# VEGS User Guide

as part of the Master of Science in business IT at the University of Göttingen

Submussion date: 28.01.2020
Author: Sebastian Kuiter
Matriculation number: 21148778
From: Thuine

Supervisor: Nils Engelbrecht
Supervisor: Tim-Benjamin Lembcke

# Contents

# List of Figures

# Listings

# 1 Introduction

### What is VEGS, and why use it?

Virtual E-Grocery Store (VEGS) is a configurable, and open-source web application for enabling researchers to design, and conduct experiments in the context of online shopping. The base application provides support for implementing a number of common treatment variations. Supported use cases are:

- Change Displayed (Product-) Information & Shop-Functions Based on Treatment Specification
- Recommendations-Systems & - Agents
- Positioning, Arrangement & (Re-) Placement
- Partitioning & Bundling
- Salience & Additional Information
- (Smart) Disclosure & Feedback
- Economic Incentives

VEGS provides a realistic web-shop Graphical User Interface (GUI), which may be configured in any number of ways to support the respective research objectives of researchers. The project is developed with widely known, and supported frameworks. JavaScript, as the overall implementation language, makes developing and customizing this tool easy. However the supported use cases can also be used without the knowledge of any programming languages. Treatments can configured, and conducted by using the visual editing means provided in the administration view.

Following are all the installation, set-up, and usage instructions needed to run, develop, and deploy this application. In addition to this examples are given, in what ways these use cases may be implemented in treatment definitions. **Disclaimer**: The script snippets, and explanations are only applicable to linux based systems (Ubuntu 19.10). For Windows or MacOS set up descriptions see the applicable online resources. The server specific instructions are based on a Ubuntu 18.04.3 LTS server installation administrated through ssh.

# 2 Installation

## 2.1 Prerequisites

Both applications are based on npm and node. Therefore first install node, which should automatically install npm as well. You could also think about installing these frameworks using nvm. If the version commands do not return the version number of either npm or node it did not install correctly.

For installation instructions specific to your system these links might be useful.

- Install Node and NPM on Windows
- Install Node and NPM on Linux
- Install Node and NPM on Mac

For testing the validity of the installation check each version command (see listing 1).

```
sudo apt install nodejs
# check installed versions
node -v
npm -v
```

Listing 1: Test installation through checking the version.

For running the back-end application you also need a running instance of MongoDB as the database server. For installation instructions see your operating system specific instructions provided here. By default MongoDB listens on localhost:27017. This port is also configured by default for the back-end application. MongoDB needs to be running before you start the back-end application. If MongoDB does not start automatically you can start it by typing the following command (see listing 2).

```
sudo service mongod start
```

Listing 2: Start MongoDB instance.

The code-base is versioned and managed using the freely available versioning tool Git. For installation instructions follow your operating system specific installation steps found here. The source files are available on Github. You can either follow the link and download the repository as a zip file or clone the repository with the following script line (see listing 3).

```
git clone https://github.com/Kuiter/vegs-repo
```

Listing 3: Clone repository from Github.

With this you cloned or downloaded the front-end, and back-end application. The dependencies for both applications are managed using npm. To install the necessary dependencies change directory into the root folders for both applications and run the following command (see listing 4). This will install all necessary dependencies to run the applications. After this you might want to update the dependencies, and audit-fix any critical security issues by running the following commands (see listing 4). With this the dependencies are updated and any security issues are fixed, when they have known fixes.

```
npm install
npm update
npm audit fix
```

Listing 4: Set up script for npm projects.

## 2.2 Local hosting and development

For localy hosting the front-end, and back-end applications input the following commands inside the root folder of each project folder.

```
# front-end, during development mode, restarts when changes are made to code-base
npm run start

#back-end, in development mode, restarts when changes are made to code-base
npm run start-watch
```

Doing this both applications are hosted in "development mode", after changing source files the applications are recompiled, and the sever is automatically restarted. The above commands act as aliases for the following commands specified in the respective package.json (see listing 5) file in the root folder of each application.

```
# front-end
"scripts": {
    ...
    "start": "ng serve --proxy-config proxy.conf.json",
  }
# back-end
"scripts": {
    ...
    "start-watch": "nodemon src/index.js",
  }
```

Listing 5: Configurations of scripts array in package.json.

After starting the development server the front-end is is hosted on localhost:4200, and the back-end listens on localhost:3000.

## 2.3 Running the application on a server

For deploying the applications to a server there are many different approaches that can be taken. In this context I will detail the configuration steps required for deploying both the front- and back-end applications on one server. The server is running Ubuntu 18.04.3 LTS. The hardware configuration of the server is variable and is dependent on your projects scope.

Before deploying the application on to your server make sure you have all software detailed in section Installation installed on your server.

### 2.3.1 Front-end

Before you can deploy the application to a server you have to compile the source code of the front-end project. Run the following command in the root folder of the application (see listing 6).

```
ng build --prod=true
```

Listing 6: Build production app from source files.

This compiles the typescript source files, and generates the resulting application files in /dist folder in the root directory. For copying the files onto the server I use the following command (see listing 7).

```
scp -r <path_to_root_dir>/dist/storefront/ <user>@<server_IP>:~/
    deployment
```

Listing 7: Upload files of front-end to server.

This copies the files generated in the dist folder onto the server into your home directory into the sub-folder /deployment. This sub-folder is specifically created for automatic deployment purposes and necessary if you want to also use the deployment script I provide in the appendix (see appendix A). You can also just secure copy the files directly into the /var/www directory but do not forget to restart nginx if you do it like this.

The front-end application will be served from a nginx HTTP server. To provide applications to be hosted you need to configure a server block inside the ngingx configuration files located under /etc/nginx/sites-available. You can look at the server configuration that are provided in the following listing 9. To configure the default configuration file of the server run the following command (see listing 8). It is recommended to copy the application files into the /var/www directory, which is referenced in the server configuration as "root".

```
# configure server block
sudo nano /etc/nginx/sites-available/default
# configure server block as seen in nginx server definition
# copy the application files to the root location
sudo mv ~/deploy/<app_name> /var/www
# reload nginx
```

Listing 8: Nginx configure sites available.

```
server {
    listen 443 ssl;
    listen [::]:443 ssl;
    client_max_body_size 50M;
    include snippets/self-signed.conf;
    include snippets/ssl-params.conf;

    server_name vegs.codemuenster.eu;

    root /var/www/storefront;
    index index.html index.htm index.nginx-debian.html;

    location / {
        try_files $uri $uri/ /index.html =404;
    }
    location /api/ {
        proxy_pass http://localhost:4000;
        rewrite /api/(.*) /$1 break;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
    ssl_certificate /etc/letsencrypt/live/vegs.codemuenster.eu/fullchain
   .pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/vegs.codemuenster.eu/
   privkey.pem; # managed by Certbot
}
```

Listing 9: Nginx server definition.

The location block configures a url rewrite for passing HTTP requests to the back-end application, in the example hosted on localhost:4000. The ssl encryption is handled by certbot a free tool from lets encrypt. To automatically set ssl_certificates up you can run the following command (see listing 10). This is optional as you can also host the application http only, or configure your own certificate schema.

```
# install certbot
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get install certbot python-certbot-nginx
# configure ssl certificates
sudo certbot --nginx
```

Listing 10: Certbot installation and nginx configuration.

You need to have a domain name for certbot configuration to work. After configuring all of this restart the nginx service, and check if no errors occur while starting the service (see listing 11).

```
# restart service nginx
sudo systemctl restart nginx.service
# check status
sudo systemctl status nginx.service
```

Listing 11: Restart nginx and check status.

### 2.3.2 Back-end

The back-end application needs a running instance of MongoDB on the server, if you want to outsource the MongoDB you need to reconfigure the back-end configuration files. In this example I created folders underneath /opt. The directory structure is as follows /opt/node/<app_name>. The back-end application is directly implemented using JavaScript, so it does not need to be compiled. To copy the files to the server I use the following command.

```
# copy back-end files to the server
rsync -av -e ssh --exclude 'node_modules' <path_to_root_folder>/ <user>@
    <server_IP>:~/deployment/<app_name>/
```

Listing 12: Upload back-end files to server.

This copies all the application files excluding the node_modules folder to the deployment sub-folder in your server users home directory. From here you need to copy the files to the desired location, in my case /opt/node. After this you need to install the node_modules, so the required dependencies of the application.

```
# copy the files to the desired location
sudo mv ~/deployment/<app_name> /opt/node
# install node modules
cd /opt/node/<app_name>
npm install
# optional
npm update
npm audit fix
# optional: test if there are any errors
npm run start
```

Listing 13: Configuration file for systemctl.

Again at this point you might want to update, and audit fix the dependencies of the project (see section Installation). For testing you might want to run the application from the command line to see if any errors arise. For easier management it is suggested creating, and registering a service for starting the application with systemd. One option is to add a *.service file to systemctl. Create a configuration file by following the steps detailed in listings 14, and 15.

```bash
# creating the service file
sudo nano /lib/systemd/system/<app_name>.service
```

Listing 14: Configuration file for systemctl.

```
[Unit]
Description=s<description>
Wants=network.target mongod.service
After=network-online.target mongod.service

[Service]
ExecStart=/usr/bin/node /opt/node/<app_name>/src/index.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

```bash
# enable the service
systemctl enable <app_name>
# reload the system daemon
sudo systemctl daemon-reload
# for debugging errors while starting the service
sudo journalctl -u <app_name>
```

Listing 15: Bash commands for enabling and reloading the systemctl services.

### 2.3.3 Email server, user confirmation

Optionally you can require users to confirm ownership over their email addresses. To enable this set the flag confirmed on a new user creation operation to false (see listing 16). This represents the confirmation of ownership over the email address, and is only set to true if the user confirms a confirmation mail. Additionally for this you need to configure an email server for to be able to send verification emails. The configuration file can be found in <root_folder>/src/app/mailer.js. Email confirmation is handled by sending a JSON Web Token (JWT) and a confirmation link to the email address of the user. After the user clicks the link provided, the back-end confirms the validity and sets the confirmed flag of the user object to true. If the confirmed flag is true the new user can log-in, he is prohibited from loggin-in if this flag is false.

