Data bases 2

Telco Service project documentation

Index

Specifications

We needed to develop 2 client applications using the same database. The customer application allows the user to login or register(if it's a new customer) but also navigate the home page without logging in. The home page allows the customer to see all the service, service package, optional products and validity periods available and create an order specifying the service package, one validity period and zero or more optional products. From this point the user have to log in to create and pay the order. If the user is logged he can also access a page where he can see all his order and their status, he can also see on the home page all the orders that are not in the "approved" status with a direct link to retry the payment.

Specifications

The employee application allows the employee to log in. After logging in the user sees a page with all the service packages, services, optional products and validity periods created.

In the same page the user can find two forms one for the creation of a new optional product and one for the creation of a new service package.

Clicking on a link the user can see the sales report page where he can request report specifying parameters using forms, but can also see more static reports like the list of insolvent users or the best seller optional product.

In all the pages the logged users can find a link to log out.

Specification interpretation

We designed the application following the specifications and adding details and features that we found useful for debugging. We tried to keep the code and architecture as scalable as possile to be able to add plausible features and options without having to rebuild the project.

When a non-logged user compiles a form for creating the order, the order is not actually created (since an order has to be associated with the user that makes it), but the order information is saved in some cookies so that the user is redirected to the payment page after logging in without losing the order. After logging-in the user is automatically redirected to the CreateOrder page so that the order is finally inserted to the database.

Specification interpretation

The order status can have 3 values: "waiting" if the order is created but no payment attempt is done yet, "approved" if the order has been payed, "rejected" if the payment of the order has been rejected one or more times.

The service activation is an entry in the service_activation table created automatically thanks to a trigger activated after the update of a tuple on the order table. This table is not visible at the front-end of the project as it is but it can be easly added if possible future expansions of the project.

The alert auditing table is a table populated by a trigger that is activated as soon as the number of failed payments on a single order is set to 3.

Cookies

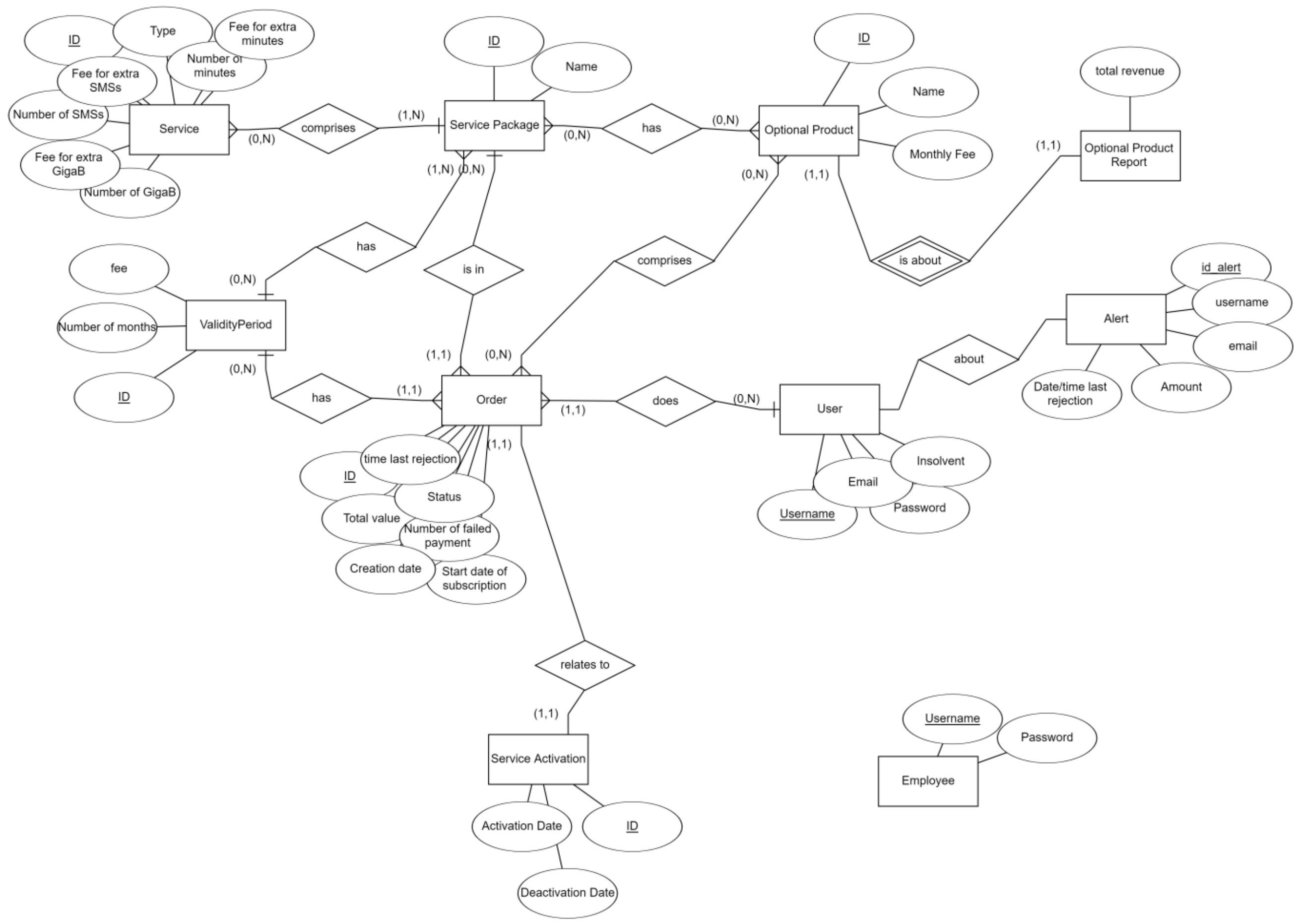
In order to exchange data among some servlets we used cookies. In particular:

- In order to store the order when a guest (not logged customer) issues an order, 4 cookies are created:
  - sp: contains the id of the service package
  - vp: contains the id of the validity period
  - op: contains a list of optional product ids separated by '-'
  - sd: contains the date of the wanted activation of the products in the format "yyyy-MM-dd"
- In order to store the id of the order to show in the Confirmation page we used the "order_to_see" cookie.

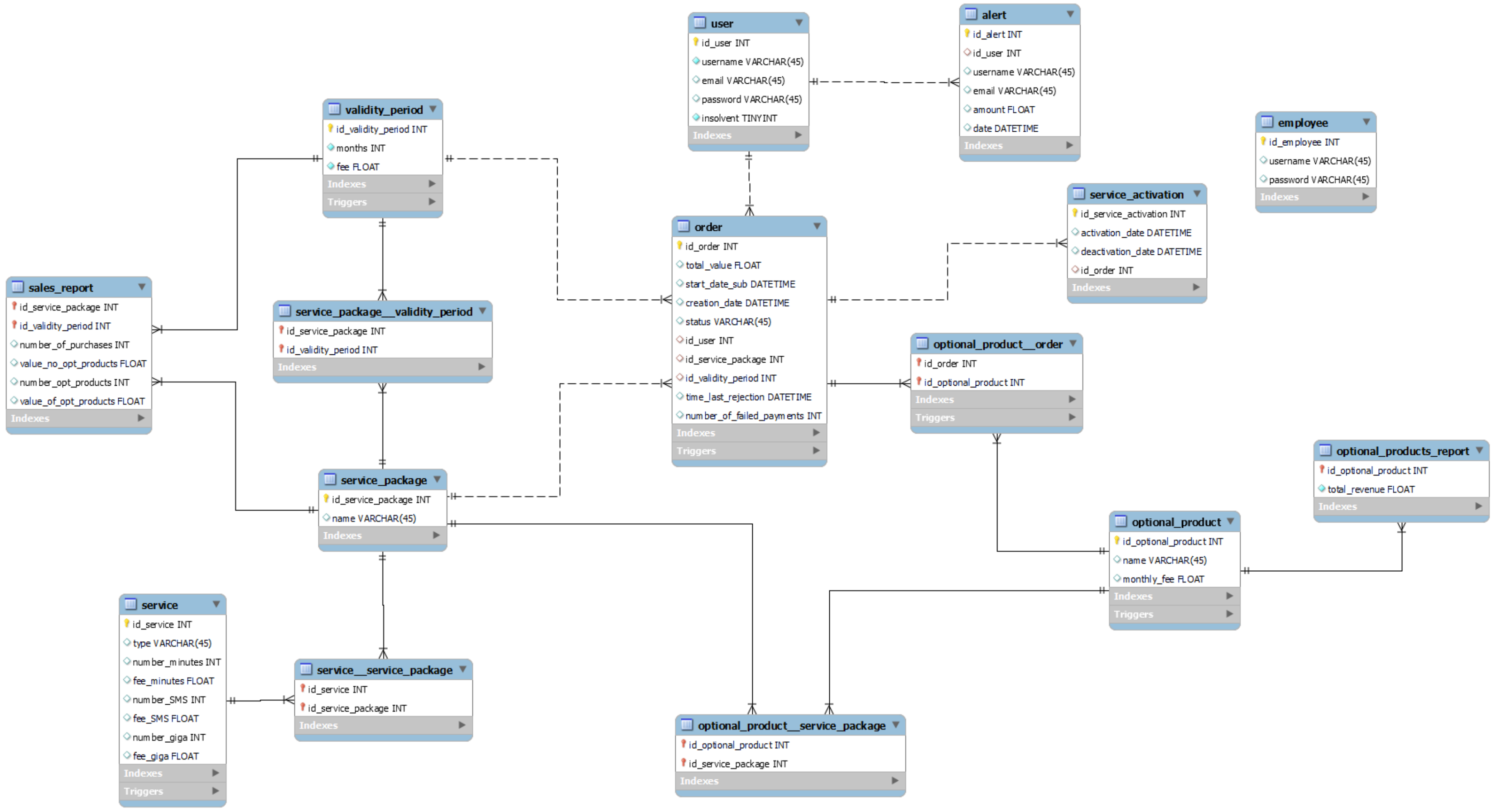
However, all the cookie values are checked and it is not possible to retrieve data not dedicated to the user logged or to create orders with non-coherent data.

Database structure design

Entity Relationship preliminary design



Entity Relationship final design



Motivations of the ER design

We made a separated table for employees because we wanted to separate the two concept, an employee can be also a customer and have an account as customer. Also we thought that an employee works for the company so the company gives him a username and a password to log in and all his informations (like email, address ecc.) are registered by the company somewhere else.

A service package is associated with one or more validity periods, one or more services and zero or more optional products.

An order is associated with one service package, one validity period, and zero or more service packages. Since one service package can be purchased by many customers those are one to many (or many to many) relationships.

A service activation record is associated with an order since all the services and possible optional products in an order have the same start of subscription and the same validity period.

Motivations of the ER design

The sales_report table and optional_product _report tables are materialized view used to make the generation of reports easier, they are populated by triggers that are presented in the next slides.

The alert table is the auditing table populated by a trigger that is activated when an order reaches the limit of 3 failed payments.

