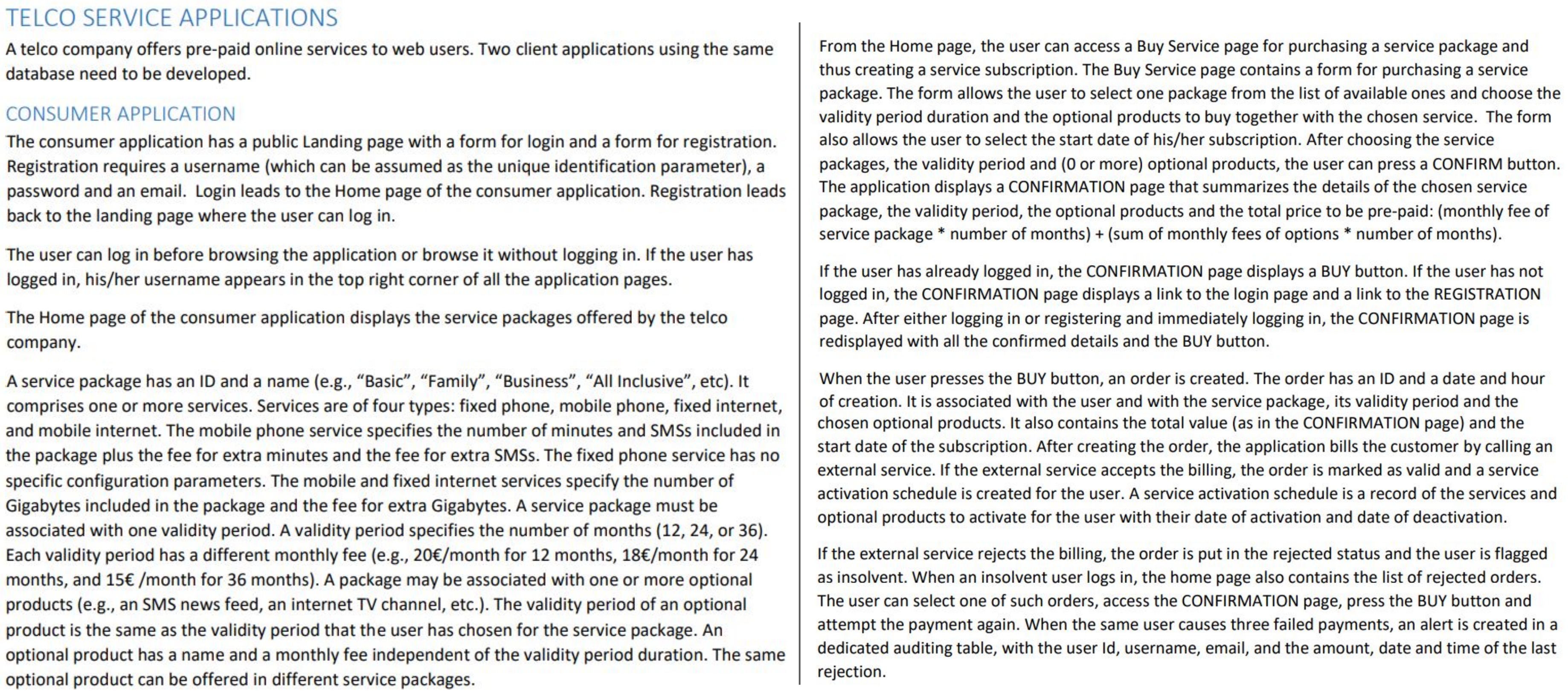
DATA BASES 2

Optional Project: Telco Service Application

Andrea Carotti, Matteo Crugnola

Index

Specification



TELCO SERVICE APPLICATIONS

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

CONSUMER APPLICATION

The consumer application has a public Landing page with a form for login and a form for registration. Registration requires a username (which can be assumed as the unique identification parameter), a password and an email. Login leads to the Home page of the consumer application. Registration leads back to the landing page where the user can log in.

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

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

A service package has an ID and a name (e.g., “Basic”, “Family”, “Business”, “All Inclusive”, etc). It comprises one or more services. Services are of four types: fixed phone, mobile phone, fixed internet, and mobile internet. The mobile phone service specifies the number of minutes and SMSs included in the package plus the fee for extra minutes and the fee for extra SMSs. The fixed phone service has no specific configuration parameters. The mobile and fixed internet services specify the number of Gigabytes included in the package and the fee for extra Gigabytes. A service package must be associated with one validity period. A validity period specifies the number of months (12, 24, or 36). Each validity period has a different monthly fee (e.g., 20€/month for 12 months, 18€/month for 24 months, and 15€ /month for 36 months). A package may be associated with one or more optional products (e.g., an SMS news feed, an internet TV channel, etc.). The validity period of an optional product is the same as the validity period that the user has chosen for the service package. An optional product has a name and a monthly fee independent of the validity period duration. The same optional product can be offered in different service packages.

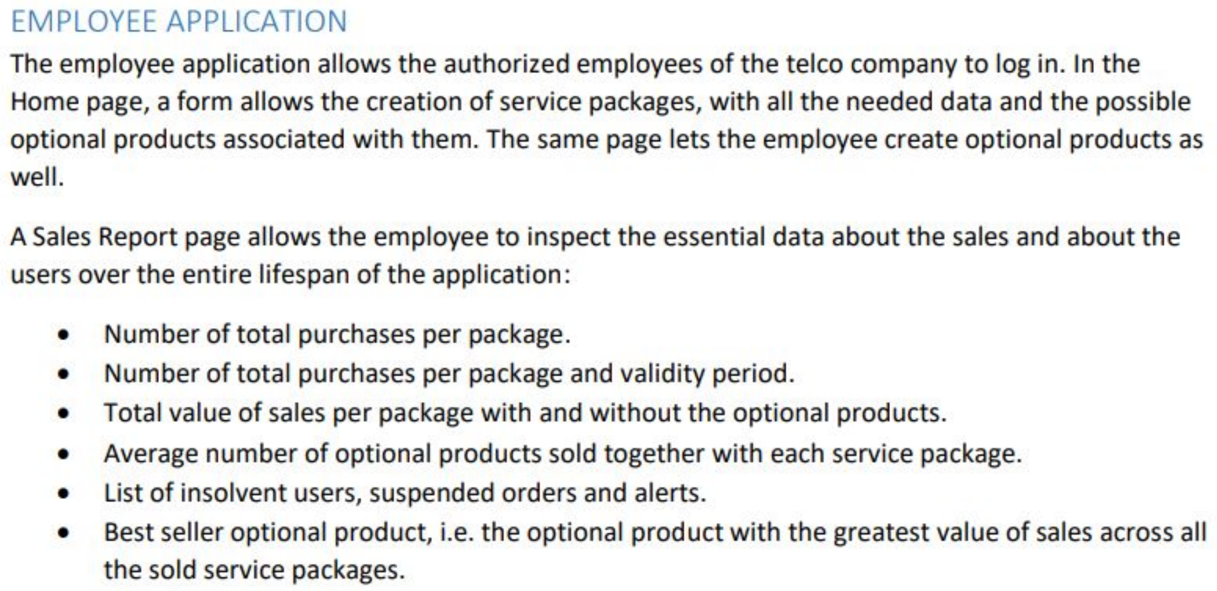
From the Home page, the user can access a Buy Service page for purchasing a service package and thus creating a service subscription. The Buy Service page contains a form for purchasing a service package. The form allows the user to select one package from the list of available ones and choose the validity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

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

When the user presses the BUY button, an order is created. The order has an ID and a date and hour of creation. It is associated with the user and with the service package, its validity period and the chosen optional products. It also contains the total value (as in the CONFIRMATION page) and the start date of the subscription. After creating the order, the application bills the customer by calling an external service. If the external service accepts the billing, the order is marked as valid and a service activation schedule is created for the user. A service activation schedule is a record of the services and optional products to activate for the user with their date of activation and date of deactivation.

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

Specification



EMPLOYEE APPLICATION

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

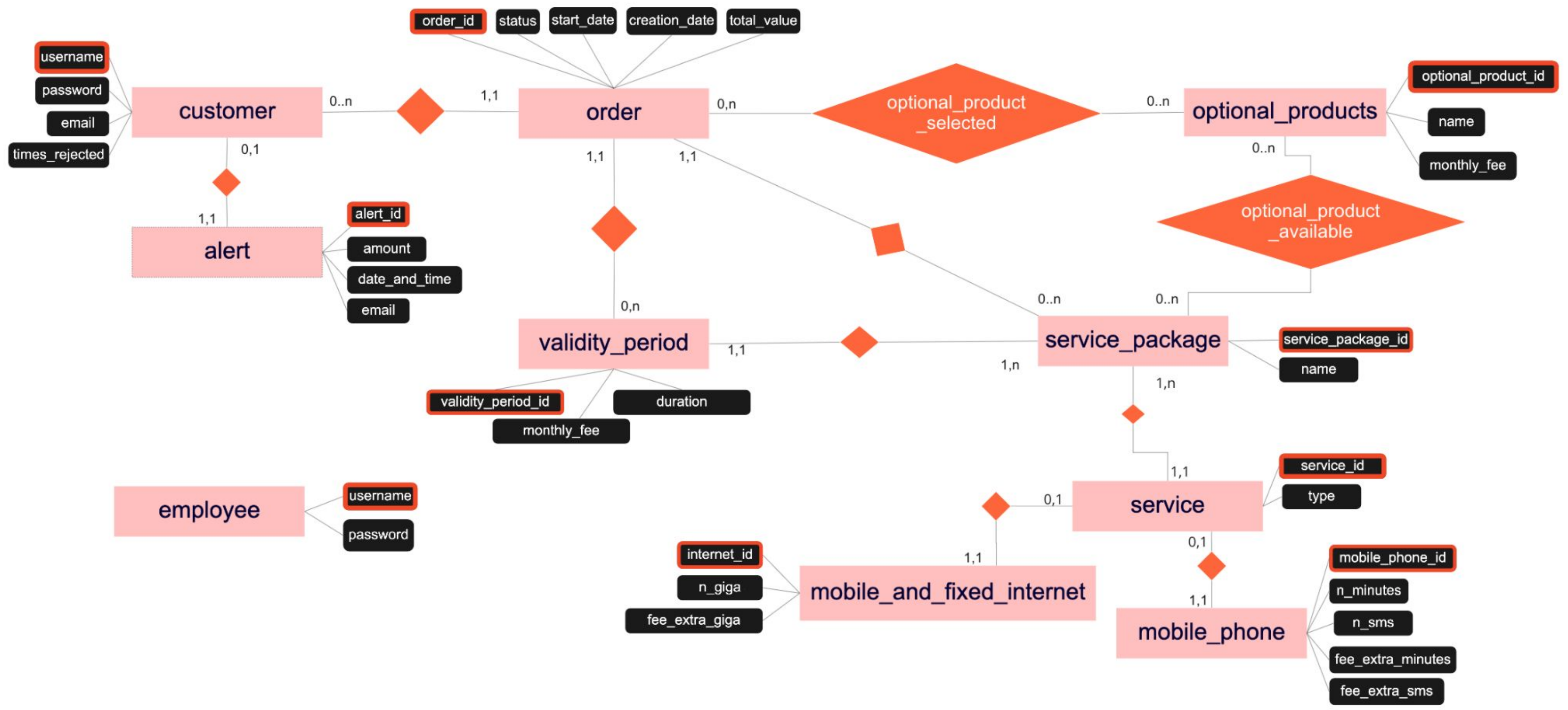
A Sales Report page allows the employee to inspect the essential data about the sales and about the users over the entire lifespan of the application:

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

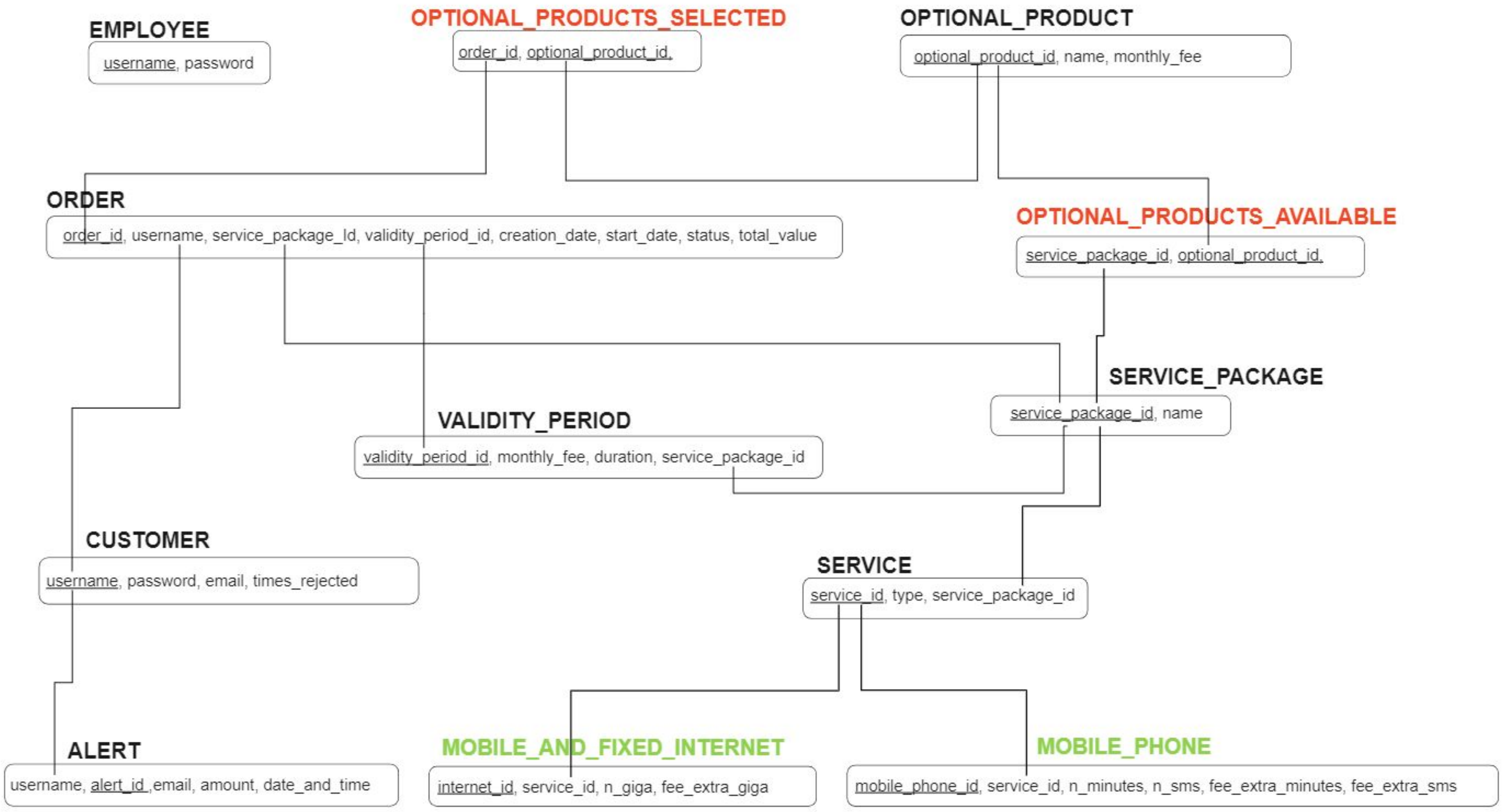
Specification interpretation

- We created a single html page for both login and registration purpose.
- The service activation schedule in our database contains references only to: order_id, service_package_id and the customer username. We didn’t include also the activation schedule for all services and optional products selected in the orders because that would take a lot of memory for something that could be computed with a simple join between the activation schedule table and the services table or the optional products selected table.
- In our service packages there can be more than one service of the same type (e.g. a family package with 4 mobile phones (one for each member) and 4 mobile internet, one fixed internet, one fixed phone).

ENTITY RELATIONSHIP



LOGICAL SCHEMA


EMPLOYEE
username, password
OPTIONAL_PRODUCTS_SELECTED
order_id, optional_product_id,
OPTIONAL_PRODUCT
optional_product_id, name, monthly_fee
ORDER
order_id, username, service_package_Id, validity_period_id, creation_date, start_date, status, total_value
OPTIONAL_PRODUCTS_AVAILABLE
service_package_id, optional_product_id,
SERVICE_PACKAGE
service_package_id, name
VALIDITY_PERIOD
validity_period_id, monthly_fee, duration, service_package_id
CUSTOMER
username, password, email, times_rejected
SERVICE
service_id, type, service_package_id
ALERT
username, alert_id ,email, amount, date_and_time
MOBILE_AND_FIXED_INTERNET
internet_id, service_id, n_giga, fee_extra_giga
MOBILE_PHONE
mobile_phone_id, service_id, n_minutes, n_sms, fee_extra_minutes, fee_extra_sms

Aggregate data tables logical schema

AA_ACTIVATION_SCHEDULE

username, service_package_id, activation_date, deactivation_date, order_id

AA_TOTAL_PURCHASES

service_package_id, number_of_purchases

AA_TOTAL_PURCHASES_PER_VAL

service_package_id, validity_period_id, number_of_purchases

AA_TOTAL_REVENUE_WITH_OPTIONALS

service_package_id, revenue_with_optionals, revenue_without_optionals

AA_AVERAGE_OPTIONALS

service_package_id, average_number_of_optionals, number_of_purchases

AA_INSOLVENT_USERS

username

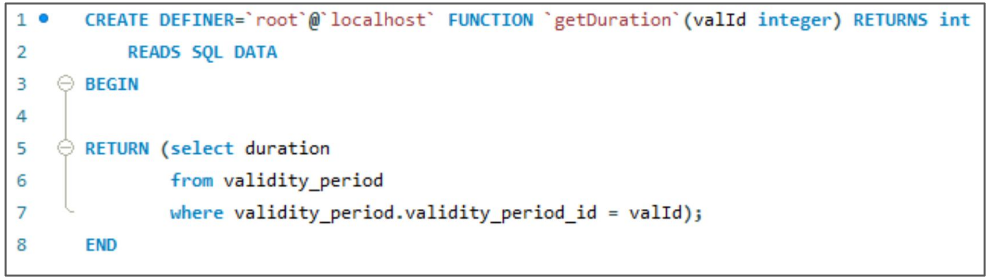
AA_SUSPENDED_ORDERS

order_id

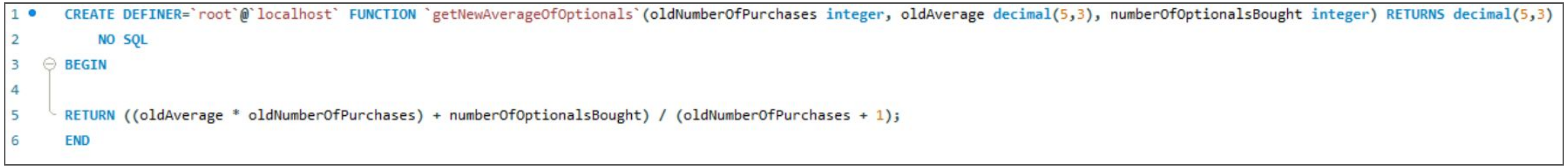
AA_OPTIONALS_TOTAL_REVENUE

optional_product_id, total_revenue

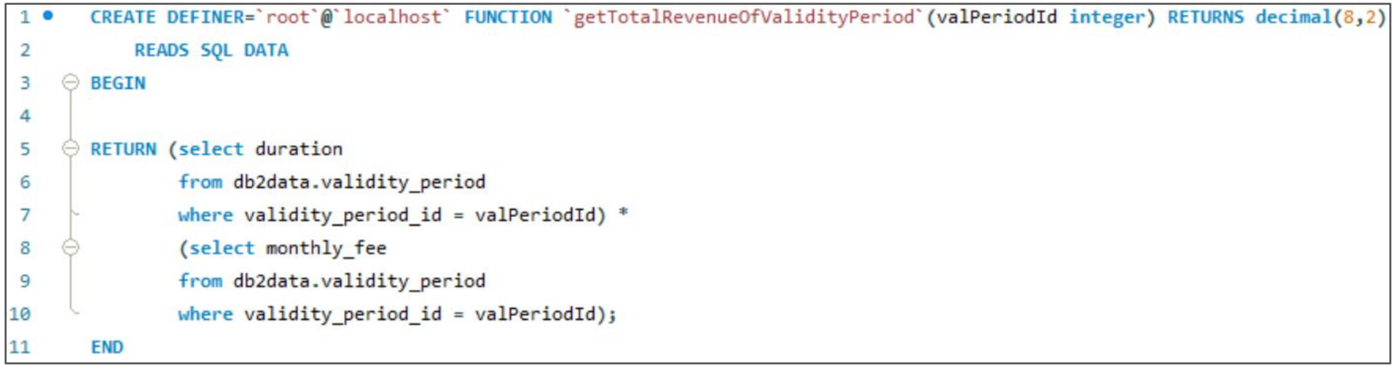
Custom SQL functions to simplify trigger code



```sql
CREATE DEFINER=`root`@`localhost` FUNCTION `getDuration`(valId integer) RETURNS int
    READS SQL DATA
BEGIN

RETURN (select duration
        from validity_period
        where validity_period.validity_period_id = valId);
END
```

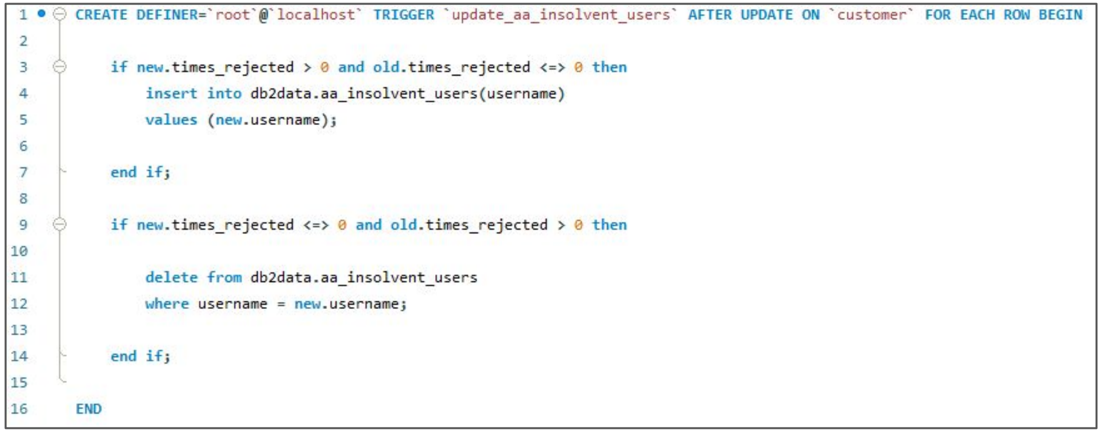


```sql
CREATE DEFINER=`root`@`localhost` FUNCTION `getNewAverageOfOptionals`(oldNumberOfPurchases integer, oldAverage decimal(5,3), numberOfOptionalsBought integer) RETURNS decimal(5,3)
    NO SQL
BEGIN

RETURN ((oldAverage * oldNumberOfPurchases) + numberOfOptionalsBought) / (oldNumberOfPurchases + 1);
END
```



```sql
CREATE DEFINER=`root`@`localhost` FUNCTION `getTotalRevenueOfValidityPeriod`(valPeriodId integer) RETURNS decimal(8,2)
    READS SQL DATA
BEGIN

RETURN (select duration
        from db2data.validity_period
        where validity_period_id = valPeriodId) *
        (select monthly_fee
        from db2data.validity_period
        where validity_period_id = valPeriodId);
END
```

Triggers on CUSTOMER

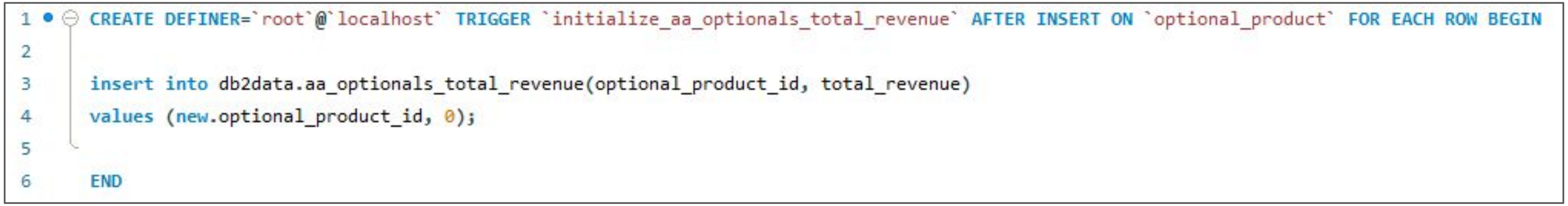


```sql
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_insolvent_users` AFTER UPDATE ON `customer` FOR EACH ROW BEGIN

    if new.times_rejected > 0 and old.times_rejected <=> 0 then
        insert into db2data.aa_insolvent_users(username)
        values (new.username);

    end if;

    if new.times_rejected <=> 0 and old.times_rejected > 0 then

        delete from db2data.aa_insolvent_users
        where username = new.username;

    end if;

END
```

This trigger keeps updated the table of insolvent users after the update on the ‘times_rejected’ attribute of customer.

We decided to reset the ‘times_rejected’ counter and remove the insolvent status to users who paid all their remaining orders.

