Design Document

RU Baked?

Scott Jeffery

Charles London

Nick Pieros

Justin Pilla

Kyle Rapps

Ryan Smith

GitHub: <https://github.com/rwsmith/SeniorProject>

Table of Contents

[Technologies Used](#_iz8ep98j81ch) ------------------------------- 2

[Mobile Application](#_eufb75c230cs) ---------------------------------3

[UI](#_a8oaseiztum)

[Application Pages](#_8am2vzeg0m1)

[Model view control](#_96m7l2qkk9ej)

[Bounds](#_q9ahiqpj999h)

[Mock UI Pictures](#_m40p7adio4xq)

[Testing](#_n8nmzkj6y78s)

[Database](#_goxfpfr79iyd)--------------------------------------------11

[Table Specification](#_nvukfscluz91)

[User’s Table](#_q7twrunldipn)

[Merchant Table](#_kl03ak98cah6)

[Store Table](#_ixtyrqusvx1r)

[Orders Table](#_6gw4anaex9b5)

[Order\_Content](#_2b704auzsqmn)

[Product Table](#_lxll9t32ifp3)

[Status Table](#_q6mnsdco0isp)

[Allergens Table](#_92y99w65fo1r)

[Rating Table](#_g8qwngyq2lgc)

[StoreFeedback Table](#_17196ef9yxyk)

[ER Diagram](#_yofx389erna9)

[Database Testing](#_ky80btlguj2f)

[Data integrity](#_wggobhvggioi)

[Base Functionality](#_r0e4rkewp2wp)

[Foreign / Primary Keys](#_y17aj0xngs1i)

[API](#_iachvrhhi390)-------------------------------------------------19

[API Specifications](#_5p2jhtdqqnix)

[Request Types](#_ev0i7hkinl29)

[Response Codes](#_g4o0kwvda4du)

[Endpoints](#_npa7xpzh5pnm)

[Testing](#_esfb91rtfo7x)

[Scalability](#_iaj7pnymbouz)------------------------------------------27

# **Technologies Used**

### React Native

React Native is an open source mobile programming framework, developed at Facebook, that will allow us to develop both native Android and iOS apps simultaneously using JavaScript. Our reason for choosing React Native is that it will allow us to develop a multi-platform app using a single codebase, whereas a traditional approach requires maintaining both an Objective-C/Swift and Android codebase.

### Postgres

PostgreSQL is an advanced, open source relational database. It is customizable with the ability to write custom stored procedures that can be used to perform advanced analytics. Some features not offered by MySQL include common table expressions and window functions. It is increasingly becoming the database of choice for companies where data integrity is paramount. Postgres is available on Amazon RDS, should we want to use the AWS platform in the future.

### Swagger

Swagger is an open source tool for specifying an API. The specification can be turned into beautiful, interactive documentation, as well as both server and client-side code. It also has the ability to be parsed by AWS Gateway which has a nice integration with AWS Lambda, should we want to use the AWS platform in the future (see the scalability section for more on this).

### PayPal API

PayPal offers access to a public API to allow developers to integrate PayPal payments into their websites or mobile apps. We decided to go with the PayPal API because PayPal is a popular method for paying for things electronically and because the API will provide us with a simple method of accepting payments from users. Another reason for choosing the PayPal API is that it will allow us the option to charge credits in the future if we decide to do so.

### Twilio API

Twilio is a SMS messaging service that allows developers to send texts to a specified number using their API. The reason for choosing Twilio is it offers us a simple solution to notifying bakers and users of various orders placed on the app. Twilio will not only be a quick and simple solution but also a very cheap, if not free solution.

1.6. GitHub

GitHub is an excellent platform for using the git version control system. One nice feature we will be using is the code review feature, which allows for comments to be made on code commits.

1.7. Android Studio

Android Studio Google’s software allowing for easy development of Android Applications. We used Android Studio to create a Mock UI to get a sense of how elements will actually be displayed on a phone. The Mock UI is semi-functional (i.e. functioning text fields) to get a feel of how the app will operate during normal usage.