Relational model

```
CREATE TABLE `alert` (
  `id_alert` int NOT NULL AUTO_INCREMENT,
  `id_user` int DEFAULT NULL,
  `username` varchar(45) DEFAULT NULL,
  `email` varchar(45) DEFAULT NULL,
  `amount` float DEFAULT NULL,
  `date` datetime DEFAULT NULL,
  PRIMARY KEY (`id_alert`),
  KEY `fk_user_a_idx` (`id_user`),
  CONSTRAINT `fk_user_a` FOREIGN KEY (`id_user`) REFERENCES `user` (`id_user`)
)
```

Relational model

```
CREATE TABLE `employee` (
  `id_employee` int NOT NULL AUTO_INCREMENT,
  `username` varchar(45) DEFAULT NULL,
  `password` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`id_employee`)
)

CREATE TABLE `optional_product` (
  `id_optional_product` int NOT NULL AUTO_INCREMENT,
  `name` varchar(45) DEFAULT NULL,
  `monthly_fee` float DEFAULT NULL,
  PRIMARY KEY (`id_optional_product`)
)
```

Relational model

```
CREATE TABLE `optional_product__order` (
  `id_order` int NOT NULL,
  `id_optional_product` int NOT NULL,
  PRIMARY KEY (`id_order`,`id_optional_product`),
  KEY `id_optional_product_idx` (`id_optional_product`),
  CONSTRAINT `fk_optional_product_op-o` FOREIGN KEY (`id_optional_product`)
REFERENCES `optional_product` (`id_optional_product`),
  CONSTRAINT `fk_order_op-o` FOREIGN KEY (`id_order`) REFERENCES `order` (`id_order`)
)
```

Relational model

```
CREATE TABLE `optional_product__service_package` (
  `id_optional_product` int NOT NULL,
  `id_service_package` int NOT NULL,
  PRIMARY KEY (`id_optional_product`,`id_service_package`),
  KEY `fk_service_package_idx` (`id_service_package`),
  CONSTRAINT `fk_optional_product_op-sp` FOREIGN KEY
(`id_optional_product`) REFERENCES `optional_product` (`id_optional_product`),
  CONSTRAINT `fk_service_package_op-sp` FOREIGN KEY
(`id_service_package`) REFERENCES `service_package` (`id_service_package`)
)
CREATE TABLE `optional_products_report` (
  `id_optional_product` int NOT NULL,
  `total_revenue` float NOT NULL DEFAULT '0',
  PRIMARY KEY (`id_optional_product`),
  CONSTRAINT `fk_optional_products` FOREIGN KEY (`id_optional_product`)
REFERENCES `optional_product` (`id_optional_product`)
)
```

Relational model

```
CREATE TABLE `order` (
  `id_order` int NOT NULL AUTO_INCREMENT,
  `total_value` float DEFAULT '0',
  `start_date_sub` datetime DEFAULT NULL,
  `creation_date` datetime DEFAULT NULL,
  `status` varchar(45) DEFAULT 'default',
  `id_user` int DEFAULT NULL,
  `id_service_package` int DEFAULT NULL,
  `id_validity_period` int DEFAULT NULL,
  `time_last_rejection` datetime DEFAULT NULL,
  `number_of_failed_payments` int DEFAULT '0',
  PRIMARY KEY (`id_order`),
  KEY `id_user_idx` (`id_user`),
  KEY `fk_service_package_idx` (`id_service_package`),
  KEY `fk_v_p_idx` (`id_validity_period`),
  CONSTRAINT `fk_servicePackage_o` FOREIGN KEY (`id_service_package`) REFERENCES
`service_package` (`id_service_package`),
  CONSTRAINT `fk_user_o` FOREIGN KEY (`id_user`) REFERENCES `user` (`id_user`),
  CONSTRAINT `fk_validityPeriod_o` FOREIGN KEY (`id_validity_period`) REFERENCES `validity_period`
(`id_validity_period`)
)
```

Relational model

```
CREATE TABLE `sales_report` (
  `id_service_package` int NOT NULL,
  `id_validity_period` int NOT NULL,
  `number_of_purchases` int DEFAULT '0',
  `value_no_opt_products` float DEFAULT '0',
  `number_opt_products` int DEFAULT '0',
  `value_of_opt_products` float DEFAULT '0',
  PRIMARY KEY (`id_service_package`,`id_validity_period`),
  KEY `fk_validity_period_sr_idx` (`id_validity_period`),
  CONSTRAINT `fk_service_package_sr` FOREIGN KEY (`id_service_package`)
REFERENCES `service_package` (`id_service_package`),
  CONSTRAINT `fk_validity_period_sr` FOREIGN KEY (`id_validity_period`) REFERENCES
`validity_period` (`id_validity_period`)
)
```

Relational model

```
CREATE TABLE `service` (
  `id_service` int NOT NULL AUTO_INCREMENT,
  `type` varchar(45) DEFAULT NULL,
  `number_minutes` int DEFAULT NULL,
  `fee_minutes` float DEFAULT NULL,
  `number_SMS` int DEFAULT NULL,
  `fee_SMS` float DEFAULT NULL,
  `number_giga` int DEFAULT NULL,
  `fee_giga` float DEFAULT NULL,
  PRIMARY KEY (`id_service`)
)
```

Relational model

```
CREATE TABLE `service_activation` (
  `id_service_activation` int NOT NULL AUTO_INCREMENT,
  `activation_date` datetime DEFAULT NULL,
  `deactivation_date` datetime DEFAULT NULL,
  `id_order` int DEFAULT NULL,
  PRIMARY KEY (`id_service_activation`),
  KEY `fk_ser_idx` (`id_order`),
  CONSTRAINT `fk_order_sa` FOREIGN KEY (`id_order`) REFERENCES `order`
(`id_order`) ON DELETE CASCADE ON UPDATE CASCADE
)
```

Relational model

```
CREATE TABLE `service_package` (
  `id_service_package` int NOT NULL AUTO_INCREMENT,
  `name` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`id_service_package`)
)

CREATE TABLE `service_package__validity_period` (
  `id_service_package` int NOT NULL,
  `id_validity_period` int NOT NULL,
  PRIMARY KEY (`id_service_package`,`id_validity_period`),
  KEY `to_validity_period_idx` (`id_validity_period`),
  CONSTRAINT `fk_service_package_sp-vp` FOREIGN KEY (`id_service_package`)
REFERENCES `service_package` (`id_service_package`),
  CONSTRAINT `fk_validity_period_sp-vp` FOREIGN KEY (`id_validity_period`)
REFERENCES `validity_period` (`id_validity_period`)
)
```

Relational model

```
CREATE TABLE `service__service_package` (
  `id_service` int NOT NULL,
  `id_service_package` int NOT NULL,
  PRIMARY KEY (`id_service`,`id_service_package`),
  KEY `to_service_package_idx` (`id_service_package`),
  CONSTRAINT `fk_service_package_s-sp` FOREIGN KEY (`id_service_package`)
REFERENCES `service_package` (`id_service_package`),
  CONSTRAINT `fk_service_s-sp` FOREIGN KEY (`id_service`) REFERENCES `service`
(`id_service`)
)
CREATE TABLE `user` (
  `id_user` int NOT NULL AUTO_INCREMENT,
  `username` varchar(45) NOT NULL,
  `email` varchar(45) DEFAULT NULL,
  `password` varchar(45) DEFAULT NULL,
  `insolvent` tinyint NOT NULL DEFAULT '0',
  PRIMARY KEY (`id_user`)
)
```

Relational model

```
CREATE TABLE `validity_period` (
  `id_validity_period` int NOT NULL AUTO_INCREMENT,
  `months` int NOT NULL,
  `fee` float NOT NULL,
  PRIMARY KEY (`id_validity_period`)
)
```

Motivations of the logical design

We used a single table to represent all the services because we assumed that even in future expansions of the application with different kind of services we will have no need to specify other information than those about minutes, gigabytes and sms.

The order table has as attributes the start date of subscription because it's chosen by the customer when making the order, but the tuple in the service activation table is generated only when the order status is "approved".

Triggers design

Trigger design & code

All of our triggers are with row-level granularity of events, because every query always have effect on at most one row for each table.

The concept behind the subdivision among BEFORE and AFTER triggers comes from the willing to optimize the changes done on the database. The BEFORE triggers usually do a validity checking on data and changes values on the new tuple that we want to insert/modify. The AFTER triggers usually update other tables/materialized-views so that we modify them only if a commit has happened.

Most of the triggers are used to populate the materialized-views (sales_reports and optional_product_reports), other triggers are used to check integrity of the data before inserting a new tuple or after updating one.

We haven't managed the deleting of tuples since in the application and in the back-end there isn't this possibility.

Triggers on optional product table

AFTER INSERT

Update the materialized view used for retrieving the best seller optional product. Update the materialized view creating a new tuple for that optional product and initializig the value of sales to 0.

```
CREATE DEFINER=`root`@`localhost` TRIGGER
   `optional_product_AFTER_INSERT` AFTER INSERT ON
   `optional_product` FOR EACH ROW BEGIN


INSERT INTO `optional_products_report` VALUES
   (new.id_optional_product, 0);

END
```

Trigger on join-table optionalproduct_order

BEFORE INSERT

Before inserting a new tuple check if the selected optional product is associated with the specific service package of the order, it shouldn't be possible to create an order with the wrong optional product since the front-end makes you choose only between the optional products associated with the selected service package and also the back-end checks before creating the order, but we added this trigger to make the database more solid.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `optional_product__order_BEFORE_INSERT` BEFORE INSERT ON
    `optional_product__order` FOR EACH ROW BEGIN

-- checking if the optional product we want to add is in the proposed ones for the service package

IF(    new.id_optional_product NOT IN (SELECT id_optional_product

   FROM `optional_product__service_package` op_sp

   WHERE op_sp.id_service_package = (

      SELECT id_service_package FROM `order` o

      WHERE o.id_order = new.id_order

   ))) THEN

   SIGNAL SQLSTATE '45000'

   SET MESSAGE_TEXT = "Can't add an optional product that isn't offered by the service package!";

END IF;

END
```

Trigger on join-table optionalproduct_order

AFTER INSERT

After inserting the tuple the price of the order is automatically calculated retrieving the price of optional products and the service package.

Also the materialized view for sales report and bestseller optional products are updated incrementing with the values of the new order.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `optional_product__order_AFTER_INSERT` AFTER INSERT ON `optional_product__order`
    FOR EACH ROW BEGIN

-- update order only if its status is "waiting"

IF(SELECT `status` FROM `order` o WHERE o.id_order = new.id_order) = "waiting" THEN

  -- set variables

  SET @months = 0; SET @fee_opt_product = 0; SET @id_service_package = 0;  SET @id_validity_period = 0;

  SELECT vp.months, id_service_package, id_validity_period INTO @months, @id_service_package, @id_validity_period

    FROM `order` o NATURAL JOIN `validity_period` vp

    WHERE o.id_order = new.id_order;

  SELECT monthly_fee INTO @fee_opt_product

    FROM `optional_product` op

    WHERE op.id_optional_product = new.id_optional_product
```

Trigger on join-table optionalproduct_order

```
MORE CODE
-- update total_value of the order
    UPDATE `order` o SET
        total_value = total_value + @months * @fee_opt_product
    WHERE o.id_order = new.id_order;
 -- update materialized view of sales
    UPDATE `sales_report` sp SET
        number_opt_products = number_opt_products + 1,
        value_of_opt_products = value_of_opt_products + @months * @fee_opt_product
    WHERE @id_service_package = sp.id_service_package and @id_validity_period = sp.id_validity_period;
      -- update materialized view of optional products
    UPDATE `optional_products_report` opr SET
        opr.total_revenue = opr.total_revenue + @months * @fee_opt_product
        WHERE new.id_optional_product = opr.id_optional_product;
ELSE
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = "You can't update the optional products of a non-waiting order!";
END IF;
END
```

Triggers on table order

BEFORE INSERT

We check that the selected validity period is in the proposed one for that service package (still it shouldn't be possible to choose one). Then we update the sales report adding the new values of purchase and the new number of purchase or we create a new tuple if one with that specific service package id and validity period id if it doesn't already exist. Then we automatically calculate the creation date of the order, and based on the status we initialize the number of failed payments and if the user is an insolvent one.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `order_BEFORE_INSERT` BEFORE INSERT ON `order` FOR EACH ROW BEGIN

