University of Zurich[UZH]

Department of Informatics

Lukas Yu

The Swiss Feed Database documentation

February 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

L. Yu:
Technical Report No. 2017.xxxx, February 2017
Database technology group
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.ifi.uzh.ch/en/dbtg.html

Chapter 1

Introduction

1.1 Swiss Feed Database

The Swiss Feed Database is a website where information about nutrient measurements in feeds can be searched and visualized with tables, heatmaps etc. Several options are available to choose exactly what is needed. Filters like specific feeds, nutrients, locations and time are used to narrow down the selection and the user can choose to get the data shown as a list of single measurements or in a summarized fashion.

The data originates from a database built from data made available from Agroscope and is continuously updated with new information.

1.2 Application overview

The application is built on top of NodeJS using AngularJS and for storing data Postgres is used. The frameworks/libraries that are used are listed thereafter.

1.2.1 Used frameworks/libraries

Client:

- AngularJS 1.6.2: `https://angularjs.org/`
  A MVVM framework based on JavaScript
- angular-google-chart: `https://github.com/angular-google-chart/angular-google-chart`
  Google charts libraries wrapped in Angular directives
- angular-ivh-treeview: `https://github.com/iVantage/angular-ivh-treeview`
  A treeview used for selecting search options written for use with AngularJS

Server:

- Node.js: `https://nodejs.org/`
  Server platform with JavaScript
- Node package manager (npm): `https://www.npmjs.com/`
  Package manager for NodeJS
- Express: `http://expressjs.com/`
  Web framework for NodeJS
- jsonwebtoken: `https://github.com/auth0/node-jsonwebtoken`
  Library for encrypting and decrypting JsonWebTokens (JWT)
- knex: `http://knexjs.org/`
  Promise based query builder
- pg (postgres): `https://github.com/brianc/node-postgres`
  PostgreSQL client for NodeJS

1.2.2 Overview

On the client side, we have angular components, which are views with integrated controllers. A view defines how the layout looks like and it is written in HTML code. A controller implement the business logic, defined as a function that gets called when initialized. These components are isolated from each other, so variables and functions are only available inside the component that defines them. To define a component one more file is needed, the module file that is used as a container that can be loaded into other modules as a dependency. To add a component to any HTML, first include the module of the component to another module and then simply add an HTML-tag with the component's name to the HTML code. (eg. `loginForm` component => `<login-form></login-form>`)

To communicate between the components, we have services. In order to use an Angular service, the module of the service has to be included as a dependency in the module that needs it and the service has to be injected into the controller function. Services are singletons, which means if multiple components loads the same service, it both references the same instance of the service. In this application, we use services to share data and functions if more than one component depends on it (eg. session data). The core functions of the application are all saved in the services.

The application is a single page application. The whole website is defined in the file `/client/index.html`, where the global controller is responsible to switch the `<ng-view></ng-view>` with the right component.

On the server. We have several endpoints for retrieveing or saving information from the database.

