Date: 24-Mar-2022

Group name: MiniMoment

Project:

MiniMoment

By:

Alex Maccagnan
Cristian-Valentin Purcea
Teodor Jonasson

Code: GitHub

List of figures: TODO

List of appendices: TODO

TABLE OF CONTENTS

Intro

We decided to design our database around a simple business idea.
It is a subscription service where customers can sign up to have 10 of their pictures sent to their door once a month.

We need to design a database that would be able to store all the needed information for the service and we need a frontend system that can use that information to render and manage our database.

Problem description

The first hurdle in the project is the data storage. We have to choose and design a database that is able to accomodate our project's needs.

The database for our project needs to be able to store:

- User information including addresses
- Delivery addresses
- Active and inactive subscriptions
- Orderable products:
    - User pictures
    - Picture frames
- Discount codes and their types
- A history of all orders
- Invoices of all purchases

We also have to be able to convert these data storage needs into TODO database types.

For the relational database we chose to use MYSQL.

For the document database we chose TODO

The frontend needs to TODO

Explanation of choices:

In order to complete our project we chose the following software solution.

Databases

MySQL

For the relational database we decided to go with MySQL as this is what we as a team know best and would be able to create a complex system within a reasonable timeframe.

Document db 1

TODO

Programming languages

TODO

Other tools

MySQL Workbench:
We use MySQL workbench to design the entity relationship diagram as it is the best tool we have seen so far for this task.

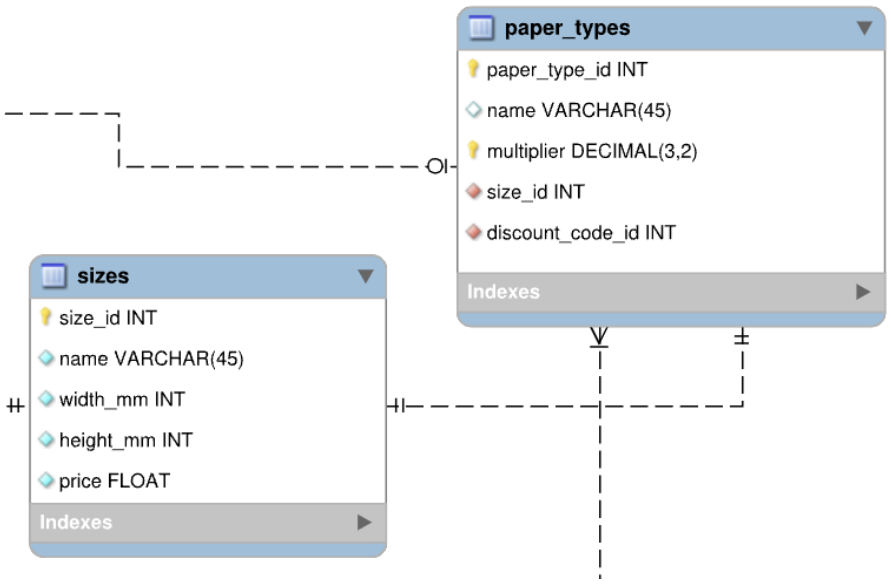
Mycli:
We use mycli as a terminal client occasionally to query the database or to prototype functions.

Relational database

Intro relational databases

The relational database maps your data objects and the relationships between them. It lets you do this by creating tables that describe each data object that you will be storing. Each table has a set of attributes that can have relationships to attributes in other tables and therefore relate the two. This is often done by each table having a unique id that can then be referenced by another table to show the relationship. Having these relationships is one of the key characteristics and what brings a lot of value to the relational database.

Each table is set up with a table name, a set of columns that describe each attribute of the data object that you are storing. Each row in the table then represents a data entry in the system. Here is an example of 2 tables with a set of attributes and relations to one and other.



When creating these tables that describe your data, you can set up different rules for each attribute of the table like the following:

- NOT NULL: This attribute is not allowed to be null. So if a new data entry comes in where an attribute which is set to be NOT NULL, the database will report the error and no data will actually be stored in the database.
- UNIQUE: This rule is mostly used for the table's id and assures that no other data point can have the same id, since it would be impractical for several objects of the same type.
- VARCHAR(10): This rule limits the attribute to only be 10 characters. With a request to put data in the database where this attribute has more than 10 characters, it would fail and report the error.
- PRIMARY KEY: This sets the main identifier of the table and does not allow for duplicates of this entry. Thereby this is often the id of the table.

