eNutri - Development of a flexible web-tool for studies utilizing Food Frequency Questionnaires

Thomas Bellebaum
Advisor: M.Sc. Julia Blaurock
Supervisor: Prof. Dr. Kurt Gedrich

September 24, 2020

Contents

1 Introduction

When studying the nutritional behaviour of individuals and populations, there are several methods for studies ranging from diaries written by participants to regular calls. Food Frequency Questionaires (FFQs) represent one such method of particular interest due to their time efficiency and low cost. Being essentially just forms asking the participant in e.g. a multiple choice style about the frequency and typical portion sizes when consuming certain food items, they can constitute an easy way of assessing the intake of certain nutrients in a highly automatable way.

Previous studies have shown that personalized dietary recommendations are more effective than general dietary recommendations (`https://academic.oup.com/ije/article/46/2/578/2622850`). One application to deliver personalized dietary recommendations based on an individual's diet is eNutri, developed by Rodrigo Zenun. Before this project, it could (with some tweaking and addition of resources like images and details to what was called "Healthy Eating Score" in code) be deployed to Google's Firebase Services.

With Great Britain, the country where some of the servers providing Firebase are located, leaving the EU, and concerns about information security and privacy gaining importance among the population, it would be beneficial to be able to deploy such an FFQ-Webapp to dedicated servers outside typical cloud infrastructure.

This is where this project comes in. The goal, simply put, is to take the existing project that is eNutri and make it work with at least one other backend that can be fully controlled by the research institution.

Another goal with this project will be to make it customizable to people without programming knowledge. Customizations should include being able to change messages shown by the application, alter the items shown in the FFQ (which was technically possible in the original project, but not from within the app), or change the criteria for which participants get accepted algorithmically into the study.

This document is organized as follows: Section 2 gives an overview of the project eNutri as it was given to me. Section 3 describes the main changes done to the project. The technical documentation of the backend and some other components can be found in section 4. Section 5 is targeted at those planning to use the reworked app for development or studies and describes the basic usage from an administrator's point of view. And finally section 6 summarizes the project and gives an outlook towards future work.

2 The original project

The project described in this document is based on eNutri, a project initiated by Rodrigo Zenun as part of a PHD thesis. eNutri was capable of managing user accounts, identified by some mail address, conducting three Food Frequency Questionaires at fixed intervals and showing the results to the users in various details. eNutri 2.0 was developed within the Quisper Project (Quality Information Services and Dietary Advice for Personalized Nutrition in Europe) which aims to investigate the effectiveness of personalized dietary recommendations compared to general dietary recommendations.

Participants in studies using eNutri were devided into two groups: After assessment of nutritional behaviour, physical activity and anthropometric data, the *web*-group was shown detailed information about (for instance) their optimal weight range, a Healthy Eating Score (Details of its implementation were sadly not included with the code) measuring how healthy one participant's nutrition was considered, as well as suggestions on how to improve their score. The *control*-group however was shown only basic information like an indication if their weight was too low or too high, and no HES-related details.

2.1 General Structure

eNutri is a single-page web application written in JavaScript and using the AngularJS framework. It consists of a main HTML page, a CSS document, and several JavaScript documents, all of which are included in the main HTML page. Dependencies of the code base are served partially from the site itself and partially as external dependencies. Additionally AngularJS fetches several smaller HTML-parts from the webserver to be embedded in the document dynamically.

The main HTML file defines the general layout of the app, consisting mainly of a blue toolbar at the top, a menu to the left and a large free space filling the rest of the page. This space is then filled using Angular-Route using one of the sites subpages, each consisting of an HTML template and an Angular Controller. Subpages include:

- *home*: A welcome page containing only some text and a button linking to the login page.
- *login*: A simple form to log in or sign up.
- *account*: A simple form to alter one's password. Also contains some text like the registered mail address. Only accessible when logged in.
- ffq: A page containing every task the user has to do when participating in the study. For details see the section below. Only accessible when logged in.
- *reports*: This page shows the reports after the first FFQ. Only accessible when logged in.

Figure 1: A excerpt from the original code. Firebase-related code is highlighted in gray.

- *research*: Targeted at the researchers and never shown to other users, this page shows all user data and allows for easy conversion to CSV. Only accessible when logged in.

The log in restriction is enforced when the page loads by manipulation of the $*routeProvider*, though the comments on where this manipulation takes place are not accurate. This practise, while arguably cleaner, has some drawbacks including not allowing to do work in the background while data is being fetched.

2.2 Firebase

To store any data, be it configuration data or user data, eNutri uses the proprietary Firebase service by Google, available to JavaScript via the AngularFire library. Features of this app include automatic syncs of Angular-managed values with the Firebase storage, preventing the need for manual saves throughout the code.

References to Firebase are littered throughout the code, defining both the control flow as well as most non-hardcoded values. Any isolation of this dependency is missing. See also figure 1

Values seem to be saved almost exclusively as strings or numbers, even where other types of data would be more desirable. E.g. the value *screeningAccepted*, which is supposed to indicate whether the user is eligible to partcipate in the study, could be a simple boolean value, i.e. $True$ or $False$. Instead it is represented as a string (hopefully) equaling either "yes" or "no". Needless to say, such practices make the code very susceptible to bugs and security vulnerabilities. It also makes the code harder to review and maintain, as it is not clear whether a possible third value exists in some bizarre place. Other boolean information, like *screeningComplete*, is encoded in the bare existence of a piece of information with that name.

The data is accessed and altered in tree-like structures (See figure 2 for common terminology throughout this document), aquired and synchronized through

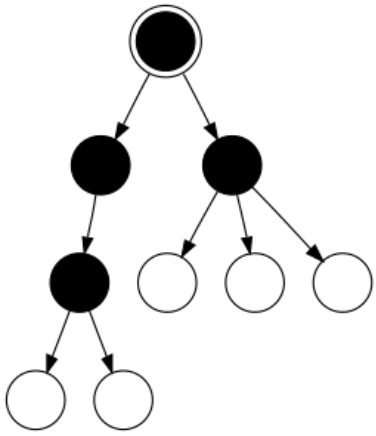

Figure 2: A rooted tree data structure. The encircled node is the root, inner nodes are black and leaf nodes are white. Arrows depict links, i.e. having access to the parent node (higher up), we can request access to the child (lower down) iff. there is a link between them). Data is stored at the leaf nodes. A subtree rooted at some node consists of that node, all of its children, grandchildren, and so on.

AngularFire. Within this tree, children of a node can be accessed via either Strings or Numbers, therefore providing a uniform interface for both arrays and JSON-objects.

