Design Document

Version 1.1

Clients: Han Limburg, Rix Groenboom
De dorpswinkel Sauwerd

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,
Andrea Capiluppi
Teaching Assistant: Hichem Bouakaz
Last Updated: Saturday 13th June, 2020

Contents

Introduction

This document will provide more insight into the design and architecture of the various components that represent our product (web-app, server, database, etc.) and details about the reason these decisions were made and the systems were designed this way. First, we will describe the design of our parser, then the design of the database, followed by the design of the backend and then the design of the front-end and the web-app. Lastly, we will describe the overview of our technology stack and give some information about the organizational matters of our team.

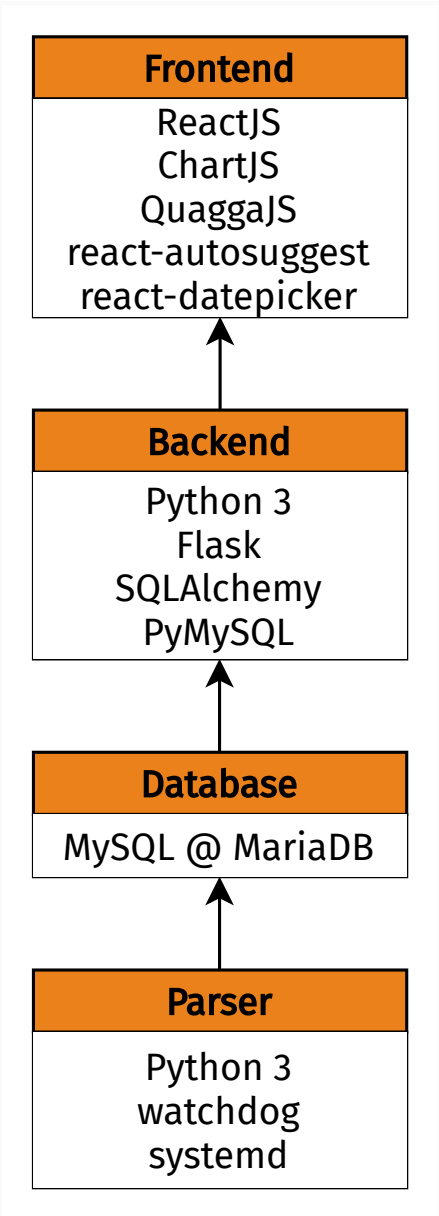
The well-known big supermarket chains in The Netherlands have large budgets available for automation of their business operations. The complete chain, from ordering (the right amount and type of) products, delivery of stock, handling losses due to overtime goods and returns, and of course the actual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

Retaily aims to provide many of these services to De dorpswinkel in Sauwerd (a village 10km north of Groningen). Our product is a system specifically built to fit the environment, and needs, of De 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, to ensure compatibility across all platforms; Windows, macOS, Linux but more importantly 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!

Design overview

To make the system more understandable, an overview of it is created below. It contains four tables: frontend, backend, database and parser. Each table contains the technologies used for it. Later in this paper, the design of each part can be found and also the design decisions for the given part.



Parser design

3.1 Parser overview

We have two parsers. One for parsing the journal records that are exported by the machines at the point of sale. Another one for parsing the invoices containing the buying prices of the products sold in the shop.

3.2 Journal record parser

3.2.1 Introduction

Our parser is pretty straightforward, as it was done in an empirical way, we looked at the structure of the journal record files and simply defined the needed functions and parsing helping functions that were parsing the corresponding parts of the journal records file. It is written in vanilla Python without any libraries. It runs as a service using the `systemd` system and service manager. Follows a given location on the system and whenever a new `.ZIP` file appears from the point of sale machine, it extracts it, which yields a `.txt` file that is afterwards opened, split into separate journal records. After this, it finally parses the journal records and sends all the required data to the database. Current features are as follows:

- Follow for the appearance of new dumps of the database.
- Extract the archive.
- Parse the whole document.
- Update the selling price of the products in the database.
- Compute the total amount of a journal record correctly.
- Store information about each transaction in the database.
- Store information about each sold product in the database.
- Store correct information about each available product for sale in the database.

3.2.2 Database connection

For every journal record file, a instance of the `DataSender` class is created, which automatically connects to the database using the right credentials. This instance is used by the parsing system to send and update the needed data to the database.

3.2.3 Parsing baseline

```
===
Journal Record: #0/8 / 2017-08-31 15:02:53
Cashier       : #1 / VERKOPER 1
Record Type   : Sale
---
********************
*** Aborted Sale ***
********************
---
  * PLU # 8712000033040
    HEIN.PILS 24X30 CL.
    1,00 x 9,00
    Discount: 0,00 Total:9,00
  * PLU # 100019
    STATIEGELD 3,90
    1,00 x 3,90
    Discount: 0,00 Total:3,90
===
```

This is what a simple journal record looks like, we have the introduction and the content itself. From the introduction we parse the following:

- `journal_record_no`:String - the number of the record
- `journal_record_date`:Date - the date when the record was recorded
- `journal_record_cashier`:String - the identifier of the cashier
- `journal_record_type`:String - the type of the sale, available variants are: [Sale, NoSale, NeutralList, ReportRecord]
- `journal_record_aborted`:Boolean - whether the record was aborted

A fragment of the parsed data from this as follows:

```
{
    'journal_record_aborted': True,
    'journal_record_cashier': '#1 / VERKOPER 1',
    'journal_record_date': '2017-08-31 15:02:53',
    'journal_record_no': '#0/8',
    ...,
    'journal_record_type': 'Sale'
}
```

Journal body

Next we accentuate our attention on a specific part of the journal record, the content that is after the introduction, the body of the journal which may contain various components.

```
* PLU # 8712000033040
  HEIN.PILS 24X30 CL.
  1,00 x 9,00
  Discount: 0,00 Total:9,00
* PLU # 100019
  STATIEGELD 3,90
  1,00 x 3,90
  Discount: 0,00 Total:3,90
```

Here we can have a few types of components:

- The products themselves
- `CashWithdrawal`
- `MixAndMatchDiscountItem`
- `CardPayment`
- `CashPayment`
- `SubtotalSaleItem`
- `PaymentRoundingCompensation`

We parse each one of them separately.