-- checking that the validity period selected is one of the proposed ones

IF(new.id_validity_period IN (SELECT id_validity_period

   FROM `service_package__validity_period` sp_vp

   WHERE sp_vp.id_service_package = new.id_service_package)

   ) THEN

   -- set variables

   SET @months = 0; SET @fee = 0;

SELECT months, fee INTO @months, @fee

      FROM `validity_period` vp

      WHERE vp.id_validity_period = new.id_validity_period;
```

Triggers on table order

MORE CODE

```
  -- sales_report update
  IF (SELECT count(*) FROM `sales_report` sp WHERE new.id_service_package =
   sp.id_service_package and new.id_validity_period = sp.id_validity_period) > 0 THEN
    -- if there is already an entry in sales_report, update it
    UPDATE `sales_report` sp SET
      number_of_purchases = number_of_purchases + 1,
      value_no_opt_products = value_no_opt_products + @months * @fee
    WHERE new.id_service_package = sp.id_service_package and new.id_validity_period =
   sp.id_validity_period;
  ELSE
    -- otherwise create an entry
    INSERT INTO `sales_report` ( id_service_package, id_validity_period, number_of_purchases,
      value_no_opt_products,  number_opt_products,  value_of_opt_products )
VALUES ( new.id_service_package,  new.id_validity_period,  1,  @months * @fee,  0,  0);
  END IF;
```

Triggers on table order

```
-- automatically calculate value and insert start date of subscription
  SET new.`creation_date` = (SELECT NOW() ;  SET new.`total_value` =  @months * @fee;
  IF new.`status` IS null THEN
    SET new.`status` = "waiting";
  END IF;
   IF new.`status` = 'rejected' THEN
    SET new.`number_of_failed_payments` = 1;
  ELSE
    SET new.`number_of_failed_payments` = 0;
  END IF;
   IF new.`status` <> 'accepted' THEN
    UPDATE user u SET insolvent = 1
    WHERE u.id_user = new.id_user;
  END IF;
ELSE
  SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = "Can't select a validity period that isn't offered by the service package!";
END IF;
END
```

Triggers on table order

AFTER INSERT

If the inserted order has an approved status we create a tuple on the service_activation table.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `order_AFTER_INSERT` AFTER INSERT ON `order` FOR EACH ROW BEGIN
IF new.status = "approved" THEN
   -- create a service_activation record
   INSERT service_activation (activation_date, deactivation_date, id_order) VALUES
       (
          (SELECT NOW()),
          (SELECT DATE_ADD((SELECT NOW()),
             INTERVAL (SELECT months FROM `order` o NATURAL RIGHT JOIN  `validity_period` vp
             WHERE o.id_order = new.id_order) MONTH)),
          new.id_order
       );
END IF;
END
```

Triggers on table order

BEFORE UPDATE

We check the update on the order status(since it's the only possibile one) if the new status is "approved" we set the number of failed payments to 0 and generate a new tuple on the service activation table, if the new status is "rejected" and the new time of last rejection is different than the old one, we increment the number of failed payment by one and if the number of failed payment is 3 we mark the user that made the order as insolvent and generate an alert adding a tuple to the auditing table.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `order_BEFORE_UPDATE` BEFORE UPDATE ON `order` FOR EACH
    ROW BEGIN

-- set variables

SET @username = ''; SET @email = '';

SELECT username, email INTO @username, @email

FROM `user` u WHERE u.id_user = old.id_user;

IF new.status = 'approved' THEN

   -- reset number of failed payments counter

   SET new.number_of_failed_payments  = 0;

END IF;
```

Triggers on table order

```
-- check if rejected
IF ((new.time_last_rejection IS NOT null and old.time_last_rejection IS null) or new.time_last_rejection <> old.time_last_rejection)THEN
   SET new.`status` = "rejected"; SET new.number_of_failed_payments = new.number_of_failed_payments + 1
   -- set the user to insolvent
   UPDATE user u SET insolvent = 1
   WHERE u.id_user = new.id_user;
   -- if reached the 3 failed payments, generate an alert
   IF new.number_of_failed_payments = 3 THEN
      INSERT INTO alert (`id_user`, `username`, `email`, `amount`, `date`)
         SELECT old.id_user, @username, @email, old.total_value, old.time_last_rejection
         FROM `order` o
         WHERE o.time_last_rejection = (
            -- selects the most recent rejected order
            select max(time_last_rejection)
            from `order` o
            where o.id_user = old.id_user and o.status = "rejected"
         ) and o.id_user = old.id_user and o.status = "rejected";
   END IF;
END IF;
END
```

Triggers on table order

AFTER UPDATE

If the new status is "approved" and in the table order the number of not approved orders with the same user id is zero we can mark the user as not insolvent and we can create a tuple in the service activation table.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `order_AFTER_UPDATE` AFTER
    UPDATE ON `order` FOR EACH ROW BEGIN
-- if the order has been approved
IF new.status = "approved" THEN
    -- if there are no more rejected orders, reset user insolvent flag
    IF (SELECT count(*) FROM `order` o WHERE o.status <> "approved" AND
    o.id_user = new.id_user) = 0 THEN
        UPDATE `user` u SET insolvent = 0
        WHERE u.id_user = new.id_user;
    END IF;
```

Triggers on table order

```
-- create a service_activation record
    INSERT service_activation (activation_date, deactivation_date, id_order) VALUES
        (
            (SELECT NOW()),
            (SELECT DATE_ADD((SELECT NOW()),
                INTERVAL (SELECT months FROM `order` o NATURAL RIGHT
    JOIN  `validity_period`vp
                WHERE o.id_order = new.id_order) MONTH)),
            new.id_order
        );
END IF;

END
```

Triggers on table order

AFTER DELETE

If after deleting an order there are no more orders with the same user id that are not approved the user must be marked as not insolvent.

(It's not possibile to delete an order from the webapp but we added this trigger to make the database more solid)

```
CREATE DEFINER=`root`@`localhost` TRIGGER `order_AFTER_DELETE` AFTER DELETE ON `order`
    FOR EACH ROW BEGIN
IF old.status = "rejected" THEN
   -- if there are no more rejected orders, update user insolvent flag
   IF (SELECT count(*) FROM `order` o WHERE o.status = "rejected") = 0 THEN
      UPDATE user u SET insolvent = 0, number_of_failed_payments = 0 WHERE u.id_user =
    old.id_user;
   END IF;
END IF;
END
```

Triggers on table service

BEFORE INSERT

Check that the service that is about to be inserted is a valid one setting to zero the attributes that has to be zero.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `service_BEFORE_INSERT` BEFORE INSERT ON `service` FOR EACH ROW BEGIN

-- validating input of services

IF(new.`type` = 'fixed phone') THEN

  SET new.`number_minutes` = 0,  new.`fee_minutes` = 0, new.`number_SMS` = 0, new.`fee_SMS` = 0, new.`number_giga` = 0,new.`fee_giga` = 0;

ELSE IF(new.`type` = 'mobile phone' and new.`number_minutes` IS NOT null and new.`fee_minutes` IS NOT null and new.`number_SMS` IS NOT null and
     new.`fee_SMS` IS NOT null) THEN

  SET new.`number_giga` = 0, new.`fee_giga` = 0;

ELSE IF((new.`type` = 'fixed internet' or new.`type` = 'mobile internet') and new.`number_giga` IS NOT null and new.`fee_giga` IS NOT null) THEN

  SET new.`number_minutes` = 0, new.`fee_minutes` = 0, new.`number_SMS` = 0, new.`fee_SMS` = 0;

ELSE

  SIGNAL SQLSTATE '45000'

  SET MESSAGE_TEXT = "Use type 'fixed phone', 'mobile phone', 'fixed internet', 'mobile internet' with the appropriate mandatory fields";

END IF;

END IF;

END IF;

END
```

Triggers on table validity period

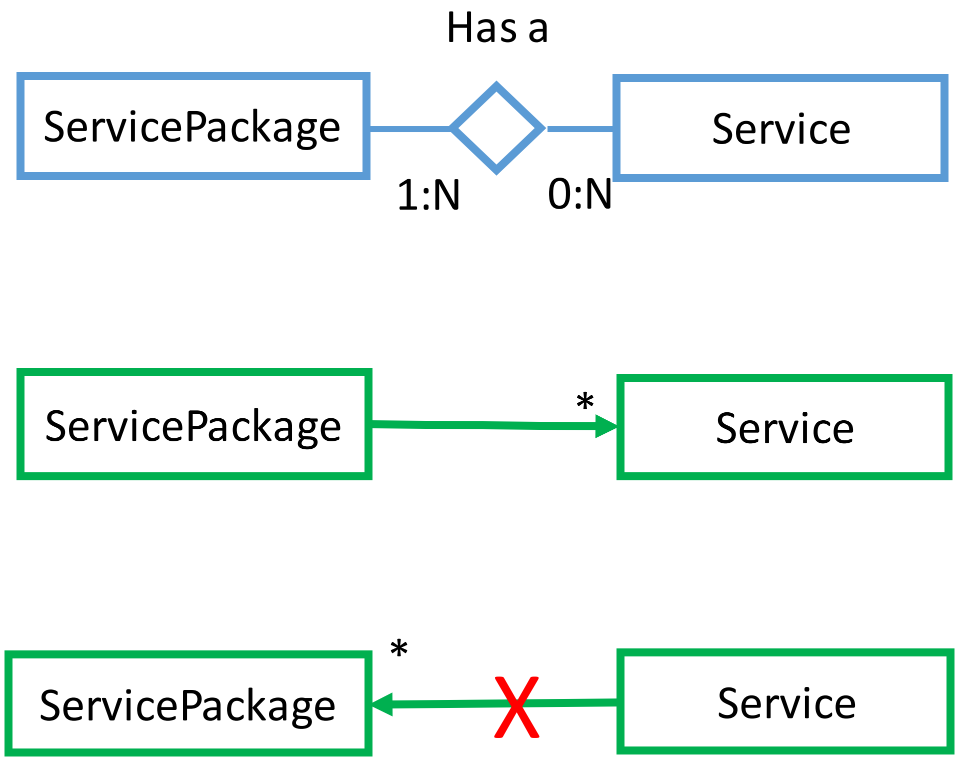
BEFORE INSERT

Check if the validity period that is about to be inserted has an allowed number of months and if it's equivalent to one that already esists in the database.