```
// auth controller, function register_new_user
var user = new User({
      username: req.body.username,
      email: req.body.email,
      password: hashedPassword,
      // for email confirmation
      confirmed: true // set to false
    });
```

Listing 16: User data when created.

# 3 Treatment Configuration

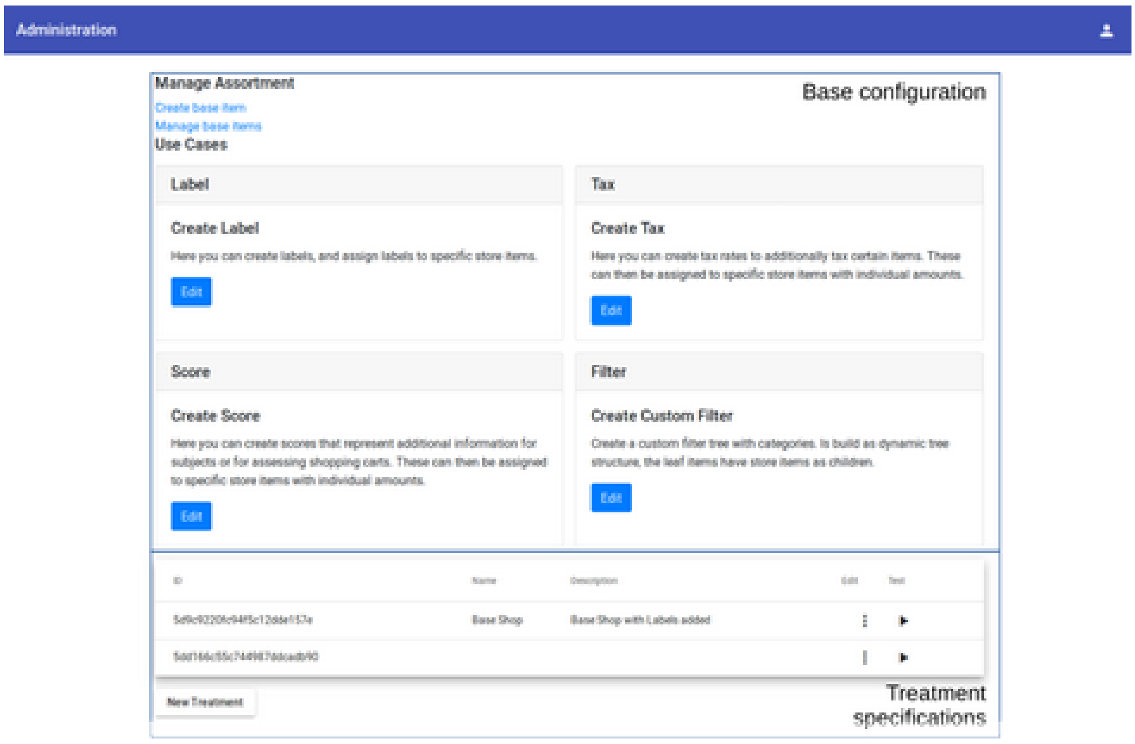
## 3.1 Base configuration options

### 3.1.1 Base items

To configure a treatment specification navigate to <base_URL>/admin. Here you can see your base configuration options. Fundamentally there are two kinds of configurations you can make. You can add base configuration, these are named this way as they are treatment independent. You can configure them once and reuse them throughout your treatment specifications. For instance, at the top of the screen in the section "Base configuration" (see figure 1) you can configure the base items that you want to use throughout your various treatment configurations. These you can configure once and allocate to your treatment specifications as needed.

For adding base items use the link "Create base item" and fill out all the information you want to add to the item. You can add an image or alter information about the item after you created the base items. Click on the "Manage base items" link and search the item you want to reconfigure. Via the dropdown you can choose to edit, delete or add an image or change the image associated with the selected item (see figure 2). The item model features all the necessary information for displaying a food item in a real world shopping environment. It features the price, vat, content type, and amount, the nutritional information, the display name, brand name, and additional description information, ingredients, and allergens (see figure 3, and 4). The currency takes the official country currency codes specified by ISO 4217. The additional information on the item model provides the use case specific information needed to provide the specific functionality e.g. score, taxes, swaps, and labels (see listing 17).

**Important:** Base items can not reference swap options when they are created as base



Manage Assortment

Create base item

Manage base items

Use Cases

**Label**

Create Label

Here you can create labels, and assign labels to specific store items.

**Tax**

Create Tax

Here you can create tax rates to additionally tax certain items. These can then be assigned to specific store items with individual amounts.

**Score**

Create Score

Here you can create scores that represent additional information for subjects or for assessing shopping carts. These can then be assigned to specific store items with individual amounts.

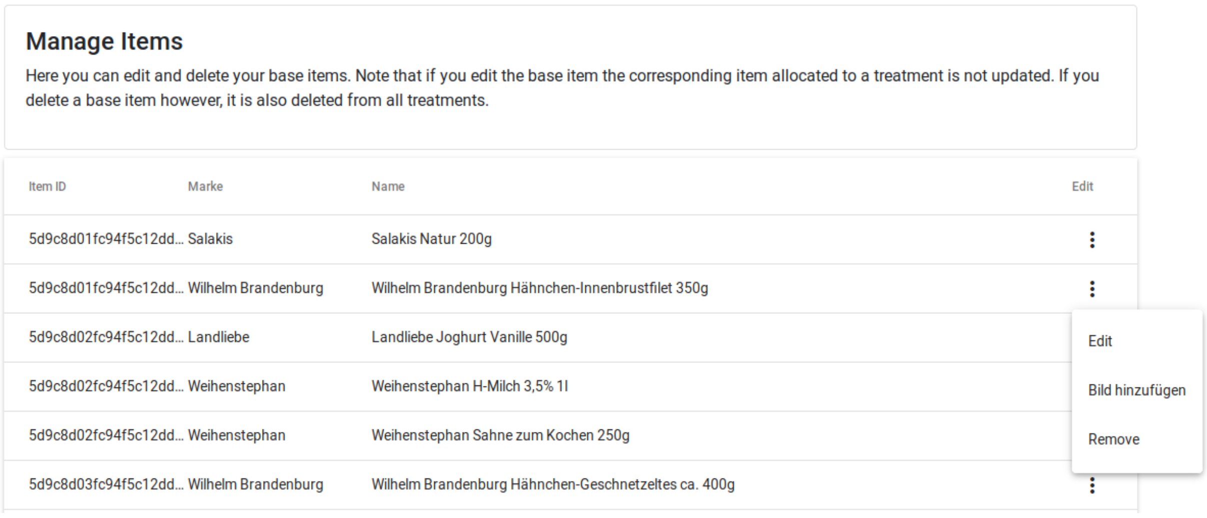
**Filter**

Create Custom Filter

Create a custom filter tree with categories. Is build as dynamic tree structure, the leaf items have store items as children.

| ID | Name | Description | Edit | Test |
|---|---|---|---|---|
| 5d9c92209c94f5c12dde157e | Base Shop | Base Shop with Labels added | ⋮ | ▶ |
| 5dd166c55c744987ddcadb90 | | | ⋮ | ▶ |

Figure 1: Admin view: Base use case configuration and treatment addition.



Manage Items

Here you can edit and delete your base items. Note that if you edit the base item the corresponding item allocated to a treatment is not updated. If you delete a base item however, it is also deleted from all treatments.

| Item ID | Marke | Name | Edit |
|---|---|---|---|
| 5d9c8d01fc94f5c12dd… | Salakis | Salakis Natur 200g | ⋮ |
| 5d9c8d01fc94f5c12dd… | Wilhelm Brandenburg | Wilhelm Brandenburg Hähnchen-Innenbrustfilet 350g | ⋮ |
| 5d9c8d02fc94f5c12dd… | Landliebe | Landliebe Joghurt Vanille 500g | |
| 5d9c8d02fc94f5c12dd… | Weihenstephan | Weihenstephan H-Milch 3,5% 1l | |
| 5d9c8d02fc94f5c12dd… | Weihenstephan | Weihenstephan Sahne zum Kochen 250g | |
| 5d9c8d03fc94f5c12dd… | Wilhelm Brandenburg | Wilhelm Brandenburg Hähnchen-Geschnetzeltes ca. 400g | ⋮ |

Figure 2: Admin view: Configure base items.

items. The reason for this is that items are deep copied into the treatment specifications and have a new _id attribute. Only the new _id reference will be considered when the reference array is checked. An image can only be added to an item after the base data was created. The data model supports only one image per item (as does the display method of the front-end application).