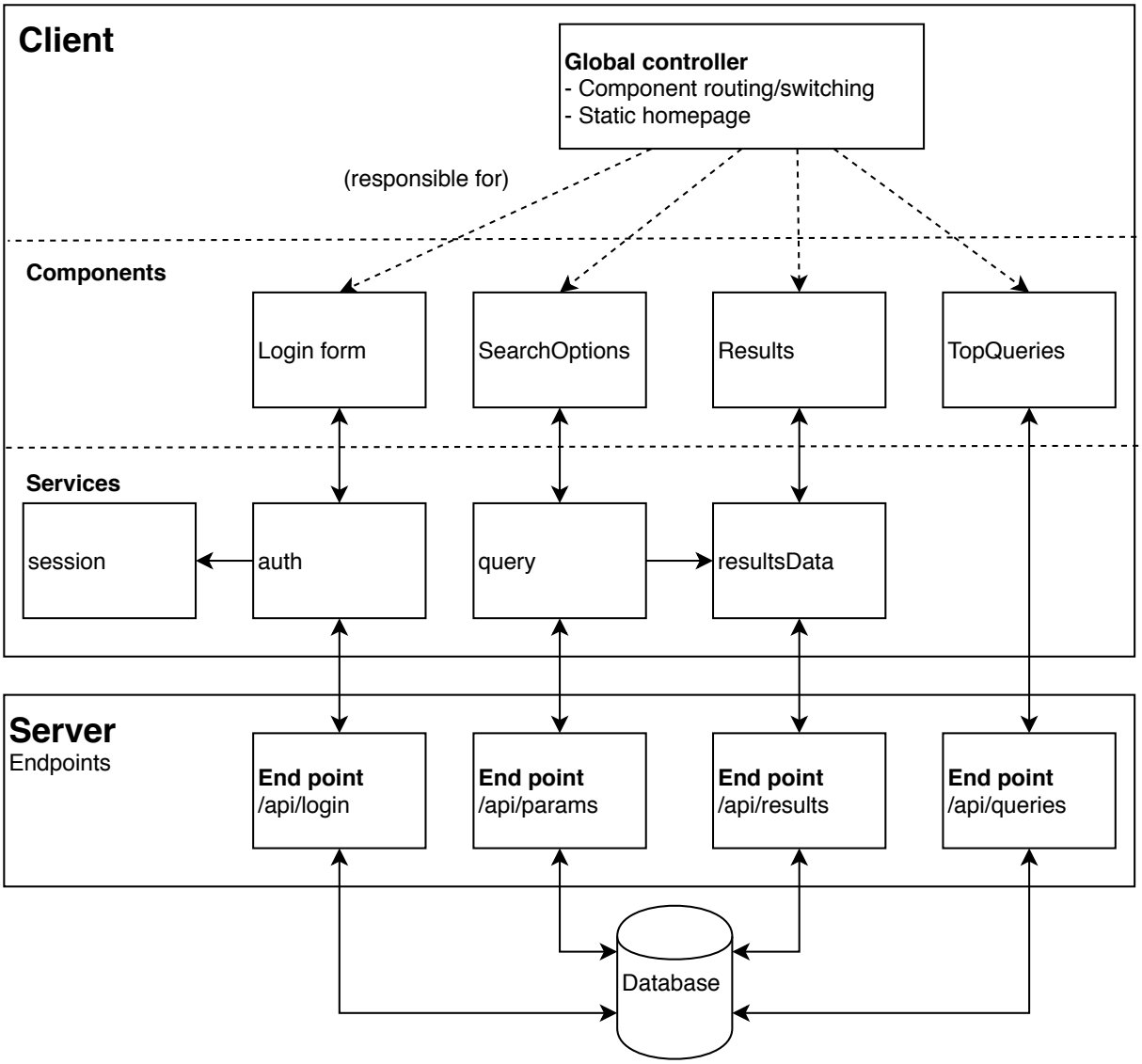

Figure 1.1: Information flow diagram

1.3 Setup

- Install node.js (from `https://nodejs.org/en/download/`)
- Start the server with the following command within the `/feedbase-version2/` folder:

  ```
  $ npm start
  ```

- The server should be running on port 3000 and can be viewed with the address `http://localhost:3000/`

The frameworks/libraries that are used are already installed and stored in the folder `/node_modules/` and they don't have to be reinstalled.

If any additional module has to be installed, it can be done by executing the following command: `$ npm install {module name} --save`

The Postgres database has to be installed and listening at port 5432. Further instruction about Postgres are written in the readme-file in the same folder. Connection information for the database can be edited in knexfile.js.

1.4 Services

Each service is defined into its own module, so it can be included into other modules (or other services) seperately. The services are stored in the folder `/client/services/`.

1.4.1 session service

The session service is used to save user information. It creates an object called session that has the following fields: `username`, `userlevel`, `language` and a `loggedIn` boolean. The function login sets the `username` and `userlevel` so that this information is acessible from any function that injects this service. The default value for `userlevel` is 0 (guest) and for `language` english. Other userlevels are: 2 for subscribers and >2 for admins.

1.4.2 auth service

This service only contains function `login()` for the login. The function takes the `username` and `password` and makes a POST-request to the `/api/login` endpoint. If the server accepts the credentials, Then an encrypted JsonWebToken is sent to the client that includes all session information (eg.`userlevel`). The client stores this token as the authorization http header. The server is stateless, so all the session information is only stored in the token. All future request from the client includes the header, which the server will use to identify the client.

1.4.3 query service

This service is responsible for the query options. It uses the variable `query.categories` that defines a query using the following attributes:

agrideaFeeds	Agridea Feeds [tree]
unclassFeeds	Unclassified Feeds [tree]
classFeeds	Classified Feeds [tree]
nutrients	Nutrients [tree]
nutrientsDerived	Derived nutrients [tree]
cantons	Cantons [list]
altitudes	Altitudes [list]
years	Years [list]
seasons	seasons [list]
radius	radius number (not used at the moment)
fresh	Fresh or dry matter boolean
dataType	Detail or summary data ('td', or 'sd')
raw	Raw data boolean

The tree and list options are stored in a way the angular-ivh-treeview can parse it and draw the tree structure menus. The structure looks like this:

```
[
  {
    label: 'Cereal grains',
    selected: false,
    children: [
      {
        label: ' Barley grains, heavy (70 - 74 kg/hl) (lat. Hordeum vulgare)',
        selected: false
      },
      {
        label: ' Triticale, grains (lat. Triticum x Secale)',
        selected: true
      }
    ]
  }
]
```

Children of category nodes can also be category nodes so the tree structure can be arbitrarily deep. Attributes of the query that has a flat hierarchy (see table: [list]) consist of lists with nodes without a parent. This service provides the following function:

- `saveQuery()` saves the current query to the database. It calls the function getParams() to get an object with all selected items (same structure as query.categories object, but only with lists of selected ids) and sends them to the server. NOTE: This function cannot be called from the user yet.
- `getQuery(queryId)` retrieves a pre-saved query from the database. The response from the server only contains ids of the selected search options, so we make another request with getParams() to get the whole tree structure menus along with names, description etc.

