Documentation

0. Forward

AFFORDABLE is a revolutionary user-centered online platform which implements a systems based healthcare approach to improving the lives of vulnerable socioeconomic populations. It is one portal, one application and one database that will automatically connect individuals who have been negatively impacted by COVID-19 to a community network to reduce barriers to healthcare access and resource distribution.
Our application has two primary goals

1. Simplify and improve the acquisition of resources and financial support to those who have been financially and physically impacted by COVID-19.
2. Promote crowd-funding, small Health-Utilizing Grants (HUGs) to increase immediate financial assistance.

Our application allows users in need of funding for healthcare purposes to send an application directly to AFFORDABLE, and explain their need as well as upload supporting documents. If accepted, the recipient can accept their funds into a Stripe Connected Account and register their bank account using Plaid, which allows them to withdraw the funds directly to their bank account. Administratively, AFFORDABLE can review all applications and support materials and award funds from their Stripe account in a streamlined fashion.

1. How to run

These instructions are for the `voila-trust-1-remerge` branch of Affordable.

1.1 Initial setup 🌅

1. For best results, use Bash. We've had better results with it than PowerShell. We only worked with Windows and Linux, so we don't know how zsh compares on new Macs.
    a. *Git Bash* is a Bash shell for Windows that comes bundled with the installation of Git: https://git-scm.com/downloads
2. Install one of the two installers, Homebrew or Chocolatey.
    a. Install Homebrew (https://brew.sh/) if using Git Bash in Windows, Linux, or Mac:
    ```
    sh -c "$(curl -fsSL https://raw.githubusercontent.com/Linuxbrew/install/master/install.sh)"
    ```
    b. Install Chocolatey if using PowerShell on Windows: https://chocolatey.org/install
3. Through Homebrew or Chocolately, install the following packages:
    a. yarn

   b. mysql
   c. nodejs
   d. git (Although this should already be present natively if you installed Git Bash)
4. mysql allows database viewing and manipulation via the terminal, but, if you'd like a GUI for the database, install MySQL Workbench:
   https://www.mysql.com/products/workbench/
   a. On Windows, if it doesn't start when you open it, you might need to install the C++ Redistributable for Visual Studio 2019:
      https://aka.ms/vs/16/release/vc_redist.x64.exe
5. ⚠️ 🐧 ⚠️ **ONLY IF ON LINUX**, you ***MUST*** set the `lower_case_table_names` mysql variable to 1!!!
   a. This applies both when running locally on Linux and deploying to Linux (e.g., to AWS).
      Due to filesystem differences, on Unix (Mac and Linux), file names are case sensitive; on Windows, they're *not* case sensitive. Historic inconsistencies of SQL table names between the Affordable database and codebase mean that the Affordable backend runs on Windows by default but doesn't run on Linux (Mac has a workaround already built-in, but we never ran it on a Mac to confirm).
         Sources:
         https://stackoverflow.com/a/6134059
         https://dev.mysql.com/doc/refman/8.0/en/identifier-case-sensitivity.html
      Solution: setting the `lower_case_table_names` variable on Linux internally sets all the table names to lowercase.
   b. Set the variable via the mysql config file, `my.cnf`:
      i. If you installed mysql with Homebrew, it *should* be at:
         `/home/linuxbrew/.linuxbrew/etc/my.cnf`
         1. If it's not there, you can find where with:
            ```
            mysqladmin --help | grep -A 1 "Default options"
            ```
            You should see a list of possible locations, like:
            ```
            Default options are read from the following files in
            the given order:
            /etc/my.cnf /etc/mysql/my.cnf
            /home/linuxbrew/.linuxbrew/etc/my.cnf ~/.my.cnf
            ```
            Look through those.
      ii. Editing `my.cnf`, add:
         ```
         lower_case_table_names = 1
         ```
         For example, mine looks like this:
         ```
         # Default Homebrew MySQL server config
         [mysqld]
         # Only allow connections from localhost
         bind-address = 127.0.0.1
         mysqlx-bind-address = 127.0.0.1
         ```

```
# Workaround for case-sensitive table names on Windows
lower_case_table_names = 1
```

   iii. Check that the var was set by entering mysql and running:

   ```
   show variables;
   ```

   After a little scrolling, you should see:

   ```
   lower_case_table_names | 1
   ```

   iv. Next, restart the mysql daemon, mysqld

      1. If mysqld is already running, kill it:

         ```
         ps aux | grep "mysqld"
         ```

         You should see something like:

         ```
         jdoe 123000  0.1  0.6 2996224 432600 ? ...17:43
         0:28 mysqld
         jdoe 444278  0.0  0.0   6288  ......grep
         --colour=auto mysqld
         ```

         According to this output, mysqld's PID is `123000`.
         Kill it via its PID:

         ```
         kill -9 123000
         ```

      2. If you already had old SQL data in your mysql folder, mysqld *won't* restart, so delete them:

         ```
         rm -rf /home/linuxbrew/.linuxbrew/var/mysql/
         ```

         (Replace with your directory if different.)

