Design Document

A software toolbox for small retail shops

Version 0.1

Client: Han Limburg - Dorpswinkel Sauwerd Supermarket

Dan Plămădeală, S3436624
Abel Nissen, S3724786
Ruben Biskupec, S4235762
Florian de Jager, S3775038
Arjan Dekker, S3726169

university of groningen
Faculty of Science and Engineering

Lecturer: Mohamed Soliman
Teaching Assistant: Hichem Bouakaz
Last Updated: Friday 3rd April, 2020

Contents

Introduction

The well-known big supermarket chains in The Netherlands have large budgets available for automation of their business operations. The complete chain, from ordering (the right amount and type of) products, delivery of stock, handling losses due to overtime goods and returns, and of course the actual sales are handled by big centralized ERP systems giving very precise management information to the retailers.

"A software toolbox for small retail shops" aims to provide many of these services to the Dorpswinkel in Sauwerd (a village 10km north of Groningen). **O**ur product is a database system specifically built to fit the environment, and needs, of the Dorpswinkel in Sauwerd. It provides a better basis to retrieve management information, perform standard queries and access hard data on sales to be able to optimize the shop.

The main interface to this database will be web based, so as to ensure compatibility across all platforms; windows, macOS, Linux but also android and iOS. This also provides managers of the shop access to the data while they are off-premise, to be able to crunch numbers at every moment of the day!
Additionally, "A software toolbox for small retail shops" adds loyalty card support for the Dorpswinkel. Transaction information will be stored in the database to allow personalized bonus discounts.

Database design

Since it is important to keep a database small in size, only the necessary information should be stored. This will not only reduce the size of the database, but also improves the reading speed.

How the products are stored

There are lots of possibilities when it comes to storing the products in the database. We think the most suitable design is one where every individual item (e.g. banana 1, banana 2 etc) is stored in the database. We prefer this over a design where you store every type of product (e.g. bananas), because then information can be coupled to every individual item such as buying price and selling price.

Table design

For every individual product, some information needs to be stored. The following information is being stored:

- Journal record
- Cashier
- Record Type
- PLU
- Product name
- Amount
- Selling price
- Discount
- Payment method (cash/vic107payment-PINNEN)
- Ticketdata
- Total
- MixMatchDiscount
- Leesmethode

- date and time
- Buying price

Product	**Comment**
PLU	
Buying Price	
Selling Price	
Discount	
Name	
Transaction ID	

Transaction	**Comment**
Transaction ID	
DateTime	
Receipt Number	
Total Amount	
Payment Method	

Card	**Comment**
Transaction ID	
Serial Number	

Cash	**Comment**
Transaction ID	
Change	

* Primary keys are underlined, foreign keys are in italic.

Products Sold				
PLU	Price	Name	Stock	Transaction ID (FK)

Transaction				
Transaction ID	Receipt Number	Transaction Type	Discount	Total Amount
Payment Method	DateTime			

Cash			
Transaction ID	Receipt Number	Total Amount	Change

Card			
Transaction ID	Receipt Number	Serial Number	Total Amount

Loyalty Card			
Barcode	Transaction ID (FK)	Customer Name	Customer Email

API design

The default representation of resources is JSON.

Used response codes:

- **200** - OK; this means that the request was successfully fulfilled.
- **400** -Bad Request; this means that there is a mistake in the syntax of the request.
- **404** -Not Found; this means that the given resource was not found.

Flask will be used to implement the endpoints. To create consistent endpoints, the following format should be used:

- *route*/parameter1=<value>/parameter2=<value>
  - Note that the endpoint can be extended for more parameters in the same way. Also, you can use one parameter by removing the second parameter.
  - The parameters should get meaningful names. For example, if you want to retrieve a product with the name "banana", the API endpoint should be something like: *route*/product/name=<value>

3.1 Requests and Responses

1. **Objective**:
   Return all transactions on a given day.
   The user is required to provide the date in UNIX timestamp.

   Endpoint:
   GET /sales/day=<date>

   Query Parameters:

Query parameter	Value
day	The date of the desired day in UNIX

   HTTP status codes:

   (a) 200 (SUCCESS)
   (b) 400 (BAD REQUEST) - Incorrect UNIX timestamp.

(c) 404 (NOT FOUND) - If no record of the specified day exist.

Response:

```
{
    "count": integer,
    "results": [
        {
            "products": array
        },
        ...
    ]
}
```

2. **Objective**:
Return all transactions from a starting datetime until an ending datetime. The user is required to provide the dates in UNIX timestamps.

Endpoint:
GET /sales/start=<datetime1>/end=<datetime2>

Query Parameters:

Query parameter	Value
datetime1	The desired starting datetime in UNIX
datetime2	The desired ending datetime in UNIX

HTTP status codes:

(a) 200 (SUCCESS)

(b) 400 (BAD REQUEST) - Incorrect UNIX timestamp.

(c) 404 (NOT FOUND) - If no record of the specified day exist.