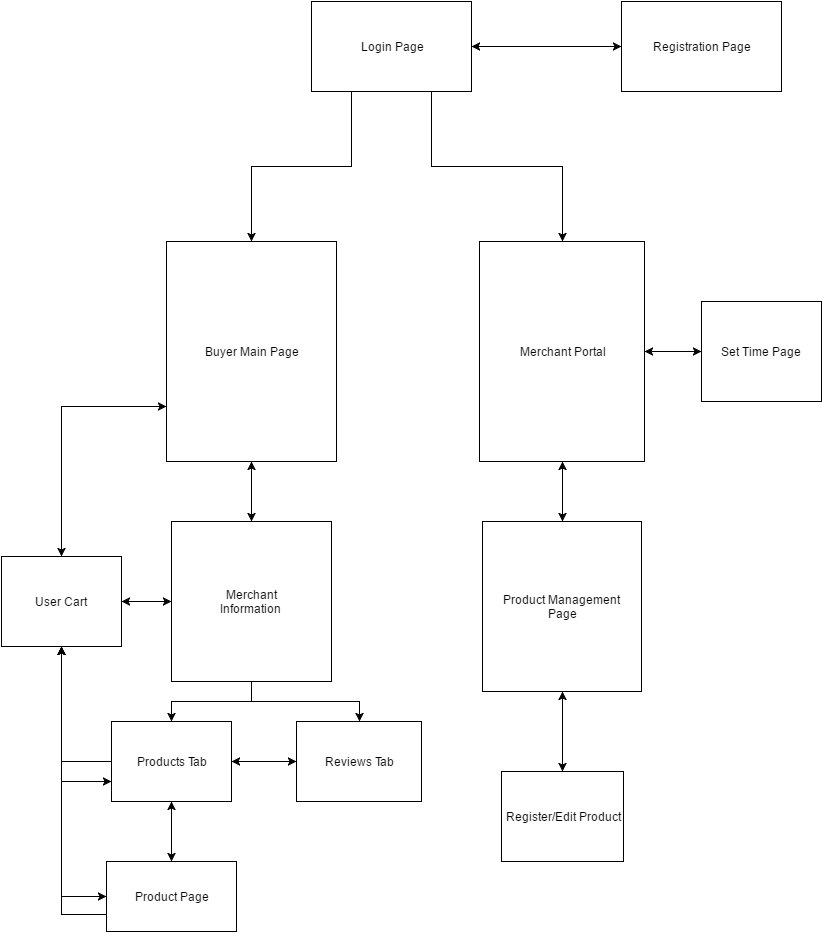
# **Mobile Application**

# UI

### Application Pages

* **Login Page**
  + The first screen that is seen upon running the application. The user provides their Rowan username and created password to login to the system by filling in the desired fields and tapping the login button. The user can also press the register button to register a new account on the registration page.
* **Registration Page**
  + The registration page is where a user or baker registers a new account onto the system. After filling in their Rowan username and password and tap the register button, they will be sent a confirmation email with a temporary token and be sent back to the login page.
    - The temporary token will be unique to each account. After registering an account, the app will ask the user for their token which the user will get from the email and provide to the app. The purpose of the temporary token is to validate their email address (e.g. spelled it correctly, etc).
* **Buyer Home Page**
  + The buyer homepage is the main page for a buyer account, allowing them to search for a specific baker either by name or by product, view that baker’s rating (not displayed in Mock UI), and click on the individual search results for more information.
* **Baker Information Page**
  + This page is accessed by users who tapped on a specific search result for a baker. The baker information page contains 2 major tabs relating to the business: the products and the reviews. The buyer can also choose to go back to the home page if they wish.
  + **Products Tab**
    - Contains an alphabetical list of all of the available products that the buyer can purchase. Each product in this list has a picture, a name, a shortened description, and a price. When a product is tapped, the user is brought to the product page (not modeled in Mock UI) with additional information and order options. If the product has any allergy warnings, it will be represented by a symbol.
  + **Reviews Tab**
    - Contains reviews and ratings from previous customers. Each review can also be rated positively or negatively to push it higher or further down the list (not displayed in Mock UI). Each review contains a rating out of 5 stars, and a brief review written by the customer.
* **Product Page (not modeled in Mock UI)**
  + When a user taps on a product in the products tab on the Baker Information Page, they are brought to this page which contains more information about the product including the picture, name, the full description, price, and allergy information. The user may also add a specific amount of the product to their cart.
* **User Cart (not modeled in Mock UI)**
  + The list of products that the user has added for purchase. They may also update each product quantity or remove them from the cart as well as view the total of their items so far. From here, the user may check out their cart by making payments with a linked paypal account.
* **Baker Portal**
  + The main page for a baker account. From here, the baker has access to their products page, setting their open and close time, viewing their business statistics and customer reviews. They can also edit their bakery name and current address.
* **Set Time Page**
  + Accessed and used to set the open and close hours of the baker’s business. Represented by a clock, this allows the bakers to make quick and accurate time changes when needed.
* **Product Management Page**
  + Displays a list of the current listed products for the baker. Also allows them to add new products to their list or tap a current product to edit that particular product’s settings.
* **Register/Edit Product Page**
  + Allows the baker to edit information about a current product or to register a new product. The baker can upload a picture and edit the name, description, allergies, and price of the current product, which will then be submitted as an updated or new product.

