

GatorMart Documentation

April 1st, 2022

Sprint 3

Team Members:

Front End

1. Nitin Ramesh

2. Bhanu Prakash Reddy Palagati

Back End

1. Gowtham Reddy Eda

2. Vamsi Krishna Reddy Komareddy

Introduction:

The GatorMart application is an online market build with Angular and Go language. Built from the ground up, the application aims to seamlessly connect buyers to their products and sellers to their target audience. Its standout features include allowing the users to select a target audience while buying or selling. This allows to buy products or sell products in single or wholesale quantities while specifying a target audience such as students, professionals, farmers, etc.

Tech Stack:

Frontend: Angular 11.0 with TypeScript
Frontend Test cases: Cypress and Jest Framework
Backend: Go Language
Database: MySQL

Frontend:

The front end of the application is built using Angular 11. Angular is a framework rather than a package that provides all the essential functional requirements out of the box. This will force the developers to follow a pattern, as a result, we have fewer decisions to take for the organization and spend time more on what matters.

We have used material design and there is a package named Angular Materials which helps us to implement the material style components easily and efficiently.

Test cases have been written using Jest for unit testing and Cypress for integration testing, this has ensured the smooth integrated working of the frontend and backend.

The application demo consists of the following pages: Login page, Registration page, “List view” with product filters, create product page and the “Detailed view” with google maps integration:

1) Registration Page:

The new user registers for the GatorMart account at this page. The user enters the details and the details are validated before the user account is created. The user identifier is the email they register by and the password is minimum 8 characters with atleast 1 capital letter, 1 number and 1 symbol.

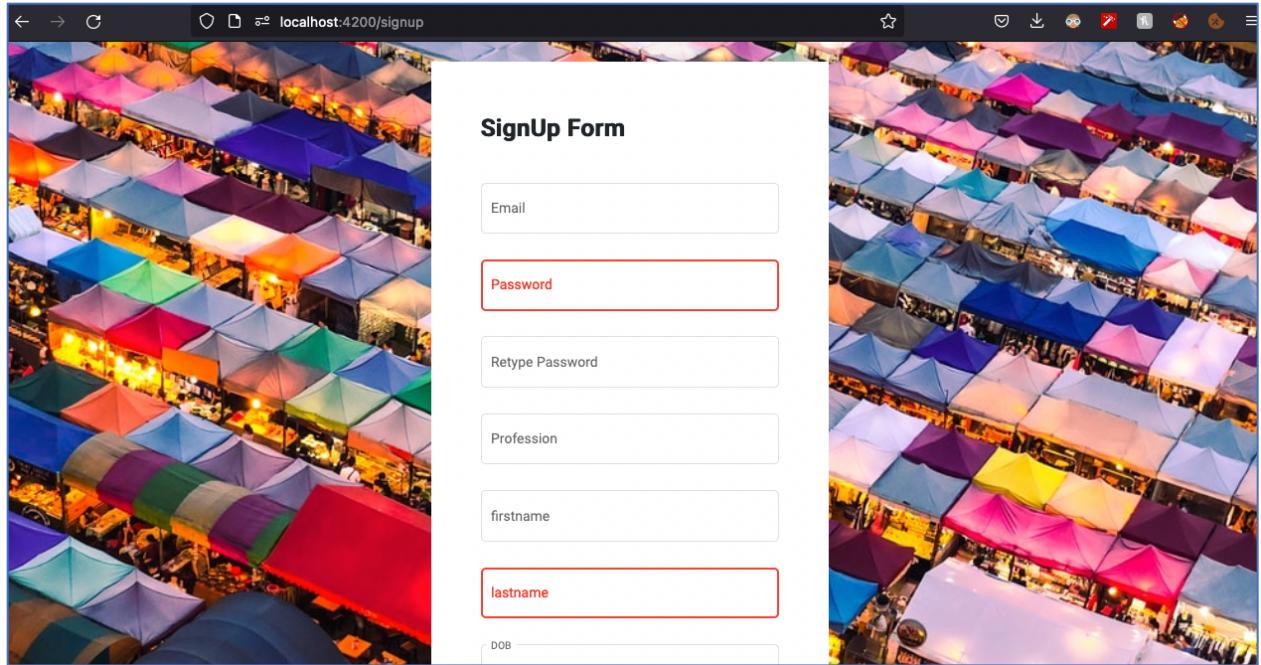


Figure 1: Registration Page

2) Login Page:

The traditional login of username and password is presented to the user upon visiting the GatorMart application. The username and password is taken and redirected to the main homepage of the application.