Response:

```
{
    "products": [
        {
            "Name": string
            "Amount": integer
```

```
                },
            ]
        }
```

3. **Objective**:
   Return average amount of sales per given time-interval
   The user is required to provide: time-interval, starting datetime, ending datetime.

   Endpoint:
   GET /sales/average/interval=<interval>/start=<datetime1>/end=<datetime2>

   Query Parameters:

Query parameter	Value
interval	The desired time-interval
datetime1	The desired starting datetime in UNIX
datetime2	The desired ending datetime in UNIX

   HTTP status codes: **Response**:

```
        {
            "intervals": [
                {
                    "start": string
                    "end": string
                    "amount": integer
                },
            ]
        }
```

4. **Objective**:
   Return information about a product
   The user is required to provide: PLU/name.

   Endpoint:
   GET /product/plu=<plu>

   Query Parameters:

Query parameter	Value
id	Either a PLU or name

HTTP status codes: **Response**:

```
{
    "PLU": string
    "name": string
    "buyingPrice": integer
    "sellingPrice": integer
    "Discount": integer
}
```

5. **Objective**:
   Update a product's buying price.
   The user is required to provide the product's PLU and new buying price.

   Endpoint:
   SET /updateproductbprice/plu=<plu>/newprice=<newprice>

   Query Parameters:

Query parameter	Value
id	PLU of the product
newprice	new buying_price of the product

   HTTP status codes:

   (a) 200 (SUCCESS)
   (b) 400 (BAD REQUEST) - Incorrect PLU.
   (c) 404 (NOT FOUND) - If no record of product exist.

6. **Objective**:
   Update a product's selling price.
   The user is required to provide the product's PLU and new selling price.

   Endpoint:
   SET /updateproductsprice/plu=<plu>/newprice=<newprice>

   Query Parameters:

Query parameter	Value
id	PLU of the product
newprice	new selling_price of the product

HTTP status codes:

(a) 200 (SUCCESS)

(b) 400 (BAD REQUEST) - Incorrect PLU.

(c) 404 (NOT FOUND) - If no record of product exist.

Front-end design

Barcode Scanner design

For the barcode scanner there are two possbile designs.

5.1 Server side

The first design is having a barcode scanner on the server. What this means is that footage from the camera of the user is captured and sent to the server. The server is then using this footage to detect the barcode number in the images.

5.1.1 Pros

The benefits of having a barcode scanner on the server-side are:

- Less computing for the client, which is nice if the client does not have powerful hardware

5.1.2 Cons

The cons of having a barcode scanner on the server-side are:

- More computing on the server side
  - In case many clients are connected, this can get heavy for the server if it is not powerful
- More data usage, due to the fact that the images have to be sent to the server
- More latency before a barcode is scanned

5.2 Client side

The second design is having a barcode scanner on the client side. In that case the client is using its own camera and an API to detect a barcode. When it detected a barcode, the barcode number will be sent to the server.

5.2.1 Pros

The benefits of having a barcode scanner on the client side are:

- Less computing on the server side
- Less data usage, due to the fact that images are not being sent to the server
- Less latency before a barcode is scanned

5.2.2 Cons

The cons of having a barcode scanner on the client side are:

- More computing on the client side

5.3 Our decision

Since we do not want to have a big strain on the server, we chose to do the scanning on the client side. Also the decreased data usage and low latency are nice to haves and will improve the user experience.

5.4 Implementation

To make the barcode scanning possible on the client side, we are going to use **JavaScript**. This is a language that is very well supported in browsers. The API that will be used is **QuaggaJS**. That API is using *getUserMedia*, which makes it possible to access the camera of the client. Since this is only possible when our website is using https, we are actually obligated to use it, which is a good thing.

When **QuaggaJS** has detected the barcode number, it will send the number to the server using **Ajax** (Asynchronous JavaScript and XML). The server can then look in the database and retrieve the product information belonging to that barcode number.

Technology Stack

Languages:

- Python 3

Libraries:

- -

Extensions:

- -

Building tools:

- MySQL
- Flask

Testing:

- SonarQube

From our first meeting with the client, he stated very clearly that the stack is fully at our discretion. For the database we use MySQL and for testing we use SonarQube at the moment, as required by the coordinators of the course.

Team organisation

For the first sprint, there were no real clear teams. The main purpose was for everybody to understand the data that was given to us and to understand the language that we need to use. Eventually, the group was split into three groups: one in data review, one for writing this document and the other group doing code and code review.

The aim of the second sprint was to start working with the APIs, which we got from our client, working more on the mappers and creating a front-end. This is done by splitting the work up to three group, in which one worked on the mappers, one on the front-end and the other on the APIs.

After the third sprint we decided to have a weekly meeting, in which we will be working on the project all together on the same time. This way, when a part of the code written is not understood, we can explain it directly and or change it.

Change log

Who	When	Which section	What	Time
Abel	13.03.20	Introduction	Added the introduction	1h
Dan	13.03.20	Team organisation, Technology Stack, API Design	Added the Team organisation, API Design and Technology Stack parts	1h
Abel & Ruben	20.03.20	Database Design	Added the database tables	1h