Computer Science 316 Final Project

Mariusz Derezinski-Choo, Robert Chen, Karoline Xiong, Justin Du, Nicolas Aldana

November 16, 2020

1 Introduction

The world's largest online marketplace had humble beginnings. Amazon launched in 1995 as a website that exclusively sold books but has since grown into the multi-billion dollar company we know it as today. As e-commerce continues to grow and become more prevalent worldwide, we recognize the importance of providing an online marketplace to bring buyers and sellers, individuals and businesses together. With our Mini-Amazon, we hope to provide the same fundamental functionalities integral to these crucial interactions.

Prior to the implementation stage, we spent a significant amount of time and effort brainstorming collaboratively. This entailed deciding the appropriate database setup and schemas and creating an ER diagram that would appropriately and adequately capture all the functionalities and interactions of the various components of our website. We explore our final design choice more closely in the following section as well the the modifications leading up to it.

Next, we focused on encapsulating these abstract relations and functionalities into the workings of our app design. We envisioned that users would be able to create accounts, add funds, search for a wide variety of items, add items to their carts, and purchase these items. On the other hand, storefronts would also have the ability to create their own accounts which they can use to put up items for sale and edit their listed items. We will highlight these features in-depth in the implementation section where we will also justify our choice of technology, namely a React App with a Flask backend connected to a SQLite database, as well as describe our techniques for procuring the data we used to populate our database.

Lastly, we will discuss the challenges that we faced in the design and implementation process as well as ideas that we have for future work.

2 Project Deliverables

The project is hosted at http://vcm-16363.vm.duke.edu:3000. The git repository for the project can be found at https://gitlab.oit.duke.edu/na158/mini-amazon

3 Design

Buyers and sellers: In our initial design, we planned a user entity set (User), where all users could act as buyers, with a boolean attribute to distinguish users with additional selling privileges. We found this could lead to difficulties regarding self-reviewing and self-purchasing, so we abandoned this idea. Instead, we separated buyers (Buyer) and sellers (Storefront) into entirely different entity sets. This model is closer to Amazon's system, where storefronts can be thought of as different accounts entirely. Practically speaking, this means that a digital storefront account can be shared by multiple managers, like how stores are operated in real life. Buying functionality would be irrelevant as well.

Both types of users are separated into unique tables, but share similar properties: for both, an email, password, security question/answer, and balance are used. However, buyers have first and last name fields, whereas storefronts only have a single name field. Storefronts have, additionally, a description field, where they can market themselves to buyers.

Items and listings: We've separated the abstraction of the item from the actual listing of the item. This is similar to how Amazon does items, where, say, an ISBN number would have an item page displaying its picture and description information contained in its Item tuple, while also displaying multiple sellers listing that item with their individual competitive pricing in the marketplace (a query of Listing w.r.t. the item's ID).