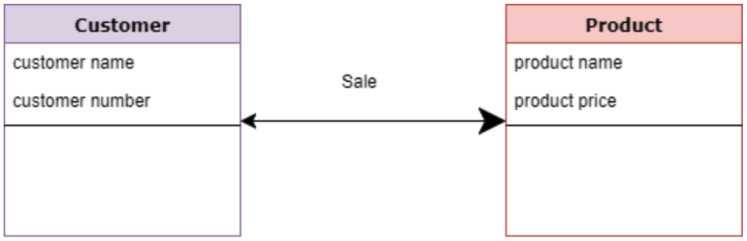
Database design

The design process for building a database consists of a series of different data models that in the end showcase working ERD (Entity Relationship Diagram). To have the final ERD, you must first conceptualise your database structure which you do in the conceptual modelling fase. Once you've done this, you extend the conceptual model and build the logical model by adding on more knowledge to your data structure. The logical model is then the foundation for you to build the physical model which translates into your database's final ERD.

Entity / relationship model

Conceptual

The conceptual data model is an organised overview of the database, its tables and their relations. Your goal, while creating the conceptual model, is to establish entities, what attributes they might have as well as the entities relationships.



At this stage of the database design modelling there's hardly any detail on the actual database structure. In this step, stakeholders can take part in the creation of the conceptual model.

Items described in the conceptual model:

- Entrity**:** a real-world thing that you want to store as data. Ex. a book, user, etc.
- Attribute**:** A characteristic of the entity or its properties. Ex. author, username, etc.

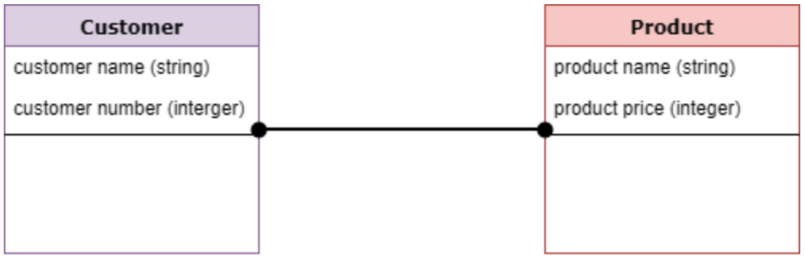
- Relationship**:** An association between two entities. Ex. book => has an author.

Characteristics of the conceptual model:

- Is a full overview of the organisation and the business concept as a whole.
- It is designed for a business audience to understand the concept.
- Developed independently from hardware specifications with the only focus being to represent data as a user will see it.

Logical

In the logical data model creation, you define the data structure and further define the relationships between them. You do this by setting data types on different attributes. The logical data model builds upon the conceptual model and adds further information on the elements. With the logical data model you are able to build the foundation for the physical and final model of your database even though the modelling structure remains generic.

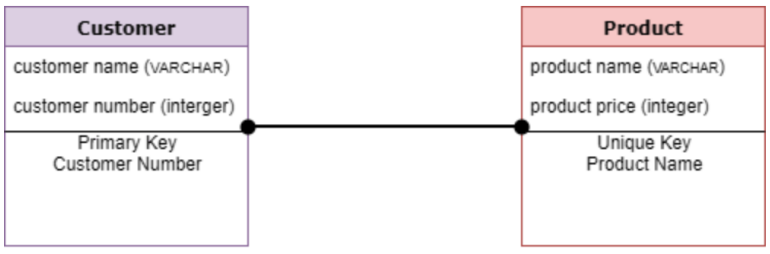


Characteristics of a logical data model:

- It describes the data it needs to build a functional project.
- It is designed independently from the actual database model.
- It is the foundation for the physical modelling done for a database.
- Data attributes have data types.

Physical

In the physical data model you describe the database specific implementation of the data model. Here you describe each attribute with the aforementioned rules like; NOT NULL, UNIQUE and PRIMARY KEY. You also specify what data type the attribute has, like INT, VARCHAR and BOOL, as well as any other extra rules the attribute of the table must follow. With the Physical data model you are able to generate the schema for your database so you can start working with it. As part of this, you also add keys, indexes, triggers and other RDBMS (Relational DataBase Management System) features.



Characteristics of a physical data model:

- Describes the data needed for a single project or application.
- Fully describes the relations between each table's attributes.
- Attributes should have exact data types and specifications and or default values.
- These attributes are described; primary keys, foreign keys, indexes, authorisations, triggers, etc.

Normalisation process

In the normalisation process, you look through the aforementioned ERD that you have developed through the three modelling fases, and look for ways to optimise your diagram. Would some relations be better as a seperate table, or could some tables be split up for clearer relations. This process is meant to optimise the ERD by making the database more flexible, eliminating redundancy and inconsistent dependency.

