Design Document

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: Dr. Mohamed Soliman
Teaching Assistant: Hichem Bouakaz
Last Updated: Monday 6th April, 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 database, then the design of the API and lastly 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.

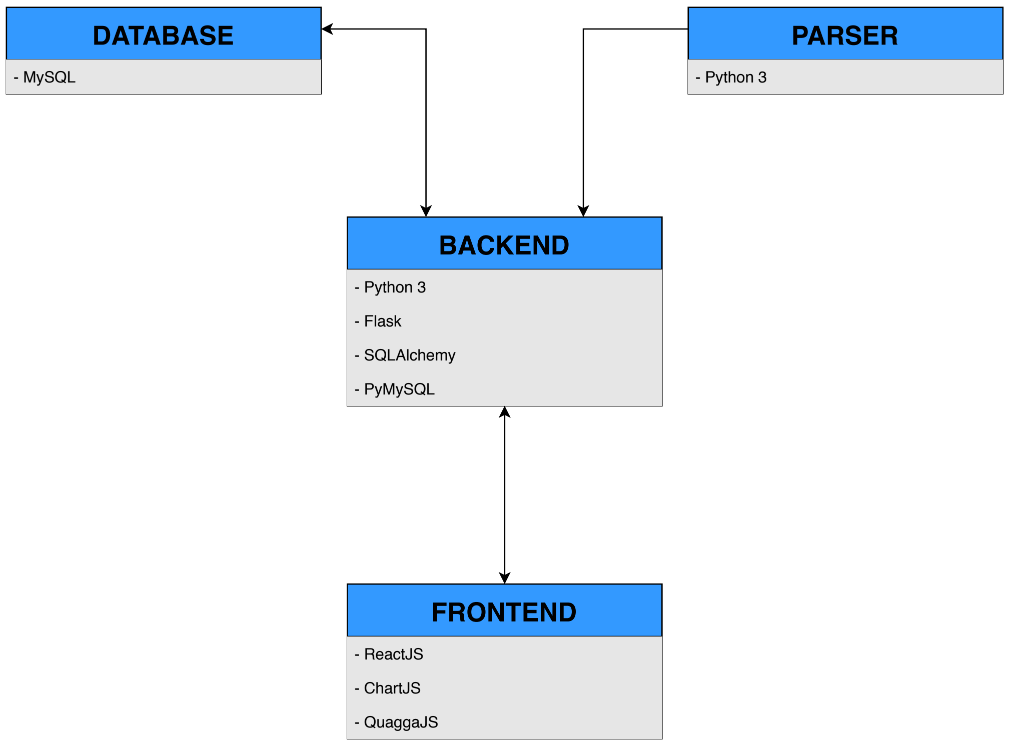
"A software toolbox for small retail shops" aims to provide many of these services to the Dorpswinkel in Sauwerd (a village 10km north of Groningen). Our product is a database system specifically built to fit the environment, and needs, of the Dorpswinkel in Sauwerd. It provides a better basis to retrieve 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, 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!

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.

Design overview

To make the system more understandable, an overview of it is created below. It contains four tables: database, parser, backend and frontend. Besides, it includes the current type of connection between the tables. 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.

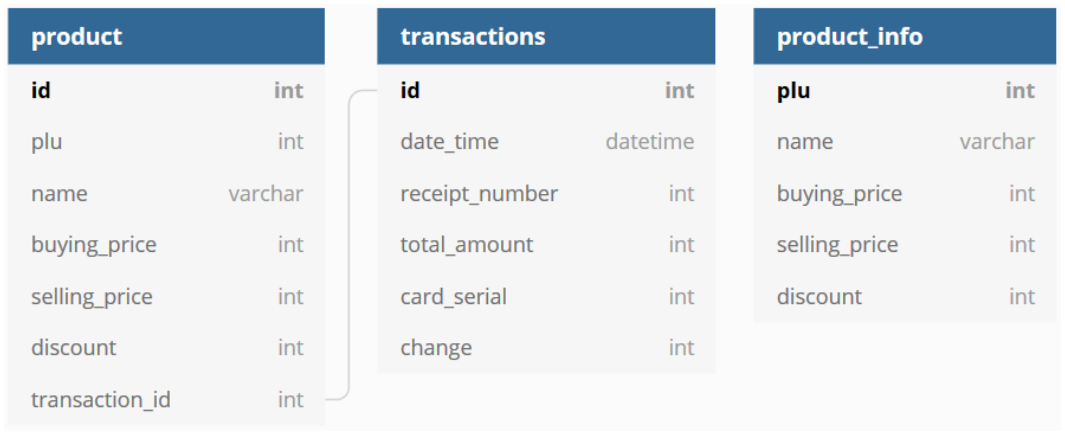


Database design

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

3.1 Overview and explanation

An overview of the database can be found below.



product	
id	**int**
plu	int
name	varchar
buying_price	int
selling_price	int
discount	int
transaction_id	int

transactions	
id	**int**
date_time	datetime
receipt_number	int
total_amount	int
card_serial	int
change	int

product_info	
plu	**int**
name	varchar
buying_price	int
selling_price	int
discount	int

3.1.1 product

The first table is the product table. This table contains all individual products that are either sold (when there exists are a reference to a transaction) or still in the shop (when a reference to a transaction does not exist). 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 and an id. The id is used as a primary key in the table. PLU could not be used as a primary key since for e.g. every banana has the same PLU. The selling_price depends on the time that a product is sold. The current information can be retrieved from the table product_info. The discount on a product can also be stored in the table. As mentioned, the transaction_id is only used when the product is sold.