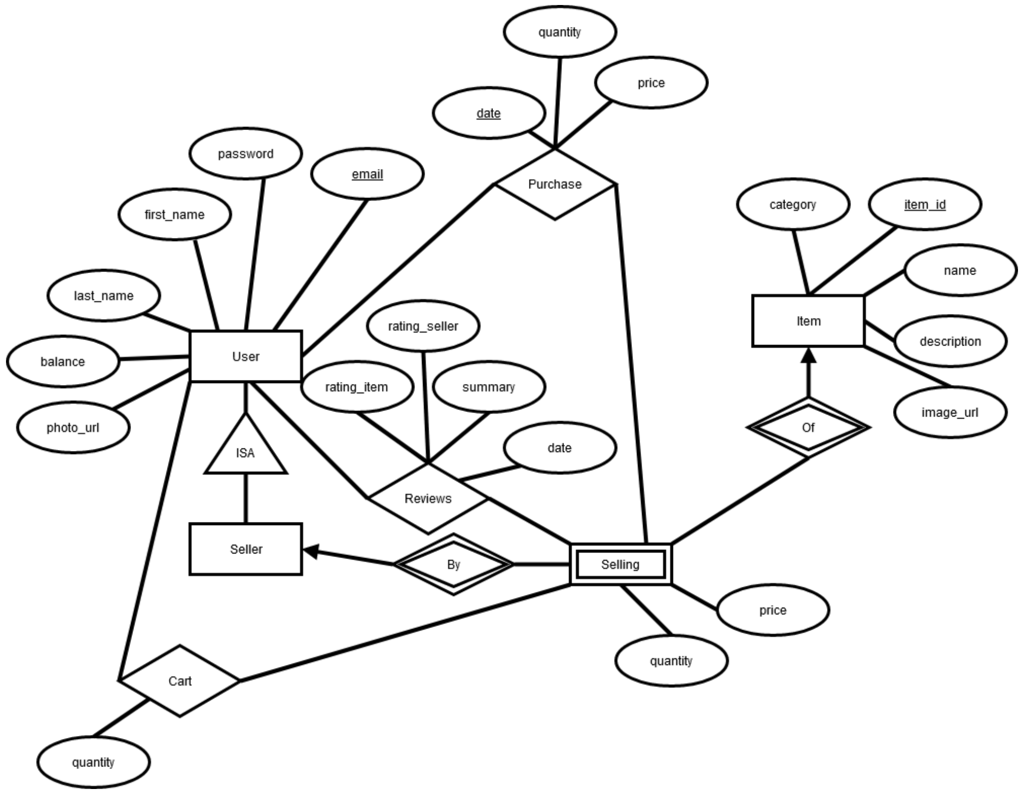
Each item, then, takes an item ID, a descriptive name, a product description, a category, and a photo URL. A listing references a particular item ID, and contains the storefront's email, their quantity, and their pricing.

Carts and purchases: When a buyer account is logged in, it wants to be able to add listings into a temporary record for eventual purchase. For this we have Cart which naturally borrows properties from Listing. A purchase is an instance of a cleared cart tuple, and thus naturally takes properties from Cart.

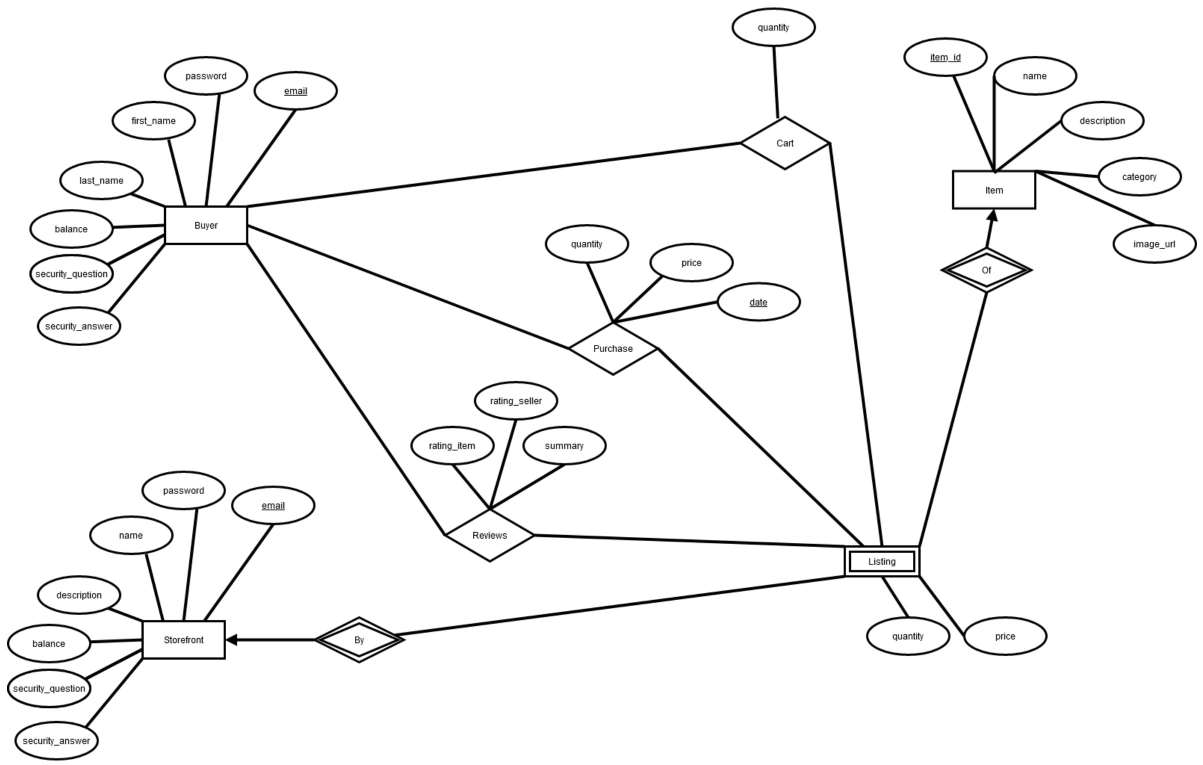
A cart thus tracks tuples of the buyer's email plus a valid Listing tuple's item ID, storefront email, and quantity (at most, or can be less). A purchase tracks the same tuples in cart, but that have also been processed, and stores additional data in price and datetime attributes.