This part is especially important because of the following:

- Redundant data will waste disk space, increase maintenance problems, slow the server and increase costs of hosting, if you are using a service like AWS (Amazon Web Services) or Azure.
- If there are duplicate entrances of data or the data exists in several tables, it must be changed everywhere in the exact same way, or everything becomes incomprehensible to developers.
- Inconsistent dependency is when tables and data are not linked properly or with an inconsistent set of rules. This makes joining tables and accessing related data very hard because the path to the data may be missing, confusing or broken.

When going through the normalisation process, there are some rules that can be applied during the process. These include:

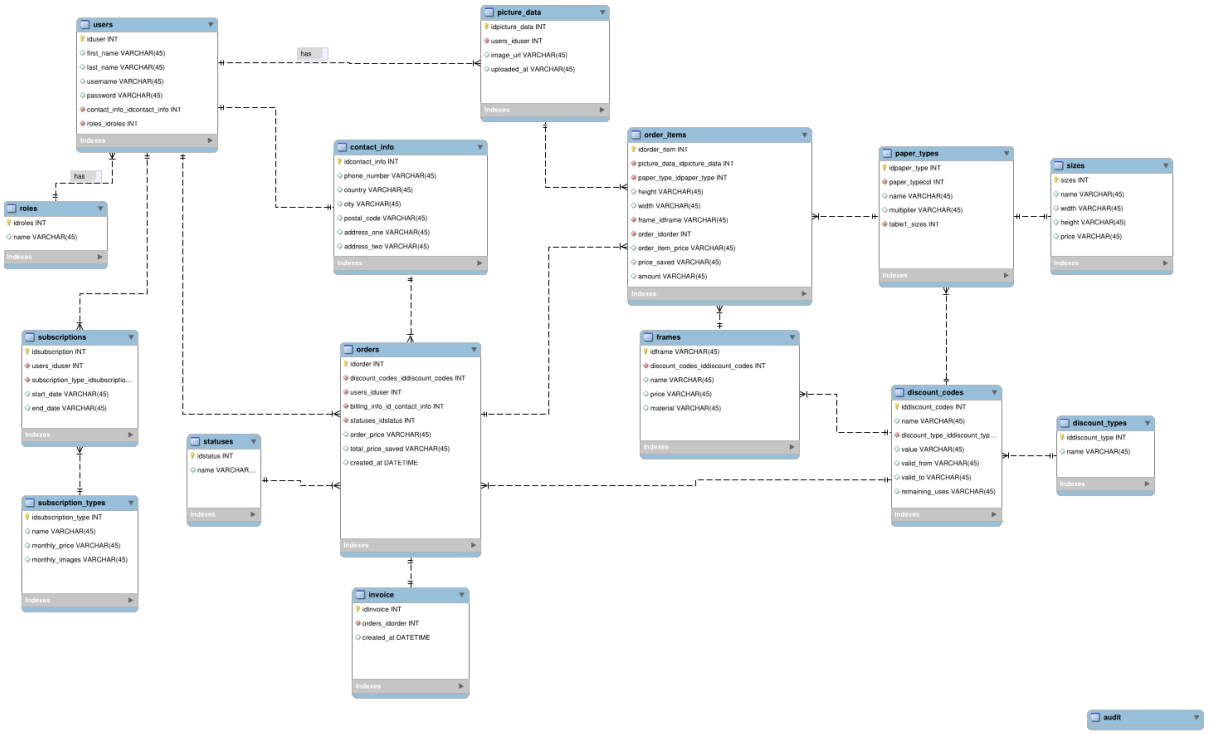
- Eliminate repeating data in individual tables.
- Creating separate tables for each set of data that relates to several tables.
- Identifying each of these tables with a primary key.

Physical data model

In order to design our database we began by creating entities that we thought we would need. This is a very simple diagram that only contains the names of the entities that will likely be used in our database.