```
CREATE DEFINER=`root`@`localhost` TRIGGER `validity_period_BEFORE_INSERT` BEFORE INSERT ON
   `validity_period` FOR EACH ROW BEGIN

-- check if the months are a valid number

IF new.months <> 12 and new.months <> 24 and new.months <> 36 THEN

  SIGNAL SQLSTATE '45000'

  SET MESSAGE_TEXT = "The only availble number of months are 12, 24 or 36";

END IF;

-- check if there exist an equal validity period

IF (SELECT count(*) FROM `validity_period` vp WHERE new.months = vp.months and new.fee = vp.fee) THEN

  SIGNAL SQLSTATE '45000'

  SET MESSAGE_TEXT = "There already exist a validity period with the same characteristics";

END IF;

END
```

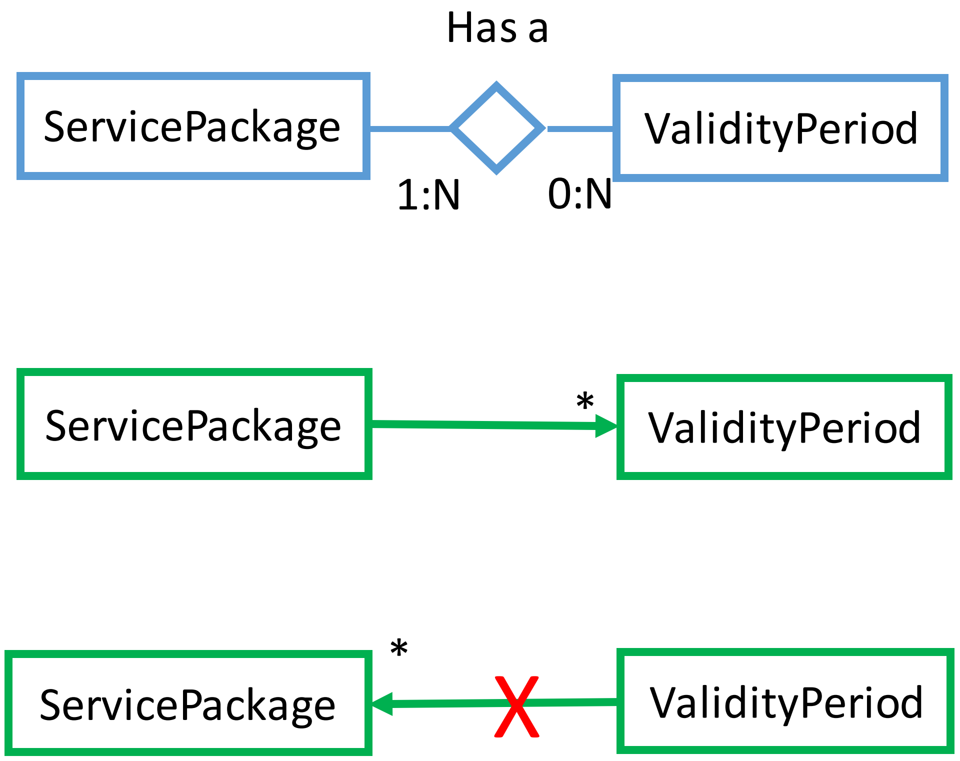
ORM design

Relationship “ServicePackage_service”



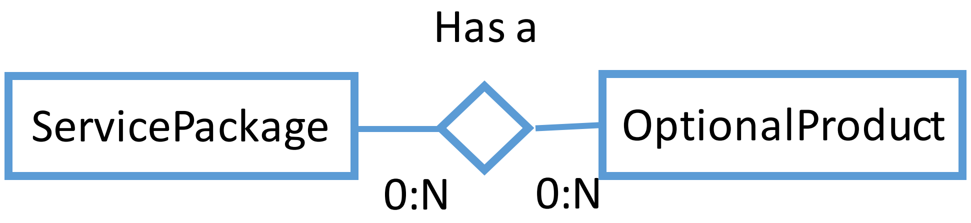
- ServicePackage → Service
  - @ManyToMany is necessary when we want to know the services of a service package
  - Fetch is EAGER because when we take a ServicePackage we want to retrieve all his informations
  - Cascading: REFRESH, so that modifying a Service it will be updated automatically
- Service→ServicePackage
  - Not necessary, we don't want to access all the ServicePackages that implement that service

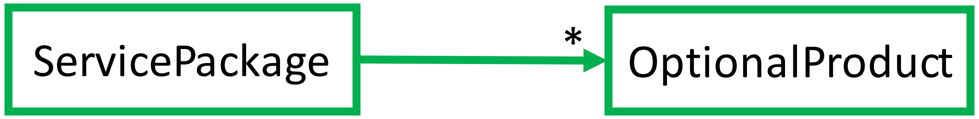
Relationship “ServicePackage_ValidityPeriod ”

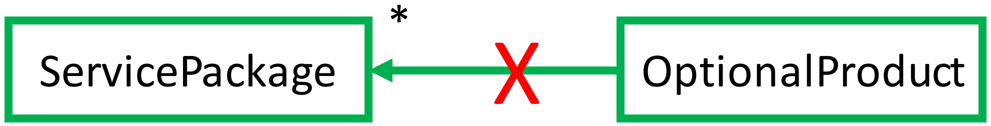


- ServicePackage → ValidityPeriod
    - @ManyToMany is necessary when we want to know the ValidityPeriods available of a service package
    - Fetch is EAGER because when we take a ServicePackage we want to retrieve all his informations
    - Cascading: REFRESH, so that modifying a ValidityPeriod it will be updated automatically
- ValidityPeriod→ServicePackage
    - Not necessary, we don't want to access all the ServicePackages that can have that ValidityPeriod

Relationship “ServicePackage_OptionalProduct”

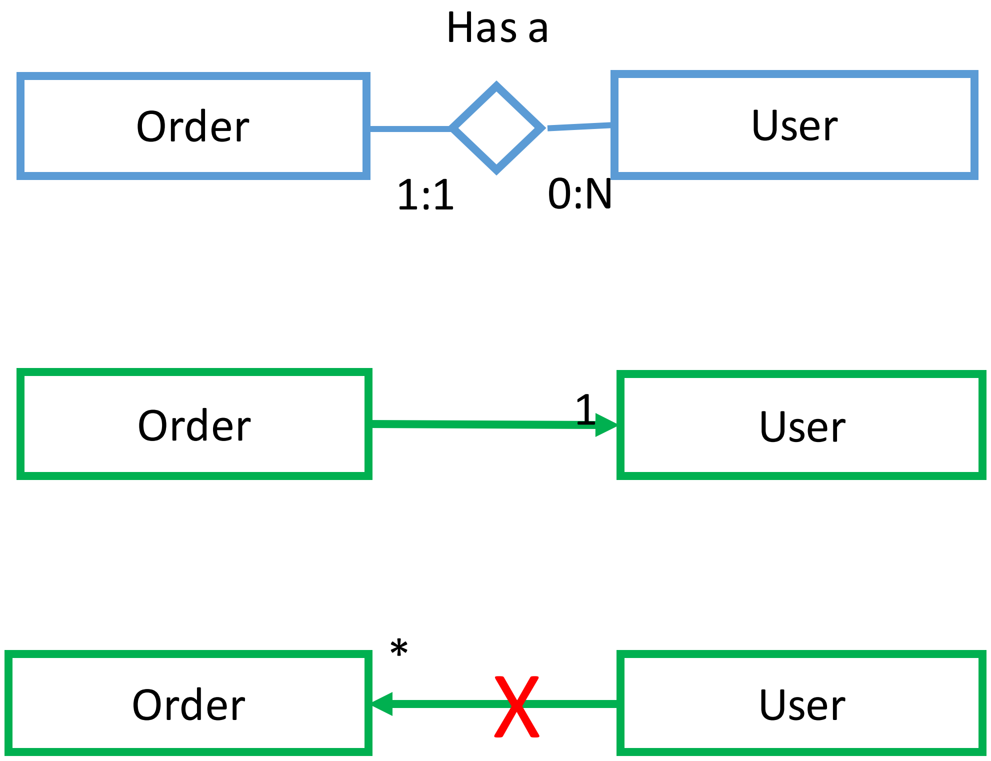






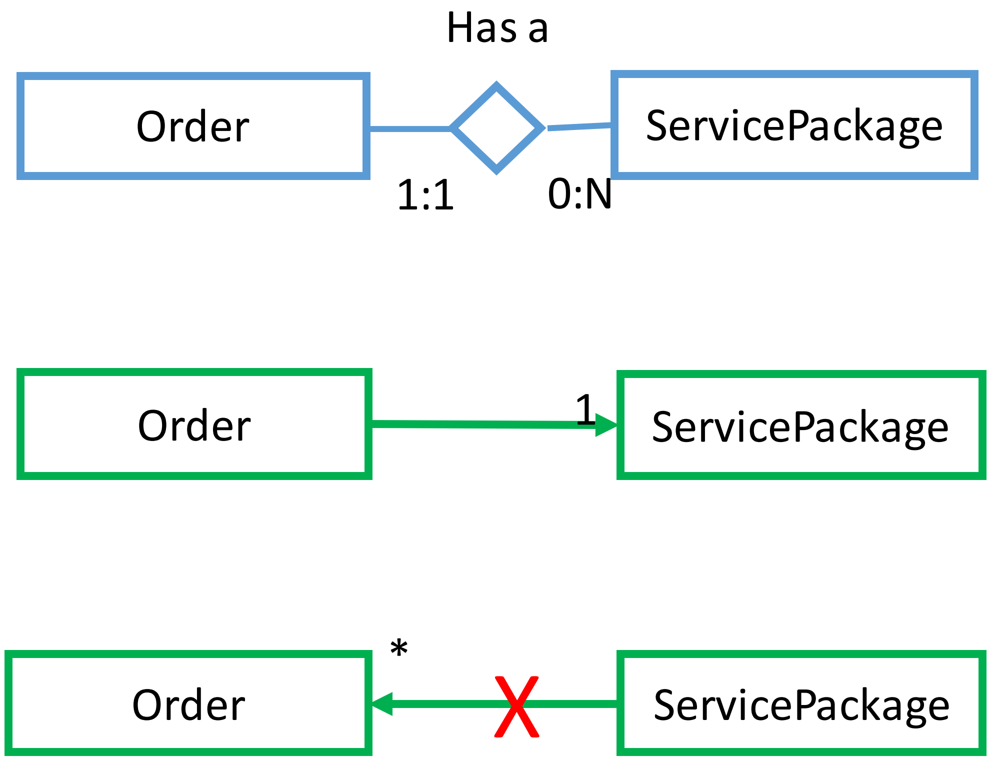
- ServicePackage → OptionalProduct
  - @ManyToMany is necessary when we want to know the optional products available for a service package
  - Fetch is EAGER because when we take a ServicePackage we want to retrieve all his informations
  - Cascading: REFRESH, so that modifying an optional product it will be updated automatically
- OptionalProduct→ServicePackage
  - Not necessary, we don't want to access all the Service Packages that can have that optional product

Relationship “Order_User”



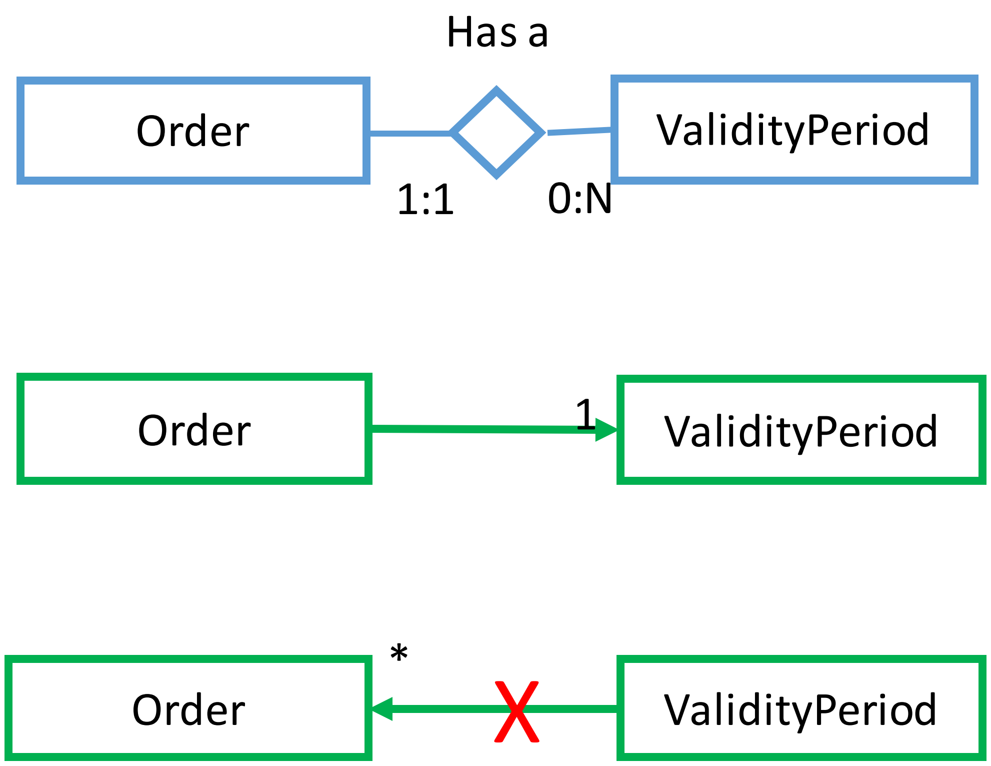
- Order→ User
  - @OneToMany is necessary when we want to know the ValidityPeriods available of a service package
  - Fetch is LAZY because maybe we don't want to access all the User every time
  - Cascading: REFRESH, so that a modified User will be updated automatically when the order is refreshed
- User→Order
  - We do not map the Orders in the user but we use a query to retrieve the correlated Orders

Relationship "Order_ServicePackage"



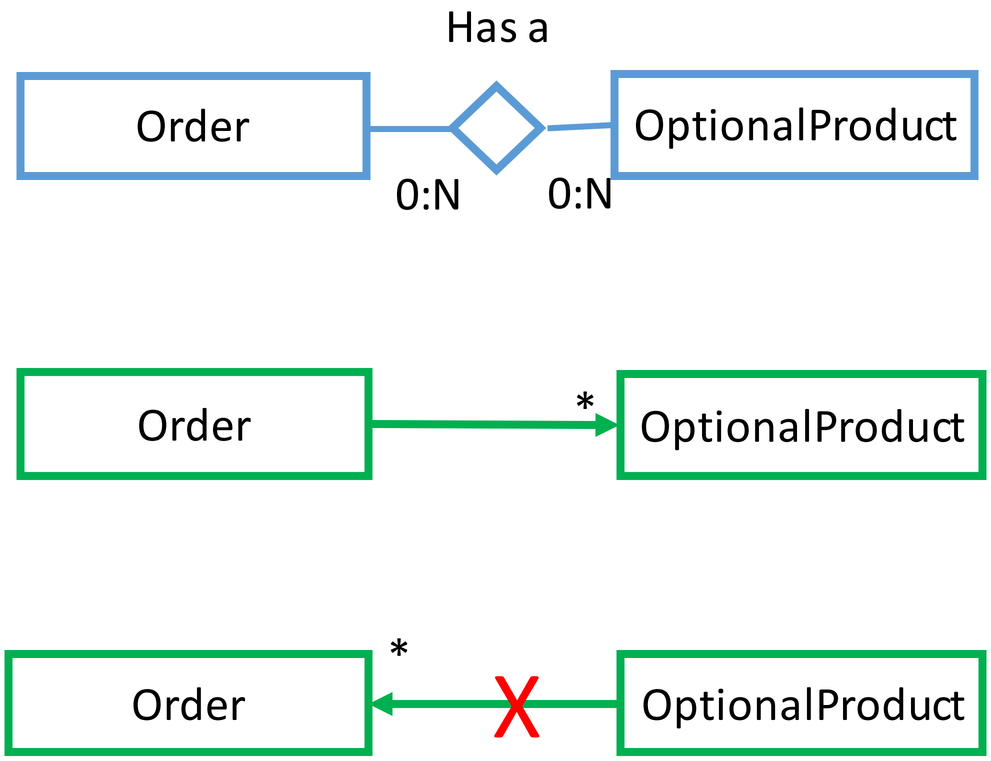
- Order→ ServicePackage
  - @OneToMany is necessary when we want to know the ServicePackage chosen for an Order
  - Fetch is LAZY because maybe we don't want to access the ServicePackage every time