Reviews: A review should track the buyer's identity, the storefront's identity, the item ID, an item rating, a storefront rating, and review text. We make a review's unique identifier a combination of buyer email, storefront email, and item ID. A user can easily update reviews for an item purchased from a storefront after initial reviews are published.

Database schema:
Buyer(buyerEmail, password, firstName, lastName, balance, securityQuestion, securityAnswer)
Storefront(storefrontEmail, password, name, storefrontDescription, balance, securityQuestion, securityAnswer)
Item(itemID, name, itemDescription, category, photoURL)
Listing(itemID [FK], storefrontEmail [FK], quantity, price)
Cart(buyerEmail [FK], itemID [FK], storefrontEmail [FK], quantity)
Purchase(buyerEmail [FK], itemID [FK], storefrontEmail [FK], quantity, price, purchaseDatetime)
Reviews(itemID [FK], storefrontEmail [FK], buyerEmail [FK], ratingItem, ratingSeller, review)



Our old ER diagram did not consider the complexity of both buyers and sellers being under the same User schema, distinguished only by a status boolean. After discussion with Kate and Rebecca, we made a new ER diagram for reference when implementing in code.

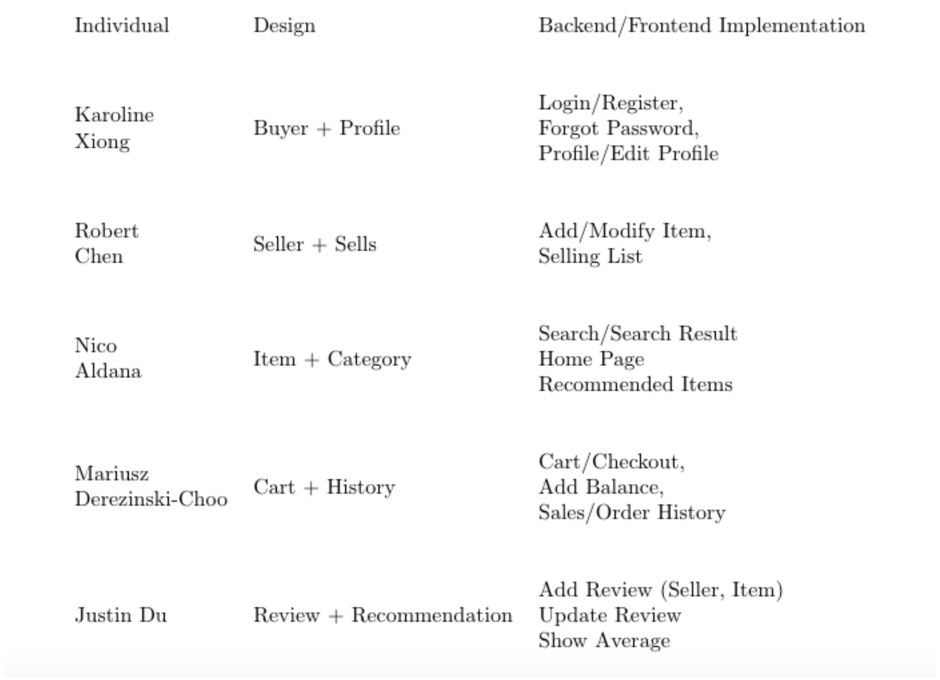


Our new ER diagram is easier to reason around. Here, we identify Cart, Purchase, and Reviews as simply relations between a Buyer and Listing, borrowing attributes from Item and Storefront to form the database schema.

4 Implementation

4.1 Division of Work

The tasks were divided as follows:



Individual	Design	Backend/Frontend Implementation
Karoline Xiong	Buyer + Profile	Login/Register, Forgot Password, Profile/Edit Profile
Robert Chen	Seller + Sells	Add/Modify Item, Selling List
Nico Aldana	Item + Category	Search/Search Result Home Page Recommended Items
Mariusz Derezinski-Choo	Cart + History	Cart/Checkout, Add Balance, Sales/Order History
Justin Du	Review + Recommendation	Add Review (Seller, Item) Update Review Show Average

4.2 Frontend

We utilized React.js in our frontend, a JavaScript library for building user interfaces. Its component-based architecture is ideal for developing responsive and scalable frontend web applications. We took advantage of React components' reusability to maximize efficiency and minimize redundancy. We mostly organized each component in its own file and created sub directories as we saw fit.

