THE VOS USER GUIDE

Content

1. Introduction

What is the VOS, and why use it? The VOS is an open-source, modular, and highly customizable virtual online supermarket application. The application allows researchers to easily design, implement, and conduct experiments in a realistic online shopping environment. The application contains two core elements: the Shop View and the Admin View.

The Shop View depicts the interface experiment subjects see and make decisions in. It was designed to emulate the store design and functions (e.g., navigation tools) of a realistic online supermarket. In the tool's base version, the front-end design is very neutral to prevent any existing customer relationship bias towards the store's design elements.

The Admin View is a visual administration interface (VAI) that enables researchers to control and modify research conditions and configure and implement different experimental treatments. It provides the means for managing the product database and various modification options that allow researchers to determine which information or functions the Shop View presents. In the following, we refer to these pre-defined modification options as Use Cases because researchers can use and modify them to create their own experimental treatments. For instance, without any mandatory knowledge in programming, Use Cases enable researchers to adjust food prices, implement different labeling strategies, or change the arrangement of food items. Hence, these options can serve as a starting point for research projects on the most common grocery interventions. All Use Cases included in the tools' base version will be introduced in Chapter 2. This chapter also includes more detailed guidance on the practical use. Nevertheless, researchers familiar with programming are not limited to basic features. Instead, they can extend or change any aspect of the tool's visual or functional implementation by altering the program's code.

The VOS is built upon widely known and well-supported frameworks. JavaScript, as the overall implementation language, makes developing and customizing this tool easy. In the following sections, more information about the technology and programming of the VOS application is provided (Chapter 1). There, you will also find all the installation, setup, and usage instructions needed to run, develop, and deploy VOS successfully. [1]

2. Information about Technology and Programming

2.1 Application Development Frameworks

Our application is built upon two development frameworks to decrease development time, reduce maintenance cost, and quality assurance reasons. First, for the back-end, Node.js is used to build a web-based service, to provide an Application Programming Interface (API) for handling requests, and to run Create Read Update Delete (CRUD) operations on the database. Second, the storefront Angular version 8 is used for implementing the web front-end. Both frameworks are based on JavaScript. The accessible and popular of all building blocks ensure longevity and support for the application as a whole. Moreover, there is a greater active developer base than for other less-used frameworks, which improves the possibilities for further development. However, as mentioned above, the knowledge of JavaScript and these frameworks are only necessary if researchers want to create advanced use cases for the context of online-shopping research. Instead, the creation of multiple experiments and treatment variations does not require any extended programming knowledge.

2.2 Model View Controller (MVC)

The implementation style of our application follows the Model View Controller (MVC) concept. Working with MVC in web application development is different from conventional application development. The architecture has to be partitioned between the client and the server-side. A web applications view is always handled by the client-side, but the model and controller can be partitioned in various ways between the

[1] **Please note:** 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 managed over SSH.

client and server. Hence, a compelling architecture would be relying exclusively on the server to refresh the client's screen. In this case, the model and the view-generating logic for the view on the client's browser would reside entirely on the server. Moreover, the controller would partially reside on the client (detecting user interaction) but mostly reside on the server (code that updates the state of the model's business objects based on HTTP request). This describes a thin-client approach with the advantages of decreasing the client machine's performance demand and providing greater security, performance, and data consistency for the application. Web application frameworks that reflect this paradigm are Django and ASP.NET.

The other extreme is maintaining the bulk of the application on the client-side (fat-client approach). This means that the model mostly resides on the client-side, but the database remains on the server-side. In particular, the view is exclusively implemented on the client-side, and the controller mostly resides there. This provides a more seamless and interactive experience through fewer load times, minimizing the need to make server calls. Frameworks that support this style of partitioning are AngularJS, EmberJS, and JavaScriptMVC.

2.3 Front-End Programming

The views are built directly with HTML5 in conjunction with style sheets written in SCSS (Sassy CSS) to provide a visually appealing and user-friendly experience. This combination is supported by most browsers natively, which means a wide range of devices can be supported. In this way, a responsive front-end web design is provided, which is consistent between devices and browsers. Also, this combination allows for the separation of presentation and content, the reduction of repetitive code, flexibility, control of the presentation, and sharing of formats between views. The web-view also utilizes Bootstrap and Angular Material, which are CSS libraries that offer standardized web-content styling and component options. Therefore, developers benefit from the ease of use and accessibility of these frameworks for building visually engaging views while also centrally determining custom styles and layouts.

The front-end codebase is designed and implemented to deliver an accessible experience both to users and developers. Features and design elements are designed to encapsulate specific functionality or business logic. Hence, the code is partitioned into feature modules, which house the logic for model, view, and controller, represented by the components contained. Utilizing this implementation logic makes it easy to tear apart distinct design elements, use case implementations, and feature sets for subsequent expansion and development. The application's distinct design elements are thus contained in one sub-folder easily recognizable (see Figure 1).

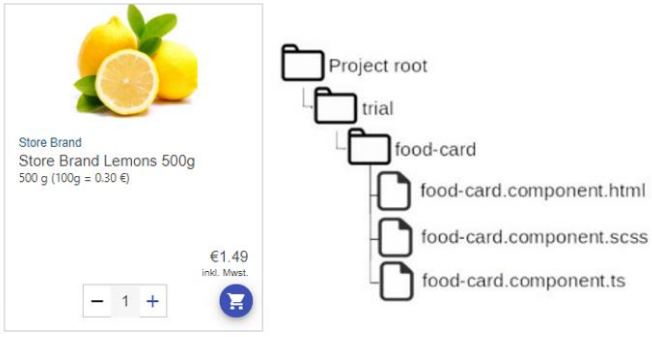

Figure 1. Example of component structure with design element "food card" on the left and associated folder structure representing the code on the right.