  - Cascading: REFRESH, so that a modification to a ServicePackage will be updated automatically when the order is refreshed
- ServicePackage→Order
  - Not necessary, we don't want to access the Orders that include that ServicePackage

Relationship "Order_ValidityPeriod"



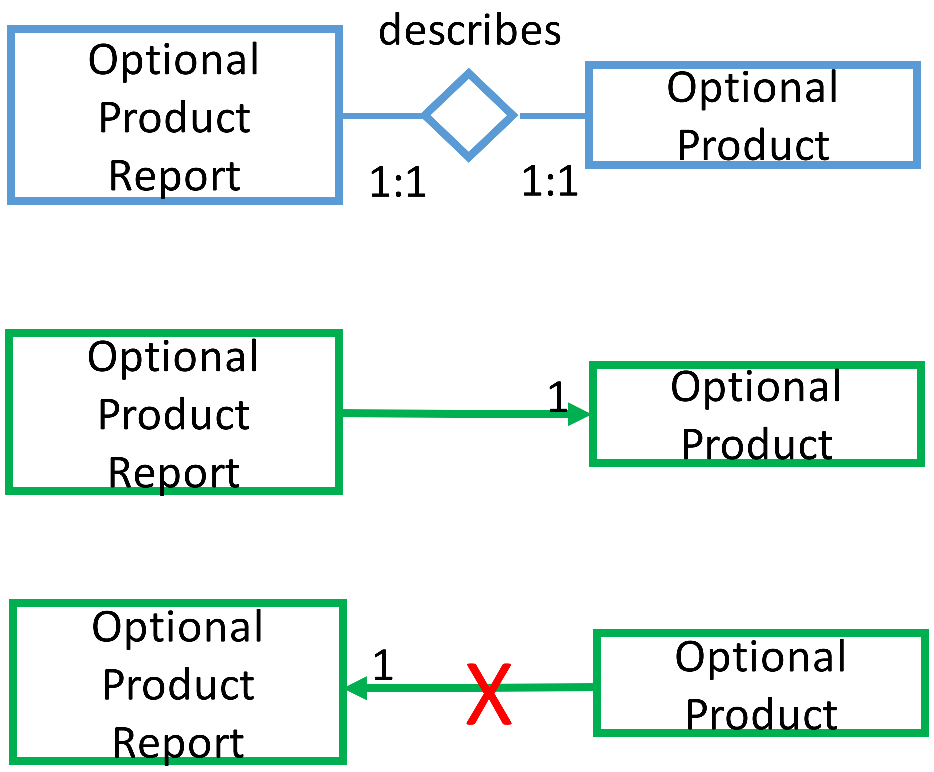
- Order→ ValidityPeriod
  - @OneToMany is necessary when we want to know the ValidityPeriod chosen for an Order
  - Fetch is LAZY because maybe we don't want to access the ValidityPeriod every time
  - Cascading: REFRESH, so that a modification to a validityPeriod will be updated automatically when the order is refreshed
- ValidityPeriod→Order
  - Not necessary, we don't want to access the Orders that use that ValidityPeriod

Relationship “Order_OptionalProduct”



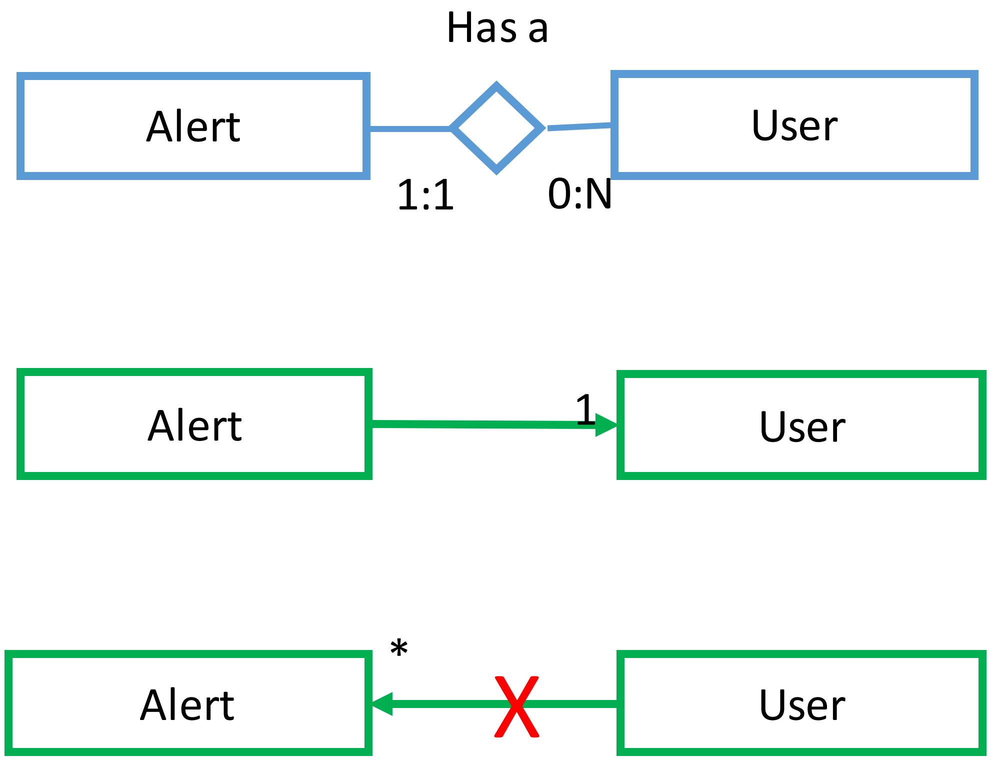
- Order→ OptionalProduct
  - @ManyToMany is necessary when we want to know the OptionalProducts chosen for an Order
  - Fetch is LAZY because maybe we don't want to access all the OptionalProducts every time
  - Cascading: REFRESH, so that a modification to an OptionalProduct will be updated automatically when the order is refreshed
- OptionalProduct→Order
  - Not necessary, we don't want to access the Orders that use that OptionalProduct

Relationship “OptionalProductReport_OptionalProduct”



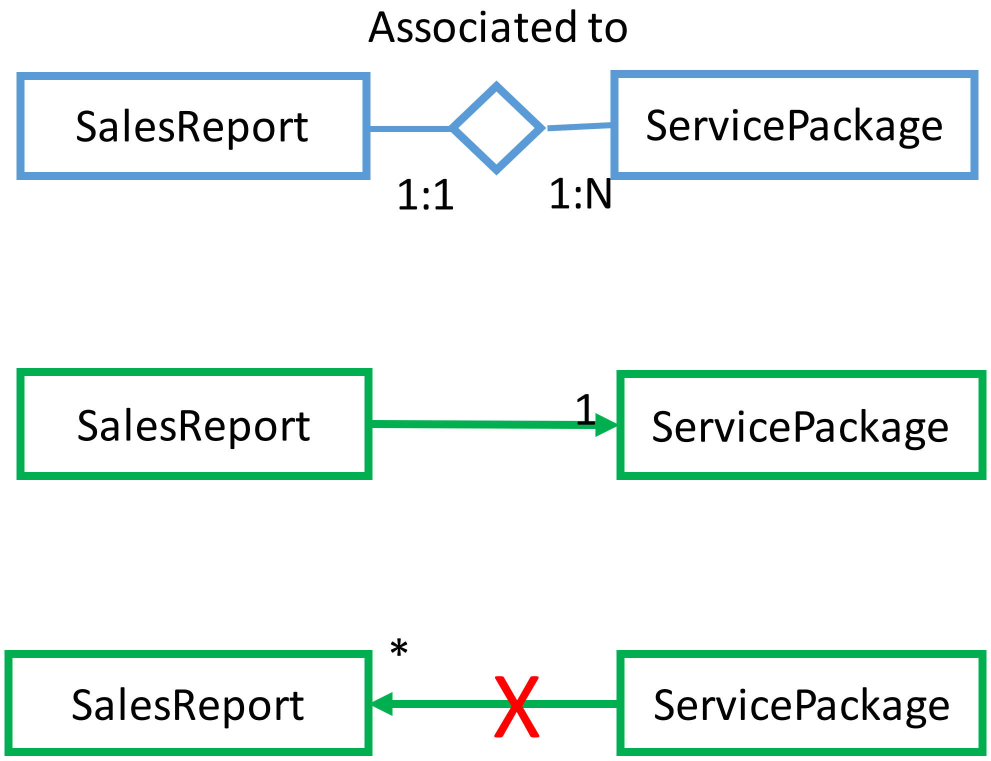
- OptionalProductReport→ OptionalProduct
  - @OneToOne is necessary when we want to know the OptionalProduct related to the report
  - Fetch is EAGER because usually want to access the optional product too when accessing his report
  - Cascading: REFRESH, so that a modification to an OptionalProduct will be updated automatically when the order is refreshed
- OptionalProduct→OptionalProductReport
  - Not necessary, we don't want to access the report from that OptionalProduct

Relationship "Alert_User"



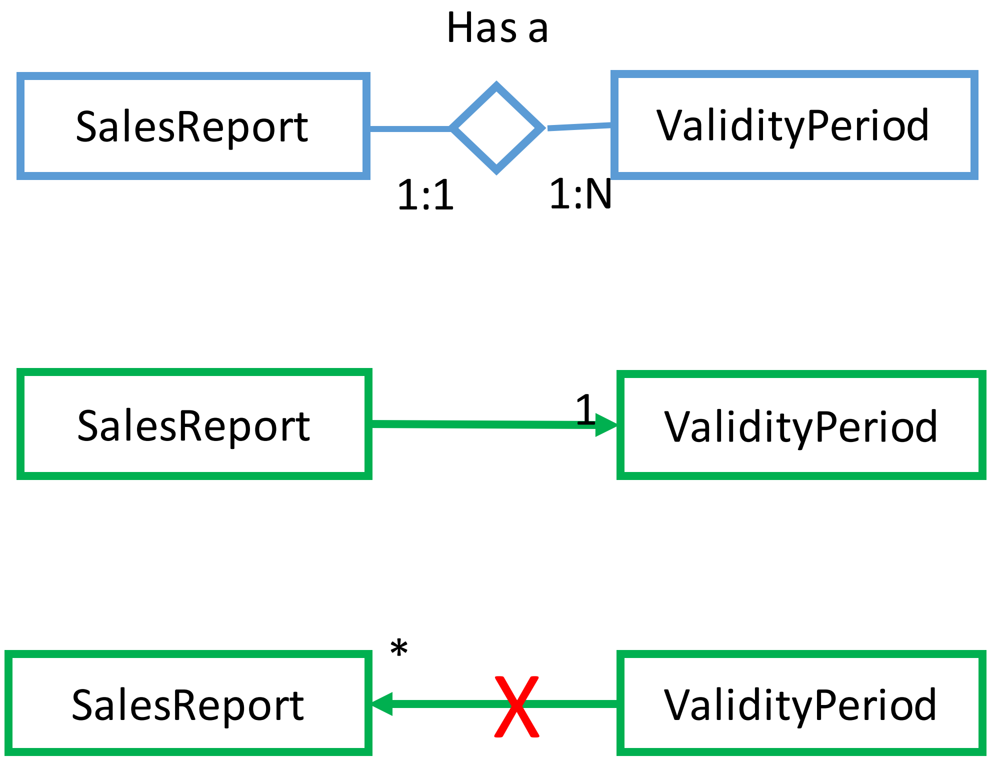
- Alert→ User
  - @ManyToOne is necessary when we want to know the User related to the Alert
  - Fetch is EAGER because usually we want to access the User too when accessing his Alert
  - Cascading: REFRESH, so that a modification to the User will be updated automatically when the Alert is refreshed
- User→Alert
  - We do not need to access the Alerts from a User

Relationship “SalesReport_ServicePackage”



- SalesReport→ServicePackage
  - @ManyToOne is necessary we want to know the ServicePackage associated to the SalesReport
  - Fetch is LAZY because we don't want to access always the Service package (we only print the report value we requested)
  - Cascading: REFRESH, so that a modification to the ServicePackage will be updated automatically when the report is refreshed
- ServicePackage→SalesReport
  - We do not need to access the report from a ServicePackage

Relationship “salesReport_ValidityPeriod”



- SalesReport→ValidityPeriod
  - @ManyToOne is necessary we want to know the ServicePackage associated to the SalesReport
  - Fetch is LAZY because we don't want to access always the ValidityPeriod (we only print the report value we requested)