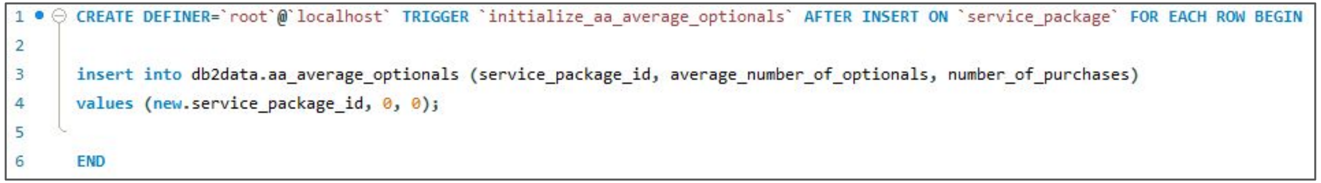
Triggers on OPTIONAL_PRODUCT



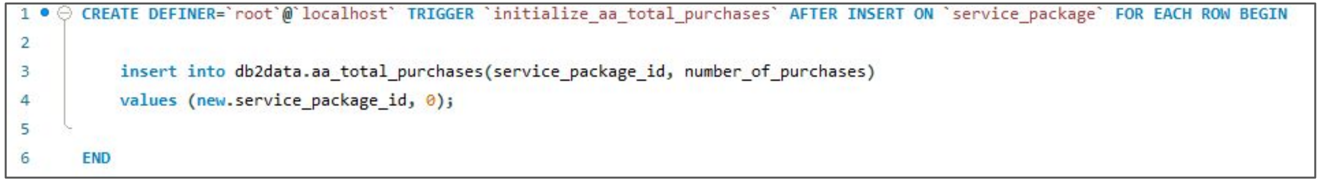
```
CREATE DEFINER=`root`@`localhost` TRIGGER `initialize_aa_optionals_total_revenue` AFTER INSERT ON `optional_product` FOR EACH ROW BEGIN

insert into db2data.aa_optionals_total_revenue(optional_product_id, total_revenue)
values (new.optional_product_id, 0);

END
```

This trigger initializes the table of total revenue per optional product with initial value of 0 for the new optional product.

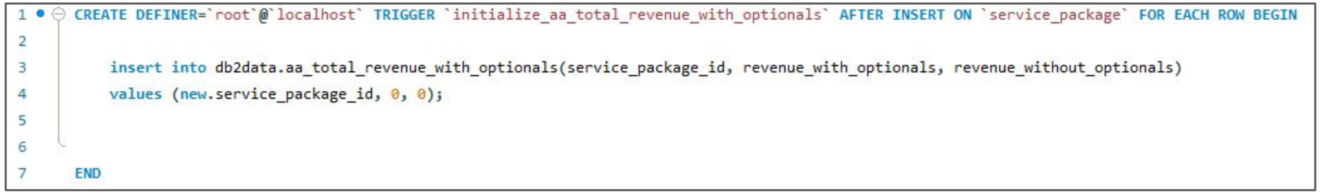
Triggers on SERVICE_PACKAGE



```
CREATE DEFINER=`root`@`localhost` TRIGGER `initialize_aa_average_optionals` AFTER INSERT ON `service_package` FOR EACH ROW BEGIN

insert into db2data.aa_average_optionals (service_package_id, average_number_of_optionals, number_of_purchases)
values (new.service_package_id, 0, 0);

END
```



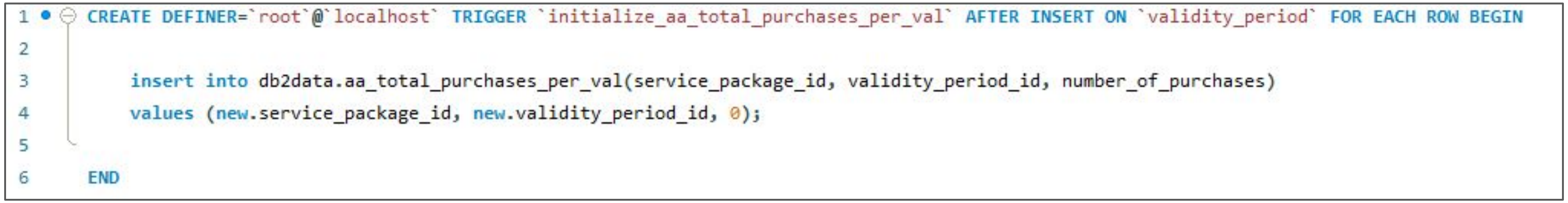
```
CREATE DEFINER=`root`@`localhost` TRIGGER `initialize_aa_total_purchases` AFTER INSERT ON `service_package` FOR EACH ROW BEGIN

    insert into db2data.aa_total_purchases(service_package_id, number_of_purchases)
    values (new.service_package_id, 0);

END
```



```
CREATE DEFINER=`root`@`localhost` TRIGGER `initialize_aa_total_revenue_with_optionals` AFTER INSERT ON `service_package` FOR EACH ROW BEGIN

    insert into db2data.aa_total_revenue_with_optionals(service_package_id, revenue_with_optionals, revenue_without_optionals)
    values (new.service_package_id, 0, 0);


END
```

These triggers are used to initialize the data of the aggregated tables, referring to service packages, once a new service package is created in the database.

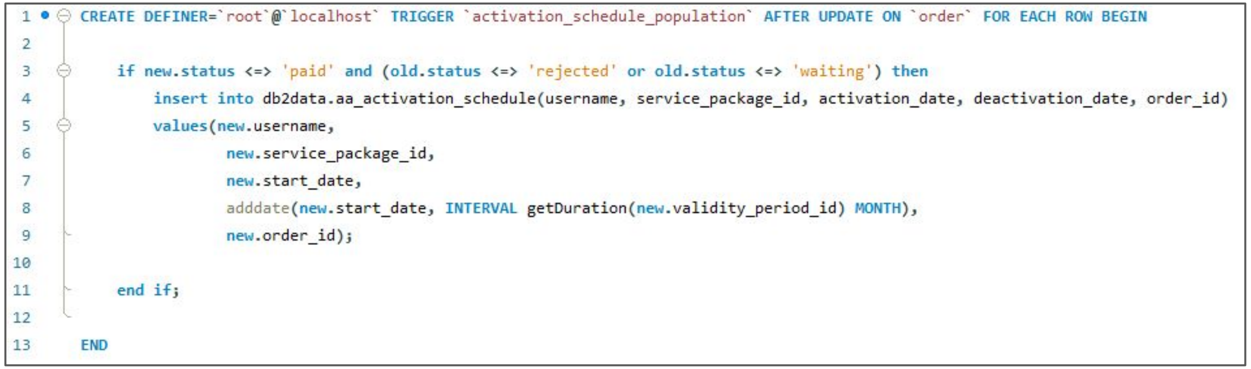
Triggers on VALIDITY_PERIOD



```
CREATE DEFINER=`root`@`localhost` TRIGGER `initialize_aa_total_purchases_per_val` AFTER INSERT ON `validity_period` FOR EACH ROW BEGIN

    insert into db2data.aa_total_purchases_per_val(service_package_id, validity_period_id, number_of_purchases)
    values (new.service_package_id, new.validity_period_id, 0);

END
```

This trigger initializes the value of the aggregated table that keeps track of the number of purchases per validity period.

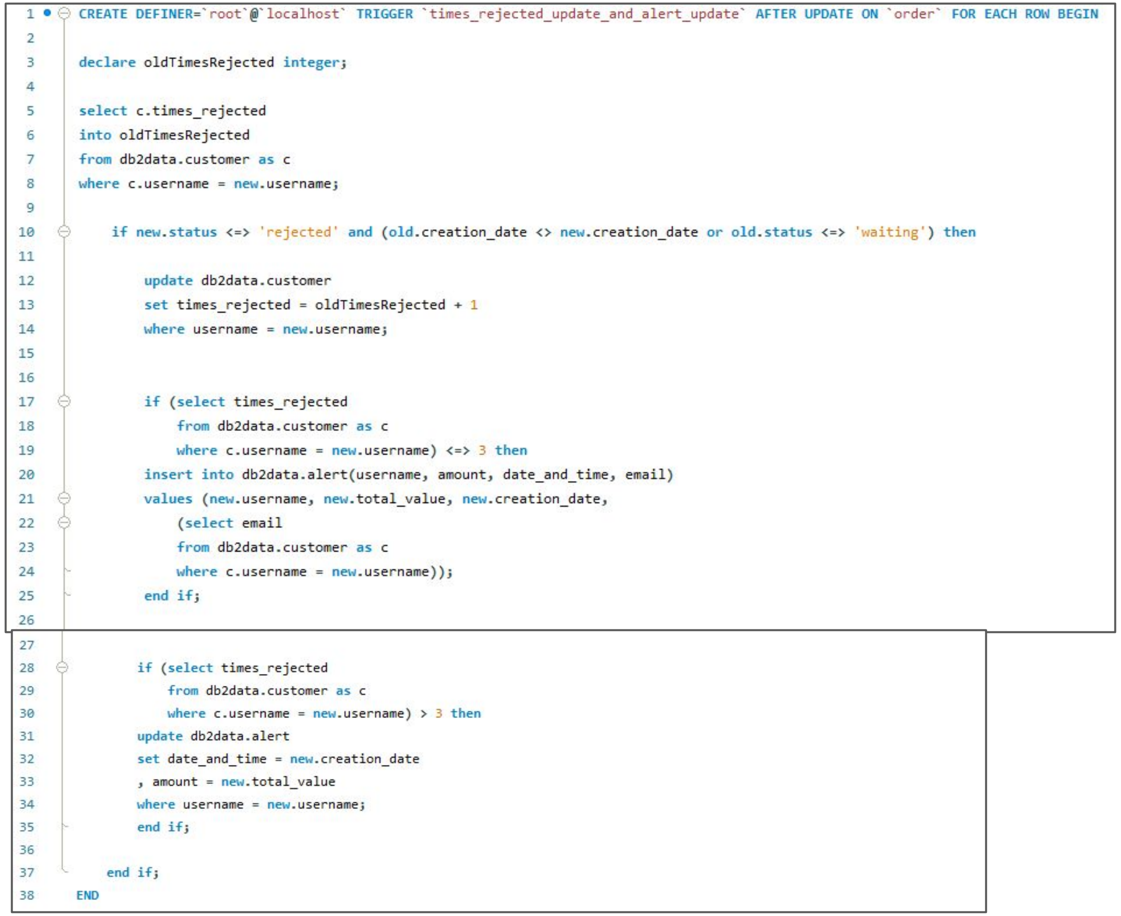
Triggers on ORDER 1/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `activation_schedule_population` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

    if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then
        insert into db2data.aa_activation_schedule(username, service_package_id, activation_date, deactivation_date, order_id)
        values(new.username,
                new.service_package_id,
                new.start_date,
                adddate(new.start_date, INTERVAL getDuration(new.validity_period_id) MONTH),
                new.order_id);

    end if;

END
```

This trigger populates the activation schedule table, it uses the SQL function adddate() to compute the deactivation date.

Triggers on ORDER 2/10

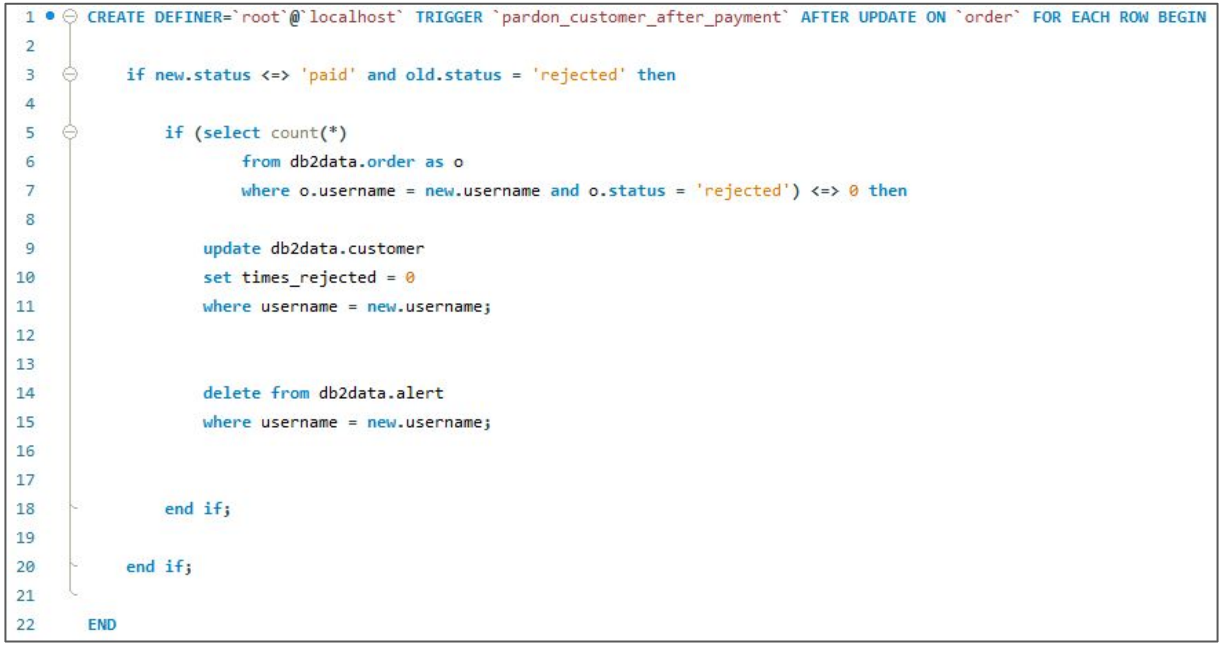