Changing the appearance or behavior of such a design feature would mean locating the associated component and altering the source files. These component definitions are divided into (i) the template (*.html file) which handles the display elements, (ii) the (*.sccs file) which defines the styles for this component, and (iii) the (*.ts file), which handles the data CRUD, data binding, and event handling of the specific component. In addition to this, the implementation of our application follows a strict separation of control. Functionality like data handling, CRUD operations, event recording is centralized and separated from component view logic. Injectable services offer reusable access to functions, which are generally

consumed by multiple components. Furthermore, the services are descriptively modeled to offer all functionality connected with a specific data model. For example, CRUD operations connected to the shopping cart component are combined into a shopping-cart.services.ts file. If any view component needs access to the specific functions and data of that topic, it has to inject the shared instance of the service into its constructor, thereby gaining access. This offers organized, reusable, and easy access to all operations needed throughout view components. Even if any implementation details change, these changes need only be applied in one file.

2.4 Back-End Programming

The back-end is a Representational State Transfer (REST) API. This exposes the data, functions, and facilitates the interaction between the database and the front-end application. Moreover, it exposes the endpoints that respond to client requests in a predictable manner. The web services are stateless, as they do not maintain the state of each client application accessing the web-service, rather offering predefined sets of stateless operations. This allows it to remain independent of the front-end application, meaning that these web services may serve different client applications and interact without using the front-end application. This independence offers the advantage that researchers are not limited to using the visual treatment edit interface to interact with and change treatment aspects or items. Scripts can be written that automate treatment creation, modification, and data analysis tasks (for example, see appendix C).

As mentioned above, the back-end is based on the application development frameworks Node.js, an event-driven JavaScript runtime environment that works outside of the browser. This allows for continuous utilization of JavaScript in both application areas and reduces entry barrier for developers, as only knowledge of one programming language is required.

However, Using JavaScript for the REST API does not incur performance decreases, as could be generally expected. Node.js is built on the libraries V8 and libuv, and these are responsible for partly converting the JavaScript code to C++ code and thereby combining the ease of use attributed to JavaScript and the high performance attributed to C++. In addition to this, it is highly scalable without threading, instead of utilizing a simplified model of event-driven programming with callbacks to signal the completion of a task. However, this single-threaded approach means that the application cannot scale vertically, which means that simply adding computing power to a given system will not directly translate to an increase in application performance. Despite this, it is still capable of scaling by running a number of concurrent instances of the same application within one cluster manager (cluster mode). This distributes the workload among the available application instances. A production-ready and open-source load balancing software for Node.js applications is already available free of charge; see, for example, PM2.

Moreover, the repository structure of the back-end application is modeled to promote easy access and clarity. Thereby, the sub-folder structure represents the data structure utilized throughout the project. For instance, a shop-item and all its associated CRUD operations are contained in one sub-folder (*see Figure 2*): Models (*.model.js files) enforce the document structure, routes (*.route.js files) define the request endpoints, and with that access to all the operations that can be performed on the data objects. Additionally, functions collect reusable logic used throughout the route definitions. Middleware functions (*.middleware.js files) provide necessary state information to the otherwise stateless endpoints. This makes the project accessible for further development because the application structure can be deduced from the repository structure. Effects from changing aspects of the data structure are contained in this sub-folder and do not affect the overall application. This makes it easy to add or change and customize the functions, data models, and route specifications implemented in the base application.

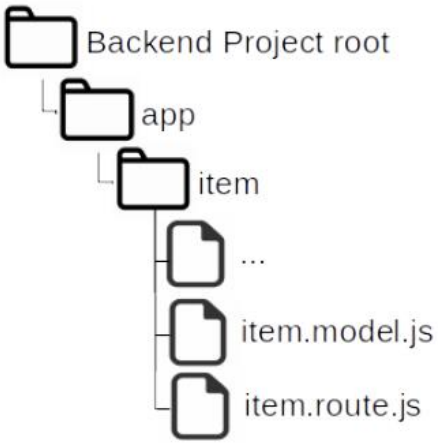

Figure 1. Sample repository structure as used in the back-end.

2.5 Availability

The code is developed using Git, a source-code versioning system. This encourages good backup and versioning practices and allows developers to synchronize files across computers, develop collaboratively, manage separate branches, and merge synchronization conflicts. The project is open for other researchers to join, collaborate, or just download the source code on GitHub.

3. Installation

3.1 Prerequisites

The application's front-end and its back-end are based on npm and node. Therefore, first install node, which should automatically install npm as well. If the version commands do not return the version number of either npm or node, it did not install correctly. Hence, for installation, 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's 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. 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 applications. The dependencies for both applications are managed using npm. To install the necessary dependencies, change the directory into the root folders for both applications, and run the following command (see listing 4.1).

```
npm install
```

Listing 4.1. Set up a script for npm projects.

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 4.2). With this, the dependencies are updated, and any security issues are fixed when they have known fixes.

```
npm update
npm audit fix
```

Listing 4.2. Set up a script for npm projects.

To be able to develop and compile the front-end application, the Angular CLI must be installed globally. Use the following command to do so (see listing 5).

```
Install angular -cli globally
npm install -g @angular / cli
```

Listing 5. Installation of angular-cli.

3.2 Local hosting and development

To locally host the front-end and back-end applications, input the following commands inside each project folder's root folder (see listing 6).

```
# 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
```

Listing 6. Hosting the front-end and back-end applications.