      3. Restart mysqld with the initialize option:
         mysqld --initialize

      4. If you already set mysql privileges (steps 1.1.6.*), redo those steps.

      5. If you already generated the Affordable database (steps 1.2.5.*), redo those too.

6. Set MySQL root privileges.

   a. Enter mysql as root:

   ```
   mysql -u root -p
   ```

   Hit Enter (no password by default).
   If you get `ERROR 2002 (HY000)`, the mysql daemon probably isn't started. Start it with:

   ```
   mysqld
   ```

   At the `mysql>` prompt, run:

   ```
   ALTER USER 'root'@'localhost' IDENTIFIED WITH
   mysql_native_password BY 'password';
   ```

   And then:

   ```
   flush privileges;
   ```

   Now, your sql root password is "password". In the future, log in to mysql as root with:

   ```
   mysql -u root -ppassword
   ```

   (Yes, no space between -p and password…)

7. Create the .env files (these hold environmental variables used by Affordable).
   In `affordable/server/`, create a file named "`.env`" (be careful; Windows Explorer loves to hide extensions or mislead about what they are) and fill it with:

```
AFFORDABLE_DB_HOST=localhost
AFFORDABLE_DB_USER=root
AFFORDABLE_DB_PASSWORD=password
AFFORDABLE_DB_NAME=Affordable

AFFORDABLE_SES_KEY_ID=<foo>
AFFORDABLE_SES_KEY_SECRET=<foo>
AFFORDABLE_SES_REGION=us-east-1

AWS_ACCESS_KEY_ID=<foo>
AWS_SECRET_KEY_SECRET=<foo>
AWS_BUCKET_NAME=ou.messenger.test

AFFORDABLE_EMAIL_ENABLED=false
AFFORDABLE_FRONTEND_URL=http://localhost:3000
AFFORDABLE_BACKEND_URL=http://localhost:4000

AFFORDABLE_ADMIN_USER=admin
AFFORDABLE_ADMIN_PASSWORD=password
AFFORDABLE_ADMIN_EMAIL=admin@affordhealth.org
AFFORDABLE_ADMIN_ID=1
AFFORDABLE_TOKEN_SIGNING_KEY=secret
AFFORDABLE_PAYOUT_ACCOUNT=<foo>

STRIPE_SECRET_KEY=<foo>
PLAID_CLIENT_ID=<foo>
PLAID_SECRET=<foo>
PLAID_PUBLIC_KEY=<foo>

MAX_FILE_SIZE=25000000
UPLOAD_FOLDER=testFolder // Folder within S3 bucket
```

   Similarly, in `affordable/app/`, create an `.env` and fill it with:

```
REACT_APP_STRIPE_PUBLISHABLE_KEY=<foo>
REACT_APP_PLAID_CLIENT_ID=<foo>
REACT_APP_PLAID_SECRET=<foo>
REACT_APP_PLAID_PUBLIC_KEY=<foo>
REACT_APP_AF_BACKEND_URL=http://localhost:4000
```

Finally, in `affordable/client/`, create an `.env` and fill it with:

```
REACT_APP_AF_BACKEND_URL=http://localhost:4000
```

1. Among other things, the admin user's name and password can be set in these files. In `affordable/server/.env`, these are in:

```
AFFORDABLE_ADMIN_USER=admin
AFFORDABLE_ADMIN_PASSWORD=password
```

Replace the `<foo>` values with your data as needed.

1.2 Rebuilding and running ♻️

After code changes, you need to recompile the `.../src/` TypeScript files to `.../dist/` JavaScript files (which are what actually run the app).

1. In the top directory, `affordable/`, run:
```
npm install react react-dom react-s3 react-plaid-link plaid moment
@stripe/react-stripe-js @stripe/stripe-js multiparty aws-sdk axios
stream fs blob
```
2. Copy the package files from `known_good_dependency_files` into the top directory:
```
cp known_good_dependency_files/* .
```
⚠️ Failure to do this can cause the Banking Information tab in the Settings page to be blank; it's some kind of inscrutable dependency issue, which is why we contrived this step.
When asked whether to overwrite existing files, say yes.
3. Install and [re]build Affordable:
```
yarn install
```
4. `yarn build`
5. Start frontend. In `affordable/app/`, run:
```
yarn start
```
   a. If you get an error like:
   ```
   ? Something is already running on port 1234. Probably: ...
   ```
   Then you likely forgot to close a previous instance of the frontend or something's already occupying port 1234 (or whatever the port is).
   In Linux and/or perhaps Git Bash, kill whatever's running at that port with:
   ```
   fuser -k 1234/tcp
   ```
6. Start backend.
   a. In `affordable/server/`, first initialize the database:
   ```
   yarn create-affordable-database
   ```
   The yarn command `create-affordable-database` is aliased in `affordable/server/package.json` to… yet another command in another file, `affordable/database/package.json`, and ultimately to the command: `mysql --user=root --password=password < ./CreateDatabase.sql`
   So it just runs the SQL commands in

        `affordable/database/CreateDatabase.sql`, which erases the database if it's already present and creates it if not.

