Durham Foodbank

Release 1.0.0

Group 12

Apr 09, 2020

CONTENTS

1 System Overview **1**

1.1 Backend Overview 1

1.2 Frontend Overview 4

2 Installation **7**

2.1 Authentication 7

2.2 Backend 8

2.3 Database 8

2.4 Frontend 9

3 Expansion **11**

3.1 API Expansion 11

3.2 Authentication Expansion 14

3.3 Backups 15

3.4 Database Expansion 15

CHAPTER

1 SYSTEM OVERVIEW

1.1 Backend Overview

1.1.1 System Structure

The whole system composes of three primary components: the MongoDB Database, the Express Server (which I will also refer to as the back-end server) and any number of clients running the front-end program.

The MongoDB database is where the data describing the contents and structure of the warehouse will be stored so that it can be accessed and edited by warehouse employees. We chose to use MongoDB as it is both easily extensible and flexible. Details on the exact structure of the database can be found in the Installation section.

The express server acts as a middle-man between the clients and the MongoDB and provides a number of useful API functions to interface with the database. It provides error handling between the client and server, and between the server and the database, ensuring that the whole system remains robust and to ensure consistency. Details on how to install and deploy the express server are also in the Installation section.

1.1.2 Back-end Functions

In this section I will give a brief overview of all back-end functions in `routes/stockTake.js`

addTray

When provided with a tray object and a MongoDB database object, this function will add the contents of the tray object to the database.

editTray

When provided with a tray object and a MongoDB database object, this function will update the contents of an existing tray object in the database, provided it exists.

removeTray

When provided with the location of a tray object and a MongoDB database object, this function will delete the tray from the database.

switchTray

When provided with the location of two different tray objects and a MongoDB database object, this function with swap the positions of two trays in the database.

addTrayMany

When provided with an array of tray objects and a MongoDB database object, this function will add all the tray objects to the database in a single command. This should be used if many trays are added at once, as adding them one-by-one will take an unacceptable amount of time.

editTrayMany

When provided with an array of tray objects and a MongoDB database object, this function will edit all the tray objects that are in the database in a single command. This should be used if many trays are edited at once, as editing them one-by-one will take an unacceptable amount of time.

removeTrayMany

When provided with an array of tray locations and a MongoDB database object, this function will remove all the trays at the positions specified. This should be used if many trays are deleted at once, as deleting them one-by-one will take an unacceptable amount of time.

getAllCategory

When provided with a category and a MongoDB database object, this function will return all trays with a matching category. This can be used to quickly obtain the locations of items of a specific type.

getNextNExpiring

When provided with a number N, an optional category argument and a MongoDB database object, this function will return the next N expiring trays. If a category is specified, it will return the next N expiring in that category only. This can be useful if we wish to find the items closest to expiring so that we can focus on taking them from the warehouse first.

getZones

This function, when provided with a MongoDB database object, simply returns all zones in the warehouse.

addZone

When provided with a zone object and a MongoDB database object, the zone will be added to the database. This can be used in the initial creation of the warehouse and to add zones temporarily when there is a need to meet a higher than normal capacity.

editZone

When provided with a zone object and a MongoDB database object, the zone in the database will be edited, provided that it does exist.

removeZone

When provided with a zone location and a MongoDB database object, the zone will be removed from the database.

addBay

When provided with a bay object and a MongoDB database object, the bay will be added to the database.

editBay

When provided with a bay object and a MongoDB database object, the bay will be edited, provided it does exist.

removeBay

When provided with the location of the bay and a MongoDB database object, the bay will be removed from the database.

getTraysInBay

When provided with the location of a bay and a MongoDB database object, the function will return a list of all tray objects inside that bay, provided the bay exists. This will be used to display bay contents in the front-end application.

getBaysInZone

When provided with the location of a zone and a MongoDB database object, the function will return a list of all the bay objects inside that zone, provided the zone exists.

moveTray

When provided with two tray locations and a MongoDB database object, the function will move the tray from one location to another provided a tray exists in the start location and does not exist in the end position.

mongoUpdate

This function takes a request body and a method code. It will first connect to the MongoDB database to get the database object. Then, using the method code, it will pass the request body to one of the functions described above. It is then responsible for handling errors that occur and returning the result to the front-end.