Both applications are hosted in "development mode," after changing source files, the applications are recompiled, and the server is automatically restarted. The above commands act as aliases for the following

commands (see listing 7). They are specified in the respective package. json file in the root folder of each application. This makes starting the applications in development mode easy.

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

Listing 7. Configurations of scripts array in package.json.

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

3.3 Running the application on a server

For deploying the applications to a server, many different approaches can be taken. In the following sections, configuration steps required for deploying both the front- and back-end applications on one server are explained. The server is running on Ubuntu 18.04.3 LTS. The hardware configuration of the server is variable and is dependent on your project's scope.

Before deploying the application onto your server, make sure you have all software detailed in section Installation installed on your server (except the angular-cli, only needed for development).

3.3.1 Front-End

Before you can deploy the application to a server, you have to compile the front-end project's source code. Run the following command in the root folder of the application (see Listing 8).

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

Listing 8. 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, you can use the following command (see Listing 9).

```
scp -r <path to root dir >/ dist / storefront / <user >@< server IP >:~/
deployment
```

Listing 9. 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 provided 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 10.

```
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 10. Nginx server definition.

To configure the server's default configuration file, run the following command (see Listing 11). 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 11. Nginx configures sites available.

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 Let's Encrypt. To automatically set ssl_certificates up, you can run the following command (see Listing 12). 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 12. Certbot installation and nginx configuration.

You need to have a domain name for the 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 13).

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

Listing 13. Restart nginx and check status.

3.3.2 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. I use the following command to copy the files to the server (see Listing 14).

```
# 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 14. Upload back-end files to the server.

This copies all the application files, excluding the node_modules folder, to the deployment sub-folder in your server user's 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 application's required dependencies (see Listing 15).

```
# 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 15. Configuration file for systemctl.

Again, at this point, you might want to update and audit fix the dependencies of the project. For testing, you might want to run the command line application to see if any errors arise. It is suggested for easier management to create and register 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 listing 16 and listing 17.

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

Listing 16. Configuration file for systemctl.

```
# 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 17. Bash commands for enabling and reloading the systemctl services.

3.3.3 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 18). 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 to be able to send verification emails. The configuration file can be found in <root_folder>/src/app/mailer.js. The email confirmation is handled by sending a JSON Web Token (JWT) and a confirmation link to the user's email address. After the user clicks the link provided, the back-end confirms the validity, and sets the user object's confirmed flag to true. If the confirmed flag is true, the new user can log-in, or they are prohibited from logging-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 18. User data when created.

3. Configuration Options

3.1 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, and 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 you want to use throughout your various treatment configurations.

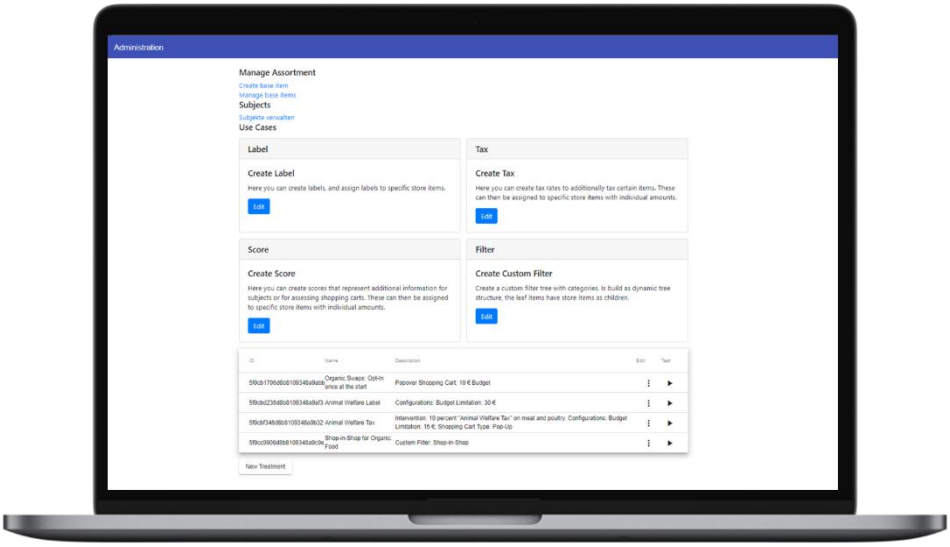

Figure 1. Admin view - Base use case configuration and treatment addition.

For adding base items, use the link "Create base item" and fill out all the information you want to add to the item.[2] 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 for the item you want to reconfigure. Via the dropdown, you can choose to edit, delete, or add an image age or change the image associated with the selected item (see Figure 2).[3]

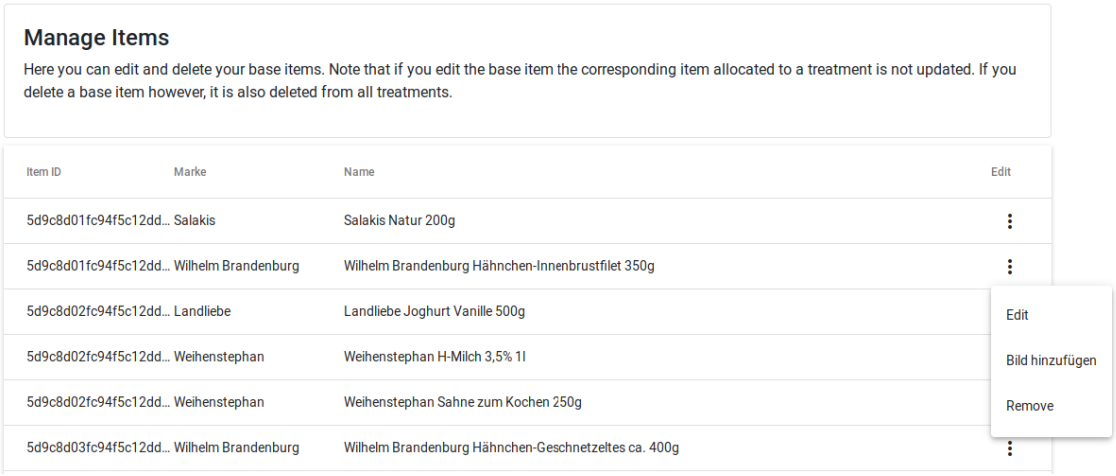

Figure 2. Admin view - Configure base items.