```
CREATE DEFINER=`root`@`localhost` TRIGGER `times_rejected_update_and_alert_update` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

declare oldTimesRejected integer;

select c.times_rejected
into oldTimesRejected
from db2data.customer as c
where c.username = new.username;

    if new.status <=> 'rejected' and (old.creation_date <> new.creation_date or old.status <=> 'waiting') then

        update db2data.customer
        set times_rejected = oldTimesRejected + 1
        where username = new.username;


        if (select times_rejected
            from db2data.customer as c
            where c.username = new.username) <=> 3 then
        insert into db2data.alert(username, amount, date_and_time, email)
        values (new.username, new.total_value, new.creation_date,
            (select email
            from db2data.customer as c
            where c.username = new.username));
        end if;


        if (select times_rejected
            from db2data.customer as c
            where c.username = new.username) > 3 then
        update db2data.alert
        set date_and_time = new.creation_date
        , amount = new.total_value
        where username = new.username;
        end if;

    end if;
END
```

This trigger updates the 'times_rejected' column of customer anytime that customer causes a failed payment.

Then based on that value it may create the alert or it may update an already existing alert.

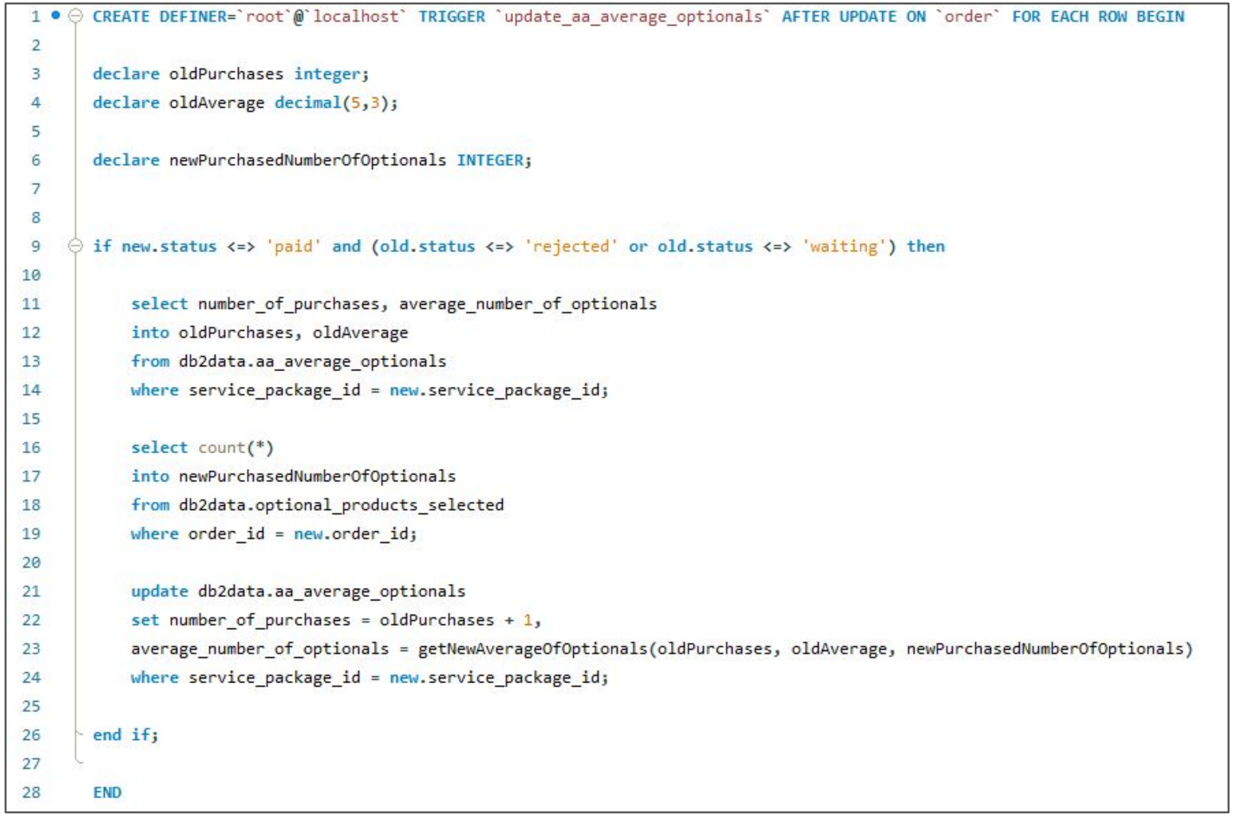
Triggers on ORDER 3/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `pardon_customer_after_payment` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

    if new.status <=> 'paid' and old.status = 'rejected' then

        if (select count(*)
                from db2data.order as o
                where o.username = new.username and o.status = 'rejected') <=> 0 then

            update db2data.customer
            set times_rejected = 0
            where username = new.username;


            delete from db2data.alert
            where username = new.username;


        end if;

    end if;

END
```

This trigger is used in our logic to absolve a customer that has no more suspended orders, we decided to reset its ‘times_rejected’ counter to 0 and to remove its alert when this happens.

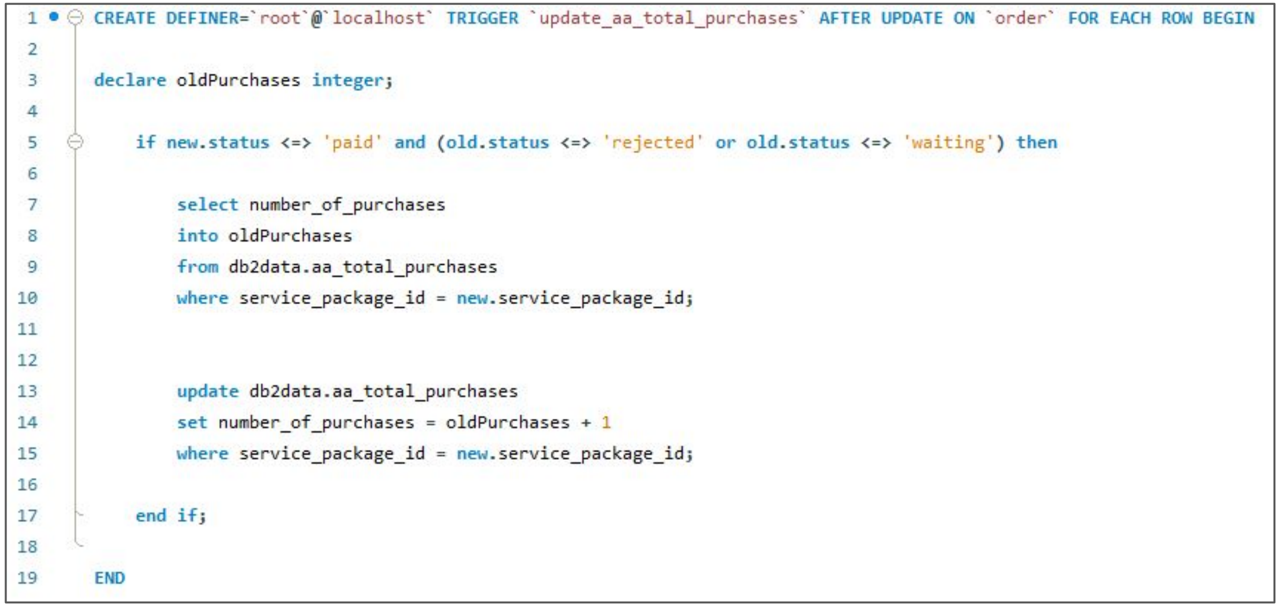
Triggers on ORDER 4/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_average_optionals` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

declare oldPurchases integer;
declare oldAverage decimal(5,3);

declare newPurchasedNumberOfOptionals INTEGER;


if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then

    select number_of_purchases, average_number_of_optionals
    into oldPurchases, oldAverage
    from db2data.aa_average_optionals
    where service_package_id = new.service_package_id;

    select count(*)
    into newPurchasedNumberOfOptionals
    from db2data.optional_products_selected
    where order_id = new.order_id;

    update db2data.aa_average_optionals
    set number_of_purchases = oldPurchases + 1,
    average_number_of_optionals = getNewAverageOfOptionals(oldPurchases, oldAverage, newPurchasedNumberOfOptionals)
    where service_package_id = new.service_package_id;

end if;

END
```

This table updates the aggregated table of number of average optional products bought for each service package anytime there is a correct payment of an order.

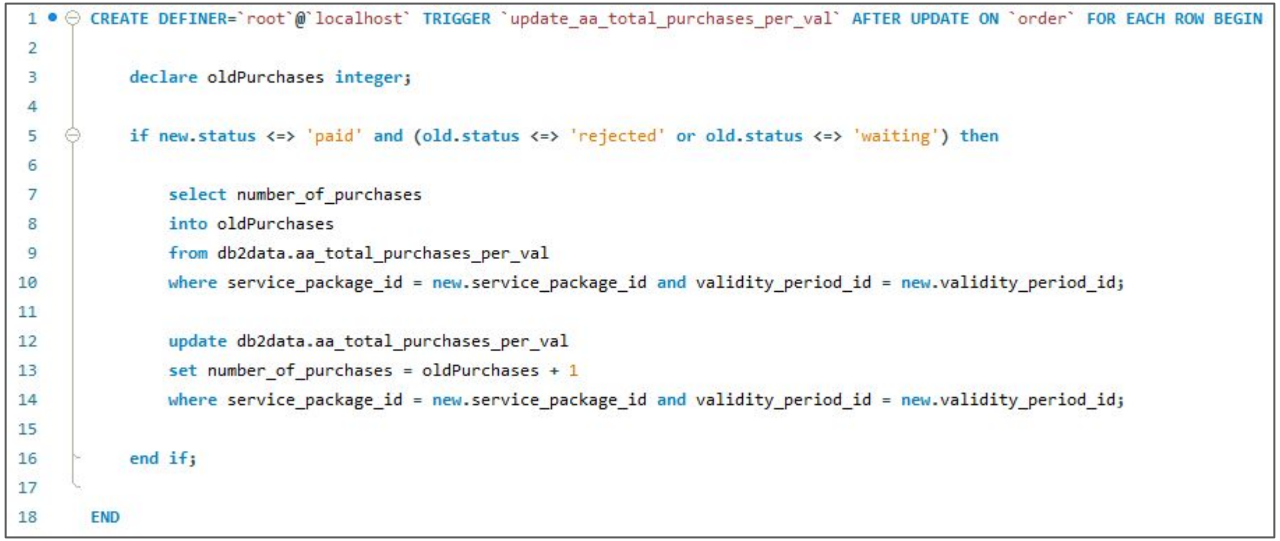
Triggers on ORDER 5/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_total_purchases` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

declare oldPurchases integer;

    if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then

        select number_of_purchases
        into oldPurchases
        from db2data.aa_total_purchases
        where service_package_id = new.service_package_id;


        update db2data.aa_total_purchases
        set number_of_purchases = oldPurchases + 1
        where service_package_id = new.service_package_id;

    end if;

END
```

This trigger updates the table that keeps track of the number of purchases per service package anytime an order is correctly paid.

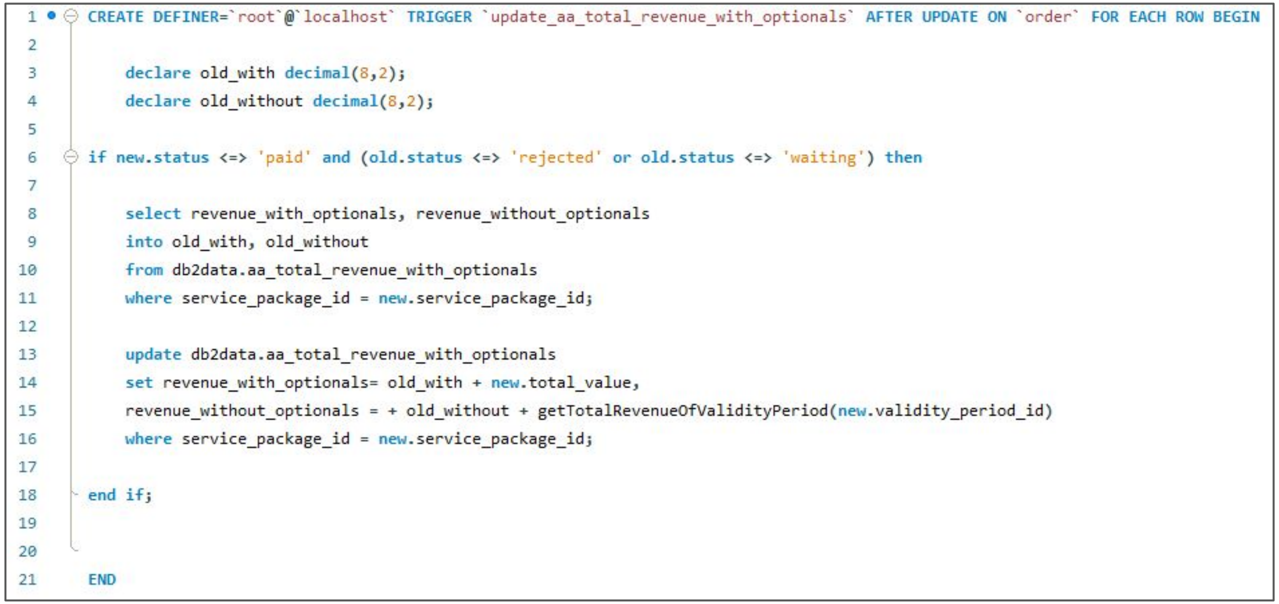
Triggers on ORDER 6/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_total_purchases_per_val` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

    declare oldPurchases integer;

    if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then

        select number_of_purchases
        into oldPurchases
        from db2data.aa_total_purchases_per_val
        where service_package_id = new.service_package_id and validity_period_id = new.validity_period_id;

        update db2data.aa_total_purchases_per_val
        set number_of_purchases = oldPurchases + 1
        where service_package_id = new.service_package_id and validity_period_id = new.validity_period_id;

    end if;