Normal product

```
* PLU # 404
  KOMKOMMER
  1,00 x 0,59
  Discount: 0,10 Total:0,50
```

From here we extract the following information:

- `product_plu`:Integer - supposedly the barcode of the product
- `product_canceled`:Boolean - whether the product was canceled
- `product_name`:String - the name of the product
- `product_amount`:Float - how many pieces
- `product_price`:Float - price of a piece
- `product_discount`:Float - whether this product has a discount for its price
- `product_total`:Float - supposedly `product_price` * `product_amount` - `product_discount`

For this specific product in the previous fragment we would have the following parsed result:

```
[    ...,
     {
          'product_amount': '1,00 ',
          'product_canceled': False,
          'product_discount': '0,10',
          'product_name': 'KOMKOMMER',
          'product_plu': '404',
          'product_price': ' 0,59',
          'product_total': '0,50'
     },
     ...
]
```

CashWithdrawal

For this part:

```
* CashWithdrawal (CANCELED)
  WithdrawalAmount: 0
  DrawerAmount: 0
  GrossAmount: 0
  DrawerId: drawers/default
  DrawerNumber: 1
  Amount: 0
  IsCancelable: True
```

we have the following output:

```
[    ...,
     {
          'cw_amount': '0',
          'cw_cancelable': 'True',
          'cw_canceled': True,
          'cw_drawer_amount': '0',
          'cw_drawer_id': 'drawers/default',
          'cw_drawer_number': '1',
          'cw_gross_amount': '0',
          'cw_withdrawal_amount': '0'
     },
     ...
]
```

MixAndMatchDiscountItem

For this part:

```
* MixAndMatchDiscountItem
  DiscountText: komkommer + sla kort
  DiscountAmount: -0,59
  DiscountTypeName: MixMatchDiscount
  MixMatchNumber: 2117014
  DiscountNumber: -2
  IsCancelable: True
```

we have the following output:

```
[    ...,
     {
          'mm_cancelable': 'True',
          'mm_discount_amount': '-0,59',
```

```
        'mm_discount_number': '-2',
        'mm_discount_text': 'komkommer + sla kort',
        'mm_discount_type_name': 'MixMatchDiscount',
        'mm_number': '2117014'
    },
    ...
]
```

CardPayment

For this part:

```
  * Vic107Payment
    Text: PINNEN
    TicketData: POI: 50244642
KLANTTICKET
--------------------------------
Terminal:                  M090LY
Merchant:                  702888
Periode:                     7244
Transactie:              01000084


MAESTRO
(A0000000043060)
MAESTRO
Kaart:       xxxxxxxxxxxxxx2148
Kaartserienummer:               7

BETALING
Datum:          01/09/2017 13:20
Autorisatiecode:          C0H02K

Totaal:                57,56 EUR

Leesmethode: Chip
PIN OK


    CardTypeId: 4105
    CardTypeText: MAESTRO
```

```
ReceiptNumber:
DrawerAmount: 57,56
Number: 3
DrawerId: drawers/default
DrawerNumber: 3
Amount: 57,56
IsCancelable: True
```

we have the following output:

```
[    ...,
     {
          'cp_authorisation_code': 'C0H02K',
          'cp_cancelable': 'True',
          'cp_card': '7',
          'cp_card_serial_number': '7',
          'cp_card_type': 'Chip',
          'cp_card_type_id': '4105',
          'cp_card_type_text': 'MAESTRO',
          'cp_date': '01/09/2017 13:20',
          'cp_drawer_amount': '57,56',
          'cp_drawer_id': 'drawers/default',
          'cp_merchant': '702888',
          'cp_period': '7244',
          'cp_poi': '50244642',
          'cp_terminal': 'M090LY',
          'cp_total': 'C0H02K',
          'cp_transaction': '01000084'
     },
     ...
]
```

3.2.4 CashPayment

For this part:

```
* CashPayment
  Text: KONTANT
  IsChange: False
  Amount: 10,0
  DrawerAmount: 10,0
```

```
    Number: 1
    DrawerId: drawers/default
    DrawerNumber: 1
    IsCancelable: True
  * CashPayment
    Text: KONTANT
    IsChange: True
    Amount: -0,7
    DrawerAmount: -0,7
    Number: 1
    DrawerId: drawers/default
    DrawerNumber: 1
    IsCancelable: True
```

we have the following output:

```
[    ...,
     {
          'cp_amount': '10,0',
          'cp_cancelable': 'True',
          'cp_drawer_amount': '10,0',
          'cp_drawer_id': 'drawers/default',
          'cp_drawer_number': '1',
          'cp_is_change': 'False',
          'cp_number': '1',
          'cp_text': 'KONTANT'
     },
     {
          'cp_amount': '-0,7',
          'cp_cancelable': 'True',
          'cp_drawer_amount': '-0,7',
          'cp_drawer_id': 'drawers/default',
          'cp_drawer_number': '1',
          'cp_is_change': 'True',
          'cp_number': '1',
          'cp_text': 'KONTANT'
     },
     ...
]
```

Note the differing `cp_is_change` values.