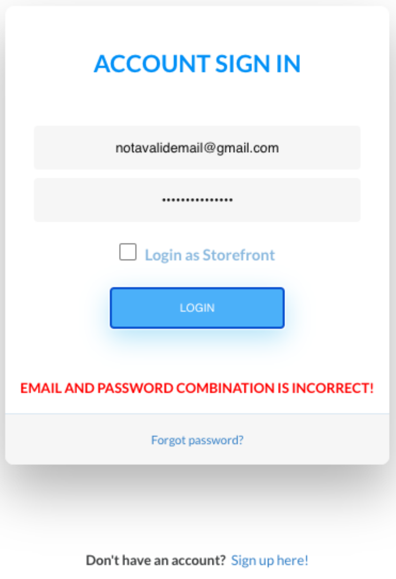
To allow our web application to be dynamic and interactive, we used state, a JavaScript object that allows components to keep track of changing data so that these components can behave and respond accordingly. When a user signs in or registers through the login portal for instance, the app retrieves data from a specified API endpoint and stores the result in state. (Note that the user must specify their role as either a buyer or seller upon signing in as per our ER design- see Section 2). We then use session storage to persist state and be able to store and retrieve data. We chose session storage over local storage since we would only like to cache data in the current browser session, such that closing the browser empties the cache but refreshing would keep the data intact, a behavior especially useful for our application where we would like user session to be saved until browser is closed (or user clicks to sign out).

Our React App is routed so that users are first brought to the login portal and have limited accessibility to the web app; they can only sign in, reset password, or create an account. Through using a protected Route component that handles private routes, we ensure that users must be authenticated in order to access the other pages (i.e. home, cart, reviews, etc.). Any attempt to go to these pages will only redirect them to the login portal.

Errors upon logging in or registering will be displayed on the user interface, whether due to a failed POST request (likely email already used in which case would violate database schema unique key constraint) or input form invalidated in the frontend (password and confirm password inputs don't match for example). Other constraints to user actions include letting a user change their password only after both answering their security question correctly and then confirming their new password. Feedback is provided in the form of error messages or status notifications that should allow for an easy to use and easy to follow user experience.