END
```

This trigger updates the table that keeps track of the number of purchases per validity period anytime an order is correctly paid.

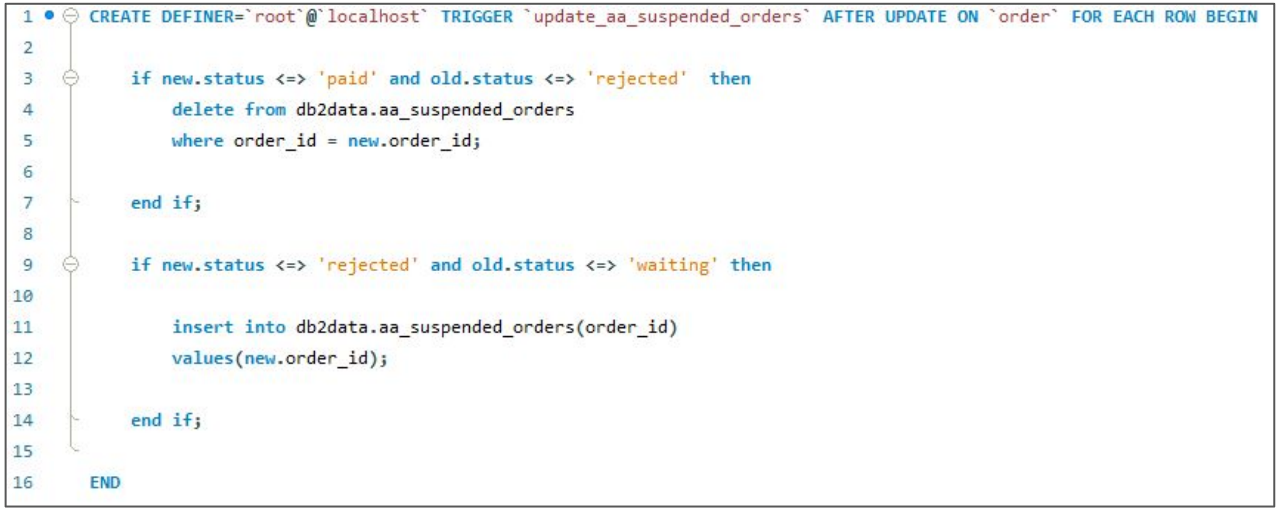
Triggers on ORDER 7/10



```
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_total_revenue_with_optionals` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

    declare old_with decimal(8,2);
    declare old_without decimal(8,2);

if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then

    select revenue_with_optionals, revenue_without_optionals
    into old_with, old_without
    from db2data.aa_total_revenue_with_optionals
    where service_package_id = new.service_package_id;

    update db2data.aa_total_revenue_with_optionals
    set revenue_with_optionals= old_with + new.total_value,
    revenue_without_optionals = + old_without + getTotalRevenueOfValidityPeriod(new.validity_period_id)
    where service_package_id = new.service_package_id;

end if;


END
```

This trigger updates the table that keeps track of the total revenue per service package with and without optional products.

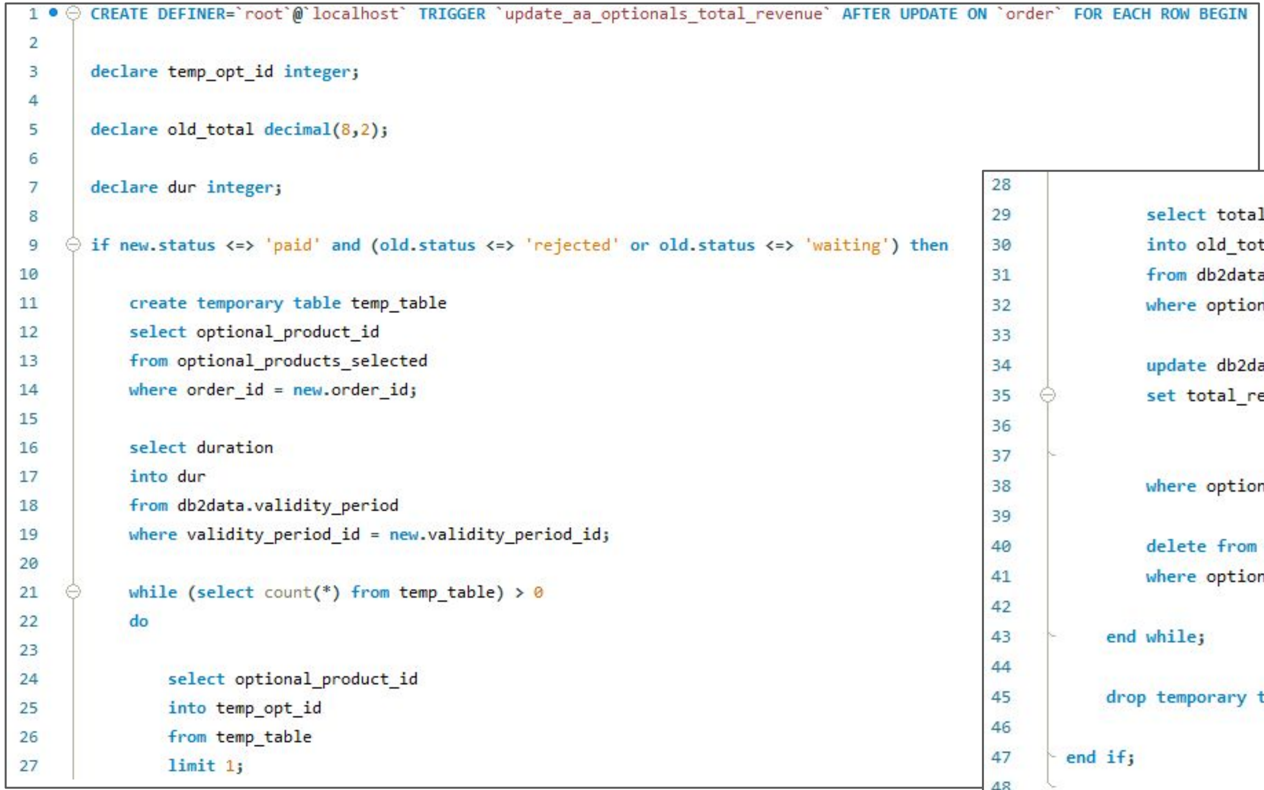
Triggers on ORDER 8/10



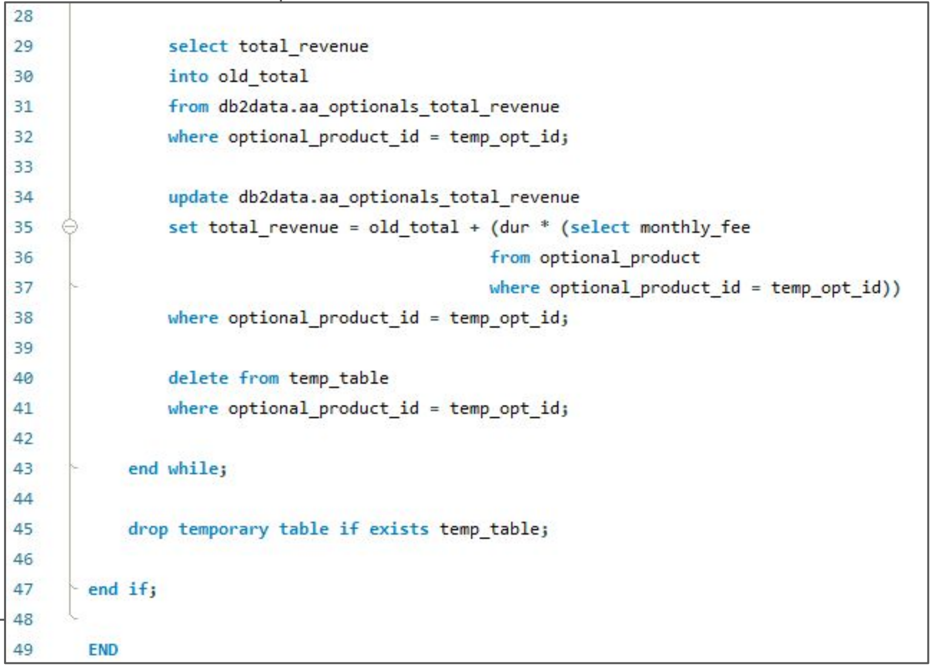
```
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_suspended_orders` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

    if new.status <=> 'paid' and old.status <=> 'rejected'  then
        delete from db2data.aa_suspended_orders
        where order_id = new.order_id;

    end if;

    if new.status <=> 'rejected' and old.status <=> 'waiting' then

        insert into db2data.aa_suspended_orders(order_id)
        values(new.order_id);

    end if;

END
```

This trigger populates and depopulates the table of suspended orders accordingly.

Triggers on ORDER 9/10 (code)



```sql
CREATE DEFINER=`root`@`localhost` TRIGGER `update_aa_optionals_total_revenue` AFTER UPDATE ON `order` FOR EACH ROW BEGIN

declare temp_opt_id integer;

declare old_total decimal(8,2);

declare dur integer;

if new.status <=> 'paid' and (old.status <=> 'rejected' or old.status <=> 'waiting') then

    create temporary table temp_table
    select optional_product_id
    from optional_products_selected
    where order_id = new.order_id;

    select duration
    into dur
    from db2data.validity_period
    where validity_period_id = new.validity_period_id;

    while (select count(*) from temp_table) > 0
    do

        select optional_product_id
        into temp_opt_id
        from temp_table
        limit 1;
```



```sql

        select total_revenue
        into old_total
        from db2data.aa_optionals_total_revenue
        where optional_product_id = temp_opt_id;

        update db2data.aa_optionals_total_revenue
        set total_revenue = old_total + (dur * (select monthly_fee
                                        from optional_product
                                        where optional_product_id = temp_opt_id))
        where optional_product_id = temp_opt_id;

        delete from temp_table
        where optional_product_id = temp_opt_id;

    end while;

    drop temporary table if exists temp_table;

end if;

END
```

explanation is on next slide

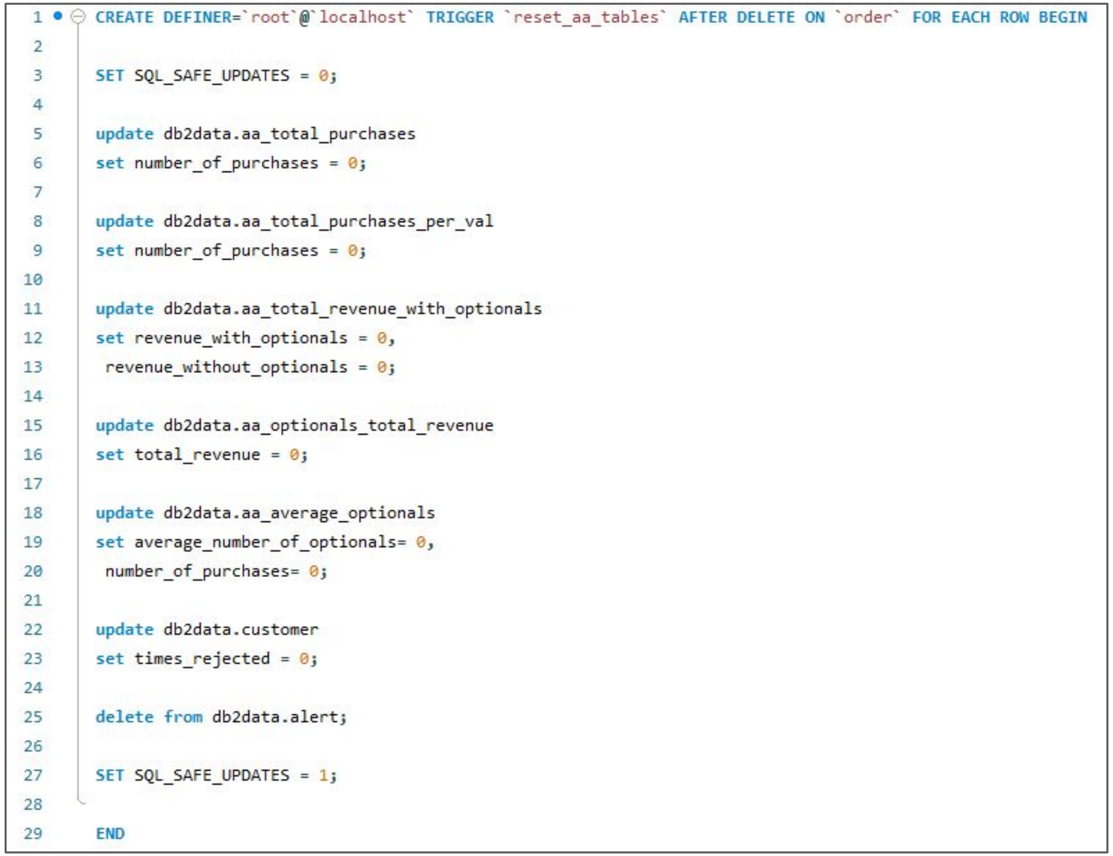
Triggers on ORDER 9/10 (explanation)

This trigger updates the table that keeps track of the total revenue per optional product.

To work it creates a temporary table of all the optional products selected in a paid order.

Then it extracts one row at the time from the temporary table and updates the corresponding optional product total revenue until the temporary table is empty.

Triggers on ORDER 10/10 (for development purpose)



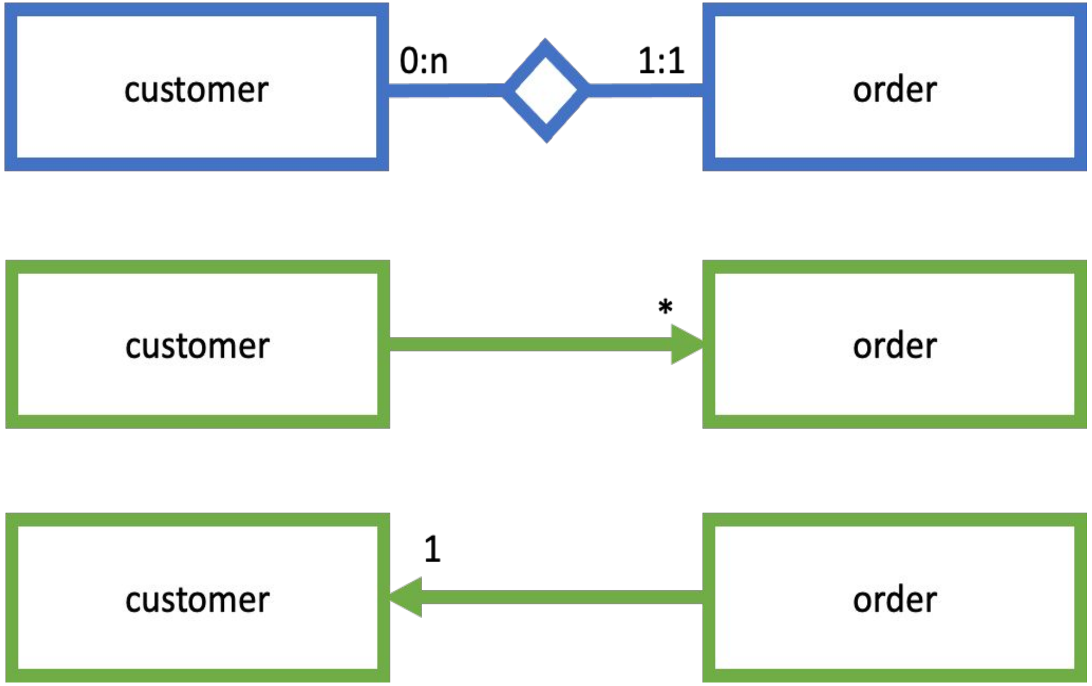
```
CREATE DEFINER=`root`@`localhost` TRIGGER `reset_aa_tables` AFTER DELETE ON `order` FOR EACH ROW BEGIN

