Data bases 2
Project Documentation

Prof. Davide Martinenghi

May 2022

Valentina Sona *Davide Rigamonti*

POLITECNICO
MILANO 1863

Contents

Chapter 1

Specifications

This project was developed as part of the **Data bases 2** course at *Politecnico di Milano a.a. 2021/2022* with the supervision of prof. Davide Martinenghi.

1.1 Overview

The goal of this project is to create two applications (one dedicated to consumers, the other to employees) for a telecommunication-oriented company that offers prepaid online services to web users.

Particular focus was given to the database-application interactions and to the database automated procedures as triggers.

For the development of the project we chose to utilize:

- **MySQL** as the relational SQL DBMS for storing and accessing data.
- **Java** in the form of **JPA**, **EJB** and **Jakarta EE** for database communication, server-side operations and client communication through REST APIs.
- **Maven** for organizing dependencies and simplifying the build process.
- **HTML**, **CSS** and **JavaScript** in order to build a dynamic and interactive client application.
- **LaTeX** for designing the requested documentation.

The external tools used to aid the development process were:

- **IntelliJ IDEA** and **Apache TomEE** for the project development and deployment.
- **Github** as a versioning and collaborative editing tool.
- **draw.io** for designing various types of diagrams.

It's important to note that some of the presented technologies were strongly recommended during the course.

Due to the fact that IntelliJ IDEA was used as the main IDE there are some differences with respect to the Eclipse projects provided as examples.

The most notable difference is the fact that an IntelliJ project is composed of different modules, therefore it's possible to have a *JPA module* and a *Web Application module* working together without the need to split them in different projects, this approach is overall less dispersive and it will preferred throughout the development.

1.2 Hypotheses

This section will address various hypotheses made during the project design and development, however, most of them will be presented while maintaining a broad scope over the given specifications, since more specific and implementation-dependant assumptions and choices will be addressed in their dedicated chapters.

Client applications

As stated in the Overview section, particular focus was put in trying to centralize the structure of the project as much as possible, following this logic and in order to maximize code reuse, the two applications (user and employee) share the same project and some utilities.

Despite the fact presented above, there is still a degree of separation between the two since they are stored in independent subdirectories and dedicated filters exist in order to properly manage authentication and authorization, nonetheless, for testing purposes, in each landing page there is a button capable of switching between the user and employee landing pages.

Employee accounts

Since the given specifications do not disclose whether the project should allow employees to register other employees (or register at all) we assumed that employee accounts are created by some application/entity that is external to the scope of this project.

Update and delete operations

Because of the reasons stated above, update and delete operations on existing entities and tables inside the database weren't considered as part of the project.

However, update and delete operations were taken into consideration in the context of defining cascading properties for the entities and in other instances where we thought that it would be an appropriate choice for the sake of scalability.

External service for payment

Since the given specifications point to the fact that the simulated external call must be able to return true or false at will, in the confirmation page the user will be presented with 3 different buttons that proceed to the payment procedure:

- one will always result in a failed payment;
- another one will always make the payment succeed;
- the last one will have a 50% chance of generating a failed or successful payment.

As far as the payment simulation goes, the called function will also wait for a tunable random time delay in order to add a bit of realism.

General project philosophy

Some of the design choices that were made during the preliminary planning phase take scalability into account quite extensively since we argue that it is a desirable property when developing an application or database, especially in the field of telecommunication.

However, not every choice was made with scalability in mind since we believe that sometimes a simple and easy solution can be a better option over a complex alternative.

This will be better explained on a case-by-case basis.

Chapter 2

Conceptual and logical data models

2.1 ER Diagram

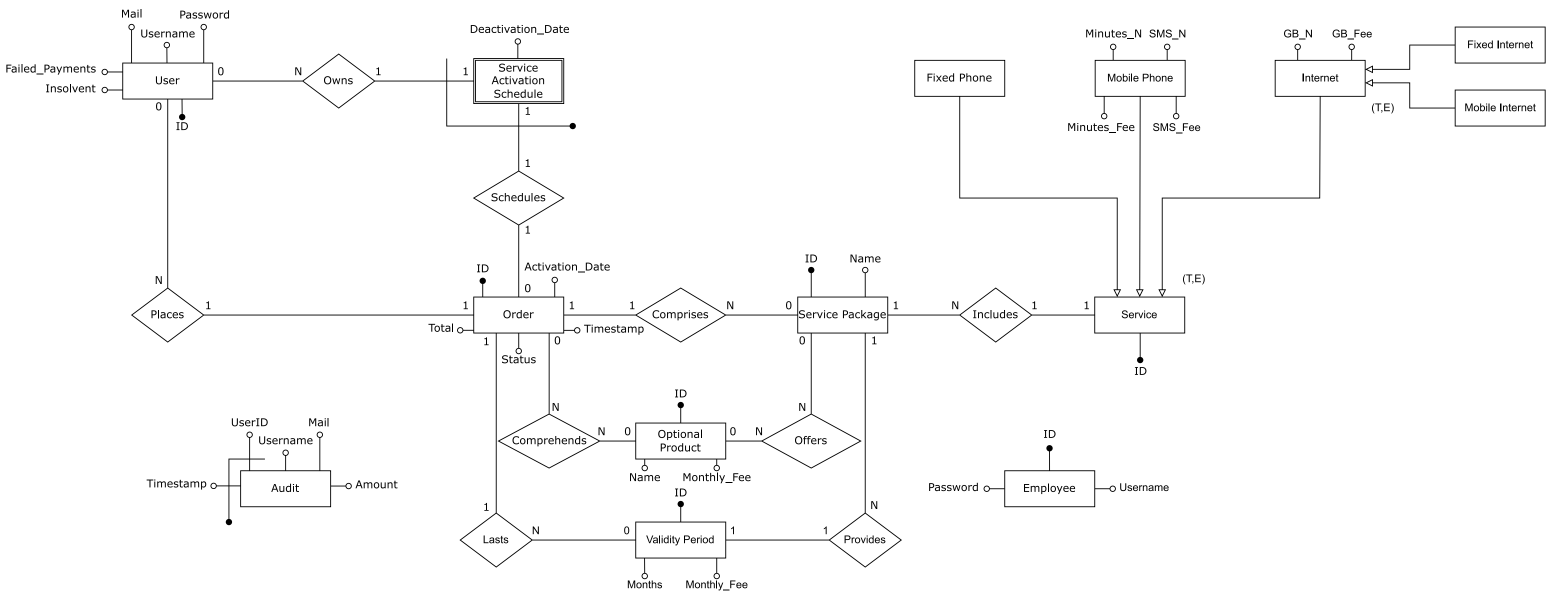

Figure 2.1: ER diagram for the database

The proposed ER diagram reflects the directives indicated in the given specifications.

In some entities we deemed appropriate to add a generated incremental ID as the primary key even if there were other candidate keys available (e.g. the username of the *User* could have been its primary key), this was done in order to simplify foreign key handling with the added benefit of being more maintainable in the long term.

Another doubt that arose during the planning phase of the database was related to the *Fixed Phone* entity since, in the specifications, it was clearly stated that it shouldn't have any particular configuration parameters, therefore modeling it as an entity would mean having an "empty" entity; in the end we decided to stick with this choice for the sake of consistency with the other services and because it is also a scalable solution in case the company needs to add parameters to the service in the future.

The specifications document is not completely transparent regarding the *Service Activation Schedule* and its purpose inside the database: although it is stated that the schedule must include the date of activation and deactivation for the services and the optional products for a given user, we opted to include just a reference to the user, a reference to the order and the deactivation date.

The previous choice was taken in order to reduce redundancy and minimize the overhead in terms of computation time and disk space for each *Order* (in a real case scenario, ideally, the company would have a high number of orders) without compromising the available information, since we can retrieve all of the specified data from the order reference due to the fact that the services and optional products share the order's validity period.

The specifications document explicitly defines the *Validity Periods* available for a certain *Service Package* as 12 months, 24 months and 36 months, however, we found this model quite reductive and decided to generalize the number of possible *Validity Periods* from 1 to N.

It's important to notice that, when an *Employee* tries to create a new *Service Package* from the designated interface, the default *Validity Periods* will be the original three options specified in the document and they will have the ability to add or remove *Validity Periods* to their liking.