**Sequencing Diagram**



### **Model view control**

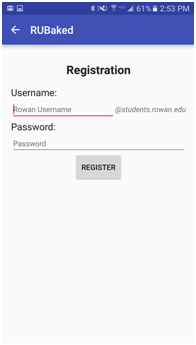
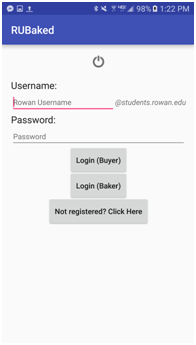
* **Overview:**
  + Each page of the application will be represented by its own structure or class (ex. activity) for displaying relevant information and handling events from the user, accomplished by switching to a different page of the application or by retrieving information via the API.
  + Specific pages need to request information from the API to display in a list to the user.
* **Login Page:**
  + Login Button:
  + Registration Button:
    - Switches to registration page.
* **Registration Page:**
  + Registration Button:
    - **POST buyers/ or POST bakers/** with passed registration field information.
* **User Home Page:**
  + Information retrieved from **GET bakers/list**
  + Displayed information will be limited based on search field
    - **GET bakers/search/<product>** will show only bakers with a particular product.
  + Tapped bakery will switch view to baker information page.
* **Product Pages**
  + Product list information retrieved from **GET bakers/<BakerID>/menu**
  + Basic information shown within product list.
  + When clicked on a specific product, shown more information only by the specific product.
* **Reviews Pages**
  + Review information obtained from **GET baker/reviews**

### **Bounds**

* **Passwords**
  + Passwords must be at least 6 and at most 30 characters containing at least 1 letter and 1 number.
* **Ordering Products**
  + Products can only be ordered from 1 baker at a time. A buyer cannot have a product in their cart from 2 different bakers
* **Emails**
  + Users may only register with a rowan email address.
    - The email address must end with ‘@students.rowan.edu’ or ‘@rowan.edu’
* **Reviews**
  + No minimum word count
  + 1500 character limit
  + Users may only rate products on a scale from 1 to 5

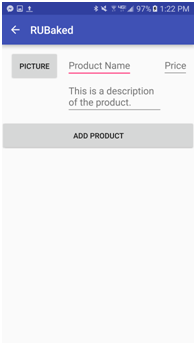
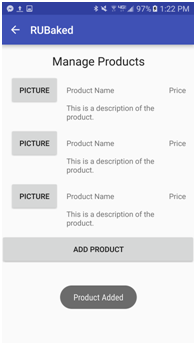
### **Mock UI Pictures**

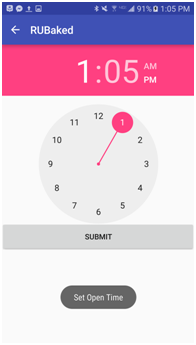
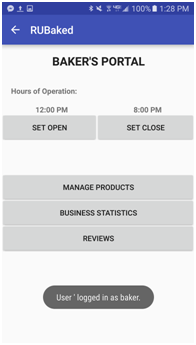
* + - 1. Merchant Buyer



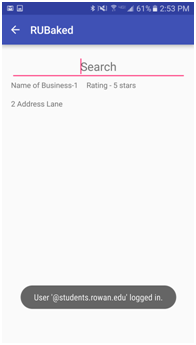
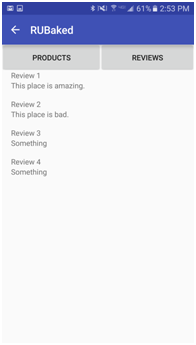
\*Note: For purposes of demonstration, two ‘Login’ buttons are displayed to ease in basic testing of app functionality. A universal ‘Login’ button will be used in the final product.

* + - 1. Merchant Only





* + - 1. Buyer Only



## Testing

**Login Screen**: When pressing the “Login” button, assert that the server is queried with the user’s login information.

If successful: Show default buyer or baker page.

If unsuccessful: Return appropriate error. Stay on the login page.

**Registration**: Check that email address ends with @students.rowan.edu or @rowan.edu. Check password length against bounds

**Baker’s Portal:** Check that any previously-set open/close time is still effective. Check that the open/close buttons change the open/close times locally as well as properly queries the API for a database-change to be reflected on end user’s applications. Check that the Manage Products, Business Statistics, and Review buttons take the user to the appropriate screen.