1.2 Frontend Overview

1.2.1 Intro

The front end is built with ReactJS an object orientated system aimed at building great user interfaces.

- It has a unique structure where in each item seen on the ui is effectively a react class component.
- These components contain subcomponents. some of these have been built buy us, others have been imported from a boot-strap component library.
- We use two of these, Grommet and React bootstrap, these are primarily used for buttons and styling that have the benefit of improving user experience and decreasing development time.

The integral components that have been built by us are:

1.2.2 Designer

The component is a tool to build the warehouse, add remove, resize zones and their respective bays, this is a very simple component visibly, however functionally it is integral. This component uses Api Endpoints:

- addTray
- removeTray
- addTrayMany
- removeTrayMany
- getZones
- addZone
- editZone
- removeZone
- addBay
- editBay
- removeBay
- getTraysInBay
- getBaysInZone

1.2.3 Reports

This component takes and manipulates the data received from the api, the component interfaces with the server. This component uses Api Endpoints:

- getNextNExpiring
- getAllCategory
- getZones
- getTraysInBay
- getReports
- publishReport

1.2.4 StockTake

This component acts as a data structure, fetching and syncing with the server, the sub components below display the data stored in StockTake. The component itself not at all complex, however its interactions with the sub components and them with each other is quite a lot more complex than the sum of the api calls. This component uses Api Endpoints:

- editTrayMany
- editBay
- getZones
- getTraysInBay
- getBaysInZone

Bayview

This component displays a collection of tray items; which represent individul trays, it stores a list of the selected trays, and allows the manipulation of tray items, by means of category buttons and several forms.

CategoryButtons

this component displays and handles the change of category.

trayItem

a graphical representation of an individual tray and the event handler for when an individual tray is selected.

CHAPTER

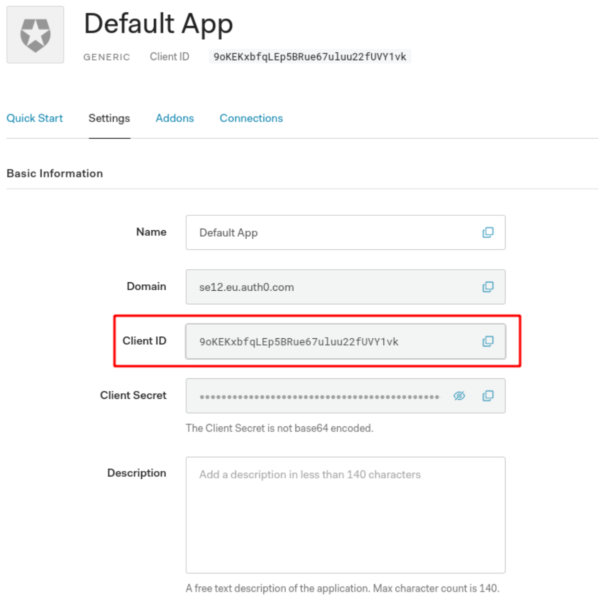
2 INSTALLATION

2.1 Authentication

Authentication for the website is handled through Auth0 the free tier allows for up to 7000 users, which should be more than ample.

The process of transferring the account to one that you own is very simple:

1. Sign up for Auth0 here
2. Under the applications tab on the dashboard create a new Generic application
3. Copy the client ID from the application, as pictured below



Default App

GENERIC Client ID 9oKEKxbfqLEp5BRue67uluu22fUVY1vk

Basic Information

Name: Default App

Domain: se12.eu.auth0.com

Client ID: 9oKEKxbfqLEp5BRue67uluu22fUVY1vk

Client Secret: ••

The Client Secret is not base64 encoded.

