DATA BASES 2

JPA Project

Mattia Siriani (10571322)

Matteo Visotto (10608623)

Index

Specification

Specifications – Customer Application

Telco is a prepaid online service that uses two applications with the same database.

A **customer application** is accessible both to unlogged and logged user displaying the service packages offered by the company. A service package has an ID and it comprises one or more services that could by of four types: fixed phone, mobile phone (minutes, SMS + extra fee), fixed internet or mobile internet (gigabytes + extra fee). Each service package is sold with different validity period (12, 24 or 36 months) having different monthly price based on the duration of the subscription. In addition, Telco offers different types of optional products that can be bought with a service package and with a validity period equals to the service package one.

From the home a user can buy a service package accessing a buy page where he can select the service package, 0 or more optional products, a validity period and the start date of the subscription. After confirm a confirmation page is displayed, showing the selected items and the total price to be prepaid. If the user is logged in a buy button is show, otherwise, he can register/login and return back in the confirmation page after completion.

The buy button creates an order record and it displays a payment page where the user can access an external service to complete the payment. If the operation is successful, the order in marked as completed and an activation record is scheduled for the chosen date. Instead, if the payment fails, the order is marked as suspended and the user as insolvent. An insolvent user when logs in can see a list of suspended orders in telco homepage, each of them has a buy button to try the payment again.

When a user fails three payments an alert is raised and stored in the database.

Specifications – Employee Application

An **employee application** is accessible only from authorized people.

In the homepage an employee can create new services, combining them into a service package and associate 0 or more optional products users can buy with. He can also set the available validity period with monthly prices.

In a sales report page, instead, different statistics can be displayed:

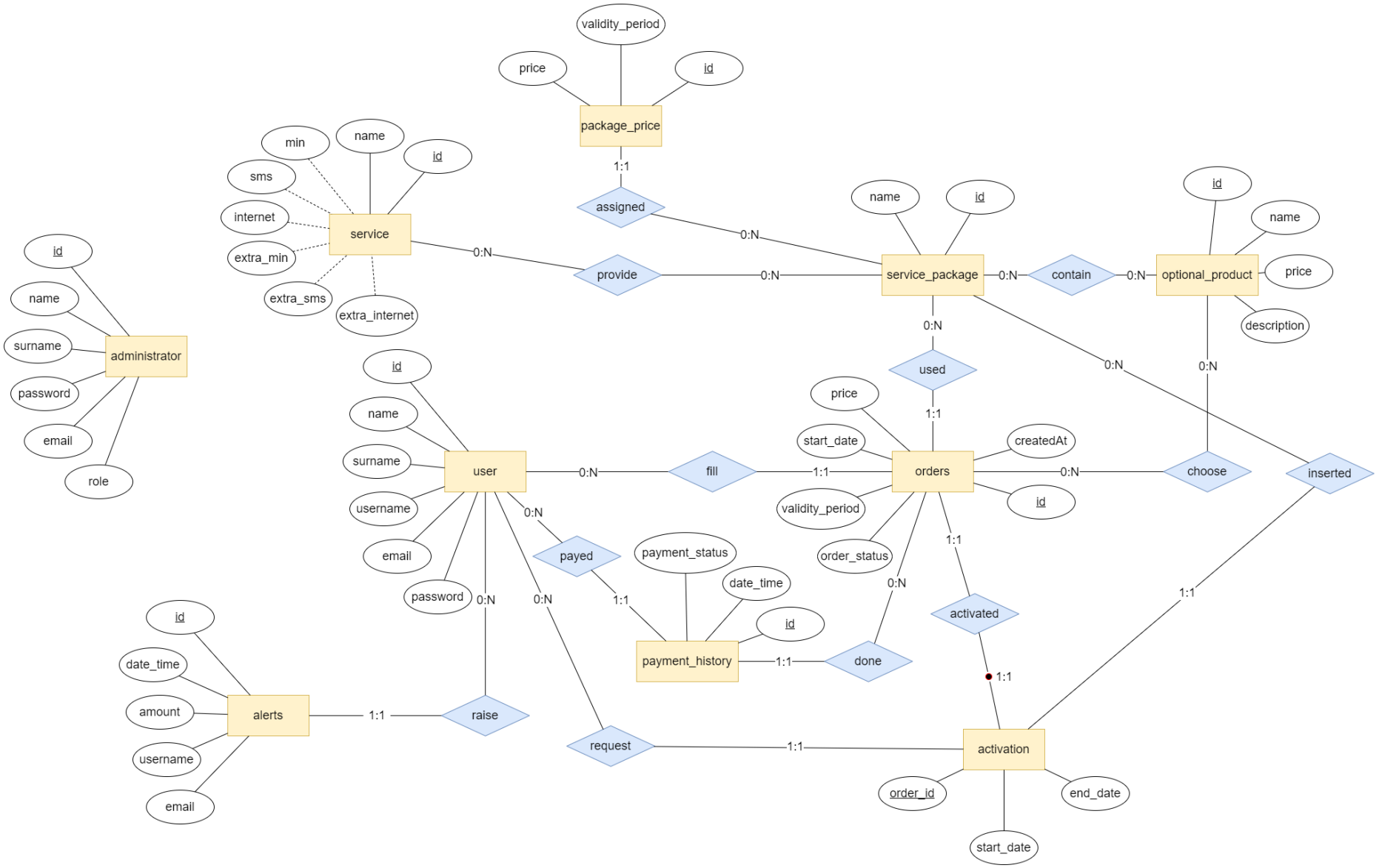
- Number of total purchase per package.
- Number of total purchase per package and validity period.
- Total value of sales per package with and without the optional products.
- Average number of optional products sold together with each service package.
- List of insolvent users, suspended orders and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

Specification interpretation

- Both username and email are unique attributes for the user and so both can be used to login
- Administrators are the employee; they cannot create an account but credential are given
- Administrators can access their application using “/admin” path
- An alert is raised only once for a user and it remains forever
- Materialized views (not available in MySQL) are been substituted with tables populated by means of triggers.
- The fixed phone service contains 50000 minutes only (more than minutes in the longest month), this because a fixed phone cannot have SMS/Internet.

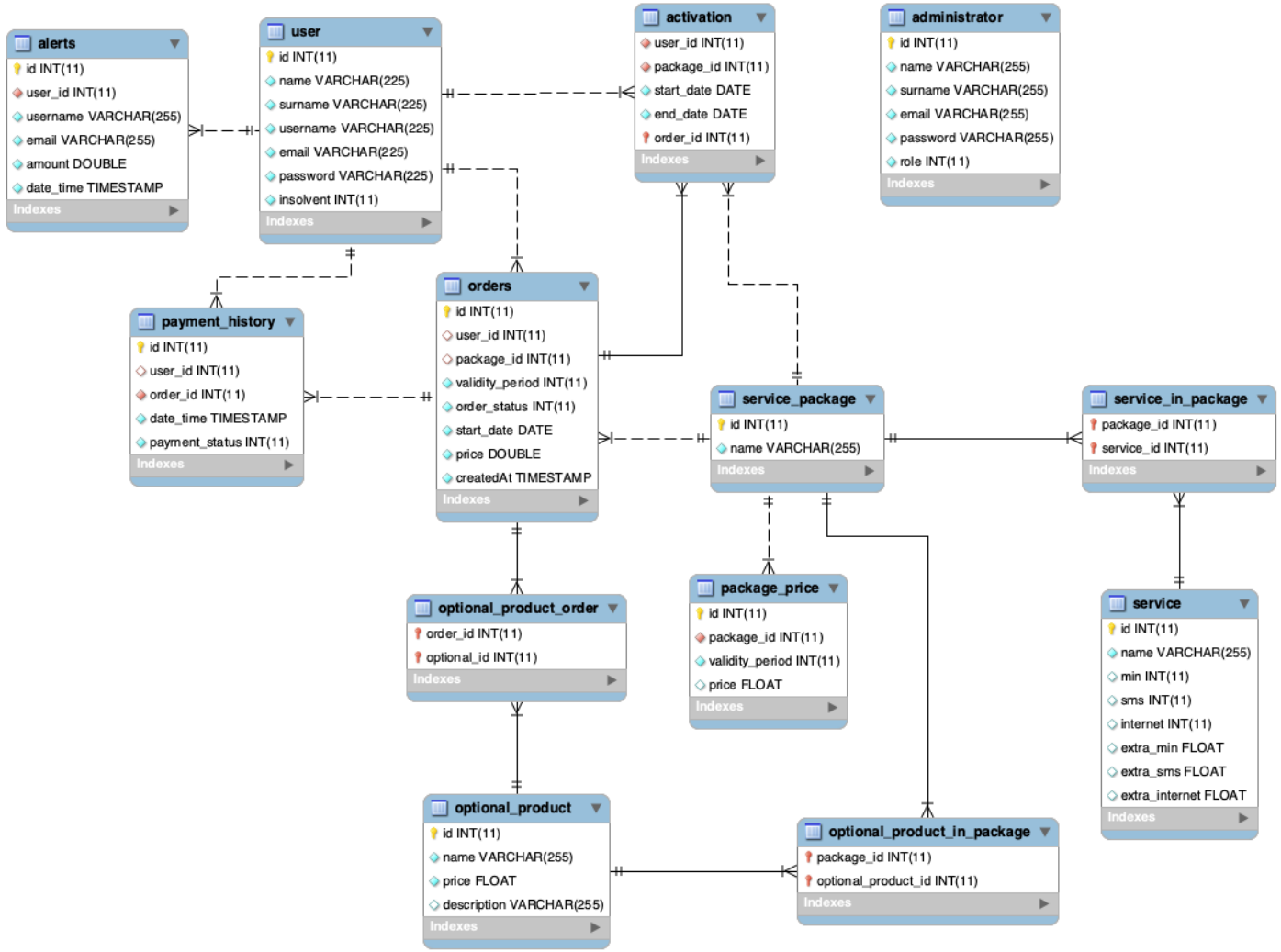
ER Diagrams

Entity Relationship



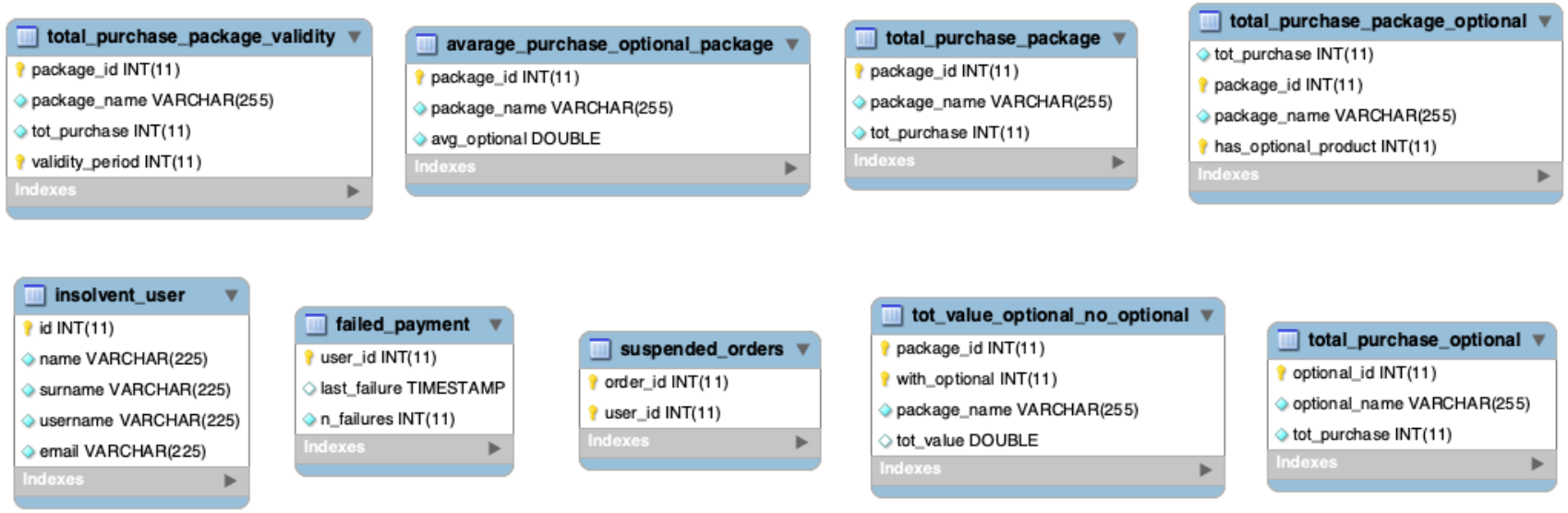
Entity Relationship

- A service contains all the fields for minute, SMS and internet. They are set to NULL when the service does not contain that ”service”
- Package prices are set in a dedicated table because, in this way, it is possible to support validity period different from 12, 24 or 36 months.
- Order status is managed as an integer number (0 – Created, 1 – Payed, 2 - Rejected)
- Payment Status is managed as an integer as well (0 – Failure, 1 - Success)



Entity Relationship – Materialized View

MySQL does not support materialized views so we decide to create tables without relationship with the database schema inserting, deleting and updating data using triggers.