```
// item model schema
let ItemSchema = new Schema({
  owner: String,
  oldID: { type: String, default: '' },
  externalID: String,
  // For display purposes
  name: { type: String },
  brand: { type: String },
  description: [
    {
      header: { type: String },
      text: { type: String },
    }
  ],
  nutritionalTable: {
    kj: { type: Number },
    kcal: { type: Number },
    totalFat: { type: Number },
    saturatedFat: { type: Number },
    totalCarbohydrate: { type: Number },
    sugar: { type: Number },
    protein: { type: Number },
    salt: { type: Number },
  },
  netPrice: { type: Number },
  currency: { type: String },
  vat: { type: Number },
  amount: { type: Number },
  content: {
    contentType: String, // fluid or solid
    amountInKG: { type: Number },
    displayAmount: { type: String }
  },
  // In front-end thumbnails will be handeled by `th_+imageID`
  image: {
    th: String,
    full: String
  },
  ingrediants: { type: String },
  allergenes: { type: String },
  baseAttributes: [String],
  taxes: [{
    taxID: String,
    header: String,
    description: String,
    shortDescription: String,
    amount: Number
  }],
  score: {
    scoreID: String,
    header: String,
    description: String,
    maxValue: Number,
    minValue: Number,
    amount: { type: Number },
  },
  subsidies: {
    subsedieID: String,
    amount: Number
  },
  // ItemIds
  swaps: [String],
  label: [String],
  // for base Filter funtionality?
  tags: [String],
  niceness: { type: Number, default: 1 }
}, { timestamps: true });
```

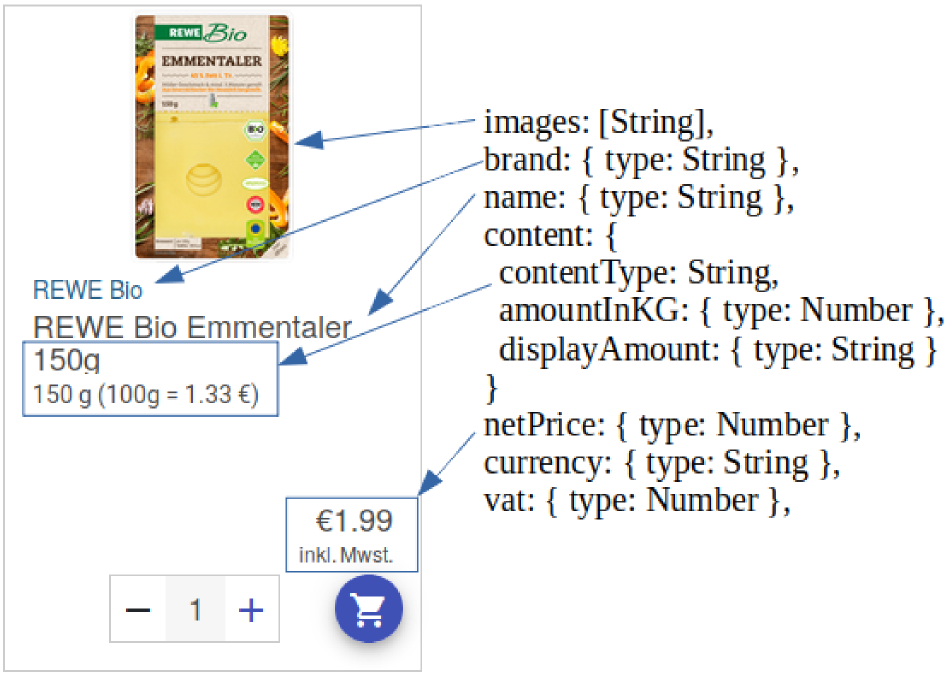
Listing 17: Item model schema.

**Important:** If you reconfigure a base item all the items you already allocated to a treatment will **not be changed**. This ensures the preservation of the original treatment configuration made by you.

### Base Attributes

The "baseAttribute" is an array in which you can specify arbitrary attributes for each item, on the basis of these subjects can limit the selection of items based on the specified attributes. E.g. base attributes like "Bio", "Lactose free", and "Free range" could be added. These names are then also displayed in the item limiting function in the shop view. **Note:** Different spellings of the same attribute will result in an additional limiting option.

### Tags


REWE Bio

REWE Bio Emmentaler

150g

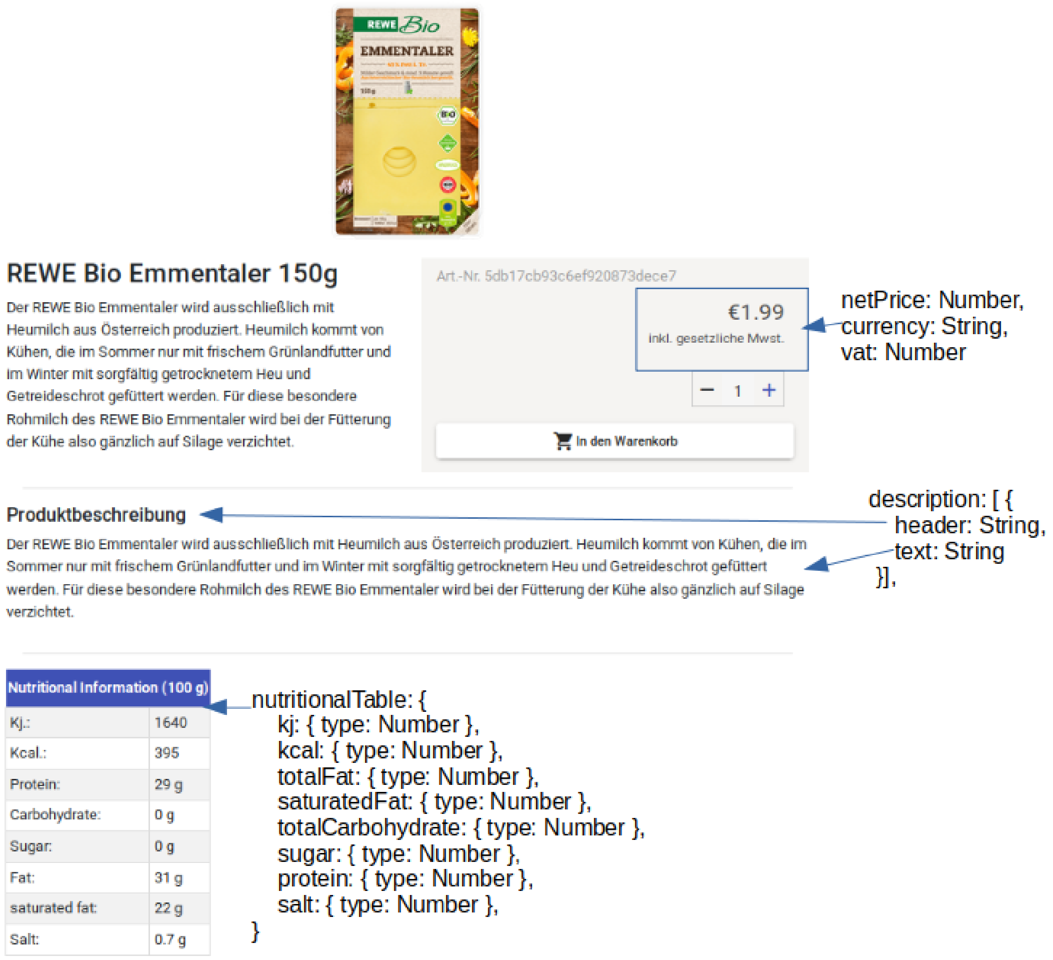
150 g (100g = 1.33 €)

€1.99

inkl. Mwst.

```
images: [String],
brand: { type: String },
name: { type: String },
content: {
  contentType: String,
  amountInKG: { type: Number },
  displayAmount: { type: String }
}
netPrice: { type: Number },
currency: { type: String },
vat: { type: Number },
```

Figure 3: Shop view: Example mapping of item model data to the food-card component.


**REWE Bio Emmentaler 150g**

Der REWE Bio Emmentaler wird ausschließlich mit Heumilch aus Österreich produziert. Heumilch kommt von Kühen, die im Sommer nur mit frischem Grünlandfutter und im Winter mit sorgfältig getrocknetem Heu und Getreideschrot gefüttert werden. Für diese besondere Rohmilch des REWE Bio Emmentaler wird bei der Fütterung der Kühe also gänzlich auf Silage verzichtet.

Art.-Nr. 5db17cb93c6ef920873dece7

€1.99

inkl. gesetzliche Mwst.

```
netPrice: Number,
currency: String,
vat: Number
```

**Produktbeschreibung**

Der REWE Bio Emmentaler wird ausschließlich mit Heumilch aus Österreich produziert. Heumilch kommt von Kühen, die im Sommer nur mit frischem Grünlandfutter und im Winter mit sorgfältig getrocknetem Heu und Getreideschrot gefüttert werden. Für diese besondere Rohmilch des REWE Bio Emmentaler wird bei der Fütterung der Kühe also gänzlich auf Silage verzichtet.

```
description: [ {
   header: String,
   text: String
}],
```

| Nutritional Information (100 g) | |
|---|---|
| Kj.: | 1640 |
| Kcal.: | 395 |
| Protein: | 29 g |
| Carbohydrate: | 0 g |
| Sugar: | 0 g |
| Fat: | 31 g |
| saturated fat: | 22 g |
| Salt: | 0.7 g |

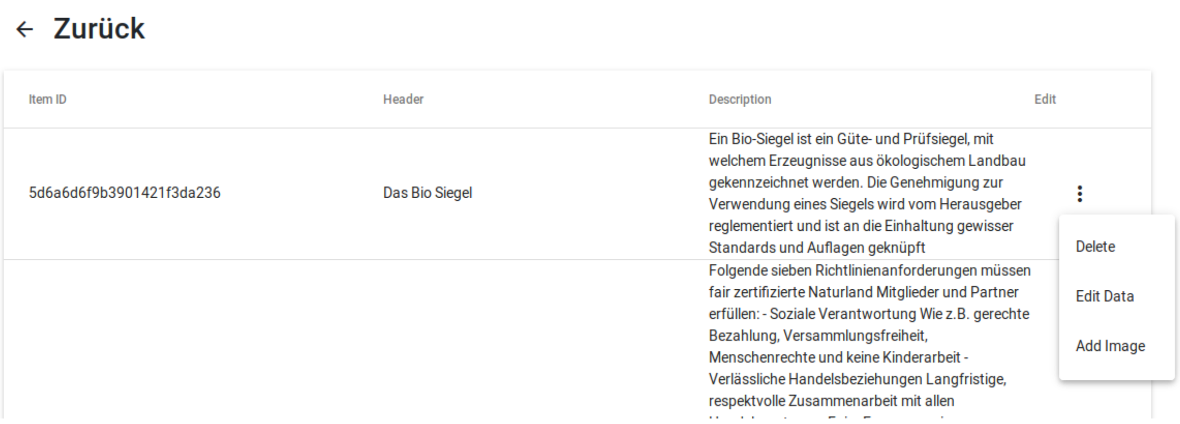
```
nutritionalTable: {
   kj: { type: Number },
   kcal: { type: Number },
   totalFat: { type: Number },
   saturatedFat: { type: Number },
   totalCarbohydrate: { type: Number },
   sugar: { type: Number },
   protein: { type: Number },
   salt: { type: Number },
}
```

Figure 4: Shop view: Example mapping of item model data to the food-details component.

The tag array configures the the base filter of the application (see section on base filter). The tag array is considered from first to second item in the array. The first item represents a parent node in the base filter, with the second element being considered its first child node. **Note:** the order in which these tags are appended in this array is important for generating the parent, child node filter tree.

**Niceness**

An additional sorting mechanism that carries between filter options is the configuration option "niceness" on the item level. This is an attribute of each item, which takes the form of a number between zero and one. Items are sorted based on their niceness, beginning with the least nice items. This ordering mechanism allows certain items to always be displayed first, or closer to the viewpoint of the subjects.



| Item ID | Header | Description | Edit |
|---|---|---|---|
| 5d6a6d6f9b3901421f3da236 | Das Bio Siegel | Ein Bio-Siegel ist ein Güte- und Prüfsiegel, mit welchem Erzeugnisse aus ökologischem Landbau gekennzeichnet werden. Die Genehmigung zur Verwendung eines Siegels wird vom Herausgeber reglementiert und ist an die Einhaltung gewisser Standards und Auflagen geknüpft | ⋮ |
| | | Folgende sieben Richtlinienanforderungen müssen fair zertifizierte Naturland Mitglieder und Partner erfüllen: - Soziale Verantwortung Wie z.B. gerechte Bezahlung, Versammlungsfreiheit, Menschenrechte und keine Kinderarbeit - Verlässliche Handelsbeziehungen Langfristige, respektvolle Zusammenarbeit mit allen | |

Figure 5: Label create view: Configure label definitions.

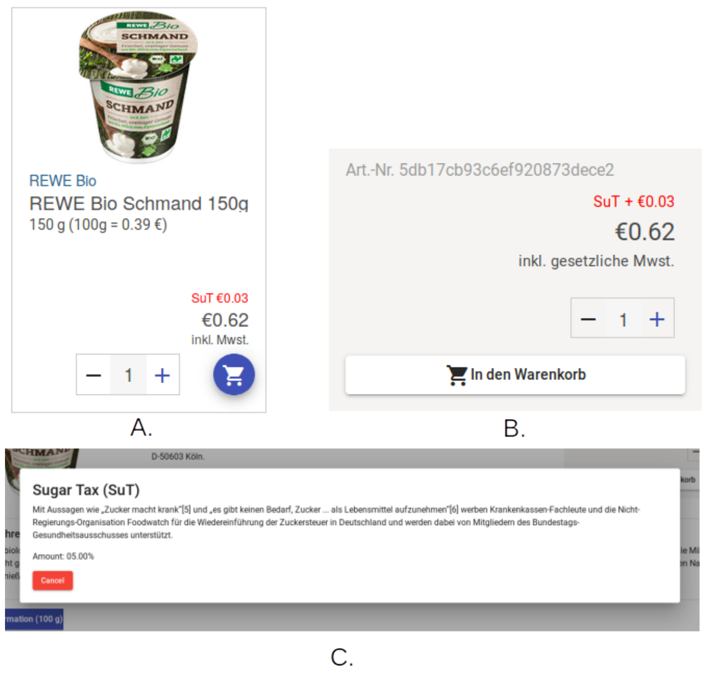
### 3.1.2 Base labels

For creating label strategies you can select the label card in the base configuration section of the admin view, this will navigate to <base_URL>/admin/labelCreate. Here you can create a new label definition by clicking the "New Label" button on the bottom left of the label create view. This view also enables you to edit, delete or add an image to an already created label (see figure 5). The following listing shows the data model of a label object (see listing 18). The header represents the label name that is being displayed. The description is text that can be viewed by a subject during a trial. The associated image is directly stored inside the label model definition.

**Important:** Label images that are not resized to be a square, and where the image background is not transparent may look out of place in the current shop implementation. To support this you might want to reconfigure the label image display mechanism. For this see the developer notes on the "food-card".