        i. If you get:
           ```
           ERROR 2002 (HY000)
           ```
           Then start mysqld with:
           ```
           mysqld
           ```

    b. Additionally, we've created useful dummy data to test with in the file `affordable/database/voila_dummy_data.sql`. You can load this data into the fresh database with (assuming you're in the `server/` directory):
       ```
       mysql --user=root -p < ../database/voila_dummy_data.sql
       ```
       Also, if for some reason `yarn create-affordable-database` doesn't work, `CreateDatabase.sql` can be imported with this same command:
       ```
       mysql --user=root -p < ../database/CreateDatabase.sql
       ```
    c. Finally, start the backend:
       ```
       yarn start
       ```
7. Listen to webhooks for balances to work and to test Stripe stuff locally:
    a. With backend running, open a CMD or Bash window (can't be PowerShell), and, in the top directory, `affordable/`, run:
       ```
       stripe.exe
       ```
       On Linux or in Bash, do:
       ```
       chmod +x ./stripe.exe
       ./stripe.exe
       ```
       (Like mysqld, this will then occupy that terminal until it terminates; it's good to reserve tmux panes for this stuff.)
    b. Run:
       ```
       stripe login
       ```
       Follow the instructions (just hit Enter).
    c. Run:
       ```
       stripe listen --forward-to localhost:4000/webhook
       ```

1.3 Changing address & port 📡

In theory, one would only need to change the environmental variables in the .env files to change port and URL. It's more complicated than that. The .env files are half the battle; the other half is other files.

1. Change the following line in `affordable/app/.env`:
   ```
   REACT_APP_AF_BACKEND_URL=http://localhost:4000
   ```
2. Change the following line in `affordable/client/.env`:
   ```
   REACT_APP_AF_BACKEND_URL=http://localhost:4000
   ```
3. Change the following lines in `affordable/server/.env`:
   ```
   AFFORDABLE_FRONTEND_URL=http://localhost:3000
   AFFORDABLE_BACKEND_URL=http://localhost:4000
   ```

4. Then *also* change the following files (the values must match between the .env files and these!):
    a. For the frontend,
    `app/node_modules/react-scripts/scripts/start.js`
    @ L60:
    ```
    58| ...
    59| // Tools like Cloud9 rely on this.
    60| const DEFAULT_PORT = parseInt(process.env.PORT, 10) || 3000;
    61| ...
    ```
    The “`3000`” there is the port number to change.
    Again, make sure it matches the port specified in `server/.env`.
    b. For the backend,
    `server/src/init.ts`
    @ L38:
    ```
    34| ...
    35| /**
    36|  * Get port from environment and store in Express.
    37|  */
    38| const PORT = normalizePort(process.env.PORT || "4000");
    39| ...
    ```
    Again, change the “`4000`” to the desired port, and then reflect this change across the three .env files (`server/.env`, `app/.env`, and `client/.env`).

2. Our new pages

2.1 Dashboard: Application

Upon registering or logging in, a recipient is directed to their dashboard. This dashboard is an application form which can be filled out by the recipient to apply to the Covid-19 grant. The recipient is asked to fill out their full name, whether they have tested positive for Covid-19, what their monthly income is, and how much grant money they are asking for. Additionally the recipient is able to upload photo proof which will justify their need for a grant. Furthermore, a recipient is able to share their story with Affordable.

The application upon submit is checked to see if the required fields have been completed before adding the application to the database. Once the fields have been verified, the recipient is presented with a popup notifying them that the application has been sent. It is at this moment that the application details are recorded in the database and the supporting documents are uploaded to S3. To facilitate logging practices, each application has a new HUG tied to it, therefore allowing the admin to award whatever amount they see fit for the given application.

2.2 Dashboard: Admin

Upon logging in, the admin is directed to the dashboard. The dashboard displays a table of all applications awaiting the admin review with basic information: Name, Request Amount, COVID-19 Status, and Monthly Income. For each application, the admin can get more information, reject, or grant the application by selecting that application and clicking on the “View Applicants” button. This generates a popup that displays, in addition to the basic information, more information: the Applicant Story, Request Document, Recommended Grant Amount, and Grant Amount. Applicant Story is the reason why the applicant is applying. The Request Document has a “Download” button that gets a document from S3 that the applicant uploaded. The Recommended Grant Amount is the amount that the admin needs to grant to the applicant for the applicant to get the full requested amount because of the 2.5% AFFORDABLE fee. The Grant Amount has a text field where the admin decided the amount to grant the applicant.

After the admin reviews the application, he/she can reject or grant the application by selecting the “Reject” or “Grant” button in the popup, respectively. When the reject button is selected, another popup is generated to ask for confirmation upon which the admin can select “Cancel” or “Confirm.” When the grant button is selected, the number in the Grant Amount must not be zero, or else an error popup will appear requesting an amount to be entered. If there is already an amount in the Grant Amount field, then a popup will appear to ask for confirmation on whether to proceed with the grant. Once the application is granted, 2.5% of the money is transferred from the admin balance to a Stripe account that holds the fee, and the rest of the money is added to the applicant balance.

In either case of rejecting or granting an application, the database is updated with the status for an application changing and the HUG associated with that application will be deleted. The admin will be returned to the dashboard and the application they rejected or granted is removed from the table.

2.3 Banking Information Tab

The banking information tab is located in the settings page and its use is storing all saved payment methods for the given user. This page also permits the user to add and delete payment methods. Depending on the user, they will be given the option to save both card and banking payments (donor) or just banking payments (recipient).

The saved payment methods table is all card and bank payment methods stored in the Card and BankAccount/ConnectedBankAccount tables in the database. The distinction between the BankAccount and the ConnectedBankAccount is that the connected bank account is required for withdrawals (see section 4.2.2 for further clarification). Finally, the banking information tab is periodically updated to display the most recent saved payment methods.

2.4 Transaction

The transaction page allows a user to view their transaction history. Here a recipient is shown their awarded HUG history along with a history of their withdrawals. Each transaction is displayed in a table for the user with the date the transaction was created, the transaction type, the amount for that transaction, and a status indicating the state of processing that transaction.