**Manage Products:** Check that the database is queried for all appropriate products that the business sells. Upon clicking a product, an “edit screen” pertaining to that product should appear.

**Add** **Product:** All text fields should accept text properly. The **“**picture” button should successfully accept a picture, and populate with a thumbnail upon being set. The “add product” button should send product information to the database.

**User Home Page:** The search bar should successfully accept text and automatically populate the screen with business or products that match the text entered. If no text is present, a sorted list of businesses should appear by default. Users should be able to click on a business to be brought to the respective business’ page.

**Business Page:** By default, the “products” tab should be selected. Upon clicking of the “reviews” tab, the bottom portion of the screen should display user reviews. The option to switch between products and reviews should always be present at the top of the screen. Upon clicking of a product or a review, users should be able to view more information of each.

**Cart/Order Page:** The page should properly have room to display a few products at a time, while having the ability to scroll through more should there be too many to fit on a single screen. The page should properly populate with products according to the database. The **Checkout** button should make all necessary API calls as well as take the user to a Paypal page to submit payment information.

# **Database**

## Table Specification

### User’s Table

User Information is stored in the Users table of the database.

The Users table will contain information that identifies specific users and their accounts.

The table contains the following fields:

* A unique identifier for each user
* The user’s first name
* The user’s last name
* Saved address (optional)
* Rowan Students email
* Phone number (of the phone the app is on)
* A password to access their account
* A paypal identifier for their paypal account
* A unique registration code for each user

The Users table has a relationship with the Orders, Rating and, StoreFeedBack tables.

The User’s table has no foreign keys and has the primary key of their UserID.

### Merchant Table

Merchants’ information, similar to users, are stored in the Merchants table.

The Merchants table will contain information identifying a specific baker.

The Merchants table contains the following information:

* A unique identifier for each merchant
* The merchant’s first name
* The merchant’s last name
* Email
* Phone number (of the phone the app is on)
* The merchant’s password
* The file path to the picture of the merchant’s License
* A paypal identifier for the merchant’s paypal account

The Merchants table has relationships with the Store table.

The Merchants table has the primary key of Merchant\_ID and no foreign keys.

### Store Table

The Store table contains the information about a specific store.

The Store table is needed to separate the Merchant account information from the information about their Store.

The Store table contains the following information:

* A unique identifier for the Store
* The unique ID of the Merchant who owns the Store
* The name of the Store
* The address of the Store
* The open time of the Store
* The close time of the Store
* The phone number of the Store
* Whether or not they deliver
* Whether or not the baker has manually marked themselves as offline

The information regarding the delivery options will be stored as an integer ranging from 0 to 2.

* A 0 in this field will indicate that the Store is pickup only
* A 1 will indicate that the Store is delivery only
* A 2 will indicate that the Store allows both pickup and delivery

The Store has a special field to indicate if they have manually marked themselves offline.

This field will be:

* True, if the Merchant has manually set themselves offline
* False otherwise

This field is going to be used to allow the Merchant to set their Store offline in case they are unable to operate during their normal hours.

Once the Merchant sets their Store to offline it will not become active again until the Merchant manually sets themselves back online.

The Store table has relationships with the Merchant and the Product tables.

The Store table has the primary key of StoreID and the foreign key of MerchantID.

### Orders Table

Information on orders are stored in the Orders table.

The Orders table will contain the following information:

* Aunique identifier for the order
* a User\_ID associated with the order
* The date the order was placed on
* The address the order will be delivered to (if the order is not being picked up)
* The status of the order

The Orders table will be used to keep track of the various orders placed by users and the details of said orders.

The address for the order will be required for orders that are being delivered and will not be used for orders that are being picked up.

The status of the order will contain an integer ranging from 0 to 3

* A 0 in this field will indicate that the order has been placed
* A 1 will indicate that the order has been accepted and payment has been processed
* A 2 will indicate that the order is being made
* A 3 will indicate that the order is out for delivery or available to be picked up

An order will be considered completed once it has a status of 3.

The Orders table is related to the User, Order\_Content, and Status tables.

The Orders table has the primary key of Order\_ID and a foreign key of User\_ID.

### Order\_Content

The Order\_Content table contains information on the content of a specific order.

Order\_Content contains the following information:

* A unique identifier for the specific order
* The unique identifier of a single Product in the order
* The quantity of the Product that was ordered

The Order\_Content table has a relationship with the Product and Order tables.

The Order\_Content table has the Primary key of Order\_ID and Product\_ID.

### Product Table

The Product table contains information about each product offered by the Stores in the database.