[2] Please note: Base items can only immediately reference swap options when the externalID references them. Referencing them with their database ID at this stage is impossible because the items are deep copied into the treatment specifications and have a new _id attribute after allocating them to a treatment. 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 front-end application display method).

[3] Please note: 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.

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 Figure 4).

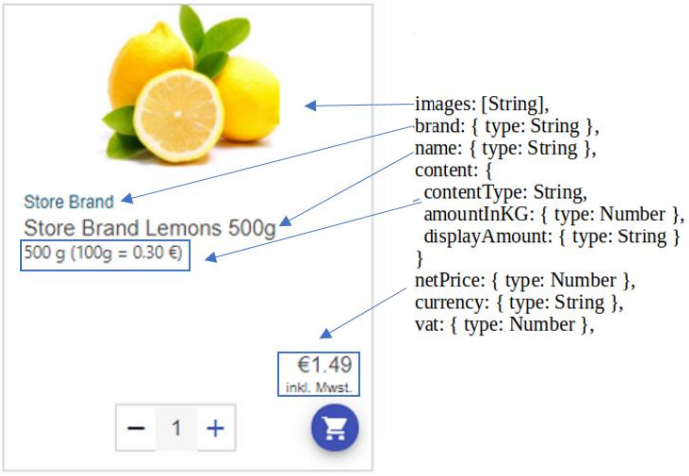

Figure 3. Shop view - Example mapping of item model data to the food-card component.

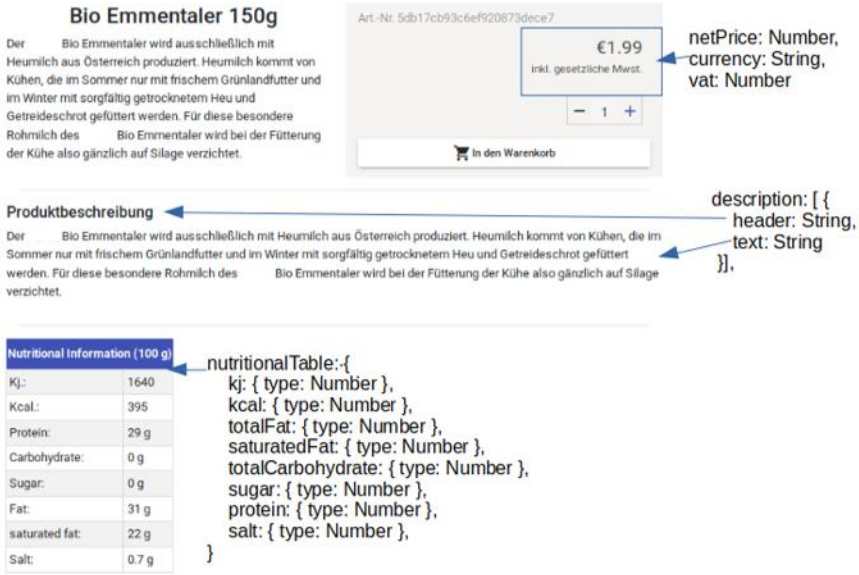

Figure 4. Shop view - Example mapping of item model data to the food-details component.