```
let LabelSchema = new Schema({
  owner: String,
  header: String,
  description: String,
  img: {
    imageID: String,
    data: Buffer,
    contentType: String
  }
}, { timestamps: true });
```

Listing 18: Label data model



Sugar Tax (SuT)

Mit Aussagen wie „Zucker macht krank"[5] und „es gibt keinen Bedarf, Zucker ... als Lebensmittel aufzunehmen"[6] werben Krankenkassen-Fachleute und die Nicht-Regierungs-Organisation Foodwatch für die Wiedereinführung der Zuckersteuer in Deutschland und werden dabei von Mitgliedern des Bundestags-Gesundheitsausschusses unterstützt.

Amount: 05.00%

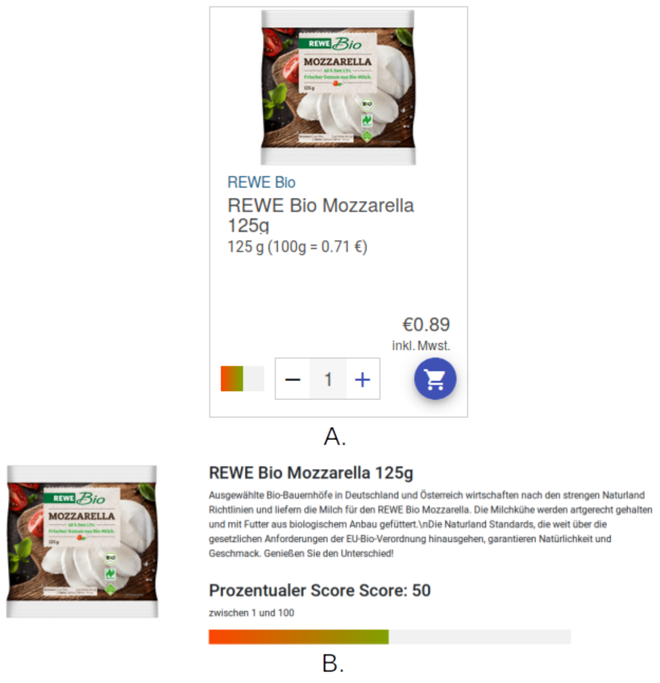
Figure 6: Shop view: (A) Tax display on food-card component, (B) tax display in food-details component, (C) tax information dialog, for showing the description of the additional tax.

### 3.1.3 Custom taxes

For creating custom tax definitions you can select the tax card in the base configuration section of the admin view, this will navigate to <base_URL>/admin/taxCreate. The following listing details the taxes data model (see listing 19). The short description will be displayed on the food card, and on the food details view (see figure 6).

```
let TaxSchema = new Schema({
  owner: String,
  description: String,
  shortDescription: String,
  header: String
});
```

Listing 19: Tax data model



REWE Bio

REWE Bio Mozzarella 125g

125 g (100g = 0.71 €)

€0.89

inkl. Mwst.

REWE Bio Mozzarella 125g

Ausgewählte Bio-Bauernhöfe in Deutschland und Österreich wirtschaften nach den strengen Naturland Richtlinien und liefern die Milch für den REWE Bio Mozzarella. Die Milchkühe werden artgerecht gehalten und mit Futter aus biologischem Anbau gefüttert.\nDie Naturland Standards, die weit über die gesetzlichen Anforderungen der EU-Bio-Verordnung hinausgehen, garantieren Natürlichkeit und Geschmack. Genießen Sie den Unterschied!

Prozentualer Score Score: 50

zwischen 1 und 100

Figure 7: Shop view: (A) Score display on food-card component, (B) score display in food-details component

### 3.1.4 Custom score

For creating custom score definitions you can select the score card in the base configuration section of the admin view, this will navigate to <base_URL>/admin/labelCreate. The following listing details the taxes data model (see listing 20). The scores can be created very freely, the base definition contains a min- and max value number. The header, and description data will be displayed in the food details view. In addition a score bar is displayed on the food-card, and food-details view (see figure 7). The actual score of an item has to be configured on an item, where the tax is allocated to.

```
const ScoreSchema = new Schema({
  owner: String,
  header: String,
  description: String,
  maxValue: Number,
  minValue: Number,
});
```

Listing 20: Score data model

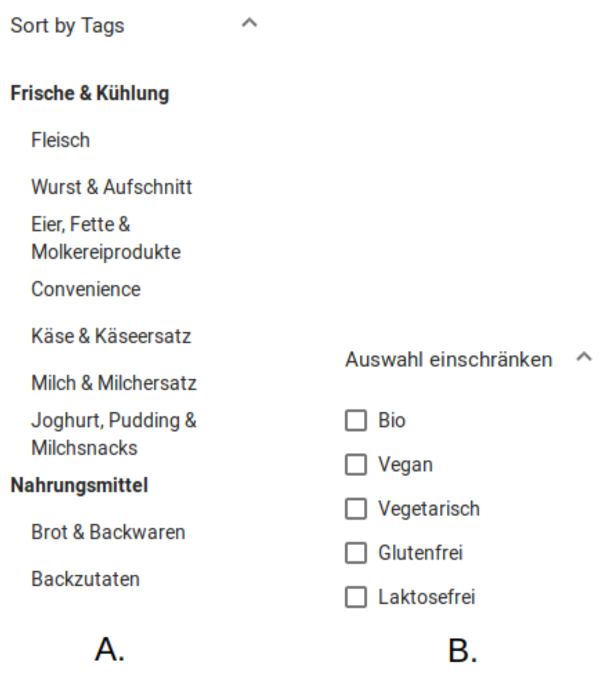


Figure 8: Shop view: (A) Base tag filter, (B) base attribute filter

### 3.1.5 Base and custom filtering

The application offers two kinds of filter types. These filters are displayed on the left hand side of the shop view. The base filter types, which are generated and displayed automatically, are based on tags, and base attributes, both of these are configured at the individual item level (see listing 17). Based on the tags array on all items allocated to a treatment a tag filter tree is generated that takes into account the first two entries of the tag array. Based on this a filter tree with one parent category, and one child category is created and displayed. The code can be found in the file .../trial/trial-services/product.service.ts. The second base filter type is based on the base attributes configured on each individual item. This is used to limit the selection to for example only items that have the trait "Bio". This filter type is also generated in the file .../trial/trial-services/product.service.ts. A depiction of both filters can be seen in figure 8. The base attribute filter can be multi-selected, where as the tag filter is single select only.

For creating custom filter definitions you can select the filter card in the base configuration section of the admin view, this will navigate to <base_URL>/admin/filterCreate. Custom filter differ from the base filter variants in that they can be created by hand. You may create any custom filter tree of you choosing. After creating a custom filter tree definition you need to add this filter tree definition to the treatment of your choice. After this is you can add items from your treatment definition to the leaf nodes of your custom filter tree. This way there are no limits to the customizability and depth of your filter

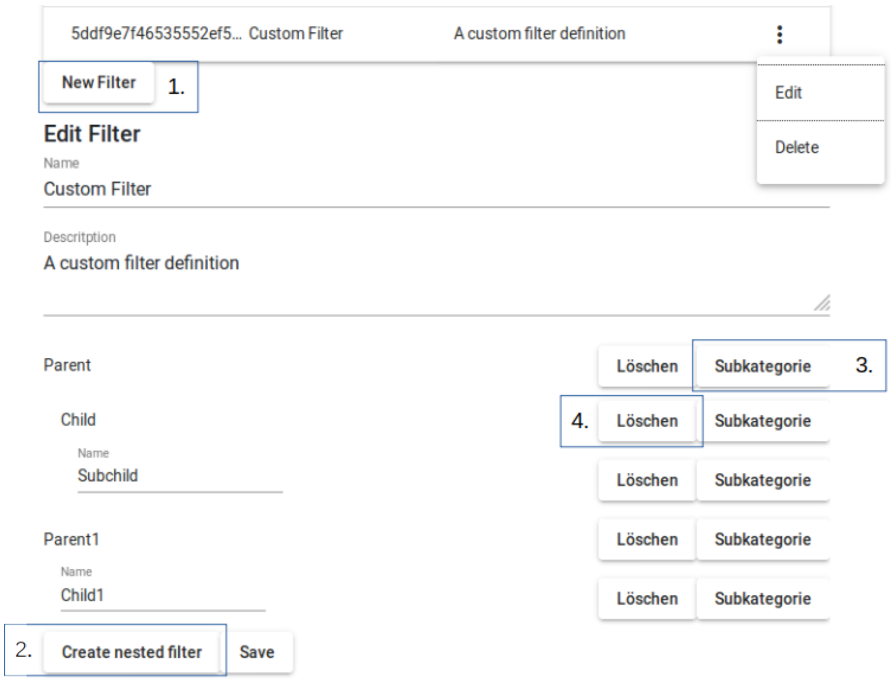


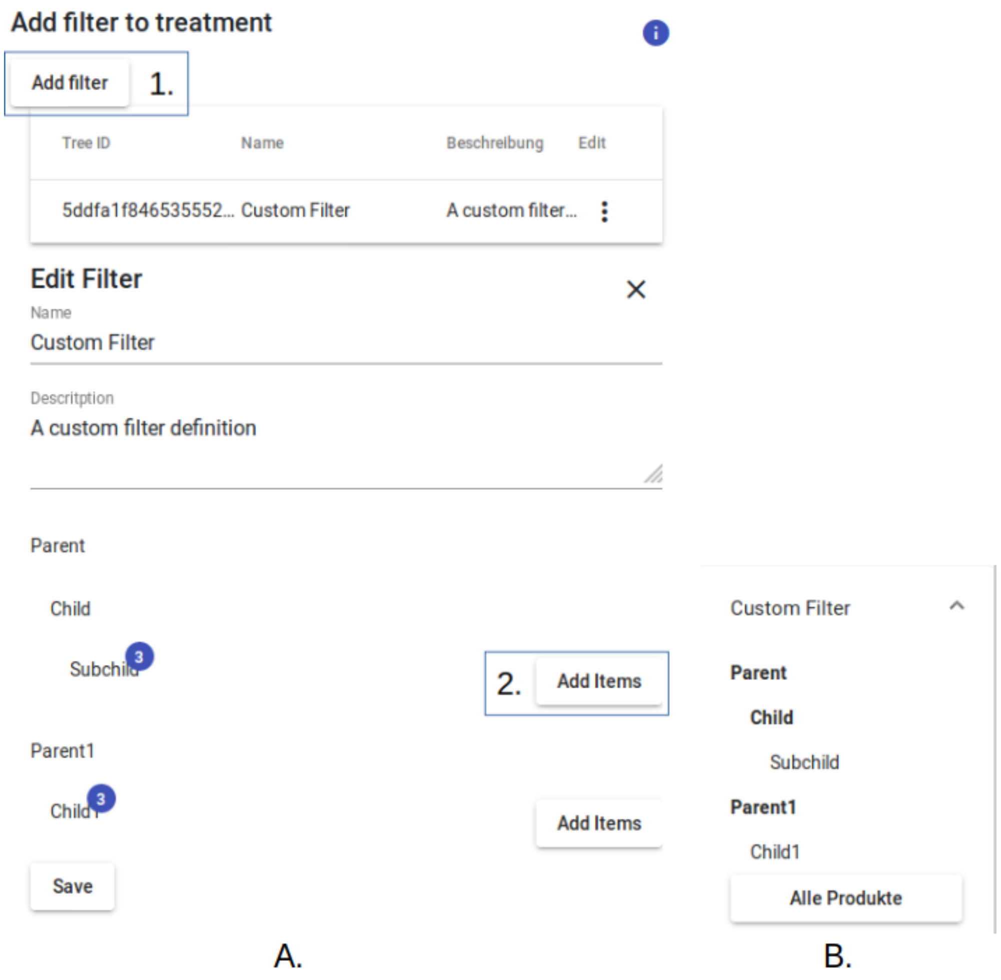
Figure 9: Admin view: 1. Create new filter, 2. create parent node, 3. create child node, 4. delete referenced node.

tree definition. See figure 9, and 10 to see individual stages you have to perform to add a custom filter to your treatment definition.

**Important:** The order in which you add the items to the leaf nodes, and the order you add filter trees to your treatment definitions will determine the order the items, and the filter are being displayed. This is partially true, as the niceness value of the product or any other ordering mechanism will rearrange the items as the are displayed when filtered. This is a side-effect of the fact that this information is pushed on to an array on the item/ treatment level.

### 3.1.6 Sorting

The base application offers a basic sorting mechanism. Subjects may sort the items based on price descending, and ascending. The sorting options are displayed as a dropdown on the top right of the trial shop view. Implementing and using other sorting mechanisms would have to be coded directly into the front-end application.



| Tree ID | Name | Beschreibung | Edit |
|---|---|---|---|
| 5ddfa1f846535552... | Custom Filter | A custom filter... | ⋮ |

Name
Custom Filter

Descritption
A custom filter definition

Figure 10: Admin view: A: 1. Add an existing filter to the treatment, 2. add treatment items to the node leaf ot the custom filter tree. B: Custom filter displayed on the left of the item grid.

## 3.2 Treatment administration

### 3.2.1 Creation and modification

Before you can use the tool as a treatment administrator you must register as a user. This is necessary to be able to match the data created to the user that created it. Navigate to the <base_URL>/auth/register screen, and input your email, and desired password. Based on the configurations made in the back-end a email confirmation is required. If this has all successfully finished, navigate to the <base_URL>/auth/login screen, and login using the newly generated user. If there are any errors observe the back-end logs, or see the returned errors in the http response. Only logged in user can access routes beginning with /admin.

For creating and modifying your treatments navigate to the URL <base_URL>/admin. At the bottom is a table with all your created treatments, and the means to create an new treatment. If you edit an existing, or create a new treatment you will be redirected to the treatment edit screen. The treatment edit screen is divided into several sections, basic information, items, filters, display options, and game options. At the top is the basic information which lets you configure the name, and description. Next is the items section, here you can add, remove, and edit items. In the filter section you can add, remove, and edit custom filters you added as custom filter (see section BASE_FILTERS_-CUSTOM). In the display options section you can configure what parts of the shop, or if extra information should be displayed, see the short information descriptions for a short functional description. The game options section contain options for configuring the shopping experience, like a maximum budget for each shopper, and if this budget is restrictive. For usage information see the corresponding use case definition and usage information in section USECASE_SECTION.

#### Creation

For creating a new treatment navigate to the main admin view <base_URL>/admin and click the button at the bottom that says "New Treatment". This will redirect you to the treatment edit screen. Before you may edit treatment specific data input the basic treatment data "Name" and "Description" situated at the top of the screen. Save the data by pressing the button "Save" directly beneath the form. This creates a new treatment with only a name and a description.

#### Modificaiton

For modifying an existing treatment navigate to the desired treatment and expand the dropdown options menu. Here click on "Edit". This will redirect you to the treatment edit view. Click "Delete" if you want to delete the specific treatment definition. **Important:** Clicking the button will immediately delete the treatment without asking for consent.

### Testing

You can view a demo of the changes you have made to the treatment by clicking on the play button on the right hand side of the "all treatments table". Or navigate to the URL: <base_URL>/t/<treatmentID>/s/0/shop/products. Notice the ../s/0 in the URL, this configures the "subjectID" to be 0, this means that no activity will be tracked while navigating the shop. This enables you to navigate, and test the treatments, and experience them as a subject would see them.

### Data model

The data model of a treatment represents the base configurations, which affect the base components, that can be customized about a treatment (see listing 23). The item array contains full copies of every base item which has been allocated to the specific treatment. This means that you are able to freely change any aspect about an item allocated to a treatment, without affecting the base items. The "showOptions" attribute contains configuration flags that configure switch on or off certain design elements of the shop. The subject options represent the object for configuring game options. For the swap configuration options see the section Recommendations-Systems & - Agents.

