GroceryBuddy

Final Documentation

David Wang (davidew2@illinois.edu)

William Wang (ww4@illinois.edy)

Ray Sun (raysun2@illinous.edu)

Sam Park (syp2@illinois.edu)

Zak Thompson (zcthomp2@illinois.edu)

Fares Talieddine (ftakie2@illinois.edu)

Zach Clewell (clewell2@illinois.edu)

Matthew Ong (mkong8@illinois.edu)

Table of Contents

Description

Trying to find the cheapest and most convenient places to buy your groceries can be hard. Often times this involves memorizing prices and sales at different stores, but by the end you may not know where you are losing or saving money. We intend to crowdsource this pricing information so that users can record the prices of specific groceries at each store via a mobile app. Then this information is aggregated and viewable by users to see where they should go shopping depending on the groceries they need to purchase. This is the basis for our application, “**GroceryBuddy**”.

User Manual

User Instructions

Quickstart

1. Download Expo on an Android device https://play.google.com/store/apps/details?id=host.exp.exponent
2. View project at https://expo.io/@mongy910/gbd
3. Scan QR code with the Expo app

Full Instructions

1. Install npm (can be downloaded from https://nodejs.org/en/download/)
2. Fork https://github.com/rsun0/GroceryBuddy on Github
3. Navigate to /frontend/ and run “npm install”
4. Run “npm start” to launch the expo interface. From here you can either generate an apk or use the Expo app on an Android device to scan the QR code and launch the app.

Main Flow

1. On the home page, create a new shopping list by clicking on the plus button in the bottom right corner and choosing a name.
2. Once your list is created, click on it to launch the search page. Here, you'll be able to add items to your list and compare store prices.
3. To search for an item, either enter text into the search bar or click on the camera button to scan a UPC code for an item you have previously purchased.
4. On the search results page, choose an item that matches what you were looking for.
5. Repeat steps 3 and 4 until you have all your desired items in the list.
6. Click on the "Total" section to compare store totals and select a store.
7. Click on the "Finish Shopping List" button to finish your list.
8. Have fun shopping! Check off items as you go, and upvote/downvote prices that are accurate or inaccurate.

System Administrator Instructions

1. Fork https://github.com/rsun0/GroceryBuddy on Github
2. Create a database on mongoDB Atlas
    a. Sign in at https://www.mongodb.com/cloud/atlas (create an account if necessary)
    b. Create a new cluster
    c. Add the IP of your Azure App Service instance to the IP whitelist tab (after creating the App Service as described below)
    d. Record your connection string URL for future reference (e.g. mongodb+srv://grocery-db-admin:<password>@cluster0-cicgk.azure.mongodb.net/grocery-db?retryWrites=true)

3. Create an Azure App Service
    a. Sign in at https://portal.azure.com (create an account if necessary)
    b. Click 'Add' in the App Services tab
    c. Enter the following options:
        i. Choose an app name (e.g. "grocerybuddybackend")
        ii. Select a subscription
        iii. Create a new Resource Group (e.g. "grocerybuddybackend")
        iv. Choose Linux as the OS
        v. Choose Python 3.7 as the Runtime Stack
        vi. Leave all other options as default
    d. Configure continuous deployment
        i. Select your newly created App Service
        ii. Select the Deployment Center tab
        iii. Choose Github as Source Control
        iv. Choose App Service build service as Build Provider
        v. Select the master branch of your forked Github repository as Code
        vi. Click Finish
    e. Configure application settings in the Configuration tab
        i. Under Application settings, create two new application settings:
            1. Name: HOST_URL, value: your App Service URL (e.g. http://grocerybuddybackend.azurewebsites.net)
            2. Name: MONGO_HOST, value: your Atlas connection string (recorded earlier, e.g. mongodb+srv://grocery-db-admin:<password>@cluster0-cicgk.azure.mongodb.net/grocery-db?retryWrites=true)
        ii. Under General settings, set the Stack settings:
            1. Set the Stack to Python 3.7
            2. Set Startup File to "gunicorn --bind=0.0.0.0 --chdir backend app:app" (without quotes)

    f. Your App Service will now be deployed automatically for every push to master

4. Insert your App Service URL (e.g. http://grocerybuddybackend.azurewebsites.net) into the frontend code
    a. Edit /frontend/utils/api.js
    b. Set "const BACKEND_URL" to your App Service URL, e.g. `const BACKEND_URL = 'https://grocerybuddybackend.azurewebsites.net'`
5. Install the Android app using Expo
    a. Run npm start in the /frontend directory, which will launch the Expo interface locally
    b. Use the interface to publish the app to your phone or generate the APK