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 19).

```
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 19. Item model schema.

3.1.1 Base Attributes

The "baseAttribute" is an array in which you can specify arbitrary attributes for each item. Based on these subjects can limit the selection of items based on the specified attributes. For instance, base attributes like "Organic," "Lactose-free," and "Free-range" could be added. These names are then also displayed in the item limiting function in the shop view.[4]

[4] Please note: Different spellings of the same attribute will result in an additional limiting option.

3.1.2 Tags

The tag array configures the base filter of the application (see the base filter section). The tag array is considered from the first to the 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.

Please note: the order in which these tags are appended in this array is important for generating the parent, child node filter tree.

3.1.3 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 is 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 always to be displayed first or closer to the viewpoint of the subjects.

3.2 Base Labels

For creating label strategies ,you can select the label card in the base configuration section of the admin view, and 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 the view. This view also enables you to edit, delete, or add an image to an already created label (see Figure 5).

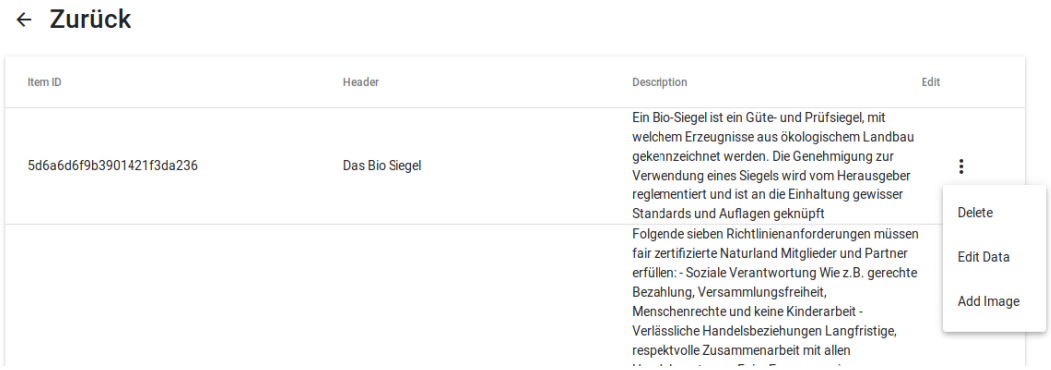

Figure 5. Label create view - Configure label definitions.

The following listing shows the data model of a label object (see Listing 20). The header represents the label name that is being displayed. The description is a text that a subject can view during a trial. The associated image is directly stored inside the label model definition.[5]

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

Listing 20. Label data model.

[5] Please note: 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."

3.3 Custom taxes

For creating custom tax definitions, you can select the tax card in the base configuration section of the admin view, and this will navigate to <base_URL>/admin/taxCreate. The following listing details the taxes data model (see Listing 21).

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

Listing 21. Tax data model.

The short description will be displayed on the food card and on the food details view (see Figure 6).

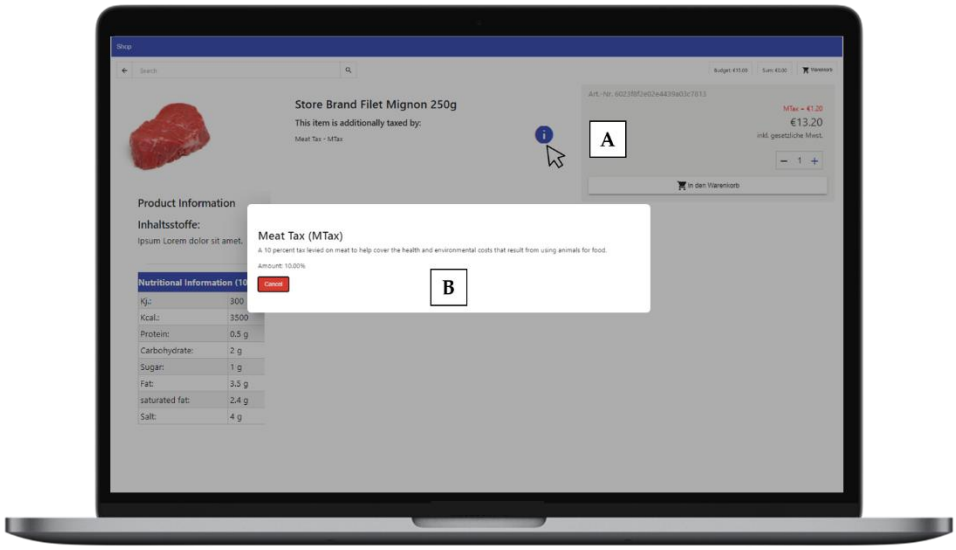

Figure 6. Shop view - (A) tax display in food details component, (B) tax information dialog, for showing the description of the additional tax.

3.4 Custom score

To create custom score definitions, you can select the score card in the base configuration section of the admin view, which will navigate to <base_URL>/admin/labelCreate. The following listing details the taxes data model (see Listing 22).

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

Listing 22. Score data model.

The scores can be created 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). An item's actual score has to be configured on an item where the tax is allocated.

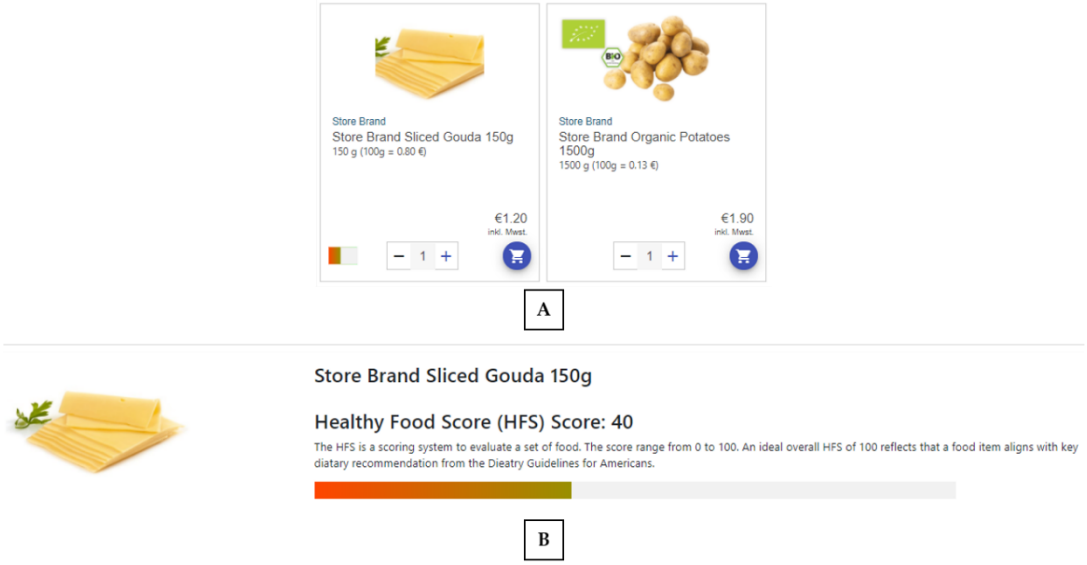

Figure 7. Shop view - (A) Score display on food-card component, (B) score display in food-details component

3.5 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 19). 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 with the trait "organic." 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, whereas the tag filter is single select only.

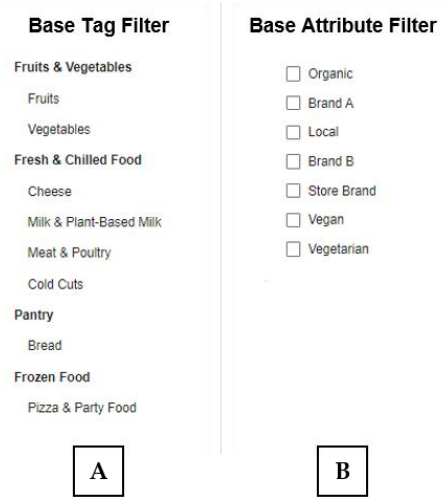

Figure 8. Shop view - (A) Base tag filter, (B) base attribute filter

For creating custom filter definitions, you can select the filter card in the base configuration section of the admin view, and this will navigate to <base_URL>/admin/filterCreate. Custom filters differ from the base filter variants in that they can be created by hand. You may create any custom filter tree of your choosing. After creating a custom filter tree definition, you need to add this filter tree definition to the treatment of your choice. After this, you can add items from your treatment definition to your custom filter tree's leaf nodes. This way, there are no limits to the customizability and depth of your filter tree definition. Figure 9 and Figure 10 illustrate the stages you have to perform to add a custom filter to your treatment definition.[6]

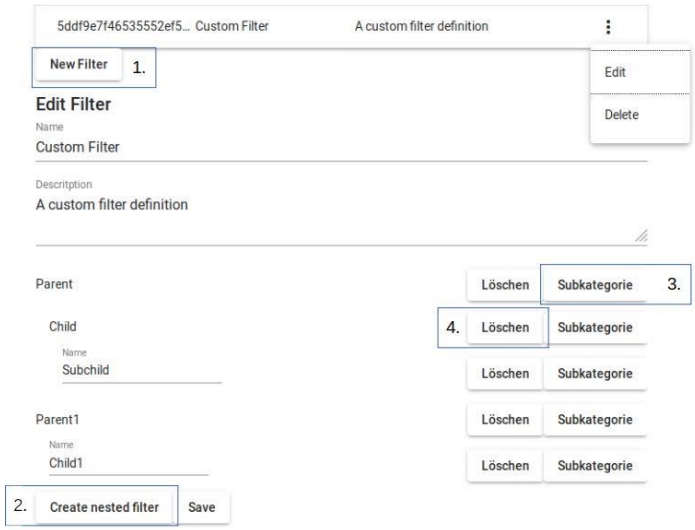

Figure 9. Admin view - 1. Create new filter, 2. create parent node, 3. create child node, 4. delete referenced node.

