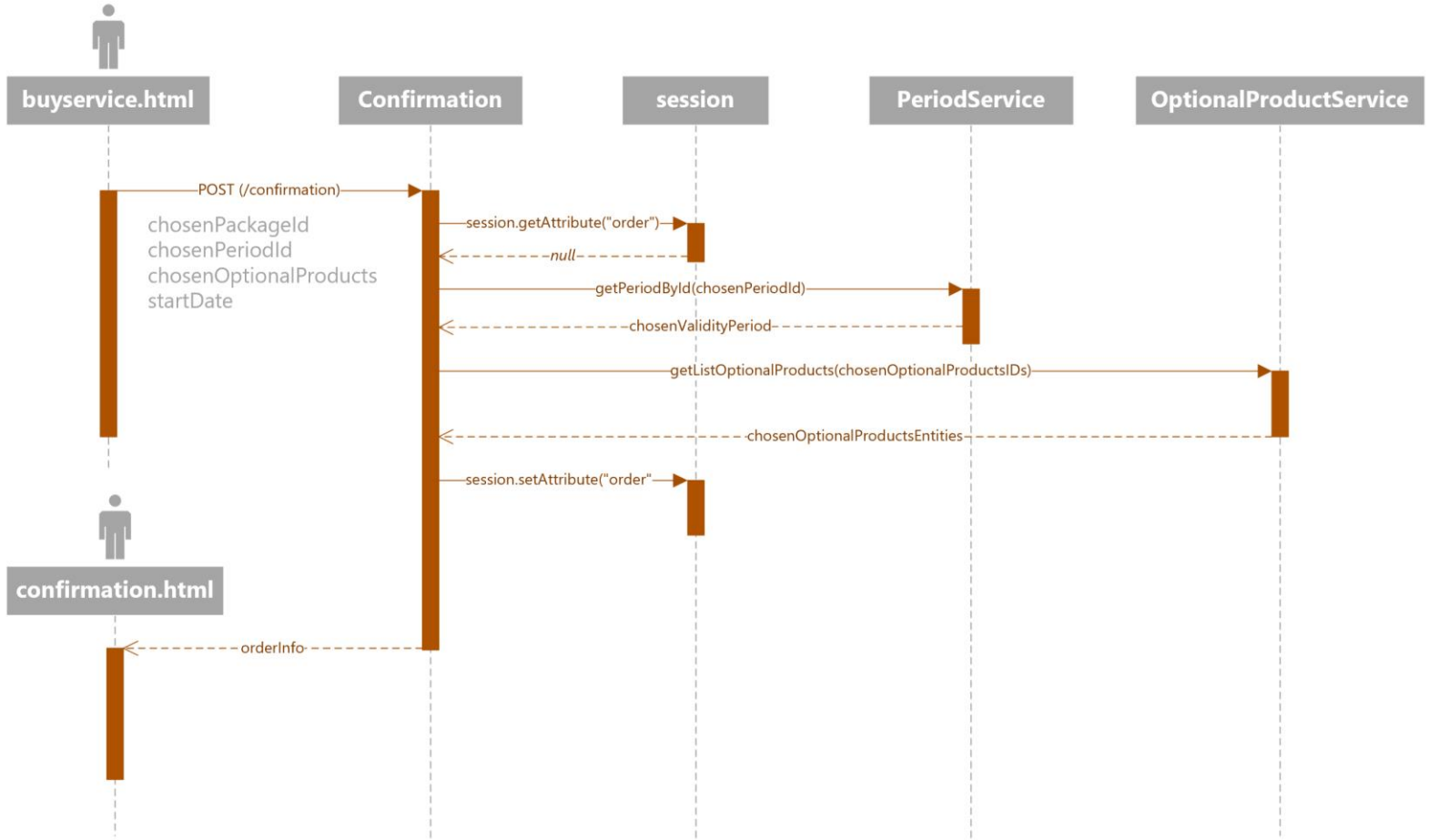
IMPLEMENTATION NOTES:

The ***BuyServicePackage*** controller needs to be called twice.

At the first time, all the service packages are retrieved and sent to the **buyservice.html** page.
Then, when the user selects one service package to buy, its ID is sent to the controller, which retrieves the associated information and sends it back to the web page. This allows the user to select a validity period and the optional products associated with the selected package.

The attribute «order» which is potentially associated with previous purchases is also removed from the session.