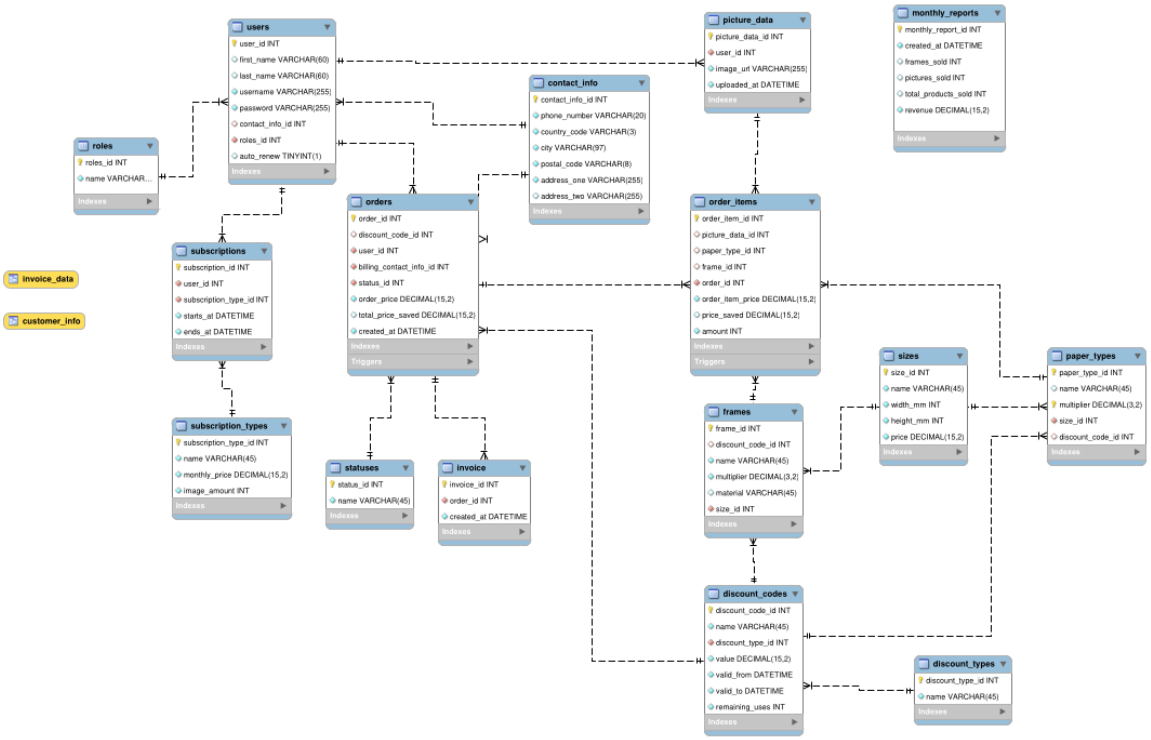
Entities for the database

We then followed that up with a conceptual model. In this model, we tried to link together these entities to see what associations would likely end up in our database. In this process began naming the attributes that would be stored in the entities. During this we discovered that some of our entities had to be split up into multiple tables and we did just that. Because we used MySql Workbench as our modelling tool, those attributes all had their default types.



Conceptual model - MySQL Workbench

Lastly, we converted our conceptual model into a fully fledged Entity Relationship Diagram. We included all foreign key and private key constraints, every association was validated and changed every attribute's type to what we thought was needed.



Completed erd

With this done, we used MySQL Workbench's reverse engineer tool to generate some creation scripts. After some tweaking and cleaning up of the code we were ready to move to creating stored objects. -> TODO -> STORED OBJECTS

MISSING INFORMATION:

What do our tables do:

- Brief explanation about why we chose to make these tables and perhaps any potential pitfalls
- What is meant to be stored in what tables, fxm user and address and billing info: How are they related

Data types - TODO

FLOATS vs DECIMAL(15,2) ?

Primary and foreign keys - TODO

??

Indexes - TODO

```
Gotta make some custom one(s).
   1. Run benchmarks on our current views to see how long they take
   2. See can be improved by making custom indexes on random
      variables i guess idk
   3. Keep the index that makes the largest difference, if any
   4. Document that shit here
```

Constraints and referential integrity - later

Stored objects - WIP

MySQL allows us to store complex objects on a database level. These allow us to implement behaviour at a database level.

We decided to implement at least one of each of these stored objects in our project and have found a fitting use for each.

Functions

MySQL has many useful built-in functions. Many of these are very useful when trying to aggregate data sets. One such example is CONCAT, it allows us to concatenate two or more strings together, for example to form a full name from a first name and last name stored on the database. Another is COUNT, this sums the total of the provided values.

While these functions are very useful on their own, we are also able to define our own functions. We have identified two places where functions would make interacting with the database easier. In our naming we append the `fn_` tag to our functions to make them instantly recognizable.

One such function we made was `fn_calculate_total`. This function takes two parameters, ‘price’ and ‘discounted’ of type DECIMAL(15,2). The simple goal of this function is to calculate the total cost after discounts have been applied. We use this function in our stored procedures to help calculate total sales in a given time period and also in a view we made to show more in-depth invoice information.

```
DROP FUNCTION IF EXISTS `fn_calculate_total`$$
DELIMITER $$
CREATE FUNCTION `fn_calculate_total`(
    `price` DECIMAL(15, 2),
    `discounted` DECIMAL(15, 2)
    )
RETURNS DECIMAL(15, 2)
DETERMINISTIC
BEGIN
IF `discounted` IS NULL THEN
    RETURN `price`;
ELSE
    RETURN `price` - `discounted`;
END IF;
END $$
DELIMITER ;
```

fn_calculate_total

Another function we decided to implement is called `fn_calculate_order_item_price`. This function exists because of the way our table structure is. We decided to keep two tables. One that contains an item's size and base price, and another table contains the paper type (such as glossy, premium and cake paper) and a multiplier. This means that if I want a picture of 10x10 and the base price is 100kr, but I choose premium paper, with a multiplier of 1.5x, the total price of that order is 150kr. We need a relatively simple way to calculate this data, and so this function was created.