Process

Our process to complete GroceryBuddy was designed with the intention to a working product as soon as possible and to enhance and extend this product as the semester progressed. This approach to iterative development was heavily dependent on the User Stories we created at the beginning of the semester. Because we designed out user stories with this goal in mind we were able to continually deliver a product with increasing functionality week after week. As we developed we refactored old code from previous iterations relevant to the current iteration in order to gradually increase the quality of the codebase. In order to maximize developer happiness we opted to write code first and test it later. This goes against XP, however it kept our team interested which was important in order to continually make iteration goals. Because we had a large team working on this project it was important that we had a formal process for committing and merging changes into the master branch. For every iteration each member of the team would have their own branch in the github repository that was focused on implementing a part of a user story. Upon a push to the github repository

Travis CI would be automatically run in order to ensure that nothing added in the commit had broken functionality. Assuming all tests passed, a branch still required approval from a team member separate from the author in order to be merged to master. This strategy of independent branches with checks on merging maximized our ability as a team to work in parallel will ensuring that we maintained a quality codebase.

Requirements & Specifications

User Stories:

- Users can access the GroceryBuddy app on their Android and iOS devices
- Users can add new items to database via UPC code
- Users can update current prices of grocery item
- Users can upvote/downvote prices of grocery item
- Users can add items already in the database to a grocery list
- Users can query grocery items by text
- Users can query grocery items by UPC
- Users can view previous prices of grocery items (item history)
- Users can take pictures of items and submit to database
- Users can compare prices between stores and pick which store they would like to go to
- Users can create a new shopping list
- Users can delete a shopping list
- Users can view a detail page for an item
- Users can check off an item on their grocery list

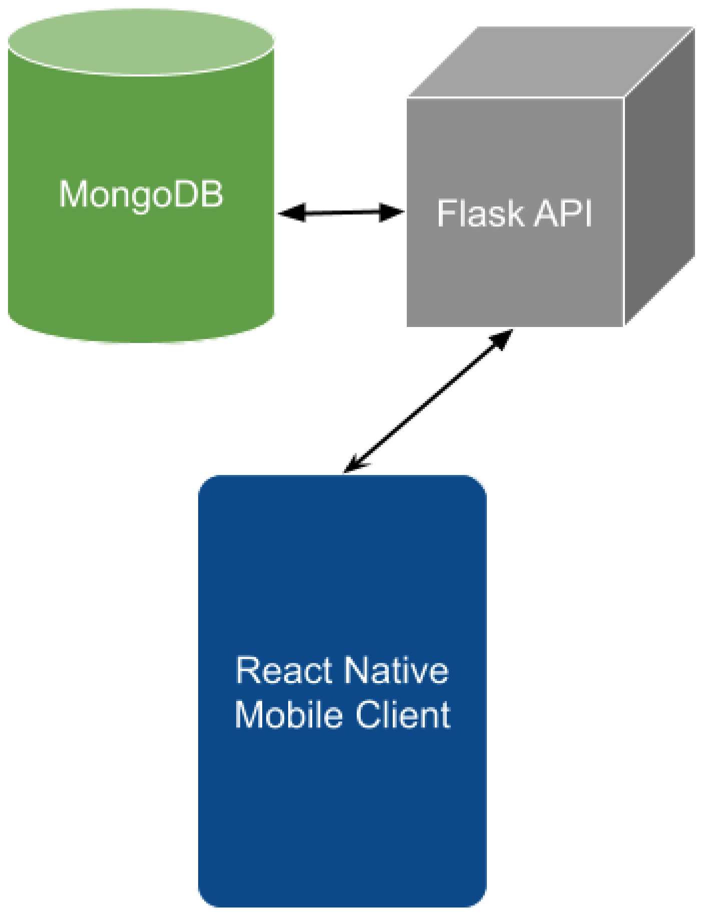
Use Cases:

- View Details About an Item (History of Prices)

- Find Cheapest Single Store for Items
- Upvote/Downvote Price of an Item
- Submit an Updated Price for an Item
- Query Item by UPC Code
- Add New Item to Database
- Query Item by Text
- Add Item to Shopping List