Description: Add a description in less than 140 characters

A free text description of the application. Max character count is 140.

4. Paste this value under `clientID` in `auth_config.json`, which is located in the `src` folder of the `frontend` server
5. Also change the domain in the `auth_config.json` file, this domain can also be found in the application, as shown below
6. Disable signups by going to the connections tab of the dashboard, clicking on `Username-Password-Authentication`, then turning on the disable signups option as shown below

This is done as otherwise any user could come and sign up for the system without permission

2.2 Backend

In our handover there is a folder named `backend`, in this is stored the server that manages the requests made in the website. It is built using Node.js and Express and can be started with `npm install` followed by `npm start`, the server will then be hosted on port **3000**

When both the front end have been started, requests can be made between them

2.2.1 Deployment

Our recommendation for deploying this is on heroku as it has a good free tier and has suitable pricing plans for expanding if usage increases.

It is also simple to set up:

1. Create an account here
2. Click the new button on your dashboard to create a new application
3. Give it a name and set the region to Europe
4. Follow the on screen instructions to get your code deployed

2.3 Database

For this application to run it also needs a database set up, for this we have chosen MongoDB.

2.3.1 Deployment

Our recommendation for deployment is to use MongoDB Atlas this has a free tier but allows for scaling up to a paid plan if your requirements exceed the limits of the free tier.

To get started sign up here then choose any of the cloud providers, and select a local cluster with a free tier

Once this is done, a particular structure of the database will need to be established if you don't want to change the code

1. Create a database named `foodbank`
2. Create 4 collections, `bays`, `dummy`, `food` and `zones` in that database

If you do want to change the names of the database or collections, please consult the expansion section of this user guide.

2.4 Frontend

In our handover there is a folder named `frontend`, in this is stored the user interface with the program. It is a react app, and can be started with the command `npm install` followed by `npm start`, you will then be taken to a browser window with the application loaded.

2.4.1 Deployment

The code can be deployed using any static hosting, our recommendation is ZEIT Now as they have a very generous free tier and it is very easy to get set up. There are two main ways of getting set up

2.4.2 Git Repository

If you don't know what this is, then proceed to the other option.

For continuous development, this is a better option as any changes you make will be automatically deployed to the website.

This can be done in 3 steps

1. Add the files to a GitHub, GitLab or Bitbucket repository
2. Create an account here
3. Link together the repository and ZEIT by clicking `Import Project`

2.4.3 Command line

1. Install npm using the instructions here
2. Create a ZEIT account here
3. Install the Now CLI using the instructions here
4. The code can then be deployed with the command `now`, as documented here

CHAPTER 3 EXPANSION

3.1 API Expansion

Once the handover has been completed, you may wish to add your own API endpoints to expand the functionality of the application further. In this document I will detail how to do this. This document is aimed towards technical users who have some experience with javascript before.

3.1.1 Writing Back-end Functions

Most of the back-end functions are written so that they can be tested with only a connection to a MongoDB server (No connection to the front end required). Connections with the front end and MongoDB server are handled by other functions in `routes/stockTake.js`, so when writing back-end functions we can assume that we have an existing connection to a MongoDB server and that some other function will handle communicating the result to the front end.

With this in mind, all the back-end functions take parameters `body` and `dbo` only, where `body` is a JSON object representing the request body and `dbo` is a MongoDB database object. The parameters of `body` will depend on the function being implemented. When writing new back-end functions, it is a good idea to emulate this, however it is not enforced.

Back-end functions are typically in the following format:

```
async function functionName(body, dbo) {
    // Check if the body is well formed
    if (!checkBody(body)) {
        return "FAIL";
    }

    try {
        // Build a request if the request is more complicated
        let req = buildRequest(body);

        // Call one, or many MongoDB functions
```

(continues on next page)

(continued from previous page)

```
        let res = await dbo.collection("food").someMongoDBFunction(req);

        // Handle the response, returning "FAIL" if necessary
        handleResponse(res);

    } catch (ex) {
        return "FAIL";
    }

    // either return "SUCCESS" or some data that was requested.
    return "SUCCESS";
}
```

For most cases this rough structure will be sufficient. The function must return either "FAIL", "SUCCESS" or the requested data in order to be handled correctly by its calling functions.