SET SQL_SAFE_UPDATES = 0;

update db2data.aa_total_purchases
set number_of_purchases = 0;

update db2data.aa_total_purchases_per_val
set number_of_purchases = 0;

update db2data.aa_total_revenue_with_optionals
set revenue_with_optionals = 0,
 revenue_without_optionals = 0;

update db2data.aa_optionals_total_revenue
set total_revenue = 0;

update db2data.aa_average_optionals
set average_number_of_optionals= 0,
 number_of_purchases= 0;

update db2data.customer
set times_rejected = 0;

delete from db2data.alert;

SET SQL_SAFE_UPDATES = 1;

END
```

This is our trigger that resets all data from all the aggregated tables after an order is deleted.

The intention of this trigger is that when we manually delete an order from the database we must delete all orders.

We made it this way only for simplicity reasons (when we had to reset the database data during development), we thought that in this kind of application the orders (and consequently also other tables) should never be deleted as they are useful data for both operational level and management level.

ORM design



customer → order

@OneToMany

Reason:

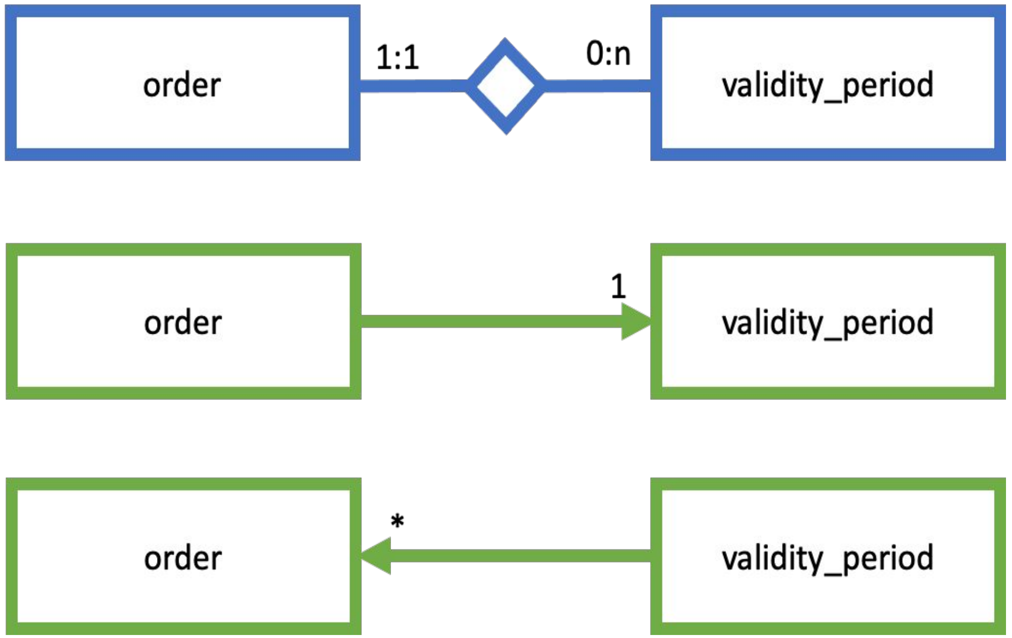
The same customer can place many orders

Order → customer

@ManyToOne

Reason: Each order is associated to only one customer

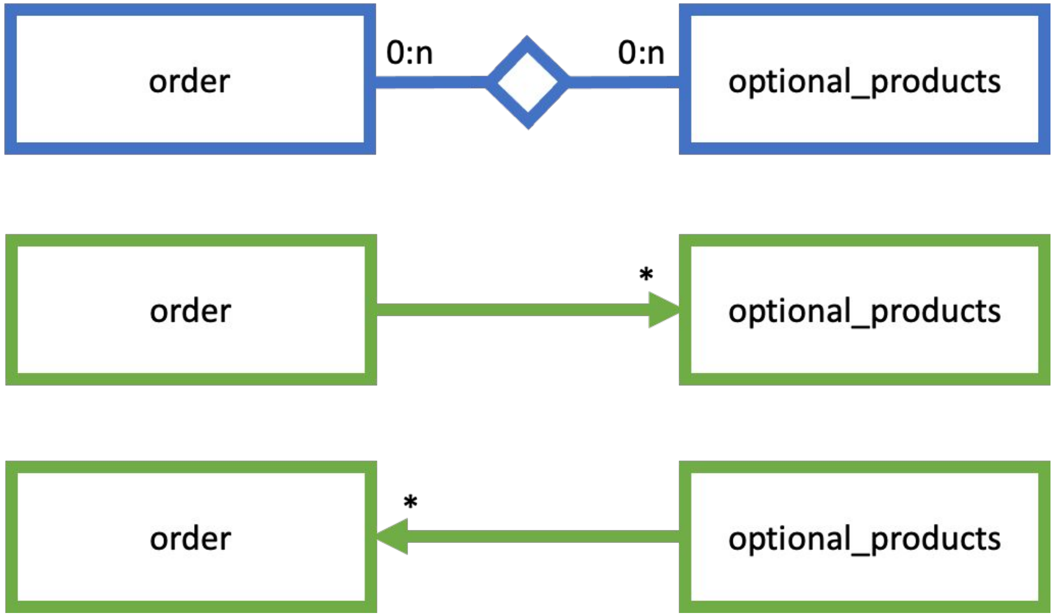
ORM design



order → validity_period
@ManyToOne
Reason: Every order has a validity period selected

validity_period → order
@OneToMany
Reason: The same validity period can be chosen in different orders

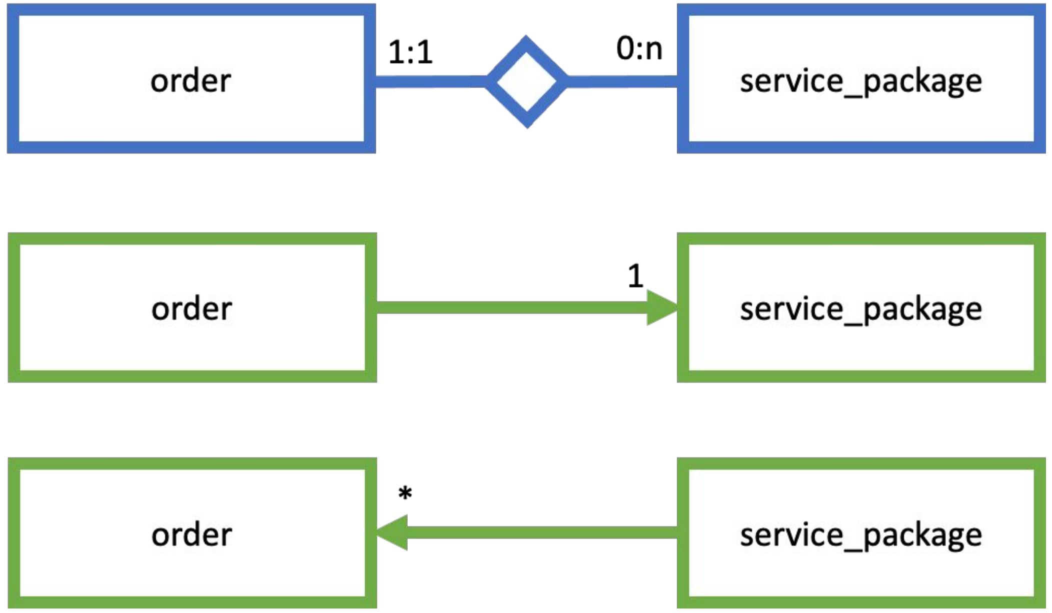
ORM design



order → optional_products
@ManyToMany
Reason: We can have in the same order several optional products selected

optional_products → order
@ManyToMany
Reason: the same optional product can be bought in different orders

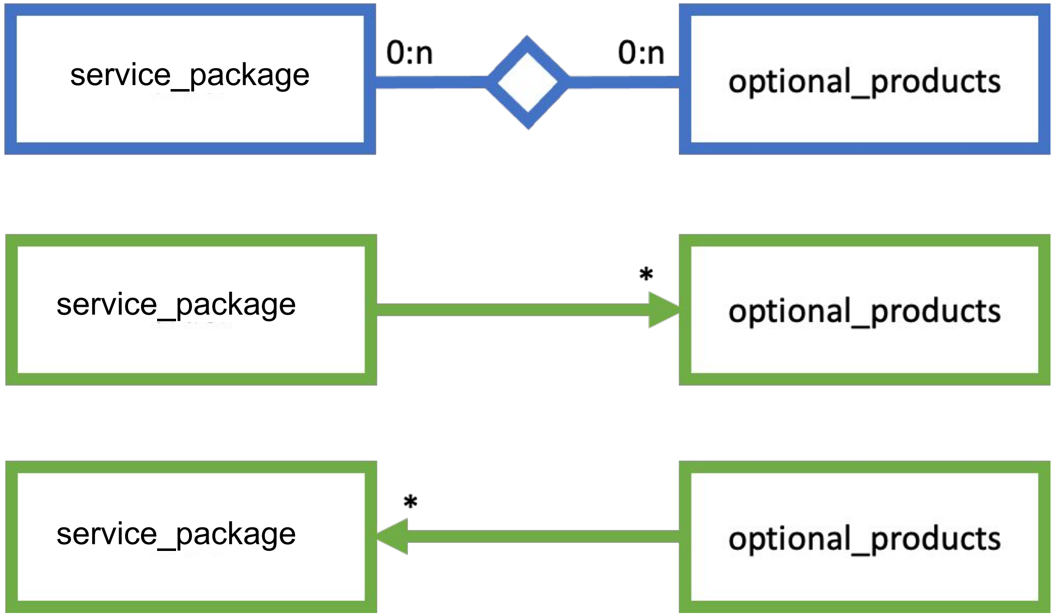
ORM design



order → service_package
@OneToMany
Reason: An order can have ONLY one service package

service_package → order
@ManyToOne
Reason: each service package can be bought multiple times

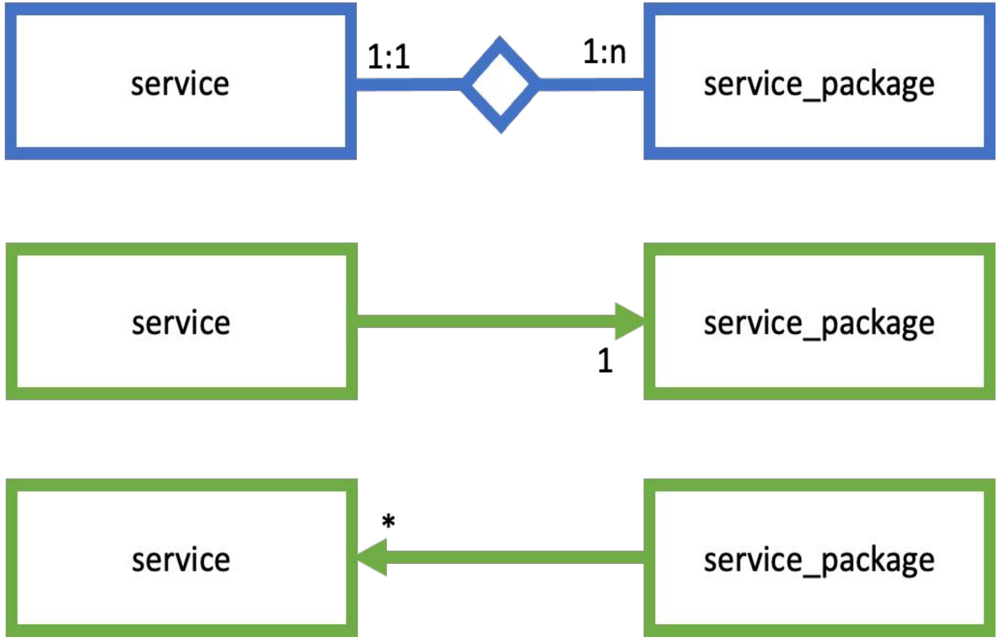
ORM design



service_package→ optional_products
@ManyToMany
Reason: A service package may have 0 or more optional products offered

optional_products → service_package
@ManyToMany
Reason: An optional product may be associated with more than one service package

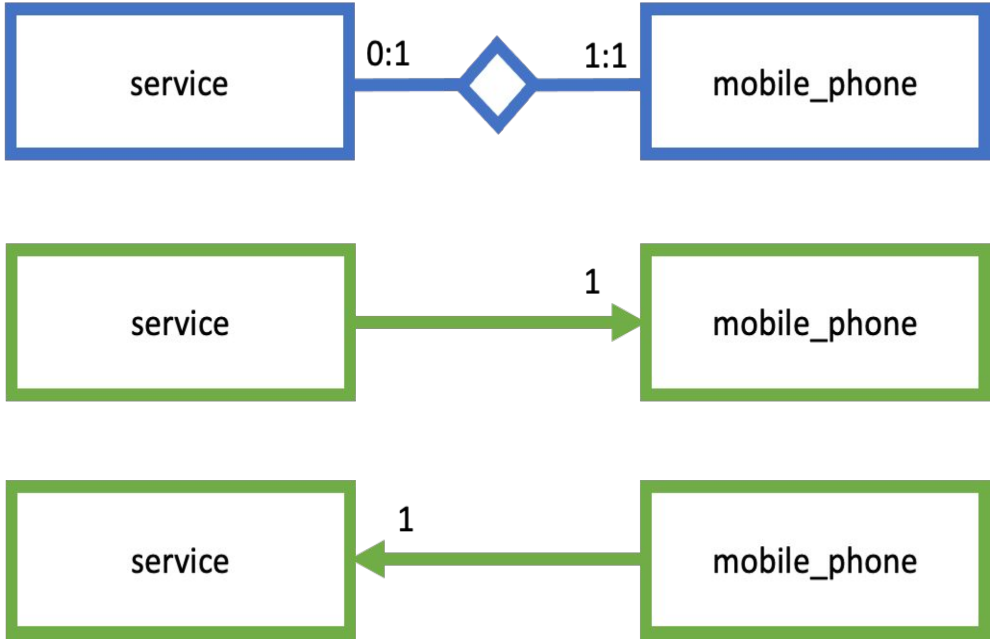
ORM design



service → service_package
@OneToMany
Reason: A service is linked to only one service package

service_package → service
@ManyToOne
Reason: each service contains one or more services

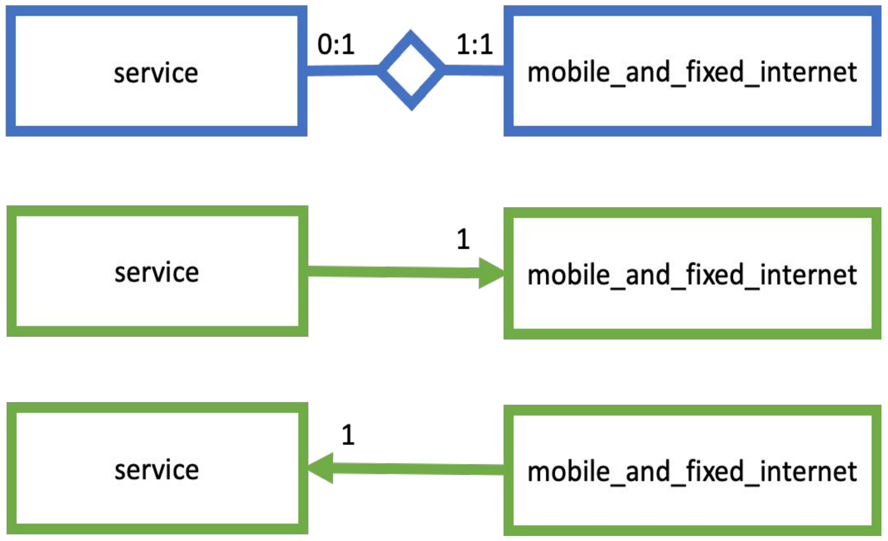
ORM design



service → mobile_phone
@OneToOne
Reason: A service may be of type mobile_phone

mobile_phone → service
@OneToOne
Reason: The tuple in mobile_phone specifies the attributes of the related service of type mobile_phone

ORM design


service
0:1
1:1
mobile_and_fixed_internet
service
1
mobile_and_fixed_internet
service
1
mobile_and_fixed_internet

service→mobile_and_fixed_internet
@OneToOne
Reason: A service may be of type mobile_internet or fixed_internet

mobile_and_fixed_internet → service
@OneToOne
Reason: The tuple in mobile_and_fixed_internet specifies the attributes of the related service of one of those two types

Entity Alert