Once logged in, a user can navigate through all the webpages via the navigation bar. These include a seller page, a cart page, reviews page, and purchase history page. The seller page is where a seller can add new items to be listed to sell as well as items they have currently listed for sale. The cart page consists of items a user has in his or her shopping cart which they can add to or remove items from. The Reviews page consists of past reviews made by the user, as well as access to a form to add new reviews. The purchase history page allows a user to view his or her past purchases. Additionally, the navigation bar also houses links to users' profile pages and an add-to-balance page. Lastly, it contains a search bar that allows users to look up items they're interested in by name. The view profile page provides account details such as user's email, balance, first name and last name (or name and description if user is logged in as a storefront/seller). Upon clicking the edit profile button, users also have the ability to change fields such as name, first name, last name, description, password, etc. which will be updated in the database and the sessionStorage upon clicking submit. Users may also change and add to balance on the corresponding page.

For a cohesive and visually appealing layout, we chose a minimalist design. Sample screenshots are provided below of sign in and registration portals, respectively, with sample inputs that would produce appropriate error messages:


ACCOUNT SIGN IN

notavalidemail@gmail.com

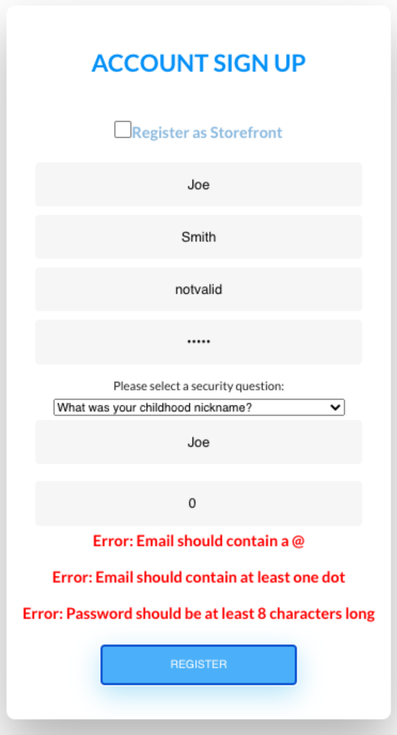
•••••••••••••••

Login as Storefront

EMAIL AND PASSWORD COMBINATION IS INCORRECT!

Forgot password?

Don't have an account? Sign up here!


ACCOUNT SIGN UP

Register as Storefront

Joe

Smith

notvalid

•••••

Please select a security question:

What was your childhood nickname?