The Product table will contain the following information:

* A unique identifier for the product
* The unique identifier of the Store that makes the product
* The name of the product
* The price per unit of the product
* The description of the product
* The type of product
* The file path to the image of the product
* A quantity for the product (how many items come in one order)

When a Merchant adds an item to their Store they must chose a type for the product.

The list of types will be defined by us so that the baker does not have to input the type themselves.

This is done to limit the number of types of products.

The type of the product will be used to allow the users to filter based on the type of product they want.

The Product table has a relationship with the Store, Ratings, Allergens and Order\_Content tables.

The Product table has a primary key of Product\_ID and a foreign key of StoreID.

### Status Table

The Status table is used to keep track of the status of an order.

The Status table will be a static, predefined table that will be used to relate a status code for an order to a text definition of the status.

The Status table will contain the following fields:

* A unique identifier for the status
* A text definition of the status

The status table has the primary key of StatusID.

The status table has a relationship with the Order table.

### Allergens Table

The Allergens table contains information on allergy warnings for products.

The Allergens table contains the following information:

* The unique identifier of the product
* The name of a specific allergy warning for the product

The choices for allergy types will be predefined by us so that the information is uniform for all products.

The types of allergy warnings will be:

* Dairy
* Eggs
* Peanuts
* Tree Nuts
* Gluton
* Soy

The Allergens table has a relationship with the Product table.

The Allergens table has the primary key of Product\_ID and Allergy\_Type.

### Rating Table

The Rating table contains information on the ratings for a specific product.

The Ratings table will store not only the numeric value of the rating but also any comments the user wants to add to their rating.

The Ratings table contains the following information:

* The unique identifier of the product being rated
* The unique identifier of the user who placed the rating
* The text of the rating(if applicable)
* The date of the rating

The Ratings table is related to the Users and Product tables.

The Ratings table has the primary key of Product\_ID and User\_ID.

### StoreFeedback Table

The StoreFeedback table contains information on comments left by users on a specific Store.

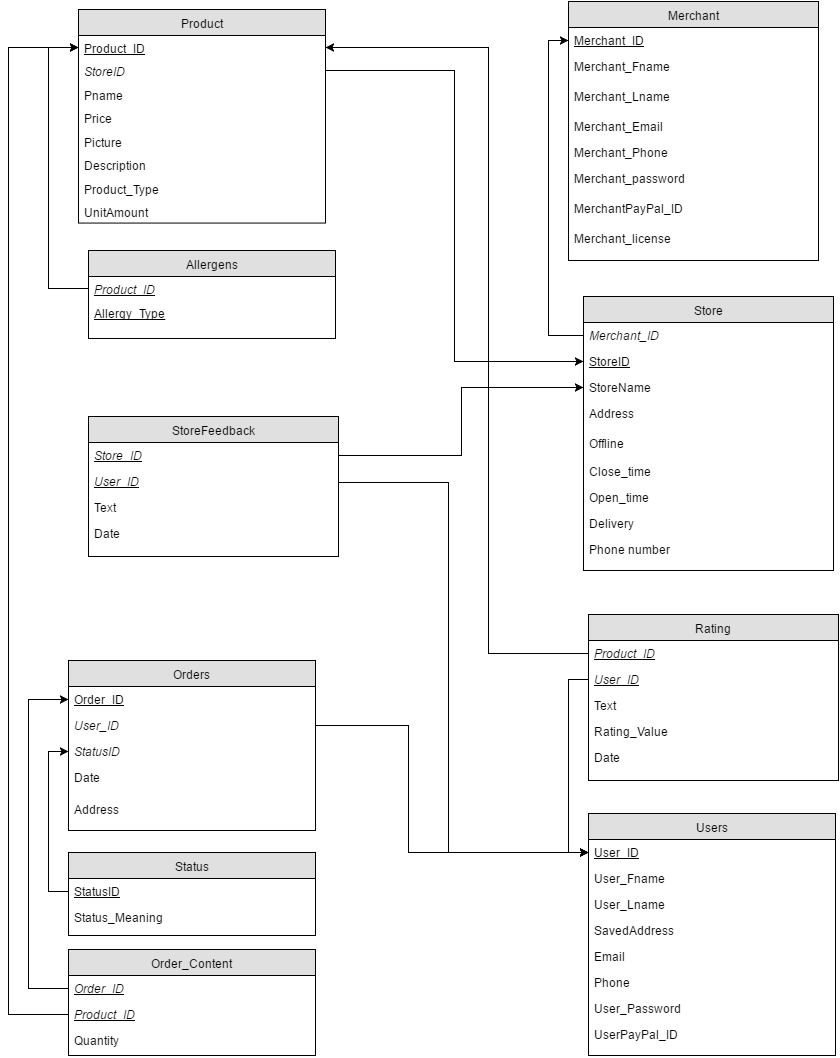
The StoreFeedback table contains the following information:

* The unique identifier of the Store that feedback is being left for
* The unique identifier of the user who left the feedback
* The feedback that was left
* The date of the feedback

The StoreFeedback table is related to the Store and Users tables.

