1. **Where can I find a copy of your video?**

Ulearn

1. **Give a description of the originally proposed functionality for your project, as well as details on what changed (things you did not complete or things you added). What caused the change? This is a good place to include user stories, sequence diagrams, timing diagrams, user flow diagrams, flowcharts of algorithms, etc. All documentation should be embedded inside the Word doc as an image.**

*This whole section below is for question b), up until question c) :]*

Antique Furniture Web App

By: Phakday Russell

CSIS3126

An ecommerce web app where owners of antique furniture can list their items on the website for sale to including necessary information (dimensions, photos, etc). Once they register their item on the website, others will be able to see all the available items for sale.

* Users have to have an account (signup, login) to post the furniture they want to sell.
* The items can be categorized by furniture type (beds, tables, desks, chairs, chests, nightstands, cabinets, etc.)
* A search bar to search for items.
* A filter for furniture type, cost, year (period), etc.,
* Authenticated users can list their items for sale with pictures of their items, a detailed description of the item, cost, condition, and other sorts of information like the estimated year it was made, contact info for additional inquiries, material of the item (type of wood)
* Customers can add the items to their shopping cart, providing their shipping address, payment info, additional delivery instructions, and other necessary details so it can be delivered to the customer.

Summary of features

* Authentication (username, password, email)
* Search bar
* Search Filter
* Category for furniture type
* Shopping cart
* Item description, cost, condition, material
* Shipping address
* Purchase History
* Listing History
* Stripe integration for credit card processing
* Customers can be notified of new furniture available that has been posted

Technology:

* Frontend: React
* Backend: Go with MongoDB

**Backend**

In the first phase of my project, I created high level outlines and diagrams to visualize how my web app would work. I needed to make sure I had my scope in front of my eyes, all the features and processes drawn out, so that when I went to implementing it, I wouldn't be stuck deciding what to do next and ending up failing because of scope creep.

So, in my User Activity Diagram, I outlined the general flow of processes for an authenticated user and a guest visiting site. Nothing too detailed yet, but just enough information to understand how the web app works.

I wanted to implement the backend first and the login and signup part seemed the most fundamental to tackle first, so I thought about what the form data would look like in the request. I create a very basic visual, not including the extra features.

A diagram of a company

Description automatically generated

As for my MongoDB database, I have four collections:

* + Listings
  + Receipts
  + shippingAddresses (which later turns out to be pointless as it can’t be used to autofill stripe’s checkout form\_
  + users

Below is the schema for each collection:

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generated

I started implementing my project by organizing my project and its directories. I have two main folders: backend and frontend. I create a new Go project in my backend and set up necessary folders for organization.

**File Structure**

|--backend/

|--api/

|--db/

|--tests/

|--types/

|--util/

|--main.go

|--frontend/

|--src/

|--assets/

|--components/

|--contexts/

|--pages/

|--tests/

|--types/

|--util/

|--App.css

|--App.tsx

|--main.tsx

**List of endpoints**

* POST /login
* POST /signup
* POST /logout
* POST /list\_furniture
* GET /get\_furnitures
* GET /get\_furniture/{listingID}
* GET /account
* PUT /account
* POST /checkout (with POST /checkout\_webhook for Stripe API)
* GET /account/address
* POST /account/address
* PUT /account/address
* DELETE /account/address/{addressID}
* GET /account/purchase\_history
* GET /account/purchase\_history/{orderID}
* GET /account/furniture\_listings
* POST /subscribe
* POST /unsubscribe
* GET /recent\_listing

From these list of endpoints, I tackled /login, /signup/ and /logout first, since it is the most fundamental, as most of the app’s features can only be used when authorized; it makes the most sense.

Inputs for /login and /signup should be a JSON encoded string of the user’s login and signup information, and the output should return “success” as a string, if the input is valid, or return an error for the invalid cases

A screenshot of a login form

Description automatically generatedA screenshot of a login form

Description automatically generated

The test plan for /login and /signup is to run each test case, where the payload is a single string, formatted in JSON containing the credentials and to ensure each case gets the expected result. If the credentials are valid, a “success” should be returned with a 200 code. If the payload is invalidly formatted, or fields are missing, or the credentials are invalid, it should return one of the 400 codes with an error message as a plain string.