SQL DDL

SQL DDL

```
CREATE TABLE IF NOT EXISTS
`activation` (
  `user_id` int(11) NOT NULL,
  `package_id` int(11) NOT NULL,
  `start_date` date NOT NULL,
  `end_date` date NOT NULL,
  `order_id` int(11) NOT NULL,
  PRIMARY KEY (`order_id`),
  KEY `package_id` (`package_id`),
  KEY `user_id` (`user_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`administrator` (
  `id` int(11) NOT NULL
AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `surname` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `role` int(11) NOT NULL DEFAULT 1,
  PRIMARY KEY (`id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS `alerts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `username` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  `amount` double NOT NULL,
  `date_time` timestamp NOT NULL
DEFAULT current_timestamp() ON UPDATE
current_timestamp(),
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`optional_product` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `price` float NOT NULL,
  `description` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS
`optional_product_in_package` (
  `package_id` int(11) NOT NULL,
  `optional_product_id` int(11) NOT
NULL,
  PRIMARY KEY
(`package_id`,`optional_product_id`)
,
  KEY `optional_product_id`
(`optional_product_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`optional_product_order` (
  `order_id` int(11) NOT NULL,
  `optional_id` int(11) NOT NULL,
  PRIMARY KEY
(`order_id`,`optional_id`),
  KEY `optional_id` (`optional_id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) DEFAULT NULL,
  `package_id` int(11) DEFAULT NULL,
  `validity_period` int(11) NOT NULL,
  `order_status` int(11) NOT NULL DEFAULT 0,
  `start_date` date NOT NULL,
  `price` double NOT NULL DEFAULT 0,
  `createdAt` timestamp NOT NULL DEFAULT
current_timestamp(),
  PRIMARY KEY (`id`),
  KEY `orders_ibfk_2` (`package_id`),
  KEY `user_id` (`user_id`) USING BTREE
)
```

```
CREATE TABLE IF NOT EXISTS `package_price` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `package_id` int(11) NOT NULL,
  `validity_period` int(11) NOT NULL,
  `price` float DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `package_id` (`package_id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS
`payment_history` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) DEFAULT NULL,
  `order_id` int(11) NOT NULL,
  `date_time` timestamp NOT NULL
DEFAULT current_timestamp(),
  `payment_status` int(11) NOT NULL
DEFAULT 0,
  PRIMARY KEY (`id`),
  KEY `order_id` (`order_id`),
  KEY `payment_history_ibfk_1` (`user_id`)
)
```

```
CREATE TABLE IF NOT EXISTS `service` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `min` int(11) DEFAULT NULL,
  `sms` int(11) DEFAULT NULL,
  `internet` int(11) DEFAULT NULL,
  `extra_min` float DEFAULT NULL,
  `extra_sms` float DEFAULT NULL,
  `extra_internet` float DEFAULT NULL,
  PRIMARY KEY (`id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS
`service_in_package` (
  `package_id` int(11) NOT NULL,
  `service_id` int(11) NOT NULL,
  PRIMARY KEY
(`package_id`,`service_id`),
  KEY `service_id` (`service_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`service_package` (
  `id` int(11) NOT NULL
AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
)
```

SQL DDL

```
CREATE TABLE IF NOT EXISTS `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(225) NOT NULL,
  `surname` varchar(225) NOT NULL,
  `username` varchar(225) NOT NULL,
  `email` varchar(225) NOT NULL,
  `password` varchar(225) NOT NULL,
  `insolvent` int(11) NOT NULL DEFAULT 0,
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`username`),
  UNIQUE KEY `email` (`email`)
)
```

SQL DDL – Materialized Views

```
CREATE TABLE IF NOT EXISTS
`avarage_purchase_optional_pack
age` (
  `package_id` int(11) NOT NULL,
  `package_name` varchar(255)
NOT NULL,
  `avg_optional` double NOT NULL
DEFAULT 0,
  PRIMARY KEY (`package_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`failed_payment` (
  `user_id` int(11) NOT NULL,
  `last_failure` timestamp NULL
DEFAULT current_timestamp() ON
UPDATE current_timestamp(),
  `n_failures` int(11) NOT NULL
DEFAULT 0,
  PRIMARY KEY (`user_id`)
)
```

SQL DDL – Materialized Views

```
CREATE TABLE IF NOT EXISTS
`insolvent_user` (
  `id` int(11) NOT NULL,
  `name` varchar(225) NOT NULL,
  `surname` varchar(225) NOT NULL,
  `username` varchar(225) NOT NULL,
  `email` varchar(225) NOT NULL,
  PRIMARY KEY (`id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`suspended_orders` (
  `order_id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  PRIMARY KEY (`order_id`,`user_id`)
)
```

SQL DDL – Materialized Views

```
CREATE TABLE IF NOT EXISTS
`total_purchase_optional` (
  `optional_id` int(11) NOT NULL,
  `optional_name` varchar(255)
NOT NULL,
  `tot_purchase` int(11) NOT NULL
DEFAULT 0,
  PRIMARY KEY (`optional_id`)
)
```

```
CREATE TABLE IF NOT EXISTS
`total_purchase_package` (
  `package_id` int(11) NOT NULL,
  `package_name` varchar(255)
NOT NULL,
  `tot_purchase` int(11) NOT NULL,
  PRIMARY KEY (`package_id`)
)
```

SQL DDL – Materialized Views

```
CREATE TABLE IF NOT EXISTS
`total_purchase_package_optional` (
  `tot_purchase` int(11) NOT NULL
DEFAULT 0,
  `package_id` int(11) NOT NULL,
  `package_name` varchar(255) NOT
NULL,
  `has_optional_product` int(11) NOT
NULL,
  PRIMARY KEY
(`package_id`,`has_optional_product`)
)
```

```
CREATE TABLE IF NOT EXISTS
`total_purchase_package_validity` (
  `package_id` int(11) NOT NULL,
  `package_name` varchar(255) NOT
NULL,
  `tot_purchase` int(11) NOT NULL,
  `validity_period` int(11) NOT NULL,
  PRIMARY KEY
(`package_id`,`validity_period`)
)
```

SQL DDL – Materialized Views

```
CREATE TABLE IF NOT EXISTS `tot_value_optional_no_optional` (
  `package_id` int(11) NOT NULL,
  `with_optional` int(11) NOT NULL,
  `package_name` varchar(255) NOT NULL,
  `tot_value` double DEFAULT 0,
  PRIMARY KEY (`package_id`,`with_optional`)
)
```

SQL DDL - View

```
CREATE VIEW `number_optional_package` AS SELECT `o1`.`package_id`
AS `package_id`, COUNT(`o`.`optional_id`) AS `number` FROM
(`telco`.`optional_product_order` `o` JOIN `telco`.`orders` `o1`
ON(`o`.`order_id` = `o1`.`id`)) GROUP BY `o`.`order_id`,`o1`.`user_id`
```

Trigger design & code

check_validity_period_validity

```
BEGIN

DECLARE val INT;

SET val = (SELECT COUNT(*) FROM package_price AS p WHERE
p.package_id = new.package_id AND p.validity_period =
new.validity_period);

IF val = 0 THEN

        SIGNAL SQLSTATE '23000' SET MESSAGE_TEXT =
        'Invalid validity period';

END IF;

END
```

- ON orders BEFORE INSERT
- This trigger checks if the inserted validity period for the new order is allowed or not.

create_activation_record

```
BEGIN

DECLARE l_package_id INT;

DECLARE l_start_date date;

DECLARE l_validity_period INT;

SET l_package_id = (SELECT package_id FROM orders WHERE id =
new.order_id);

SET l_start_date = (SELECT start_date FROM orders WHERE id =
new.order_id);

SET l_validity_period = (SELECT validity_period FROM orders WHERE
id = new.order_id);

IF new.payment_status = 1 THEN

        INSERT INTO activation (user_id, package_id, start_date,
        end_date, order_id) VALUES (new.user_id, l_package_id,
        l_start_date, DATE_ADD(l_start_date, INTERVAL
        l_validity_period MONTH), new.order_id);

END IF;

END
```

- ON payment_history AFTER INSERT
- This trigger is used to create automatically the activation record for the user order after a payment has been completed successfully.

create_failure_user

```
BEGIN

INSERT INTO failed_payment (user_id, last_failure, n_failures)
VALUES (new.id, NULL, 0);

END
```

- ON user AFTER INSERT
- This trigger is used to populate the failed_payment table when a new user is created, initializing the record to 0 failure.

create_purchase_optional

```
BEGIN
INSERT INTO total_purchase_optional (tot_purchase,
optional_id, optional_name) VALUES (0, new.id, new.name);
END
```

- ON optional_product AFTER INSERT
- This trigger is used to populate the total_purchase_optional table for statistic purpouse when a new optional product is created.

create_purchase_optional_avg

```
BEGIN
INSERT INTO avarage_purchase_optional_package (package_id, package_name, avg_optional) VALUES (new.id, new.name, 0);
END
```

- ON service_package AFTER INSERT
- This trigger is used to populate the avarage_purchase_optional_package table for statistic purpouse when a new service package is created.

create_purchase_package

```
BEGIN
INSERT INTO total_purchase_package (package_id,
package_name, tot_purchase) VALUES (NEW.id, NEW.name, 0);
END
```

- ON service_package AFTER INSERT
- This trigger is used to populate the total_purchase_package table for statistic purpouse when a new service package is created.

create_purchase_package_optional

```
BEGIN

INSERT INTO total_purchase_package_optional (tot_purchase, package_id, package_name, has_optional_product) VALUES (0, new.id, NEW.name, 0), (0, new.id, NEW.name, 1);

END
```

- ON service_package AFTER INSERT
- This trigger is used to populate the total_purchase_package_optional table for statistic purpouse when a new service package is created.

create_purchase_package_validity

```
BEGIN

INSERT INTO total_purchase_package_validity (tot_purchase,
package_id, package_name, validity_period) VALUES (0, new.id,
new.name, 12), (0, new.id, new.name, 24), (0, new.id,
new.name, 36);

END
```

- ON service_package AFTER INSERT
- This trigger is used to populate the total_purchase_package_validity table for statistic purpouse when a new service package is created.

create_tot_value_package

```
BEGIN

INSERT INTO tot_value_optional_no_optional (tot_value,
package_id, package_name, with_optional) VALUES (0, new.id,
NEW.name, 0), (0, new.id, NEW.name, 1);

END
```

- ON service_package AFTER INSERT
- This trigger is used to populate the tot_value_optional_no_optional table for statistic purpouse when a new service package is created.

delete_failure_user

```
BEGIN
DELETE FROM failed_payment WHERE user_id = old.id;
END
```

- ON user AFTER DELETE
- This trigger is used to remove the user from failed_payment table when the user is deleted

delete_purchase_optional

```
BEGIN
DELETE FROM total_purchase_optional WHERE optional_id = old.id;
END
```

- ON optional_product AFTER DELETE
- This trigger is used to remove the corresponding record from total_purhcase_optional when an optional product is deleted.

delete_purchase_optional_avg

```
BEGIN

DELETE FROM avarage_purchase_optional_package WHERE
package_id = old.id;

END
```

- ON service_packege AFTER DELETE
- This trigger is used to remove the corresponding record from avarage_purchase_optional_package when a service package is deleted.

delete_purchase_package

```
BEGIN

DELETE FROM total_purchase_package WHERE package_id =
old.id;

END
```

- ON service_packege AFTER DELETE
- This trigger is used to remove the corresponding record from total_purchase_package when a service package is deleted.

delete_purchase_package_optional

```
BEGIN

DELETE FROM total_purchase_package_optional WHERE
package_id = old.id;

END
```

- ON service_packege AFTER DELETE
- This trigger is used to remove the corresponding record from total_purchase_package_optional when a service package is deleted.

delete_purchase_package_validity

```
BEGIN

DELETE FROM total_purchase_package_validity WHERE
package_id = old.id;

END
```

- ON service_packege AFTER DELETE
- This trigger is used to remove the corresponding record from total_purchase_package_validity when a service package is deleted.

delete_tot_value_package

```
BEGIN

DELETE FROM tot_value_optional_no_optional WHERE
package_id = old.id;

END
```

- ON service_packege AFTER DELETE
- This trigger is used to remove the corresponding record from tot_value_optional_no_optional when a service package is deleted.

manage_insolvent_user

```
BEGIN
DECLARE number int;
IF new.payment_status = 0 THEN
        UPDATE user set insolvent = 1 WHERE id = new.user_id;
ELSE
        SET number = (SELECT COUNT(*) FROM
        payment_history AS p1 WHERE p1.payment_status = 0
        AND p1.user_id = new.user_id AND p1.order_id NOT IN
        (SELECT p2.order_id FROM payment_history AS p2
        WHERE p2.payment_status = 1 AND p2.user_id =
        new.user_id));
IF number = 0 THEN
        UPDATE user set insolvent = 0 WHERE id = new.user_id;
END IF;
END IF;
END;
```