[6] Please note: 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 items and the filter are being displayed. This is partially true as the product's niceness value or any other ordering mechanism will rearrange the items as they are displayed when filtered. This is a side-effect of the fact that this information is pushed onto an array on the item/treatment level.

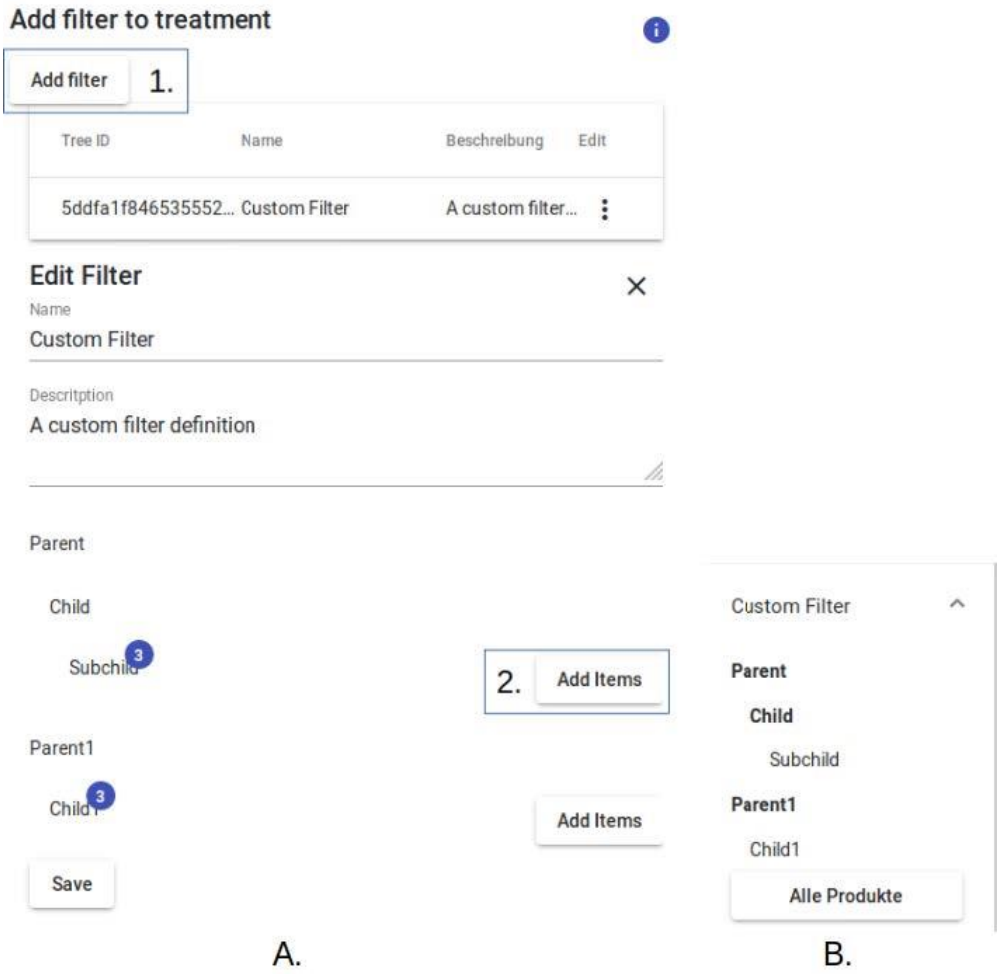

Figure 10. Admin view – (A) 1. Add an existing filter to the treatment, 2. add treatment items to the node leaf of the custom filter tree. (B) Custom filter displayed on the left of the item grid.

3.6 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.

3.7 Swap Options

The VOS supports the offering of swap options[7] for food items at different intervention points: (i) when adding items into the shopping cart and (ii) when finishing the treatment by checking out. Additionally, you can show an opt-in popup, which can also be configured in two ways. Either the opt-in popup appears before each event that 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 tool, 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

[7] Swap options are an intervention that offers consumers the opportunity to replace a selected food item, for instance, with a healthier or more sustainable one.

inside the "swaps" array. This array holds references to "item._id" attributes of items that should be shown when offering swaps.[8]

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 to 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.[9]

3.7.1 Enabling the Display

For enabling the display of swap options during treatment, you then have to set a flag on the treatment object. These flags can be set on the treatment edit screen. To enable the display, tick the "Show swaps" option. This will enable the base swap configuration, which shows 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 box 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 attempt to finish the treatment by hitting the checkout button.

3.7.2 Enabling Opt-In strategies

The Opt-In strategies are also controlled by flags set on the treatment definition, directly beneath the swap display options. Here you may only choose one of the two strategies. The strategy "Opt-In once at the start" will have different behaviors when choosing the base mode of displaying swaps versus the show swaps at the end mode. The first prompts the subject before the treatment begins. Here, they opt-in or out of the swap display in general. The second mode prompts the subject when clicking the checkout button, and before the swap dialogues are iterated, here again, they are prompted once to opt-in or out of swap display.[10]

4. Treatment administration

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, an email confirmation is required. If this has all successfully finished, navigate to the <base_URL>/auth/login screen and log in using the newly generated user. If there are any errors, observe the back-end logs or see the returned HTTP response errors. Only logged-in users 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 creating a 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 the custom filters you added as a custom filter (see section 3.5). 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 contains options for configuring the shopping experience, like a maximum budget for each shopper and if this budget is restrictive.

[8] Please note: Important: You cannot upload base items with swap options directly referenced because you need the "item._id" reference for adding a valid reference into the swaps array.

[9] Please note: Saving a treatment may take a few seconds depending on the number of items allocated.

[10] Please note: 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 losing all the updates when leaving the treatment edit screen.

4.2 Creating new Treatments

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.

4.3 Modifying existing treatments

For modifying an existing treatment, navigate to the desired treatment and expand the dropdown options menu. Once here, click on "Edit." This will redirect you to the treatment edit view. Click "Delete" if you want to delete the specific treatment definition.[11]

4.4 Testing Treatments in a Sandbox Environment

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, meaning 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.

4.5 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 & Listing 24). The item array contains full copies of every base item which has been allocated to the specific treatment. This means that you can freely change any aspect of an item allocated to a treatment without affecting the base items. The "showOptions" attribute contains configuration flags that configure switch on or off certain shop design elements. The subject options represent the object for configuring game options.