2.2 Logical model

What follows here is the definition of the tables in the database. It adheres to the ER diagram proposed above, resolving the ISA of the various service types into three different tables (`Internet, Mobile_Phone, Fixed_Phone`). A hybrid approach has been chosen, simplifying the two kinds of internet services into one, but leaving the other services as separate. This was a compromise between keeping the tables slim and the database scalable and optimizing the complexity away into a single table.

Simple constraints and checks are already enforced at this level, while some more complex one are left to the triggers or to java (as checking that the optionals and validity period selected for a package are indeed in the list of features that package offers).

The code relating to the materialized views is omitted, as they are simple mirrors of the values present in this table and contain no checks or foreign key constraints, and their workings will be described in the trigger section.

As for the foreign key constraint, we have imagined that when a service package is deleted, all the associated services would be deleted as well, but the orders related to it would remain, to keep a trace of the financial movements. Conversely, if a user were to be deleted, we assume that all of their data, including past transaction, is no longer relevant, and delete their orders as well.

```sql
CREATE TABLE IF NOT EXISTS Users (
    ID INT NOT NULL AUTO_INCREMENT,
    Mail varchar(255) UNIQUE NOT NULL,
    Username varchar(255) UNIQUE NOT NULL,
    Password varchar(64) NOT NULL,
    Failed_Payments INT DEFAULT 0 NOT NULL CHECK (Failed_Payments  >= 0),
    Insolvent BOOLEAN DEFAULT FALSE NOT NULL,
    PRIMARY KEY ( ID )
);

```

```
CREATE TABLE IF NOT EXISTS Service_Pkgs (
    ID INT NOT NULL AUTO_INCREMENT,
    Name varchar(255) NOT NULL,
    PRIMARY KEY ( ID )
);

CREATE TABLE IF NOT EXISTS Mobile_Phone (
    ID INT NOT NULL AUTO_INCREMENT,
    Pkg_ID INT NOT NULL,
    Minutes_N INT NOT NULL,
    SMS_N INT NOT NULL,
    Minutes_Fee FLOAT NOT NULL CHECK (Minutes_Fee  >= 0),
    SMS_Fee FLOAT NOT NULL CHECK (SMS_Fee  >= 0),
    PRIMARY KEY ( ID ),
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Internet (
    ID INT NOT NULL AUTO_INCREMENT,
    Pkg_ID INT NOT NULL,
    GB_N INT NOT NULL,
    GB_Fee FLOAT NOT NULL CHECK (GB_Fee  >= 0),
    Fixed BOOLEAN DEFAULT FALSE NOT NULL,
    PRIMARY KEY ( ID ),
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS Fixed_Phone (
    ID INT NOT NULL AUTO_INCREMENT,
    Pkg_ID INT NOT NULL,
    PRIMARY KEY ( ID ),
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Optional_Products (
    ID INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(255) NOT NULL,
    Monthly_Fee FLOAT NOT NULL CHECK (Monthly_Fee  >= 0),
    PRIMARY KEY ( ID )
);

CREATE TABLE IF NOT EXISTS Validity_Periods (
    ID INT NOT NULL AUTO_INCREMENT,
    Pkg_ID INT NOT NULL,
    Months INT NOT NULL CHECK (Months  > 0),
    Monthly_Fee FLOAT NOT NULL CHECK (Monthly_Fee  >= 0),
    PRIMARY KEY ( ID ),
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);


CREATE TABLE IF NOT EXISTS Orders (
    ID INT NOT NULL AUTO_INCREMENT,
    User_ID INT NOT NULL,
    Pkg_ID INT,
```

```sql
    Validity_Period_ID INT,
    Activation_Date DATE,
    Timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    Status ENUM('PENDING', 'FAILED', 'VALID') NOT NULL,
    Total FLOAT NOT NULL CHECK (Total  >= 0),
    PRIMARY KEY ( ID ),
    FOREIGN KEY ( User_ID )
        REFERENCES Users ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE,
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE SET NULL,
    FOREIGN KEY ( Validity_Period_ID )
        REFERENCES Validity_Periods ( ID )
            ON UPDATE NO ACTION ON DELETE SET NULL
);

CREATE TABLE IF NOT EXISTS OrderComprehendsOptional (
    Order_ID INT NOT NULL,
    Optional_ID INT NOT NULL,
    PRIMARY KEY ( Order_ID, Optional_ID ),
    FOREIGN KEY ( Order_ID )
        REFERENCES Orders ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE,
    FOREIGN KEY ( Optional_ID )
        REFERENCES Optional_Products ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS ServicePkgOffersOptional (
    Pkg_ID INT NOT NULL,
    Optional_ID INT NOT NULL,
    PRIMARY KEY ( Pkg_ID, Optional_ID ),
    FOREIGN KEY ( Pkg_ID )
        REFERENCES Service_Pkgs ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE,
    FOREIGN KEY ( Optional_ID )
        REFERENCES Optional_Products ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Employees (
    ID INT NOT NULL AUTO_INCREMENT,
    Username varchar(255) UNIQUE NOT NULL,
    Password varchar(64) NOT NULL,
    PRIMARY KEY ( ID )
);

CREATE TABLE IF NOT EXISTS ServiceActivationSchedule (
    Order_ID INT NOT NULL,
    User_ID INT NOT NULL,
    Deactivation_Date DATE NOT NULL,
    PRIMARY KEY ( Order_ID, User_ID ),
    FOREIGN KEY ( Order_ID )
        REFERENCES Orders ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE,
    FOREIGN KEY ( User_ID )
        REFERENCES Users ( ID )
            ON UPDATE NO ACTION ON DELETE CASCADE
);


CREATE TABLE IF NOT EXISTS Audits (
    User_ID INT NOT NULL,
    Timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
    Mail varchar(255) NOT NULL ,
    Username varchar(255) NOT NULL ,
    Amount FLOAT NOT NULL ,
    PRIMARY KEY ( User_ID , Timestamp )
);
```

Chapter 3

Views and triggers

This section will address the behaviour of the triggers developed for the project, most of them were developed with the aim of operating on the materialized view tables (as per the project specification) but some include additional functionalities that we deliberately chose to implement via triggers such as the handling of the insolvent flag, failed payments, auditing mechanism and the service activation schedule creation.

3.1 SQL View Code

Since most of the triggers are related to the materialized views required for the sales report, this section prefaces the actual implementation with the code that would have been used for the actual views, had they not been materialized. Most of the code can still be reused in the actual triggers and the views share their name with their materialized counterpart for easier tracking.

3.1.1 Number of total purchases per package

```sql
CREATE VIEW PurchasesPerPackage AS
SELECT Name, COUNT(Pkg_ID) AS Purchases
FROM Orders INNER JOIN Service_Pkgs ON Orders.Pkg_ID=Service_Pkgs.ID
GROUP BY Pkg_ID;
```

In the triggers, it is mapped by `NewPackage` and `OrderCompleted`, which respectively create an entry in the `PurchasesPerPackage` table with zero purchases and increment it by one for each successful order containing the package.

3.1.2 Number of total purchases per package and validity period

```sql
CREATE VIEW PurchasePerPackagePeriod AS
SELECT  Name,
        Validity_Periods.Months AS Months,
        COUNT(Orders.Pkg_ID) AS Purchases
FROM Orders
INNER JOIN Service_Pkgs ON Orders.Pkg_ID=Service_Pkgs.ID
INNER JOIN Validity_Periods ON Orders.Validity_Period_ID=Validity_Periods.ID
GROUP BY Orders.Pkg_ID, Validity_Period_ID;
```

Similar to the one above, it requires an extra join to group by the selected validity period as well. In the trigger it is still implemented by `OrderCompleted` in a similar fashion as before, while the creation of the entry in the `PurchasePerPackagePeriod` table is delegated to the trigger `NewPeriod`.

This reflects the fact that the Validity_Periods table is the one containing the join column with respect to the service packages, and therefore validity periods are created after the service package they refer to: the `NewPackage` trigger would not have all the information needed to create an entry yet.