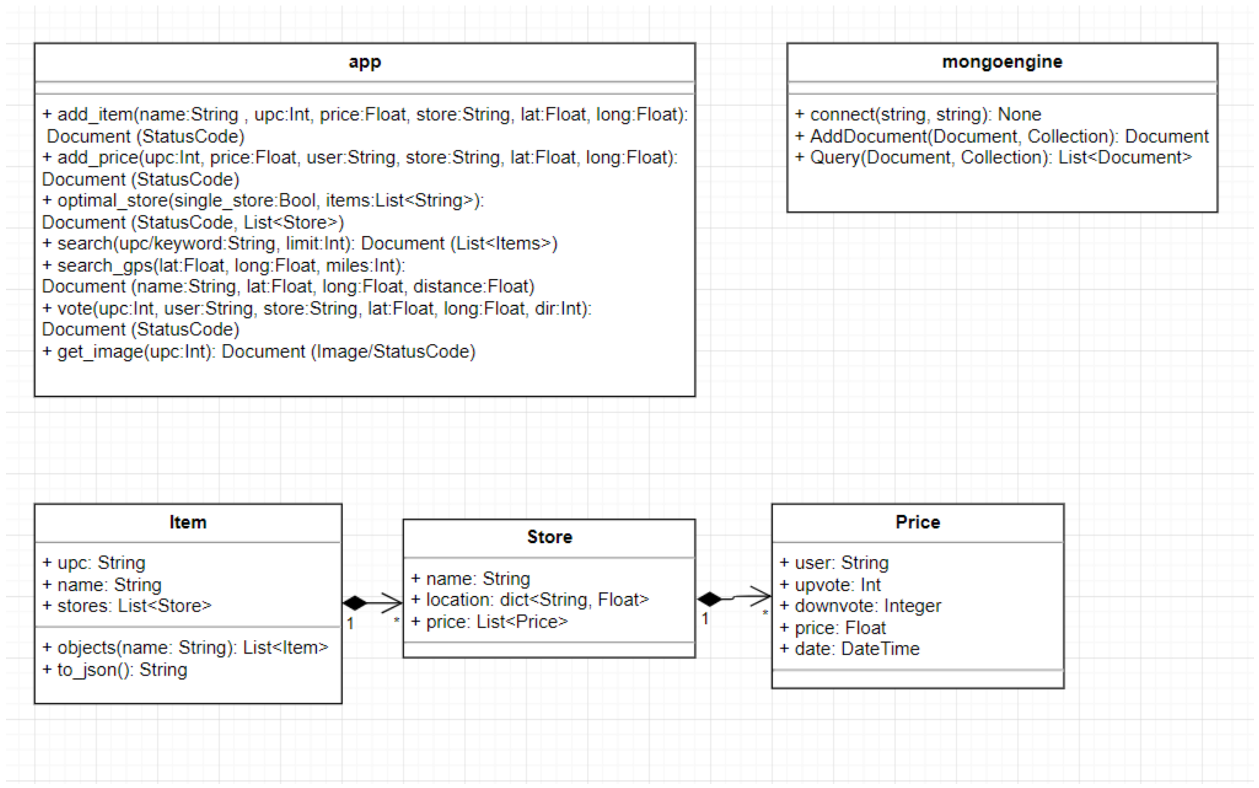
Architecture & Design

Architecture

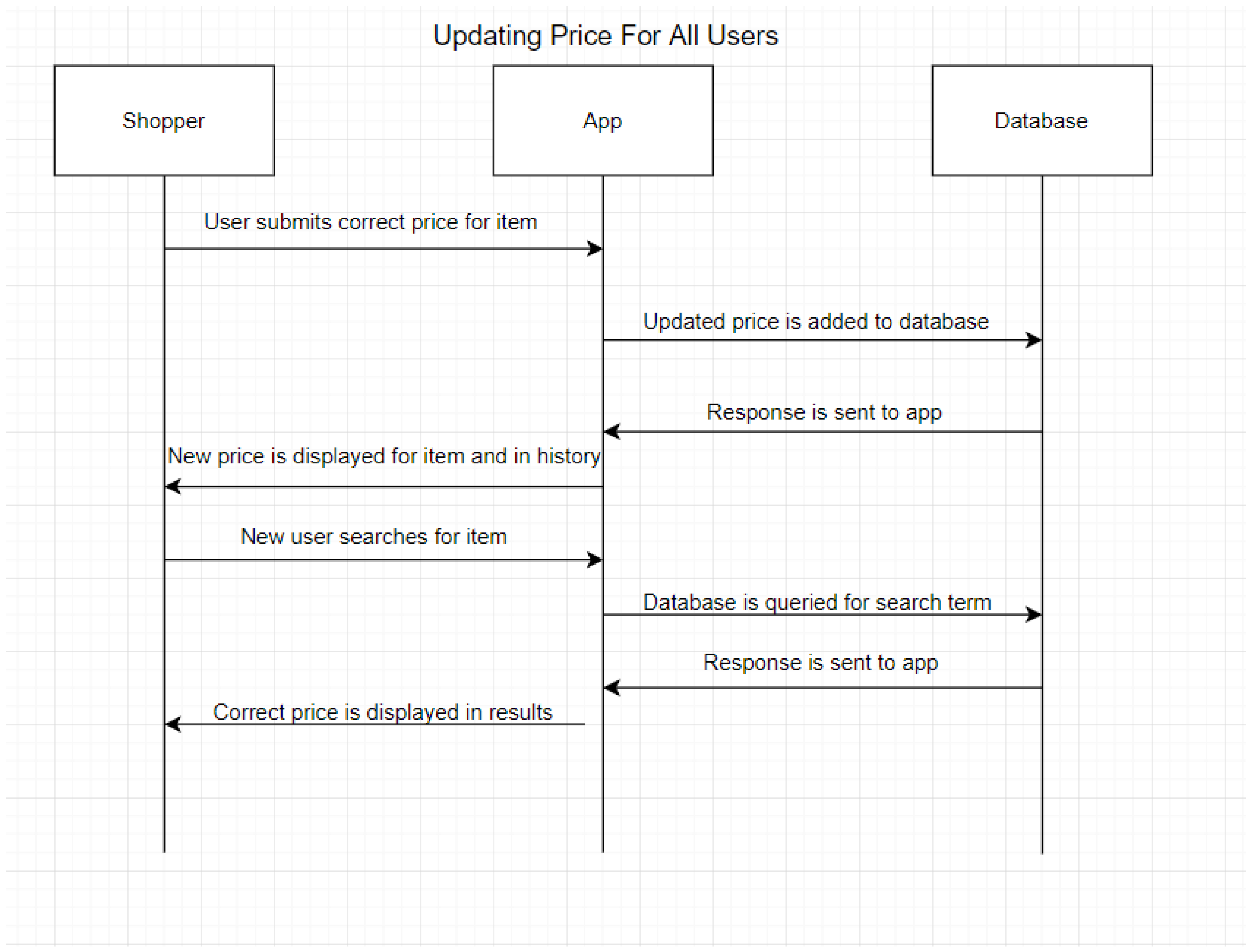


The application has a pretty simple architecture of three components: a MongoDB instance, a vm running a Flask API, and a client running the React Native mobile app. In order for the app to receive or change data within the app it will send a GET or POST request to the Flask API. Then, the API will use its connection to the database to execute database operations for either fetching or changing the data. The API and schema documentation can be found in the Appendix.