The transaction table is all the records tied to the user from the InternalTransaction and ExternalTransaction tables in the database. The transaction page also does a timed update to guarantee that the most recent transactions appear on the table. Furthermore, using webhooks described in section 3.4, the statuses of the transactions get updated when webhooks respond with an event indicating transfer succeeded.

2.5 Withdrawal Component

The withdrawal component is only used by the applicant and before it can be used the bank account needs to be added, described in section 2.3, and verified using Stripe, described in section 4.1. To verify a bank account, the applicant needs to click on the withdraw button on the sidebar after adding a bank account. The applicant will be redirected to another page where they will be entering information to verify themselves. Applicants may be redirected to this page multiple times to ask for more information before their bank account is verified.

The withdrawal component is composed of 3 popups. The first popup asks the applicant to select the saved bank account and the amount to withdraw. The amount to withdraw the field will change color to notify the applicant if the amount they enter is valid. The text field will be green for valid input and red for invalid input. Invalid input are inputs that are not numbered, have more than 2 decimal places, and the amount entered is greater than the balance. There is a submit button that leads to the second popup. This popup shows the amount of withdrawal with the processing fee. Since there is no processing fee of withdrawal, the applicant gets the full amount they enter to withdraw. There is a confirm button that leads to the third popup which is a basic thank you for using the service. The actual transfer of money to the applicant bank account is done when the confirm button in the second popup is clicked.

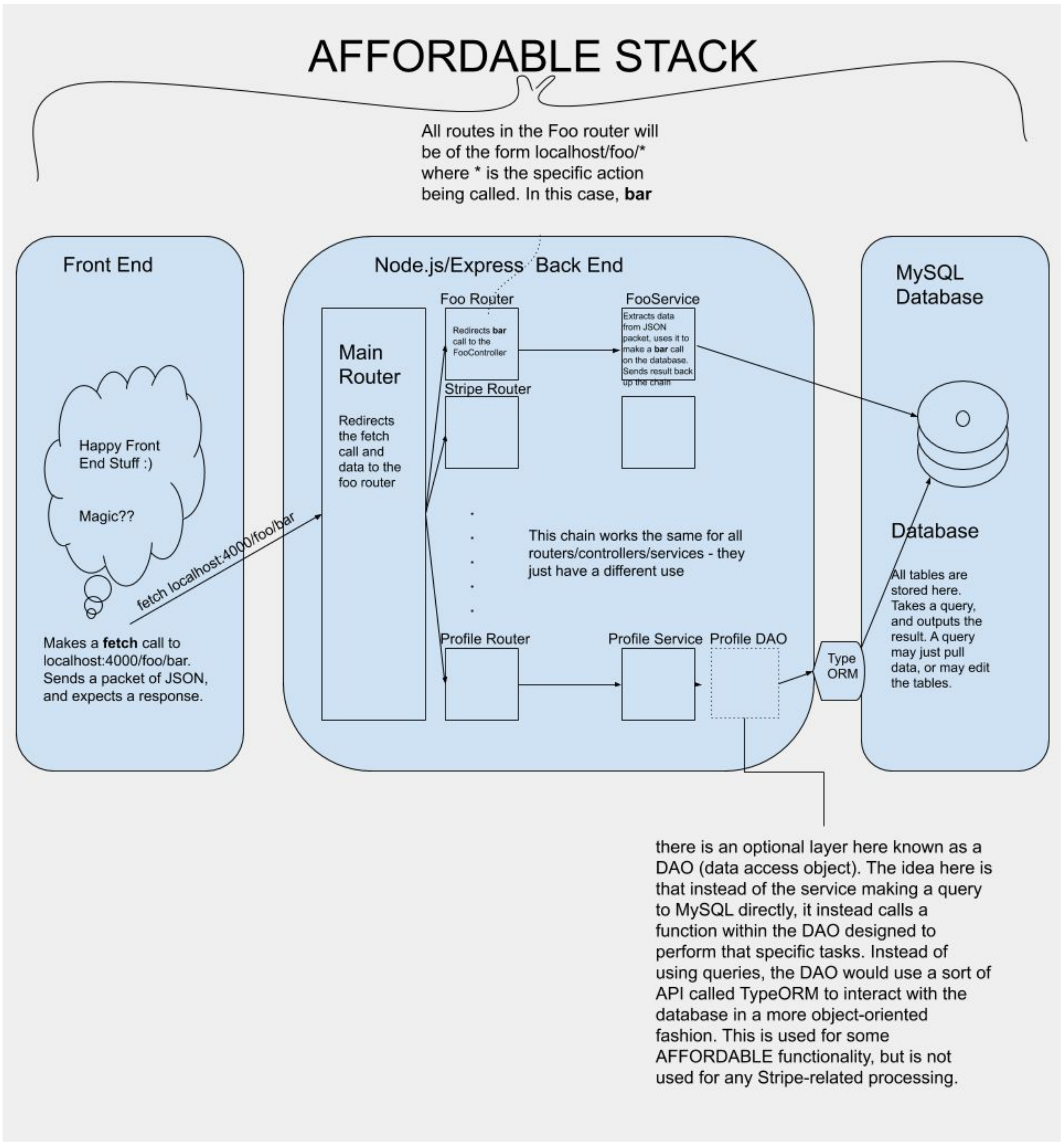
2.6 Donation Component

The original project scope included a donor user and donations within the application. Since the rescoping, this functionality has been offloaded to the AFFORDABLE donation page, available on the main page.