- ON payment_history AFTER INSERT
- This trigger is used to set a user as insolvent if a new payment fails, otherwise to remove the insolvent status when all his pending payments has been payed.

optional_in_package_for_order

```
BEGIN

DECLARE val INT DEFAULT 0;

SET val = (SELECT COUNT(*) as verify FROM
optional_product_in_package AS o JOIN orders AS t ON
t.package_id = o.package_id WHERE t.id=new.order_id AND
o.optional_product_id = new.optional_id);

IF val = 0 THEN

        SIGNAL SQLSTATE '23000'

        SET MESSAGE_TEXT = 'Optional product not in
        package';

END IF;

END
```

- ON optional_product_order BEFORE INSERT
- This trigger is used to verify if the optional products associated with a service package in an order really belongs to the service package.

raise_new_alert

```
BEGIN

DECLARE l_username varchar(255);

DECLARE l_email varchar(255);

DECLARE l_amount double;

SET l_amount = (SELECT SUM(price) FROM orders AS o WHERE
o.user_id = new.user_id AND o.id NOT IN (SELECT p.order_id FROM
payment_history AS p WHERE p.user_id = new.user_id AND
p.payment_status = 1));

SET l_email = (SELECT email FROM user WHERE id = new.user_id);

SET l_username = (SELECT username FROM user WHERE id =
new.user_id);

IF new.n_failures = 3 THEN

        INSERT INTO alerts (user_id, email, username, amount,
        date_time) VALUES (new.user_id, l_email, l_username,
        l_amount, new.last_failure);

END IF;

END
```

- ON failed_payments AFTER UPDATE
- This trigger is used to rise an alert when the number of failed payment for a user reach the number 3.

retrieve_insolvent_users

```
BEGIN
IF old.insolvent = 0 and new.insolvent = 1 THEN
        INSERT INTO insolvent_user (id, name, surname,
        username, email) VALUES (new.id, new.name,
        new.surname, new.username, new.email);
ELSEIF old.insolvent = 1 and new.insolvent = 0 THEN
        DELETE FROM insolvent_user WHERE id = new.id;
END IF;
END
```

- ON user BEFORE UPDATE
- This trigger is used to add a user to the insolvent user lists when he wasn't insolvent but he fails a payment.

retrieve_suspended_orders

```
BEGIN
DECLARE number int;
IF new.order_status = 2 THEN
        SET number = (SELECT count(*) FROM
        suspended_orders WHERE order_id=new.id AND
        user_id=new.user_id);
        IF number = 0 THEN
                INSERT INTO suspended_orders
                (order_id, user_id) VALUES (new.id,
                new.user_id);
        END IF;
ELSEIF new.order_status = 1 or new.order_status = 0 THEN
        DELETE FROM suspended_orders WHERE order_id =
        new.id;
END IF;
END
```

- ON orders AFTER UPDATE
- This trigger is used to add an order to the suspended list when its status is a failed payment (status=2), otherwise, the order is removed from the suspended list when it is payed (status = 1) or created (status=0).

set_base_price

```
BEGIN

DECLARE amount double;

SET amount = (SELECT price FROM package_price WHERE
package_id = new.package_id AND validity_period =
new.validity_period);

SET new.price = amount * new.validity_period;

END
```

- ON orders BEFORE INSERT
- This trigger calculate and set the price for a new order without taking into account the value of the optional product if included.

update_failed_payment

```
BEGIN
IF new.payment_status = 0 THEN
        UPDATE failed_payment SET n_failures = n_failures +
        1, last_failure = CURRENT_TIMESTAMP() WHERE
        user_id = new.user_id;
END IF;
END
```

- ON payment_history AFTER INSERT
- This trigger in used to update the number of payment failures of an user when a new failed payment is inserted.

update_order_price_optional

```
BEGIN

DECLARE p DOUBLE;

DECLARE v_period INT;

SET p = (SELECT price FROM optional_product WHERE id =
new.optional_id);

SET v_period = (SELECT validity_period FROM orders WHERE
id=new.order_id);

UPDATE orders SET price = price + (p*v_period) WHERE id =
new.order_id;

END
```

- ON optional_product_order AFTER INSERT
- This trigger update the amount of an order summing the price of the optional products bought with (considering also the validity period selected within the order).

update_order_status

```
BEGIN
IF new.payment_status = 0 THEN
        UPDATE orders SET order_status = 2 WHERE id =
        new.order_id;
ELSE
        UPDATE orders SET order_status = 1 WHERE id =
        new.order_id;
END IF;
END
```

- ON payment_history AFTER INSERT
- This trigger is used to update the order for which a user make a payment. Status 2 means suspended (payment failure -> 0), status 1 means success.

update_purchase_optional

```
BEGIN
DECLARE oId int;
DECLARE done INT DEFAULT FALSE;
DECLARE cur CURSOR FOR select optional_id from optional_product_order where order_id = new.id;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
IF (old.order_status = 0 or old.order_status = 2) and new.order_status = 1 THEN
        OPEN cur;
            ins_loop: LOOP
                FETCH cur INTO oId;
                IF done THEN
                    LEAVE ins_loop;
                END IF;
                UPDATE total_purchase_optional SET tot_purchase = tot_purchase + 1
                WHERE optional_id = oId;
            END LOOP;
        CLOSE cur;
END IF;
END
```

- ON orders BEFORE UPDATE
- When an order is successfully payed, this trigger increments the number of purchase for the optional products included in the order.

update_purchase_package

```
BEGIN

IF (old.order_status = 0 or old.order_status=2) and
new.order_status = 1 THEN

        UPDATE total_purchase_package SET tot_purchase =
        tot_purchase + 1 WHERE package_id =
        new.package_id;

END IF;

END
```

- ON orders BEFORE UPDATE
- When an order is successfully payed, this trigger increments the number of purchase for the ordered service package.

update_purchase_package_avg

```
BEGIN
IF new.order_status = 1 THEN
        UPDATE avarage_purchase_optional_package SET avg_optional =
        IFNULL((SELECT SUM(number) FROM number_optional_package WHERE
        package_id = new.package_id),0) / (SELECT count(*) FROM orders
        WHERE package_id=new.package_id) WHERE package_id=new.package_id;
END IF;
END
```

- ON orders AFTER UPDATE
- When an order is successfully payed, this trigger recalculate the average optional products users buy within the package.
- In this case is after update due to the usage of a view.

update_purchase_package_optional

```
BEGIN

DECLARE counter int;

SET counter = IFNULL((SELECT count(*) FROM optional_product_order
WHERE order_id = new.id GROUP BY order_id), 0);

IF (old.order_status = 0 or old.order_status=2) and new.order_status = 1
THEN
                IF counter = 0 THEN

                                UPDATE total_purchase_package_optional set
                                tot_purchase = tot_purchase + 1 where
                                package_id = new.package_id AND
                                has_optional_product = 0;

                ELSE

                                UPDATE total_purchase_package_optional set
                                tot_purchase = tot_purchase + 1 where
                                package_id = new.package_id AND
                                has_optional_product = 1;

                END IF;

END IF;

END
```

- ON orders BEFORE UPDATE
- When an order is successfully payed, this trigger increments the number of purchase for the service package with or without optional products included.
- (NOT REQUESTED)

update_purchase_package_validity

```
BEGIN

IF (old.order_status = 0 or old.order_status=2) and
new.order_status = 1 THEN

        UPDATE total_purchase_package_validity SET
        tot_purchase = tot_purchase + 1 WHERE package_id
        = new.package_id AND validity_period =
        new.validity_period;

END IF;

END
```

- ON orders BEFORE UPDATE
- When an order is successfully payed, this trigger increments the number of purchase for the service package considering the selected validity period.

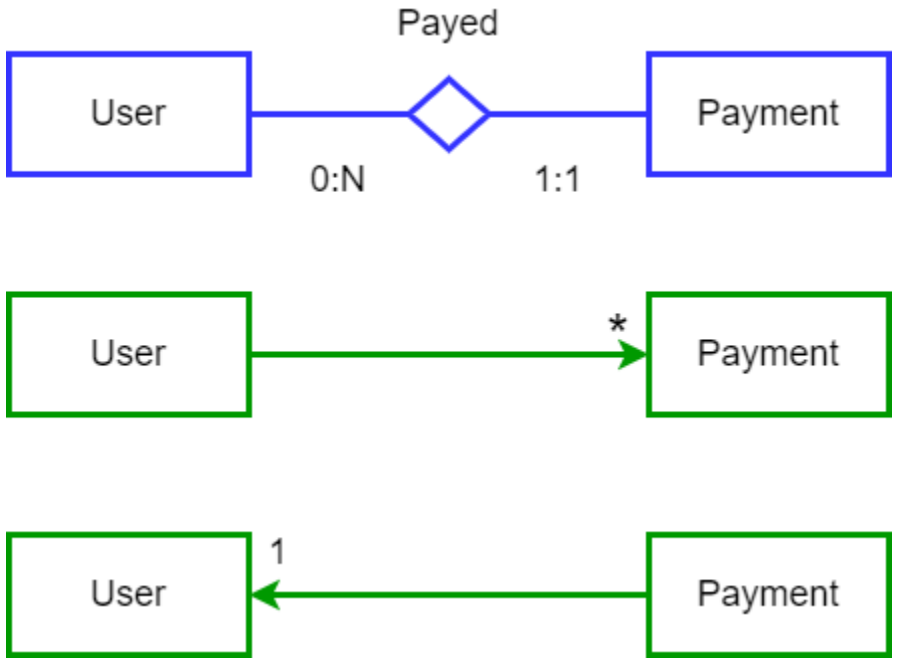
update_tot_value_package

```
BEGIN

DECLARE optional_price double;

DECLARE validity_price double;

SET optional_price = (SELECT COALESCE(SUM(o.price) , 0) FROM
optional_product_order AS x JOIN optional_product AS o ON o.id = x.optional_id
WHERE x.order_id = new.id);

SET validity_price = (optional_price * old.validity_period);

IF (old.order_status = 0 or old.order_status = 2) and new.order_status = 1 THEN

        UPDATE tot_value_optional_no_optional SET tot_value = tot_value +
        new.price where package_id = new.package_id AND
        with_optional = 1;

        UPDATE tot_value_optional_no_optional SET tot_value = tot_value
        + (new.price - validity_price) where package_id = new.package_id
        AND with_optional = 0;

END IF;

END
```

- ON orders BEFORE UPDATE
- When an order is successfully payed, this trigger update the total sold value of the selected service package both considering and not optional products.

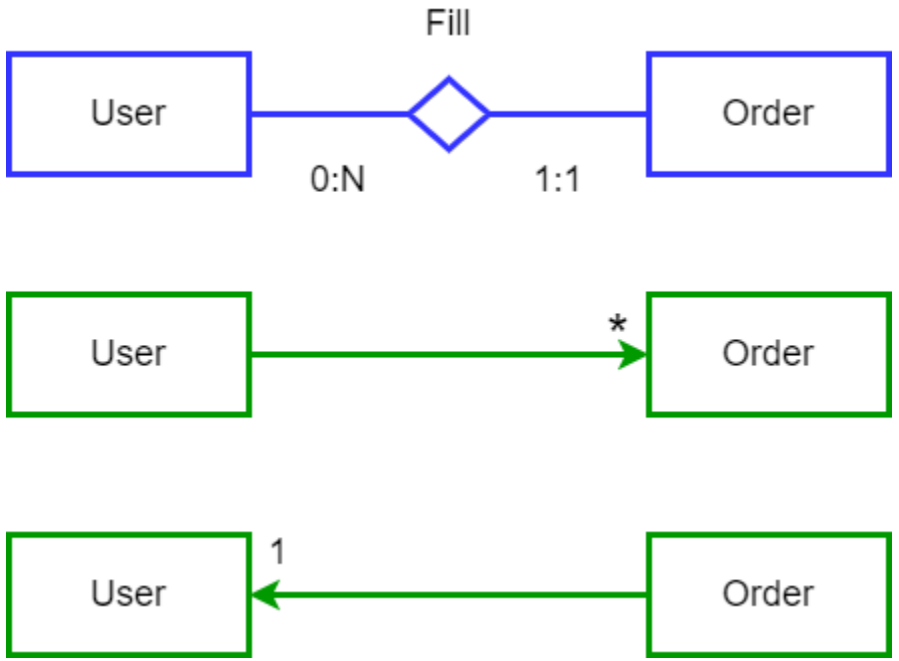
ORM design

Relationship ”Payed”



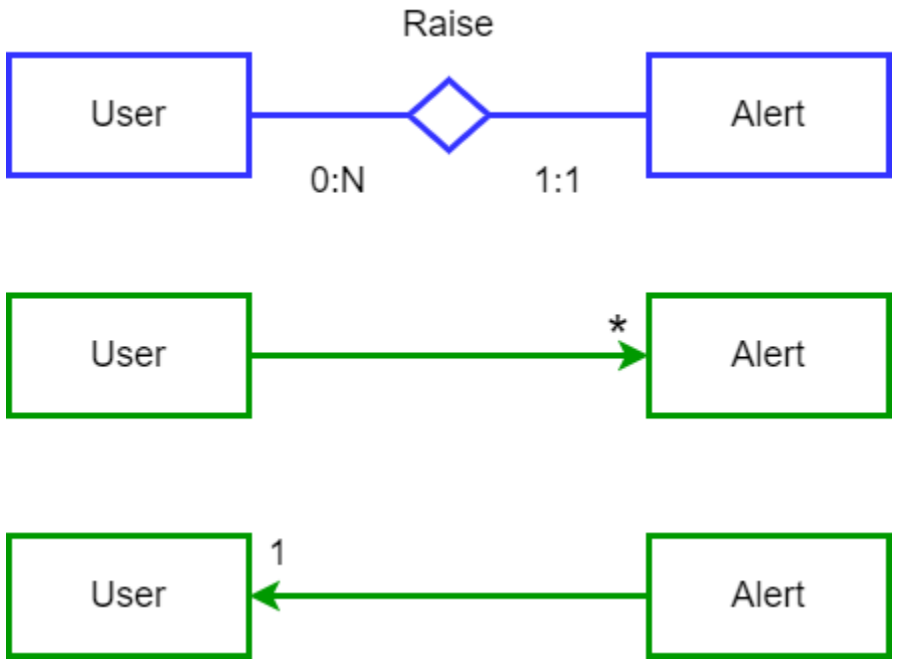
- User -> Payment @OneToMany to access all the payments that a certain user has done.
- Payment -> User @ManyToOne to access the user that has done a specific payment.