3.1.3 Total value of sales per package with and without the optional products

```sql
CREATE VIEW TotalPerPackage AS
SELECT
    Service_Pkgs.Name as Name,
    SUM(Validity_Periods.Monthly_Fee*Validity_Periods.Months) +
        SUM(Optional_Products.Monthly_Fee* Validity_Periods.Months) as Total,
    SUM(Validity_Periods.Monthly_Fee*Validity_Periods.Months)
        as "Total before optionals"
FROM Orders
    LEFT JOIN Service_Pkgs ON Orders.Pkg_ID=Service_Pkgs.ID
    LEFT JOIN Validity_Periods
        ON Orders.Validity_Period_ID=Validity_Periods.ID
    LEFT JOIN OrderComprehendsOptional
        ON Orders.ID=OrderComprehendsOptional.Order_ID
    LEFT JOIN Optional_Products
        ON OrderComprehendsOptional.Optional_ID=Optional_Products.ID
GROUP BY Orders.Pkg_ID;
```

All of the information included in an order have to be used to calculate the results of this view, making it one of the most complex. The calculation is done from scratch, not using the calculated value in the `Total` column of the `Orders` table as to not complicate the query further (since it is not a value that depends on the GROUP BY column, using it would have required a nested query).

The entry creation in the homonym table is handled by the trigger `NewPackage`, while the updates is found in the `OrderCompleted` trigger. There, having the option of running two separate updates, the one for the complete total uses the value in the `Total` column for ease, while the total before the optionals cost is computed by a simplified version of the query above.

3.1.4 Average number of optional products sold together with each service package

```sql
CREATE VIEW AvgOptPerPackage AS
SELECT Pkg_ID, Name, AVG(OptNum) FROM (
    SELECT Service_Pkgs.Name as Name, Service_Pkgs.ID as Pkg_ID,
    COUNT(coalesce(OrderComprehendsOptional.Optional_ID, 0)) as OptNum
        FROM Orders
            INNER JOIN Service_Pkgs ON Orders.Pkg_ID=Service_Pkgs.ID
            LEFT JOIN OrderComprehendsOptional
                ON Orders.ID=OrderComprehendsOptional.Order_ID
        GROUP BY Pkg_ID, Orders.ID) as Numbers
GROUP BY Pkg_ID;
```

The query uses a nested select statement to return an intermediate table where each row reflects an order and the number of optionals it includes. The coalesce function is used to account for the NULL value that packages with no optionals would have in the join table. The average is then computed on the values of this table.

The triggers relating to this table are the same ones as above. The query is reused pretty much untouched inside the `OrderCompleted` trigger to update to the new value the entry for each service package.

3.1.5 List of insolvent users, suspended orders and alerts

```sql
CREATE VIEW InsolventUsers AS
SELECT * FROM Users
WHERE Insolvent=True;

CREATE VIEW RejectedOrders AS
SELECT * FROM Orders
WHERE Status='Failed';
```

The views are pretty simple select statements. A view for the alerts has not been created because it would coincide exactly with the whole `Audits` table.

All of the views are implemented as conditional statements in the `OrderFailed` trigger, while deletion of records from the tables are inside the `OrderCompleted` trigger.

3.1.6 Best seller optional product

```sql
CREATE VIEW BestSellerOptional AS
SELECT
    Optional_Products.Name as Name,
    COUNT(OrderComprehendsOptional.Order_ID) as TotalSales
FROM Optional_Products
    LEFT JOIN OrderComprehendsOptional
        ON OrderComprehendsOptional.Optional_ID=Optional_Products.ID
GROUP BY Optional_Products.ID
ORDER BY TotalSales DESC
LIMIT 1;
```

The view computes the number of sales per optional product and sorts it, returning only the highest record.

Having only one entry at any time, the updating of the materialized view is completely handled by the `OrderCompleted` trigger.

3.2 Triggers

This section illustrates the aforementioned triggers. All of the triggers rely on the `DELIMITER $$` command to change the delimiting character in order to contain multiple statements.

Also, all the triggers here are row level triggers - while this would have been the choice anyway, since each trigger acts on the data of a single record, MySQL does not support statement level triggers, so there was no other alternative.

3.2.1 NewPackage and NewPeriod

```sql
CREATE TRIGGER NewPackage
AFTER INSERT ON Service_Pkgs
FOR EACH ROW
BEGIN
    INSERT INTO PurchasesPerPackage VALUES ( new.ID,  new.Name, 0 );
    INSERT INTO TotalPerPackage VALUES ( new.ID,  new.Name, 0, 0 );
    INSERT INTO AvgOptPerPackage VALUES ( new.ID,  new.Name, 0 );
END $$
```

The last trigger fires on the same condition as the one above: after an update of orders. They have been separated both for readability, as they are the longest triggers in the project already, and for logical reasons: while `OrderCompleted` fires upon an order being marked valid, this one deals with what should happen when a payment is rejected.

```
CREATE TRIGGER NewPeriod
AFTER INSERT ON Validity_Periods
FOR EACH ROW
BEGIN
    INSERT INTO PurchasesPerPackagePeriod  (Pkg_ID, Name, Months, Purchases)
        SELECT Service_Pkgs.ID, Service_Pkgs.Name, new.Months AS Months, 0
        FROM Service_Pkgs WHERE ID = new.PKG_ID;
END $$
```

Both of these packages trigger after an insertion on the respective tables, and behave the same way, creating an entry for the corresponding package or package-period combination inside the affected tables.

Consequently, these triggers fire only when an employee is configuring new packages, and only once per package. They must act for each new record inserted, hence the for each row. Since if the trigger is firing the package is new, all counts are set to zero.

The trigger `NewPeriod` also implicitly enforces the constraint that no package has two validity periods of the same length, as trying to insert one such period would raise a duplicate primary key error on the `PurchasesPerPacakgePeriod` table.

3.2.2 OrderCompleted

```
CREATE TRIGGER OrderCompleted
AFTER UPDATE ON Orders
FOR EACH ROW
BEGIN
    IF new.Status='VALID' AND old.STATUS<>'VALID' THEN
        DELETE FROM SuspendedOrders WHERE ID=new.ID;
        INSERT INTO ServiceActivationSchedule
            VALUES(new.ID, new.User_ID,
            TIMESTAMPADD(MONTH, (SELECT Months FROM Validity_Periods
                WHERE ID=new.Validity_Period_ID ),
            new.Activation_Date)
        );

        UPDATE PurchasesPerPackage set Purchases = Purchases + 1
        WHERE Pkg_ID=new.Pkg_ID;

        UPDATE PurchasesPerPackagePeriod set Purchases = Purchases + 1
        WHERE Pkg_ID=new.Pkg_ID AND Months =
            ( SELECT Months FROM Validity_Periods WHERE ID=new.Validity_Period_ID );

        UPDATE TotalPerPackage SET TotalPerPackage.Total =
            TotalPerPackage.Total + new.Total WHERE Pkg_ID=new.Pkg_ID;

        UPDATE TotalPerPackage
            SET TotalPerPackage.TotalBeforeOptionals =
                TotalPerPackage.TotalBeforeOptionals +
                ( SELECT Validity_Periods.Monthly_Fee*Validity_Periods.Months
                    FROM Orders
                    LEFT JOIN Validity_Periods
```

```
                        ON Orders.Validity_Period_ID=Validity_Periods.ID
                        WHERE Orders.ID = new.ID
                ) WHERE Pkg_ID=new.Pkg_ID;

        UPDATE AvgOptPerPackage SET AvgOptPerPackage.AvgOptionals =
            ( SELECT AVG(OptNum) FROM
                (SELECT Orders.ID as ID, Service_Pkgs.ID as Pkg_ID,
                    COUNT(coalesce(OrderComprehendsOptional.Optional_ID, 0))
                        as OptNum
                    FROM Orders
                    INNER JOIN Service_Pkgs ON Orders.Pkg_ID=Service_Pkgs.ID
                    LEFT JOIN OrderComprehendsOptional
                        ON Orders.ID=OrderComprehendsOptional.Order_ID
                    GROUP BY Pkg_ID, Orders.ID) as Numbers
                GROUP BY Pkg_ID HAVING Pkg_ID=new.Pkg_ID
            ) WHERE Pkg_ID=new.Pkg_ID;


        DELETE FROM BestSellerOptional;

        INSERT INTO BestSellerOptional (ID, Name, TotalSales) SELECT
            Optional_Products.ID as ID, Optional_Products.Name as Name,
            COUNT(OrderComprehendsOptional.Order_ID) as TotalSales
            FROM Optional_Products
            LEFT JOIN OrderComprehendsOptional
                ON OrderComprehendsOptional.Optional_ID=Optional_Products.ID
            GROUP BY Optional_Products.ID
            ORDER BY TotalSales DESC LIMIT 1;


        IF (
            SELECT COUNT(*) FROM Orders WHERE
                Status='FAILED' AND User_ID=new.User_ID ) = 0
        THEN
            UPDATE Users SET Insolvent = FALSE, Failed_Payments = 0
                WHERE ID=new.User_ID;
            DELETE FROM InsolventUsers WHERE ID=new.User_ID;
        END IF;
    END IF;
END $$
```

This is the biggest trigger in the project, as it handles everything that happens when an order is marked valid. Only orders that have been paid for are taken into account for the statistics shown in the sales report, therefore this trigger handles the insertion of data in most of the materialized views, and the removal of data from the list of rejected orders and insolvent users.