3. Backend Flow and Architecture

3.1 General

The Affordable Node.js/Express server was designed as an *n-tier* architecture, consisting of multiple layers such that the multi-purpose functionality of the back end could be modularized somewhat. For our purposes, all functions related to Stripe and payment processing run through one of multiple 2-layer channels, consisting of a router layer and a service layer. The router layer is responsible for taking a given call from the front-end and delivering the *request* and *result* variables to the intended service function, which in turn performs the specified operation, be it a Stripe API call or a database read/write. A diagram explaining this flow can be found below:



All routes in the Foo router will be of the form localhost/foo/* where * is the specific action being called. In this case, **bar**

Makes a **fetch** call to localhost:4000/foo/bar. Sends a packet of JSON, and expects a response.

Redirects the fetch call and data to the foo router

Redirects **bar** call to the FooController

Extracts data from JSON packet, uses it to make a **bar** call on the database. Sends result back up the chain

This chain works the same for all routers/controllers/services - they just have a different use

All tables are stored here. Takes a query, and outputs the result. A query may just pull data, or may edit the tables.

there is an optional layer here known as a DAO (data access object). The idea here is that instead of the service making a query to MySQL directly, it instead calls a function within the DAO designed to perform that specific tasks. Instead of using queries, the DAO would use a sort of API called TypeORM to interact with the database in a more object-oriented fashion. This is used for some AFFORDABLE functionality, but is not used for any Stripe-related processing.

3.2 Stripe

The Stripe channel consists of the *StripeRouter* and *StripeService*. This channel is chiefly responsible for handling back-end calls related to Stripe API and general stripe logic, be it initializing a recipient connect account, or making a transfer of funds. The *StripeService* file also contains many functions related to Stripe Charges and payment processing, those these are now deprecated in the final project scope.

3.3 Transaction

The Transaction channel also consists of a Router and Service. The Transaction channel is responsible for handling most of the logic with regards to transacting funds internally within the Affordable system, though like before much of these functions are deprecated. The chief responsibility of the StripeService is to keep track of fees and move funds to a user balance internally when a user is awarded funds.

3.4 Application

The Application channel, again using a Router and Service, is used for all functionality regarding submitting, awarding, rejecting, and recording HUG applications.

3.5 Webhook Endpoint

The Webhook Endpoint is responsible for recording data based on hooks sent from Stripe, and making internal changes based on the data within the hooks. All webhooks are directed to the endpoint at */webhook*, but this may be altered on the Stripe dashboard at any time. With the more limited scope of the final project, the remaining hook being processed by this endpoint is *transfer.created*, which informs the server asynchronously when funds have been transferred to a user's Stripe connect account, meaning that the payout process is underway. When this hook is received, the transaction is recorded as *Cleared* in the database's *ExternalTransactions* table, and the associated balance is removed from the user account balance.

3.6 MySQL Database

The MySQL database is an integral component for recording all movements of funds, and storing information about Stripe for API calls.

- **internalTransactions**

Variable	**Description**
`internalTransactionID`	integer value used as default primary key
`from_id`	integer value corresponding to a HUG.
`from_type`	a string value identifying the transaction as a HUG-to-recipient award
`to_id`	integer value corresponding to a recipient.

`amount`	floats value specifying how much funds were involved in the transaction, in US dollars

- **HUGs**

Variable	**Description**
`HUGID`	integer value used as default primary key
`ownerID`	string value used to identify the with a user (will always be admin)
`recipientID`	string value to store the recipientID of the applying recipient
`fundingGoal`	float value corresponding to the requested amount from the recipient

- **StripeRecipient**

Variable	**Description**
`recipient_id`	integer value used as primary key, corresponds to existing account ID
`stripe_id`	string value used to identify the Stripe account ID of the user
`balance`	floating point value corresponding to the amount of funds ready to be withdrawn by the recipient

ExternalTransactions

Variable	**Description**
`externalTransactionID`	integer value used as default primary key
`userID`	string value corresponding to either the recipient receiving the funds
`transaction_type`	a string value identifying the transaction as a Recipient Withdrawal
`paymentMethodID`	string value corresponding to the recipient Stripe account ID
`amount`	float value specifying how much funds were involved in the transaction, in US dollars

`status`	enumerated string value - Pending or Cleared
`creation_timestamp`	datetime value corresponding to when the transaction process was heard by the webhook endpoint

- **Fees**

Variable	Description
`feeID`	integer value used as default primary key
`HUGID`	a string value identifying the awarded HUG which this fee was taken from
`managementFee`	integer value specifying how much funds were taken by the AFFORDABLE management fee

4. Our new Tech

4.1 Stripe

Stripe API allows us to collect donations in the form of ACH payments. Unlike debit and credit card payments ACH payments need further verification from a bank. And Hence this can delay donations. In Order to address this problem, we decided to integrate Plaid api. Plaid api gives us the capability to instantly verify bank accounts without the need to collect and store account and routing numbers. This inturn solves the security issues that arise in transactions. At the same time the transaction service logic implemented will comply with transaction policies such as GDPR.