- `getParams(loadlevel, selected)` retrieves new options from the database according to the already selected options. The loadlevel indicates what we want: 0 means feed, 1 nutrient and 2 geo and time options. The selected options normally should be the current selected ones. We get them with the function getSelected(). With the response from the server we update the current query with updateParams(newParams, loadlevel).
- `updateParams(newParams, loadlevel)` updates the the trees in query.categories.
- `getSelected()` gets the selected ids from the tree structure menus and returns an object with the same structure as query.categories, but only contains lists with the selected ids.
- `validateTree()` is necessary to update the tree after it has been modified automatically (not from the user manually clicking). It is used to update any parent's checkbox state based on their children.

1.4.4 resultsData service

The resultsData service is used to request data from endpoint `/api/results`, format and store it in a way that the google charts table (from angular-google-chart library) can display the data. There are two types of data: detail data and summary data. Detail data lists every single measurement in a row with information about date, location etc. Summary data lists reference values about every feed/nutrient combination. The response from the server consists of two parts: column headers (nutrient names) and rows (measurement values). Note that a single row only includes one single measurement, which means that multiple rows from the server have to be joined together. The service implements the following functions:

- `getResults()` gets the results. It doesn't need any parameters. It takes the parameter directly from the query service. It fills the table by calling functions `setColumnsSummary` and `setRowsSummary` (if summary data is selected ofr functions `setColumnsDetail` and `setRowsDetail` (if detail data is selected)
- `setColumnsSummary()` takes the response from the server and prepares the column headers of the table with the nutrients names. First it adds a column for the feed name. Then it loops through every selected nutrient and adds a column with their name and unit measure for each of them. After that it fills the `columnsHash[nutrientId]` object with the nutrient id as the key/membername and the column index as the value. The `columnsHash` object is defined outside of this function so it does not get deleted after the function finishes. The object is then used by `setRowsSummary()` to find the right column to fill in the measurement values.
- `setColumnsDetail()` the same like `setColumnsSummary()`, but it additionally adds some more columns like location, date, altitude etc. and if multiple `analysis_id` exists for one nutrient, it adds an `indicative` column, which should take the value of

the analysis with the highest priority (lowest number). At the end it saves the index of the any column to the `columnsHash` object, which `setRowsDetail()` will use to find out the position of the column. The key/membername of the `columnsHash` consists of the nutrient id and the analysis id separated with a '/' (eg. '24/2'). The indicative columns are stored in the columnsHash with the nutrient id and the string 'indicative' (eg. '24/indicative) as key/membername.

- `setRowsDetail()` fills the table rows with actual values. First it creates two variables: `newRows` list as the return value and `rowsHash` to find a row based on the LIMS measurement number `lims_number`. Then it loops through the values from the response of the server. If a row has not been added to `rowsHash`, it creates a row with the `lims_number`, `date`, `canton`, `postal_code` and `feedname` and adds it to both `rowsHash` with the `lims_number` as key and the row object as value and `newRows`. It adds the measurement value to the right position of the row based on the nutrient id and analysis number (column index from the `columnsHash` object filled by `setColumnsDetail()`). If the current nutrient has an indicative column, check the `priority` number of the measurement. if it is lower than the previous, fill the indicative column with the current measurement value.

- `setRowsSummary()` does the same as `setRowsDetail()`, but only adds the `feedname` to the first column and doesn't have any priorities.

The final `data` variable of the service should look like this (summary data example):


```
{"cols": [
  {
    "id": "feed",
    "label": "Feed Type",
    "type": "string"
  }, {
    "id": 180,
    "label": "DM g/kg",
    "type": "number"
  }, {
    "id": 158,
    "label":"Ash g/kg DM","type":"number"
  }
], "rows": [
  {
    "c": [
      {
        "v": "Barley grains, decorticated (lat. Hordeum vulgare)"
      }, {
        "v": 870
      }, {
        "v":14.97
      }
    ]
  }, {
    "c": [
      {
        "v": "Barley flakes (lat. Hordeum vulgare)"
      }, {
        "v": 870
      }, {
        "v":13.22
      }
    ]
  }
]}
```

1.5 components

The components are stored in `/client/components/`. Each component defines a view and a corresponding controller. Each component is defined by three files. A module file which declares the module, a view file which specifies the layout in HTML code and a controller file. In some cases view or controller file is trivial. The components are defined in their own module so that each can be loaded into other modules seperately.