Since `BestSellerOptional` always contains only one row, the trigger deletes the whole table before computing the new best seller.

Furthermore, the trigger handles the `ServiceActivationSchedule` table, as this was the most natural way to implement the insertion of data into that table.

3.2.3 OrderFailed

```
CREATE TRIGGER OrderFailed
AFTER UPDATE ON Orders
FOR EACH ROW
BEGIN
    IF new.Status='FAILED' THEN
        IF new.ID NOT IN (SELECT ID FROM SuspendedOrders) THEN
            INSERT INTO SuspendedOrders VALUES (
            new.ID, new.User_ID, new.Pkg_ID, new.Timestamp, new.Status, new.Total);
```

```sql
        ELSE
            UPDATE SuspendedOrders SET Status='FAILED' WHERE ID=new.ID;
        END IF;

        IF (SELECT Insolvent FROM Users WHERE ID=new.User_ID)=FALSE THEN
            UPDATE Users SET Insolvent=TRUE WHERE ID=new.User_ID;
            INSERT INTO InsolventUsers (
            ID, Mail, Username, Failed_Payments, Insolvent)
                SELECT ID, Mail, Username, Failed_Payments, Insolvent
                FROM Users WHERE ID=new.User_ID;
        END IF;

        UPDATE Users SET Failed_Payments = Failed_Payments + 1 WHERE ID=new.User_ID;
        UPDATE InsolventUsers SET Failed_Payments = Failed_Payments + 1
            WHERE ID=new.User_ID;
        IF (SELECT Failed_Payments FROM Users WHERE ID=new.User_ID) = 3 THEN
            INSERT INTO Audits (User_ID, Mail, Username, Amount)
            SELECT ID, Mail, Username, new.Total FROM Users
            WHERE ID=new.User_ID;
        END IF;
    END IF;
END $$
```

Other than handling the aforementioned InsolventUsers and SuspendedOrders, the auditing mechanism is also implemented here: upon a user failing their third payment, an entry is created in the Audits table.

Chapter 4

Object Relational Mapping (ORM)

4.1 ORM relationship design

Relationship 'Owns'

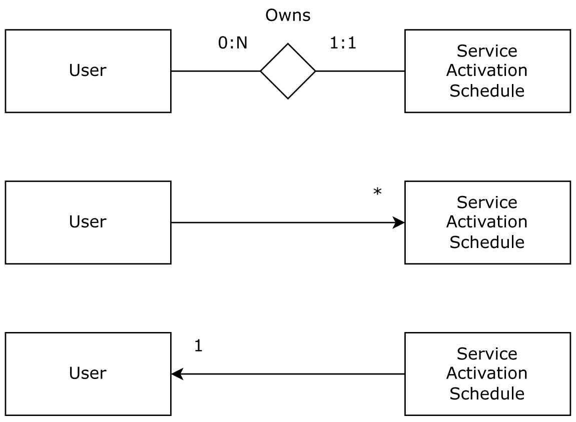
Owns
User
0:N
1:1
Service Activation Schedule
User
*
Service Activation Schedule
User
1
Service Activation Schedule

Figure 4.1: 'Owns' relationship

- **User → ServiceActivationSchedule** `@OneToMany` is necessary to list the activation schedules for a given user; fetch mode is `LAZY` since the activation schedule will not always be consulted when loading a user (e.g. during login procedure).
- **ServiceActivationSchedule → User** `@ManytoOne` is not required by the specification, however, mapping the relationship as bidirectional and using only the needed side is preferable; fetch mode is therefore `LAZY`.

Relationship 'Places'

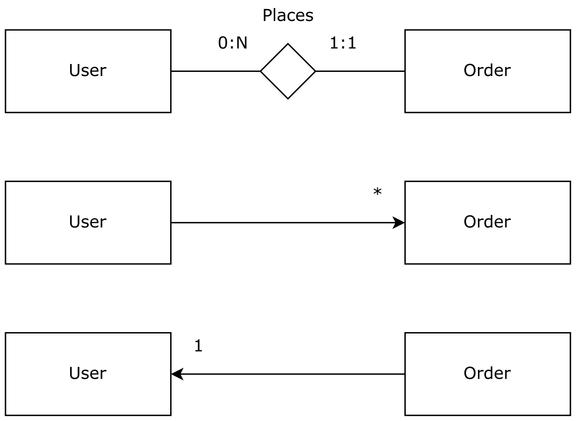

Figure 4.2: 'Places' relationship

- **User → Order** `@OneToMany` is necessary to list the orders that have been made by a user but fetch mode is `LAZY`, since it is unnecessary to load all the orders of a given user every time this user is fetched from the database.
- **Order → User** `@ManyToOne` is necessary since the order has to contain the information about its user; fetch mode is `LAZY` because the information about the user is not always relevant when viewing an order (e.g. listing the rejected orders for a user).

Relationship 'Schedules'

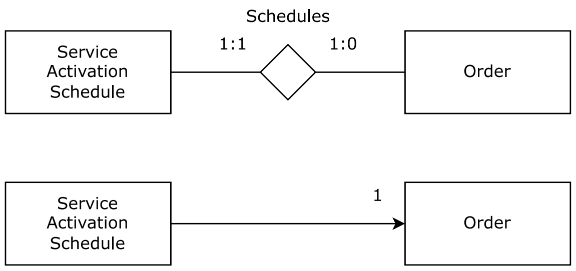

Figure 4.3: 'Schedules' relationship

- **ServiceActivationSchedule → Order** `@OneToOne` is necessary because the schedule includes the service package and the optional packages from the order and fetch mode is `EAGER` because without the order, the schedule information is incomplete.
- The relationship is unidirectional, since the order doesn't ever need to retrieve the activation schedule.

Relationship 'Comprises'

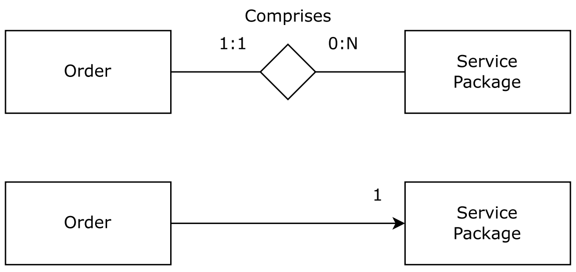

Figure 4.4: 'Comprises' relationship

- **Order → ServicePackage** `@OneToOne` is necessary because the service package contains all the information about the services related to the order; fetch mode is `LAZY` for conservative reasons, however it might also have been `EAGER` but when in doubt it is good practice to pick `LAZY` in order to not load unnecessary information every time an order is retrieved.
- The relationship is unidirectional since the only data required by the dashboard is aggregated and can be derived via a simple GROUP BY query.