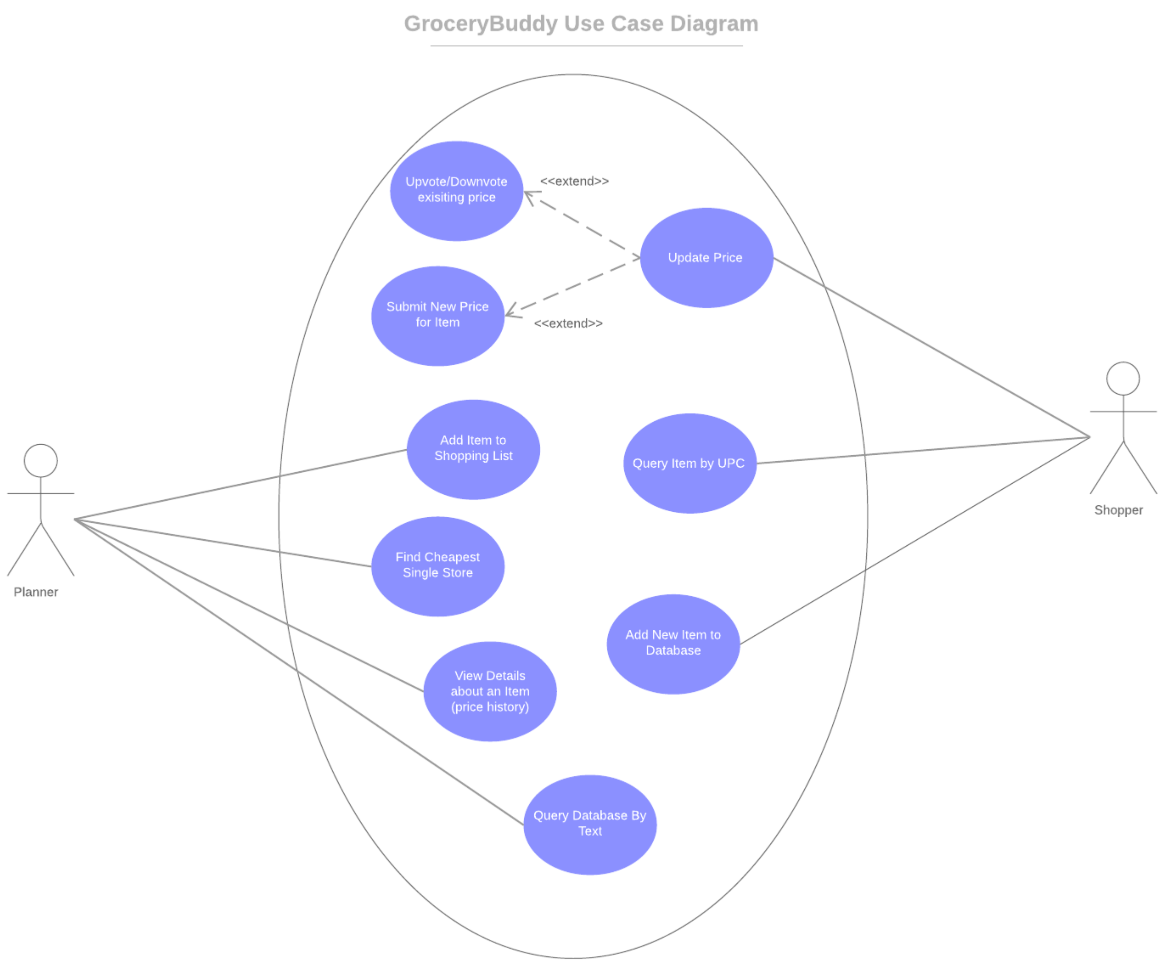
UML Class Diagram


app
+ add_item(name:String , upc:Int, price:Float, store:String, lat:Float, long:Float): Document (StatusCode)
+ add_price(upc:Int, price:Float, user:String, store:String, lat:Float, long:Float): Document (StatusCode)
+ optimal_store(single_store:Bool, items:List<String>): Document (StatusCode, List<Store>)
+ search(upc/keyword:String, limit:Int): Document (List<Items>)
+ search_gps(lat:Float, long:Float, miles:Int): Document (name:String, lat:Float, long:Float, distance:Float)
+ vote(upc:Int, user:String, store:String, lat:Float, long:Float, dir:Int): Document (StatusCode)
+ get_image(upc:Int): Document (Image/StatusCode)
mongoengine
+ connect(string, string): None
+ AddDocument(Document, Collection): Document
+ Query(Document, Collection): List<Document>
Item
+ upc: String
+ name: String
+ stores: List<Store>
+ objects(name: String): List<Item>
+ to_json(): String
1
*
Store
+ name: String
+ location: dict<String, Float>
+ price: List<Price>
1
*
Price
+ user: String
+ upvote: Int
+ downvote: Integer
+ price: Float
+ date: DateTime

UML Sequence Diagram Example


Updating Price For All Users
Shopper
App
Database
User submits correct price for item
Updated price is added to database
Response is sent to app
New price is displayed for item and in history
New user searches for item
Database is queried for search term
Response is sent to app
Correct price is displayed in results

UML Use Case Diagram


GroceryBuddy Use Case Diagram
Upvote/Downvote exisiting price
<<extend>>
Update Price
Submit New Price for Item
<<extend>>
Add Item to Shopping List
Query Item by UPC
Shopper
Planner
Find Cheapest Single Store
Add New Item to Database
View Details about an Item (price history)
Query Database By Text

The design of our system was quite influenced by the choice of our frameworks. For example, choosing MongoDB made sense as it allowed for data to be represented in JSON which made it easy for a JS frontend, like React Native, to easily parse and understand the data. This is because JSON is well understood by JavaScript. Also choosing Flask as the API library/framework made it so our backend had to follow the guidelines of a RESTful API where the client would query the API for all data changes.