A component is broken into three parts: a module definition, the controller and the view (eg. searchOptions.modules.js / searchOptions.controller.js / searchOptions.template.js)

1.5.1 global

It is a controller to display the different components. It configures the different routes. For example, if `http://localhost:3000/#!/results` is called, the `<ng-view></ng-view>` element from the `index.html` file gets replaced with the `results` component. Anytime else will show the `top-queries` component. The other component like `login-form` and `search-option` will always be shown.

1.5.2 login-form

view

The view consists of two text inputs for username and password and one submit button. If the submit button is clicked, the login function of the controller is called. If the loggedIn boolean of the session service is true, the whole form gets hidden.

TODO: logout button if logged in

controller

All this controller does is call the login() function of the auth service when the submit button is pressed. This function sends the credentals to the server and gets the response (sucess/fail). See auth service for more detail.

1.5.3 search-options

view

The view respresents the data in the query service:

agrideaFeeds	Selection tree
unclassFeeds	Selection tree
classFeeds	Selection tree
nutrients	Selection tree
nutrientsDerived	Selection tree
cantons	Selection tree
altitudes	Selection tree
years	Selection tree
seasons	Selection tree
radius	Text input box (not implemented yet)
fresh	Radio buttons
dataType	Radio buttons
raw	checkbox
executeQuery()	Button
saveQuery()	Button

The selection trees are angular-ivh-treeviews and get their data from category member of the query service a treeview is drawn with a 'ivh-treeview' attribute with the value set to the data it should display (eg. `<div ivh-treeview="vm.query.categories.cantons">`). Any changes to the selection gets immediately represented in the query service thanks to the two way binding of AngularJS.

Note that the classifiedFeeds are broken into two trees/tabs: `Raw Materials` and `Roughage`

controller

The controller is responsible to bind the query service to the view. It also handles the tab navigation and disabling options based on selected options.

The two functions executeQuery() and saveQuery() essentially calls functions in the query service, but also does some changes to the layout (mostly AngularJS `ng-hide` directives to simulate tabs). These two functions get their input directly from the query service. No input from this controller is needed.

1.5.4 top-queries

view

The view consist of the tabs: top-queries and user-queries (only shown if logged in). The top-queries consists of sponsored queries and queries created and made available from the admin. The user-queries consists of queries that the user has made himself. In each of the two tabs there is a list with clickable queries for detail data queries and another list for summary data. At the bottom there is another container for news.

TODO: execute the query and change the url to show results after link is clicked

controller

The controller retrieves all the queries and the view displays them it is only executed at the start. It refreshes the data if the component gets reloaded (eg. after login). The 3 categories of queries are saved in the variables: sponsorQueries, topQueries and userQueries.

The `description` and `explanation` attribute of a query is a HTML string.

The `ng-sanitize` dependency is needed for parsing the description and explanation of a query and removing any harmful HTML tags.

The functionality is not wrapped in a service (like all the other functionalities of other components) because this is the only component that uses it. If a query is clicked, query.getQuery(queryId) is called with the respective query Id.

1.5.5 results

view

The view of the results is a google chart table. The table reflects data in the resultsData service. This component also only gets shown if the url changes to `http://localhost:3000/#!/results`.

controller

The controller only binds the resultsData service data to the view/table. Any changes to the `data` variable of the resultsData service immediately gets reflected in the table.

1.6 Server end points

The end points are defined in the files in the /server/ folder. The database queries are seperated from these files in the /server/dbQueries/ folder, where everything is saved as functions. The server is heavily Promise based (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`). Promises are used to execute asynchronous tasks. The functionality is similar to callbacks, but it has a more readable syntax and more useful functionalities.

Every file in this section are either directly in the /server/ folder, which contains end point definitions or in the /server/dbQueries/ folder, which contain the database queries.

Every exported functions from the dbQueries folder always return a promise with the result as response.

1.6.1 WhereStatement class

This class is defined in file /server/dbScripts/WhereStatement.js and it is used to construct SQL WHERE conditions as a string. It takes an object with selected ids (the object from the function getSelected() from query service) and transforms them in a "WHEREable" string. It joins the three feed categories (`agrideaFeeds`, `classFeeds` and `unclassFeeds`) and the two nutrient categories (`nutrients` and `nutrientsDerived`) into one list each.

The function getWhere() constructs a whole where-statement with every condition joined with ANDs, empty categories get ommited. Parts of the whole where-statement can be retrieved by getting the member of the class and it will return a string of ids seperated by commas (eg. WHERE feeds IN (statementInstance.feeds)), empty lists gets replaced with a 'null' (cannot leave parenthesis after SQL IN empty).