Relationship ’Comprehends’

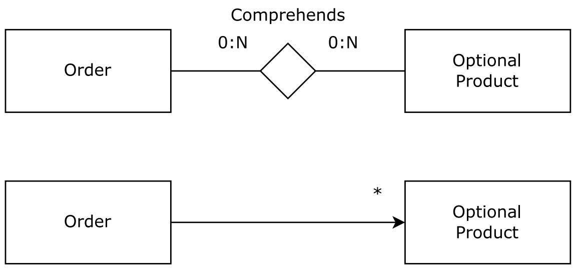

Figure 4.5: ’Comprehends’ relationship

- **Order → OptionalProduct** `@OneToMany` is necessary for each order to contain information about the selected optional products; fetch mode is `LAZY` for the same reasons listed in the previous relationship.
- The relationship is unidirectional since the optional product does not need to know which orders it is part of.
- Since the mapped relationship is `Many to many` the generated join table is called `OrderComprehendsOptional`

Relationship ’Lasts’

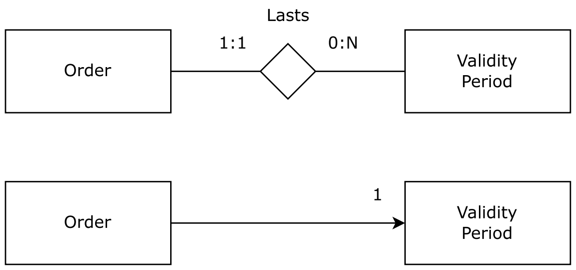

Figure 4.6: ’Lasts’ relationship

- **Order → ValidityPeriod** `@OneToOne` is necessary for each order to be associated with one validity period; fetch mode is `LAZY` because the information is used only inside the summary.
- The relationship is unidirectional since the validity period does not need to know in which orders it is used.

Relationship ’Offers’

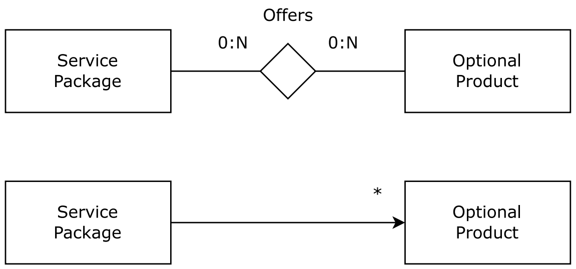

Figure 4.7: ’Offers’ relationship

- **ServicePackage → OptionalProduct** `@OneToMany` is necessary for each service package to be associated with the optional products it includes; fetch mode is `LAZY`, because the information about which optional products are available is relevant only during the subscription and creation phase.
- The relationship is unidirectional since there is no need to actively track in which service packages is offered a given optional product.

Relationship 'Provides'

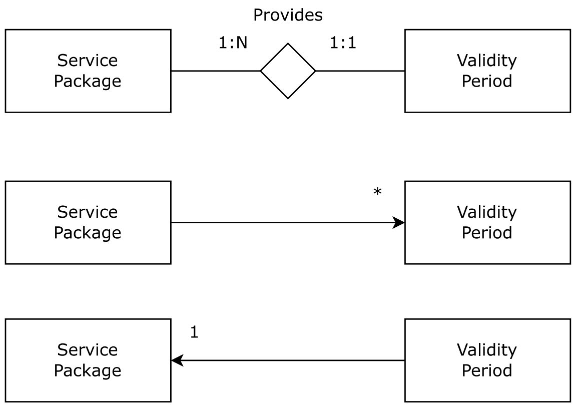

Figure 4.8: 'Provides' relationship

- **ServicePackage → ValidityPeriod**
  `@OneToMany` is necessary for each service package to be associated with the possible validity periods that the user may pick; fetch mode is `LAZY` since the information about which validity periods are available is relevant only during the subscription and creation phase, in addition, the cascading type is set to `PERSIST` in order to automatically create the validity period when we persist the service package.
- `@ManytoOne` is not required by the specification, however, mapping the relationship as bidirectional and using only the needed side is preferable; fetch mode is therefore `LAZY`.

Relationship 'Includes'

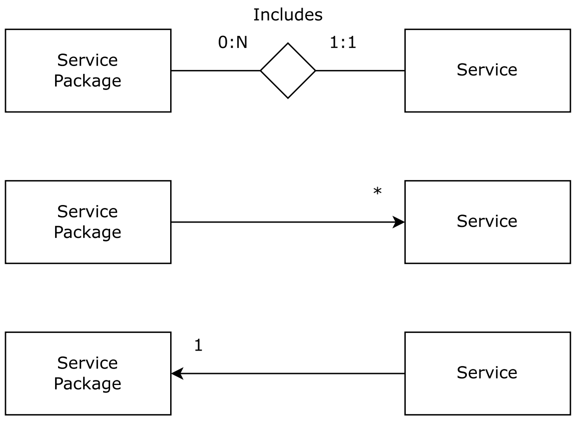

Figure 4.9: 'Includes' relationship

In order to avoid replicating three times the same relationship, the generic `'Service'` represents the three types of services: `MobilePhone`, `FixedPhone` and `Internet`.

- **ServicePackage → Service**
  `@OneToMany` is necessary for each service package to be associated with the services it contains; fetch mode is `EAGER` because this is the main information inside the service package and it would be pointless to load it without its services, in addition, the cascading type is set to `PERSIST` in order to automatically create the services when we persist the service package.
- **Service → ServicePackage**
  `@ManytoOne` is not required by the specification, however, mapping the relationship as bidirectional and using only the needed side is preferable; fetch mode is therefore `LAZY`.

4.2 Java entities

Only the entities that represent proper independent tables inside the database will be reported in this section, since entities derived from materialized views don't have any particular or interesting JPA annotations and have already been extensively covered in the Triggers chapter.

Audit

```java
@Entity
@IdClass(AuditID.class)
```

```java
@Table(name= "Audits", schema = "db2telco")
@NamedQuery(name = "Audit.findAll", query = "SELECT a FROM Audit a")
public class Audit implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "User_ID")
    private int userID;

    @Id
    @Column(name = "Timestamp")
    private Timestamp timestamp;

    @Column(name="Mail")
    private String mail;

    @Column(name="Username")
    private String username;

    @Column(name = "Amount")
    private float amount;

    ...

}
```

Employee

```java
@Entity
@Table(name= "Employees", schema = "db2telco")
@NamedQuery(name = "Employee.findByUsername", query = "SELECT u " +
                                                      "FROM Employee u " +
                                                      "WHERE u.username = :usr"
)
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @Column(name="Username")
    private String username;

    @Column(name="Password")
    @JsonIgnore
    private String password;

    ...

}
```

Fixed phone

```java
@Entity
@Table(name= "Fixed_Phone", schema = "db2telco")
public class FixedPhone implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "Pkg_ID")
    @JsonBackReference
    private ServicePackage servicePackage;

    ...

}
```

Internet

```
@Entity
@Table(name= "Internet", schema = "db2telco")
public class Internet implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "Pkg_ID")
    @JsonBackReference
    private ServicePackage servicePackage;

    @Column(name="GB_N")
    private int gigabytes;

    @Column(name="GB_Fee")
    private float gigabyteFee;

    @Column(name="Fixed")
    private boolean is_fixed;

    ...

}
```

Mobile phone

```
@Entity
@Table(name= "Mobile_Phone", schema = "db2telco")
public class MobilePhone implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "Pkg_ID")
    @JsonBackReference
    private ServicePackage servicePackage;

    @Column(name="Minutes_N")
```

```java
    private int minutes;

    @Column(name="SMS_N")
    private int sms;

    @Column(name="Minutes_Fee")
    private float minuteFee;

    @Column(name="SMS_Fee")
    private float smsFee;

    ...

}
```

Optional product

```java
@Entity
@Table(name= "Optional_Products", schema = "db2telco")
@NamedQuery(name = "OptionalProduct.findAll", query = "SELECT s " +
                                                  "FROM OptionalProduct s"
)
public class OptionalProduct implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @Column(name="Name")
    private String name;

    @Column(name="Monthly_Fee")
    private float fee;

    ...

}
```

Order

```java
@Entity
@Table(name= "Orders", schema = "db2telco")
@NamedQuery(name = "Order.getRejectedOrdersByUser",
            query = "SELECT o FROM Order o " +
                    "WHERE o.user.id = :id AND " +
                    "(o.status = " +
    "it.polimi.db2project_telco.server.entities.util.OrderStatus.FAILED OR " +
                    "o.status = " +
    "it.polimi.db2project_telco.server.entities.util.OrderStatus.PENDING)"
)
public class Order implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "User_ID")
    private User user;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "Pkg_ID")
    private ServicePackage servicePackage;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumnname = "Validity_Period_ID")
    private ValidityPeriod validityPeriod;

    @OneToMany(fetch = FetchType.LAZY)
    @JoinTable(name="OrderComprehendsOptional",
            joinColumns = @JoinColumn(name = "Order_ID"),
            inverseJoinColumns = @JoinColumn(name = "Optional_ID")
    )
    private List<OptionalProduct> optionalProducts;

    @Column(name="Activation_Date")
    private Date activationDate;

    @Column(name="Timestamp")
    private Timestamp timestamp;

    @Column(name="Status")
    @Enumerated(EnumType.STRING)
    private OrderStatus status;

    @Column(name="Total")
    private float total;

    ...

}
```