  - Cascading: REFRESH, so that a modification to the ValidityPeriod will be updated automatically when the report is refreshed
- ValidityPeriod→SalesReport
  - We do not need to access the report from a ValidityPeriod

ORM design motivations

Idea behind cascading rules:

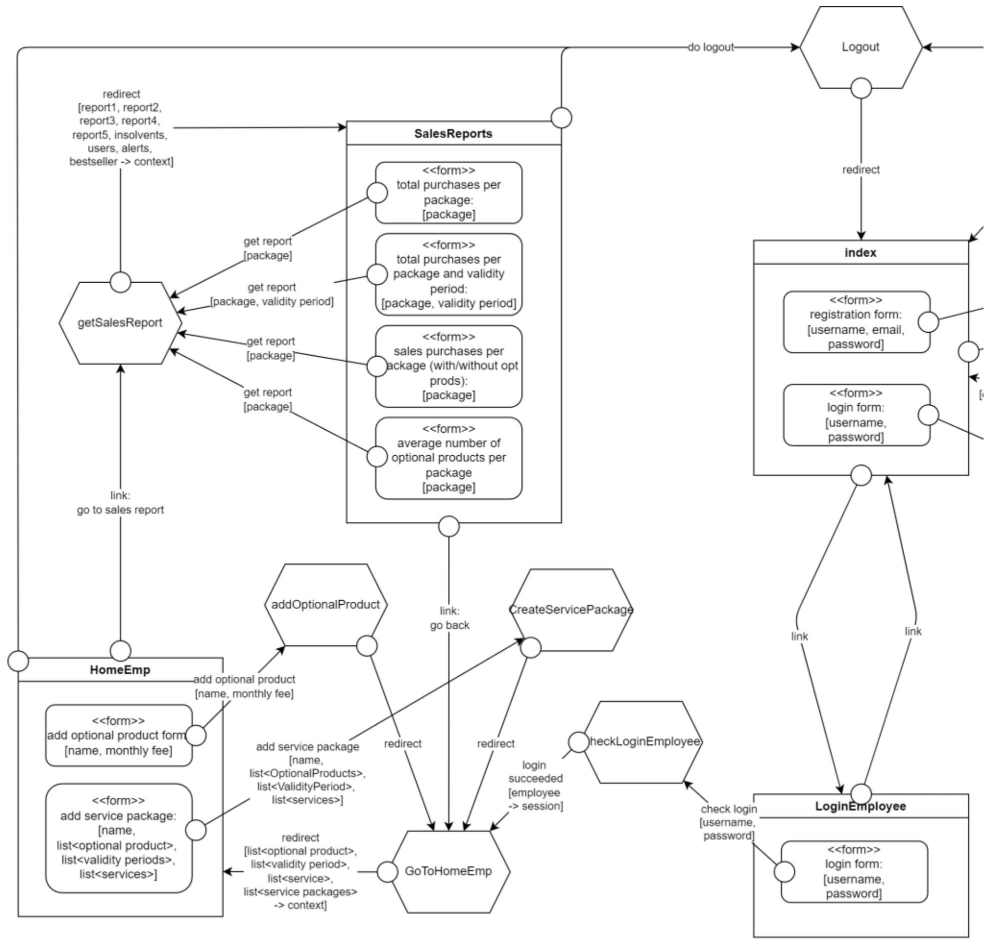
- As a default behaviour we would like to have all the entities B that belong to an entity A to be refreshed automatically when A is refreshed. For this the "CascadeType.REFRESH" is present in all the relations;
- In addition, as a feature, in the employee section also the "CascadeType.PERSIST" is added, in order to implement in the future an automatic persistance of optional products, validity periods, services while creating a service package

Idea behind fetch rules:

- As a default behaviour we preferred to have a LAZY fetching, so that the load of data is decreased and responsivity is increased.
- When we know for sure that the relative entity-relation is accessed we preferred the EAGER fetching

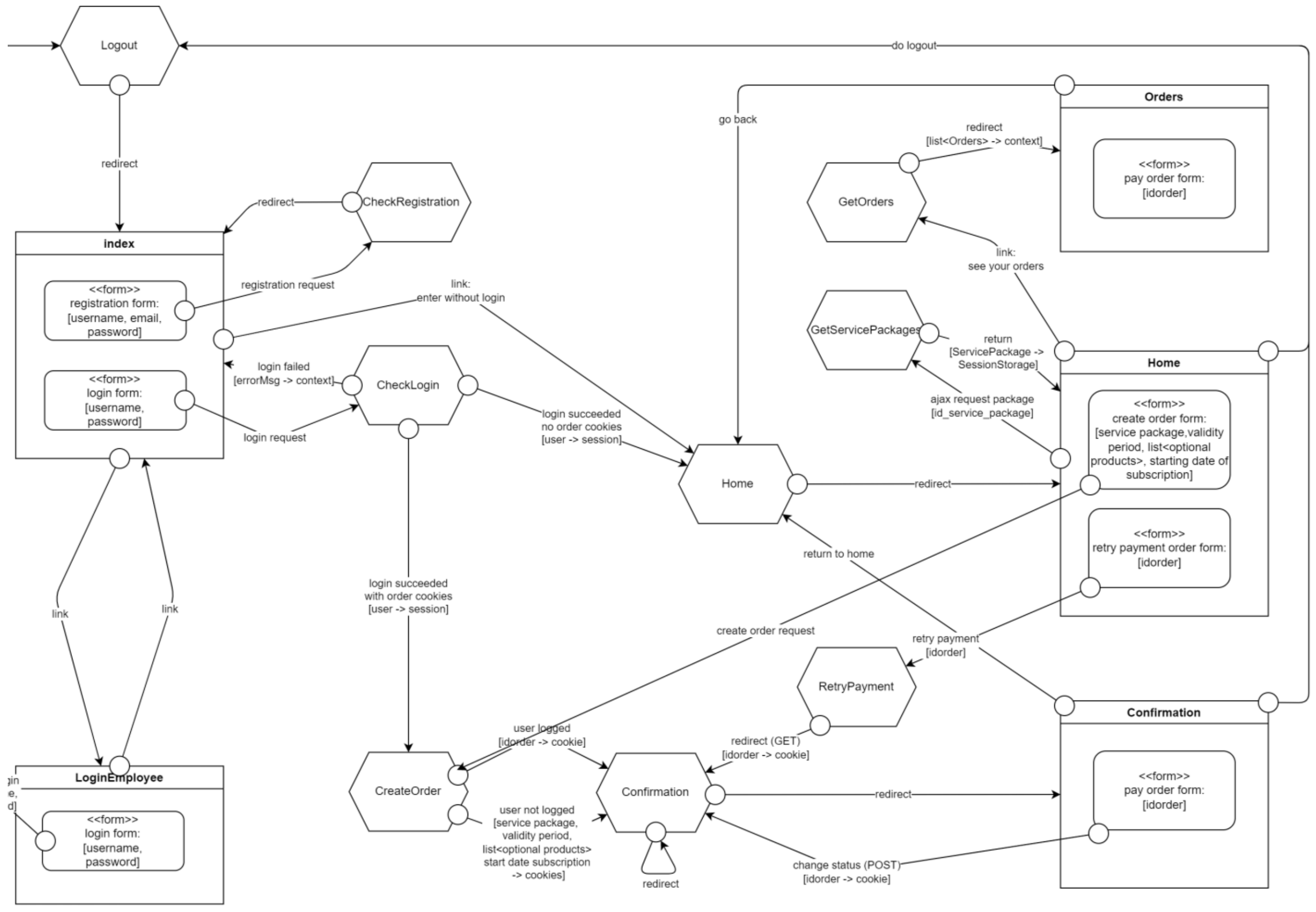
Interaction flow

Interaction Flow – employees application

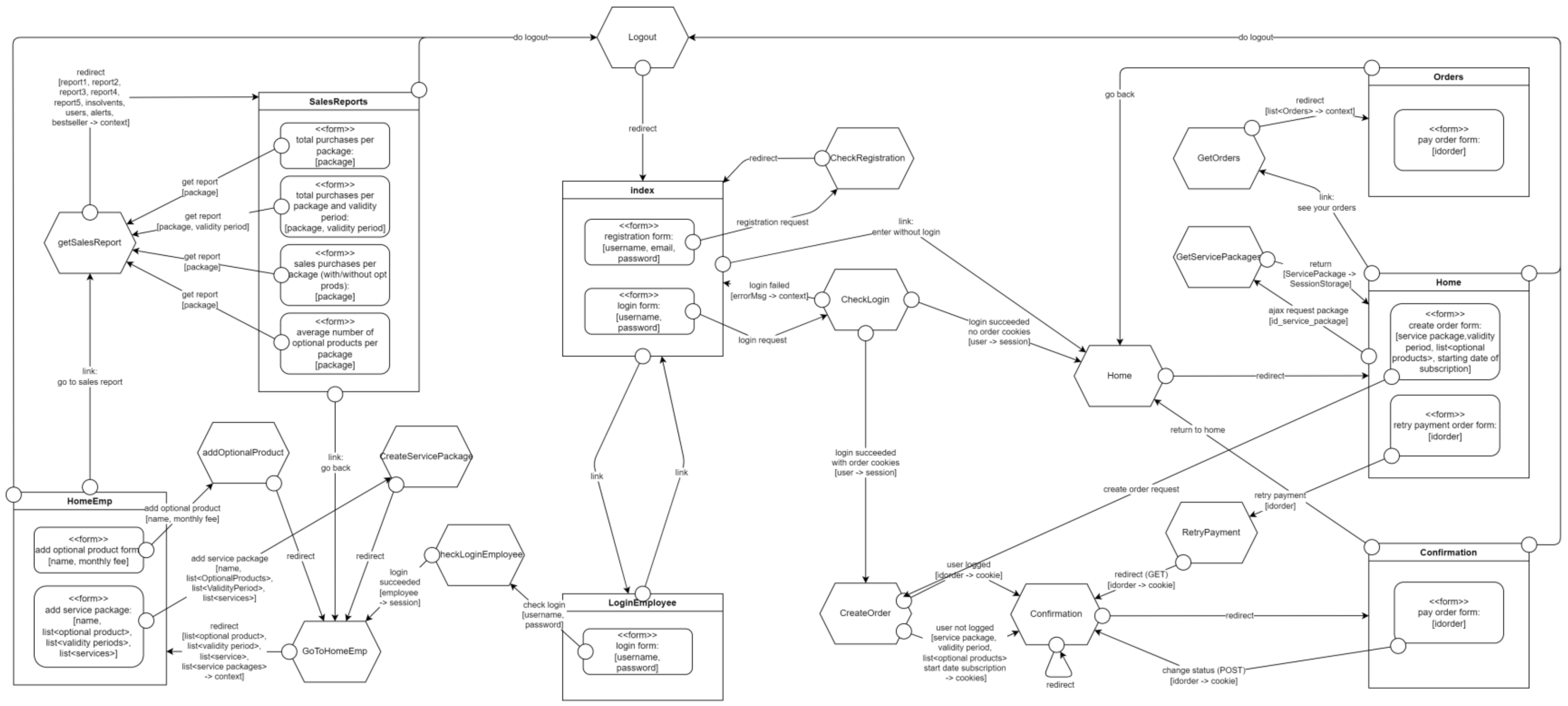


Logout
do logout
redirect
index
<<form>> registration form: [username, email, password]
<<form>> login form: [username, password]
link
link
LoginEmployee
<<form>> login form: [username, password]
check login [username, password]
checkLoginEmployee
login succeeded [employee -> session]
SalesReports
<<form>> total purchases per package: [package]
<<form>> total purchases per package and validity period: [package, validity period]
<<form>> sales purchases per package (with/without opt prods): [package]
<<form>> average number of optional products per package [package]
redirect [report1, report2, report3, report4, report5, insolvents, users, alerts, bestseller -> context]
get report [package]
get report [package, validity period]
get report [package]
get report [package]
getSalesReport
link: go to sales report
link: go back
HomeEmp
<<form>> add optional product form [name, monthly fee]
<<form>> add service package: [name, list<optional product>, list<validity periods>, list<services>]
add optional product [name, monthly fee]
add service package [name, list<OptionalProducts>, list<ValidityPeriod>, list<services>]
addOptionalProduct
CreateServicePackage
redirect
redirect
GoToHomeEmp
redirect [list<optional product>, list<validity period>, list<service>, list<service packages> -> context]

Interaction Flow – Customer application



Interaction Flow - overview



Logout
do logout
do logout
redirect
index
<<form>> registration form: [username, email, password]
<<form>> login form: [username, password]
redirect
CheckRegistration
registration request
link: enter without login
login failed [errorMsg -> context]
CheckLogin
login request
login succeeded no order cookies [user -> session]
login succeeded with order cookies [user -> session]
link
link
LoginEmployee
<<form>> login form: [username, password]
check login [username, password]
heckLoginEmployee
login succeeded [employee -> session]
GoToHomeEmp
redirect [list<optional product>, list<validity period>, list<service>, list<service packages> -> context]