As we have two products we decided to make a function that would accept the data for all of them, and count the total. It takes in as parameters: Frame price, frame multiplier, paper price, paper multiplier. Once we have all those values, we do simple multiplication on them and return the result.

One of the issues we ran into was that some of the passed parameters could be null. A simple example of this is a user ordering a picture but no frame. To resolve this we used a built-in function called IFNULL. IFNULL takes a value, and a default. If the value given to the function is null, it will instead return the default. We used this to set any potential null values in our data to 0 to make our function work as expected.

```
DROP FUNCTION IF EXISTS `fn_calculate_order_item_price`$$
DELIMITER $$
CREATE FUNCTION `fn_calculate_order_item_price`(
    `frame_price` DECIMAL(15, 2),
    `frame_multiplier` DECIMAL(15, 2),
    `paper_price` DECIMAL(15, 2),
    `paper_multiplier` DECIMAL(15, 2),
    `amount` INT
    )
RETURNS DECIMAL(15, 2)
DETERMINISTIC
BEGIN
    RETURN (IFNULL(`frame_price`, 0) * IFNULL(`frame_multiplier`, 0) * IFNULL(`amount`, 0)) +
           (IFNULL(`paper_price`, 0) * IFNULL(`paper_multiplier`, 0) * IFNULL(`amount`, 0));
END $$
DELIMITER ;
```

fn_calculate_oder_item_price

This function is being used in a stored procedure named `get_order_item_price` to help count the total cost of an order.

Stored procedures

MySQL also allows the creation of another type of stored objects, which are procedures. An SQL procedure is essentially a block of SQL statements that are executed in a sequence when the procedure gets called.
They can be called either from within a MySQL script by using `CALL` followed by the name of the procedure, or from within the code of our application.

The first procedure that we implemented is `use_discount`. It takes in an id of type `INT` as its parameter and updates the entry with that id from the table `discount_codes` to reflect the usage of a discount code. It does so by taking the `remaining_uses` attribute of that entry and decrementing it by one.

```
-- ------------------------------------------------------------------------------------------------------------
-- Procedure `use_discount` -> Decrements the amount of remaining uses for the given discount.
-- ------------------------------------------------------------------------------------------------------------
DELIMITER $$
DROP PROCEDURE IF EXISTS use_discount$$
CREATE PROCEDURE use_discount(IN `id` INT)
BEGIN
    UPDATE `discount_codes` SET `remaining_uses` = `remaining_uses` - 1 WHERE `discount_code_id` = `id`;
END $$
DELIMITER ;
```

use_discount

The next procedure that we implemented is `get_revenue`. This one is used to calculate the revenue that has been generated in a given time interval. To do so, we made the procedure take two `IN` parameters that mark the `start` and the `end` dates of this interval.
The calculated revenue is then stored in the third parameter, which is of type `OUT`.

We make use of the function `fn_calculate_total` to get the amount of money that was earned after the potential discounts have been applied, and we sum these amounts for every order that was created in between the start and the end dates.

It is worth mentioning that we built this function with the idea of calculating the revenue on a monthly basis, which is why the function only takes into consideration the year and the month of each `DATETIME` parameter when determining the amount.

```
-- ------------------------------------------------------------------------------------------------------------------------
-- Procedure `get_revenue` -> Calculates and retrieves the revenue generated within the given range of year and month.
-- ------------------------------------------------------------------------------------------------------------------------
DELIMITER $$
DROP PROCEDURE IF EXISTS get_revenue$$
CREATE PROCEDURE get_revenue(IN `start` DATETIME, IN `end` DATETIME, OUT `revenue` DECIMAL(15,2))
BEGIN
    SELECT SUM(fn_calculate_total(`order_price`, `total_price_saved`))
        INTO `revenue`
    FROM `orders`
        WHERE YEAR(`created_at`) >= YEAR(`start`) AND YEAR(`created_at`) <= YEAR(`end`) AND
              MONTH(`created_at`) >= MONTH(`start`) AND MONTH(`created_at`) <= MONTH(`end`);
END $$
DELIMITER ;
```

get_revenue

Along with the monthly revenue, we thought it might be worth it to also be able to see how many products we have sold within a specified amount of time. When we talk about products, we mean the pictures that a user has ordered, and the frames that we offer through our services.