Service activation schedule

```java
@Entity
@IdClass(ScheduleID.class)
@Table(name= "ServiceActivationSchedule", schema = "db2telco")
public class ServiceActivationSchedule implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "User_ID")
    private User user;

    @Id
    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "Order_ID")
    private Order order;

    @Column(name = "Deactivation_Date")
    private Date deactivationDate;

    ...

}
```

Service package

```java
@Entity
@Table(name= "Service_Pkgs", schema = "db2telco")
@NamedQuery(name = "ServicePackage.findAll", query = "SELECT s " +
                                                     "FROM ServicePackage s"
)
public class ServicePackage implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @Column(name="Name")
    private String name;

    @OneToMany(mappedBy = "servicePackage", fetch = FetchType.EAGER,
               cascade = CascadeType.PERSIST)
    @JsonManagedReference
    private List<FixedPhone> fixedPhoneServices;

    @OneToMany(mappedBy = "servicePackage", fetch = FetchType.EAGER,
               cascade = CascadeType.PERSIST)
    @JsonManagedReference
    private List<MobilePhone> mobilePhoneServices;

    @OneToMany(mappedBy = "servicePackage", fetch = FetchType.EAGER,
               cascade = CascadeType.PERSIST)
    @JsonManagedReference
    private List<Internet> internetServices;

    @OneToMany(mappedBy = "servicePackage", fetch = FetchType.LAZY,
               cascade = CascadeType.PERSIST)
    @JsonManagedReference
    private List<ValidityPeriod> validityPeriods;

    @OneToMany(fetch = FetchType.LAZY)
    @JoinTable(name="ServicePkgOffersOptional",
            joinColumns = @JoinColumn(name = "Pkg_ID"),
            inverseJoinColumns = @JoinColumn(name = "Optional_ID")
    )
    private List<OptionalProduct> optionalProducts;

    ...

}
```

User

```java
@Entity
@Table(name= "Users", schema = "db2telco")
@NamedQuery(name = "User.findByUsername", query = "SELECT u " +
                                                  "FROM User u " +
                                                  "WHERE u.username = :usr"
)
public class User implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @Column(name="Mail")
    private String mail;

    @Column(name="Username")
    private String username;

    @Column(name="Password")
    @JsonIgnore
    private String password;

    @Column(name="Failed_Payments")
    @JsonIgnore
    private int failed_payments;

    @Column(name="Insolvent")
    @JsonIgnore
    private boolean is_insolvent;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<ServiceActivationSchedule> schedules;

    ...

}
```

Validity period

```
@Entity
@Table(name= "Validity_Periods", schema = "db2telco")
public class ValidityPeriod implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "Pkg_ID")
    @JsonBackReference
    private ServicePackage servicePackage;

    @Column(name="Months")
    private int months;

    @Column(name="Monthly_Fee")
    private float fee;

    ...

}
```

Chapter 5

Functional analysis of the specifications

The upcoming IFML diagrams will describe, respectively, customer and employee applications; it is important to keep in mind that the provided diagrams offer just an overview of the functional analysis for the applications, there may be slight mismatches between the documentation and the actual implementation even if quite some effort was put into trying to fit the finished project to the design.

Notes on the design and structure of the front end

The views have been completely developed using pure HTML + CSS + JS with the support of some free templates for the CSS files.

Every view is composed by an HTML and a JS file with the same name, the JS file is stored in a subfolder named `js` together with the JS files for the other views of the application (customer or employee) and another JS file containing some components used by all the views of the application named `components.js`.

Outside the application-specific folder there is a shared folder containing CSS and JS file that are used by all of the views for both applications.

Most of the data related to client-server interactions is stored in server-side sessions but in some cases the client-side browser local storage has also been used for storing less critical data.