```
@Entity
@Table(name = "alert", schema = "db2data")
@NamedQuery(name = "Alert.findAll", query = "SELECT a FROM Alert a")
public class Alert implements Serializable {
		private static final long serialVersionUID = 1L;
		public Alert() {super();}

		@Id
		@GeneratedValue(strategy = GenerationType.IDENTITY)
		private int alert_id;

		private String email;
		private BigDecimal amount;

		@Temporal(TemporalType.TIMESTAMP)
		private Date date_and_time;

		@OneToOne(fetch = FetchType.LAZY)
		@JoinColumn(name = "username")

		private Customer customer;

		//getters and setters…
}
```

Entity Customer

```
@Entity
@Table(name = "customer", schema = "db2data")
@NamedQuery(
        name = "Customer.checkCredentials",
        query = "SELECT c FROM Customer c WHERE c.username = ?1 and c.password = ?2")
public class Customer implements Serializable {
        private static final long serialVersionUID = 1L;

        public Customer() {}
        public Customer(String username, String password, String email) {
                this.username = username;
                this.password = password;
                this.email = email;
        }

        @Id
        private String username;
        private String password;
        private String email;
        private int times_rejected;
        @OneToOne(mappedBy = "customer", cascade = CascadeType.MERGE, orphanRemoval = true)
        Alert alert;
        @OneToMany(mappedBy = "customer", fetch = FetchType.LAZY)
        List<Order> orders;
        //getters and setters…
}
```

Entity Employee

```
@Entity
@Table(name = "employee", schema = "db2data")
@NamedQuery(        name = "Employee.checkCredentials",
                    query = "SELECT e FROM Employee e  WHERE e.username = ?1 and e.password = ?2")
public class Employee implements Serializable {
        private static final long serialVersionUID = 1L;

        public Employee() {super();}

        @Id
        private String username;
        private String password;

        //getters and setters…

}
```

Entity MobileAndFixedInternet

```
@Entity
@Table(name = "mobile_and_fixed_internet", schema = "db2data")
public class MobileAndFixedInternet implements Serializable {
        private static final long serialVersionUID = 1L;
        public MobileAndFixedInternet() {super();}
        public MobileAndFixedInternet(int n_giga, BigDecimal fee_extra_giga) {
                this.n_giga = n_giga;
                this.fee_extra_giga = fee_extra_giga;
        }

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private int internet_id;
        private int n_giga;
        private BigDecimal fee_extra_giga;
        @OneToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "service_id")
        private Service serviceInternet;

        //getters and setters…
}
```

Entity MobilePhone

```
@Entity
@Table(name = "mobile_phone", schema = "db2data")
public class MobilePhone implements Serializable {
	private static final long serialVersionUID = 1L;
	public MobilePhone() {super();}
	public MobilePhone(int n_minutes, int n_sms, BigDecimal fee_extra_minutes, BigDecimal fee_extra_sms) {
		this.n_minutes = n_minutes;
		this.n_sms = n_sms;
		this.fee_extra_minutes = fee_extra_minutes;
		this.fee_extra_sms = fee_extra_sms;
	}

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int mobile_phone_id;
	private int n_minutes;
	private int n_sms;
	private BigDecimal fee_extra_minutes;
	private BigDecimal fee_extra_sms;
	@OneToOne(fetch = FetchType.LAZY)
	@JoinColumn(name = "service_id")
	private Service serviceMobilePhone;
	//getters and setters…
}
```

Entity OptionalProduct

```
@Entity
@Table(name = "optional_product", schema = "db2data")
@NamedQueries({
 @NamedQuery(
 name = "OptionalProduct.findAll",
 query = "SELECT o FROM OptionalProduct o"),
 @NamedQuery(
    name = "OptionalProduct.findByName",
    query = "Select o FROM OptionalProduct o WHERE o.name = ?1")
    })

public class OptionalProduct implements Serializable {
        private static final long serialVersionUID = 1L;
        public OptionalProduct() {
                super();
        }
        public OptionalProduct(String name, BigDecimal cost) {
                this.name = name;
                this.monthly_fee = cost;
        }

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int optional_product_id;
private String name;
private BigDecimal monthly_fee;
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "optional_products_selected",
   joinColumns = @JoinColumn(name = "optional_product_id"),
   inverseJoinColumns = @JoinColumn(name = "order_id"))
private List<Order> orders;

@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "optional_products_available",
   joinColumns = @JoinColumn(name = "optional_product_id"),
   inverseJoinColumns = @JoinColumn(name = "service_package_id"))
private List<ServicePackage> servicePackages;
//getters and setters…
}
```

Entity Order

```java
@Entity
@Table(name = "order", schema = "db2data")
@NamedQueries({
	@NamedQuery(
	name = "Order.findAllRejectedOfCustomer",
	query = "SELECT o FROM Order o WHERE
o.customer.username = ?1 AND o.status = 'rejected'")
	})

public class Order implements Serializable {
	private static final long serialVersionUID = 1L;
	public Order() {
		super();
	}
	public Order(Date creation_date, Date start_date,
String status, BigDecimal total_value) {
		this.creation_date = creation_date;
		this.start_date = start_date;
		this.status = status;
		this.total_value = total_value;
		this.optionalProducts = new ArrayList();
	}
```

```java
@Id
@GeneratedValue(
strategy = GenerationType.IDENTITY)
private int order_id;
@Temporal(TemporalType.TIMESTAMP)
private Date creation_date;
@Temporal(TemporalType.DATE)
private Date start_date;

private String status;
private BigDecimal total_value;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "username")
private Customer customer;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "validity_period_id")
private ValidityPeriod validityPeriod;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "service_package_id")
private ServicePackage servicePackage;
@ManyToMany(mappedBy = "orders", fetch = FetchType.LAZY)
private List<OptionalProduct> optionalProducts;
//getters and setters…
}
```

Entity Service

```
@Entity
public class Service implements Serializable {
        private static final long serialVersionUID = 1L;
        public Service() {super();}
        public Service(String type) {this.type = type;}

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private int service_id;
        private String type;
        @OneToOne(mappedBy = "serviceMobilePhone", cascade = CascadeType.PERSIST)
        private MobilePhone mobilePhone;

        @OneToOne(mappedBy = "serviceInternet", cascade = CascadeType.PERSIST)
        private MobileAndFixedInternet mobileAndFixedInternet;

        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "service_package_id")
        private ServicePackage myServicePackage;
        //getters and setters…
}
```

Entity ServicePackage

```
@Entity
@Table(
        name = "service_package",
        schema = "db2data")
@NamedQueries({
        @NamedQuery(
                name = "ServicePackage.findAll",
                query = "SELECT p FROM ServicePackage p"),
        @NamedQuery(
                name = "ServicePackage.findByName",
                query = "SELECT p FROM ServicePackage p WHERE p.name = ?1")})
public class ServicePackage implements Serializable {
        private static final long serialVersionUID = 1L;
        public ServicePackage() {super();}

        public ServicePackage(String name) {
                this.name = name;
                this.optionalProducts = new ArrayList();
                this.services = new ArrayList();
                this.validityPeriods = new ArrayList();
        }
//getters and setters…
}
```

Entity ValidityPeriod

```
@Entity
@Table(name = "validity_period", schema = "db2data")
@NamedQuery(name = "ValidityPeriod.findAll", query = "SELECT v FROM ValidityPeriod v")
public class ValidityPeriod implements Serializable {
	private static final long serialVersionUID = 1L;
	public ValidityPeriod() {super();}
	public ValidityPeriod(BigDecimal monthly_fee, int duration) {
		this.monthly_fee = monthly_fee;
		this.duration = duration;}
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int validity_period_id;
	private BigDecimal monthly_fee;
	private int duration;