4.1.1 Connect Accounts and Onboarding

In order to payout funds through Stripe's API, it has to be done through Stripe's Connected Accounts feature. Connected accounts are smaller, sub-accounts that are attached to the main one. The use case for Connected Accounts are for companies that employ independent contractors like ride-sharing services. For AFFORDABLE's purposes, a Connected account is assigned to each potential recipient. Each Connected account contains their banking information for funds to payout to. This can be accessed on the Stripe Dashboard under *Connect>Accounts*

4.1.1.2 Onboarding Process

In order to enable payouts of funds to a recipient, Stripe first must verify the identity of the holder of the Connected account (ie. the recipient). The information that Stripe needs can be found under the Stripe API Account object under *requirements*. Information can be injected manually but AFFORDABLE currently uses Stripe Connect Onboarding. Onboarding is Stripe's automated process of collecting information from the user. An API call is made for a Connected account and Stripe will return a link to a generated form based on the requirements for the user to fill out.

Typically, for an individual, the form will ask for basic information such as name, address and email but it may also ask for full SSN and photo ID later if it cannot verify the user. Stripe may not ask for all this information at once and the user will need to keep checking the form to make sure if Stripe needs more information. On the AFFORDABLE webpage, the user can check if their Connect account needs more information by pressing the "Withdraw" button on the sidebar. If a notification comes up saying that the account needs more information, Stripe is either processing some information already or it needs more info.

When Stripe is completely satisfied with the information and has verified the user's identity, the "Withdraw" button on the AFFORDABLE webpage will pop up a funds withdrawal window.

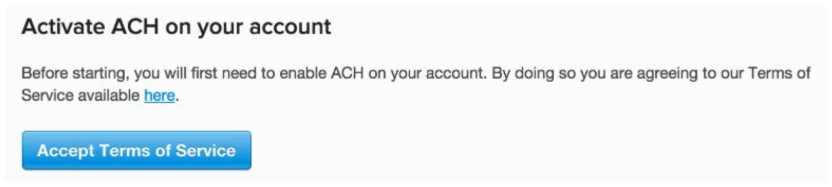
4.2 Plaid

In order to integrate Plaid follow the following step by step instruction.

1. Set up Plaid and stripe accounts

   1.1 You will need to create a Plaid and a Stripe account.

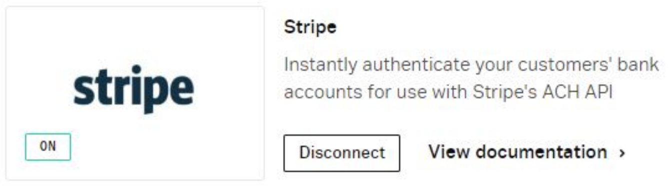
   1.2 Enable ACH access on the Stripe account



Activate ACH on your account

Before starting, you will first need to enable ACH on your account. By doing so you are agreeing to our Terms of Service available here.

   1.3 Enable Plaid account for Stripe Integration. This is located in the Plaid dashboard under Settings =>Integrations.



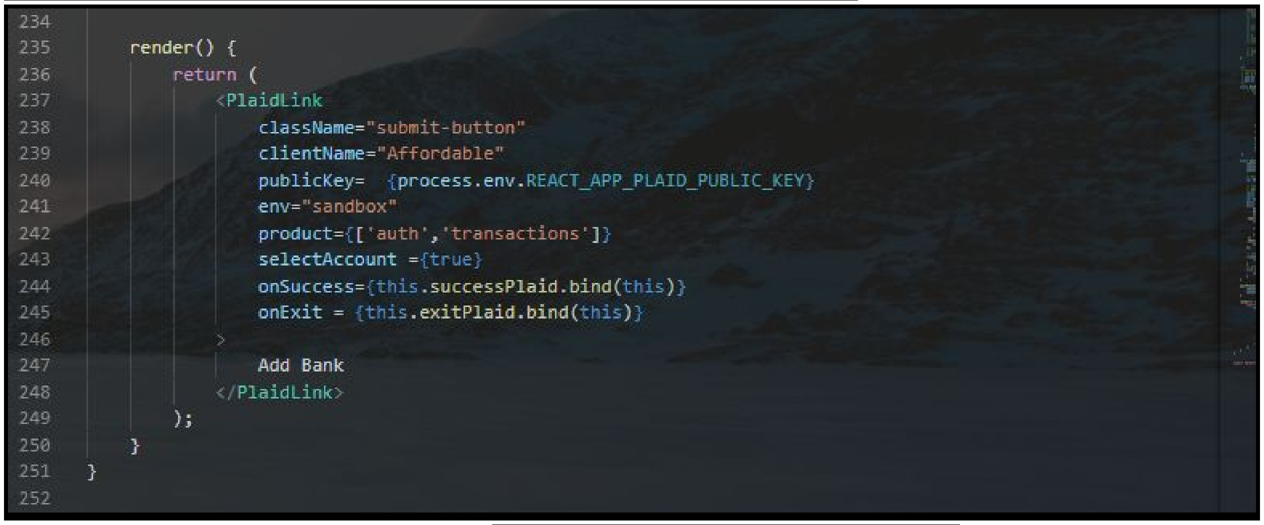
Stripe

Instantly authenticate your customers' bank accounts for use with Stripe's ACH API

   1.4 Get your Plaid public_key from the Plaid dashboard.

The public_key is a non-sensitive public identifier that is used to initialize the Plaid link. The Plaid initialization code is located in

`/app/src/Components/Modal/AddBankaccountButton.ts`



```
    render() {
        return (
            <PlaidLink
                className="submit-button"
                clientName="Affordable"
                publicKey=  {process.env.REACT_APP_PLAID_PUBLIC_KEY}
                env="sandbox"
                product={['auth','transactions']}
                selectAccount ={true}
                onSuccess={this.successPlaid.bind(this)}
                onExit = {this.exitPlaid.bind(this)}
            >
                Add Bank
            </PlaidLink>
        );
    }
}
```

In order for Plaid link to be integrated `npm install plaid-link plaid` must be run in the main directory.