And once the user has logged in, they also need to log out. When they’re logged in, they will have a session stored in memory, which can be retrieved from the client’s session cookie and used to index the map storing the client’s session and then be deleted, essentially logging them out.

After that, I began working on /list\_furniture. When a user creates a new furniture listing, they need to provide a bunch of information on their furniture. Here is the struct in Go.

A screen shot of a computer code

Description automatically generated

ListingID, UserID, and Bought, are the only fields that are set in the backend and which the user doesn’t have to provide

Upon a successful furniture listing creation, the listingID of the new inserted listing should be returned. ObjectIDs in Mongo are hex values and they get generated automatically for each document.

So, in the test, it checks that a string was returned. A valid test case returns a hex ID and a 200 code. Using that hex ID, we compare it against the database by searching for the document in the collection using the returned ID from the test request. The test also checks for validation of the form input and returns an error message if one or more fields is missing.

GET /get\_furnitures & GET /get\_furniture/{listingID}

Next, was writing the tests for /get\_furnitures and /get\_furniture/{listingID}. The tests for these were not complicated at all. For /get\_furnitures, the handler function queries the database listings collection to retrieve all the documents and puts them into an array and returns it in JSON. For /get\_furniture/{listingID}, it does the same thing, but only looks for one document with the listingID and returns the one object in JSON. Errors occurring during the database operation or the encoding into JSON are returned in the response with an error message and the appropriate status code.

For valid test cases, we should expect to see a 200 with the appropriate data. Invalid test cases for /get\_furniture/{listingID} really much only includes the case where the client provides an invalid listingID, in which case a status code of 400 for bad request is returned to the client. The other possible case is an HTTP method that is not GET.

Implementing the handler function includes getting the query param (for /get\_furniture/{listingID}, querying the database, storing the data in an appropriate data structure, encoding that structure into JSON, and then returning that in the response, stopping and returning any errors occurred along the way.

GET /account & PUT /account

The only data related to /account is the user’s password, phone, and email, which the user can see and update.

Writing tests for these were a bit more tedious. For GET /account, it included simulating a logged in session, retrieving the data of the test account, and then running the test case where the response from the request is then compared with the data retrieved prior to check for a match. For PUT /account, the updated data in the database is compared with the input, which is the changes the user has made to their password, phone, and/or email. Writing the implementation for GET /account was simple. For PUT /account, if a user changes their password, that new password also needs to be hashed, which I almost forgot to do. If users didn’t make any change to their information, then the input fields will be empty or nil, which means the Mongo database will not update anything.

POST /checkout & POST /checkout\_webhook

The logic for the checkout processing was complicated enough that I couldn’t write the test first because I did not know what to expect from the Stripe API, which means I didn’t know what the input and output would be for the tests initially. So, writing the implementation is what I did first.

All I knew was the user’s shopping cart would be needed to process the payment. The Stripe API has two kinds of checkout system you can use: an embedded checkout form on your website or a link to a stripe-hosted checkout page. I chose the latter since it seemed the simplest after viewing the examples on their website. In order to do use this API, information needs to be provided so that it can be displayed on the page. This means I need to retrieve all the details about each furniture listing that the user has purchased and is checking out.

The shopping is a string array of listingIDs, which then gets iterated to query the database and insert the resulting document, after being parsed, into an array of FurnitureListing, which is the same type used for listing a furniture. Now, I can access its properties and create the Stripe checkout session.

A screen shot of a computer code

Description automatically generated

When using the Stripe API, you also need to create a test account on their website to get the API key. Below is a snippet of the logic for creating a checkout session using the Stripe API.

A screen shot of a computer code

Description automatically generated

After defining the arguments for creating a checkout session, you then get a URL to the Stripe-hosted page with everything set up, which gets returned in the response to be used in the frontend.

A screen shot of a computer program

Description automatically generated

After all that, you get a page that looks like this:

A screenshot of a computer

Description automatically generated

When the user finishes and clicks the “Pay” button, the Stripe server sends a request to another endpoint, one in which you define in your Stripe dashboard on the website. I called mine /checkout\_webhook.