2.3 FFQ subpage

The FFQ subpage has a lot of functions, which I had to reverse engineer because I didn't have a working copy at the time. It provides some HTML and JavaScript for:

- A consent form which is shown to the users repeatedly until they accept it
- Two forms for aquiring basic personal details, like a name, age, where they first heard of the study and various illnesses and dietary requirements, together with some pop-ups asking the user to specify details
- A form to enter one's current weight
- A short questionaire about how much exercise the user does, allowing to specify several sports, if applicable
- The FFQ itself. Every question consists of specifying how often a given piece of food is eaten and (if ever) selecting from a total of three typical portion sizes.

- Various partially static, partially generated messages
- A usability survey and
- A feedback questionaire, each allowing to further specify some details
- Code for controlling the flow of actions and the various parts.

As one can see, this page is the heart of the entire application, containing almost all relevant contents.

Once all relevant informations are retrieved, the current action is determined as depicted in figure 3.

2.4 Administration

Configuring the original application is virtually impossible without at least basic knowledge of either some programming language or data interchange format. Most data is hardcoded. This includes e.g. the time between FFQs or most messages not related to the web-, or controlreport. They are however specified in several languages and one language was chosen at runtime.

Some configuration options however, like the messages for the reports, are fetched from Firebase and thus could be changed even during a live study. Needed for this is some external interface like another control program or a configuration webpage.

Configuration from within the application is only possible from the research page, and even there the options are limited to accepting or rejecting participants who have been filtered out by the screening process.

3 Description of Changes made

3.1 General Changes

Most time went into reverse engineering the entire application and getting a minimal example to run. This alone required either a working instance to tinker with (which I did not have mainly due to a lack of a Firebase Account), or a complete retructuring of the code and the blind removal of anything related to Firebase or the AngularFire API.

I went for the latter solution. To not break any functionality, I decided on an alternative backend, which was Parse Server. The original code was littered with references to Firebase-specific code.

After reverse engineering the main functionalities used, I began working on a Parse equivalent to AngularFire, though it became quickly apparent this was an unreasonable amount of work for the task at hand, and any foreign projects to do something similar I could find online had been abandoned.

Luckily, by removing only a few key features like automatic synchronization, I was able to (over many iterations) define a relatively simple API, which could, if implemented carefully, be reimplemented without (that) much work for any

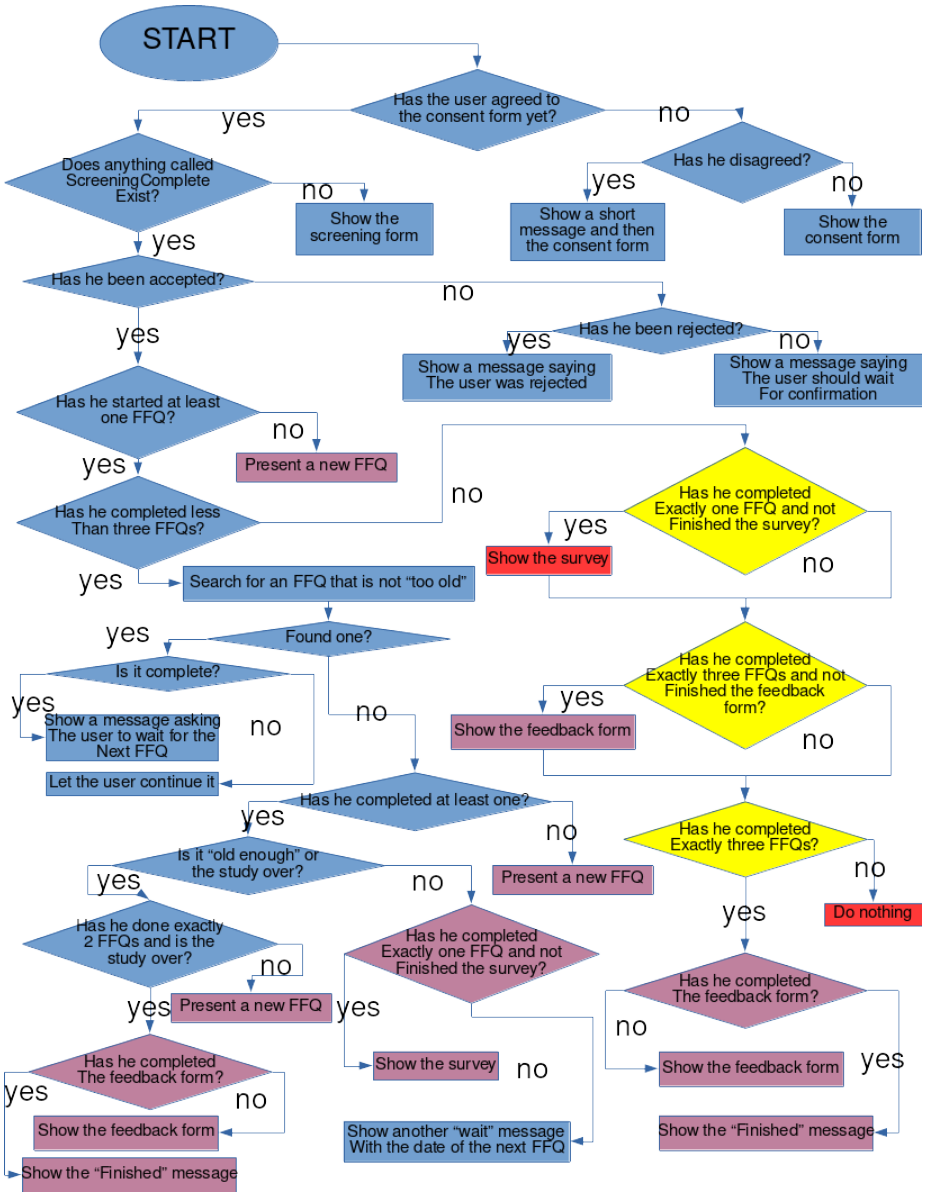

Figure 3: The control flow graph of the original function determining the action to take currently. Unreachable statements are depicted in red, (partially) redundant ones in yellow, and purple ones represent code duplications.

storage backend. This provides a better isolation of external projects and allows for future infrastructure changes without reverse engineering the code all over again.

However, some aspects of the existing program had to be changed, most notably:

- Three-Way-Binding: A feature of AngularFire providing automatic synchronization of data between the client and the backend was not implemented in the backend API. While this is theoretially possible with Parse Server, and would make the code cleaner sometimes, it requires additional connection tracking, does not allow to define atomic changes, thus can corrupt the data in case of a power outage, and is an unreasonable amount of work for a task which can be handled by a simple call to a "save" function. Notably, this does mean that the client will not immediately respond to configuration changes on the server, which is not a concern for this version of eNutri or any static tool for FFQs, but might become important in the future, should database resets be implemented one day.

- State resolution: When loading the page, some information about the configuration or the user must be retrieved from the server before the page is ready to be rendered. E.g. the page needs to know whether the user is currently logged in. In the original project this was done within the routeProvider, thus happening before the page was rendered. Now the page is rendered empty once and then gets filled with content once all necessary data has arrived, by a callback sent to the backend API. Noticable changes should be minimal for the user.