Reflections

David: I really enjoyed the time I had this semester to work on GroceryBuddy. Going from just an idea to a functioning application was a process that helped me learn a lot from proper coding standards to efficient github practices. I would say that this was much more rewarding than implementing iTrust on a very old set of technologies (JSP). In contrast, using React Native, Flask, and MongoDB made development feel a lot nicer and left us with a relatively nice looking product. Overall, the semester was very engaging and gave me lots of hands-on experience.

William: I learned a lot this semester working on the GroceryBuddy application. I got to experience what it was like working in a real world scenario on a project, including the time commitments required of each team member to ensure a working product. I also got to learn how to use React Native, Travis CI, Codecov, and the various other software our team used in our project. It’s really satisfying to see the culmination of everyone’s hard work manifested in this application, and this knowledge will most definitely serve me well in the future.

Fares: Starting from CS 427, we learned how to work with an existing code base while following extreme programming processes. In CS 428, we extend the knowledge we gained while also starting a project from scratch. The process of developing a proposal, failing and joining another team was very educational. Furthermore, architecting and discussing a project from scratch with a group is very different from doing it alone. Not only is it much more fun and rewarding, but it also solidifies the concepts we are learning as we get to practice them.

Sam: I don't have much experience with React Native/JS so this project was a great learning opportunity for me. I also learned how difficult it can be to work with 7 other college students when it comes to scheduling and trying to find good times to meet with a larger group of people. I enjoyed how much freedom we got when it came to the actual project. With iTrust, the use cases/user stories were already set for us to implement. Overall, I learned more about the software development process and how extensive the planning can be for that.

Matt: This project was a great learning experience for me. Unlike with iTrust, we had the ability to use modern frameworks that are actually being used in the real world. I gained my first exposure to React Native and CI. In addition, collaborating on a project that we ourselves designed was both challenging and rewarding. It was not easy to estimate what tasks we could or could not complete, but we were able to adapt as we ran into difficulties (for example, moving two people from backend to frontend). Finally, I'm thankful for my teammates, many of whom were more knowledgeable than me about proper project management and JS coding style. I was really able to learn a lot from them.

Zach: I feel like this class was a much better experience than working on iTrust in CS427. Having to create our own user stories as well as our own features was a much more engaging experience. It was also interesting to learn about the CI tools available to developers and to gain experience in them.

Ray: I think I learned a lot from this project, because it many respects it was a real project. Unlike most class projects, we had much more control: we could choose our process, our specification of the project, our tools, etc. I especially appreciated my first experience of using continuous integration and proper branch management. Also, I gained a lot from the experience of working in a team. In the past, my groups for class projects were very ad-hoc and haphazard. I felt like this was the first time I worked in a student team with organization and an established process. I believe this team experience will be very applicable to working in industry.

Zak: Over the course of the semester and this project I feel like I've become a better software developer and learned a lot. Having to build everything up from scratch based on our own idea was much more rewarding as well as a better learning experience than iTrust was last semester. Developing our own use cases and working from there gave a much more well-rounded development experience. I definitely feel that I improved my skills with git most of all.

Appendix

PyDoc

Help on module app:

NAME
    app

DATA

```
   add_item_blueprint = <flask.blueprints.Blueprint object>
     add_price_blueprint = <flask.blueprints.Blueprint object>
     app = <Flask 'app'>
     environ = environ({'TERM_SESSION_ID':
'w1t2p0:78255939-A92...ks/Python...
     get_image_blueprint = <flask.blueprints.Blueprint object>
```

```
    hello_world_blueprint = <flask.blueprints.Blueprint object>
    optimal_store_blueprint = <flask.blueprints.Blueprint
object>
    search_blueprint = <flask.blueprints.Blueprint object>
    search_gps_blueprint = <flask.blueprints.Blueprint object>
    vote_blueprint = <flask.blueprints.Blueprint object>
```

Help on module get_image:

NAME
  get_image

FUNCTIONS
```
    get_image()
```
   This route serves the JPEG image saved for the item with the given UPC.
   The HTTP response resembles the response for a direct link to a JPEG file

(as if the frontend had made a GET request to example.com/photo.jpg).