1.3 On Success of Plaid link Integration the Plaid API will return a public_token and metadata.

- We then pass thispublic_token to the backend. The Service functions that use the public token for further API calls are located in Server/src/Services/StripeServices.ts
- Using this public_token we make further Plaid and Stripe API calls in the backend.
- env = " sandbox" should be changed to env = " production" for production.

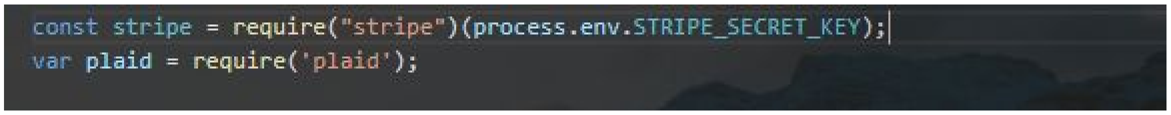
1.4 The Server Side Plaid handler is located in Server/src/Services/StripeServices.ts under the function exchangeTokens(req, res)

- This function will authenticate the Plaid account by using the public_key, client_id, and secret_key.
- Once the Plaid account is authenticate it will a stripe API function call to generate a bank_account_token

4.2.1 Stripe API

- When the public_key is generated the bank account is instantly verified.
- Once we generate the bank_account_token we create a bank account object using the `attachBankToCustomer` function call.
- Once A bank account object is attached to a user, we can charge the customer from the bank account object using the deposit function call.

- The Stripe and Plaid account test keys need to be switched to production keys once testing is done.



```
const stripe = require("stripe")(process.env.STRIPE_SECRET_KEY);
var plaid = require('plaid');
```

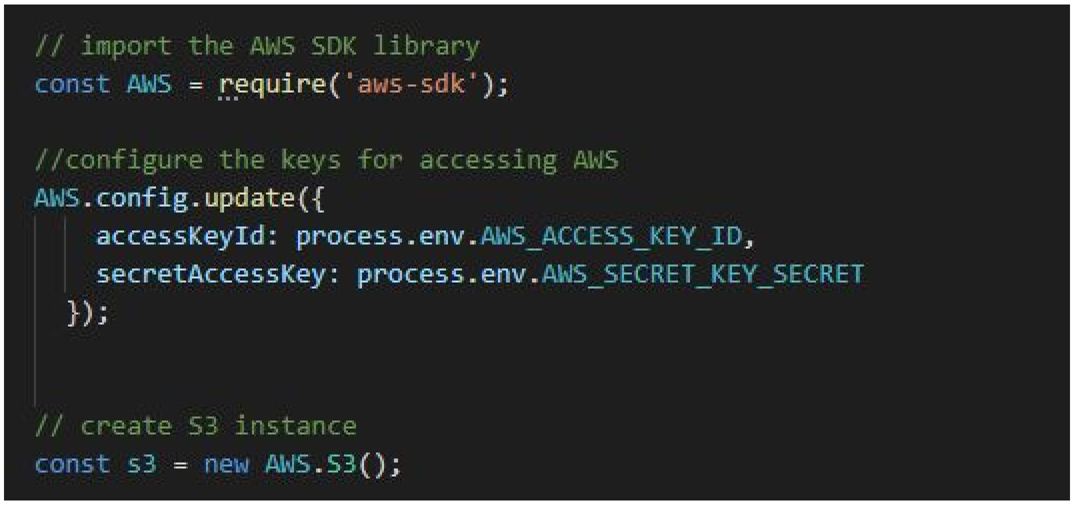
4.3 AWS Simple Storage Service (S3)

During the application process recipients are allowed to upload up to three supporting documents to strengthen their application. Amazon S3 is a preferable storage service for this purpose due it’s scalability, performance and security. S3 supports a storage service for any type of data of any size.

The AWS SDK for NodeJs provides an interface that allows our application to interact with an S3 via api calls. Prior to using the AWS SDK, we have to install the aws-sdk package via npm, and import the library.

4.3.1 AWS Configuration

After importing the aws-sdk library, we can configure our AWS account to use S3. The configuration step requires **AWS Access Key**, and **AWS Secret Key**.These configuration keys are externalized into environment variables. This helps avoid hard coding keys into our code and allows us to easily change the parameters without going through the source code.



```
// import the AWS SDK library
const AWS = require('aws-sdk');

//configure the keys for accessing AWS
AWS.config.update({
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_KEY_SECRET
  });


// create S3 instance
const s3 = new AWS.S3();
```

After configuring our AWS account, we created an S3 instance that supports operations such as putting and getting files from S3. Both put and get operations require a bucket name parameter which is directly being read from our environment variable file.

For security reasons all api calls that interact with our AWS account are implemented on the server side. Therefore we have full leveraged a secure file storage and retrieval system.