3.2 Database Layout

As mentioned, one goal with the major architecture changes coming with switching the backend storage was to provide a simple API to separate the project from the backend. To achieve this task, I decided to retain the tree-like data structure for users while simplifying the interface. Listed below are some design decisions that went into this:

- Subtree saves: Having to save data explicitly comes with the decision of which data to save with a single call. Parse Server provides the Option of cascading saves, which means that upon a requested save on one node, nested objects (children) are saved recursively. This does have its limitations, discussed below. While cascading saves could be disabled, saving leaf nodes is not a feature provided by Parse. This is due to the fact that what would be leaf nodes are considered key-value pairs belonging to their parent. Additionally, saving single objects would not provide the benefit of enabling larger atomic changes as discussed above. The other alternative might have been to save the entire tree with a single save. However, for backends that do not track changes, this could lead to high bandwidth,

thus deterring mobile users. All things considered, saving a subtree at a time seemed like a reasonable balance.

- No arrays near the root: In the original project it was possible to address children by an index, thus effectively implementing an array. Some backends however are less flexible and unable to provide similar functionality. For this reason I decided to try to omitt this functionality until it is really needed. One downside of this approach is that currently the layout of the database would have to be changed to allow for more than three FFQs.

  To describe the alternatives in Parse: Parse Server does provide one-to-many associations via relations. These, along with some index saved in the other relation endpoints, could be used to allow for arbitrary amounts of FFQs. However, this would make the implementation more complicated, as saves cannot cascade through relations and neither do fetches. Additionally, the code would become less maintainable with every instance of such an association if we were to check whether we need to save additional specified objects, unless we automated this, which would in turn raise the requirements on the backend to include the possiblity of listing all children of an object for tree-traversal. In some cases, this could again render atomic saves impossible. Another alternative would have been to save either entire objects or pointers to objects in an actual array. Both would have the same disadvantages while the former would also render subtree-saves more expensive in terms of bandwidth, since subtree-saves could only be emulated in Parse.

- Redundant data: Especially between FFQs and Reports, some data should be equivalent once an FFQ is over. This includes for instance the baecke or nutrients subtrees. (See section 4.3.1 for details.) These could have been shared within Parse, thus saving on both storage space and the need to copy data over. Doing this would however break the tree-like structure and the app would maybe in the future need to be aware of this. In order to ensure maintainability, the tree structure is preserved and some subtrees just happen to carry equivalent data.

- Unlogical structure: The original project had some questionable design decisions when it came to the tree layout. E.g. the SUS score was saved twice. Once in the survey node and once in the state node. Also the FFQ node contained a lot of information about the user's environment, which is barely FFQ-related. To minimize changes to the functionality of the code while changing the backend, the structure was largely retained. However, a future project may very well change the structure.

3.3 FFQ subpage

The original code for determining the current action (See figure 3) had some problems making it nearly unmaintainable. Firstly, the control flow graph had some branches which by logical deduction could not be executed. For instance,

to reach the "do nothing" statement, one would need to negate both having completed less than three FFQs and exactly three FFQs, in an application which never requires more than three FFQs. These branches are typically a sign for repeated changes to a small part of the code without any long term planning.

Second and related to the first issue, there were redundant queries. If we already know that the user has completed three FFQs, we do not need to check for this again. The problem here is that future changes to e.g. the way we represent the number of completed FFQs need to be addressed in these unnessasary checks as well, making this seemingly simple task more complex.

Third, a lot of code is written out several times. This has mainly the same problems as the second issue and one can observe this effect in small differences between the clones, which (mostly) do not actually show any effect.

And last but not least, when writing actual code, these nested if-then-else-statements ("branching nodes") make the code very unreadable in any text editor, even with code collapsing or similar functionality.

With these changes in mind, I addressed problems 1 to 3 with a revised control flow depicted in figure 4.

This graph is however not functionally equvalent to the original one. The main change is:

- The survey is now shown after finishing all FFQs, not after the first one.

The fourth problem I countered by splitting the functionality within the code into functions, one for each of the if-statements along the main "yes-line". Each such function either handles the rest of the decision process, if it falls into its functionality, or returns with an indication that the next decision function should take over. This way actions can, for testing purposes, be omitted or even rearranged.

To proceed to the next action, the page now reloads itself. This ensures a clean environment and no dependencies between e.g. the screening and the FFQ. It also makes the code a lot cleaner.

3.4 Admin Panel

The original project did not contain any method to customize the FFQ and any routines from the page itself. Therefore I created an admin panel, which allows among other things:

- Changing basic parameters such as the time between FFQs.
- Managing food items and portion sizes.
- Managing nutrients for any food item.
- Customizing a lot of messages. (The code itself was even designed with support for multiple languages in mind, though the UI does not currently reflect this, as the language changing methods would not cover hard-coded strings anyway.)

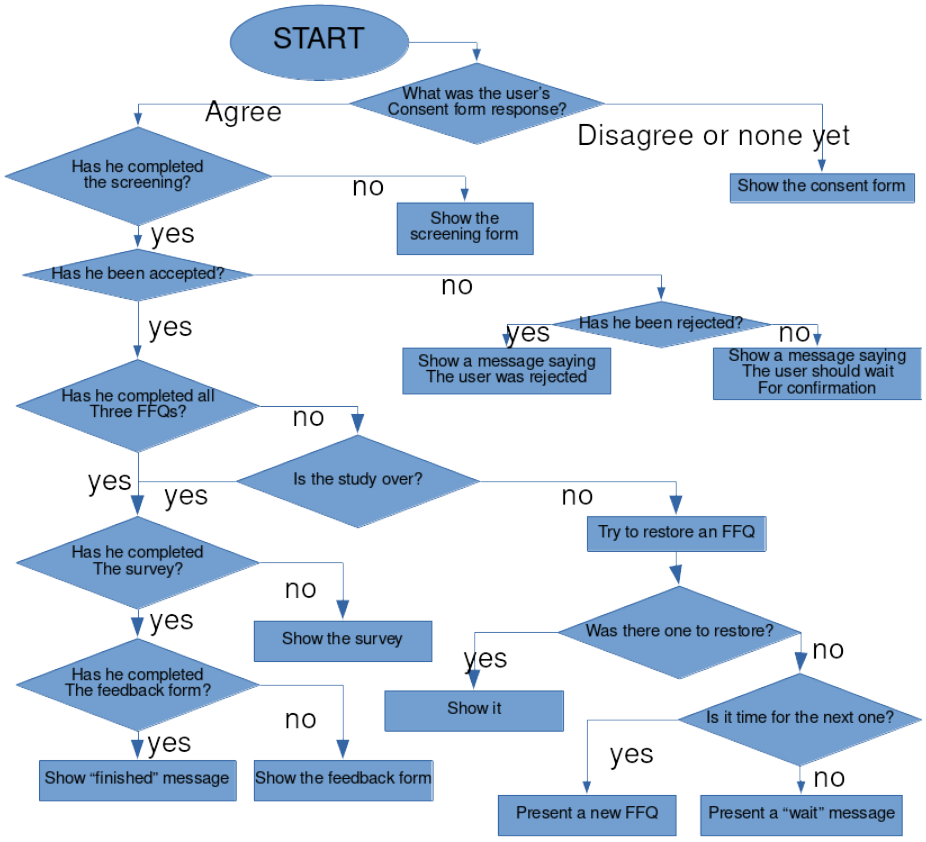

Figure 4: The control flow graph of the new function determining the action to take currently.