Body: {"upc"}

Response: image/jpeg if successful

Otherwise,

    {"success": false,
     "error": error description}

DATA

```
get_image_blueprint = <flask.blueprints.Blueprint object>
```

request = <LocalProxy unbound>

Help on module search:

NAME

search

FUNCTIONS

```
search()
```

Returns all items containing the given keyword or

```
        with the given UPC.
        If both UPC and keyword are given, uses the UPC.
        If limit is given, returns at most that number of items.

        Body: {one of ["keyword", "upc"], ["limit"]}
        Response:
            - {"success": true or false},
            - {"error": error description}

DATA
    request = <LocalProxy unbound>
    search_blueprint = <flask.blueprints.Blueprint object>
```

```
Help on module add_item:

NAME
    add_item
```

FUNCTIONS

`add_image(image_b64, item)`

Adds an image to the database.

image_b64 should be a string of an image encoded in Base64.

item should be the Item object to save under.

`add_item()`

Adds a new item to the database.

Body: {"name", "upc", "price", "user", "store", "lat", "long"[, "image", "image_url"]}

Response:

- {"success": true or false},
- {"error": error description}

DATA

`add_item_blueprint = <flask.blueprints.Blueprint object>`

environ = environ({'TERM_SESSION_ID': 'w1t2p0:78255939-A92...ks/Python...

request = <LocalProxy unbound>

Help on module get_image:

NAME

get_image

FUNCTIONS

```
get_image()
```

This route serves the JPEG image saved for the item with the given UPC.

The HTTP response resembles the response for a direct link to a JPEG file

(as if the frontend had made a GET request to example.com/photo.jpg).

Body: {"upc"}

Response: image/jpeg if successful

Otherwise,

{"success": false,

"error": error description}

DATA

```
get_image_blueprint = <flask.blueprints.Blueprint object>
```

request = <LocalProxy unbound>

Help on module utils:

```
NAME
   utils

CLASSES
   enum.Enum(builtins.object)
      Error


   class Error(enum.Enum)
    |  Error(value, names=None, *, module=None, qualname=None, type=None, start=1)
    |
    |  An enum class that holds all custom errors and messages
    |  returned by this app.
    |
    |  Method resolution order:
    |      Error
    |      enum.Enum
    |      builtins.object
    |
    |  Data and other attributes defined here:
    |
    |  ALREADY_DOWNVOTED = <Error.ALREADY_DOWNVOTED: 'User has already
downvo...
    |
    |  ALREADY_UPVOTED = <Error.ALREADY_UPVOTED: 'User has already
upvoted'>
    |
    |  INVALID_DIR = <Error.INVALID_DIR: 'Invalid vote direction'>
    |
    |  INVALID_JSON = <Error.INVALID_JSON: 'Could not parse JSON body'>
    |
    |  INVALID_LIMIT = <Error.INVALID_LIMIT: 'Limit argument was not a positi...
    |
    |  ITEM_DNE = <Error.ITEM_DNE: 'Item does not exist in database'>
    |
    |  ITEM_EXISTS = <Error.ITEM_EXISTS: 'Item already exists in database'>
    |
    |  MISSING_FIELDS = <Error.MISSING_FIELDS: 'Must fill all required fields...
    |
    |  MISSING_KEYWORD_UPC = <Error.MISSING_KEYWORD_UPC: 'Request does
not co...
    |
    |  MISSING_UPC = <Error.MISSING_UPC: 'No UPCs provided'>
    |
    |  NOT_VOTED = <Error.NOT_VOTED: 'User has not voted, cannot undo'>
    |
    |  NO_ERROR = <Error.NO_ERROR: None>
    |
    |  NO_IMAGE = <Error.NO_IMAGE: 'No image uploaded for requested UPC'>
    |
    |  STORE_DNE = <Error.STORE_DNE: 'Store does not exist in database'>
    |
    |  UPC_DNE = <Error.UPC_DNE: 'Some UPCs provided were not found in the da...
    |
    |  ----------------------------------------------------------------------
    |  Data descriptors inherited from enum.Enum:
    |
    |  name
    |      The name of the Enum member.
    |
    |  value
    |      The value of the Enum member.
    |
    |  ----------------------------------------------------------------------
    |  Data descriptors inherited from enum.EnumMeta:
    |
    |  __members__
    |      Returns a mapping of member name->value.
    |
    |      This mapping lists all enum members, including aliases. Note that this
    |      is a read-only view of the internal mapping.
```