1.6.2 server/params.js /api/param (search options)

This POST end point accepts a body like the following, which is a JSON serialization of the selected query.categories values created by the function getSelected() in service query.

```
{
  params: {
    agrideaFeeds: [list of ids],
    unclassFeeds: [list of ids],
    etc. the same as query.categories (from service query)
  }
  language: 'en', 'de' or 'fr
  loadlevel: 0, 1 or 2
}
```

It also checks the authorisation header of the http request and tries to parse it as a JWT. If its successful, it sets the userlevel accordingly. (see server/auth/local.js)

Then it checks the dataType option if it is a valid string (either 'td' => detail / 'sd' => summary), or else it will default to detail data 'td'. The dataType string then gets changed according to detail/summary and raw input to be easily inserted into database queries (detail clean: 'vc', detail raw: 'vr', summary: 'vs').

Based on loadlevel, selected and dataType, the promise variables get set. If a promise variable is not replaced with an actual database query promise, it stays as an empty list and also returns an empty list if evaluated as a promise.

Checking the selected feeds and nutrients besides the loadlevel is required if the user already chose some nutrients and goes back and changes selection for the feeds (loadlevel=1). In that case the nutrients (LL=1) and also the geo/time (LL=2) options have to be updated.

getFeeds.js

This file stores the queries for getting feed search options.

The query for getAgrideaFeeds(param, lang) and getUnclassFeeds(param, lang) are pretty straightforward.

For getClassFeeds(param, lang) we need two queries. One for the actual feeds and another for the tree structure (groups/categories).

Result attributes for vc/vr/vs_classified_feeds:
`(name, key, feed_group_id, selected)`

Result attributes for vc/vr/vs_classified_feeds_tree:
`(name, feed_group_id, parent_feed_group_id)`

The function `constructClassifiedTable(feeds, groups)` takes those 2 query results and constructs a tree.

The algorithm first creates a hashmap `hashmap` of the groups/parents, so they can be found by their group_id. We then add each of the classified feeds to their parent as children. The same gets done for the groups. If a group doesn't have a parent, it gets added to a root object as its child, which then gets returned. The root object now holds the entire tree.

While the tree shouldn't have more than 3 levels right now, the algorithm can construct a tree with any depth.

getNutrients.js

This files stores the queries for getting nutrient search options.

The query for detail data derived nutrients is pretty straightforward, but keep in mind that this part still works, getting the final results with detail data and derived nutrients doesn't (see errors chapter)

For detail data standard nutrients, we use the table `vc_cube` (`vr_cube` for raw data), which stores the join result of `fact_table`, nutrient, feed and time for every single measurement. We select distinct nutrients where the `feed_key` is contained in the selected ids. This query has been changed from the old implementation. It still does the exact same thing, but readability has been improved significantly. Go to the 'Improved queries' section for more info.

For summary data standard nutrients, we also use the table `d_nutrient` and select all the nutrients except for ones that don't have a specie and group specified (only concerning 4 nutrients).

Query result attributes:
`(key, description, specie_name, specie_id, group_name, z_order, selected)`

The algorithm to create the tree for both detail and summary nutrients needs 3 attributes from the row: group_id, specie_id and name of the nutrient. It then constructs a tree with 3 levels. the first level consists of the different species. The second level of the different groups. In the groups as leave nodes we have the actual name of the nutrient. The algorithm uses a helper hashtable `tempRoot` to access previously created lists in the `tree` variable, which will contain the endresult.

From these queries we get a list or tree of feed objects with the attributes: nutrient_key, name and selected.

getProperties.js

Queries for time and geo options are stored in this file.

The four queries are very similar. We look at the vc/vr_cube (joined fact_table) again. Search for matching feeds and nutrients and get a list of seasons/years/cantons/altitudes.

1.6.3 queries.js /api/queries

We have 3 operations for queries:

POST /api/queries. This is where it tries to save a query. The body should be the same as query.categories, but with a description included. It first checks the authorization header and gives a 401 error response if the user is not allowed to save queries. It then tries to execute the function postQuery defined in postQuery.js file.

GET /api/queries/?id=queryId. This end point gets a single query with the matching id. The response object has the same members as query.categories.

GET /api/queries. This end point returns 3 groups of queries. Sponsored queries, queries defined and made available from the admin and the personal queries.

postQueries.js/getQueries.js

Saving queries is pretty simple with the function `saveQuery(query, username, userlevel)`. We save the values given in the database if the user is authorized to do so.

For getting a single query, we have to find the row in the database. Since all the attributes of the query in the database as plain text (even numerical values), we first have to convert those strings into the fitting javascript types. This is done with the function processStringsIntoJS(row).