The StoreFeedback table has the primary key of StoreID and User\_ID.

## ER Diagram



## Database Testing

### Data integrity

In order to test data integrity, the database will be subjected to several tests to input new data and retrieve that data after operations have been performed on it. Validity will be determined by ensuring the expected data matches the output.

|  |  |  |
| --- | --- | --- |
| **Type** | **Input** | **Expected Output** |
| Entry Creation | Create a new entity in each table. | An entity with a unique identifier containing all specified fields. |
| Positive Entry Modification | Modify existing fields to a valid value. | Field modified is retrieved with the new value. |
| Negative Entry Modification | Modify existing fields to an invalid value. | Field modified is retrieved with original value. |
| Entry Deletion | Deletion of an entry in any table. | All remnants of the entry are removed including entries using the deleted entry as a foreign key. |

### Base Functionality

To test functionality of the database, stored procedures such as retrieving data will be verified that they return the expected values.

|  |  |  |
| --- | --- | --- |
| **Type** | **Input** | **Expected Output** |
| Single Entry Retrieval | Retrieve a singular field from a specified table. | The correct value of the field specified.. |
| Aggregate Entry Retrieval | Retrieve an aggregate of the selected field(s) from a specified table(s). | The correct calculated aggregate of the field(s) specified by the procedure |

### Foreign / Primary Keys

Testing of foreign and primary keys ensures that relationships between tables are functioning properly. Testing involves verification of correct entry creation and modification with respect to keys.

|  |  |  |
| --- | --- | --- |
| **Type** | **Input** | **Expected Output** |
| Positive Related Entry Creation | Creation of a new entry utilizing existing table entries for foreign keys | Creation of a new entry with fields corresponding to the specified entries. |
| Negative Related Entry Creation | Creation of a new entry specifying invalid or nonexistent entries as foreign keys. | No new entries added to database. |
| Duplicate Primary Key Generation | Creation of an entry with primary key field(s) duplicate to another entry. | No new entries added to database. |

# **API**

## 4.1 - API Specifications

We have chosen a RESTful architecture for our API as it is both easy to develop for and prevalent in the industry. A principle benefit of having an API is code reuse. We are targeting mobile, however, having a REST API means making a Windows phone application or a desktop application only requires creating a GUI and an API client. We have chosen the Python language with Flask, an open-source robust Python framework. Flask allows us to bring in any dependencies we desire such as an ORM like *SQLAlchemy* or a task queue like *Celery*. For development, the API and PostgreSQL DB will be hosted on an Ubuntu VPS via Digital Ocean. Our API will only support JSON and not XML as JSON is far more modern and lightweight.

## 4.2 Request Types

4.2.1 GET:

gets an immutable entity or a list of entities. A success returns a HTTP 200 OK response code.

4.2.2 POST:

creates some entity. A success returns HTTP 201 CREATED as well as the internal id to be used for future requests.

4.2.3 PUT:

updates an entity. Only fields in the JSON will be updated. Missing fields will retain the values already stored. An id is required to update. A success returns HTTP 202 ACCEPTED.

4.2.4 DELETE:

removes an entity. An id is provided by the client. A success returns HTTP 204 NO CONTENT.

## 4.3 Response Codes

4.3.1 An unsuccessful request will return an error code:

I. HTTP 400 BAD REQUEST:

Occurs if an invalid endpoint was hit or invalid JSON was provided

II. HTTP 401 UNAUTHORIZED:

Occurs if an API token is not provided.

III. HTTP 403 FORBIDDEN:

if an API token was invalid, or if the permissions forbid access to this endpoint

IV. HTTP 404 NOT FOUND:

if an id used in a request does not exist or is otherwise invalid.

V. HTTP 503 SERVICE UNAVAILABLE:

the server is unavailable, should wait 1 minute or something

4.3.2 Authentication

We will be using OAuth for authentication. OAuth 2.0 is an open standard used for authorization, which can allow for third party applications to access our platform, without exposing any passwords. This opens up the possibility for a third-party to create an application for a device we do not support. Authentication is done via an access token.