4.3.2 Upload

The upload feature uses Express File Upload library to take files from the front-end and pass it to the backend. After the file is passed to the backend, it is converted to a buffer and then the buffer is stored into the S3 using the put api call. To maintain the uniqueness of file keys of the files uploaded to S3, the userID of the recipient and a timestamp of upload time are appended to the original filename. This makes it easier to retrieve files, without conflicting keys, during the download process.

The upload feature limits recipients from uploading a file that exceeds 25mb. In addition to that, we can specify the folder in our S3 bucket where we want our files to be dumped. The maximum file size and the upload folder are configurable by changing the MAX_FILE_SIZE and UPLOAD_FOLDER environment variables in /server/.env.

4.3.3 Download

The download procedure first requires getting the file as a buffer from our S3 bucket. After the buffer is retrieved from our S3, it is piped as part of the response for the fetch call making the download request. Then the download call receives a blob response which gets downloaded in the admins browser.

5. Future Work

5.1 Known Bugs

On occasion, after rebuilding the front end, the Banking Information page will be seen as blank in the browser, with associated errors in the browser console. This is known to be a dependency issue related to the Plaid bank account API, and can be solved by replacing the main directory's *package.json* and *package-lock.json* files with the previously described (section 1.2) known good dependency files, and then fully rebuilding the full stack.

On occasion, the verification credentials will seemingly be lost client-side after returning from Stripe Connect onboarding pages. This can be solved by logging out and logging back in, which will refresh the user's JWT credentials.

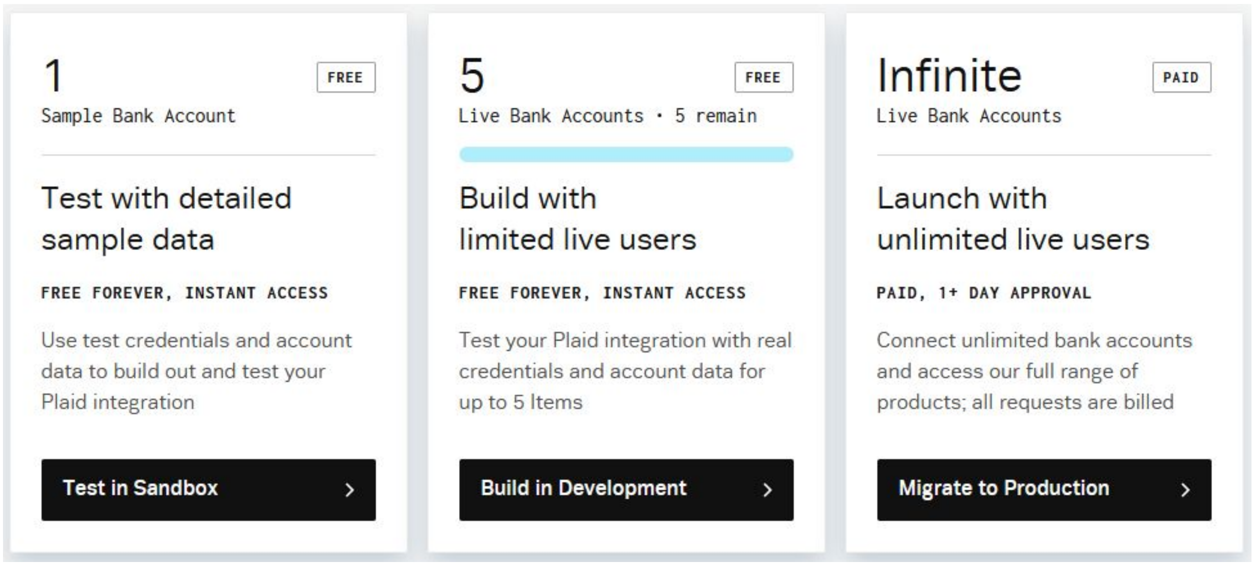
5.2 HTTPS

Currently, Affordable uses HTTP, which makes it unusable in production mode with Stripe and Plaid. We attempted to make it use HTTP but were not successful.
Editing `affordable/server/src/init.ts` seems like the most obvious starting place.
We followed the idioms laid out here: https://stackoverflow.com/a/24371473
Although the backend runs, the frontend cannot communicate with it following our modifications.

5.3 Switching to Plaid Production Environment

5.3.1 Requesting the Production Environment

1. Log into the plaid Dashboard
2. Hit the “Migrate to Production”
3. Follow the steps and fill out the needed information



1 FREE
Sample Bank Account

Test with detailed sample data

FREE FOREVER, INSTANT ACCESS

Use test credentials and account data to build out and test your Plaid integration

Test in Sandbox >

5 FREE
Live Bank Accounts • 5 remain

Build with limited live users

FREE FOREVER, INSTANT ACCESS

Test your Plaid integration with real credentials and account data for up to 5 Items

Build in Development >

Infinite PAID
Live Bank Accounts

Launch with unlimited live users

PAID, 1+ DAY APPROVAL

Connect unlimited bank accounts and access our full range of products; all requests are billed

Migrate to Production >

5.3.2 Changing the Code

Change `app/src/components/Modal/AddBankAccountButton.js` @ L241 from “`sandbox`” to “`production`”:

```
240 | publicKey=  {process.env.REACT_APP_PLAID_PUBLIC_KEY}
241 | env="production"
242 | product={['auth','transactions']}
```