Because cookies aren’t sent with this request, you have to send the user’s authentication information through the Metadata property when creating the checkout session parameters.

A screenshot of a computer program

Description automatically generated

Then, you will be able to access all the authentication information in the handler function for /checkout\_webhook

A screen shot of a computer

Description automatically generated

Using this, you will be able to finalize the checkout, such as updating the furniture listing, creating a receipt stored in the database, and adding the purchase to the user’s purchase history.

When writing the test for this, my goal was to check if the /checkout endpoint would return the proper response, which is the link to the Stripe-hosted checkout page. I think it’s almost impossible to actually test, besides doing it manually, because each time a different link is returned. So, instead, I just made sure to check my API responded appropriately to invalid inputs for /checkout.

GET /account/address, POST /account/address, PUT /account/address, & DELETE /account/address/{addressID}

The /address endpoint handles the user’s addresses, which I thought I would be able to use to autofill the Stripe checkout form shipping address field, but apparently, after I found out too late, it is impossible to prefill it. So, this feature was kind of pointless. I left the code for it, though, as it still works as expected, excluding the shipping address field autofill.

Users can create shipping addresses, have at most one default address (to be used for the autofill), and can edit and remove addresses, as well as change their default address.

Writing tests for these were not complicated. For GET, I compare the result returned by the server to the data held in the database. For POST, I check to make sure a 200 and “success” was returned, which means the new address was inserted correctly into the database. For PUT, I compare the input to the new data in the database to ensure that data in the database reflected the changes in the input. And for DELETE, I check to make sure I get a 200 and a “success” for the delete operation, and then use the addressID that was given as input to query the database and check that I get no documents returned when filtering by the ID; that means it was deleted successfully.

Implementation was straightforward, based on the tests. PUT /account involved a bit more work than the others because the input validation required more logic to consider. If the input was empty, then that means no changes were given, so an error must be returned. If the changes included setting a new default address, then I’d have to mark the old default address as invalid and set the provided address as the new default address.

GET /account/purchase\_history, GET /account/purchase\_history/{orderID}, & GET /account/furniture\_listings

These endpoints handle the user’s history of all the furniture they’ve purchased and have listed.

When a purchase order is finalized in /checkout\_webhook, a receipt is created with a reference to the buyer’s userID, which is just called userID in the database, and a reference to the seller’s ID, sellerID. Then, it gets stored in the receipts collection in the Mongo database. From the database, we can retrieve a user’s purchase history by filtering with their userID and see all the documents returned. That is for GET /account/purchase\_history.

GET /account/purchase\_history/{orderID} has the same logic, except it filters with the orderID and the ID of the receipt to return one document.

GET /account/furniture\_listings has the same logic as well, but this time is queries the listings collection in the database and filters it using the user’s ID because the listing has a reference to the user who posted the listing, and then all of their furniture listings are returned.

I wrote tests for GET /account/purchase\_history and GET /account/furniture\_listings, but not for GET /account/purchase\_history/{orderID} because the code was very basic and I was confident that it wouldn’t bug out. It works.

The tests were a simple checking to make sure a 200 was returned because that means the data was retrieved and returned, encoded into JSON, without any errors.

POST /subscribe & POST /unsubscribe

These two endpoints deal with subscribing/unsubscribing the user to receive/stop receiving email notifications of each new listing. It sets the state of the subscription status.

I didn’t write any tests for either of them. The implementation for both was like 20 lines of code. The logic was very simple. Actually, they both have the same code, except it differs in one word, which is the state: true or false. Perhaps creating two different endpoints with the same repeating same code was not necessary.

A screen shot of a computer program

Description automatically generated

The code for HandleUnsubscribe is literally line 12, but instead of “true”, it’s “false.”

The actual logic for sending the emails runs during POST /list\_furniture in a separate thread, or goroutine, so that it doesn’t yield and delay the response for /list\_furniture.

Using the built-in Go net/smtp package, sending an email involves authenticating by logging into the email you’re going to use to send.

A screen shot of a computer

Description automatically generated

My password is stored as an environment variable on my system and the host is smtp.gmail.com. Once authenticated, you create the message you want to send.