Relationship ”Fill”



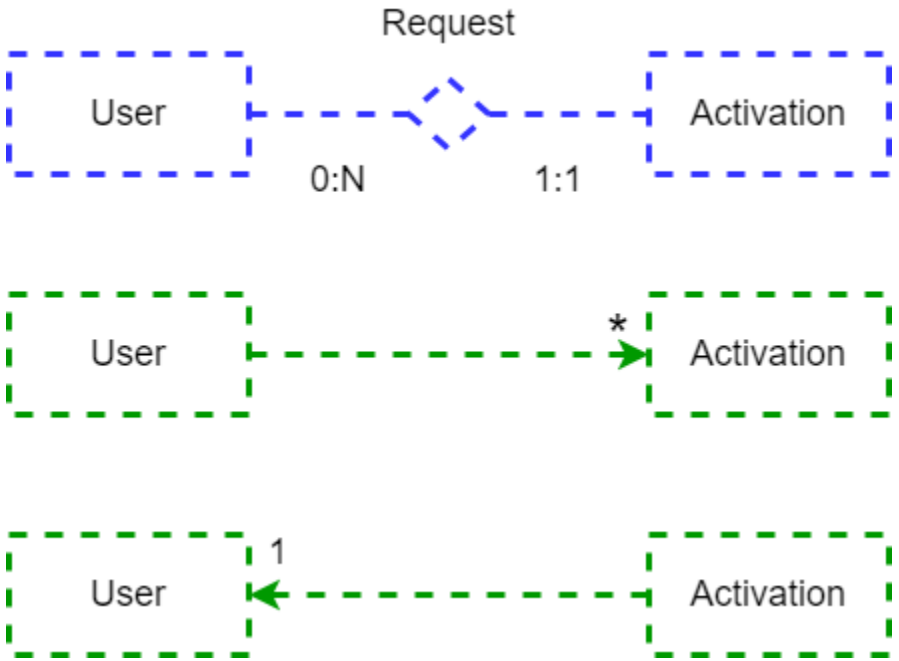
- User -> Order @OneToMany to access all the orders that a user has created.
- Order -> User @ManyToOne to access the user that has created a specific order.

Relationship ”Raise”



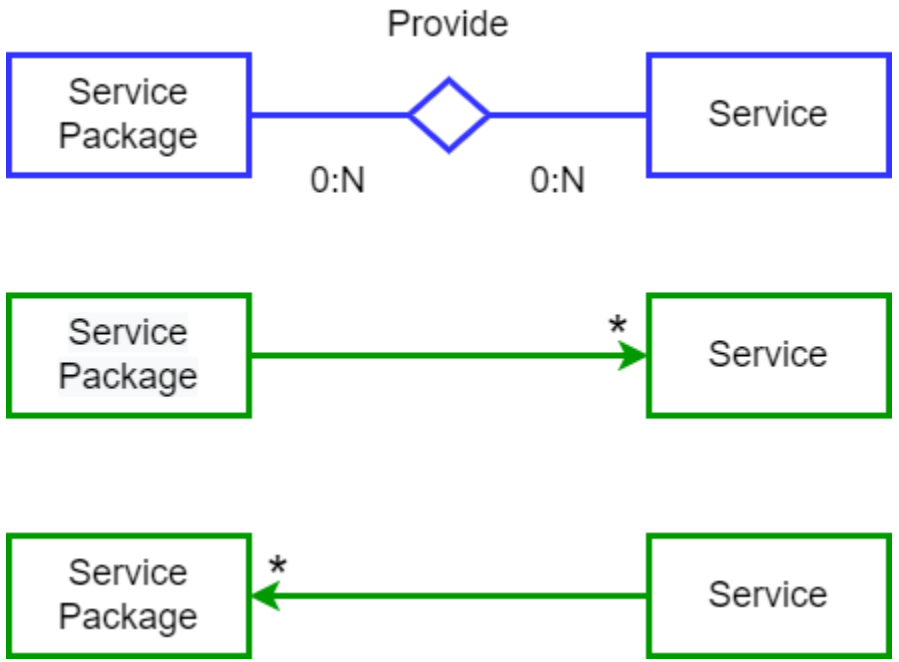
- User -> Alert @OneToMany to access all the alerts raised by a user, according to our specific interpretation, only an alert per user could be raised.
- Alert -> User @ManyToOne to access the user that has raised a specific alert.

Relationship ”Acquire”



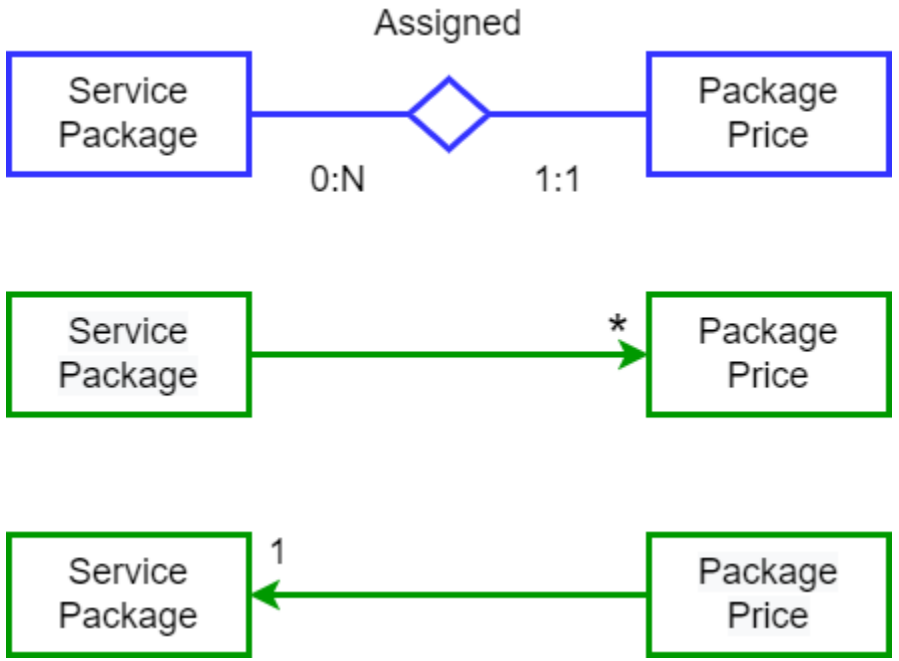
- User -> Activation @OneToMany to access all the activations acquired by a user. Not implemented.
- Activation -> User @ManyToOne to access the user that has acquired a specific activation. Not implemented.

Relationship ”Provide”



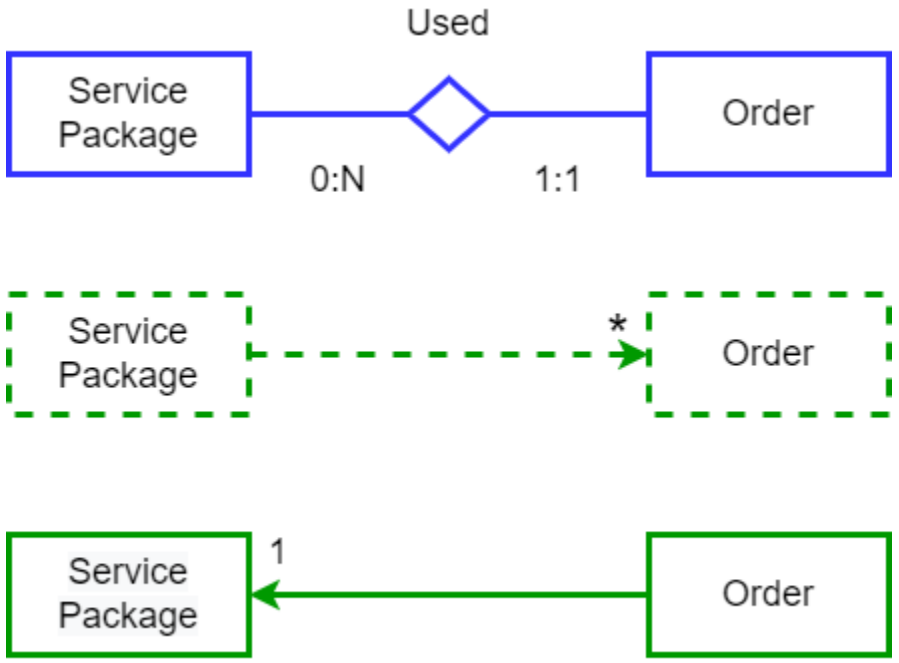
- Service Package -> Service @ManyToMany to access all the services provided in a service package.
- Service -> Service Package @ManyToMany to access all the service packages that have a specific service.

Relationship ”Assigned”



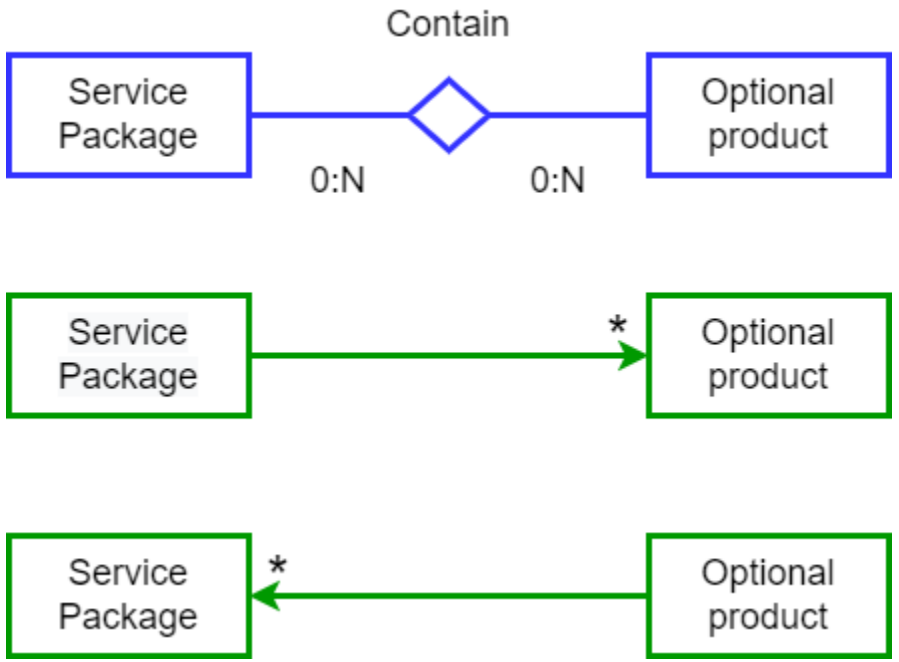
- Service Package -> Package Price @OneToMany to access all the prices assigned to a service package.
- Package Price -> Service Package @ManyToOne to access the service package that has a specific package price.