3.1.2 Routing Requests

Within `routes/stockTake.js` there is a function called `mongoUpdate` that handles the calling of the created back-end functions, as well as the connection to the MongoDB server. It also handles the returning of data and error codes to the correct API endpoint. In order to use this for your new function, you will need to add an entry to the switch statement contained within this file.

This is simple to do, add the following code:

```
switch "functionID":
    code = await functionName(body, dbo);
    break;
```

to the switch statement.

The next step is to add a new router endpoint. This differs depending on what type of function you have implemented.

Functions that get something with no parameters

```
router.get('/functionEndpoint', async function (req, res, next) {
    let result = await mongoUpdate(req.body, "functionID");
    res.setHeader('Content-Type', 'application/json');
    res.status(200).send({'result': result});
})
```

Functions that get something with parameters

```
router.post('/functionEndpoint', async function (req, res, next) {
    let result = await mongoUpdate(req.body, "functionID");
    res.setHeader('Content-Type', "application/json");
    res.status(200).send({'result': result});
})
```

Functions that make changes and only expect success/failure as result

```
router.post('/functionEndpoint', async function(req, res, next) {
    let code = await mongoUpdate(req.body, "functionID");
    if (code !== "SUCCESS") {
        res.sendStatus(400);
    } else {
        res.sendStatus(200);
    }
})
```

Your back-end function should now be able to serve requests from the client.

3.1.3 Making Requests

We now need to add the end-point we have created to the public API so we can access it from the front-end server. The code for accessing the existing endpoints is in `public/api.js`. When this is imported by a front end program or part of the html of a website the functions can be called from there.

API functions are less strictly formatted compared to the back-end functions and can take any number of parameters. They should be written in a way that assists the developer when writing code by abstracting the awkward nature of hand writing JSON bodies.

Typically, the API functions should follow the following structure:

```
async function functionName(arg_1, arg_2, ..., arg_n) {
    let req = buildBody(arg_1, arg_2, ..., arg_n);
    let res = await fetch(URL + functionEndpoint, {
        method: 'GET or POST',
        mode: 'cors',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(req)
    });

    if (res.ok) {
        return "OK";
    }
    return "FAIL";
}
```

3.1.4 Recompiling

Javascript is a scripting language and does not require recompiling after changes have been made. However, changes to the back-end will require a restart of the back-end server before the changes will be reflected. Changes will also need to be redeployed to whatever hosting service you have chosen to use.

3.1.5 Additional Information

The above information should allow technical users to make simple changes to the back-end so they can add their own features. For more complex changes, you should consult the MongoDB documentation (for communication with the MongoDB database) and the express server and node documentation (for communication between the front and back end).

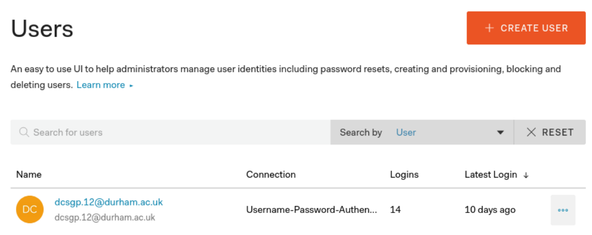
3.2 Authentication Expansion

Before expanding the authentication, it should first be installed, this can be done by following the guide in the installation section of this user manual.

3.2.1 Adding and removing users

One of the most common functions when expanding will be to add and remove users. This can be done from the Auth0 dashboard.