3.2.5 SubtotalSaleItem

For this part:

```
* SubtotalSaleItem
  Value: 9,30
  UseSwissRounding: False
  IsCancelable: True
```

we have the following output:

```
[    ...,
     {
          'ssi_cancelable': 'True',
          'ssi_use_swiss_rounding': 'False',
          'ssi_value': '9,30'
     },
     ...
]
```

3.2.6 PaymentRoundingCompensation

For this part:

```
* PaymentRoundingCompensation
  Amount: -0,01
  PaymentNumber: 1
  IsCancelable: True
```

we have the following output:

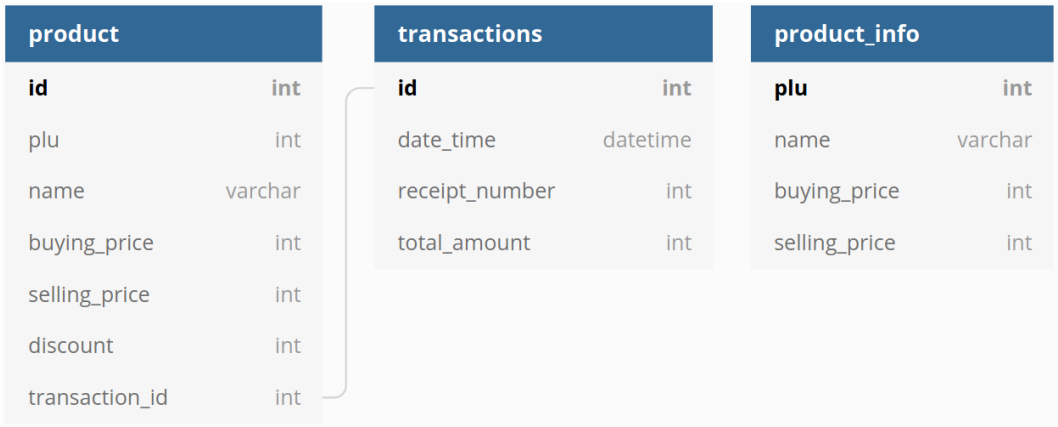
```
[    ...,
     {
          'prc_amount': '-0,01',
          'prc_cancelable': 'True',
          'prc_payment_no': '1'
     },
     ...
]
```

Database design

It is important to keep a database as small as possible in order to preserve space and keep it fast. Therefore the database needs to store the data in an efficient manner. For this application, a relational database is used to accomplish that (later on it will be explained why that is the case). Hence we decided to choose and SQL based database, in our case the MariaDB implementation. It is one of the most used relational databases and has excellent integration with Flask-based web-apps.

4.1 Overview and explanation

An overview of the database can be found below.



4.1.1 product

The first table is the product table. This table contains all individual products that are sold. The reason this table needs to exist is quite simple. The buying_price (the price that the shop pays for a product) and selling_price (the price that a customer pays for a product) of a product changes over time. Hence, these values are coupled to a specific product. Every product has a PLU which is also known as a Price Look-Up code (EAN-13), a name (e.g. banana), a buying_price, selling_price and an id. The id is used as a primary key in the table. PLU could not be used as a primary key since e.g. every banana has the same PLU.

4.1.2 transaction

When a customer buys a set of products, it is called a transaction. Therefore it contains a data_time (the time a set of products is bought), receipt_number (generated by the cash register) and a total_amount (total price of the products combined). The id of a transaction is used to uniquely identify a transaction.

4.1.3 product_info

The product_info table contains the current information about a product. In contrast to the first table, here the PLU can be used as a primary key, since there is only one set of up to date information about a product. This table includes the PLU, the name, the buying_price and the selling_price of a product.

4.1.4 Decisions

To preserve space in the database, the products and transactions are stored separately. That is because almost all transactions have multiple products. Thus storing transaction information for every product leads to duplicate information, which should be avoided.

Backend design

The backend of the application is written in Python using the Flask web framework since it is one with a very comfortable learning curve and allows for a fast development and prototyping of RESTful APIs.

5.1 File structure

This section is about the file structure of the backend. It contains a main file called `app.py`. That file is responsible for starting the backend.

The file `app.py` needs configuration parameters in order to be able to interact with the database. These parameters are imported from the file `config.py`.

Apart from the configuration parameters, the app also needs to have information about the structure of the database in order to interact with it. This structure can be found in `models.py`. For every table in the database, it corresponding class that contains the information about the columns of that table. On top of that, every class has a function called `serialized`. That function is used to be able to send a query as a JSON.

The file `views.py` contains all the endpoint blueprints with their corresponding url prefix, e.g. `url_prefix="/example"`.

The endpoint blueprints mentioned before, can be found in the folder `blueprints` . Every blueprints has one or more endpoints. An endpoint example can be found below.

```
1 @blueprintName_bp.route("endpoint-URL")
2 def method():
3     result = interactWithDatabase()
4     return jsonify(result)
```

Furthermore, every endpoint blueprint can have functions to keep the code clean.

Since it is not guaranteed that every endpoint call is successful, errors need to be used to inform the user about this. These errors can be found in the file `error.py`. In case something goes wrong, an error message is sent to the frontend and displayed.

5.2 Endpoints

5.2.1 Response codes

Interaction with the backend is accomplished with HTTP requests. Currently, the only request method accepted is GET. That implies that the backend is solely used for gathering data, not modifying data. Since not every request is going to be successful, it is useful to provide the client with useful response codes. The backend is using the following response codes:

- **200** - OK; this means that the request was successfully fulfilled.
- **404** - Bad Request; this means that there was a mistake in the syntax of the request (e.g. a wrong URL) or the PLU/name of a product was not found.
- **500** - Server Error; this means that someting went wrong with the server e.g. due to a bad connection with the database.

5.2.2 API endpoints

The application uses Flask Blueprints. That is to keep the backend structured. For every subdomain (e.g. "API-URL/subdomain/") a Blueprint is used. That structure is also used in the following part.

Subdomain: inventaris

1. **Endpoint**: BACKEND_URL/inventaris/tabel
   Method: GET
   Parameters: -
   Objective: GET a list with all the unique product names in the ProductInfo table of the database
   HTTP status codes:
   - 200 (SUCCESS)
   - 500 (SERVER ERROR)

   Response:

```
[
    {
        "name": "STRING"
    }
]
```

Subdomain: koppelverkoop

1. **Endpoint**: BACKEND_URL/koppelverkoop/lijst/
   Method: GET
   Parameters:

   - plu: number (required if name is not used)
   - name: string (required if plu is not used)
   - start: date (required)
   - end: date (required)

   Objective: GET a list of the top 10 products that are sold together with the specified product (koppelverkoop is the Dutch translation) starting from and ending at a specified date
   HTTP status codes:

   - 200 (SUCCESS)
   - 404 (BAD REQUEST)
   - 500 (SERVER ERROR)

   Response:

```
[
    {
        "count": "",
        "name": "Geselecteerd Product: STRING"
    },
    {
        "count": "NUMBER",
        "name": "STRING"
    }
]
```

   Note: The first product in the list is the specified product in the request. After that comes the top 10 products that are sold together with that specified product.