```
let TreatmentSchema = new Schema({
  owner: String,
  name: String,
  description: String,
  active: { type: Boolean, default: false },
  items: [Item],
  // featuredItems: [String],
  filters: [Tree],
  showOptions: {
    numOfItems: { type: Number, default: 10 },
    showSum: { type: Boolean, default: false },
    showSumScore: { type: Boolean, default: false },
    showBudget: { type: Boolean, default: false },
    showScore: { type: Boolean, default: false }, // see section "Custom
    score"
    showTax: { type: Boolean, default: false }, // see section "Custom
   taxes"
    showPopOverCart: { type: Boolean, default: false }
  },
  // specific score configurations
  swapConfig: {
    // if swaps are shown at all
    showSwaps: { type: Boolean, default: false },
    // if swaps are shown immediatly or at the end
    showSwapEnd: { type: Boolean, default: false },
    // if there should be a consent popup once at the start of the
   configured swap type (either swaps at the end or immediately at the
   start
    showOptInStart: { type: Boolean, default: false },
    // if consent should be given before each swap option popup
    showOptInEachTime: { type: Boolean, default: false }
  },
  subjectOptions: {
    money: Number,
    restricted: Boolean
  },
  questionnaire: { type: Boolean, default: false }
}, { timestamps: true });
```

Listing 21: Treatment data model.

### 3.2.2 Treatment, and base information administration using scripts

If a high number of configurations need to be made, you can think about automating this by writing scripts that automate the task. Following are a few examples of administration scripts.

#### Base items

Here an example script is given, which details the way users may upload base items at scale. The item data is stored as a JSON-file, which holds a number of valid item instances. Only authenticated user can upload items, so be sure to provide the session, and session-signature strings. These can be extracted from the cookies tab of the domain under which the website is hosted. These need to be added to the request headers for authentication. The script listing uses python3. The items of item_data.json have the following format (see listing 22).

```
{
  "netPrice": Number,
  "currency": String,
  "vat": Number,
  "content": {
    "contentType": String,
    "amountInKG": Number,
    "displayAmount": String
  },
  "tags": [String],
  "name": String,
  "brand": String,
  "description": [
    {
      "header": String,
      "text": String
    }
  ],
  "nutritionalTable": {
    "kj": Number,
    "kcal": Number,
    "totalFat": Number,
    "saturatedFat": Number,
    "totalCarbohydrate": Number,
    "sugar": Number,
    "protein": Number,
    "salt": Number
  },
  "imagePath": "<path_to_image>"
}
```

Listing 22: Item data model in json file.

The script loads all items from item_data.json file, stores it to a list. Then it iterates over the data list, and checks if an image is available, if not it continues without uploading the item. After this, using requests.post the item data is uploaded. If the item is successfully uploaded the script then proceeds to upload the image to the created item. If all of this worked, it will proceed with the next item.

```
import json
import requests

# URLs
baseURL = '<BASE_URL/api>'
addItem = '/item'        # enpoint for creating items
addImage = '/add/image' # enpoint for saving image to item
# Get and modify treatment

# Information for api access and URLs
```

```python
cookies = {"express:sess.sig": "<YOUR_SESS:SIG_STRING>", "express:sess":
    "YOUR_SESSION_STRING"}
httpHeader = {
        'Content-Type': 'application/json'
        }

data = []
with open('item_data.json', 'r') as f:
    for row in json.load(f):
        data.append(row)

for row, ind in zip(data, range(len(data))):
    print(f'item num: {ind}, of {len(data)}')
    try:
        open(f'{row["imagePath"]}', 'rb')
    except:
        print('no picture')
        continue
    try:
        resp = requests.post(
                f'{baseURL}{addItem}',
                headers=httpHeader,
                cookies=cookies,
                json=row
                )
    except requests.exceptions.RequestException as e:
        print(e)
        break
    # if successfully created
    # add image to the item
    files = {'image': open(f'{row["imagePath"]}', 'rb')}
    try:
        respImage = requests.post(
                f'{baseURL}{addImage}',
                files=files,
                cookies=cookies,
                data={'itemID':f'{resp.json()["_id"]}'}
                )
    except requests.exceptions.RequestException as e:
        print(e)
        break
```

Listing 23: Base item upload script.

### 3.2.3 Trial configuration and execution

To be able to start a trial a treatment definition has to be present. The treatment definition by default has the "active" attribute set to false. To be able to generate subjects, and start recording data, set this attribute to true. This can be done on the admin landing page, navigate down to the treatment table, open the dropdown menu for the treatment you wish to enable, and press the menu item "Enable". After pressing this it should now present you with the option to disable it by displaying a button "Disable". When a treatment is disabled links referencing this treatment will be redirected to an info page, which says that the given treatment is inactive or otherwise not present. You can at any point disable, and re-enable it.

Trials can be conducted in a number of different ways, either in a controlled environment e.g. class room settings, or over the internet. Both are supported and do not entail feature reductions from choosing one over the other. An advantage of the controlled environment would be that you could host the application on the local area network, which might translate to less latency and better network connectivity. The decision will ultimately depend on your resources and scope of the experiment.

**Important:** In relation to this application it is important to keep in mind, that different physical devices, and e.g. browser-software used may influence the user experience provided by the tool. Especially the **view port** may influence the user experience. Variable device width may influence the number of items a subject can initially see on screen. Furthermore it has implications on the ease of use, as generally the less screen is available the more effort it is to navigate the shopping environment.

#### Trial route

To be able to reload the trial route during experiment execution the URL path represents the central information necessary to rebuild, and fetch the necessary data (see listing 24). The treatmentID is the reference id (_id) of the treatment that should be conducted. The subject id references the subject that should be used. This combination is used to reference and save the data that is being created by the subject during the treatment execution.