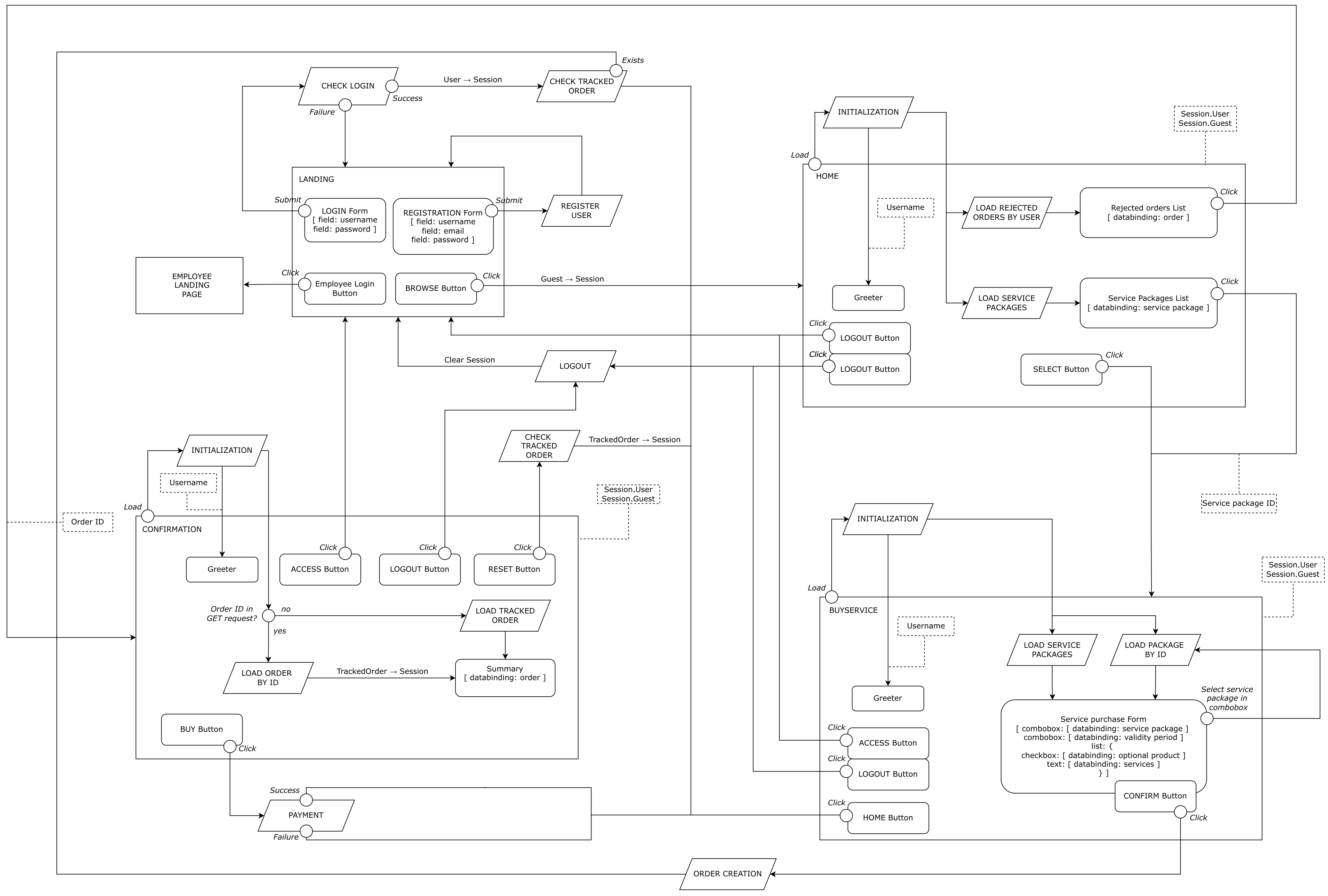

Figure 5.1: IFML diagram for the customer

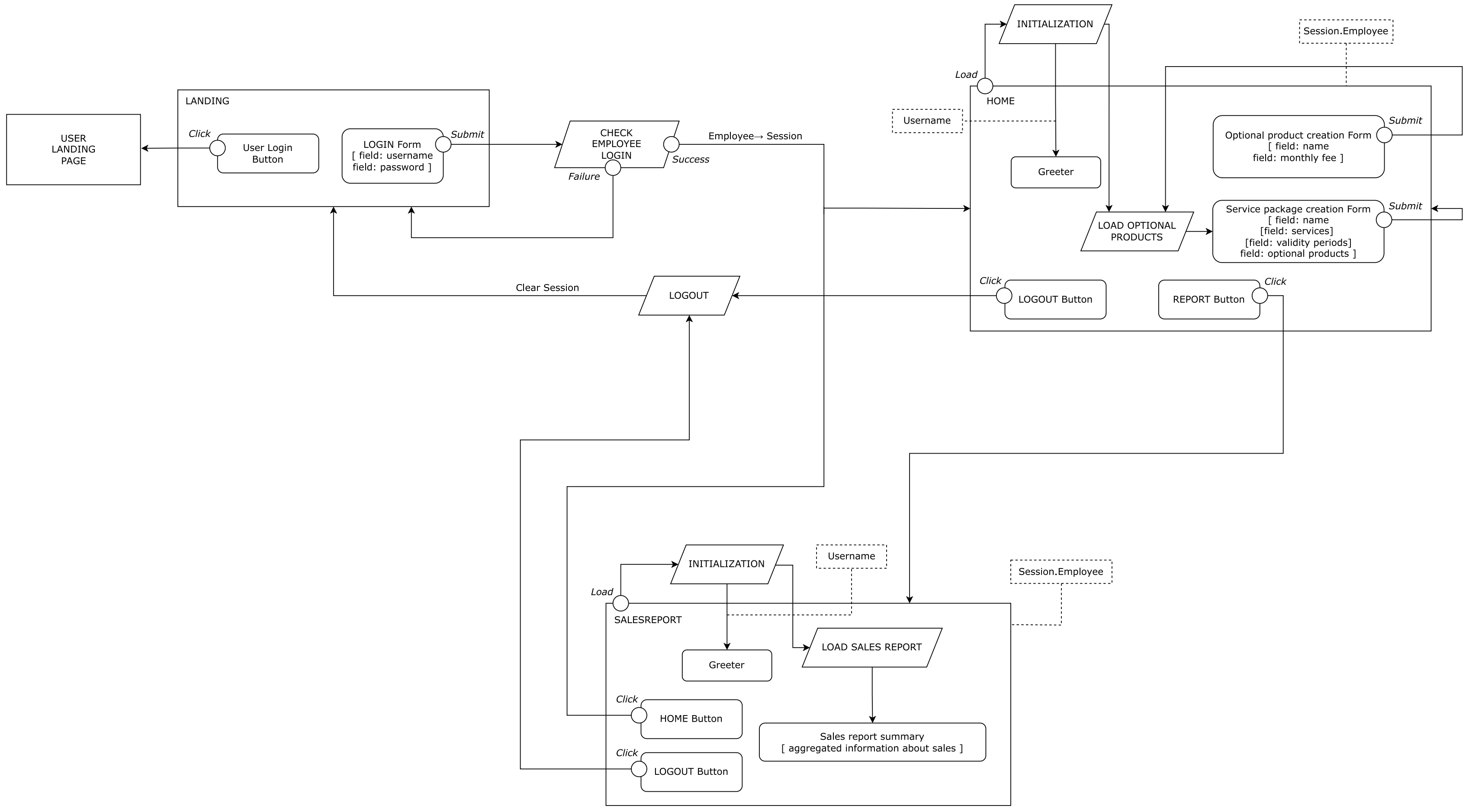

Figure 5.2: IFML diagram for the employee

Chapter 6

List of components

This chapter contains an overview of all the components that are contained in the project subdivided in `front-end` components and `back-end` components.

Notes

As far as the `back-end` components are concerned, we tried to associate a service to each relevant entity defined by the **JPA** mapping; in the case of materialized views, however, we though that it would be more appropriate to have a single service capable of interacting with all the entities dedicated to the *Sales Report* since that is the only purpose of their existence.

6.1 Front end

6.1.1 Views

Customer

Views of the customer are located in the `webapp/user` subfolder.

- **Landing page** (landing.html + landing.js)
- **Home page** (home.html + home.js)
- **Buy service page** (buyservice.html + buyservice.js)
- **Confirmation page** (landing.html + landing.js)

Employee

Views of the employee are located in the `webapp/employee` subfolder.

- **Landing page** (landing.html + landing.js)
- **Home page** (home.html + home.js)
- **Sales report page** (sales.html + buyservice.js)

6.1.2 Controllers (Servlets)

+ 1em

- **CheckEmployeeLogin**
  Verifies the credentials of an employee for the login procedure.
- **CheckLogin**
  Verifies the credentials of a customer for the login procedure.
- **CheckTrackedOrder**
  Verifies the existence of a tracked order in the server-side session.
- **CreateOptionalProduct**
  Creates a new optional product with a name and a fee and stores it in the database.
- **CreateOrder**
  Creates a new order given a set of wanted features storing it in the server-side session as the tracked order.
- **CreateServicePackage**
  Creates a new service package with a name, a set of services, a set of validity periods and a set of optional products and stores it in the database.
- **LoadOptionalProducts**
  Returns all of the optional products present in the database.
- **LoadOrderByID**
  Returns information about an order given its ID and only if the user requesting the order owns it.
- **LoadPackageByID**
  Returns information about a service package given its ID.
- **LoadRejectedOrdersByUser**
  Returns all of the `REJECTED` or `PENDING` orders of the current user.
- **LoadSalesReport**
  Returns aggregate information about sales and users.
- **LoadServicePackages**
  Returns all of the service packages present in the database.
- **LoadTrackedOrder**
  Returns the tracked order from the current session.
- **Logout**
  Invalidates the session.
- **Payment**
  Moves the tracked order from the session to the database, effectively saving it; it also simulates a payment and updates the status of the order accordingly with respect to the outcome of the payment.
- **RegisterUser**
  Registers a new user by adding their credentials to the database.

6.2 Back end

6.2.1 Entities

Entities and their mappings are discussed more in depth inside the ORM chapter.

- **Audit**
- **Employee**
- **FixedPhone**
- **Internet**
- **MobilePhone**
- **OptionalProduct**
- **Order**
- **ServiceActivationSchedule**
- **ServicePackage**
- **ValidityPeriod**
- **User**

Materialized Views

- **AvgOptPerPackage**
- **BestSellerOptional**
- **InsolventUsers**
- **PurchasesPerPackage**
- **PurchasesPerPackagePeriod**
- **RejectedOrders**
- **TotalPerPackage**

6.2.2 Business Components (EJBs)

- **EmployeeService** `@Stateless`
  - `Employee` **`checkCredentials`**`(String username, String password)`
- **OptionalProductService** `@Stateless`
  - `List<OptionalProduct>` **`findAll`**`()`
  - `void` **`createProduct`**`(String name, float fee)`
- **OrderService** `@Stateless` + 1em
  - `Order` **`findByID`**`(int orderID)`
  - `Order` **`composeOrder`**`(int servicePackageID, int validityPeriodID, List<Integer> optionalProductsIDs)`
  - `Order` **`insertNewOrder`**`(Order trackedOrder)`

- void **updateOrderStatus**(Order order, OrderStatus status)
- List<Order> **getRejectedOrders**(int userID)

- **SalesReportService** @Stateless
  - List<PurchasesPerPackage> **findPurchasesPerPackage**()
  - List<PurchasesPerPackagePeriod> **findPurchasesPerPackagePeriod**()
  - List<TotalPerPackage> **findTotalPerPackage**()
  - List<AvgOptPerPackage> **findAvgOptPerPackage**()
  - List<InsolventUsers> **findInsolventUsers**()
  - List<RejectedOrders> **findRejectedOrders**()
  - List<Audit> **findAudits**()
  - List<BestSellerOptional> **findBestSellerOptional**()
- **ServicePackageService** @Stateless
  - List<ServicePackage> **findAll**()
  - ServicePackage **findByID**(int packageID)
  - void **createServicePackage**(ServicePackage servicePackage)
- **UserService** @Stateless
  - User **checkCredentials**(String username, String password)
  - void **createUser**(String mail, String username, String password)