The information about which products have been bought can be seen in the `order_items` table where each order item can have: either an id to the picture that was ordered, an id to a frame, or both ids in case the customer ordered a picture with a fitting frame, along with an `amount` of how many copies of that exact order item the customer wanted.

With that in mind, we created a procedure, `get_products_sold`, that calculates how many of these items we managed to sell. Just like the previous procedure, it takes in a `start` and an `end` input parameters, along with two booleans `count_pictures` and `count_frames` which are used so that we can get either a count of all pictures, frames, or both of them all together.

Lastly, we pass to our procedure a `total` parameter of type `INT` in which we store the results of our count.

In order to determine our total of products, we take each entry of our `order_items` table and we do the following calculation:

```
IFNULL((`picture_data_id` / `picture_data_id` * `amount`),0)
    * `count_pictures` +
IFNULL((`frame_id` / `frame_id` * `amount`),0)
    * `count_frames`))
```

Products calculation

We take each `picture_data_id` and divide it by itself to get to the number `1`, which we multiply by the `amount` to get how many pictures come with that specific order item.
In case the order item does not have a picture, then the picture id will be null, and thus the whole calculation will also equal to null.
That is why we wrapped it with an `IFNULL` that sets the result to `0` if it receives a null value.

We applied the same principle in the case of the frame, and then we multiplied each of the calculations by their respective boolean: `count_pictures` and `count_frames` to leave out the information that was not requested by multiplying by `0` if the boolean is `false`, and only take the information we need, in case the boolean is `true`.

We then sum up all of these counts from the entire `order_items` table, as long as the entry has been created within the specified time interval, and we store the result in our `total` variable.

```
-- ------------------------------------------------------------------------------------------------------------------------
-- Procedure `get_products_sold` -> Retrieves the amount of products sold within the given range of year and month.
-- ------------------------------------------------------------------------------------------------------------------------
DELIMITER $$
DROP PROCEDURE IF EXISTS get_products_sold$$
CREATE PROCEDURE get_products_sold(IN `start` DATETIME, IN `end` DATETIME, IN `count_pictures` BOOLEAN, IN `count_frames` BOOLEAN, OUT `total` INT)
BEGIN
    SELECT
        SUM(
            IFNULL((SELECT SUM((
                IFNULL((`picture_data_id` / `picture_data_id` * `amount`),0)
                    * `count_pictures` +
                IFNULL((`frame_id` / `frame_id` * `amount`),0)
                    * `count_frames`))
            FROM `order_items` WHERE `order_items`.`order_id` = `orders`.`order_id`),0))
        INTO `total`
    FROM `orders`
    WHERE YEAR(`created_at`) >= YEAR(`start`) AND YEAR(`created_at`) <= YEAR(`end`) AND
          MONTH(`created_at`) >= MONTH(`start`) AND MONTH(`created_at`) <= MONTH(`end`);
END $$
DELIMITER ;
```

get_products_sold

One of our most complex procedures is `get_order_item_price`, which is used to calculate the full price of an order item. It is used by a trigger before the creation of an `order_item` entry to automatically its price when it is added.

This idea came with quite a few complications as we could not just pass one parameter with the id of the item we want to calculate the price of, because at the time, that item did not yet exist.
After a little bit of troubleshooting, we came up with the following solution, where we separately provide the `id` of the order item, its `frame` id, `paper` type id, and `amount` of copies for that item, as well as an output parameter called `total`.

Using all of this information, we then run a few `SELECT` queries in a sequence to gather all of the prices we need to perform our calculation. We get the id of the `sizes` table entry for both the frame and the printing paper, from which we get the base price for each product. Then we get the multiplier values that both the frame and the paper will add to the equation, and we pass all of this data over to our `fn_calculate_order_item_price` function.

The result of this function call is then stored into the `total` variable and is ready to be used.

```
-- ------------------------------------------------------------------------------------------------------------------
-- Procedure `get_order_item_price` -> Calculates the order item full price
--                     *Needs Refactoring*
-- ------------------------------------------------------------------------------------------------------------------
DELIMITER $$
DROP PROCEDURE IF EXISTS get_order_item_price$$
CREATE PROCEDURE get_order_item_price(IN `id` INT, IN `frame` INT, IN `paper` INT, IN `amount` DECIMAL(15, 2), OUT `total` DECIMAL(15, 2))
BEGIN
    -- Getting all the needed ids.
    SELECT `size_id` INTO @frame_size_id FROM `frames` WHERE `frames`.`frame_id` = `frame`;
    SELECT `size_id` INTO @paper_size_id FROM `paper_types` WHERE `paper_types`.`paper_type_id` = `paper`;

    -- Getting all the needed data.
    -- -> Frame data
    SELECT `price` INTO @frame_size_price FROM `sizes` WHERE `size_id` = @frame_size_id;
    SELECT `multiplier` INTO @frame_multiplier FROM `frames` WHERE `frames`.`frame_id` = `frame`;
    -- -> Paper data
    SELECT `price` INTO @paper_size_price FROM `sizes` WHERE `size_id` = @paper_size_id;
    SELECT `multiplier` INTO @paper_multiplier FROM `paper_types` WHERE `paper_types`.`paper_type_id` = `paper`;

    SELECT fn_calculate_order_item_price
    (
        @frame_size_price,
        @frame_multiplier,
        @paper_size_price,
        @paper_multiplier,
        `amount`
    )
    INTO `total`;
END $$
DELIMITER ;
```