Subdomain: product

1. **Endpoint**: BACKEND_URL/product/
   Method: GET
   Parameters:

   - plu: number (not required)
   - name: string (not required)

   Objective: If a plu or number is given, GET the tuple with that plu or name from the Product Info table in the database. Else GET all the tuples from the Product Info table in the database.
   HTTP status codes:

   - 200 (SUCCESS)
   - 404 (BAD REQUEST)
   - 500 (SERVER ERROR)

   Response if plu or name is specified:

```
{
    "buying_price": "NUMBER",
    "name": "STRING",
    "plu": "NUMBER",
    "selling_price": "NUMBER"
}
```

   Response if no plu or name is specified:

```
{
    "products": [
        {
            "buying_price": "NUMBER",
            "name": "STRING",
            "plu": "NUMBER",
            "selling_price": "NUMBER"
        }
    ]
}
```

Subdomain: verkoop

1. **Endpoint**: BACKEND_URL/verkoop/
   Method: GET
   Parameters:

   - plu: number (not required)
   - name: string (not required)
   - start: date (required)
   - end: date (required)
   - interval: string (required) ("half_an_hour", "hour", "day", "week", "month")
   - revenue: anything (not required) (the backend checks if this parameter was given or not)

   Objective: If a plu or name is specified, GET the sales of that specified product between a start date and end date with a given interval. If no plu or name is specified, GET the sales of all products combined from a start date and end date with a given interval. If the revenue parameter is given, the price of the sales are added up, yielding the revenue instead of the amount of sales.
   HTTP status codes:

   - 200 (SUCCESS)
   - 404 (BAD REQUEST)
   - 500 (SERVER ERROR)

   Response:

```
[
    {
        "t": "DATE_TIME",
        "y": "NUMBER"
    }
]
```

2. **Endpoint**: BACKEND_URL/verkoop/kort/
   Method: GET
   Parameters:

   - plu: number (required if name is not used)
   - name: string (required if plu is not used)

   Objective: Retrieve a summary of the sales of the specified products (sales last week, last month, last quarter, last year).
   HTTP status codes:

   - 200 (SUCCESS)
   - 404 (BAD REQUEST)
   - 500 (SERVER ERROR)

   Response:

```
{
    "sales_last_month": "NUMBER",
    "sales_last_quarter": "NUMBER",
    "sales_last_week": "NUMBER",
    "sales_last_year": "NUMBER"
}
```

5.3 Database connection

For the connection with the database, the Flask extensions Flask-SQLAlchemy is used. It is based on SQLAlchemy and it makes integration with Flask easier, as the name suggests. Flask-SQLAlchemy is widely used and has a lot of documentation.

5.3.1 Connecting with the database

In order to connect with the database, Flask-SQLAlchemy requires a database-uri. The database-uri is a concatination of the protocol, username, password, hostname, portnumber and charset of the database. The uri will be set as a configuration value of the flask app. When the value is set, a connection can be made with `db = SQLAlchemy(app)`.

5.3.2 Defining database tables

Flask-SQLAlchemy requires the database tables to be set in the app. This can be done with the following example code:

```
class Table(db.Model):
    column_name = db.Column(db.Type)
```

However, our application needs to return a query in JSON-format. Hence the serialization property should be defined:

```
    @property
    def serialized(self):
        return {
            "column_name": self.column_name,
        }
```

5.3.3 Interacting with the database

After the setup is completed successfully, the application can interact with the database. Since the application uses Flask-SQLAlchemy, plain SQL-statements are not needed. Flask-SQLAlchemy has its own statements. To demonstrate this, a few examples can be found below.

```
# Return all sales of a product between two datetimes
db.session.query(Product)
.join(Transaction)
.filter(
    (Product.plu == plu)
    & (Transaction.date_time >= dt1)
    & (transaction.date_time <= dt2)
)
```

```
# Return all items that are not sold (inventory)
Product.query.filter(Product.transaction_id == None)
.order_by(Product.name)
```

```
# Update selling price of a product with a certain plu
ProductInfo.query.filter(ProductInfo.plu == plu).update(
    {ProductInfo.selling_price: price}
)
```

```
# Only return the first product with a certain name
ProductInfo.query.filter(ProductInfo.name == name).first()
```

It is clear that Flask-SQLAlchemy can do advanced queries while keeping it very clear and understandable. Therefore it is very helpful for interacting with the database compared to regular SQL-statements.

Frontend design

Users are solely interacting with the frontend. Hence it is of great importance that the frontend is fast, responsive and has an intuitive UI and UX. That is easier to achieve by using a framework. For this project, we decided that React would be suitable. The main reasons for using React are that it is fast and it has a vast amount of users, thus also a lot of documentation and explanations.

6.1 API Connection

Connecting with the API is rather straightforward using React. React has a built-in function called `fetch()`. The documentation of React contains a template which can be found below:

```
fetch("https://api.example.com/items")
  .then(res => res.json())
  .then(
    (result) => {
      this.setState({
        isLoaded: true,
        items: result.items
      });
    },
    // Note: it's important to handle errors here
    // instead of a catch() block so that we don't swallow
    // exceptions from actual bugs in components.
    (error) => {
      this.setState({
        isLoaded: true,
        error
      });
    }
  )
```

The `result` variable contains the JSON information that is needed for the frontend. That information can be used to create Charts for example.

6.2 Displaying Data

6.2.1 Charts

As said in the user stories, the application should be able to display various statistics. Charts are therefore very useful in this case since they visualize the data for the user.

There are two popular libraries that we were considering when choosing the way to implement our charts. Those are D3.js and Chart.js. The former is more advanced than the latter. However, the features of D3.js is giving over Chart.js are not necessary for our application. Hence, we chose for simplicity Chart.js.

The integration of Chart.js in React is rather easy. There exists a React wrapper for Chart.js which is called `react-chartjs-2`, but the implementation without the wrapper is easy enough. As a result, we are solely dependent on Chart.js and we do not have the restrictions of `react-chartjs-2`.