## 4.4 Endpoints

##### 4.4.1 **POST buyers/** and **POST** merchants**/**

**Description**: Creates a baker/buyer account in our system.

**Parameters**:

{

"Username": "String",

"Password": "String" # hashed with *bcrypt* or other crypto-hash,

"Address": "String"

"Email: "String",

"Phone":"String"

}

**Example Response**:

{

"id": 123, # the client SAVES this for future requests

"Username": "String",

"Password": "String" # hashed,

"Address": "String",

"Email": "String",

"Phone": "String"

}

##### 4.4.2 **PUT buyers/<BuyerID>**

**Description**: Updates a buyer’s information. Only one parameter is required. Returns everything we have on this user.

**Parameters**:

{

"Password: "String", # hashed

"Email": "String",

"Address": "String",

"Phone": "String"

}

**Example Response:**

{

"Username": "String",

"Password": "String", # hashed

"Email": "String",

"Address": "String",

"Phone": "String"

}

##### 4.4.3 **PUT** merchants**/<**MerchantID**>**

**Description**: Updates a merchants information. Only one parameter is

required. Returns everything we have on this user.

**Parameters**:

{

"Password: "String", # hashed

"Email": "String",

"Address": "String",

"Phone": "String"

"Address": "String"

}

**Example Response**:

{

"Username": "String",

"Password": "String", # hashed

"Email": "String",

"Address": "String",

"Phone": "String"

"Address": "String"

}

##### 4.4.4 **GET** merchants**/list**

**Description**: Retrieves a menu of all merchants.

**Example Response**:

{

"merchants": [

{

"merchant\_address": "123 Shortest Path Road",

"merchant\_email": "john\_doe@gmail.com",

"merchant\_id": 123,

"merchant\_name": "John Doe",

"merchant\_phone": "123-456-7890",

},

{

"merchant\_address": "456 LaTeX Drive",

"merchant\_email": "don\_knuth@gmail.com",

"merchant\_id": 456,

"merchant\_name": "Don Knuth",

"merchant\_phone": "123-456-7897",

}],

}

##### 4.4.5 GET stores/list

Description: Retrieves a menu of all stores. Optionally include a parameter

Include\_closed\_stores.

Example Response:

"stores": [

{

"avg\_rating": "4.2",

"store\_address": "123 Shortest Path Road",

"store\_email": "get\_dental\_insurance@gmail.com",

"store\_id": 123,

"store\_name": "Dijkstra's Sweet Treats",

"store\_phone": "123-456-7890",

"does\_shipping": true

},

{

"avg\_rating": "3.5",

"store\_address": "456 LaTeX Drive",

"store\_email": "knuth\_shuffle@gmail.com",

"store\_id": 456,

"store\_name": "Don Knuth's Cavities",

"store\_phone": "123-456-7897",

"does\_shipping": false

}],

"include\_closed\_bakers": false

}

##### 4.4.6 **GET** stores**/<**StoreID**>/menu**

**Description**: Get a list of all items from this particular store.

**Parameters**: StoreID(supplied in the URI)

**Example Response**:

{

"store\_id": 456,

"menu": [

{

"productavg\_rating": "5.0",

"product\_description": "Filled with a *secret* ingredient so that you can read The Art of

Computer Programming without stopping!",

"product\_id": "938",

"product\_name": "Knuth's Special Brownies",

"product\_pic": "http://s3.aws.com/XXX.jpg",

"product\_price": "$2.00",

"product\_amount\_unit": "5",

"product\_type": "Brownies"

},

{

"product\_avg\_rating": "4.9",

"product\_description": "Delicious cookies ALMOST worthy of the Ada name",

"product\_id": "203",

"product\_name": "Ada Lovelace's Cookies",

"product\_pic": "http://s3.aws.com/XXX.jpg",

"product\_price": "$1.20"

"product\_amount\_unit": "10",

"product\_type": "Cookie"

}]

}

##### 4.4.7 **POST buyers/rate/<**StoreID>**/<**ProductID**>**

**Description**: Allow a user to rate a specific product from a store.

**Parameters**:

{

"store\_id": 234,

"product" : 2,

"rating": "5.0",

"comment": "This tastes great"

}

**Example Response**:

{

"store\_id": 234,

"product\_id": "2",

"product\_avg\_rating": "4.1",

"product\_rating": "5.0",

"Comment": "This tastes great",

"comment\_id": 456,

"product\_description": "A delicious treat! Made with love and 56-bit keys.",

"product\_name": "Crypto-Coffee Crunch",

"product\_pic": "http://s3.aws.com/XXX.jpg",

"product\_price": "$4.99"

}

##### 4.4.8 **GET** stores**/search/<product>**

**Description**: Users can enter a String to be matched with product descriptions in the database,

the bakers who sell these products are returned.