- Customiation of the user selection process, using information gathered during the screening.
- Saving any changes using the Parse Master Key for authorization.
- Importing and Exporting of any Configuration to JSON for easy re-configuration and out-of-band manipulation.

Default values are not supplied directly, as this would have to be set up server-side. Instead, I provided a tool for manipulation and generation of importable JSON strings. The tool allows for:

- Generation of an importstring using default values.
- Resetting of individual configuration categories (such as the user selection process or the list of food items.
- Importing Nutritional values from a BLS-formatted xlsx file.

As automatic password resets would require some sort of mail agent, which I did not have access to, an administrative tool for password resets, requiring authorization via the master key, is also provided.

Both tools can be accessed through the Admin Panel's front page.

3.5 Install System

One argument in favor of Firebase is the easy deployment of applications. Since we are not using Firebase anymore, and instead have to provide a database, Parse Server, optionally the Parse Dashboard, and an HTTP Server ourselves, I have prvided a Bash script for setting these things up with one command on a linux-based system.

The script uses Docker, a common containerization utility, to set up and configure a container for each required service.

A second script is provided to remove the containers and any created volumes.

For details, please review the below documentation.

4 Documentation

4.1 Install System

The install system, consisting of all files named *setup.sh*, *init.sh* and *cleanup.sh*, is a set of bash scripts initializing the docker containers running the actual project. Only the root scripts (in the project's root directory) are meant to be run directly. Other setup scripts will be invoked by this root script and may make use of the environment variables set in the root script. Where this is the case, the corresponding files will contain a comment near the top explaining its

requirements. Where noted, scripts may also depend on the existence of docker containers already set up by other scripts.

What follows is a brief description of the actions the install system currently performs, in order of execution, and the requirements for future changes. If any part of the script (such as the Dashboard in a production setting) is unrequired, the invocation of the corresponding script may simply be commented out.

Setup:

- *setup.sh*: This is the only script which should be called directly in a normal installation. It sets up three environment variables:
  - $eNutri\_WEB\_URL$: This variable is used for accessing Parse Server by the dashboard and the $parse - server/init.sh$ script and should reflect the url where the httpd docker container will be accessible right after its creation. A future implementation might not need this variable anymore and instead set up the database from within the server.
  - $eNutri\_MASTER\_KEY$: Access to Parse Server can be limited in several ways, though for maintenance or debug purposes, one might wish to circumvent these restrictions. E.g. we might want to be able to view the database contents from Parse Dashboard or change other user's password or settings for the FFQ. For these purposes Parse Server uses a "Master Key", which acts as a password to authorize any action the user wants to take. For this reason, the key is randomly generated in a cryptographically secure way via a call to $/dev/random$.
  - $eNutri\_APP\_ID$: To allow for multiple apps sharing the same underlying database, Parse Server indexes applications via a unique ID. The IDs here are generated randomly via a call to $/dev/urandom$, to minimize the chance of random collisions. This ID, though chosen randomly, is not required to be cryptographically secure. To minimize the chance of blocking calls, this is initialized after the master key.

  The script then communicates those variables to the user and procedes to calling the other scripts.

- $parse-server/mongodb/setup.sh$: Called immediately $parse-server/setup.sh$, this script downloads and runs the latest MongoDB docker image from Docker Hub. The well-known Non-relational database is used as a backend for Parse Server and only needed if no instance is already present.

- $parse - server/setup.sh$: While we could just use the image present on Docker Hub for Parse Server, we do want to add some additional configuration options, the main one being disabling $allowClientClassCreation$, which would allow adversaries to use our database as free storage in any way they would like. While this restriction only requires data to be of

a certain shape, it only really ensures (together with class specific restrictions) that any tampering would eventually come to light within the application. Any actual validation of user data would have to be done in some cloud code script in the server.

Parse Server is traditionally configured via a JSON formatted config file. The script, after initializing the database via a call to *mongodb/setup.sh*, generates a config file, asks docker to build a new image using a static Dockerfile copying the configuration, then runs a new container based on that image and cleans up the config file.

If one chooses to use an existing instance of MongoDB, the steps would be as follows:

- Remove or comment out the call to the MongoDB setup script.
- Remove or comment out the $--link$ option in the *docker run* command.
- Replace the $databaseURI$ field in the configuration generation with the existing instance. Parse Server also provides ways to specify authentication methods in the configuration. For this, please refer to the Parse Server documentation.

- *httpd/setup.sh*: This script sets up a container running Apache's well-known web server HTTPD. It does this by generating a configuration file, then starting the container, and finally copying the config file into the container and removes any unneeded files. Additionally, it communicates the App ID to the client via a generated $config.js$ file accessible from the client.

  If you wish to use a preexisting instance of Parse Server, you may specify it in this file.

  Note that the documents for the webserver are linked to the *httpd/app/* folder. This is useful for debugging. If you prefer to have all docker volumes in one place, copying these files might be an alternative for you.

- $parse - dashboard/setup.sh$: This script sets up a container for Parse Dashboard. It is mainly useful for debugging purposes and may be commented out in a production setting.

- $parse - server/init.sh$: Until now, the database has not been filled with any data. This script initializes the database structure and setting reasonable restrictions on any objects within. Most notably, it prevents the client from expanding the layout and storing arbitrary contents in the database. It then inserts the first user, *admin@admin.com*, and grants him privileges allowing him to see all data in the database.

  A future implementation might not need this script and may perform these actions from the server itself.

Note: All of this is done using Parse's REST API. In some cases, on some computers, this API may not be accessible right away. If you notice a lot of errors during this stage of the install process, try executing the root script in your current shell instead of calling it. This way you retain any environment variables and are able to execute this script directly later on.

Cleanup:

- *cleanup.sh*: This script stops and removes any containers related to this project, then removes any unneeded volumes. It will print errors if not the entire install script was run, but will still do its job.

4.2 Backend Communication

Parse is a project origininally started by Facebook, which is now available as a free and open source platform. Detailed Information about Parse and its various APIs is available under `https://parseplatform.org`.