Relationship ”Used”



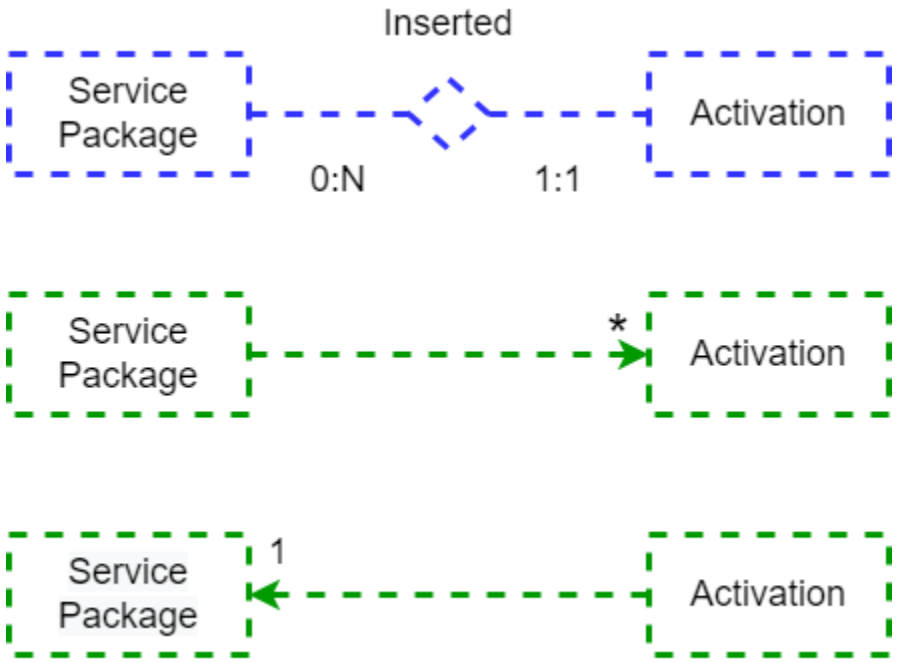
- Service Package -> Order @OneToMany to access all the orders of a service package. Not implemented.
- Order -> Service Package @ManyToOne to access the service package that has been bought in a specific order.

Relationship ”Contain”



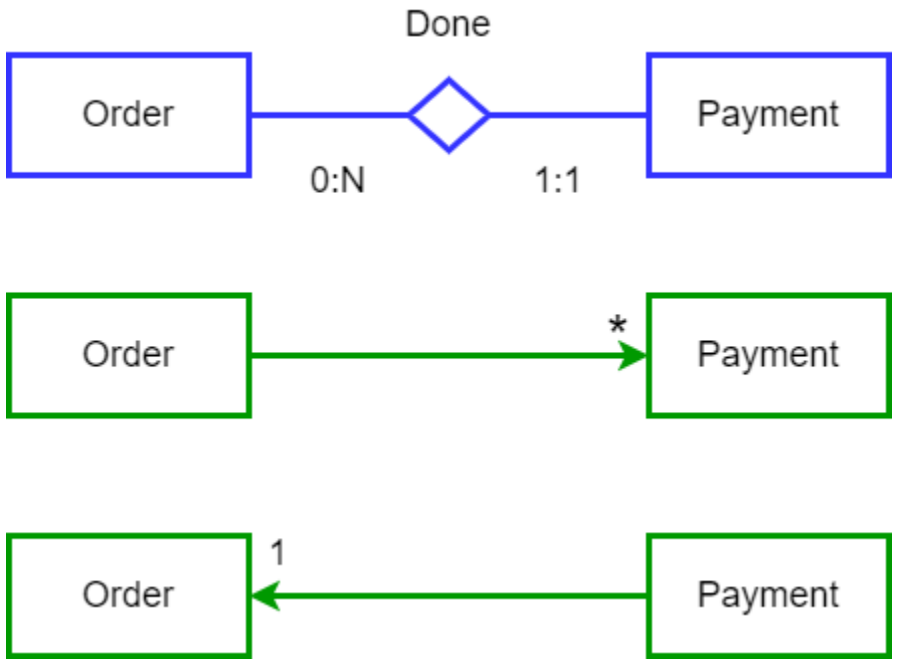
- Service Package -> Optional product @ManyToMany to access all the optional products contained in a service package.
- Optional product -> Service Package @ManyToMany to access all the service packages that have a specific optional product.

Relationship ”Inserted”



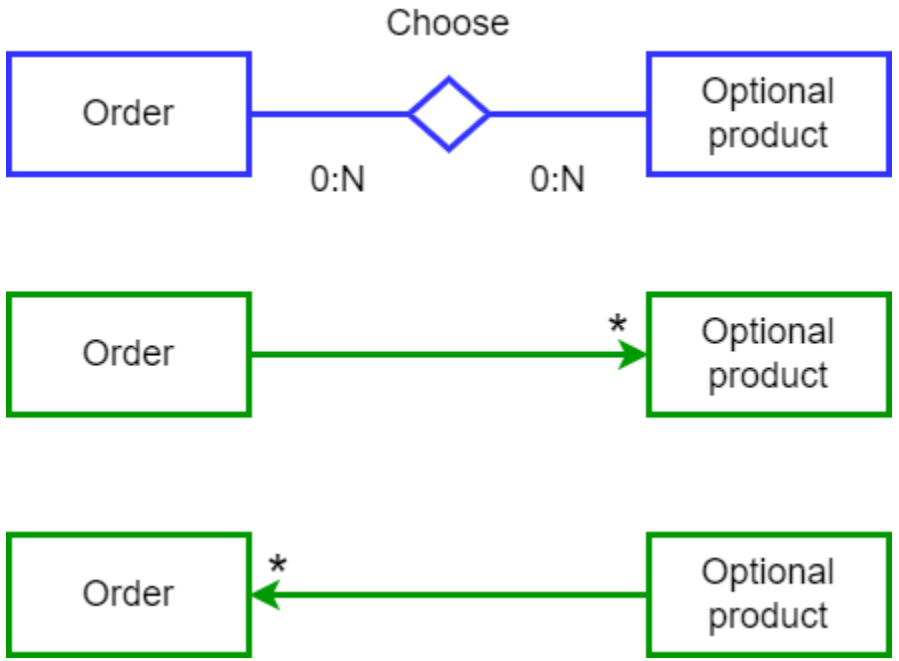
- Service Package -> Activation @OneToMany to access all the activations related to a service package. Not implemented.
- Activation -> Service Package @ManyToOne to access the service package that has been inserted in a specific activation. Not implemented.

Relationship ”Done”



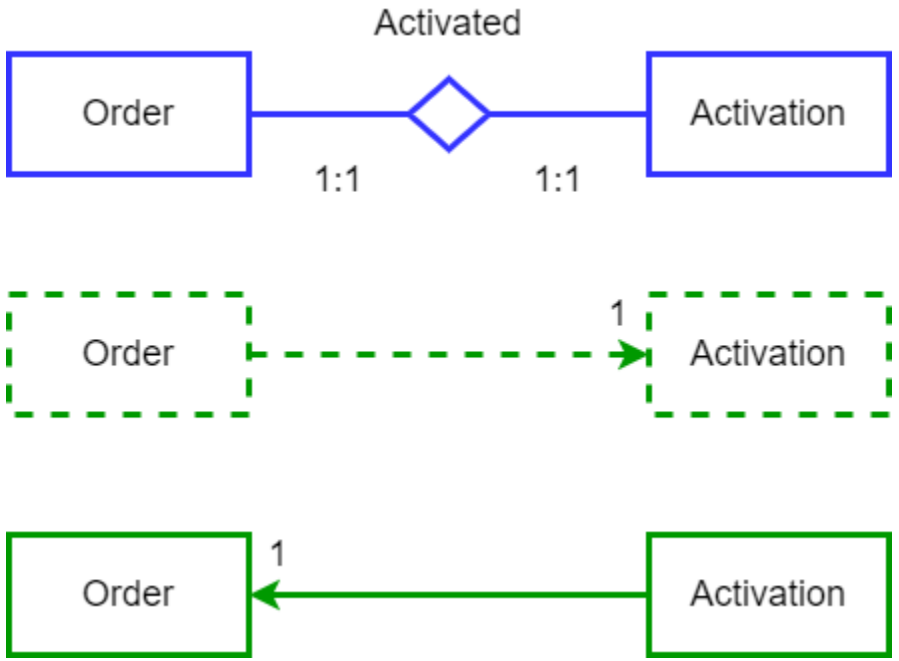
- Order -> Payment @OneToMany to access all the payments done in a specific order.
- Payment -> Order @ManyToOne to access the order that has try to pay by a specific payment.

Relationship ”Choose”



- Order -> Optional product @ManyToMany to access all the optional products contained in an order.
- Optional product -> Order @ManyToMany to access all the orders that have a specific optional product.

Relationship ”Activated”



- Order -> Activation
  @OneToMany to access the activation schedule related to the correspondent order. Not implemented.
- Activation -> Order
  @ManyToOne to access the order that corresponds to a specific activation.

Entities

Activation

```
@Entity
@Table(name = "activation")
public class Activation {
   @Id
   @Column(name = "order_id", nullable = false)
   Integer id;

   @MapsId
   @OneToOne(fetch = FetchType.LAZY, optional = false)
   @JoinColumn(name = "order_id", nullable = false)
   Order orders;

   @Column(name = "user_id", nullable = false)
   Integer userId;

   @Column(name = "package_id", nullable = false)
   Integer packageId;

   @Column(name = "start_date", nullable = false)
   LocalDate startDate;

   @Column(name = "end_date", nullable = false)
   LocalDate endDate;
}
```

Administrator

```
@Entity
@Table(name = "administrator")
public class Administrator {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

    @Column(name = "surname", nullable = false)
    String surname;

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

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

    @Column(name = "role", nullable = false)
    Integer role;
}
```

Alert

```
@Entity
@Table(name = "alerts")
public class Alert {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

    @Column(name = "username", nullable = false)
    String username;

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

    @Column(name = "amount", nullable = false)
    Double amount;

    @Column(name = "date_time", nullable = false)
    Date dateTime;
}
```

AveragePurchaseOptionalPackage

```
@Entity
@Table(name = "avarage_purchase_optional_package")
public class AveragePurchaseOptionalPackage {
    @Id
    @Column(name = "package_id", nullable = false)
    Integer id;

    @Column(name = "package_name", nullable = false)
    String packageName;

    @Column(name = "avg_optional", nullable = false)
    Double avgOptional;
}
```

FailedPayment

```
@Entity
@Table(name = "failed_payment")
public class FailedPayment {
    @Id
    @Column(name = "user_id", nullable = false)
    Integer id;

    @Column(name = "last_failure")
    Date lastFailure;

    @Column(name = "n_failures", nullable = false)
    Integer nFailures;
}
```

InsolventUser

```
@Entity
@Table(name = "insolvent_user")
public class InsolventUser {
    @Id
    @Column(name = "id", nullable = false)
    Integer id;

    @Column(name = "name", nullable = false, length = 225)
    String name;

    @Column(name = "surname", nullable = false, length = 225)
    String surname;

    @Column(name = "username", nullable = false, length = 225)
    String username;

    @Column(name = "email", nullable = false, length = 225)
    String email;
}
```

OptionalProduct

```
@Entity
@Table(name = "optional_product")
public class OptionalProduct {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

@Column(name = "price", nullable = false)
    Double price;

    @Column(name = "description")
    String description;

    @ManyToMany(mappedBy = "optionalProducts")
    List<ServicePackage> servicePackages = new ArrayList<>();
}
```

OptionalProductInPackage

```
@Entity
@Table(name = "optional_product_in_package")
public class OptionalProductInPackage {
    @EmbeddedId
    OptionalProductInPackageId id;

    @MapsId("packageId")
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "package_id", nullable = false)
    ServicePackage _package;

    @MapsId("optionalProductId")
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "optional_product_id", nullable = false)
    OptionalProduct optionalProduct;
}
```

OptionalProductOrder

```
@Entity
@Table(name = "optional_product_order")
public class OptionalProductOrder {
    @EmbeddedId
    OptionalProductOrderId id;

    @MapsId("orderId")
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "order_id", nullable = false)
    Order order;

    @MapsId("optionalId")
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "optional_id", nullable = false)
    OptionalProduct optional;
}
```

Order

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "package_id")
    ServicePackage _package;

    @Column(name = "validity_period", nullable = false)
    Integer validityPeriod;

  @Column(name = "order_status", nullable = false)
    Integer orderStatus;

  @Column(name = "order_status", nullable = false)
    Integer orderStatus;

    @Column(name = "start_date", nullable = false)
    LocalDate startDate;

    @Column(name = "price", nullable = false)
    Double price;

    @Column(name = "createdAt", nullable = false)
    Date createdAt;

    @ManyToMany
    @JoinTable(name = "optional_product_order",
          joinColumns = @JoinColumn(name = "order_id"),
          inverseJoinColumns = @JoinColumn(name = "optional_id"))
    List<OptionalProduct> optionalProducts = new ArrayList<>();

    @OneToMany(mappedBy = "order")
    List<PaymentHistory> paymentHistories = new ArrayList<>();
}
```

PackagePrice

```
@Entity
@Table(name = "package_price")
public class PackagePrice {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "package_id")
    ServicePackage _package;

    @Column(name = "validity_period",
nullable = false)
    Integer validityPeriod;

    @Column(name = "price")
    Double price;
}
```

PaymentHistory

```
@Entity
@Table(name = "payment_history")
public class PaymentHistory {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "order_id", nullable = false)
    Order order;

    @Column(name = "date_time", nullable = false)
    Date dateTime;