Joe

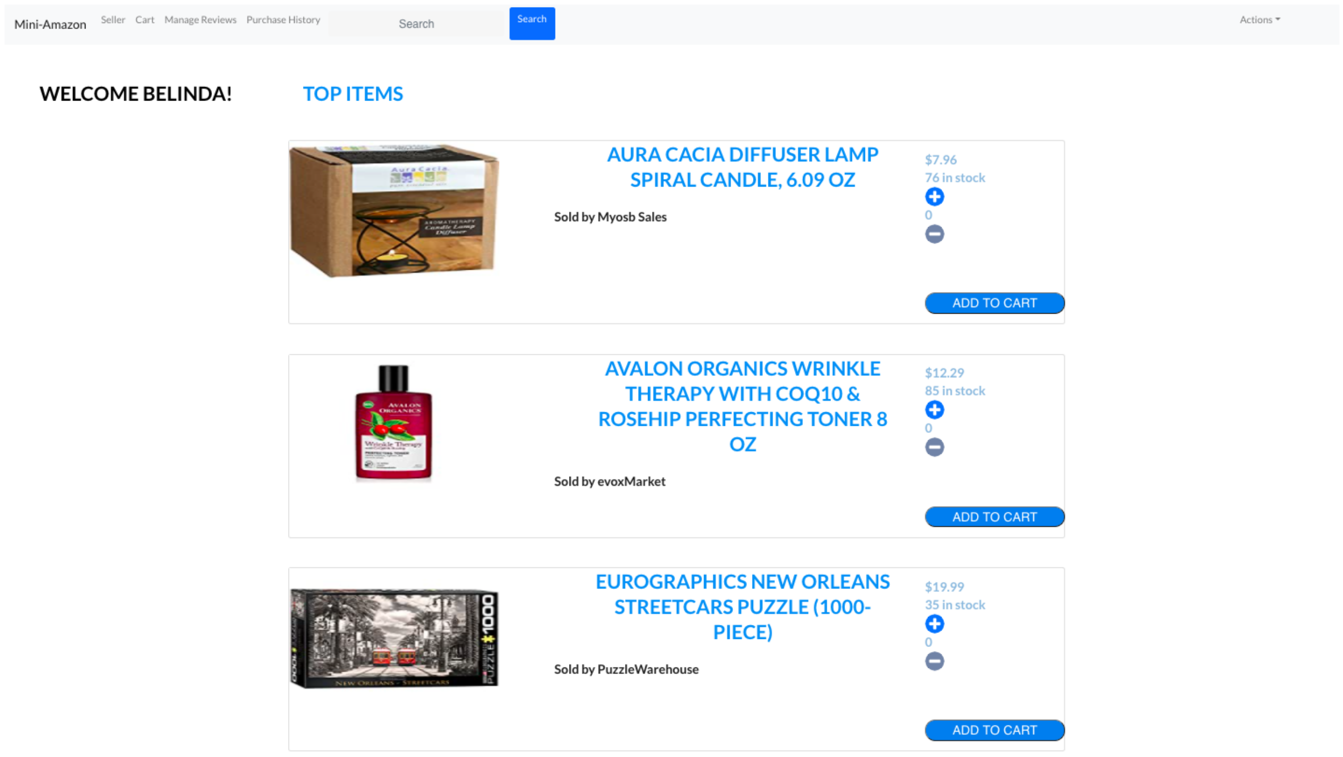
0

Error: Email should contain a @

Error: Email should contain at least one dot

Error: Password should be at least 8 characters long

A view of the home screen:



WELCOME BELINDA!

TOP ITEMS

AURA CACIA DIFFUSER LAMP SPIRAL CANDLE, 6.09 OZ

Sold by Myosb Sales

$7.96
76 in stock
0

AVALON ORGANICS WRINKLE THERAPY WITH COQ10 & ROSEHIP PERFECTING TONER 8 OZ

Sold by evoxMarket

$12.29
85 in stock
0

EUROGRAPHICS NEW ORLEANS STREETCARS PUZZLE (1000-PIECE)

Sold by PuzzleWarehouse

$19.99
35 in stock
0

4.3 Backend

We utilized the Flask micro-framework for Python to implement our backend, along with a SQLite database and Flask-SQLAlchemy Object-Relational Mapping (ORM) framework.

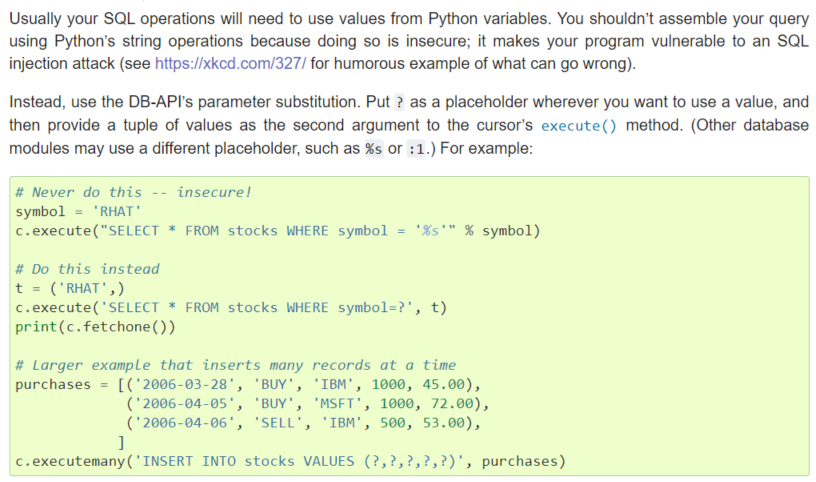
Since several team members did not have extensive prior experience with web development, we decided to go with Flask, which is relatively quick to pick up. Flask is a lightweight micro-framework that allowed us to create endpoints for our content. We wrote HTTP requests for these endpoints to implement the necessary logic for our website. Flask also plays very well with the SQLAlchemy ORM, which allows us to query our database more conveniently, as well as provide some added robustness and security.