Now, because I ran the function for sending emails in another goroutine, handling errors would be a bit wonky as it’s not synchronous. Because this feature of the web app – sending emails to subscribers – is not, at least I think, that important, I figured that if there is an error sending an email, then what’s the big email if the subscribers misses a notification for one furniture listing. They’ll probably just get the next one. So, I didn’t bother handling the “err” variable.

I did have the option of retrying sending the emails if an error occurred, but I don’t think it was worth it.

I didn’t write an automated test for this because I have no idea how I would write it. Instead, I just did it manually by listing a furniture on another test account and then I watched as I received the email update for the furniture listing.

GET /recent\_listing

This endpoint fetches the most recently posted furniture listing and returns it in the response. In MongoDB, the ObjectID of documents include a timestamp referring to the insertion of the document. So, by sorting the collection by \_id (ObjectID) as -1 and limiting it to return one document, it will give us the most recently added document. Below is a snippet of the code.

A screenshot of a computer program

Description automatically generated

I also did not write an automated test for this because I was confident. Besides, I would just be using the same code to retrieve the value from the database to compare with the result from the endpoint handler, and that seems redundant. I’ll see if it works, if it all works, when I make a request to the it from my frontend.

**Frontend**

To start the frontend, I created designs for some of the layouts that I wanted to look nice. Below are the layouts.

*Login and Signup page (as shown previously way above)*

A screenshot of a login screen

Description automatically generated

*Home page*

A screenshot of a computer

Description automatically generated

*Navigation bar*

A screenshot of a computer

Description automatically generated

*Market page*

*A computer screen shot of a computer

Description automatically generated*

Writing the algorithm for the search filter of the market page was a bit annoying. The filter section using the select input was no big deal, but refining the search using the search input, which is a string, was tricker.

What I did was split the search string into an array of words, all transformed into lowercase. Then I iterated through the array, checking if each word is an appropriate furniture attribute so that I could append it into another array meant only for that attribute. So, if the word is “queen”, that would be a bed size, so it would go into the bedSize array. If the word is a material type, it would go into the material array. If the word can’t be found in any of the furniture attributes, then searching for the word in the title of the furniture listing will be the last option. If the word is in the title, it gets added to the title array.

All that logic is encapsulated in a createSearchCategories function, which returns a SearchCategories type; an object.

A screen shot of a computer code

Description automatically generated

Then I would filter through the dataset of furniture listings using the SearchCategories object, checking if each word in each array of the object is found in any of the furniture listing’s attributes or title.

A screen shot of a computer program

Description automatically generated

*The code could be better though.*

*Purchase History page*

*A screenshot of a document

Description automatically generated*

*Detailed Receipt page*

*A screenshot of a computer

Description automatically generated*

*Address Creation form*

As I explained before, the address feature of my app, turned out impossible because I wouldn’t be able to prefill the checkout form shipping address field with my own data. If I was able to do so, it would probably be some convoluted way that I didn’t think was worth doing since this address feature isn’t as important as the others. I really only wanted it for a nice quality of life UX. But, here is what the general layout of the form looks like, anyway.

A screenshot of a computer

Description automatically generated

Establishing the routes was one of the first things I did. Some routes had to be private, meaning only authenticated users can access them, and the rest were public. For example, the /dashboard route is private and the /market route is public.

A screenshot of a computer program

Description automatically generated

All these routes define different pages, which I’ve put in a **pages** folder to organize them. Then, I also had a **components** folder for more reusable and isolated functionalities, like the navbar and my image slider, which I could just plop into anywhere I needed it.

I also created contexts, which allowed me to share states across nested components very easily instead of directly passing them through all my nested components as props and then creating the prop drilling issue. They’re basically like global variables that only certain functions can access, except in my case, I just shared it with my entire application because I needed it everywhere. I had a context for my authentication state, the shopping cart, and account data.

Writing the logic for all my React components was very straightforward. It was a matter of determining what states I needed and how my HTML should be structured so that it made styling with CSS easier. There were only very few components that contained more of the complicated than average logic, such as the market filter.

Most of my time working on the frontend was spent on CSS; I wanted it to look good, or at least, not something that looks sketchy. This was a nice opportunity to better my styling skills.