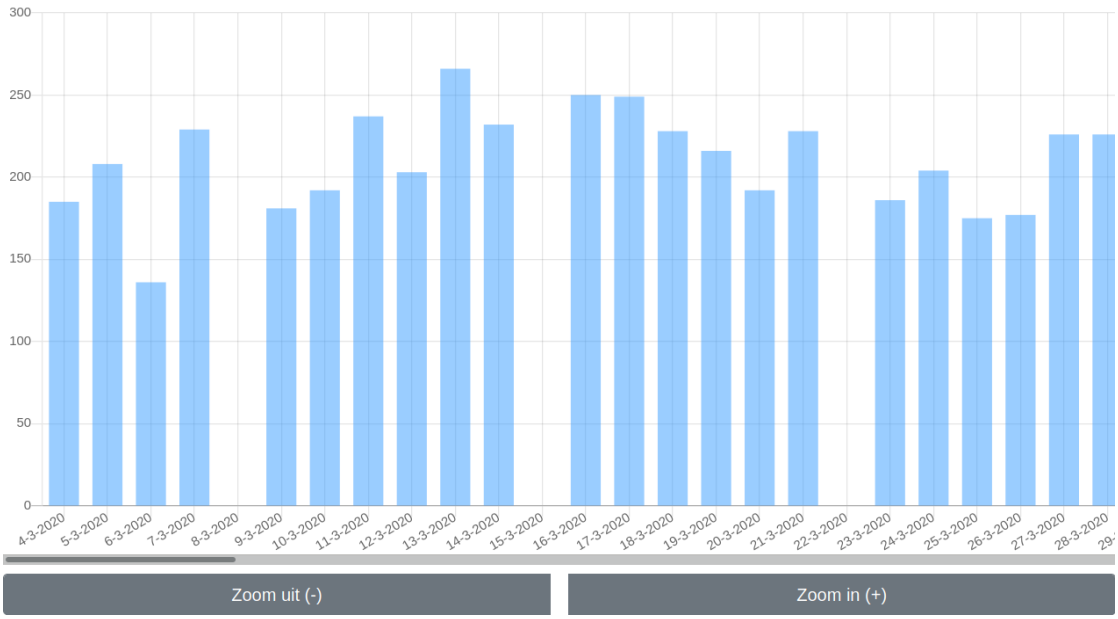


Figure 6.1: Chart of customers over time

Zooming

Per request of the client, we implemented a zoom feature to the chart. We put the button beneath the chart, so the user is easily able to reach those buttons on mobile, where the feature is most necessary due to the small screen width. Zoom in decreases the amount of data visible on the screen, whereas zooming out increases it. The fully zoomed out view provides a good overview of the trend, whereas fully zoomed in the details are better visible (such as specific dates).

Geselecteerd Product: Suikerbrood klein	
WENSKAARTEN	26
BROOD	22
CROISSANT ROOMB.	14
BROOD NS	13
KRENTENBROOD HALF GESN.	13
karnemelk	13
ZEEBONK GESN.	11
BANANEN FAIRTRADE 18KG	10
melk halfvol.	10
DE WIE-LEN	9

Figure 6.2: Example of a Koppelverkoop table

6.2.2 Tables

At two places tables are used in the web-app, at Koppelverkoop and Product Info. Tables are an clear and easy way display simple data. To make to the tables more readable by alternating the colors of the rows, we used the *table-striped* Bootstrap class.

6.3 Barcode Scanner

The application should have the ability to scan the barcode of a product. That is possible with two implementations. The first implementation sends camera footage to the backend. The backend processes the footage and checks for a barcode. The second implementation scans on the frontend. For this application we chose to do it on the frontend due to the following reasons:

- Less latency before a barcode is scanned
- Less data usage
- Less computing on the server-side

Since the frontend uses React, the barcode scanner will be a React Component. For the scanning, we are going to use the library QuaggaJS. It has support for scanning live camera footage, which is a really nice feature for the user. It also has good documentation and therefore it can be implemented rather easy.

When the scanner scans a barcode, the scanner is automatically closed to free up screen real estate. This way the user is immediately able to see the retrieved

result. Another reason is that we only want the scanner open to be open when needed, as it is quite a battery draining process. To complement this, the scanner also closes when the user starts editing the PLU.

Making the scanner work on IOS was a little tricky. When we had it ultimately working, the size and position of the scanner were messed up. Here none of the Bootstrap classes were able to fix the problem fully, so some custom css classes were necessary to fix the weird empty white space that kept appearing under scanner and fix the sizing on desktops. The additional white space came only after the scanner was initialized, so the spacing should be adjusted after that. This is the reason for the loaded state variable in *BarcodeScanner.jsx*, it toggles the css class *posFix*.

6.4 Bootstrap

To make the web-app responsive and prettier we use Bootstrap. Using it is as simple as adding an import in the css and using their predefined CSS class names. We opted to use a global import of Bootstrap (version 4.2.0) in App.css. This way our web-app did not need that much handwritten css classes and it saved us a lot of time. A good example are the datepickers: they display side by side on desktop, but the second one is under the first on mobile.

By giving a html element a Bootstrap class name, it will automatically be 'more responsive' than the bare html, but bootstrap provides additional tools to fine tune the behaviour. One of tools are the Bootstrap spacers. This tool allows for setting specific margin and padding for different screen sizes. For example: adding "my-md-4 my-3" sets the margin y (top and bottom) to 4 on medium sized screens or larger, whilst setting it to 3 on smaller displays.

6.5 Pages

All the pages on the web-app have the same structure, so we use *BluePrintPage.jsx* and pass the content as prop to render the page. The blueprint contains the Navigation Bar and sets up the container card where the actual content of the page is displayed. To set up the different page routes for the frontend, we use react-router. The different routes are visible in *index.js*. React-router renders React components and sets a specific address to that component, so that when the user goes to that address that component is loaded.