In order to isolate Parse from the main part of the project, the project uses and expects the following API available as a $BEutil$-object defined in the module *backend.utils*, found in file httpd/app/backend.utils/backend.utils.js , which must be implemented by any replacement for Parse. This has been done to allow for future changes to the architecture. Additional config data can be stored under httpd/app/config.js . Notice however that this file is dynamically generated during installation, so please adjust the init-script instead.

- $BEConfig$: This must be an item representing the configuration of the app and implementing the object interface below, or *null* if the configuration was never queried before. Must contain specific data only after a call to $BEWait$. May be initialized as *null*.
- $BEUser$: This must be an item representing the current user object, implementing the object interace below, or *null* if the user is not logged in. Must contain specific data only after a call to $BEWait$. May be initialized as *null*.
- $BEAllUsers$: This must be an array representing a list of all users which can be aquired from the backend with the current privileges, or *null*. Must contain specific data only after a call to $BEWait$. May be initialized as *null*.
- $BEWait(conf, user, all)$: This function is passed three booleans, representing the need for $BEConfig$,$BEUser$, and $BEAllUsers$ to contain up-to-date values respectively. When called, the data for all specified flags should be aquired and above values be set accordingly. When *user* is specified, additionally *signed_in* is expected to return up-to-date information. The function returns a promise which will resolve once all requested values are set and up-to date.

- *signed_in*: A boolean value indicating whether the user is signed in. Must contain specific data only after a call to $BEWait$.
- *sign_up*($username, password$): This function is passed two strings, an e-mail and a password, and is expected to try to sign the user up, as well as log them in. It must initialize the entire datastructure representing the user as described in section 4.3.1 . It must then call $BEWait$ with the *user*-flag set and wait for the return's resolution. It returns a promise to be resolved once the log-in is concluded (sucessful or not).
- *sign_in*($username, password$): This function is passed two strings, an e-mail and a password, and is expected to try to log the user in. It must then call $BEWait$ with the *user*-flag set and wait for the return's resolution. It returns a promise to be resolved once the log-in is concluded (sucessful or not).
- *sign_out*(): This function is expected to try to log the user out. It must then call $BEWait$ with the *user*-flag set and wait for the return's resolution. It returns a promise to be resolved once the log-out is concluded (sucessful or not).

Likewise every object returned by the API, which is representing data from the backend and is not a leaf-node, is expected to implement the following interface. For arrays, every contained element representing such data must implement it.

- *get*(key): This function returns the child under the given key key. Note that it must also implement this interface, if applicable.
- *set*($key, value$): This function locally updates the value under the given key key to the value $value$. For the current project the object under key is guaranteed to not have children and thus not implement the API.
- *save*(): This function pushes all local changes to the backend for the object on which it is called and its children recursively. It returns a promise to be resolved once that is done.

4.3 Database Layout

eNutri's view on any data is one of a few trees that must be available to the project via an implementation of the above API. This section is a short description of the actual tree layout and the data types expected. Every non-leaf node is currently represented by a Wrapper on a ParseObject implementing the above API. For maximum compatibility the only leaf types used are Strings, Numbers (Integers or Floating Points) and Booleans, as well as objects and arrays.

4.3.1 Users

User The root node for the user.

Child	Type	Description
username	String	The user's e-mail address.
group	String	A classification of users. Possible values include "rejected", "waiting", "web", "control", "researcher", and undefined.
report	String	"web" or "control" depending on the type of report the user will see. Can be undefined.
state	State	The user's progress.
utm	Utm	Information about the user's cookies.
screening	Screening	Information about the user collected during the screening.
feedback	Feedback	The feedback results.
surveyy	Survey	The survey results.
ffq1	Ffq	The first FFQ.
ffq2	Ffq	The second FFQ.
ffq3	Ffq	The third FFQ.
report1	Report	The first FFQ's results.
report2	Report	The second FFQ's results.
report3	Report	The third FFQ's results.

State The state node holds all information describing the progress of a participant.

Child	Type	Description
consentResponse	String	The user's response to the consent form. Usually either "Agree" or "Disagree" or undefined.
consentTimestamp	Number	The Unix time of the response.
screeningComplete	Boolean	Whether the user has completed the screening.
screeningAccepted	Boolean	Whether the user was accepted for the study.
surveySUS	Number	The result of the Survey.
FFQCompleteCount	Number	The number of completed FFQs.
firstFFQCompletionDate	String	A formatted date of the completion of the first FFQ.
firstFFQCompletionTimestamp	Number	The Unix time of the completion of the first FFQ.
lastFFQCompletionDate	String	A formatted date of the completion of the most recent FFQ.
lastFFQCompletionTimestamp	Number	The Unix time of the completion of the most recent FFQ.
lastFFQCompleteId	Number	The ID of the most recently completed FFQ.
nextFFQDueDate	String	Formatted date of the next scheduled FFQ.
feedbackComplete	Boolean	Whether the user has completed the feedback form.
ffqStartedCount	Number	The number of started FFQs.

Utm Some cookie data from the original eNutri. The data is never actually used within the application and may be removed in the future.

Child	Type
source	String
medium	String
campaign	String
term	String
content	String

Screening The data collected during the screening. Where Boolean/String is denoted as type for an object xyz, xyz has type Boolean while the additional field xyzDesc has type String. xyzDesc in this case always represents some input by the user further describing their condition.

Child	Type	Description
screeningDate	String	The formatted date the screening took place.
screeningTimestamp	Number	The Unix time the screening was finished.
firstname	String	The first name of the user.
surname	String	The surname of the user.
gender	String	The user's gender. Currently only "male" and "female" are supported.
age	Number	The user's age in years.
height	Number	The user's height in cm.
education	String	A description of the user's education.
notgoodhealth	Boolean/String	Additional information the user wants to provide.
lactose	Boolean	Whether the user is intolerant to lactose.
foodallergy	Boolean/String	Whether the user has any allergies.
diabetes	Boolean	Whether the user has diabetes.
methabolicDisorder	Boolean/String	Whether the user has a methabolic disorder.
illnesses	Boolean/String	Any diseases the user may have.
medication	Boolean/String	Whether the user takes any medicine.
medicalInformation	Boolean/String	Any other diseases.
vegan	Boolean	Whether the user is vegan.
dietaryreq	BooleanString	Additional dietary requirements.
recruitment	String	How has the user heard of this study?
pregnant	Boolean	Whether the user is pregnant.
livinguk	Boolean	Whether the user is living outside the UK. Can be repurposed for ther countries.
consultation	Boolean	Whether the user regularly seeks nutritional advice.

Feedback A node containing the Feedback results.

Child	Type	Description
reportDate	String	A formatted date the feedback was given.
answers	Array	The user's answers, in order.
user61/user71/user81	Array	Answers given by the user in text.

Survey A node containing the SUS results.

Child	Type	Description
reportDate	String	A formatted date the feedback was given.
answers	Array	The user's answers, in order.
problems	String	Any problems specified by the user in text.
surveySUS	Number	The SUS result.