    @Column(name = "payment_status", nullable = false)
    Integer paymentStatus;
}
```

Service

```
@Entity
@Table(name = "service")
public class Service {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

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

    @Column(name = "min")
    Integer min;

    @Column(name = "sms")
    Integer sms;

    @Column(name = "internet")
    Integer internet;

    @Column(name = "extra_min")
    Double extraMin;

    @Column(name = "extra_sms")
    Double extraSms;

    @Column(name = "extra_internet")
    Double extraInternet;

    @ManyToMany(mappedBy = "services")
    List<ServicePackage> servicePackages = new ArrayList<>();
}
```

ServiceInPackage

```
@Entity
@Table(name = "service_in_package")
public class ServiceInPackage {
   @EmbeddedId
   ServiceInPackageId id;

   @MapsId("packageId")
   @ManyToOne(fetch = FetchType.LAZY, optional = false)
   @JoinColumn(name = "package_id", nullable = false)
   ServicePackage _package;

   @MapsId("serviceId")
   @ManyToOne(fetch = FetchType.LAZY, optional = false)
   @JoinColumn(name = "service_id", nullable = false)
   Service service;
```

ServicePackage

```
@Entity
@Table(name = "service_package")
public class ServicePackage {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
     Integer id;


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


    @OneToMany(mappedBy = "_package", cascade =
CascadeType.ALL)
     List<PackagePrice> packagePrices = new ArrayList<>();
```

```
@ManyToMany
   @JoinTable(name = "optional_product_in_package",
          joinColumns = @JoinColumn(name = "package_id"),
          inverseJoinColumns = @JoinColumn(name =
"optional_product_id"))
     List<OptionalProduct> optionalProducts = new ArrayList<>();


    @ManyToMany
    @JoinTable(name = "service_in_package",
          joinColumns = @JoinColumn(name = "package_id"),
          inverseJoinColumns = @JoinColumn(name =
"service_id"))
     List<Service> services = new ArrayList<>();
}
```

SuspendedOrder

```
@Entity
@Table(name = "suspended_orders")
public class SuspendedOrder {
    @EmbeddedId
    SuspendedOrderId id;
}
```

TotalPurchaseOptional

```
@Entity
@Table(name = "total_purchase_optional")
public class TotalPurchaseOptional {
    @Id
    @Column(name = "optional_id", nullable = false)
    Integer id;

    @Column(name = "optional_name", nullable = false)
    String optionalName;

    @Column(name = "tot_purchase", nullable = false)
    Integer totPurchase;
}
```

TotalPurchasePackage

```
@Entity
@Table(name = "total_purchase_package")
public class TotalPurchasePackage {
    @Id
    @Column(name = "package_id", nullable = false)
    Integer id;

    @Column(name = "package_name", nullable = false)
    String packageName;

    @Column(name = "tot_purchase", nullable = false)
    Integer totPurchase;
}
```

TotalPurchasePackageOptional

```
@Entity
@Table(name = "total_purchase_package_optional")
public class TotalPurchasePackageOptional {
    @EmbeddedId
    TotalPurchasePackageOptionalId id;

    @Column(name = "tot_purchase", nullable = false)
    Integer totPurchase;

    @Column(name = "package_name", nullable = false)
    String packageName;
}
```

TotalPurchasePackageValidity

```
@Entity
@Table(name = "total_purchase_package_validity")
public class TotalPurchasePackageValidity {
    @EmbeddedId
    TotalPurchasePackageValidityId id;

    @Column(name = "package_name", nullable = false)
    String packageName;

    @Column(name = "tot_purchase", nullable = false)
    Integer totPurchase;
}
```

TotValueOptionalNoOptional

```
@Entity
@Table(name = "tot_value_optional_no_optional")
public class TotValueOptionalNoOptional {
    @EmbeddedId
    private TotValueOptionalNoOptionalId id;

    @Column(name = "package_name", nullable = false)
    private String packageName;

    @Column(name = "tot_value")
    private Double totValue;
}
```

User

```
@Entity
@Table(name = "user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    Integer id;

    @Column(name = "name", nullable = false, length = 225)
    String name;

    @Column(name = "surname", nullable = false, length = 225)
    String surname;

    @Column(name = "username", nullable = false, length = 225)
    String username;

@Column(name = "email", nullable = false, length = 225)
    String email;

    @Column(name = "password", nullable = false, length = 225)
    String password;

    @Column(name = "insolvent", nullable = false)
    Integer insolvent;

    @OneToMany(mappedBy = "user")
    List<PaymentHistory> paymentHistories = new ArrayList<>();

    @OneToMany(mappedBy = "user")
    List<Alert> alerts = new ArrayList<>();