6.6 Testing vs Production

6.6.1 Context

React context is used for a single Boolean value switch. It is setup in the *Absolute.jsx* file and when set to true in *index.js*, all the API fetches done go trough 'https://retaily.site:7000', which corresponds to the API in the shop. When set to false, however, the user able to use their own API running on port 5000 of their computer. This mode is used to test changes made to the API, so that the people at the shop are still able to use the system while it is being changed.

6.7 React DatePicker

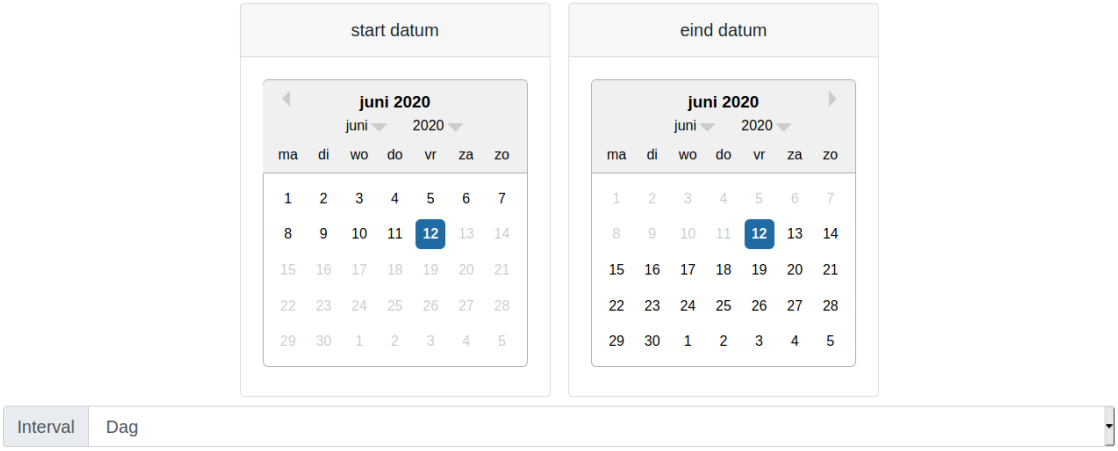


Figure 6.3: Datepickers with interval select

To make the datepickers look a lot nicer and work a lot more intuitive, we opted to use react-datepicker by hackerone. The datepicker has some handy features like dutch language support, month and year dropdowns and setting an minimum and maximum selectable date. The datepicker we use in the web-app always consists of two datepickers, one for the starting date and one for the end date. The implementation we used can be found in *IntervalDatePicker.jsx*.

6.7.1 interval

Some of our DatePickers also come with an interval select, which enables the user to select from a list of options. The options are: half an hour, hour, day, week

and month. If the user selects for example the month, the days in the datepicker will all be greyed out except for the first day of every month. This way the user is only able to select specific months. Moreover, when the user changes the interval to week or month, the datepicker of the start date will move to the first of the selected interval (week or month), whereas the datepicker of the end date will move to the beginning of the next interval (week or month). We chose to do it this way, because the month and week intervals are often used to see ranges of data, so more often than not the user would be interested in the interval after the currently selected. The user is still able to manually set the interval to a single interval, but that will show up as a single bar in the chart.

6.8 Input fields

6.9 PLU input

Figure 6.4: Characters other than numbers do not work

The PLU input field got type number, so it starts to glow red when the user tries to input other characters and on mobile devices the keyboard that pops up will display numbers instead of the usual keyboard.

6.9.1 Name input

To make finding the names of the products easier, we implemented react-autosuggest. We found that the names of the products in the database are not always that intuitive, so when we wanted to retrieve by name, we usually had to retrieve by PLU first to find the name. The autosuggest solves that problem, it will show a list of suggestions of product names after typing in 3 characters in the name input field. After the autosuggest component is loaded, it will first fetch all the product names from the API. This way it does not make an unnecessary API call.

6.10 Error handling

We implemented error messages to give the user some insight on what went wrong with the API call. If the user put in the PLU wrong, they willase the user will

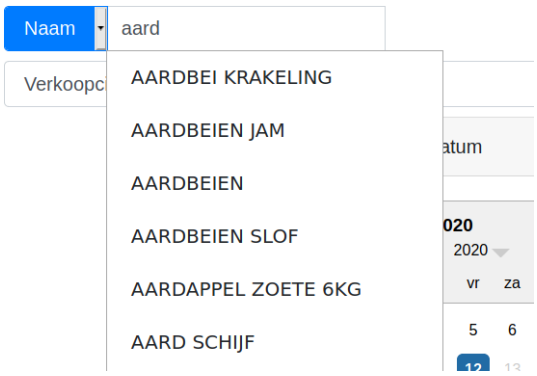


Figure 6.5: Autosuggest in action

Fout: Product naam niet gevonden.

Figure 6.6: Example of an error

see the error: 'Fout: Verbinding mislukt'. Then the user knows that they will just have to retrieve again, beca get the message that a product with that PLU is not found. If something went wrong whilst parsing the response of the API in the frontend, usually it is a connection error, so in that cuse the error was not their fault. The box in which the error appear is the bootstrap yellow *alter-warning* window. Another option we considered was the red *alter-danger* window. We found that the red one was a bit too intrusive, so we chose the softer yellow window. The error disappears when the user retrieves again.

The errors messages are defined in the API and are returned by it with the appropriate code. The frontend then checks if the response code was 200, if it is then parse the response like normal. If the code is not 200, then try to parse a message from the response. If that is a valid message, then display the error, otherwise something undefined went wrong, which is usually a network error. Here is how that looks in code:

```
1     await fetch(url, {
2       method: 'GET',
3     })
4       .then((response) => {
5         if (response.ok) {
6           return response.json();
7         }
8         response.text().then((msg) => {
9           try {
```

```
10              const parsed = JSON.parse(msg);
11              onError(parsed.message);
12            } catch (error) {
13              onError('Verbinding mislukt');
14            }
15          });
16          return null;
17        })
```

6.11 Navbar

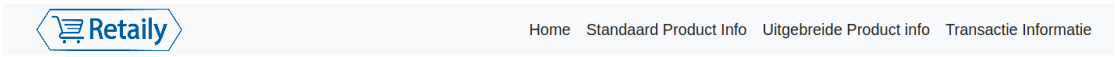


Figure 6.7: The Navbar on desktop

The web-app has a navigation bar at the top. There are only a couple of pages shown there, so the user can easily to find the page they are looking for. On mobile, the navbar becomes a dropdown menu, for ease of use.