Baecke The results of the Baecke Physical Activity Questionaire.

Child	Type	Description
answers	Object	The user's answers.
work	Number	The user's activity score at work.
sports	Number	The user's activity score regarding sports.
leisure	Number	The user's activity score during leisure time.
overall	Number	The user's overall activity score.

Nutrients A node describing the calculated nutrient intake.

Child	Type	Description
results	Object	The intake of nutrients, with the nutrients as keys.

Ffq A node describing a single FFQ.

Child	Type	Description
preFFQTimestamp	Number	The Unix time before the Backe Questionaire.
FFQStartTimestamp	Number	The Unix time before the FFQ.
FFQComplete	Boolean	Whether the FFQ has been completed.
lastFoodItem	Number	The index of the last food item, for resumption of an FFQ.
id	Number	The FFQ ID. Unique within one user account.
baecke	Baecke	The results of the Baecke Questionaire.
nutrients	Nutrients	The calculated nutrient intake.
results	Array	The given answers, in order.
currentWeight	Number	The user's weight, in kg.
weightUnit	String	The user's preferred weight unit.
preFFQComplete	Boolean	Whether the user completed the Baecke Questionaire.
preFFQDate	String	The formatted date of the Backe Questionaire.
appName	String	Information on the app.
appCodeName	String	Information on the app.
appVersion	String	Informtion on the app.
userAgent	String	Infromation on the user's PC.
platform	String	Infromation on the user's PC.
product	String	Infromation on the user's PC.
language	String	Infromation on the user's PC.
vendor	String	Infromation on the user's PC.
mostLikelyBrowser	String	Infromation on the user's PC.
screenWidth	String	Infromation on the user's PC.
screenAvailWidth	String	Infromation on the user's PC.
screenHeight	String	Infromation on the user's PC.
screenAvailHeight	String	Infromation on the user's PC.
FFQCompletionTimestamp	Number	The Unix time of the FFQ completion.
FFQCompletionDate	String	A formatted date of the FFQ completion.

Report A node describing the results of a single FFQ.

Child	Type	Description
id	Number	The report ID.
ffqId	Number	The corresponding FFQ ID.
currentWeight	Number	The weight provided by the user.
weightUnit	Number	The user's preferred weight unit.
reportDate	String	A formatted date of the FFQ completion.
baecke	Baecke	The results of the Baecke Questionaire.
nutrients	Nutrients	The calculated nutrient intake.
results	Array	The FFQ answers.

4.3.2 Configuration Data

The configuration data is stored in a tree with only one root node and several leaf nodes. Below a description of these leafs

constantsList This is an object with the following keys:

- FfqValidity (Number): Number of hours an FFQ can be worked on.
- FfqInterval (Number): Number of days between FFQs.
- studyDuration (Number): The maximum time between the first and last FFQ, in days.
- defaultWeightUnit (String): The default weight unit.
- defaultHeightUnit (String): The default height unit.

userEligibilityList An array containing objects with three keys: "type" (String), "arguments" (Array of Strings), and "decision" (String).

"decision" is always one of "reject","wait" or "accept", and represents a decision to accept or reject a user for a study, or to have them wait until a researcher manually decides on a specific case.

A decision matches a user based on "type", "arguments" and their screening results. If the "type" is "unconditional", the rule matches always. If the "type" is "or", the rule matches on a disjunction of the boolean screening variables defined in "arguments". If the "type" is "and", the rule matches on a conjunction of the boolean screening variables defined in "arguments".

The array is traversed in order and the first matching decision is applied. The app's behaviour if no decision matches, is undefined.

foodsList An array containing food items as objects, with the following keys: The order of items defines the order of items in an FFQ and their index defines the matching images under $httpd/app/assets/images/foods/$, where e.g. the small portion of the food with index 0 should have an image in file $food - small - 0.jpg$.

- foodname (String): The food item's Name.
- foodexample (String): A example for the food item (optional).
- pSmall (Number): The size of a small portion in g.
- pMedium (Number): The size of a medium portion in g.
- pLarge (Number): The size of a large portion in g.

nutrientsList An array containing objects specifying nutrients per gramm of a food item. The indices of food items correspond to the indices in foodsList. Each object contains a key for each nutrient and a number value. The implementation is unit-agnostic for the nutrients.

messagesList An object containing message identifiers and "TEMPLATE" as keys, and objects as values. These message objects contain the following keys (All values are Strings):

- TYPE: "general", "web", or "control". An indication where this should appear on the admin page.
- DESC: A human readable description of where this message will appear.
- For every supported language, under a key corresponding to a language code like "en_UK", the message in this language.

To see a list of all currently available messages keys, see the example configuration in *httpd/EXAMPLECONF*.

4.4 Admin Panel

The admin panel, whose code can be viewed in the directory *httpd/app/admin/*, is in its essence just a page rendering parts of the Config tree described above in an interface which allows to alter the data. Currently it is working on a shallow copy of the data. A more sophisticated version would use smart deep copies to allow actions such as resetting the view to the current data model. Code for such a reset button has been provided, and the copy functionality isolated into its own function. To experiment with deep copies, a simple encoding and reparsing of any data to and from JSON suffices. However, with a lot of data this can become VERY slow. If reset functionality is required, a future project may implement deep copies. External projects for this seem to exist, but were not included in this version to minimize unneeded dependencies.

As per the Parse JS API Documentation (`https://parseplatform.org/Parse-SDK-JS/api/2.15.0/` at the time of writing), JavaScript applications not running in NodeJS are not allowed to use a master key in the JS API when updating the configuration. Since it is however write-protected and without more fine-grained access control enforced by some future server wrapper, we

need to use the master key. To circumvent this problem, instead of the JS API, the REST API is used for updating ParseConfig.

Apart from this, saving and loading are simple copy instructions, and importing and exporting just convert an object containing all configuration children as key-value pairs to and from JSON.

Note: Angular does become very slow when given a lot of data to keep track of in the DOM, even if parts of it are never shown. The nutrition table can easily contain > 100 food items and > 100 nutrients, thus consisting of more than $10,000$ items. To counter this, items in large tables are devided into pages, in the case of the nutrientsList in both dimensions. Any actions on these lists must take this into account. Functions for repagination, if needed, are provided.

Angular does not like this kind of messing with the DOM either. A quick and dirty way to fix this is to erase any $\$\$hashKey$ attributes off objects and thus force Angular to reindex these objects.

Below a mapping between the Config children and the tabs they are altered on.

Child	Tabs
constantsList	GENERAL → CONSTANTS
userEligibilityList	USER SELECTION
foodsList	FFQ → FOODS
nutrientsList	FFQ → NUTRIENTS, though it can be altered by adding or removing food items in FFQ → FOODS
messagesList	REPORTS → WEB for messages with type web, REPORTS → CONTROL for messages with type control, and GENERAL → MESSAGES for messages with type general. TEMPLATE is never shown and only used to infer implemented languages.