Chapter 7

UML sequence diagrams

7.1 Tracked order recovery

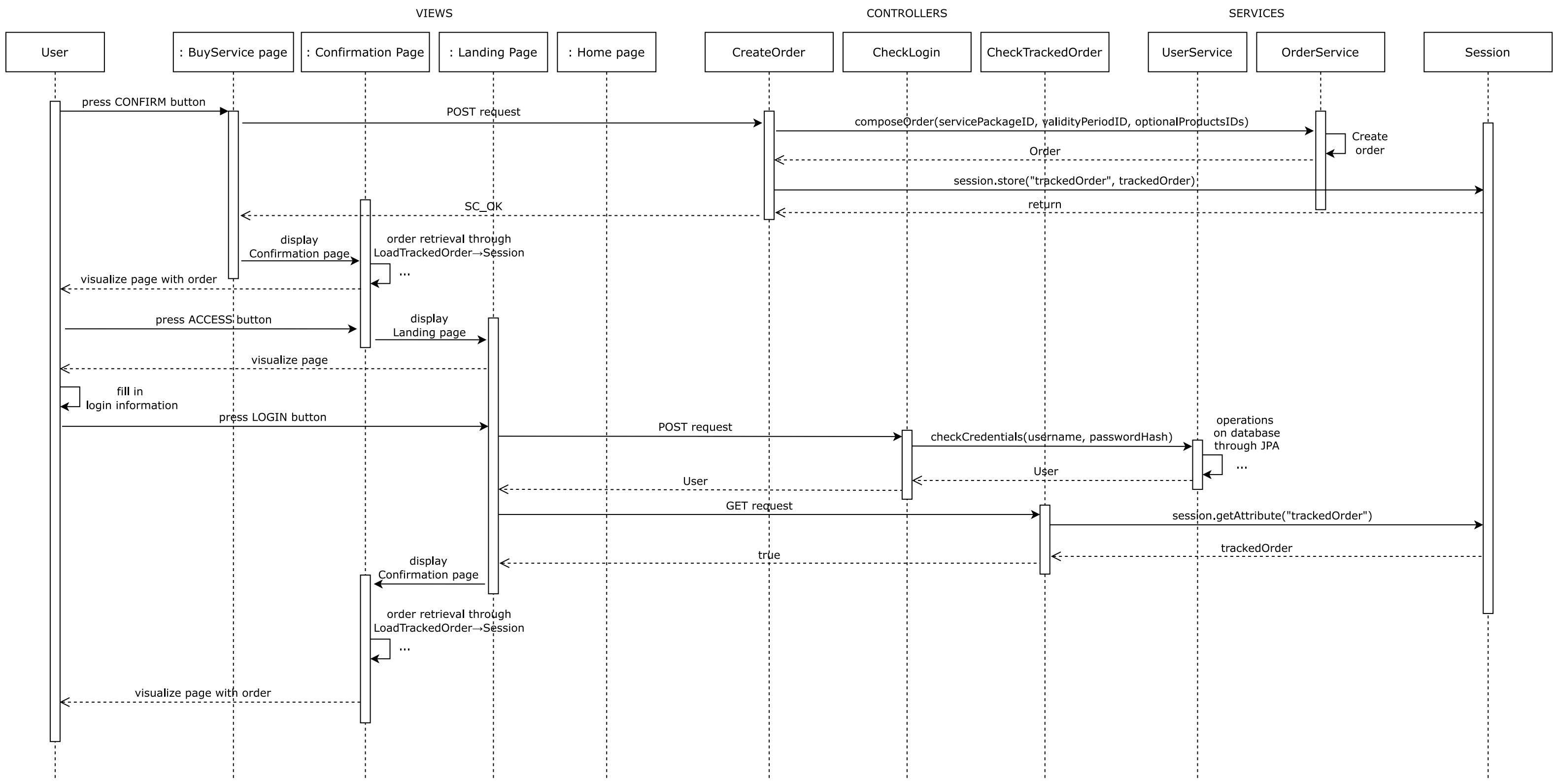
VIEWS
CONTROLLERS
SERVICES
User
: BuyService page
: Confirmation Page
: Landing Page
: Home page
CreateOrder
CheckLogin
CheckTrackedOrder
UserService
OrderService
Session
press CONFIRM button
POST request
composeOrder(servicePackageID, validityPeriodID, optionalProductsIDs)
Create order
Order
session.store("trackedOrder", trackedOrder)
SC_OK
return
display Confirmation page
order retrieval through LoadTrackedOrder→Session
...
visualize page with order
press ACCESS button
display Landing page
visualize page
fill in login information
press LOGIN button
POST request
checkCredentials(username, passwordHash)
operations on database through JPA
...
User
User
GET request
session.getAttribute("trackedOrder")
trackedOrder
true
display Confirmation page
order retrieval through LoadTrackedOrder→Session
...
visualize page with order

Figure 7.1: UML sequence diagram for recovery of tracked order after login

Notes

Some interactions between the views and the servlet were left out and marked with '...' for brevity purposes.

7.2 Employee creation services

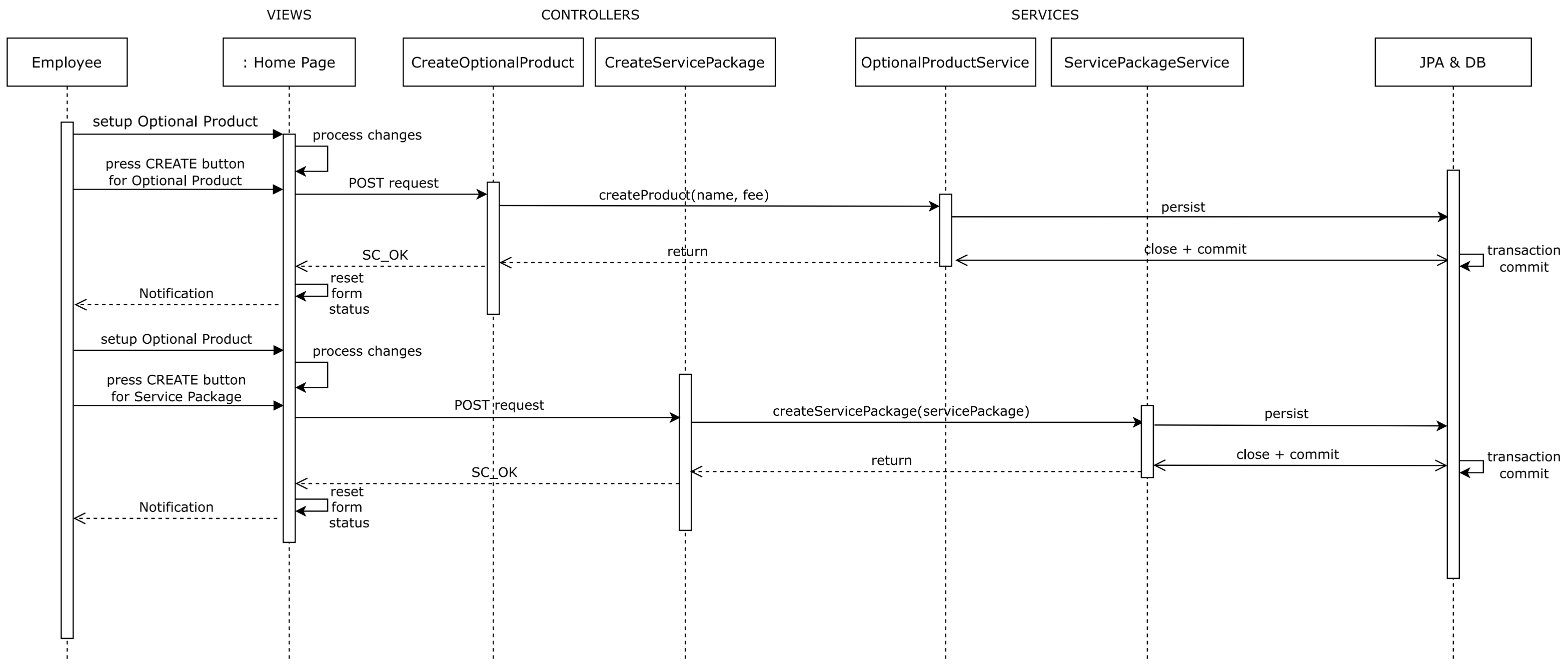

Figure 7.2: UML sequence diagram for creation of optional product and service package