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

	@ManyToOne(fetch = FetchType.LAZY)
	@JoinColumn(name = "service_package_id")
	ServicePackage linkedServicePackage;
	//getters and setters…
}
```

Functional analysis of the specification

Pages(views), view components, events, actions

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

CONSUMER APPLICATION
The consumer application has a public Landing page with a form for login and a form for registration.
Registration requires a username (which can be assumed as the unique identification parameter), a password and an email.
Login leads to the Home page of the consumer application. Registration leads back to the landing page where the user can log in.

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

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

Functional analysis of the specification

Pages(views), view components, events, actions

A service package has an ID and a name (e.g., “Basic”, “Family”, “Business”, “All Inclusive”, etc). It comprises one or more services. Services are of four types: fixed phone, mobile phone, fixed internet, and mobile internet. The mobile phone service specifies the number of minutes and SMSs included in the package plus the fee for extra minutes and the fee for extra SMSs. The fixed phone service has no specific configuration parameters. The mobile and fixed internet services specify the number of Gigabytes included in the package and the fee for extra Gigabytes. A service package must be associated with one validity period. A validity period specifies the number of months (12, 24, or 36). Each validity period has a different monthly fee (e.g., 20€/month for 12 months, 18 €/month for 24 months, and 15€ /month for 36 months). A package may be associated with one or more optional products (e.g., an SMS news feed, an internet TV channel, etc.). The validity period of an optional product is the same as the validity period that the user has chosen for the service package. An optional product has a name and a monthly fee independent of the validity period duration. The same optional product can be offered in different service packages.

Functional analysis of the specification

Pages(views), view components, events, actions

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

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

Functional analysis of the specification

Pages(views), view components, events, actions

When the user presses the BUY button, an order is created. The order has an ID and a date and hour of creation. It is associated with the user and with the service package, its validity period and the chosen optional products. It also contains the total value (as in the CONFIRMATION page) and the start date of the subscription. After creating the order, the application bills the customer by calling an external service. If the external service accepts the billing, the order is marked as valid and a service activation schedule is created for the user. A service activation schedule is a record of the services and optional products to activate for the user with their date of activation and date of deactivation.

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

Functional analysis of the specification

Pages(views), view components, events, actions

EMPLOYEE APPLICATION

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

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

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

List of components

- Client components
  - Servlets
    - CheckLogin
    - CreateOptionalProduct
    - CreateOrderPaid
    - CreateOrderRejected
    - CreateServicePackage
    - CustomerRegistration
    - GoToBuyService
    - GoToConfirmationAfterLogin
    - GoToConfirmationPage
    - GoToConfirmationWithOldOrder
    - GoToHomeCustomer
    - GoToHomeEmployee
    - GoToShowStatistics
    - Logout
  - Filters
    - EmployeeLogCheck
  - Views
    - index.html
    - CustomerHome.html
    - BuyAServicePage.html
    - ConfirmationPage.html
    - EmployeeHome.html
    - SalesReport.html
  - Java Beans
    - TempOrder
  - Javascript & CSS
    - employee.js
    - tables.css
    - flex.css

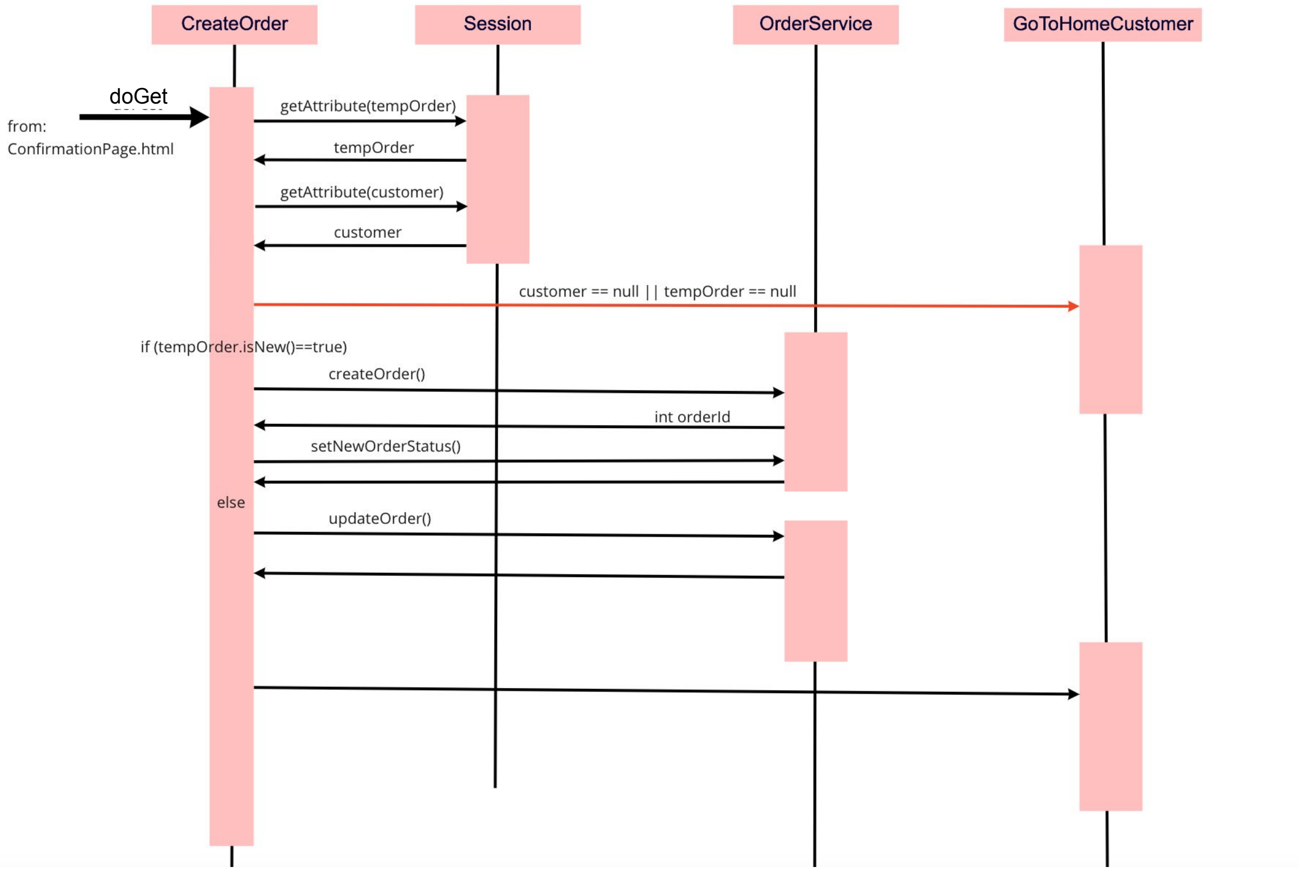
List of components

- BackEnd Components
  - Entities
    - Alert
    - Customer
    - Employee
    - MobileAndFixedInternet
    - MobilePhone
    - OptionalProduct
    - Order
    - Service
    - ServicePackage
    - ValidityPeriod
    - (+ all the ones of the aggregated tables)
  - Business Components (EJBs) (all stateless)
    - AaTablesService
    - CustomerService
    - EmployeeService
    - OptionalProductService
    - OrderService
    - ServicePackageService
    - ValidityPeriodService

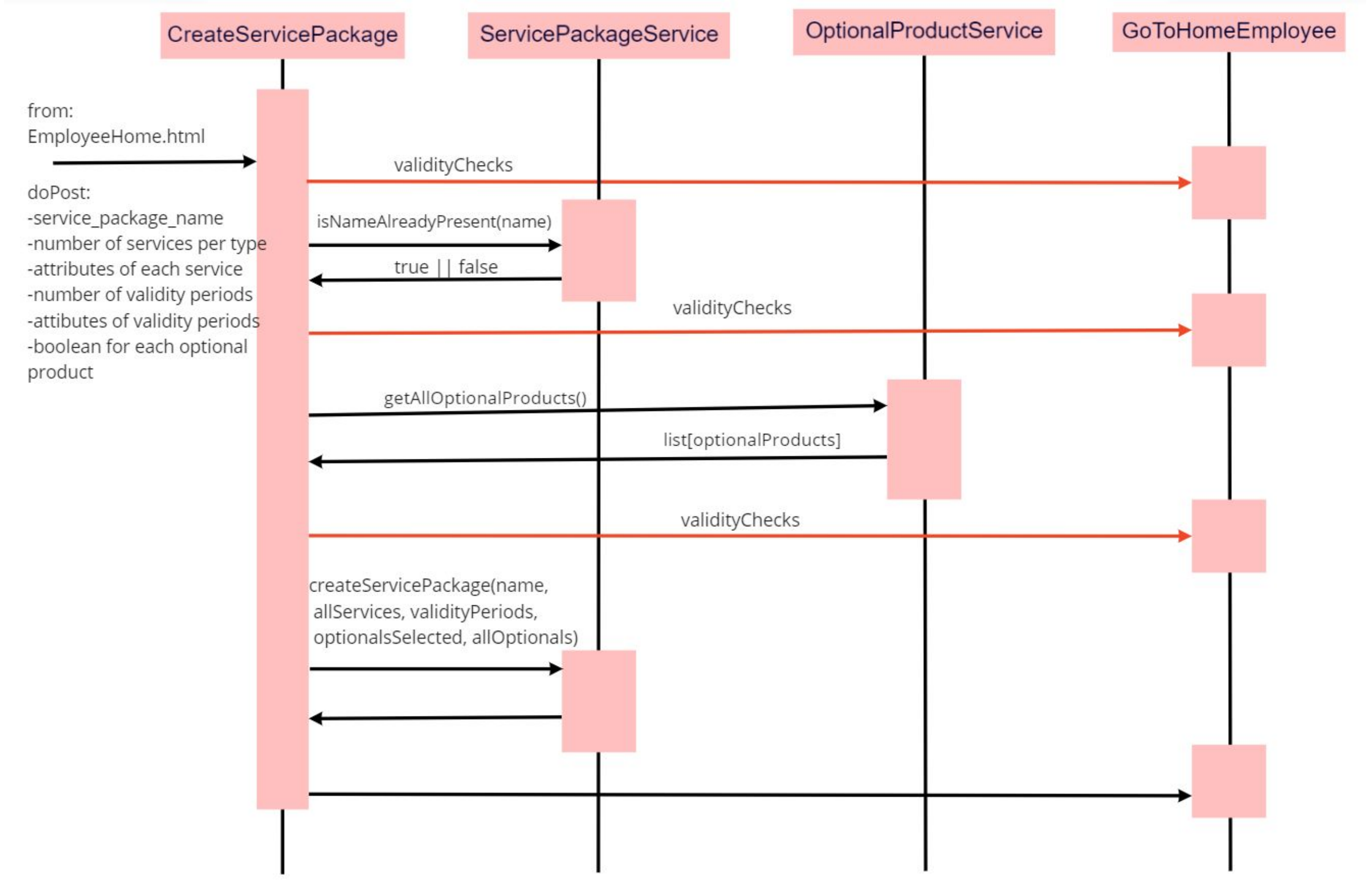
List of components

- Business Components (EJBs)
  - AaTablesService
    - stateless
    - getTotalPurchasesOfPackages()
    - getTotalPurchasesOfPackagesPerVal()
    - getTotalRevenueWithOptionals()
    - getAverageOptionals()
    - getInsolventUsers()
    - getSuspendedOrders()
    - getAlerts()
    - getOptionalsTotalRevenueBestSellers()
  - CustomerService
    - stateless
    - checkCredentials(username, password)
    - createCustomer(username, password, email)
    - isCustomerAlreadyPresent(username)
  - EmployeeService
    - stateless
    - checkCredentials(username, password)
    - isEmployeeAlreadyPresent(username)
  - OptionalProductService
    - stateless
    - getAllOptionalProducts()
    - isOptionalProductAlreadyPresent(optProdName)
    - addNewOptionalProduct(name, cost)
  - OrderService
    - stateless
    - createOrder //constructor
    - setNewOrderStatus(orderId, status)
    - getAllRejectedOrdersOfCustomer(username)
    - getOrder(orderId)
    - updateOrder(orderId, status, newCreationDate)
  - ServicePackageService
    - stateless
    - getAllAvailableServicePackages()
    - getServicePackage(servicePackageId)
    - isNameAlreadyPresent(name)
    - createServicePackage(name, allServices, validityPeriods)
  - ValidityPeriodService
    - stateless
    - getAllValidityPeriods()
    - getValidityPeriod(validityPeriodId)

UML sequence diagram of CreateOrder



UML sequence diagram of CreateServicePackage


CreateServicePackage
ServicePackageService
OptionalProductService
GoToHomeEmployee
from:
EmployeeHome.html
doPost:
-service_package_name
-number of services per type
-attributes of each service
-number of validity periods
-attibutes of validity periods
-boolean for each optional
product
validityChecks
isNameAlreadyPresent(name)
true || false
validityChecks
getAllOptionalProducts()
list[optionalProducts]
validityChecks
createServicePackage(name,
allServices, validityPeriods,
optionalsSelected, allOptionals)