On the sidebar, click on the **Users & Roles** tab, this will take you to the following screen



An easy to use UI to help administrators manage user identities including password resets, creating and provisioning, blocking and deleting users. Learn more ▸

Name	Connection	Logins	Latest Login ↓
DC dcsgp.12@durham.ac.uk dcsgp.12@durham.ac.uk	Username-Password-Authen...	14	10 days ago

Adding a user can be done by clicking the `CREATE USER` button, and removing a user can be done by clicking the three dots next to a given user.

3.2.2 Allowing social integration

Social integration is a very popular choice, such as signin with Google as it means you don't have to remember passwords.

This can easily be done in Auth0 by clicking on the **Connections** tab followed by the **Social** tab, then enable the switch of the social providers you want.

3.2.3 Customising the login page

This is something that would most likely be done when the company logo is updated to ensure brand consistency.

From the dashboard, click on the **Universal Login** tab, and this will allow you to specify a company logo and colours.

3.3 Backups

This page will detail how to set up backups for your database if you have chosen MongoDB as your database of choice.

3.3.1 Setup

It is possible to create backups and restore data using the tools provided with MongoDB. However, this requires knowledge of command line instructions and will likely be difficult for users without this knowledge. You can find more information here on how to perform backups in this way.

The second way of creating backups is using MongoDB Atlas. However, this service is not available on the free tier (M0) and will require a paid plan, the pricings of which can be found here.

If you choose an M2 or M5 cluster, Atlas will create daily snapshots of the cluster. They will be taken automatically, starting 24 hours after the creation of the cluster. These snapshots can then be viewed in a table. From there, you can restore or download your existing snapshots. Atlas will retain the last 8 daily snapshots. The full documentation on this can be found here.

If you require custom policies (e.g. weekly backups), then you will need an M10 or larger cluster. It is then possible to create different backup policies and modify the retention time of these snapshots. Additionally, you can take on-demand snapshots, which occur immediately instead of at regular intervals. Full information can be found here

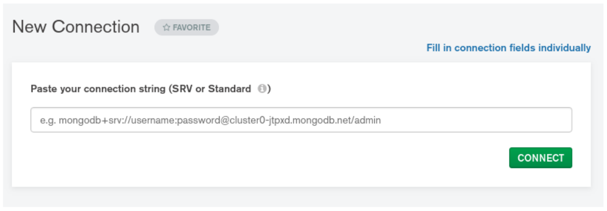
Please note that you will incur extra costs for Cloud Provider Snapshots if you are using an M10+ tier.

3.4 Database Expansion

The database we have chosen is MongoDB some details on the basic setup of this are included in the database installation section.

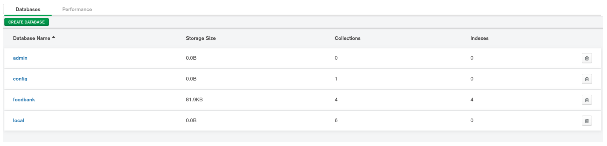
The easiest way to expand the database is using MongoDB Compass which is available for all operating systems. The full documentation for this software is available here.

When you open the program you will be greeted with this input:


New Connection
☆ FAVORITE
Fill in connection fields individually
Paste your connection string (SRV or Standard)
e.g. mongodb+srv://username:password@cluster0-jtpxd.mongodb.net/admin
CONNECT

We will include this URL in the handover package, it can also be found in the `routes/stocktake.js` file of the backend assigned as `URL`.

Once you have connected, you will see a screen showing the databases, below is a screenshot of what this looks like for the default setup.



Database Name	Storage Size	Collections	Indexes
admin	0.0B	0	0
config	0.0B	1	0
foodbank	81.9KB	4	4
local	0.0B	6	0

If you want to add more features to the database you can use `CREATE DATABASE` to add a new database for storing more information.

In each database there are collections, which store your data, you can access the collections in a database by clicking on the name of the database. If you want to add more collections, this can be done here in the same way as adding a database.

By clicking on a collection, you can see the stored data, this is useful for testing purposes as it allows you to immediately see the changes made to the data.