```
// route used for trial execution
<base_URL>/t/<treatmentID>/s/<subjectID>/...
```

Listing 24: Trial route composition.

### Manually start, and generate subject

For manually configuring a treatment you can navigate to <base_URL>/t. This will show a form where you first need select a valid treatmentID, and can then proceed to choose a specific subject, or automatically generate a subject for the treatment execution. After choosing a valid combination you will be redirected to the treatment start page, from which subjects may start the trial.

### Automatically generate subject and start

The manual process can be skipped by providing a link with additional query parameters. The query parameter is named "genSubject" by providing this with the parameter "yes" (see listing 25). This will skip all configuration steps, and redirect the participants to the products page. This is the recommended way if you do not want to reuse a subject for a different treatment.

```
// route used for automatic trial execution
<base_URL>/t/<treatmentID>?genSubject=yes
```

Listing 25: Automatically start and generate a subject.

### Automatically start and reuse a subject

For this you just need to combine the necessary information described in the trial route section (see listing 24). If a subjectID is given, and the subject has not yet finished the referenced treatment, the link will automatically redirect the subject to the products page of the referenced treatment.

### Custom questionnaire

Users of this application may configure custom questionnaires, that can be performed either before, after, or at both points of an experiment. This feature is not developed beyond a rudimentary implementation stage. For instance custom questionnaire may not be created through visual aids at the treatment administration screen. In its current implementation, questionnaires can be hard-coded into the application. This option requires programming knowledge as the templates, styling, and data-bindings would need to be implemented by hand.

The questionnaires that was configured for the project thesis is still in the source code. Its implementation can be observed in the components q1, and q2. Q1 referencing the questionnaire that is shown before accessing the shop, and q2 referencing the questionnaire after accessing the shop. These questionnaires are recorded in the trial data model, making it easy to map a specific subject to its questionnaire items.

**Important:** as q1, and q2 are hard-coded into the shop, they can not adaptively change corresponding to the different treatment definitions.

### 3.2.4 Trial data

The subject- and treatment ID are referenced at the top so the generated data can easily be assigned to treatment and subject. Started, and ended are timestamps which represent the time the subject has started, and ended the trial. Started is set when the trial data reference is first saved to the database. The end is set if the subject ends the trial by pressing the checkout button. Along with the end timestamp the finished flag is also set to true, this prohibits the accidental addition or modification of data after the trial has been ended. The routing array saves all route changes. By providing the origin, and destination routes the navigation path of the subject can be easily tracked. The final cart and transactions arrays represent the final shopping cart and all changes to the shopping cart, which was made by the subject (see listing 26). The use case specific data collection is described in the section use cases. The pagination array saves all page changes a subject makes, which items are visible on the page, the number of all items, the current page, the page size, and a timestamp when the pagination event occurred.

```
let TrialSchema = new Schema({
  treatmentID: String,
  subjectID: String,
  started: String,
  ended: String,
  owner: String,
  finished: { type: Boolean, default: false },
  data: {
    // routes visited and navigated
    routing: [
      {
        origin: String,
        destination: String,
        time: String
      }
    ],
    pagination: [
      {
        currentPage: Number,
        pageSize: Number,
        itemsOnPage: Array,
        numInTotal: Number,
        time: String
      }
    ],
    // final cart
    finalCart: [
      {
        itemID: String,
        amount: Number
      }
    ],
    // addition and substraction from shopping cart
    transaction: [
      {
        time: String,
        itemID: String,
        identifier: String,
        delta: Number
      }
    ],
    // swap information
    swaps: [
      {
        started: String,
        ended: String,
        originalItem: String,
        originalAmount: Number,
        resultItem: String,
        resultAmount: String,
        swapOptions: [String],
        success: Boolean
      }
    ],
    swapOpts: [
      {
        sourceItem: String,
        rememberMyAnswer: Boolean,
        result: Boolean
      }
    ],
    // information if description of label or score is viewd by subject
    infoViewed: [
      {
        started: String,
        ended: String,
        infoID: String
      }
    ],
    // filter actions the subject makes
    itemsFiltered: [
      {
        time: String,
        filter: Object
      }
    ]
  }, { timestamps: true });
```

Listing 26: Trial model schema.

Trial data can be retrieved from the back-end from the get-request endpoint <back-end_URL>/download/data/<treatment_ID>. This will return a list of all trial records generated in association with a treatment-id. The records are send in JSON-format. The following listing details a script that loads all trial records of a treatment and converts the records from JSON- to csv- or excel-format (see B).

## 3.3 Use Cases - Usage Examples

### 3.3.1 Change Displayed (Product-) Information & Shop-Functions Based on Treatment Specification

#### Changing the Filter and Sorting Mechanisms

There are several ways of changing the filter, and sorting mechanisms. For reference see the the section Base and custom filtering, and Sorting. Also see the section base item on niceness, an additional ordering mechanism provided by the base application.

#### Translate information: E.g. different labeling strategies

The tool offers several ways to translate, and enrich the displayed information. For labeling strategies see section "Base labels". In addition researchers are able to change any information about an item allocated to a treatment. You are able to change the displayed image of an item between treatments, offer more information, change content, nutritional, and all other information saved to an item, for more information see the section on base items.

### 3.3.2 Recommendations-Systems & - Agents

**Swaps: Offering consumers the opportunity to replace their usual food with a more sustainable alternative.**

The application supports the offering of swap options for food items at different intervention points.

1. When adding items into the shopping cart

2. When finishing the treatment by checking out

In addition to this you can show a opt-in popup, this can also be configured in one of two ways. Either the opt-in popup appears before each event which precedes a swap dialog, or once per subject (either at the start, or end of the treatment, based on when the swap options should be displayed.

For using this mechanic you first have to have a treatment with allocated items. Swap options are configured for each individual item, not for categories or other information. Swaps references are saved to the item object inside the "swaps" array. This array holds references to "item._id" attributes of items that should be shown when offering swaps.

**Important:** You can not upload base items with swap options directly referenced, as you need the "item._id" reference for adding a valid reference into the swaps array.

For configuring swap options on a treatment you need to navigate to the edit screen of the desired treatment (/admin -> treatment dropdown "edit"). Select the item you want to add swap options, and click edit on the dropdown menu. On the item edit card click the button "Add swap", this will open a dialog for adding items to the swap array. Select the items you want to add, and press the save button. After this, save the whole treatment definition by clicking the "Save" button on the bottom of the screen. **Note:** Saving a treatment may take a few seconds depending on the number of items allocated to it.

**Enabling displayment:** For enabling the display of swap options during a treatment you then have to set a flag on the treatment object. These flags can be set on the treatment edit screen in the section titled "Swap configuration options". To enable the displayment generally tick the "Show swaps" option. This will enable the base swap configuration, which show swaps to the participants when they add an item to their shopping cart. If you want to utilize the second intervention point strategy you have to tick the boy labeled "Show swaps at the end of treatment". With this the second intervention strategy is used. Here all swap dialogues are iterated when the subjects attempts to finish the treatment by hitting the check out button.

**Enabling Opt-In strategies:** The Opt-In strategies are also controlled by flags that are set on the treatment definition, directly beneath the swap deisplayment options. Here you may only choose one of the two strategies. The strategy "Opt-In once at the start" will have different behaviors when you choose the base mode of displaying swaps versus the show swaps at the end mode. The first prompts the subject before the treatment begins here he opts in or out of swap displayment in general. The second mode prompts the subject when clicking the checkout button, and before the swap dialogues are iterated here again he is prompted once to opt in or out of swap displayment.

**Important:** If you edit any aspect of the treatment definition you have to hit save at the bottom of the treatment edit screen, failing to do this will result in all the updates being lost on leaving the treatment edit screen.

### 3.3.3 Positioning, Arrangement & (Re-) Placement

**Product Arrangement: Consumers judge a product to be less expensive when exposed to a high price context (e.g. a specific higher animal welfare section)**

**Positioning: We find that a product's choice probability increases when presented on the first screen or located near focal items**

See section on "Base and custom filtering", and "TRANSLATE INFORMATION"

### 3.3.4 Partitioning & Bundling

- Partitioning of Options Into Groups
- Product Bundling: Product bundling could serve as a behavioral intervention to both lessen cognitive effort and increase the selection of healthful fruit and vegetable items.
- Substituting many small decision with a larger, but simper one (e.g. change from a „free choice“ to pre-defined boxes) Partitioned Shopping Carts: Partitioning a section of a shopping cart for fruits and vegetables (produce) may increase their sales

These effects you can achieve by either implementing custom filters, or you might think about adding different base attributes to items between treatments. The first alternative would enable the subject to more easily filter items based on some arbitrary attribute (see section on custom filters). The second alternative would give subjects the choice to limit items based on base attributes, which characterize an item (also see section on custom filters). **Note:** The limiting mechanic may be used in conjunction with filtering operations, as filtered items may further be limited by selecting base attributes.

In addition you may bundle a selection of products into a single product (single purchase decision) by creating one item that represents a product bundle. This references the bundling, and substitution use case.

### 3.3.5 Salience & Additional Information

**Additional Information: Increasing consumers knowledge and exposing themselves to key environmental cues (e.g. organic, free range, calorie information) can help an individual self-nudge ethical behavior**

You can offer different information based on treatments by altering the item data between treatments, adding label strategies, changing the picture associated to an item. Additional exposure, and salience may also be achieved by using the means described in the section Partitioning & Bundling.

### 3.3.6 (Smart) Disclosure & Feedback

**Real-Time Feedback: Real-time spending feedback has a diverging impact on spending depending on whether a person is budget constrained ("budget" shoppers) or not ("non-budget" shoppers).**

For this mechanic you can utilize different display mechanics offered in the base version of the application. You may think about adding spending feedback by visibly displaying a shopping cart sum, a fictional budget, and the remainder of the budget versus the shopping cart sum. These display options can be enabled under the display options section in the treatment configuration screen.

### 3.3.7 Economic Incentives

#### Taxes & Subsidies

Here you could think about changing the value added tax (vat) on individual items. In addition to this you may create custom taxes, which can then be allocated to individual items (see section 3.1.3). For subsidies you may think about adding negative taxes to items.

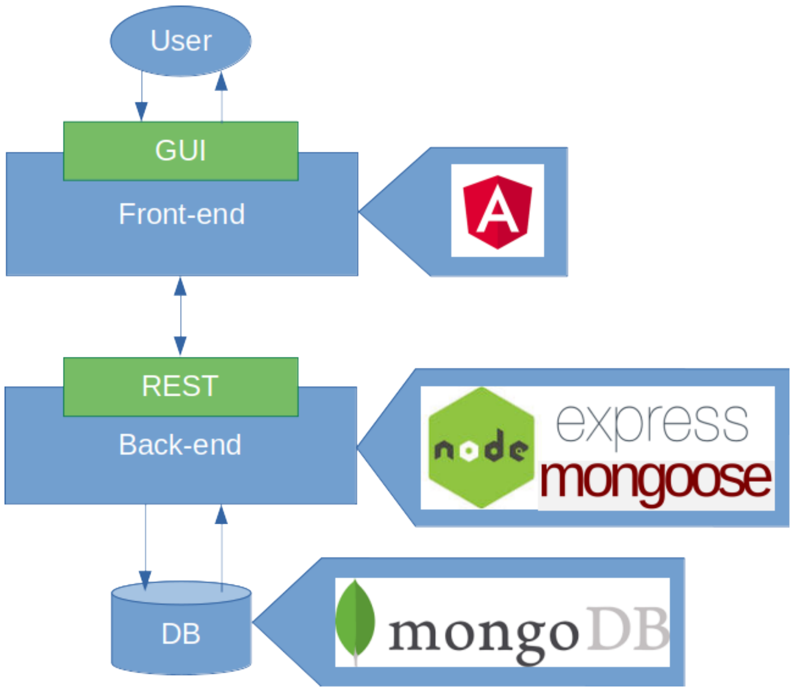


Figure 11: Technology stack of the application.

## 4 Developer Notes

In this section general implementation concepts, the structure, and development concepts will be discussed. The application devides into two separate applications. The Front- and Back-end applications. The front-end is a single page application that provides the graphical user interface for all user. The back-end application is structured as a Respresentational State Transfer (REST) Application Programming Interface (API). This offers the means to perform Create Read Update Delete (CRUD) operations on the underlying data. The technology stack can be observed in figure 11. This is a popular stack also known as the mean stack.

### 4.1 Front-end

The front-end application is implemented using the popular SPA framework Angular. For in-depth documentation, and developer instructions see the docs.

The front-end repository devides into three main feature modules. The trial module holds all the code necessary for all trial functionality of the tool. For instance the shopping view, services and functions performing, and recording the results of an experiment are contained in the trial module. The admin module then holds all the logic for treatment configuration, and base functionality needed for providing visual administration aids for users. The shared module holds code, and components that are used throughout different

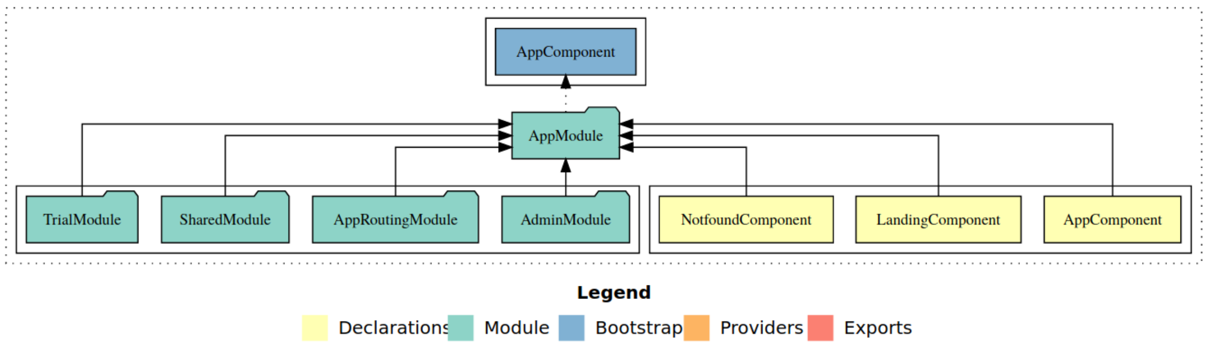


Figure 12: Modules and components of the app module.

modules. Understanding this structure is key. This structure makes it easy for developers to locate, and change specific aspects about the front-end application. All code associated with a given aspect can be easily found and isolated by traversing the directory structure. Changes to the trial experience, for instance the shop view, can be made in the trial module. These changes do not carry, and affect code in other modules.

The code is documented with doc-strings in the source code. In addition to this Compodoc is added to the code base. This tool provides a comprehensive, and visually appealing documentation representation. See listing 27 for the command with which to run, and serve the documentation. This command starts an HTTP-Server by default listening on localhost:8080. Navigate there to see full documentation.