Testing

After finishing styling my components, I then moved on to very basic testing with Selenium, which only includes testing the Login and Signup page.

Selenium is a bit wonky because it throws out some weird error messages in the console about some browser setting that I don’t know how to fix, but the tests still run and it doesn’t affect the test results.

For the **login** page, my test plan was to:

* + Test blank username
  + Test blank password
  + Test valid login
  + Test invalid login

Valid logins will get redirected to the dashboard, invalid logins will return an “Invalid login” message, and a blank username and password will return a “Username cannot be blank” error message and a “Password cannot be blank" error message, respectively.

Here is what the code for one of the test looks like:

A screen shot of a computer program

Description automatically generated

The other 3 tests for the login page follow this exact same logic, as well.

For the **signup** page, my test plan was to:

* Test blank fields
* Test a used username
* Test a used email
* Test mismatching password

I didn’t want to test for a valid signup because I’d only get to use the credentials once and I wouldn’t be able to reuse it; it wouldn’t make sense.

Here’s the test for the first case

A screenshot of a computer program

Description automatically generated

1. **In what areas did things not go well? What parts went exceptionally well?**

The only parts I had trouble figuring out was learning the Stripe API. That took quite a bit of time. I later found out that I wouldn’t be able to incorporate this default address feature I planning on doing, which was to prefill the checkout form shipping address field with the user’s default shipping address, but Stripe doesn’t allow us to prefill fields (at least that’s what I’ve learned after searching for an hour). Everything else went well, though. I especially enjoyed working on the backend logic because it was less annoying to deal with, but very satisfying once I knew it was working and passing my tests. The frontend was alright; it went well, but it was monotonous. It was mostly me figuring out which CSS styles looked best.

1. **What technology stack did you use for your project? Why did you choose this option? Give supporting details with specific reasons, whether human or technology.**

I used React with TypeScript on the frontend and Go with MongoDB on the backend. I chose React with TypeScript because I’ve worked with React before in my previous projects and I enjoyed working with it because of how easy and fast it is to create more complex webpages; it’s very modular. And TypeScript is just so nice to have because of the types. I don’t like working without types because it’s difficult to know what each variable is.

I chose Go because I was already learning it at the time and it’s very easy to build an web API with it; it feels like it’s made for it. Later into my project, Go 1.22 was released and it became so much easier to define routes because they added enhanced routing patterns, which only further increased my like for Go. And I decided on MongoDB because I wanted to work with a different kind of database to have a working knowledge of how it operates.

1. **Does your project include at least two platforms? Identify them and the communication that happens between them.**

My project has a client and a server. The server provides the client with information whenever the client asks, which includes

* + Login, Signup, and Logout
  + Getting a user’s account information, which includes
    - User’s purchase history
    - User’s furniture listings
    - User’s shipping addresses
  + Updating a user’s account information
  + Deleting a user’s account information
  + Getting all the furniture listings
  + Subscribing to receive email updates
  + Checkout out an order
  + Fetching the most the recently posted furniture listing

1. **What does your application use for persistent storage? If a database, provide an ERD or other schema documentation. If file or cloud storage, give details on the structure. Why did you choose this method?**

My application uses MongoDB. I wanted to use MongoDB because I had only ever worked with SQL with MySQL database and ever since I heard the term non-relational database in my database class, I was curious. Then I came upon MongoDB and thought this project would be a good opportunity to learn it.

In my database, I have four collections defined as so:

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generated

I took some of the things I learned from SQL and applied it to my Mongo database. For example, I have references in some of the fields to avoid duplicating data. The listings collection has a reference, userID, to the id of user who posted the listing. With this structure, I am able to fetch every listing a user has posted based on that userID. With this same structure, I am similarly able to fetch a user’s purchase history from the Receipts collection. I didn’t plan this schema from the beginning. I had to make changes along the way when I stumbled upon an issue.

1. **How did you implement the requirement for hashing or encryption? What algorithm did you choose and why? Are there any downsides or concerns about the way it is implemented?**

I used the **golang.org/x/crypto/bcrypt**Go bcrypt ibrary. I chose it because I read that it was one of the recommended encryption algorithms that is commonly used for passwords and is great against brute-force attacks. So I searched if Go had a library for it and it did, so I used it.