Some of the queries have a `description` variable with a '$'. These descriptions must be split in two, the first part before the dollar sign set to `description`, the other part set to `explanation`.

Getting sponsored, admin and user queries should be trivial.

Note: Availability numbers => 0 means everyone, 2 means all subscribers and 3 means admin or creator of the query. 1 doesn't exist.

1.6.4 login.js /api/login

Searches for a user with matching credentials. If successful, create a JWT with username and userlevel as payload. JWT encoding and decoding functions in /server/auth/local.js. It returns a success boolean, a userlevel and the token. Everytime a user successfully logs in, another query gets executed to update the login counter and last login timestamp.

TODO: The two queries should be refactored into two functions. The password has yet to be encrypted with md5. All the current passwords in the database are encrypted.

1.6.5 Results /api/results

Detail data

For quering the detail data we first get the columns/nutrients and then the measurements/rows which are included in the selected search options, put them together and send it back to the client. Be aware that a single row only represents one measurement/-value. The client handles the table construction.

NOTE: Derived nutrients for detail data requires a special implementation of Postgres. This query hasn't been tested on its functionality at all. For more detail look at the errors section.

Summary data

Getting the columns for summary data is almost the same. Getting the rows is a bit more complicated.

First the formulas for calculating derived nutrients (nutrient derived from other nutrients) is queried and saved in the list `formulaList`. Then another list gets created, `involvedNutrients`, to store the ids of the nutrients needed for calculating the formulas.

After the measurements for selected nutrients AND additional nutrients for the formulas are retrieved and saved into measurementsHash[feedkey][nutrientkey], we try to evaluate the formulas with data from the hash and also save the new values to measurementsHash. In the end we get the values from measurementsHash that we want and send it back to the client.

getDetailResults.js

`getNutrients(options, lang)`: The nutrients are retrieved from the `fact_table_clean`. Any other tables with information needed are joined.

`getNutrientsDerived(options, lang)`: Derived nutrients aren't working right now. See error section.

Columns/nutrients result attributes:
`(id, an(alysis)_id, abbreviation, unit, an(alysis)_name)`

Rows/measurements result attributes:

```
(lims_number, id, an_id, priority, avg_quantity, day,
   postal_code, origin_key, latitude, longitute, altitude,
   season, canton, feedname, feedkey)
```

Retrieving measurements are done with the function `getRows(options, lang)`. The subquery randomly selects 150 sample keys with the feeds and nutrients selected. Then all the other information of these 150 samples are joined and aggregated.

getSummaryResults.js

The query for getting the nutrients for summary data (getNutrients(options, lang)) is split into three parts. First it gets all the nutrient information from the ones selected, then it removes all the nutrients which are not dry matter if fresh matter is selected (or vice versa). At the end it adds nutrients which are selected and are not dry matter not dry matter if fresh matter is selected (or vice versa. same as before.).

The purpose of this 3 part query is somewhat confusing, but it works. It probably can be simplified by cutting the first two parts, but no testing has been done for this matter at this moment.

The query for formulas formulas(options, lang) is pretty simple except for the regexp_replace, where it checks the formula for invalid characters. This is important because the code in expanded_formula_eval gets executed with eval().

The query for getting the measurements is also straightforward. It joins the tables summary_data, d_nutrient and d_feed and selects the relevant data.

Formula result attributes:

```
(feed_key, nutrient_key, expanded_formula_eval (actual php
   code), involved_nutrients_ids, correct)
```

Measurement result attributes:
`(nid, raw_value, feedkey, fname)`

1.7 Additional notes

This application is programmed with the old implementation as the template. While only the core functionality has been included in this new application, parts of the it has been improved while trying to copy the old one.

1.7.1 Missing functionalities

Fresh/dry matter, raw, radius selection

The interface of the search-options controller is not able to set these options yet and have to be included. However, the server should already be able to handle theses options.

These options should be fields in query.categories object with the following variable names: ”fresh” as boolean, ”raw” as boolean and ”radius” as a number.

Range search

In the old implementation, the user can specify a range for the nutrients after the query is executed. It filters out the measurements from the results which don’t satisfy the conditions.

It hasn’t been implemented on either server or client except in the end point for saving queries. These conditions are named rng0 to rng4 in the database.

Saving/deleting query

While the server end point for saving queries exists, the client misses the functionality of making a request to that end point. It should send the selected ids of the queries.categories object along with a description string to the server.

The name of the description string should be ”description_en” while the ”en” must be replaced based on current language in session.language.

Logout / keeping login state

Currently there’s no way to log out once logged in except for reloading the page. A button should be implemented that resets the session service and clears the http authorization header.

On the other side, the session service and the http header shouldn’t be reset once the page gets reloaded. These information should be saved in the local storage or in a cookie of the browser.