    @OneToMany(mappedBy = "user")
    List<Order> orders = new ArrayList<>();
```

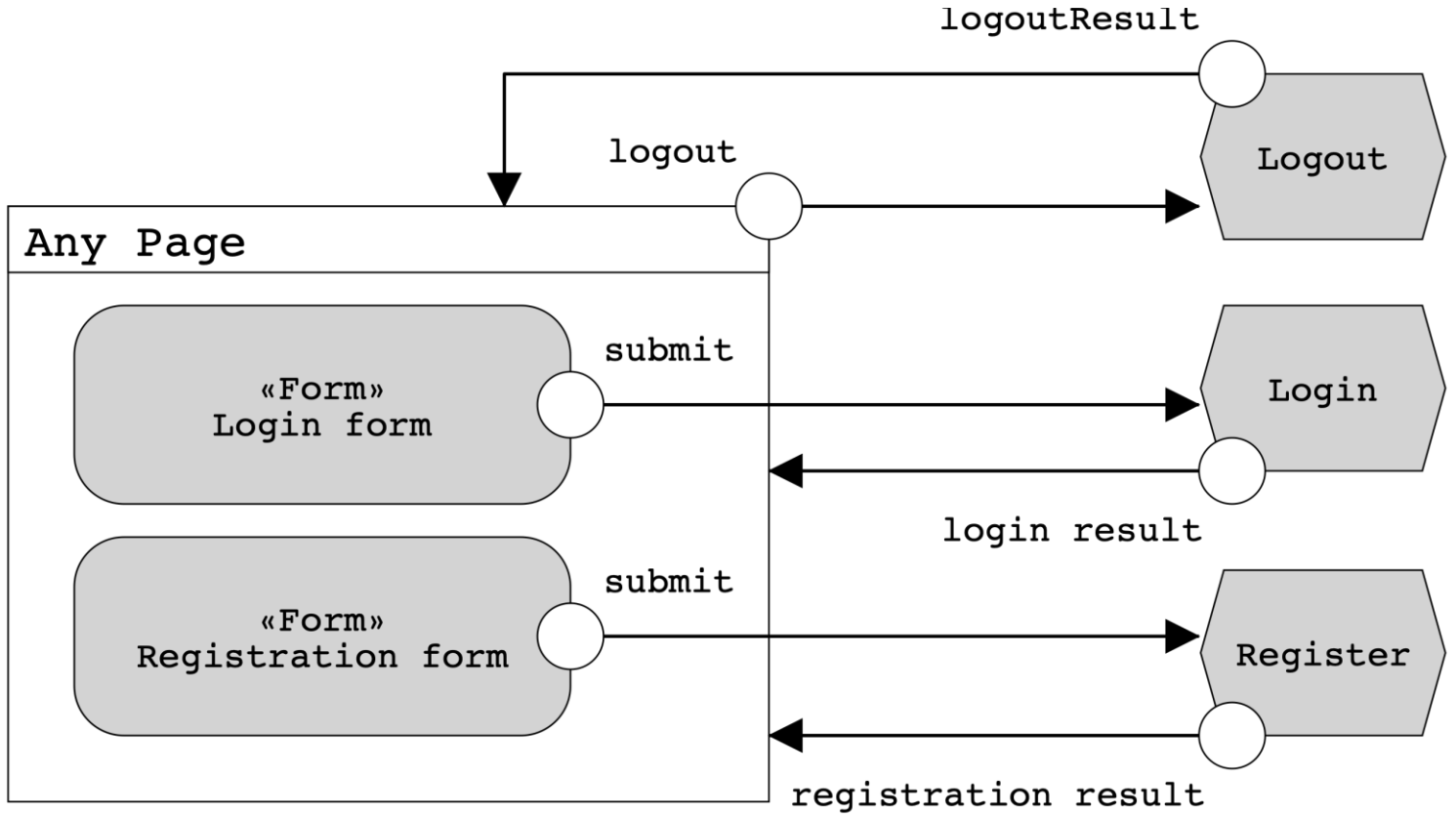
Entities with @EmbeddedId interpretation

- We have used @EmbeddedId in different entities when classes have a composite primary key. In order to use this Id we exploit the @Embeddable annotation to use a class as Id of an entity and all the attribute present in that class represent the composite primary key of the entity.

Functional analysis of the interactions

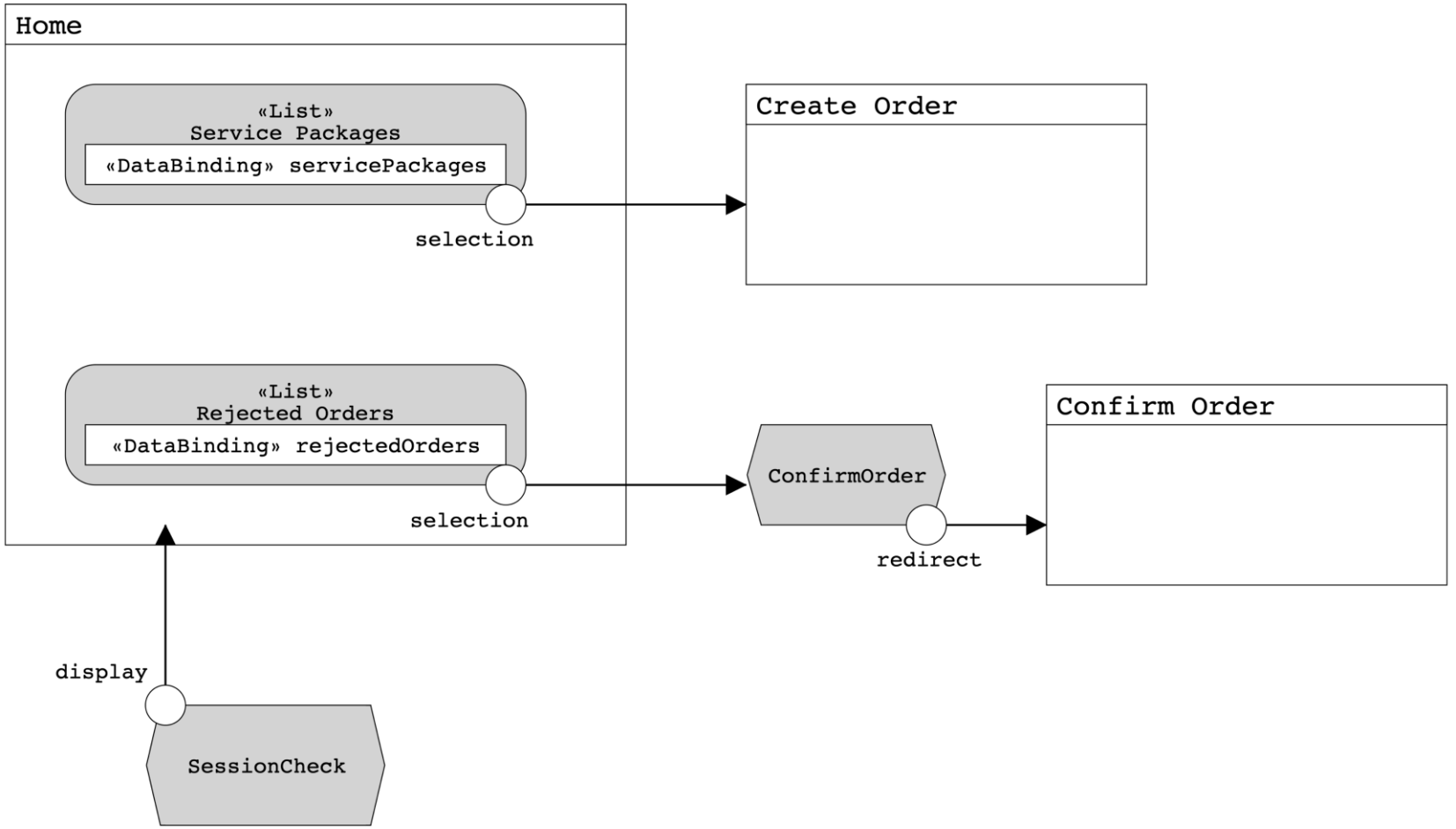
Functional analysis of the interaction

Login, registration and logout (if user is logged in) are available in any moment. Errors and result redirects are performed in the same page the user starts the flow. For this reason the view is identified as ‘Any Page’



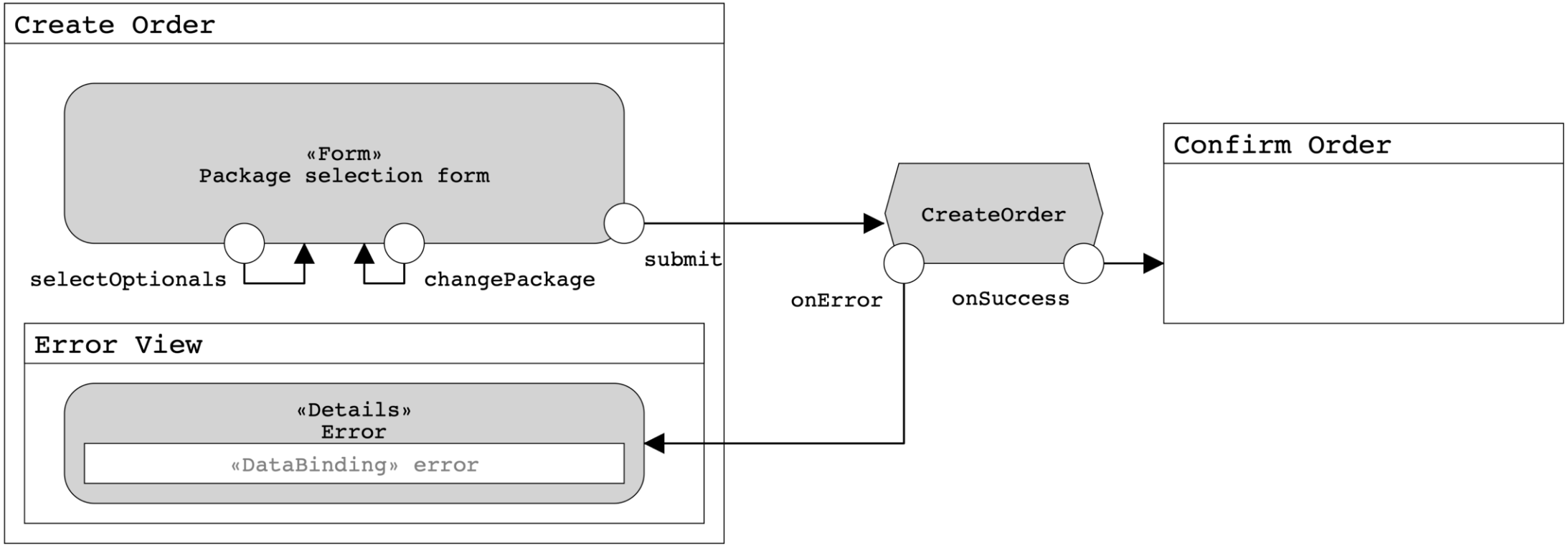
Functional analysis of the interaction

In the home page a user can select a service package to buy it; while, if he is logged in, he can also see the list of his rejected orders and confirms them again.



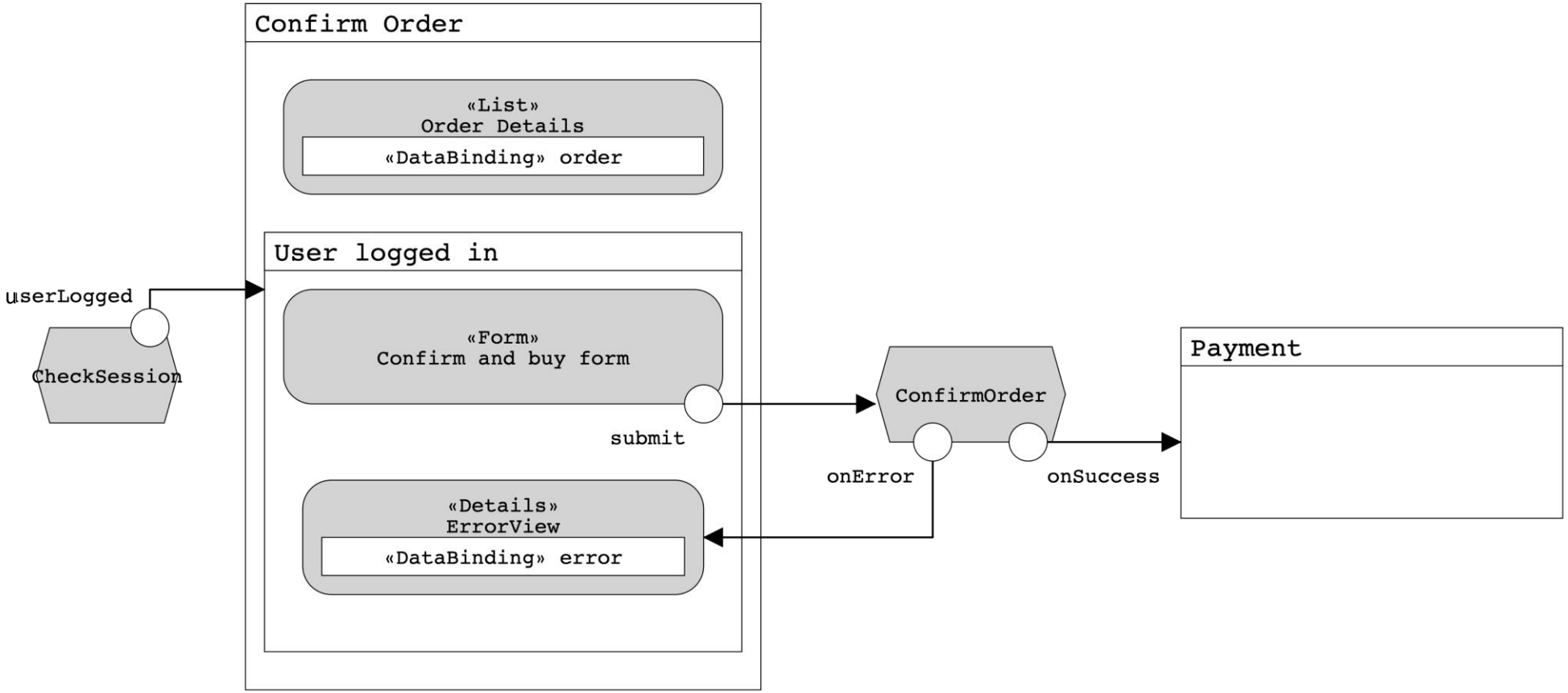
Functional analysis of the interaction

The create order page has the same functionalities for a logged or an unlogged user. A user can change the selected service package and he can also select optional products to buy with. On submit, the order is displayed in the confirmation page.



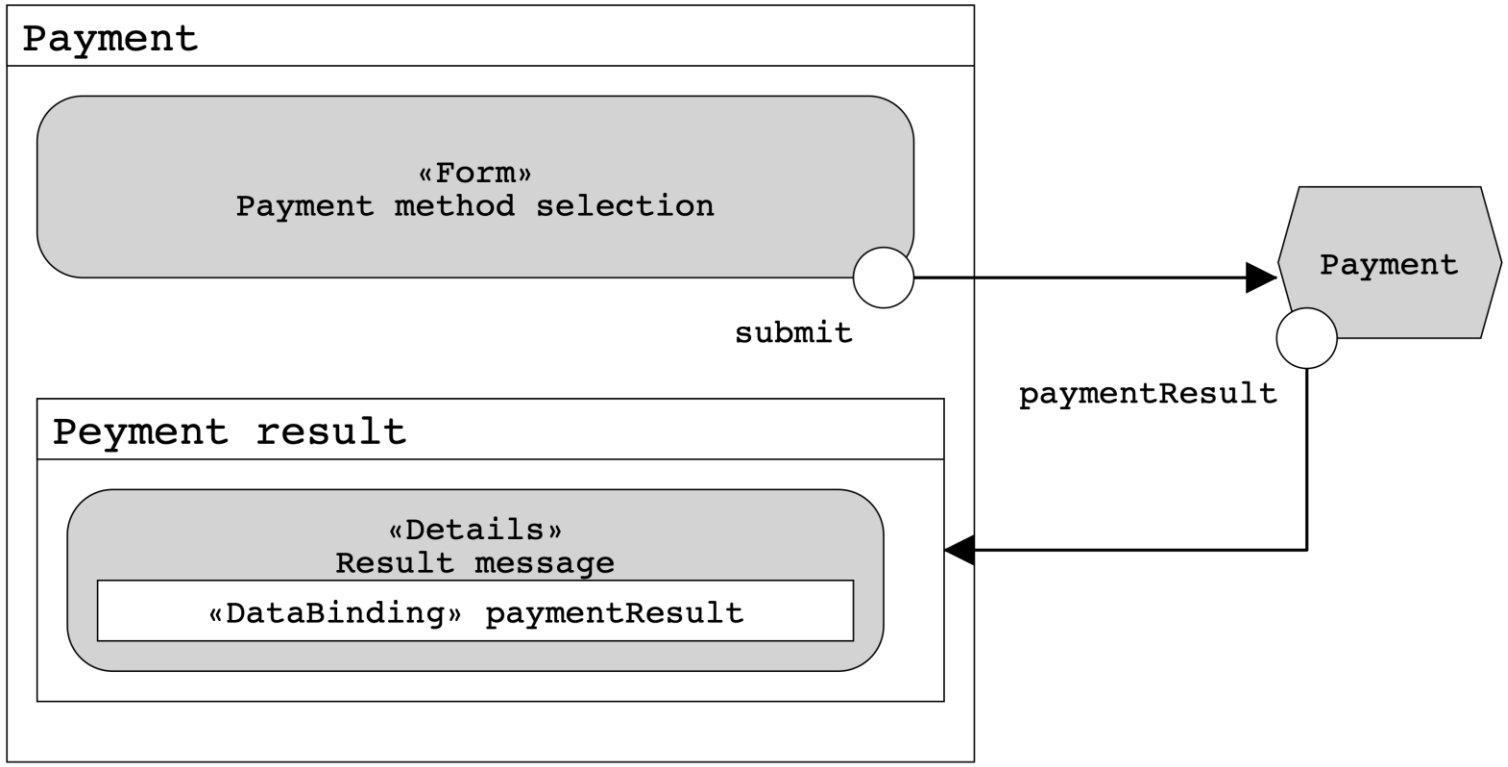
Functional analysis of the interaction

To confirm an order a user has to be logged in. If not he can only see the details of the order but he has to complete the login/registration to proceed. Once an order is confirmed the payment page is displayed.



Functional analysis of the interaction

Finally, a logged user, can access the payment page. Once selected the payment method it is possible to confirm the operation, and the result is then displayed.



Components

Client components

- AdminLoginServlet: handle the login process of an administrator.
- LogoutAdminServlet: handle the logout process of an administrator.
- AdminHomeServlet: retrieve from the DB all the information useful for the dashboard page.
- AdminOptionalProductServlet: let the administrator create a new optional product, checking if it is already existing or not.
- AdminServicePackageServlet: let the administrator create a new service package, checking if it is already existing or not.
- AdminServiceServlet: let the administrator create a new service, checking if it is already existing or not.
- AdminStatsServlet: retrieve from the DB all the information stored in all the materialized view.

Client components

- RegisterServlet: handle the register process of a user.
- LoginServlet: handle the login process of a user.
- LogoutServlet: handle the logout process of a user.
- HomeServlet: retrieve from the DB all the service packages and all the orders in pending of the current user (if it is logged).
- CreateOrderServlet: retrieve in GET all the service package and do a POST when the user has fill all the mandatory fields (creating a pending order and setting it in the session).
- ConfirmOrderServlet: retrieve in GET the order from the session and show all its parameters. A POST is done when the logged user confirm the order and an order entity is stored in the DB. If the order is a rejected one, instead of managing the order from the session, it reads it from the DB.
- PaymentServlet: retrieve and show in GET all the information of an order that the user has to pay. A POST is done when the user tries to do a payment and the servlet save in the DB the result of the current payment (modifying also the status of the order if the payment has been successful (1) or if the payment has failed (2)).

Views

- adminLogin.html: where the administrator logs in. Mapped at “/admin/login”.
- adminDashboard.html: where the administrator can create service packages, optional products or services. Mapped at “admin/dashboard”.