3.1.2 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), total_amount (total price of the products combined) and it has either a card_serial (payment with PIN) or a change (cash payment).

3.1.3 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. This table includes the PLU, the name, the buying_price, the selling_price and discount of a product.

3.1.4 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

4.1 API 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. When a Flask route (API endpoint) is called, Flask requests the underlying data from the database, transforms it as required and returns it in JSON format. A very shallow example can be found below.

```
@app.route("endpoint-URL")
def method():
    result = interactWithDatabase()
    return jsonify(result)
```

4.1.1 Response codes

The frontend is a separate part of the application which interacts with the backend using HTTP methods, such as GET, POST, DELETE and other, in other words, typical CRUD API requests. Hence it is useful to provide the client (frontend in our case) with useful response codes. The following response codes are being used:

- **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.

4.1.2 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 consistency is also used in the following part.

inventory

1. **Objective**: GET all the items that are currently in the shop (inventory)
   Endpoint: /inventory/
   Method: GET

Keys:
HTTP status codes:

(a) 200 (SUCCESS)
(b) 404 (NOT FOUND)

Response:

```
{
    "products": [
        {
            "buying_price": "number",
            "discount": "number",
            "id": "number",
            "name": "string",
            "plu": "number",
            "selling_price": "number",
            "transaction_id": "number"
        }
    ]
}
```

product

1. **Objective**: GET the information of a product with a certain PLU or name
   Endpoint: /product/
   Method: GET
   Keys:

   - plu - only required without a name
   - name - only required without a PLU

   HTTP status codes:

   (a) 200 (SUCCESS)
   (b) 400 (Bad Request) - dt1 and/or dt2 is missing
   (c) 404 (NOT FOUND)

   Response:

```json
{
    "buying_price": "number",
    "discount": "number",
    "id": "number",
    "name": "string",
    "plu": "number",
    "selling_price": "number",
    "transaction_id": "number"
}
```

2. **Objective**: PUT a new buying price on a product with a certain PLU or name
   Endpoint: /product/buyprice
   Method: PUT
   Keys:

   - plu - only necessary without a name
   - name - only neceassary without a PLU
   - price - necessary

   HTTP status codes:

   (a) 200 (SUCCESS)
   (b) 400 (Bad Request) - price is missing or plu and name are missing
   (c) 404 (NOT FOUND)

   Response:

```json
{
    "buying_price": "number",
    "discount": "number",
    "id": "number",
    "name": "string",
    "plu": "number",
    "selling_price": "number",
    "transaction_id": "number"
}
```

3. **Objective**: PUT a new selling price on a product with a certain PLU or name

Endpoint: /product/sellprice
Method: PUT
Keys:

- plu - only necessary without a name
- name - only neceassary without a PLU
- price - necessary

HTTP status codes:

(a) 200 (SUCCESS)
(b) 400 (Bad Request) - price is missing or plu and name are missing
(c) 404 (NOT FOUND)

Response:

```
{
    "buying_price": "number",
    "discount": "number",
    "id": "number",
    "name": "string",
    "plu": "number",
    "selling_price": "number",
    "transaction_id": "number"
}
```

sales

1. **Objective**: GET the sales between two timestamps, optionally for a certain product
   Endpoint: /sales/
   Method: GET
   Keys:

   - plu - only necessary without a name
   - name - only necessary without a PLU
   - dt1 - necessary (dt = datetime)
   - dt2 - necessary (dt = datetime)

   HTTP status codes:

(a) 200 (SUCCESS)
(b) 400 (Bad Request) - dt1 and/or dt2 is missing
(c) 404 (NOT FOUND)

Response:

```
{
    "products": [
        {
            "buying_price": "number",
            "discount": "number",
            "id": "number",
            "name": "string",
            "plu": "number",
            "selling_price": "number",
            "transaction_id": "number"
        }
    ]
}
```

4.2 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.

4.2.1 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)`.

4.2.2 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 a serialization property should be defined:

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

4.2.3 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.

4.3 Parser design

4.3.1 Cases of parsing

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 simply loads the whole file at once, then splits the journal records in the file separately and extracts all the existing information from them.

```
===
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:       xxxxxxxxxxxxxxx2148
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'
    },
    ...
]
```

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.

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'
     },
     ...
]
```

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'
     },
     ...
]
```

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.

5.1 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.

5.2 Displaying Data

5.2.1 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`.

5.3 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.

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 MySql databases.
- 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.

Extensions:

- 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:

- MySQL - 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 MySQL and for testing, we use SonarQube at the moment, as required by the coordinators of the course.

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
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, rewriting some ideas, 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