get_order_item_price

Views

MySQL allows us to create objects called Views. These are simply queries that are given a name and stored on the database. We can use these to get repeated information with a simple command. One big advantage of these views is that they behave like real tables in many ways and can this be queried as though they were tables.

A consequence of using a relational database, is that the data is not all stored in one table that is easy to access, but rather, it is divided into small tables all of which have a single responsibility area. An example of this is the `frames` table, it contains information about itself and the id of a size, the`sizes` table that describes the size of it. This means that to get human readable information we need to get information from two different tables.

We have two views to resolve this issue in two concrete problem areas.

The first is a very simple view, that inner joins the `contact_info` table on `users` with the user id and displays it all as one single view. This table is useful to quickly see all the customer information at a glance and will likely be used in the front end to gather the user's information for display purposes. It is also in use as part of another view we have made.

```
-- -----------------------------------------------------------------------
-- View `customer_info`
-- -----------------------------------------------------------------------
DROP VIEW IF EXISTS `customer_info`;

CREATE VIEW `customer_info`
AS SELECT
    `users`.`user_id` AS `customer_id`,
    `users`.`first_name`,
    `users`.`last_name`,
    `contact_info`.`phone_number`,
    `contact_info`.`country_code`,
    `contact_info`.`city`,
    `contact_info`.`postal_code`,
    `contact_info`.`address_one`,
    `contact_info`.`address_two`
FROM `users` INNER JOIN `contact_info` ON `users`.`contact_info_id`=`contact_info`.`contact_info_id`;
```

customer_info view

Our `invoice` table only contains its own id, the id of an order and a date. However, this table needs to give us a lot of data that is stored in other tables. Instead of duplicating all that data here, we decided to create a view for it.

This view joins the orders table with invoice, contact info, the customer info view and discount codes in order to get all the information that we would need to generate and print an invoice for a customer or for accounting purposes. It also includes a new data set called `final_price` that is not otherwise stored, as it is just aggregate data of price and price saved.

The large benefit of having this stored as a view is that we do not need to store this as a .sql file that we give to each developer that will interact with our database, and we do not need to figure out how to mix all this information again in the future. It lives in our database with the name `invoice_data` and when ever we need an invoice, we can simply run `SELECT * FROM invoice_data WHERE ID = $id`. It makes it much easier to retrieve the data needed to generate invoices.

```sql
DROP VIEW IF EXISTS `invoice_data`;

CREATE VIEW `invoice_data` AS
    SELECT
        `invoice`.`invoice_id`,
        `invoice`.`created_at`,
        `orders`.`order_id`,
        `orders`.`order_price`,
        `orders`.`total_price_saved`,
        `fn_calculate_total`(`orders`.`order_price`, `orders`.`total_price_saved`) AS `final_price`,
        `discount_codes`.`name` AS `discount_code`,
        `customer_info`.`first_name`,
     `customer_info`.`last_name`,
     `customer_info`.`phone_number`,
     `customer_info`.`country_code`,
     `customer_info`.`city`,
     `customer_info`.`postal_code`,
     `customer_info`.`address_one`,
     `customer_info`.`address_two`,
        `contact_info`.`phone_number` AS `billing_phone_number`,
        `contact_info`.`country_code` AS `billing_country_code`,
        `contact_info`.`city` AS `billing_city`,
        `contact_info`.`postal_code` AS `billing_postal_code`,
        `contact_info`.`address_one` AS `billing_address_one`,
        `contact_info`.`address_two` AS `billing_address_two`
    FROM `orders`
        INNER JOIN `discount_codes` ON `orders`.`discount_code_id` = `discount_codes`.`discount_code_id`
        INNER JOIN `customer_info` ON `orders`.`user_id` = `customer_info`.`customer_id`
        INNER JOIN `contact_info` ON `orders`.`billing_contact_info_id` = `contact_info`.`contact_info_id`
        INNER JOIN `invoice` on `orders`.`order_id` = `invoice`.`order_id`;
```

invoice_data view

Triggers

Through the use of triggers, MySQL allows us to automatically run entire blocks of SQL statements when a certain action occurs.
We can create triggers that execute on `INSERT`, `UPDATE` or `DELETE` either before or after the action has completed.