```
# from the root folder of the front-end project
compodoc -p src/tsconfig.app.json -s
```

Listing 27: Prepare and serve Compodoc.

Additions to specific parts of the functionality of the tool should be made in their respective feature modules. If the modifications include data to be saved, and loaded to and from the back-end think about, at which point an existing data model could be extended, or implement the functionalities in a new sub-folder following the structure described in the next section. If any of these aspects need to be editable by use of GUI components, then these must be added in the admin module.

## 4.2 Back-end

The back-end application is based on a Node.js application, you can find the docs here. The server application it self is based on the framework express, and data storage is handled by mongoDB, which is a NOSQL data store. This implementation styles gives flexibility, and provides the means with which to rapidly develop, test, and deploy new functionality. The usage of the NOSQL data model means that their is no mismatch between the form of the data that is used in the front-, and back-end applications. This makes developing this tool even easier.

The code base of the back-end application is structured following the data models utilized. All CRUD operations, model, and general functions are contained in the associated folders. In-depth descriptions about the functions, and end-points are also provided through doc-strings directly in the source code.

### 4.2.1 Basic configuration

For basic configuration of the back-end application you can use environmental variables, or command-line arguments. The configuration options can be observed in file: <root_-folder>/src/config.js. Additionally you can configure these basic variables by using a .env file. For further information see convict, and dotenv. If you have different istances of these basic configurations for different environments you can also configure these in the <root_folder>/src/config/<environment>.json files.

### 4.2.2 Authenticating Users

For user authentication the application uses a custom username, and password strategy using the passport middleware. Passport is Express-compatible authentication middleware for Node.js applications. When a user successfully authenticates, the back-end sets a session, and session signature in the browsers cookies. They are secure and http only. Passport also handles the serialization, and deserialization of user specific data. On request end-points where user specific data is needed the passport middleware desirializes the session cookie to retrieve the user identifier. Based on this identifier all necessary information can be gathered. The user information is accessible on the request object (req.user), this is standard behavior of the passport middleware.

### 4.2.3 Image handling

When uploading an image to an item two separate images are created. One thumbnail image with the dimensions 175x175, and the original image. The images are not saved to disk, both images are saves to the mogodb database. This is done to prevent any complications during the deployment step. This means there is no direct image download link, images need to be queried by providing their database id.

Images that are uploaded in the context of label definitions are nested inside the label data model. This also means that images associated with labels are also not saved to disk, but into the database. To retrieve the picture you have to load the whole label definition.

# Appendices

## A Deployment script for server hosting

```sh
#!/usr/bin/sh
if [ $# -eq 0 ]
then
        echo "Missing options!"
        echo "(run $0 -h for help)"
        echo ""
        exit 0
fi
while getopts "hBFA" OPTION; do
        case $OPTION in
                b)
                        ECHO="true"
                        ;;
                h)
                        echo "Usage:"
                        echo "deploy.sh -B "
                        echo "deploy.sh -F "
      echo "deploy.sh -A "
                        echo ""
                        echo "\t-B\tto deploy both Frontend and API"
                        echo "\t-F\tto deploy only Frontend"
      echo "\t-A\tto deploy only API"
                        exit 0
                        ;;
    B)
      sudo systemctl stop nginx
      sudo systemctl stop api_store
      # For replaving the frontend
      sudo rm -rf /var/www/storefront
      sudo mv /home/sebastian/deployment/storefront /var/www/
      sudo chown sebastian:sebastian -R /var/www/storefront
      sudo systemctl start nginx
      # same for api
      rm -rf /opt/node/api_store
      mv /home/sebastian/deployment/api_store /opt/node/
      sudo chown sebastian:sebastian -R /opt/node/api_store
      echo "NODE_ENV=prod" >  /opt/node/api_store/.env
      echo "IP_ADDRESS=134.76.18.221" >> /opt/node/api_store/.env
      cd /opt/node/api_store && /usr/bin/npm install
      sudo systemctl start api_store
      exit 0
      ;;
    A)
      sudo systemctl stop api_store
      rm -rf /opt/node/api_store
                        mv /home/sebastian/deployment/api_store /opt/node/
                        sudo chown sebastian:sebastian -R /opt/node/api_store
                        echo "NODE_ENV=prod" >  /opt/node/api_store/.env
                        echo "IP_ADDRESS=134.76.18.221" >> /opt/node/api_store/.env
                        cd /opt/node/api_store && /usr/bin/npm install
                        sudo systemctl start api_store
                        exit 0
                        ;;
    F)
      sudo systemctl stop nginx
                        # For replaving the frontend
                        sudo rm -rf /var/www/storefront
                        sudo mv /home/sebastian/deployment/storefront /var/www/
                        sudo chown sebastian:sebastian -R /var/www/storefront
                        sudo systemctl start nginx
      exit 0
      ;;

        esac
done
```

## B Script for retrieving, and converting trial data

```python
import json
import requests
import pandas as pd
from benedict import benedict

# URLs
baseURL = 'https://vegs.codemuenster.eu/api'
treatmentID = '<treatment_ID>'
trialDataRoute = f'/download/data/{treatmentID}'
treatmentData = f'/t/{treatmentID}'
base_path = r'<root_dir_path>'
json_file = r'all_items.json' # file where all treatment items are kept