HomeEmp
<<form>> add optional product form [name, monthly fee]
<<form>> add service package: [name, list<optional product>, list<validity periods>, list<services>]
add optional product [name, monthly fee]
addOptionalProduct
add service package [name, list<OptionalProducts>, list<ValidityPeriod>, list<services>]
CreateServicePackage
redirect
redirect
link: go back
link: go to sales report
getSalesReport
redirect [report1, report2, report3, report4, report5, insolvents, users, alerts, bestseller -> context]
SalesReports
<<form>> total purchases per package: [package]
<<form>> total purchases per package and validity period: [package, validity period]
<<form>> sales purchases per ackage (with/without opt prods): [package]
<<form>> average number of optional products per package [package]
get report [package]
get report [package, validity period]
get report [package]
get report [package]
go back
Orders
<<form>> pay order form: [idorder]
redirect [list<Orders> -> context]
GetOrders
link: see your orders
GetServicePackages
return [ServicePackage -> SessionStorage]
ajax request package [id_service_package]
Home
<<form>> create order form: [service package,validity period, list<optional products>, starting date of subscription]
<<form>> retry payment order form: [idorder]
Home
redirect
return to home
create order request
retry payment [idorder]
RetryPayment
redirect (GET) [idorder -> cookie]
CreateOrder
user logged [idorder -> cookie]
user not logged [service package, validity period, list<optional products> start date subscription -> cookies]
Confirmation
redirect
redirect
Confirmation
<<form>> pay order form: [idorder]
change status (POST) [idorder -> cookie]

Components

Components: Client components

Servlets:

- Customers:
  - CheckLogin
  - CheckRegistration
  - Confirmation
  - CreateOrder
  - GetOrders
  - GetServicePackages
  - GoToHomePage
  - Logout
  - RetryPayment
- Employees
  - AddOptionalProduct
  - CheckLoginEmployee
  - CreateServicePackage
  - GetSalesReport
  - GoToHomeEmp

Views:

- Common:
  - index
- Customers:
  - Home
  - Confirmation
  - Orders
- Employees:
  - LoginEmployee
  - HomeEmp
  - SalesReport

Components: back-end entities

- Alert
- Employee
- OptionalProduct
- OptionalProductsReport
- Order
- SalesReport
- SalesReportId
- Service
- ServicePackage
- User
- ValidityPeriod

Components: back-end: EJBs

- AlertService
  - Stateless
  - getAllAlerts()
- EmployeeService
  - Stateless
  - ckeckCredentials(username, psw)
- OptionalProductService
  - Stateless
  - CreateOptionalProduct(name, monthlyFee)
  - getAllOptionalProducts()
  - getOptionalProductById(id)
  - getBestSellers()
- OrderService
  - Stateless
  - CreateOrder(status, user, servicePackage, validityPeriod, optionalProducts, dateSubscription)
  - GetSuspended()
  - GetSuspendedByUser(id_user)
  - GetOrderById(id)
  - ChangeStatus(status)
  - GetOrdersPerUser(id_user)
- SalesReportService
  - Stateless
  - GetTotalPurchasePerPackage(id)
  - GetTotalPurchasePerPackageAndVp(idP, idVP)
  - GetValueNoOptionalProducts(id)
  - GetValueWithOptionalProducts(id)