5 Usage

Please note: This usage quick guide is targeted at the researchers conducting the study, not the participants.

5.1 Installation

Note: The installation, while straight forward, does not work in a point-and-click manner and may be done by someone in the IT-Department.

System Requirements:

- Any Unix-Like System
- */dev/random* and */dev/urandom* existent and returning (pseudo-)random numbers upon read.
- Docker installed and accessible

To install the application, simply edit and run the script *setup.sh*. For most use-cases, you only need to edit the value of $eNutri\_WEB\_URL$ to where your app will be accessible. The application needs to be accessible there immediately. Make sure to comment out the dashboard, if you don't need it. Also make sure you have access to the Docker CLI.

During installation you will be shown the randomly generated master key. Please note that one down. You will need it later.

For further automation, running the script via *source setup.sh* will set the corresponding environment variable $eNutri\_MASTER\_KEY$.

To deinstall the application, simply run *cleanup.sh* Note that it is currently not possible to have several instances at once on the same machine.

5.2 Administration

5.2.1 Before the study

To configure the FFQ, visit the app in your browser and log in using the default credentials. It is normal at this point to not see any messages, since they have not been configured yet.

The default credentials are:

- eMail: admin@admin.com
- password: admin (PLEASE CHANGE IMMEDIATELY AFTER LOGIN)

After login, please change your password, since this research account is capable bypassing all restrictions on access to other user's informations, in order to allow for viewing and exporting the study's result. Select $ADMIN\ PANEL$ from the menu on the left.

Your screen hould look something like figure 5 by now. If applicable, you want to import your previous settings under the $IMPORT$ tab. If you do not have a previous configuration, a link within the first tab will navigate you to a site where you can generate or alter one. For your first setup, just select a BLS-like XML file and hit *Edit*. The resulting string may be passed to the $IMPORT$ tab.

You can save your configuration by entering your Master Key on the top and hitting $SAVE$.

What follows is a description of the configuration options the $ADMIN\ PANEL$ has to offer:

GENERAL tab The options within this tab allow you to customize parts of the app not related to the questions in the FFQ itself. Under the CONSTANTS subtab, you may specify:

- FFQ Validity: The amount of time within which a begun FFQ has to be completed, in hours. If an FFQ is not completed within this time, it is marked as uncomplete and the user has to start a new one.

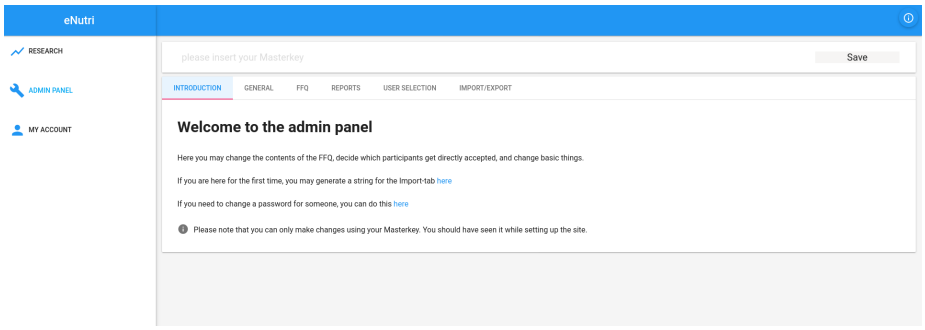

Figure 5: The admin panel's front page. The configuration options are devided into different tabs. To save the current state, enter your Master key at the top and click "Save".

- FFQ Interval: The minimum amount of time between the first and second FFQ, in days.
- Study Duration: The minimum amount of time between the first and third FFQ, in days.
- Default Height Unit: The height unit shown by default.
- Default Weight Unit: The weight unit shown by default.

Under the MESSAGES subtab, you can alter the messages shown throughout the application. Note that some messages cannot be altered at all currently while others (like the messages on the reports) can be changed in the REPORTS tab. Hovering over i-icons will give a short description of where the text is shown.

FFQ tab The FFQ tab contains two subtabs. It should only be changed before the study, as the database is not designed for intermediate changes.

In the FOODS subtab, you may specify food items by a name and optionally some examples. The FFQ is conducted using these items and both the name and examples are shown. You may add or remove items using the provided buttons. The table is split into pages for easy viewing and website speed. You may also specify three portion sizes in gramm.

Note: There are also buttons to upload photos for the portion sizes. Currently these buttons do not work. To provide images for food sizes, they can be placed manually in $httpd/app/assets/images/foods/$, where they are expected to have the filename "food-¡portion size¿-¡index of the food item starting at 0¿.jpg". E.g. the small portion of the food with index 0 should have an image in file $food - small - 0.jpg$. The buttons will likely work in a future version of the app.

The NUTRIENTS tab allows to specify the nutritional values for each food item per gramm. You may add or remove nutrients if desired by either entering

their name and clicking Add, or clicking the small x next to a nutrient's name in the table.

Note: eNutri is agnostic of nutrient value units (i.e. the energy could be given as kcal, J, kJ, MJ,...). However, when entering values, you should keep two things in mind: Not every implementation may handle fractional values well, and eNutri does not convert between units, i.e. if you specify MJ/g here, the participant's results will specify their intake in MJ, if you choose kJ/g, they will report kJ.

REPORTS tab Here you may alter the messages shown in the reports. The subtabs devide between the messages for the web group and the control group. Again, hovering over the i-icons shows a short description of the context in which a text can be seen.

USER SELECTION tab Within this tab, you may customize the process in which participants, after screening, are accepted or rejected. It works as follows:

In the tab, you will see a list of "rules", the "decision" of one of which is applied to a participants. "Decisions" can be "accept", in which case a participant is accepted instantly after their screening, "reject", which will lead to the participant being rejected immediately, or "wait", which will show a message to the participants to return later. In the latter case, researchers may choose to accept or reject the participant manually in a process described in section 5.2.2.

Which rule is applied for each participant depends on the information given during the screening and is determined as follows:

Rules are tested in order from top to bottom and the first "matching" one is applied. A rule matches based on the "condition":

- "unconditional": This rule applies every time.
- "or": A list of "selectors" must be given with each possible item corresponding to a checkbox during the screening. The rule applies if the user has checked **at least one** of the specified checkboxes. (Think: Participants having the first condition OR the second OR the third)
- "and": Again, a list of "selectors" must be given with each possible item corresponding to a checkbox during the screening. But now, the rule applies only if the user has selected **all** of the specified checkboxes. (Think: Participants having the first condition AND the second AND the third)

Please note that you do not want to have participants matching no rule. The current implementation defaults to accepting these users, but future versions may differ. It is therefore good practice to explicitly decide on "leftover participants" via a final "unconditional" rule.