- adminStats.html: where the administrator check the statistics retrieved from the materialized views. Mapped at “admin/stats”.
- index.html: the landing page where are present all the service packages available. Mapped at “/”.
- buyService.html: where the user fill all the required parameters to buy a service package. Mapped at “/order/create/pId=?”.
- confirmationPage.html: where the user confirm the order done in the previous page. Mapped at “/order/confirm”.
- paymentPage.html: where the user can select the preferred payment method and then pay. Mapped at “/order/pay?orderId=?”.
- paymentResult.html: where the user can see if the payment of his order has been successful or not. Mapped at “/order/pay”.

Java beans

- OptionalProductBean: This bean is being used when we call the API, in order to retrieve only the relevant data.
- PendingOrderBean: This bean is being used when we need to handle an order in the session, before the user confirm it (persisting it).

Business tier

- @Stateless AdministratorService
  - getAdministratorByEmail(String)
- @Stateless AlertService
  - getAllAlerts()
- @Stateless AveragePurchaseOptionalPackageService
  - getAllAveragePurchaseOptionalPackages()
- @Stateless InsolventUserService
  - getAllInsolventUsers()
- @Transactional OptionalProductService
  - getOptionalProductById(Integer)
  - isOptionalProductAlreadyExisting(String)
  - getAllOptionalProducts()
  - createOptionalProduct(OptionalProduct)
- @Transactional OrderService
  - getOrderById(Integer)
  - getOrdersOfUser(Integer)
  - createOrder(Order)
- @Transactional PaymentService
  - makePayment(PaymentHistory)
- @Transactional ServicePackageService
  - getServicePackageById(Integer)
  - getServicePackageByName(String)
  - isServicePackageNameAlreadyExist(name)
  - getAllServicePackages()
  - createServicePackage(ServicePackage)

Business tier

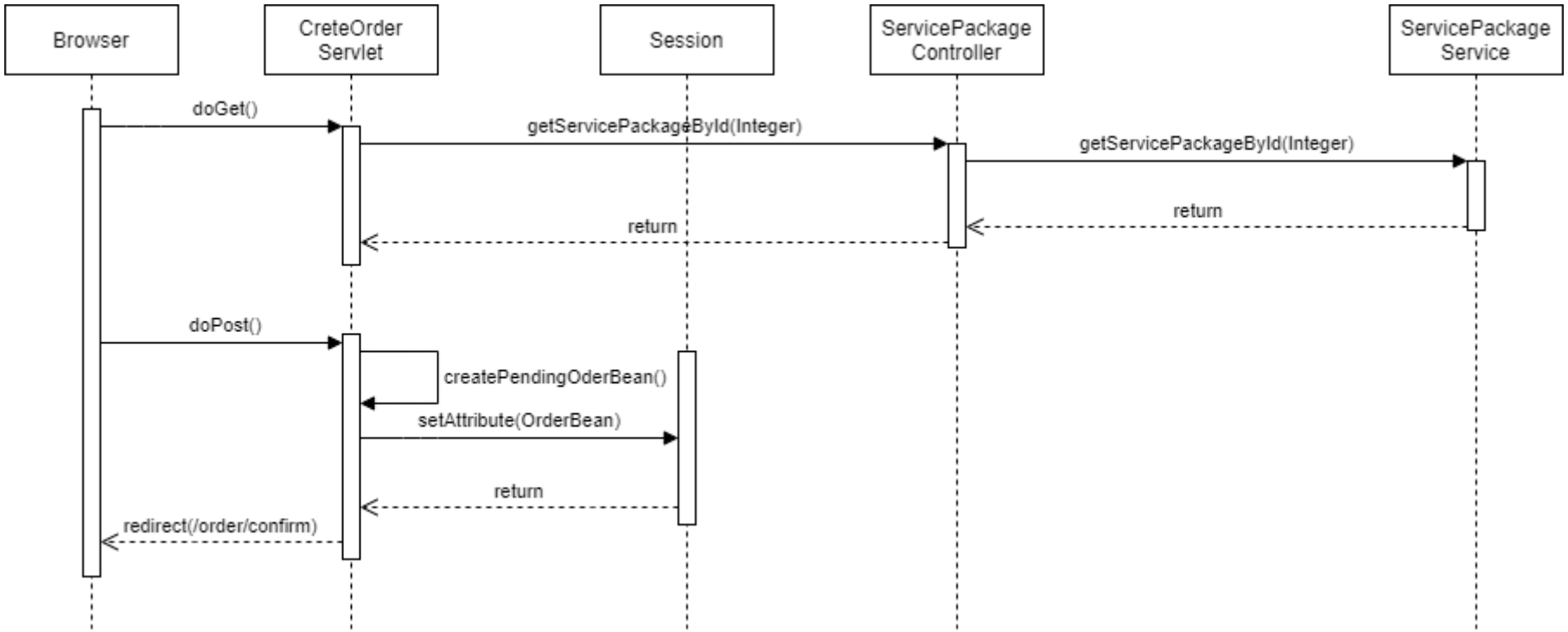
- @Stateless ServiceService
  - getServiceById(Integer)
  - getServiceByName(String)
  - isServiceAlreadyExisting(String)
  - getAllServices()
  - createService(Service)
- @Stateless SuspendedOrderService
  - getAllSuspendedOrders()
- @Stateless TotalPurchaseOptionalService
  - getAllTotalPurchaseOptional()
- @Stateless TotalPurchasePackageOptionalService
  - getAllTotalPurchasePackageOptional()
- @Stateless TotalPurchasePackageService
  - getAllTotalPurchasePackages()
- @Stateless TotalPurchasePackageValidityService
  - getAllTotalPurchasePackageValidity()
- @Stateless TotValueOptionalNoOptionalService
  - getAllTotValueOptionalNoOptional()
- @Stateless UserService
  - getUserById(Integer)
  - getUserByEmail(String)
  - getUserByUsername(String)
  - checkUsername(String)
  - checkEmail(String)
  - createUser(User)

Additional info of the components design

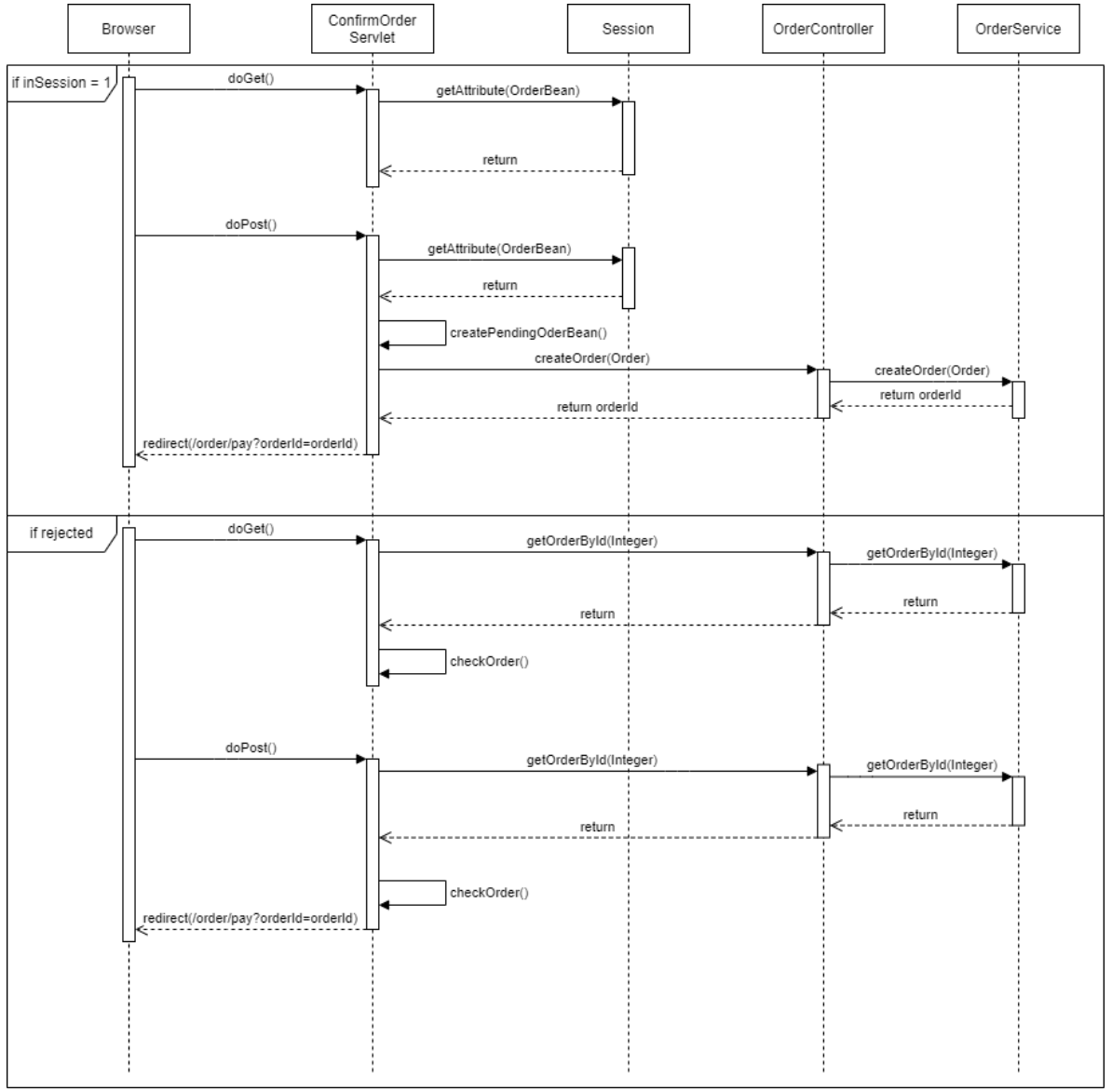
- We have implemented different controllers that call the services described in the slides before, but those controllers have the same name and do the same job of the correspondent service (performing additional controls), so we have decided to not describe them.
- We have implemented a single API function “getOptionalProductByPackageId” that retrieves the optional products of a selected Service Package.
  This API has the following path: “/api/optional/package/{id: [0-9]+}”
- We have implemented different exceptions that are catch when something unexpected happens.
- In the Business Tier we have chosen @Transactional when the correspondent service needs to perform multiple queries in one transaction and using this notation when a query fails, the whole transaction will rollback.

UML sequence diagrams

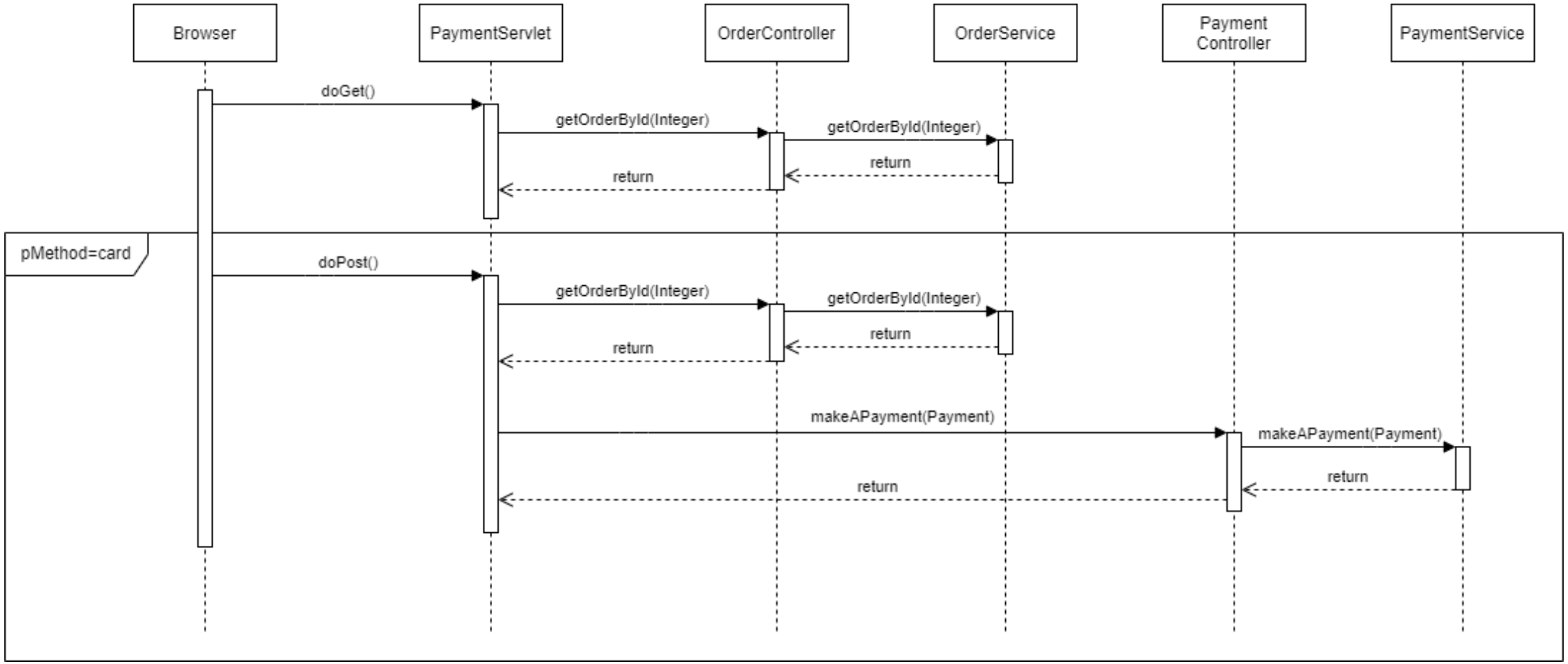
Create order



Confirm order



Payment


Browser
PaymentServlet
OrderController
OrderService
Payment Controller
PaymentService
doGet()
getOrderById(Integer)
getOrderById(Integer)
return
return
pMethod=card
doPost()
getOrderById(Integer)
getOrderById(Integer)
return
return
makeAPayment(Payment)
makeAPayment(Payment)
return
return