HTTP RESTFUL API

GET /search

- **Query parameters**
    - ?keyword
        - A word to look for in the description of an item in the database
    - ?upc
        - UPC code to search
    - ?limit
        - The max number of results to return. Must be a positive integer.
- **Response**

```
{
        [
                {
                        item
                }
        ]
}
```

POST /item

Field Name	**Description**	**Example**	**Required**	**Constraints**
name	short item description	Apples	x	Max length = 64

upc	item upc code	042100005264	x	Length = 12
				Format = ([0-9])*

price	Initial user reported price	5.20	x	Must be a positive num
store	name of store	County Market	x	Max length = 64
lat	latitude of store		x	Valid lat format
long	longitude of store		x	Valid long format
user	username	admin	x	(in the future - existing user)
image_url	URL string (presumably for an	https://photos.google.com/xyz		Max length = 512

	image)			

Response

```
{
        “success”: bool
        “error”: null | list of errors
}
```

POST /price

Field Name	**Description**	**Example**	**Required**	**Constraints**

upc	item upc code	042100005264	x	Length = 12
				Format = ([0-9])*

price	New price to add	5.20	x	Must be a positive num
store	name of store	County Market	x	Max length = 64

lat	latitude of store		x	Valid lat format
long	longitude of store		x	Valid long format
user	username	admin	x	(in the future - existing user)

Response

```
{
        "success": bool
        "error": null | list of errors
}
```

POST /vote

Field Name	Description	Example	Required	Constraints

upc	item upc code	042100005264	x	Length = 12
				Format = ([0-9])*

store	name of store	County Market	x	Max length = 64
lat	latitude of store		x	Valid lat format
long	longitude of store		x	Valid long format
user	username	admin	x	(in the future - existing user)
dir	-1 for downvote, 0 for no vote (to undo a vote), 1 for upvote	-1	x	If 0, user must have already voted

Response

```
{
        "success": bool
        "error": null | list of errors
}
```

POST /optimal_store

Field Name	**Description**	**Example**	**Required**	**Constraints**
single_store	single store or multiple stores returned	True, False	x	boolean
items	list of item upc codes	[042100005264, 042100005367, 049800008264]	x	Must be a list of valid UPCs

Response

```
{
        “success”: bool
        “error”: null | list of errors
        “optimal_prices”: list of objects
                                {‘store’: store obj,
                                 ‘upcs’: list of upcs,
                                 ‘price’: optimal price}
}
```

Backend Dependencies

Can be installed using PIP:

```
Click==7.0
Flask==1.0.2
itsdangerous==1.1.0
Jinja2==2.10.1
MarkupSafe==1.1.0
Werkzeug==0.14.1
pymongo==3.7.2
mongoengine==0.16.3
mongomock==3.15.0
```

Frontend Dependencies

Can be installed using npm

```
"@expo/vector-icons": "^10.0.1",
"@shoutem/ui": "^0.23.12",
"create-react-context": "^0.2.3",
"expo": "^32.0.0",
"jest": "^24.1.0",
"lodash": "^4.17.11",
"node-fetch": "^2.3.0",
"react": "16.5.0",
"react-native": "https://github.com/expo/react-native/archive/sdk-32.0.0.tar.gz",
"react-native-action-button": "^2.8.5",
"react-native-collapsible": "^1.4.0",
"react-native-elements": "^1.1.0",
"react-native-view-transformer": "0.0.28",
"react-navigation": "^3.8.1",
"react-test-renderer": "^16.8.1",
"yarn": "^1.13.0"
"babel-eslint": "^10.0.1",
"babel-preset-expo": "^5.0.0",
"eslint": "^5.16.0",
"eslint-config-airbnb": "^17.1.0",
"eslint-plugin-import": "^2.17.2",
"eslint-plugin-jsx-a11y": "^6.2.1",
"eslint-plugin-react": "^7.12.4",
"jest-expo": "^32.0.0"
```