**Parameters**: product (string such as “chocolate chip cookies”)

**Example Response**:

{

"bakers": [

{

"product\_id": "512",

},

{

"product\_id": "22",

},

{

"product\_id": "123",

}]

}

##### 4.4.9 **POST buyer/purchase**

**Description**: Order for payment and delivery.

**Parameters**: List of ProductID/Quantity

**Parameters**:

{

"Purchases": [

{

"Product\_id": "512",

"Quantity": "12",

},

{

"Product\_id": "99",

"Quantity": "2",

},

{

"Product\_id": "37",

"Quantity": "1",

}

]

}

**Example Response**:

{

"Purchase\_Id": "123",

"Purchases": [

{

"Product\_id": "512",

"Quantity": "12",

},

{

"Product\_id": "99",

"Quantity": "2",

},

{

"Product\_id": "37",

"Quantity": "1",

}

]

}

##### 4.4.10 **GET** store**/**comments**/<**StoreID>

**Description**: Users can view all comments for a certain Store.

**Parameters**: StoreID

**Example Response**:

{

"Comments" : [

{

"Comment\_id": "512",

"Comment\_Date": "3/14/16",

"Comment\_Description" : "This bakery is very good, would purchase from again - quick delivery!"

},

{

"Comment\_id": "111",

"Comment\_Date": "4/15/15",

"Comment\_Score" : "0.0",

"Comment\_Description" : "Delivery took over 2 hours! Cookies were stale when they arrived, an

awful experience."

}]

}

4.4.11 GET store/reviews/<StoreID>/<ProductID>

**Description**: Users can view all reviews for products of a certain Store or the reviews of a specific product from the store.

**Parameters**: StoreID

**Optional Parameter:** ProductID

**Example Response**:

{

"Reviews" : [

{

"Review\_id": "512",

"Product\_id": "123",

"Review\_Date": "3/14/16",

"Review\_Score" : "4.3",

"Review\_Description" : "This cookie is very good, would purchase again - very tasty!"

},

{

"Review\_id": "111",

"Product\_id": "123",

"Review\_Date": "4/15/15",

"Review\_Score" : "0.0",

"Review\_Description" : "The cookies were stale when they arrived, an

awful experience."

}]

}

## 4.5 Testing

We will have a basic unit test suite that will test each API endpoint, by supplying dummy data and having an expected response. We will also have a basic integration test suite, which will test different API endpoints together (e.g. updating a baker’s information with PUT and then doing a GET to confirm the database did the writing). While a methodology called test-driven development (TDD) exists stating we writing the tests first, I think it might hinder our creativity. Should this project be deployed to the Rowan community or beyond, it is an absolute must to have a more comprehensive test suite, which should also include end-to-end (e2e) tests.

We will also check for bounds, such as GET on an entity that doesn’t exist, DELETE on an entity that doesn’t exist, DELETE twice, etc.

## 5. Scalability

Some thought has been placed on scalability. By scalability, we mean how well our platform will respond to increased traffic. For example, the core functionality of Facebook is trivial to implement. Engineering problems become much harder at that scale. Making a platform that scales to almost 2 billion people is a very impressive engineering feat.

For our purposes, the VPS on Digital Ocean should be able to handle the Rowan community. We can scale vertically by paying for a more powerful instance on Digital Ocean. Another option is to scale horizontally by adding more machines and employing something like *Celery*, a robust open-source task queue we used at Optimizely, to distribute work among different machines. We could take advantage of PostgreSQL *sharding* capability, too. Containerization technology like Docker could make deployment easier, or an open-source tool called Dokku, which is a self-hosted Heroku clone.

The obvious bottleneck in our platform is going to be the database. We can speed up database reads by using a cache such as Redis. We can have the *illusion* of faster database writes by putting the write operation in a task queue like Celery, give the user feedback immediately, and let a worker complete the task in the background.

Another approach to handle increased traffic is the amazing AWS platform. We can use the AWS Gateway and AWS Lambda, AWS S3 (for file/photo storage) and AWS RDS (PostgreSQL), and let Amazon do the hard work (though this may cost more than doing it ourselves). AWS RDS also allows us to easily setup a read-only replica slave, that an allow us to perform analytics without interfering with the production DB. Other alternatives include Heroku (very expensive) or the Google Compute Platform.

The bottom line: By using Python & Flask with Postgres, we can use Digital Ocean, Linode, AWS, Heroku, or several other competitors in this market.