ULM Sequence diagrams
Customer application: purchase of a service package (2)

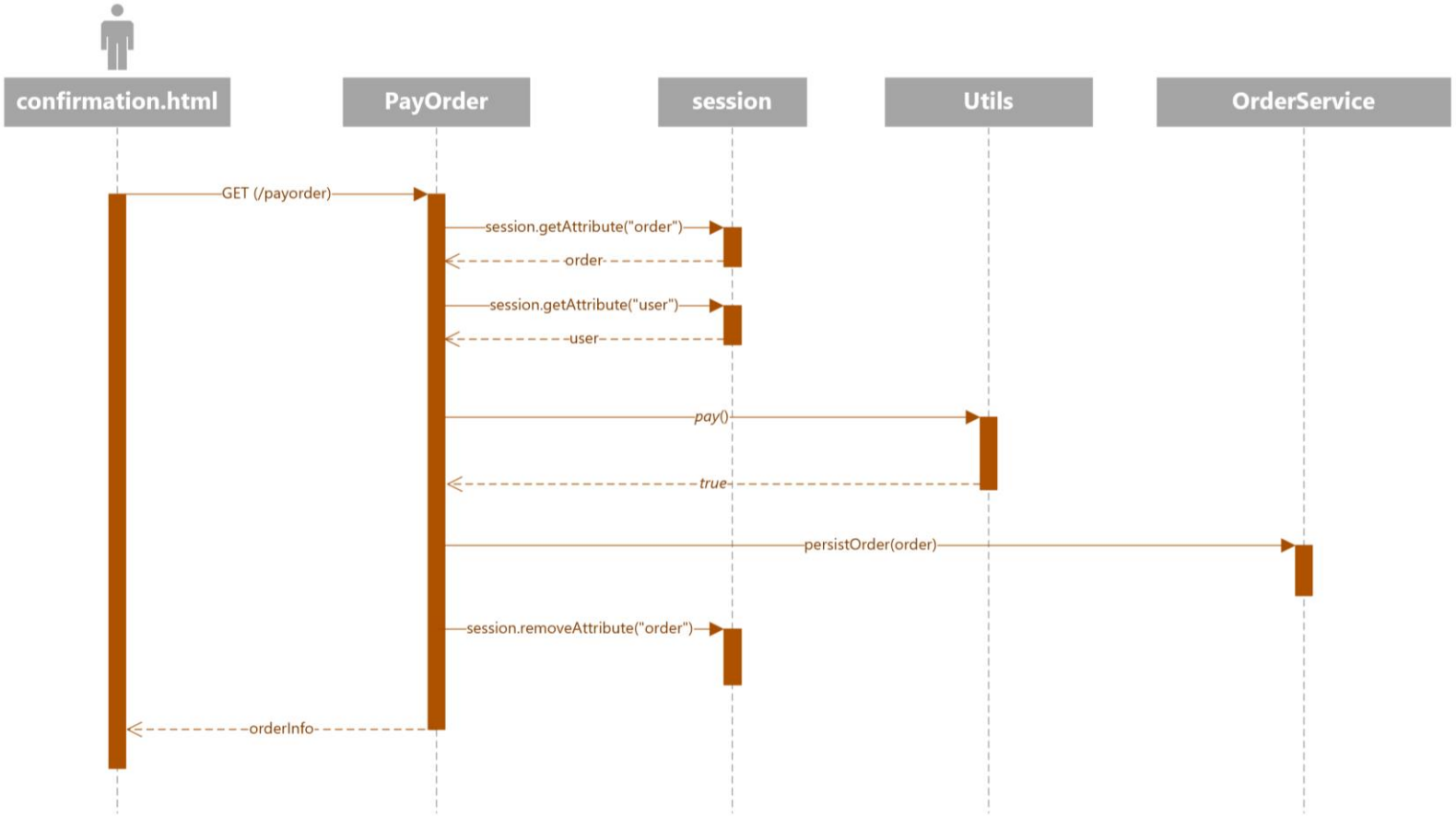


IMPLEMENTATION NOTES:

In this example, since this is a simulation of a new purchase and it's not an attempt to pay a previously failed order, the method ***getAttribute()*** returns null.

For this reason, an **OrderEntity** object is created within the ConfirmationPage controller and it's sent back to the **confirmation.html** page

ULM Sequence diagrams
Customer application: purchase of a service package (3)



IMPLEMENTATION NOTES:

The **OrderEntity** object is modified within the ***PayOrder*** controller with the new information (creation time, order state) and it is sent back to the confirmation.html page, which displays the payment status (accepted or rejected) to the user.

In this example, the order has been successfully paid.

ULM Sequence diagrams

Employee application: creation of a service package