```
let TreatmentSchema = new Schema ({
owner : String ,
name : String ,
description : String ,
active : { type : Boolean , default : false },
items : [ Item ],
// featuredItems : [ String ],
filters : [ Tree ],
showOptions : {
1numOfItems : { 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 3.1.4
showTax : { type : Boolean , default : false }, // see section 3.1.3
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
```

[11] Please note: Clicking the “Delete” button will immediately delete the treatment without asking for consent.

```
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 23. Treatment data model.

```
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
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
```

Listing 24. Base item upload script.

4.7 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.

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 several valid item instances. Only authenticated users 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 Python 3.0. The items of item_data.json have the following format (see Listing 25).

```
{
" netPrice ": Number ,
" currency ": String ,
"vat": Number ,
" content ": {
" contentType ": String ,
" amountInKG ": Number ,
" displayAmount ": String
},
" tags ": [ String ],
" name ": String ,
" brand ": String ,
" description ": [
1{
" 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 25. Item data model in json file.

The script loads all items from item_data.json file and stores them in a list. 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 works, it will proceed with the next item.

5. 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 "Disable" button. 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 several different ways, either in a controlled environment, e.g., classroom 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 the scope of the experiment.[12]

5.1 Trial route

In order 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 26). 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 created by the subject during the treatment execution.

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

Listing 26. Trial route composition.

5.2 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 to select a valid treatmentID and 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.

5.3 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 27). This will skip

[12] Please note: Concerning 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 viewport 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 because generally, the less screen is available, the more effort it is to navigate the shopping environment.

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 27. Automatically start and generate a subject.

5.3 Automatically start and reuse a subject

For this, you need to combine the necessary information described in the trial route section (see Listing 26). If a subjectID is given, and the subject has not yet finished the referenced treatment, the link will automatically redirect the subject to the referenced treatment's products page.

5.4 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, a 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.

5.5 Trial Data

The subject- and treatment ID are referenced at the top, so the generated data can easily be assigned to the treatment and subject. *Started* and *Ended* are timestamps that represent the time the subject has begun and completed 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 28). The pagination array saves all page changes the 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. The use of case-specific data collection is described in section 6.

```
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 28. Trial model schema.

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

6. Developer Notes

In this section, general implementation concepts, structure, and development concepts will be discussed. The application divides into two separate applications: the Front- and the Back-end. The front-end is a single-page application that provides a graphical user interface for all users. The back-end application is structured as a Representational 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 popular stack is also known as the mean stack.