6.12 Retrieving

Figure 6.8: Spinning retrieve button

The retrieve button is a grey button at the bottom of almost every page. The button is at the bottom, as users usually navigate the page top to bottom. After the button is clicked, the button is disabled and displays a spinner. This makes it clear to the user that their click on the button actually did something and prevents the user from clicking on it when it is in the middle of loading data. The button returns to its standard form when it gets the message from the component that fetches the data that the data is loaded. The spinner used is *spinner-border* from Bootstrap.

6.12.1 Shortcuts

As a shortcut, we implemented a keylistener that retrieves on pressing Enter. This also works on phones. Also after a product is scanned by the barcodescanner, the results will be automatically retrieved. We implemented these shortcuts to make the users experience with the web-app a little more streamlined.

6.13 ProductInfo

We implemented two separate ProductInfo pages, extended and simple. The simple one is created for people in the shop that do not need all the extra options, but only the barcode scanner and the Product Info Table. These people are on mobile devices, so a small scanner button like on extended product info would not suffice. That is why the "Start scanner" button is so large on the simple Product Info.

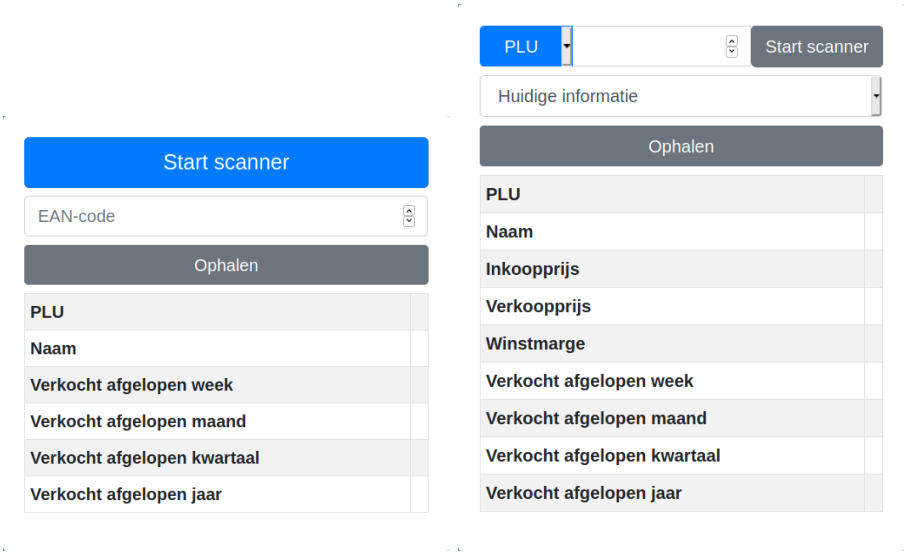


Figure 6.9: Comparison of simple and extended product info on mobile devices

6.14 Code Styling

For proper code styling of the frontend we used the ESLint linting tool with the styling preset from Airbnb.

6.15 Testing

The testing of the frontend we did using Enzyme and Jest. The tests can be found in the *frontend/src/__tests__* directory.

Technology Stack

Languages:

- SQL - the go-to language for relational databases management systems. Since the DBMS we use is MySql, we use SQL for operating with the data within the database.
- Python 3 - the language we use for building our backend, and more precisely the API since the web framework we chose, Flask, is built upon this language. The advantages of this language are that it is very easy to learn for newcomers and does a lot of the management and low-code parts automatically, without having to worry a lot about memory allocation and management.
- JavaScript - the language we use for our frontend part, more precisely the language that ReactJS is built upon, which is the framework we use for developing fast user interfaces.
- HTML - this one and the next one don't need an explanation for the simple reason that they're the lowest languages one uses for web development and web page layout.
- CSS

Libraries:

- SQLAlchemy - we use this library for easier management and access to the database where we store all the information.
- PyMySQL - the driver for access to MariaDB database.
- ChartJS - library for showing/drawing charts on the frontend side. We covered the reason for choosing this library in the frontend design part.
- QuaggaJS - library for reading PLUs from products. Why we chose this was said earlier in a different section.
- react-autosuggest - library for the product names suggestions on the frontend.
- react-datepicker - library for the date pickers we have on the various product pages.

- watchdog - library that allows to monitor a given folder on the system for the appearance of new files.

Extensions:

- systemd - a system and service manager on linux systems that allows creating services that always run with a specific given task. Our MariaDB database, Flask backend using Gunicorn and frontend using Nginx runs on this.
- Ajax - this is a group of technologies that allow for asynchronous data transfer of a web-page with a server without reloading the page. It is used by ReactJS for achieving much of the functionality it provides, including the server requests.

Building tools:

- MariaDB - the DBMS (database management system) we use for storing the data related to the shop. It is one of the main DBMS used these days and has a lot of documentation and tooling that help us achieve our goals and implement the required functionality.
- Flask - the web framework we use for creating the backend, more precisely the API. Written in Python. Is lightweight and allows for fast prototyping.
- ReactJS - the JavaScript library that we use for building our frontend, which interacts with the backend and defines the look and the functionality of the UI.

Testing:

- SonarQube - as proposed by the course coordinators, we use this for testing our code to see various possible problems, such as code smells and bugs.

From our first meeting with the client, he stated very clearly that the stack is fully at our discretion. For the database we use MariaDB and for testing, we use SonarQube and self-written unit tests.

Deployment

We were given access to a virtual machine on `CentOS` for the deployment of all our services. We used the `systemd` service management system to create services that run at all times with their required goal:

- The parser is a vanilla Python script that runs at all times, monitoring a given folder on the system for the appearance of new files.
- The database is a `MariaDB` service that serves and gives access to the database. At the moment the access is allowed only internally, from localhost addresses.
- The backend is served using a `Gunicorn` service, that is an enterprise-grade deployment framework for `Flask`-based websites.
- Since the frontend is static JavaScript and HTML, we deployed it using a `Nginx` service that simply forwards all the requests to the `build` folder on the machine.