SQLite is a file-based Relational Database Management System (RDBMS) with ACID transactions, even in the case of system crashes and outages. Like Flask, SQLite is also very lightweight, which is why we decided to use it. While it was easy to set up and interacted well with SQLAlchemy, we ran into some inconveniences working with SQLite. Because SQLite is file-based, it was a bit awkward to work with the database collaboratively, because any changes to the database file would conflict in Git across our group members. SQLite served our purposes well for this project, but it may be more beneficial to use a cloud database service like Amazon RDS for more serious projects, and if we continued development on this project it would be worthwhile to migrate.

Our database models (tables) can be found in models.py, which shows the schema of our database. Our Flask endpoints are located in views.py, and database.db is our SQLite database file.

4.4 SQL Injection

One of the reasons we chose to use an ORM was for the security benefits in preventing SQL injection. While nothing is 100% safe, using Flask-SQLAlchemy was a good step toward protecting our database against malicious intentions. If we did use raw SQL, we made sure to use parameterized queries. This is achieved by using a question mark (?) as a placeholder for values of the query, then passing in the values as the other argument. Here is an example from the sqlite3 python documentation:



Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see https://xkcd.com/327/ for humorous example of what can go wrong).

Instead, use the DB-API's parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

Note that while we decided to use Flask-SQLAlchemy's way of executing raw SQL with db.engine instead of sqlite3 cursor, the parameterized query format is the same for both and achieves the same function.

4.5 Data Procurement

In order to populate our database with a production sized dataset, we found a dataset of over 30,000 amazon items from Kaggle.[1] The dataset contained the needed information of product names, image urls, product descriptions, product category, and up to 4 different sellers with their associated price for that product. To make it easy to have a deterministic baseline database that we could reset to at any time, our script sets its random seed to 316 and deletes the old database to start with. It then loads the kaggle dataset from a csv file using the pandas python package, and removes all rows that do not have a valid price row, since we had no way of estimating the price without this field, and thus excluded those tuples. The data procurement script will then insert a tuple into the item relation for each row of the csv, and add a listing for every seller and pricepoint for that item, adding seller tuples to the seller relation as appropriate as new sellers are discovered.

Procuring data was a more challenging task because due to privacy reasons, it is difficult to acquire a dataset

[1] available at https://www.kaggle.com/promptcloud/amazon-product-details

that contains first names, last names, and emails. For this reason, we chose to generate mock dataset for our user, review, purchase, and cart relations. To generate the data, we made use of json-generator.com, a free service that allows for the generation of randomly generated json based on a user-defined schema, including support for a variety of randomly generated values such as first names, last names, emails, datetimes, lorem text, and numbers. A copy of the json generation schema is in the Appendix. Our schema generates a list of 500 users with randomized first names, last names, emails, passwords, and balance. Within each user in the list, the schema defines rules for randomly generating 50 reviews for each person, 3-5 purchases, and between 5-10 items in their cart. Once the json was generated according to this schema, we were able to write a python script to load the json file and insert tuples into the database accordingly.
The size of the data we procured and generated can be verified by running the check_table.sh bash shell script from the root of the project directory. The buyer relation has 501 tuples, the storefront has 3921 tuples, the item relation has 8366 tuples, the cart relation has 3722 tuples, the purchase relation has 7996 tuples, the listing relation has 9224 tuples, and the reviews relation has 25172 tuples

5 Challenges

One challenge was implementing login such that only authenticated users would have access to the various web pages as well as being able to pass signed in user's data through all the various components and retaining state information throughout the duration of the user's session. Through trial and error and research, we realized sessionStorage was able to accomplish this and through collaborative efforts, we were able to pass user's data through all our separate components such that our website functions cohesively. Additionally we had to ensure that a user who is not signed in did not have access to the webpages. Hiding the navigation bar from the login page was not enough, as a user could simply type in the desired path into the url to access. Therefore, we implemented a class protected routes such that if variable isAuthenticated is false, attempts to access all pages (other than login, register, and forgot password page which are accessible to all as they should be) would be redirected to the login portal. Only upon successful login will isAuthenticated be true, in which case one would be able to access the desired pages.