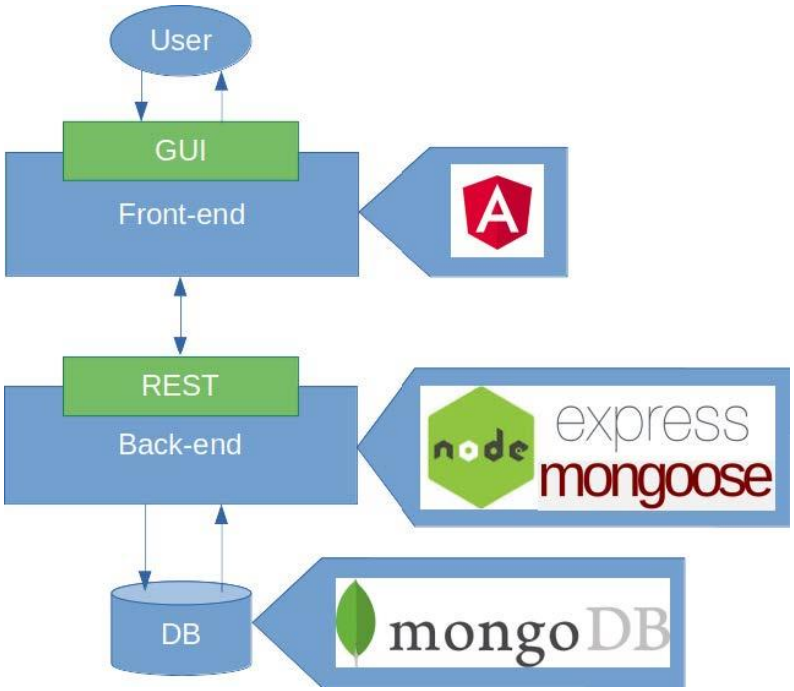

Figure 11. Technology stack of the application.

6.1 Front-End

The front-end repository divides 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 an experiment's results are contained in the trial module. The admin module then holds all the logic for treatment configuration and base functionality to provide visual administration aids for users. The shared module contains code and components that are used throughout different modules. Understanding this structure is key. This structure makes it easy for developers to locate and change specific aspects of 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 over and affect code in other modules (see figure 12).

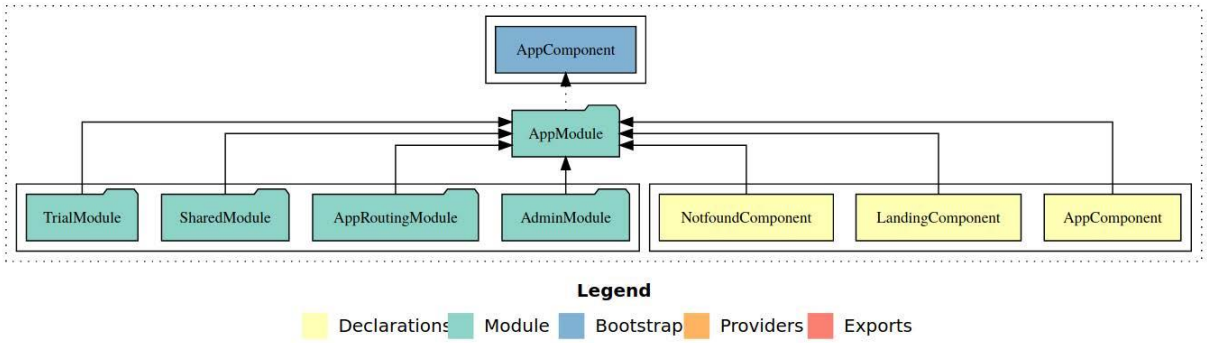

Figure 12. Modules and components of the app module.

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 29 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 the full documentation.

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

Listing 29. 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. You can also implement the functionalities in a new sub-folder following the structure described in the next section. If any of these aspects need to be edited by GUI components, then these must be added in the admin module.

6.2 Back-end

The back-end application is based on a Node.js application. The server application itself is based on the framework Express, and data storage is handled by MongoDB, which is a NoSQL data store. This implementation style gives flexibility and provides the means to develop, test, and deploy new functionality rapidly. The NoSQL data model usage means that there is no mismatch between the form of the data used in the front and back-end applications. This makes developing this tool even easier. The codebase 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 docstrings directly in the source code.

6.2.1 Basic configuration

For the basic configuration of the back-end application, you can use environment variables or command-line arguments. The configuration options can be observed in the 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 instances of these basic configurations for different environments, you can also configure these in the <root_folder>/src/config/<environment>.json files.

6.2.2 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 browser's cookies. They are secure and HTTP only. Passport also handles the serialization and deserialization of user-specific data. The passport middleware deserializes the session cookie to retrieve the user identifier on request end-points where user-specific data is needed. Based on this identifier, all necessary information can be

gathered. The user information is accessible on the request object (req.user). This is the standard behavior of the passport middleware.

6.2.3 Image handling

When uploading an image to an item, two separate images are created—one thumbnail image with 175x175 dimensions and the original image. The images are not saved to a disk, and both images are saved to the MongoDB 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 a disk but into the database. To retrieve the picture, you have to load the whole label definition.

Appendix

Appendix A - Deployment script for server hosting

```
#!/ 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
```

Appendix B - Script for retrieving and converting trial data

```
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|3 gso |4 thp |50[1 -6] i |770 s |802
s|a
wa| 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 ']: continuesubjectID = subject ['subjectID ']

# general data about the trial
```

```
# if not subject [’ questionnaire ’][’ personalInfo ’]: continue
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 ’] = subjectIDswapData = 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 ()
```