# Information for api access and URLs
cookies = {"express:sess.sig": "<YOUR_SESS:SIG_STRING>", "express:sess": "YOUR_SESSION_STRING"}

httpHeader = {
        'Content-Type': 'application/json'
        }
# function that checks if userAgent header is of mobile or desktop browser
def checkIfMobile(user_agent):
    reg_b = re.compile(r"(android|bb\\d+|meego).+mobile|avantgo|bada\\/|blackberry|blazer|compal|elaine|fennec|hiptop|iemobile|ip(hone|od)|iris|kindle|lge |maemo|midp|mmp|mobile.+firefox|netfront|opera m(ob|in)i|palm( os)?|phone|p(ixi|re)\\/|plucker|pocket|psp|series(4|6)0|symbian|treo|up\\.(browser|link)|vodafone|wap|windows ce|xda|xiino", re.I|re.M)
    reg_v = re.compile(r"1207|6310|6590|3gso|4thp|50[1-6]i|770s|802s|awa|abac|ac(er|oo|s\\-)|ai(ko|rn)|al(av|ca|co)|amoi|an(ex|ny|yw)|aptu|ar(ch|go)|as(te|us)|attw|au(di|\\-m|r |s )|avan|be(ck|ll|nq)|bi(lb|rd
    )|bl(ac|az)|br(e|v)w|bumb|bw\\-(n|u)|c55\\/|capi|ccwa|cdm\\-|cell|
    chtm|cldc|cmd\\-|co(mp|nd)|craw|da(it|ll|ng)|dbte|dc\\-s|devi|dica|
    dmob|do(c|p)o|ds(12|\\-d)|el(49|ai)|em(l2|ul)|er(ic|k0)|esl8|ez
    ([4-7]0|os|wa|ze)|fetc|fly(\\-|_)|g1 u|g560|gene|gf\\-5|g\\-mo|go(\\.
    w|od)|gr(ad|un)|haie|hcit|hd\\-(m|p|t)|hei\\-|hi(pt|ta)|hp( i|ip)|hs
    \\-c|ht(c(\\-| |_|a|g|p|s|t)|tp)|hu(aw|tc)|i\\-(20|go|ma)|i230|iac(
    |\\-|\\/)|ibro|idea|ig01|ikom|im1k|inno|ipaq|iris|ja(t|v)a|jbro|jemu|
    jigs|kddi|keji|kgt( |\\/)|klon|kpt |kwc\\-|kyo(c|k)|le(no|xi)|lg( g
    |\\/(k|l|u)|50|54|\\-[a-w])|libw|lynx|m1\\-w|m3ga|m50\\/|ma(te|ui|xo)
    |mc(01|21|ca)|m\\-cr|me(rc|ri)|mi(o8|oa|ts)|mmef|mo(01|02|bi|de|do|t
    (\\-| |o|v)|zz)|mt(50|p1|v )|mwbp|mywa|n10[0-2]|n20[2-3]|n30(0|2)|n50
    (0|2|5)|n7(0(0|1)|10)|ne((c|m)\\-|on|tf|wf|wg|wt)|nok(6|i)|nzph|o2im|
    op(ti|wv)|oran|owg1|p800|pan(a|d|t)|pdxg|pg(13|\\-([1-8]|c))|phil|
    pire|pl(ay|uc)|pn\\-2|po(ck|rt|se)|prox|psio|pt\\-g|qa\\-a|qc
    (07|12|21|32|60|\\-[2-7]|i\\-)|qtek|r380|r600|raks|rim9|ro(ve|zo)|s55
    \\/|sa(ge|ma|mm|ms|ny|va)|sc(01|h\\-|oo|p\\-)|sdk\\/|se(c(\\-|0|1)
    |47|mc|nd|ri)|sgh\\-|shar|sie(\\-|m)|sk\\-0|sl(45|id)|sm(al|ar|b3|it|
    t5)|so(ft|ny)|sp(01|h\\-|v\\-|v )|sy(01|mb)|t2(18|50)|t6(00|10|18)|ta
    (gt|lk)|tcl\\-|tdg\\-|tel(i|m)|tim\\-|t\\-mo|to(pl|sh)|ts(70|m\\-|m3|
    m5)|tx\\-9|up(\\.b|g1|si)|utst|v400|v750|veri|vi(rg|te)|vk
    (40|5[0-3]|\\-v)|vm40|voda|vulc|vx(52|53|60|61|70|80|81|83|85|98)|w3c
    (\\-| )|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas\\-|your|zeto|zte\\-
    ", re.I|re.M)
    b = reg_b.search(user_agent)
    v = reg_v.search(user_agent[0:4])
    if b or v:
        return True
    else:
        return False

def seperateBasedOnMobile(dataFrame):
    return {'mobile': dataFrame.loc[dataFrame['mobile'] == 1], '
    notMobile': dataFrame.loc[dataFrame['mobile'] != 1]}

def getWriter(file_name):
    return pd.ExcelWriter(f'{base_path}{file_name}.xlsx', engine='
    xlsxwriter')

# get Trial data
resp = requests.get(
        f'{baseURL}{trialDataRoute}',
        headers=httpHeader,
        cookies=cookies,
        )

# prepare dicts for additional data eg. match item attributes to final
    cart ...
# only get item data if not present
itemCatalog = {}
try:
    with open(f'{base_path}{json_file}', 'r') as infile:
        itemCatalog = json.load(infile)
except: pass

if not itemCatalog:
    print("Item data is being fetched")
    respShopItems = requests.get(
        f'{baseURL}{treatmentData}',
        headers=httpHeader,
        cookies=cookies
        )

    for item in respShopItems.json()['items']:
        # print(item['_id'])
        itemCatalog[item['_id']] = item

    with open(f'{base_path}{json_file}', 'w') as f:
        json.dump(itemCatalog, f)

"""
Extract general Data about subjects from trial data
"""

generalData = pd.DataFrame()
routingData = pd.DataFrame()
paginationData = pd.DataFrame()
finalCart = pd.DataFrame()
transactionData = pd.DataFrame()
swapData = pd.DataFrame()
swapOptData = pd.DataFrame()
infoViewed = pd.DataFrame()
itemsFilteredData = pd.DataFrame()

questionnaireItems = pd.DataFrame()


attributes = {
        "subject": 'subjectID',
        "device":'userAgentHeader',
        "deviceWidth":'deviceWidth',
        "deviceHeight":'deviceHeight',
        'age': 'questionnaire|personalInfo|age',
        'gender':'questionnaire|personalInfo|gender',
#         'location':'questionnaire|personalInfo|location',
        'occupation':'questionnaire|personalInfo|occupation',
        'education':'questionnaire|personalInfo|education',
        'housing':'questionnaire|personalInfo|housing',
        'foodPurchaseResp':'questionnaire|personalInfo|foodPurchaseResp'
    ,
        'shoppingFrequency':'questionnaire|personalInfo|
    shoppingFrequency',
        'income':'questionnaire|personalInfo|income',
        'expenditures':'questionnaire|personalInfo|expenditures',
        'maritalStatus':'questionnaire|personalInfo|maritalStatus',
        'email': 'questionnaire|personalInfo|email',
        'finished': 'questionnaire|personalInfo|finished',
        'started': 'started',
        'ended':'ended'
        }

for subject, ind in zip(resp.json(), range(len(resp.json()))):
    # if not subject['finished']: continue
    subjectID = subject['subjectID']

    # general data about the trial
    # if not subject['questionnaire']['personalInfo']: continue
```

```python
    print(ind)
    data = benedict(subject, keypath_separator='|')
    temp = {}
    for key, value in attributes.items():
        # Ecept any error and place value of None in df
        try:
            if data[value] == 'none':
                temp[key] = pd.np.nan
            else:
                temp[key] = data[value]
        except Exception as error:
            print(error)
            temp[key] = None
    temp['mobile'] = checkIfMobile(temp['device'])
    generalData = generalData.append(temp, ignore_index=True)

    # for routing data
    for routing in subject['data']['routing']:
        routing['subject'] = subjectID
        routing['mobile'] = temp['mobile']
        routingData = routingData.append(routing, ignore_index=True)

    # for pagination data
    for pagination in subject['data']['pagination']:
        pagination['subject'] = subjectID
        pagination['mobile'] = temp['mobile']
        paginationData = paginationData.append(pagination, ignore_index=True)

    # for final cart
    for finalC in subject['data']['finalCart']:
        cartItem = dict(finalC)
        cartItem['subject'] = subjectID
        cartItem['mobile'] = temp['mobile']
        ## add item data with mapping
        cartItem.update(itemCatalog[cartItem['itemID']])
        del cartItem['_id']
        del cartItem['oldID']
        # unest nutritional table
        try:
            del cartItem['nutritionalTable']
            print(itemCatalog[cartItem['itemID']]['nutritionalTable'])
            cartItem.update(itemCatalog[cartItem['itemID']]['nutritionalTable'])
        except Exception as error:
            print(error)
        # unnest content description
        try:
            del cartItem['content']
            cartItem.update(itemCatalog[cartItem['itemID']]['content'])
        except Exception as error:
            print(error)
        # unnest baseAttributes
        if len(itemCatalog[cartItem['itemID']]['baseAttributes']) > 0:
            for attr in itemCatalog[cartItem['itemID']]['baseAttributes']:
                cartItem[attr] = 1

        finalCart = finalCart.append(cartItem, ignore_index=True)

    # for transactions
    for trans in subject['data']['transaction']:
        trans['subject'] = subjectID
        trans['mobile'] = temp['mobile']
        ## add item data with mapping

        transactionData = transactionData.append(trans, ignore_index=True)

    # for swap data
    for swap in subject['data']['swaps']:
        swap['subject'] = subjectID
        swapData = swapData.append(swap, ignore_index=True)

    # for swap opts selected
    for swapOpt in subject['data']['swapOpts']:
        swapOpt['subject'] = subjectID
        swapOpt['mobile'] = temp['mobile']
        swapData = swapData.append(swapOpt, ignore_index=True)

    # for info viewed
    for info in subject['data']['infoViewed']:
        info['subject'] = subjectID
        info['mobile'] = temp['mobile']
        infoViewed = infoViewed.append(info, ignore_index=True)

    # filter operations
    for itemsfiltered in subject['data']['itemsFiltered']:
        itemsfiltered['subject'] = subjectID
        itemsfiltered['mobile'] = temp['mobile']
        itemsFilteredData = itemsFilteredData.append(itemsfiltered, ignore_index=True)

    # Questionnaire items
    quest = {}
    quest['subject'] = subjectID
    quest['mobile'] = temp['mobile']
    quest['finished'] = subject['finished']
    try:
        quest.update(subject['questionnaire']['questions1'])
        quest.update(subject['questionnaire']['questions2'])
    except Exception as error:
        print(error)
    questionnaireItems = questionnaireItems.append(quest, ignore_index=True)

## Basic data
generalData.fillna(value=pd.np.nan, inplace=True)
generalData['ended'].replace(pd.np.nan, '', inplace=True)
num_total = len(generalData)
num_mobile = len(generalData.loc[generalData['mobile'] == 1])
num_notMobile = num_total - num_mobile
num_finished = len(generalData.loc[generalData['ended'] != ''])
num_finished_mobile = len(generalData.loc[(generalData['ended'] != '') & (generalData['mobile'] == 1)])
num_finished_notMobile = num_finished - num_finished_mobile
```

```

"""
Write to Excel file, with sheets
"""
# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = getWriter('basic_data')
b_data = {
        'number_total': num_total,
        'number_finished': num_finished,
        'num_mobile': num_mobile,
        'num_notMobile': num_notMobile,
        'num_finished_mobile': num_finished_mobile,
        'num_finished_notMobile': num_finished_notMobile
        }
df_1 = pd.DataFrame(b_data, index=[0])
df_1.to_excel(writer, sheet_name='Basic_Data')
# export to excel
generalData.to_excel(writer, sheet_name='General_Data')
routingData.to_excel(writer, sheet_name='Routing_Data')
paginationData.to_excel(writer, sheet_name='Pagination_Data')
finalCart.to_excel(writer, sheet_name='Final_Cart')
transactionData.to_excel(writer, sheet_name='Transaction_Data')
swapData.to_excel(writer, sheet_name='Swap_Data')
swapOptData.to_excel(writer, sheet_name='SwapOpt_Data')
infoViewed.to_excel(writer, sheet_name='Info_Viewed')
itemsFilteredData.to_excel(writer, sheet_name='Items_Filtered')
questionnaireItems.to_excel(writer, sheet_name='Questionnaire_Items')
# Close the Pandas Excel writer and output the Excel file.
writer.save()
```