As of the beginning of June 2020, the whole stack is set as follows: there is a script running on the machine in the store that copies the backups to the virtual machine that we use as the main server. The parser monitors this exact folder, and whenever a new file appears, it parses it and sends that data to the database. The backend always runs on port 7000. Whenever required the backend makes queries to the database for retrieving information. The frontend, on the other hand, runs on port 80 for http connections (that are forwarded to an https connection) and on port 443 for https connections. It makes queries to the backend on the given port (7000).

It is important to note that we're serving everything on https because, given the scope of our web-app, it has to use the camera in the browser, which is only allowed via https connections. This obliged us to buy an SSL certificate for the web-app. Since SSL certificates can only be applied on domain names, we had to buy a domain name as well. As a bonus, this step made access to the web-app easier.

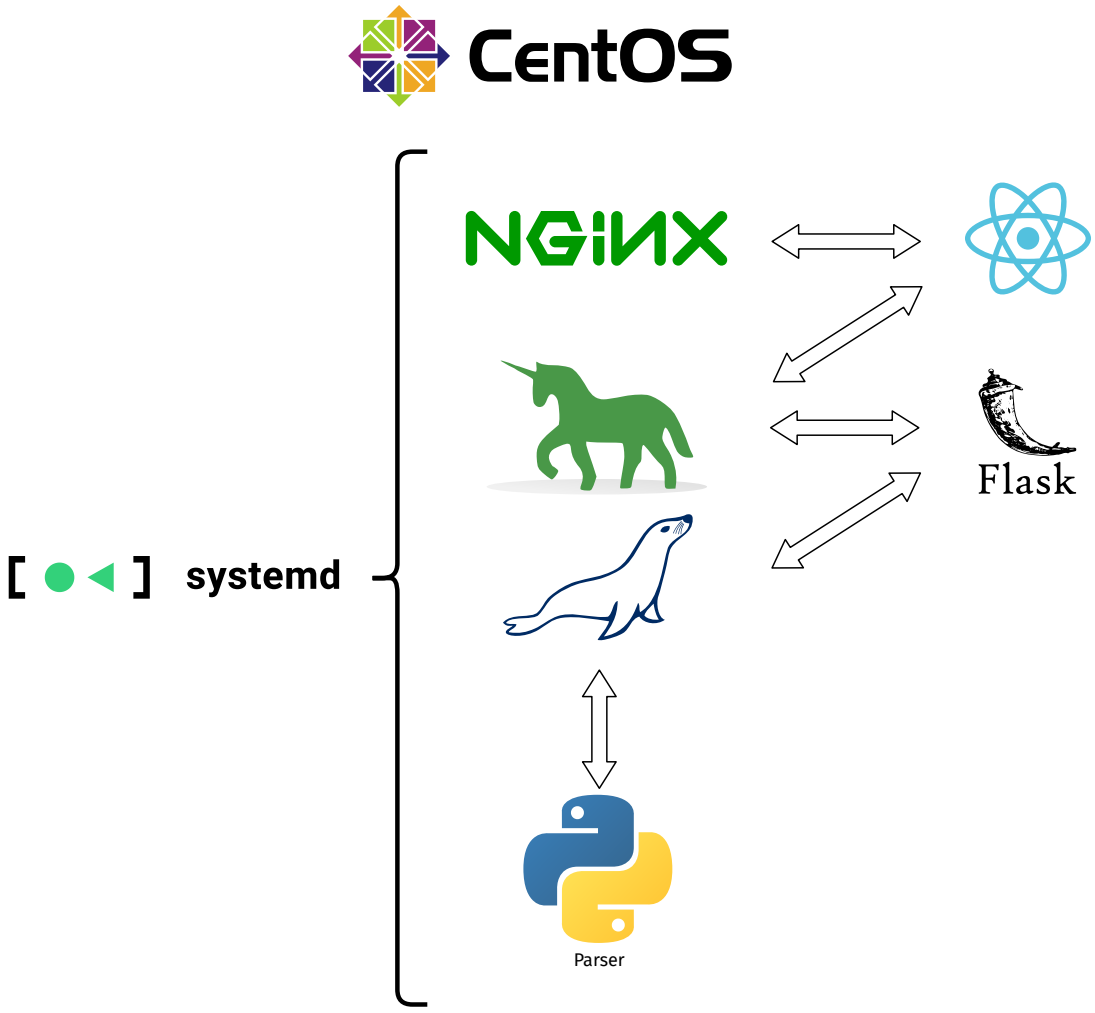


Figure 8.1: Visual representation of the actual stack.

Team organisation

For organisational matters and for optimizing our workflow, we are using the **Scrum** agile process framework. We divide our tasks in sprints and attempt to get them done by the end.

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.

We use Trello for managing the tasks that need to be done, the tasks that have already been done and the ones in progress. There we assign people to these tasks so we can keep track of the progress. Moreover, every week we gather with the team on Discord and work together for a few hours, which provides us with the opportunity to be more productive and effective as everyone is on the same page.

The second sprint aimed 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 groups, 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 altogether at 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
Abel	27.03.20	API Design	Added some API endpoints	1h
Arjan	05.04.20	Frontend design	Added design decisions	1h
Arjan	06.04.20	Design overview	Created diagram with explanation	1h
Arjan	06.04.20	Database design	Cleaning up	1h
Dan	06.04.20	Whole document	Fixing mistakes, adding more justifications, improving document layout and appearance, added Parser design	4h
Arjan	06.04.20	Backend design	Rewritten to make it more readable and consistent	3h
Arjan	11.06.20	Backend design	Updated to correspond to the current version of the application	3h
Florian	12.06.20	Frontend design	Updated to correspond to the current version of the application	4.5h
Arjan	12.06.20	Title page, database design	Add logo to title page, updated the database design	1h
Dan	12.06.20	Parser design, Deployment, whole document	Updated to correspond to the current version of the application	3h