  - GetAvgOptProducts(id)
- ServicePackageService
  - Stateless
  - CreateServicePackage(name, valPerIds, opProdIds, servIds)
  - GetAllServicePackage()
  - GetServicePackageById(id)
- ServiceService
  - Stateless
  - GetAllServices()
- UserService
  - Stateless
  - FindUserByUsername(username)
  - CretaUser(string, email, password)
  - CheckCredentials(usrn, psw)
  - GetInsolvents()
- ValidityPeriodService
  - Stateless
  - GetAllValidityPeriod()
  - GetValidityPeriodById()

Motivations of the components design

We isolated the customer and employee applications by means of filters, so that the two applications are totally disjunct: a customer won't be able to access employees pages or issue post requests in order to create new service packages, new optional products or in order to retrieve report data.

The exchange of data among servlets is done by means of the session (only stores the user/employee logged) and cookies (for non-sensible data, like the order to see in the confirmation page, retrieved only if the order belongs to the user logged, and the order info when a user isn't logged and wants to create an order).

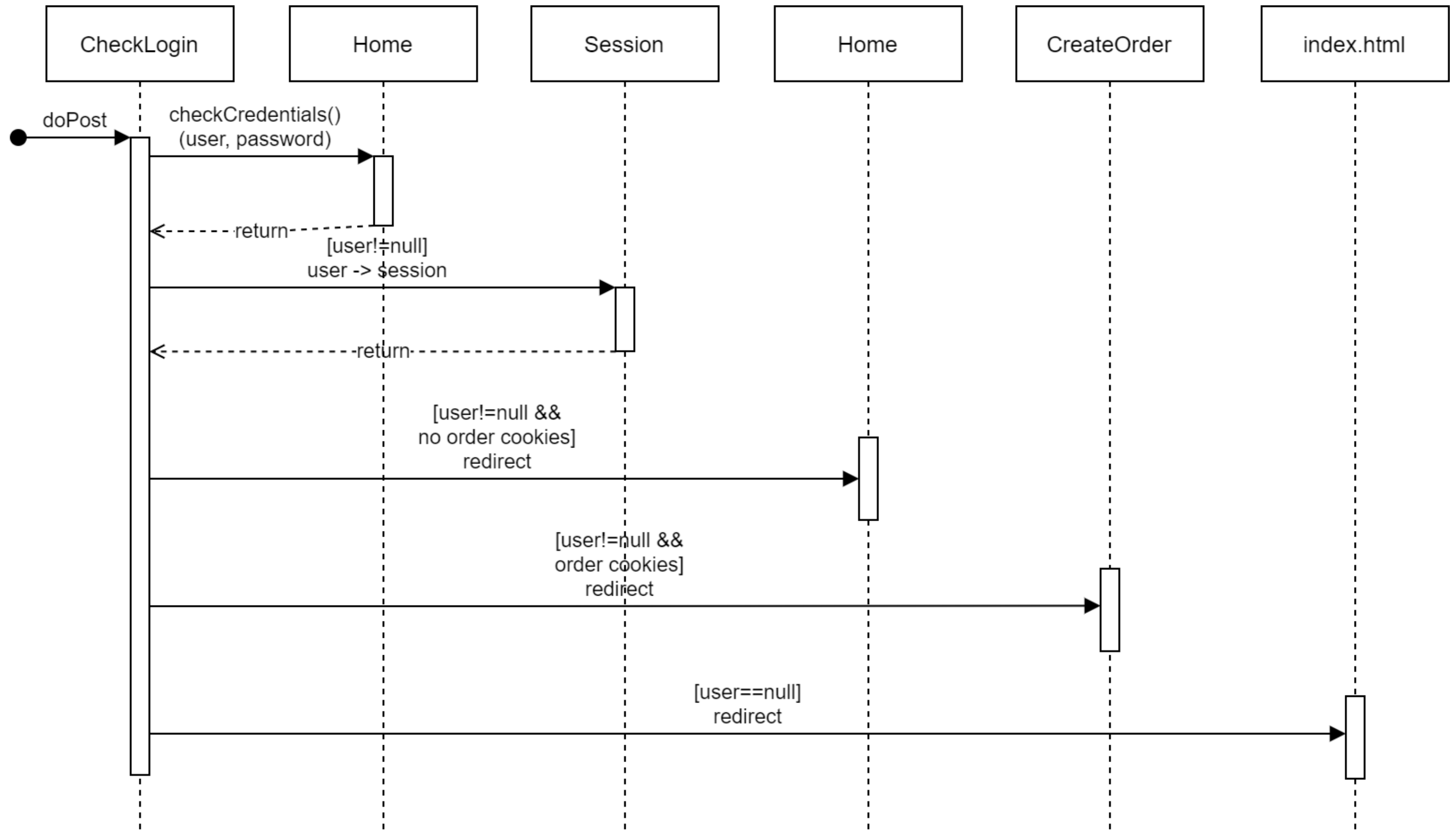
Sequence diagrams

UML sequence diagrams

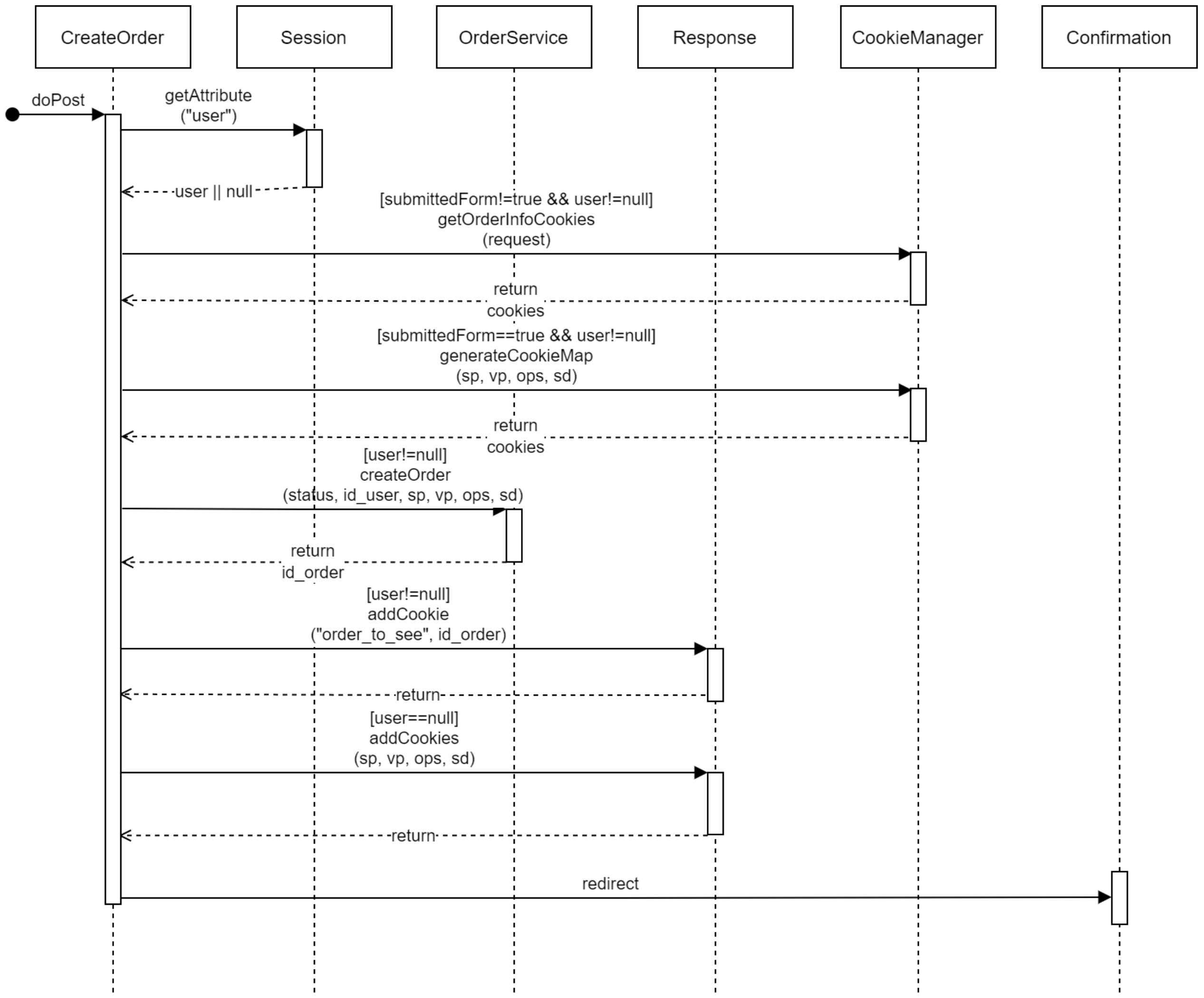
In the following slides we are providing some sequence diagrams of the most important mechanisms of the application:

- Logging of an user (the process of logging for the employees is similar)
- Creation of an order both if the user is or isn't logged
- Retrieving of a report in the employee application

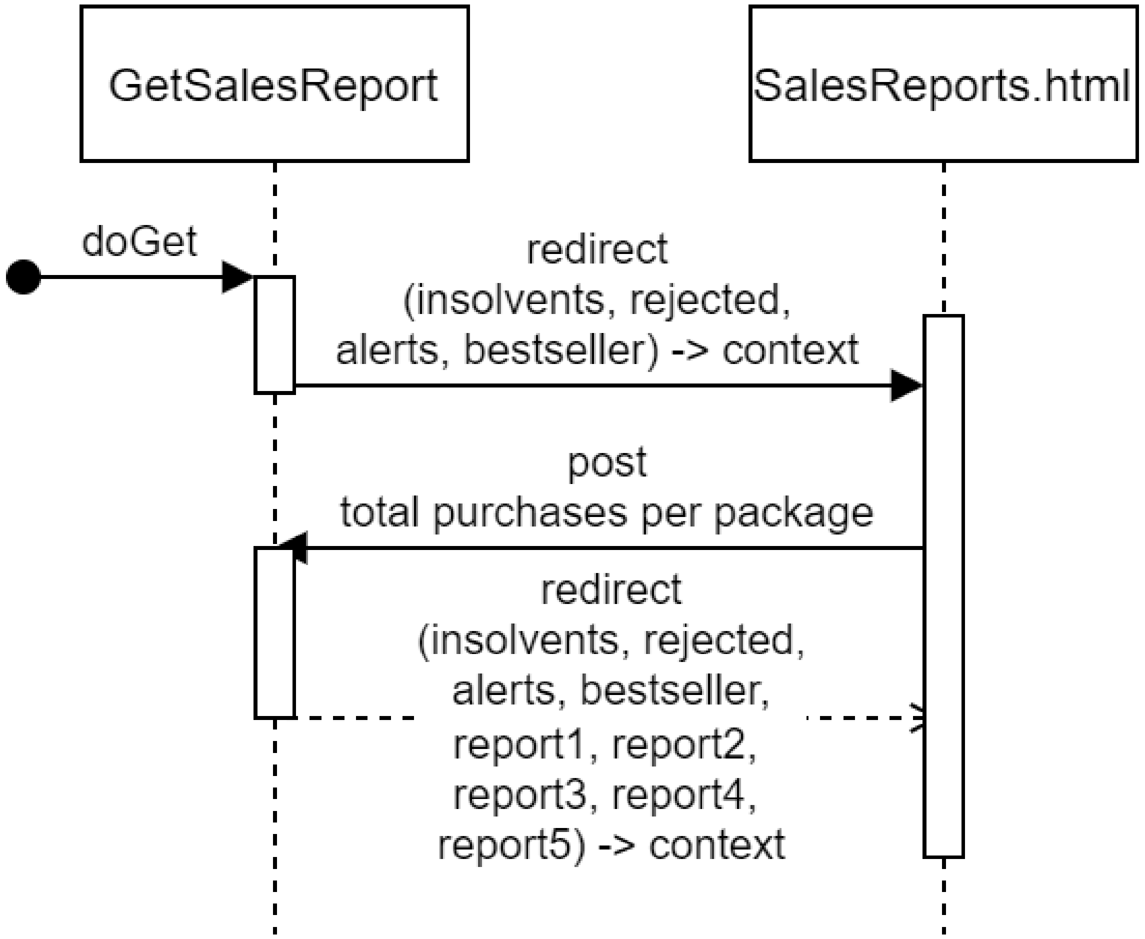
Sequence diagram for the logging of a user



Sequence diagram for the creation of an order


CreateOrder
Session
OrderService
Response
CookieManager
Confirmation
doPost
getAttribute ("user")
user || null
[submittedForm!=true && user!=null] getOrderInfoCookies (request)
return cookies
[submittedForm==true && user!=null] generateCookieMap (sp, vp, ops, sd)
return cookies
[user!=null] createOrder (status, id_user, sp, vp, ops, sd)
return id_order
[user!=null] addCookie ("order_to_see", id_order)
return
[user==null] addCookies (sp, vp, ops, sd)
return
redirect

Sequence diagram for request of a report



Front-end considerations

In the front-end developement of the application we considered the factors of usability and security.

- Usability: we used the well-known css framework bootstrap in order to have a more user-friendly front-end.
- Security: We added a superficial layer of protection for the inputs of the user, setting attributes to user inputs and using dynamic generated drop-down menù with the correct limited set of options the user should use. Most of the data is embedded in the back-end with the templating framework "thymeleaf", so that very little data is disclosed to the user (only the necessary data).

Project tools and repository

For the project we used:

- Intellij IDE in order to implement the application code in a multiplatform envirnonment (two different operating systems, iOS and Windows)
- mySQL workbench
- Tomee
- Github as versioning system A1iceCariboni/ProjDB2 (github.com)