1. **Identify the areas of your application that use an object-oriented component. A class diagram is strongly recommended here.**

Really, the main area of my application that uses an object-oriented component is my entire backend because in Go, you can create structs, attach methods to them, and define types and such, etc. The code I wrote for my session manager is object-oriented and it is a singleton because there is only ever one SessionManager. The SessionManager struct has these methods:

* + CreateSession()
  + GetSession()
  + DeleteSession()
  + GetSessionID()
  + IsSessionExpired()
  + IsLoggedIn()

Also, every type in Go is basically just an object because each type has its own attributes.

1. **How does user authentication work for your application? What information is stored, what is the process for account creation, login, and management?**

This is the scheme for the users collection, which is where all the information regarding authentication is stored:

A screen shot of a computer code

Description automatically generated

When a user sends their signup request to the server, the server validates each field, checking to make sure there are no blank fields, that the password and confirm password matches, and that the username and email is unique. Then it sets the user’s balance to 0 because that’s the default starting balance. Then, a new unique sessionID is created for them. Before saving all this in the database, the password is hashed with the bcrypt algorithm, which is what gets stored, and then everything is inserted into the database.

Once the signup process is complete, the user will be instructed in the frontend to login with the newly created account. When the login request is sent, the server reads the username input, and queries the users collection for the password. If the username can’t be found, an error is returned. If the password input when hashed with bcrypt doesn’t match the stored hashed password, then an error is returned. If all those checks pass, then the server checks for an expired session based on the cookie sent in the request.

If there is an expired session or no session at all, a new sessionID will be generated, stored, and sent to the client in a cookie. If it’s not expired, then just log the user in. The session gets stored in the server’s memory until they log out, which is when the session will be deleted from the server’s memory and when the client’s sessionID cookie is deleted.

1. **What form does your in-app help and documentation take? Why did you choose this form?**

In takes form in the intuitive and easy-to-read headers kind of way. I didn’t consciously choose what form it would take; I didn’t methodically plan anything for this. I just knew that I needed to provide a nice user experience through the use of clean and simple UI, with good input validation and error indications for forms, and nice, clear labels.

1. **How did you test your project? Include your formal test plan, results of testing, and explanations for any testing failures.**

Most of the testing was directed towards my backend because I found it was easier to test. I employed the use of TDD for most of it, for the endpoints where I knew the defined input and outputs. I explain more in detail in my documentation which is in question b)., but the general test plan for the backend tests were to

* + Test the endpoint with invalid inputs
  + Test the endpoint with valid inputs
  + And comparing the results inserted into the database with the input to ensure the two are the same and the request was processed successfully.

For the frontend, I used selenium to simulate the user input, only testing the login and signup page.

For the **login** page, my test plan was to:

* + Test blank username
  + Test blank password
  + Test valid login
  + Test invalid login

For the **signup** page, my test plan was to:

* + Test blank fields
  + Test a used username
  + Test a used email
  + Test mismatching password

1. **Reflect upon the last 3 months of work, if you had to do it again, what would you do differently? What recommendations do you have for future iterations of this class?** 
   1. **Please identify at least three things you liked and want to make sure happen if you had to take the class again**
   2. **Please identify at least three things you disliked and wish were not a part of the class.**

**Three things I liked and want to make sure happen again:**

* + I liked the freedom of not coming to class once class expectations were established
  + I liked used Git and committing to GitHub to track my code versions; I felt much safer knowing I had a backup of all my previous code, incase I needed to revert to something. So many people are using GitHub, anyway, so it’s great to get used to.
  + I liked seeing you help other people with their projects because I was able to learn from it.

**Three things I disliked and do not want in the class:**

* + I disliked not being able to work on a collaborative project; I would’ve enjoyed working with a partner on a project because teamwork is a very nice skill to have. And there are things to keep a person’s integrity in check when working in a group, such as seeing who has committed to GitHub recently and what they committed. It could have at least been an option.
  + I disliked that I didn’t get a chance to deploy my web app project online because I don’t know how and it would’ve been nice to learn in class.
  + I disliked that you didn’t give comments on the weekly updates. It’s another nice opportunity to give feedback on our project.