We currently use these triggers to enforce a set of rules on our `orders` table.

One such rule is that we want to ensure that every order entry gets created with no price and no discounted value, so we created the `reset_order_price` trigger to make sure that before a new entry gets inserted, the `order_price`, as well as the `total_price_saved` are reset to zero, in case they weren't already that.

```
-- -----------------------------------------------------------------------------------------------------------------------------------
-- Trigger `reset_order_price` -> Resets the price and money saved of an order in case they have been created with a pre-existing value.
-- -----------------------------------------------------------------------------------------------------------------------------------
DELIMITER $$
DROP TRIGGER IF EXISTS reset_order_price$$
CREATE TRIGGER reset_order_price BEFORE INSERT ON `orders`
    FOR EACH ROW
        BEGIN
            IF NEW.`order_price` != 0 THEN
                SET NEW.`order_price` = 0;
            END IF;
            IF NEW.`total_price_saved` != 0 THEN
                SET NEW.`total_price_saved` = 0;
            END IF;
        END; $$
DELIMITER ;
```

reset_order_price

Another behaviour that we use triggers for is to automatically update the price of a certain order when an order item for it gets inserted.
For that purpose, we created the `calculate_order_price` trigger, which executes before the insertion of an entry in the `order_items` table.

We use the `get_order_item_price` procedure to calculate the item's price, make sure to assign this value to the `order_item_price` attribute to initialise it properly, and then use it once again to update the order that the item corresponds to by adding it to the already existing `order_price`.

Events

Much like the Triggers, MySQL Events allow us to automate the execution of certain blocks of SQL statements. Events, however, run not on the completion of some actions, but on specific schedules that we set for them.

With this functionality in mind, we thought it might prove useful to create an event that would generate a monthly report for our company.

First, we created a new table that would contain the initial structure of our report. It holds information about our revenue for that month, as well as how many products we have sold.

```
-- -----------------------------------------------------------------------------------------------------
-- Table `monthly_reports`
-- -----------------------------------------------------------------------------------------------------
DROP TABLE IF EXISTS `monthly_reports` ;

CREATE TABLE IF NOT EXISTS `monthly_reports` (
  `monthly_report_id` INT NOT NULL AUTO_INCREMENT,
  `created_at` DATETIME NOT NULL DEFAULT NOW(),
  `frames_sold` INT,
  `pictures_sold` INT,
  `total_products_sold` INT,
  `revenue` DECIMAL(15,2) NOT NULL,
  PRIMARY KEY (`monthly_report_id`))
ENGINE = InnoDB;
```

monthly_reports

Then, we created an event, `create_monthly_report`, set to run on a monthly schedule. It uses our `get_revenue` and `get_products_sold` to gather all the data we need for the previous month, then creates a new entry in our newly created table with all of the information we need.

```
-- -----------------------------------------------------------------------------------------------------
-- Event `create_monthly_report`
-- -----------------------------------------------------------------------------------------------------
DROP EVENT IF EXISTS create_monthly_report;

DELIMITER $$
CREATE EVENT create_monthly_report
    ON SCHEDULE
        EVERY 1 MONTH
    COMMENT 'Creates a new report every month.'
    DO
        BEGIN
            -- Revenue
            CALL get_revenue(NOW() - INTERVAL 1 MONTH, NOW(), @monthly_revenue);
            -- Pictures & Frames
            CALL get_products_sold(NOW() - INTERVAL 1 MONTH, NOW(), true, true, @products_sold);
            -- No Pictures & Frames
            CALL get_products_sold(NOW() - INTERVAL 1 MONTH, NOW(), false, true, @frames_sold);
            -- Pictures & No Frames
            CALL get_products_sold(NOW() - INTERVAL 1 MONTH, NOW(), true, false, @pictures_sold);
            INSERT INTO `monthly_reports` (`revenue`, `total_products_sold`, `frames_sold`, `pictures_sold`) VALUES (
                @monthly_revenue, @products_sold, @frames_sold, @pictures_sold
            );
        END $$
DELIMITER ;
```

create_monthly_report

Transactions - not now

Structure

Implementation

Auditing - not now

Structure

Triggers

Security - not now

Users & privileges

SQL Injection

What it is

How to mitigate it

CRUD application for RDBMS - not now

REST API

Service Layer

Security

Log in

Transactions

Etc. (???)