Additionally, the server should send a 403 (or alternatively 440) http code if the JWT token has expired. The client should prompt the user to log in again, getting a new JWT token.

Authorization control

The response from the server endpoints should differ based on the userlevel of the user. Right now the userlevel usually does not get checked. An example how the userlevel is taken from the token is in the /api/param, where the functions to encode/decode JWTs are stored int /server/auth/local.js.

It also should check the userlevel on the client, stored in the session service. Based on the userlevel of the current user, specific parts should be shown/hidden.

Layout / Design

Almost nothing has been done to improve the asthaethics of the website.

Static elements on the home page

Static elements like info, glossary or sponsors are not included yet

Advanced visualization

The results are shown in a simple table. More Visualizations like google maps, more google charts are missing.

1.7.2 Query error

```
SELECT DISTINCT
id_nutrient_fkey AS id,
id_nutrient_analyses_fkey AS an_id,
d_nutrient.abbreviation_en AS abbreviation,
d_nutrient.unit_measure_en AS unit,
d_nutrient_analyses.name_en AS an_name
FROM fact_table
JOIN d_time ON fact_table.id_time_fkey = d_time.time_key
JOIN d_origin ON fact_table.id_origin_fkey = d_origin.origin_key
JOIN d_nutrient ON fact_table.id_nutrient_fkey = d_nutrient.nutrient_key
JOIN d_nutrient_analyses ON fact_table.id_nutrient_analyses_fkey = d_nutrient_analyses.nutrient_analyses_key
WHERE
    d_m_b <> true AND
    (( Feed id IN (selected feed ids) ))
ORDER BY abbreviation, an_name
ERROR:  column "id_nutrient_analyses_fkey" does not exist at character 41
```

This error occurs if a query with detail data and raw data selected is executed. For getting the measurements from the database, the table "fact_table_clean" is used when working without raw data. This table holds a an attribute "id_nutrient_analyses_fkey". However, when working with raw data, the table "fact_table" is used, which does not have the attribute. This causes an error while querying.

1.7.3 Improved queries

Almost every query to retrieve database information has been changed in some way. The majority of changes are ommitting attributes which are not used. Other queries have been changed significantly while maintaining the same functionality.

Formula query for summary data

```
SELECT
id_feed AS feed_key,
nutrient_fkey AS nutrient_key,
regexp_replace(expanded_formula_eval, 'coalesce\(.[^+*/()-]{1,30}','(','g' ) AS expanded_formula_eval,
involved_nutrients_ids,
correct
FROM
    t_formula_feed
    JOIN t_formula ON t_formula_feed.id_formula = t_formula.id
    JOIN d_feed ON d_feed.feed_key = t_formula_feed.id_feed
WHERE
    t_formula.nutrient_fkey IN ('+conditions.nutrients+') AND
    d_feed.feed_key IN ('+conditions.feeds+') AND
    t_formula.abbr_generic_de IS NOT NULL AND
    trim(t_formula.abbr_generic_de)) NOT LIKE '%'+freshString+']"
```

This query has been modified in a way that it gets all the required formulas at once instead of executing a query for each formula. This is done with the line
`t_formula.nutrient_fkey IN ('+conditions.nutrients+')`,
which evaluates to true if any nutrient id of the formula matches with any of the selected nutrients. Before, it would execute this query in a for loop for every single selected nutrient, only getting one formula each loop.

Nutrient options for detail data

```
SELECT DISTINCT
nutrient_key AS key,
abbreviation_'+lang+'AS name,
description_'+lang+' AS description,
specie_name_'+lang+' AS specie_name,
specie_id,
group_name_'+lang+'AS group_name,
z_order AS z_order,
CASE WHEN nutrient_key IN ('+conditions.nutrients+')
    THEN true ELSE false END AS selected
FROM d_nutrient
WHERE nutrient_key IN (
    SELECT distinct nutrient_key
    FROM '+options.dataType+'_cube
    WHERE feed_key IN ('+conditions.feeds+')
```

```
)
ORDER BY specie_id, group_name, z_order
```

This query is functionally identical to the one in the previous implementation, except that it uses an `DISTINCT` keyword to ommit identical rows instead of grouping and aggregating.

Nutrient options for summary data

```
SELECT
nutrient_key AS key,
abbreviation_'+lang+' AS name,
description_'+lang+' AS description,
group_name_'+lang+' AS group_name,
specie_name_'+lang+' AS specie_name,
group_id,
specie_id,
CASE WHEN nutrient_key IN ('+conditions.nutrients+')
  THEN true ELSE false END AS selected
FROM d_nutrient
WHERE
  group_name_'+lang+' IS NOT NULL AND
  specie_name_'+lang+' IS NOT NULL
ORDER BY z_group_order, z_order
```

The old query joined four tables (`vs_classified_nutrients`, `d_nutrients`, `vs_classified_nutrien` and `vs_classified_nutrient_groups`) together, when all the needed information was already availabe in d_nutrient.