To illustrate this functionality, here are two examples demonstrating the flexibility of this system:

- **Example 1:** You want to conduct a study on the general population that is neither vegan nor pregnant. Participants taking medicine you might reject based on what medicine they are taking. The rules for this scenario could look as follows:
  - First we reject people that are vegan OR pregnant.
    Condition: "or", Selectors: "vegan" and "pregnant", Decision: "reject"
  - Then we identify those users which we are unsure about.
    Condition: "or", Selectors: "medication", Decision: "wait"
  - And finally, all participants making it through the first two checks may be accepted.
    Condition: "unconditional", Decision: "accept"
- **Example 2:** You want to conduct a study on pregnant vegans, as long as they do not have diabetes. In this case the rules may resemble:
  - Before accepting pregnant vegans, we filter the ones with diabetes out.
    Condition: "or", Selectors: "diabetes", Decision: "reject"
  - After that, we may accept all those still undecided provided they are indeed vegans AND pregnant.
    Condition: "and", Selectors: "vegan" and "pregnant", Decision: "accept"
  - Since at this point, we already dealt with pregnant vegans, we may reject the rest.
    Condition: "unconditional", Decision: "reject"

IMPORT/EXPORT tab This tab allows you to import and export any configuration entered into this site as a simple string. This may be useful to backup entered values.

5.2.2 During the study

Administration during a study mainly comes down to two actions: Accepting waiting participants and helping with technical problems.

Accepting participants Based on your settings, users may have to be accepted explicitly during the study. To do this, log in to the research account select *RESEARCH* from the menu on the left. Here you can view most of what user's have entered during their time working on the study.

Under the tab SCREENING, you will find a table containing the columns Accept and Reject at the far right. If there are any participants which have entries here, you can accept or reject them by clicking the buttons, after you have seen the comments left by participants as to what their conditions are.

Fixing login issues If users forget their password, they might be unable to finish the study. If you have left a contact method to reach you in the messages defined before the study, you may however reset their password manually using your Master Key.

To do this, after logging in, navigate to the *ADMIN PANEL*. In the first tab, you will find a link to a password reset form. Once there, enter the participant's eMail address, a new password you should communicate to the participant, and the Master Key in their respective input fields. After clicking Apply you should see a short confirmation message. The password has now been altered and the participant may continue with the study.

5.2.3 After the study

Once the study is over, you may download any participant data for further processing. eNutri is capable of generating CSV (Comma Separated Values) files. To download the data, after logging in, navigate to the *RESEARCH* subpage using the menu on the left. Under the tab EXPORT you will find two buttons. The user data CSV can be generated by clicking "EXPORT FFQ RESULTS". Save the resulting file when prompted.

Preparing for the next study eNutri currently does not provide a way to reset the database inbetween studies. Therefore, to conduct another study, the application has to be reinstalled. To speed things up, here are a few tips:

- You may export your current configuration by navigating to the *ADMIN PANEL*, selecting IMPORT/EXPORT, clicking "Export Settings" and saving the appearing text in its entirety to somewhere on your PC for later retrieval. When setting up the new instance as described above, you don't need to generate a new Import string and can just use the exported one, thereby retaining all your settings.
- The Import string generator is capable of modifying only certain settings. If you want to retain the configured messages, but would like to change the set of food items, a good way to do this is to copy the import string into the output field of the generator, selecting what you want to overwrite and possibly a new BLS file, then clicking edit. The string will change to reflect your changes.

6 Conclusion and Outlook

6.1 Personal Summary

This project took a lot more time than expected. Even though some goals specified in the beginning were dropped (including translating the app into german, which had apparently been done before), a lot of time went into reverse engineering the application, taking months during the normal semester. Given the

size of the project, my lack in understanding of Angular and its inner workings, the missing isolation of Firebase-related code, the lack of any visualizations of the working application apart from some screenshots, and the burden of working alone, it was also regularly quite frustrating to sometimes spend hours blindly guessing what within angular did prevent the site from loading. I had to rewrite large parts of the application (The menu, home page, routing, ...) before I even saw the characteristic blue bar at the top. For these reasons, I was also unable to implement effective tests for a long time, which did however force me to think ideas through more thoroughly.

Additionally, but not nealy as bad, I also had to figure out Parse Server and design an API to have it work with the application. Luckily, Parse is (for the most part, some pieces are still missing) well documented, which helped a lot. Once I had a clearer picture of the application, designing the API and adapting it to new challenges while keeping it simple was actually somewhat fun.

Part of my Interdisciplinary Project involved visiting an introductory course in nutritional science. While I very much enjoyed learning some background on what I was doing and the field in general, and would maybe learn some more in the future, I do not think that the kind of work involved in this project is what I want to do a lot of in the future.

6.2 The state of eNutri

The finished project delivers on its two main promises, but not much more: It runs completely free of any Firebase dependencies thanks to the new Backend Server that is Parse. Additionally, if this server should become unsupported in the future, a clearly defined API allows to swap it out with relative ease.

An admin panel was also implemented, which allows to change basic parameters and messages. It does not, however, allow to change everything desirable within the application. E.g. some strings are still hardcoded into the application and have should loaded dynamically in a future version. Likewise, the general layout is currently fixed as well. The option for multiple languages was included in the database layout and can be added with little work once the above is done. Mainly there need to be buttons to add new languages in the admin panel and a convenient way to switch languages for the user.

A dedicated server was not implemented yet. This meant that some features could not be implemented properly, like input validation, image uploads from within the application, or mail password resets. Some of these problems I countered with temporary solutions, like a simple web page using Parse's REST API to rewrite passwords within parse directly, which can be operated by the researchers using Parse's Access-Control-Bypassing Master Key. Some problems could at least be mitigated. E.g. the database layout is fixed and can only be altered using the master key.

Moving away from Firebase also came with some burdens when deploying and monitoring the application. This was countered with a number of Bash scripts setting up docker containers and configuring the database from the outside.

As for the usability of the application in a real study: While it should theoretically be possible to use the application as-is, I would advice to only use it with small numbers of participants at first, since there are likely still bugs undiscovered.

I will, with permission of the chair, release the current source code under the MIT License, just as the original project. This will allow for future improvements to the code with minimal legal restrictions.

6.3 Future Work

There is still a lot of work left to make eNutri an easy-to-use application. First of, a proper server should be implemented to enable among others

- sanity-checking of everything the user might supply to the database.
- image uploads from within the application.
- database resets, to remove the restriction that the application has to be reinstalled for every study.
- user-requested password resets via mail.
- dynamic database layout changes, to allow for a variable amount of FFQs.

Some of these goals may be achieved within Parse (using e.g. Cloud Code) or HTTPD.

Clientside, at least the following should be done:

- Make more static elements dynamically configurable. Includes strings and layout.
- Add an option to switch languages.
- Improve usability by means of better guidance for the participants.