Another challenge we had was working with pictures for items and listings. Because pictures were an essential part, we had to ensure that the dataset we found in the data procurement section had photo urls for all of our cleaned entries. Then came the problem of getting and storing photos for new listings. We discussed several options, such as hosting the photos on the virtual machine along with our application. However, we ended up deploying our application with Netlify, so we looked toward other options. We ended up deciding to use Imgur's API to upload photos, get back the photo url, and store it in our database for future retrieval. The process of writing this code was a challenge in itself, but we were able to successfully implement this.

Our database also presented us with a bit of a challenge, or at least an annoying inconvenience. As previously stated, we used SQLite for our database, which is file-based and gave us some trouble when working on it collaboratively. Once we had end-to-end functionality in our website, we constantly made changes to the database to test our components, so the database file kept changing. At first, we didn't discuss how we should handle this problem, so we kept running into issues with version control by getting merge conflicts in the database file. This problem could have been mitigated by using a cloud hosted database, but that also comes with a learning curve and would have been a challenge initially to set up.

We faced some challenges with hosting our application. Because Netlify requires HTTPS, it was a struggle for us to figure out how to ensure Flask had a valid SSL certificate and complied with HTTPS.

6 Future Work

Given more time, we may have implemented additional features and enhanced certain aspects of the website to improve user and seller experience. For example, we would want for the storefronts to be able to show when an item is on sale as opposed to simply changing the price, which would also let buyers know when they are taking advantage of a deal. Additionally, the user is currently unable to filter their search results by category and price ranges, so we would add those functionalities to enable advanced searches for more specific results. While we currently recommend the top rated items on the site on the user's main page, it would be useful to have a more personalized home page that recommends items to the user based on the specific user's purchase history.

To see how well our website scales, we would also be interested in procuring more data to be able to judge how performance is affected with upwards of a million tuples. Some components of the website are already suffering from scalability issues, so addressing those would be an immediate next step if we had more time.

Additionally, authentication should ideally be accomplished by sending a hash derived from the password as opposed to the password itself or some alternative form of password encryption and decryption to address security concerns.

7 Appendix

7.1 Buyer Data Generation Schema

```
[
    '{{repeat(500, 501)}}',
    {
      balance: '{{floating(1000, 4000,2)}}',
      firstName: '{{firstName()}}',
      lastName: '{{surname()}}',
      email: '{{email()}}',
      password: '{{guid()}}',
      reviews: [
        '{{repeat(50,51)}}',
        {
          rating_item: '{{integer(1,5)}}',
          rating_storefront: '{{integer(1,5)}}',
          review: '{{lorem(3)}}',
          storefrontIndex: '{{integer(1,3920)}}',
          timeSubmitted: '{{date(new Date(2010, 0, 1), new Date(),"YYYYMMdd hh:mm:ss A")}}'
        }
      ],
      purchase: [
        '{{repeat(3,5)}}',
        {
          time: '{{date(new Date(2010, 0, 1), new Date(),"YYYYMMdd hh:mm:ss A")}}',
          items: [
              '{{repeat(3,5)}}',
              {
                  storefrontIndex: '{{integer(1,3920)}}',
                  quantity: '{{integer(1,10)}}'
              }
          ]

        }
        ],
      cart: [
        '{{repeat(5,10)}}',
        {
          storefrontIndex: '{{integer(1,3920)}}',
          quantity: '{{integer(1,10)}}'
        }
        ]
    }
  ]
```