1.7.4 Performance

All the testing is done with a MacBook Pro 13" 2014 8GB I5-4278U. Both server (with the database) and client are run on the same machine natively in maxOS Sierra version 10.12.3.

Detail data

selected feed keys (cereal grains): 751, 749, 750, 1321, 799, 756, 879, 741

selected nutrient keys (ruminants): 248, 258, 256, 254, 247, 234, 235, 238, 203, 99, 237, 183, 133, 136, 126, 124, 125, 5, 6, 148, 151, 162

	Old app	New app
Server response	9.1s	4.33s
Table drawing	6.3s	3.5s

Note that the old implementation also has to draw the map and graph, which could affect the table drawing time. However, the response from the server was only 4.2 MB, whereas the new one was 10 MB.

Number of rows: 1858

Summary data

Selected feed ids (every roughage): 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 866, 938, 939, 940, 941, 942, 943, 944, 945, 946, 933, 934, 935, 936, 937, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 986, 985, 989, 988, 987, 992, 990, 991, 995, 993, 994, 1322, 998, 997, 996, 999, 1000, 1001, 1003, 1004, 1005, 1002, 1007, 1006, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1020, 1019, 1021, 1022, 1023, 1024, 1025, 1026, 1028, 1027, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1362, 1363, 1364, 1365, 1366, 1344, 1345, 1353, 1354, 1326, 1327, 1335, 1336, 1367, 1368, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1038, 1039, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1050, 1051, 1052, 1053, 1054, 1065, 1066, 1067, 1037, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1101, 1103, 1102, 1105, 1104, 1108, 1106, 1107, 1323, 1109, 1111, 1110, 1112, 1113, 1114, 1115, 1117, 1116, 1118, 1119, 1120, 1121, 1122, 1124, 1123, 1125, 1126, 1128, 1127, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1417, 1418, 1419, 1420, 1421, 1399, 1400, 1408, 1409, 1381, 1382, 1390, 1391, 1422, 1423, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1129, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1154, 1155, 1156, 1157, 1158, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1468, 1469, 1470, 1471, 1472, 1459, 1460, 1443, 1444, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1227, 1228, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1229, 1230, 1231, 1232, 1233, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1100, 1130, 1226, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1282, 1281, 1279, 1280, 1283, 1290, 1288, 1284, 1294, 1295, 1285, 1291, 1289, 1286, 1293, 1287, 1292, 794, 1296, 1297

Selected nutrient ids (all nutrients): 112, 180, 158, 144, 163, 160, 159, 142, 166, 174, 231, 132, 1, 2, 137, 279, 13, 100, 236, 46, 75, 76, 96, 138, 17, 78, 81, 80, 167, 79, 83, 85, 19, 21, 23, 27, 29, 33, 37, 48, 60, 72, 170, 25, 31, 35, 39, 50, 62, 74, 127, 41, 43, 45, 52, 54, 56, 58, 64, 66, 68, 70, 154, 283, 285, 284, 286, 149, 291, 116, 77, 82, 84, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 98, 161, 282, 171, 128, 155, 287, 290, 289, 288, 150, 293, 292, 121, 120, 123, 122, 87, 86, 177, 176, 179, 178, 115, 114, 119, 118, 147, 146, 182, 181, 190, 189, 8, 7, 111, 110, 4, 3, 11, 10,

105, 104, 107, 106, 153, 152, 169, 168, 9, 88, 145, 157, 129, 117, 141, 165, 91, 101, 94, 233, 130, 113, 92, 173, 131, 223, 12, 228, 229, 230, 224, 226, 227, 225, 143, 156, 102, 16, 90, 15, 232, 248, 258, 256, 254, 247, 234, 235, 238, 280, 281, 203, 99, 237, 183, 133, 136, 5, 6, 148, 151, 162, 126, 124, 125, 259, 240, 187, 205, 251, 261, 257, 255, 249, 14, 195, 188, 135, 272, 273, 268, 275, 276, 270, 271, 274, 277, 278, 267, 269, 216, 217, 212, 219, 220, 214, 215, 218, 221, 222, 211, 213, 252, 193, 184, 185, 186, 245, 246, 241, 262, 263, 243, 244, 253, 264, 265, 239, 242, 201, 202, 192, 207, 208, 198, 200, 204, 209, 210, 191, 197, 266, 196, 250, 260, 194, 206, 175, 139, 97, 89, 103, 164

	Old app	New app
Server response	7.8s	28.03s
Table drawing	7.5	5.5s

Number of rows: 389

The old implementation is multiple times faster than the new one. The new query can probably optimized.