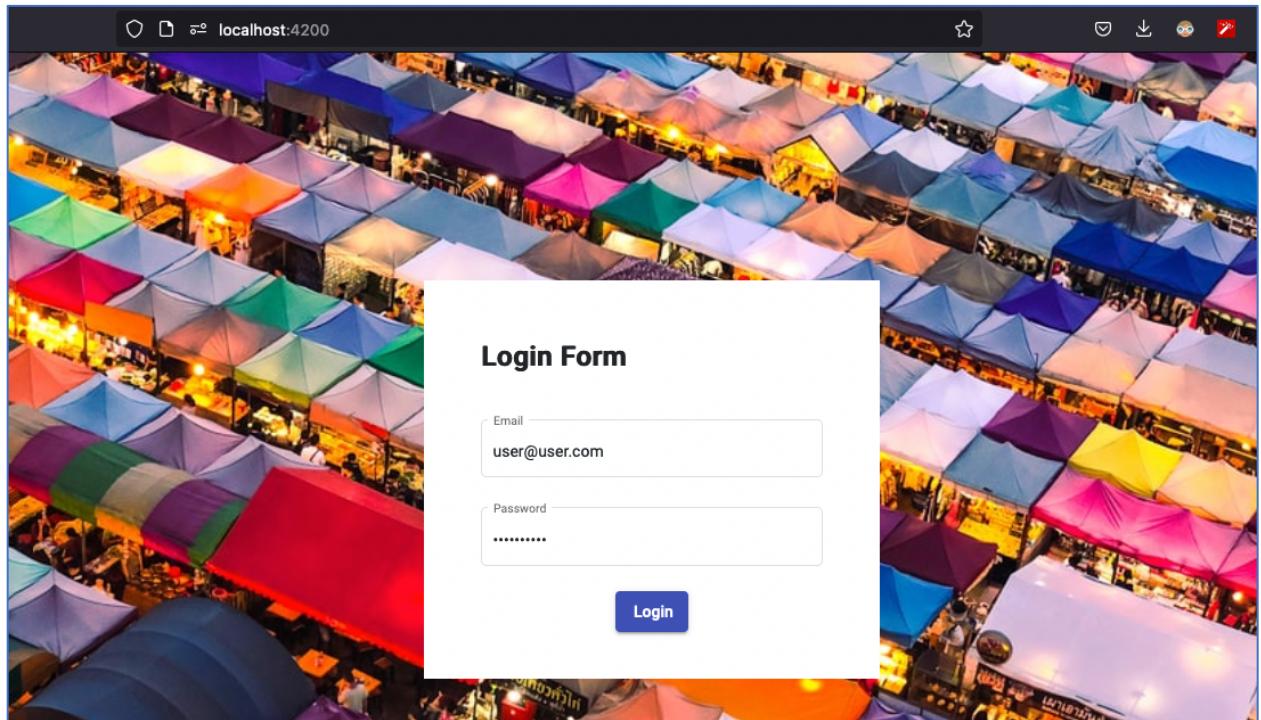


Figure 2: Login Page

3) List View:

The List view is the main screen that greets the user when opening the GatorMart application. It is a block view of all the products, that have been hand-picked by our algorithms to suit the user's taste.

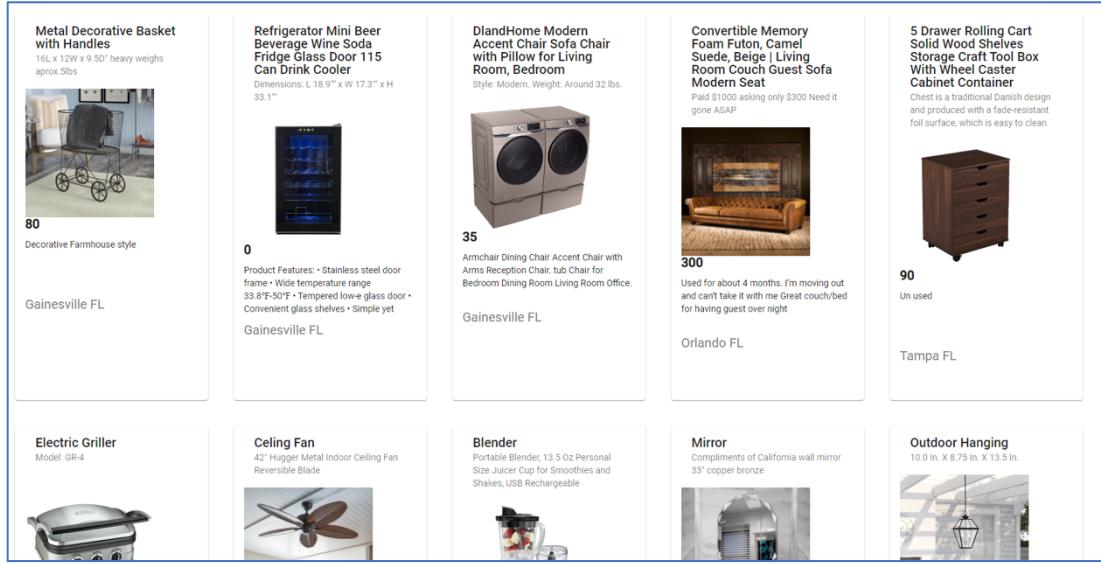


Image: List View

The products are displayed here in a small window shaped material design as shown below:



Image: Product Details

This method favors readability, as the most important product information is conveyed to the user directly as they scroll the different listings on the site.

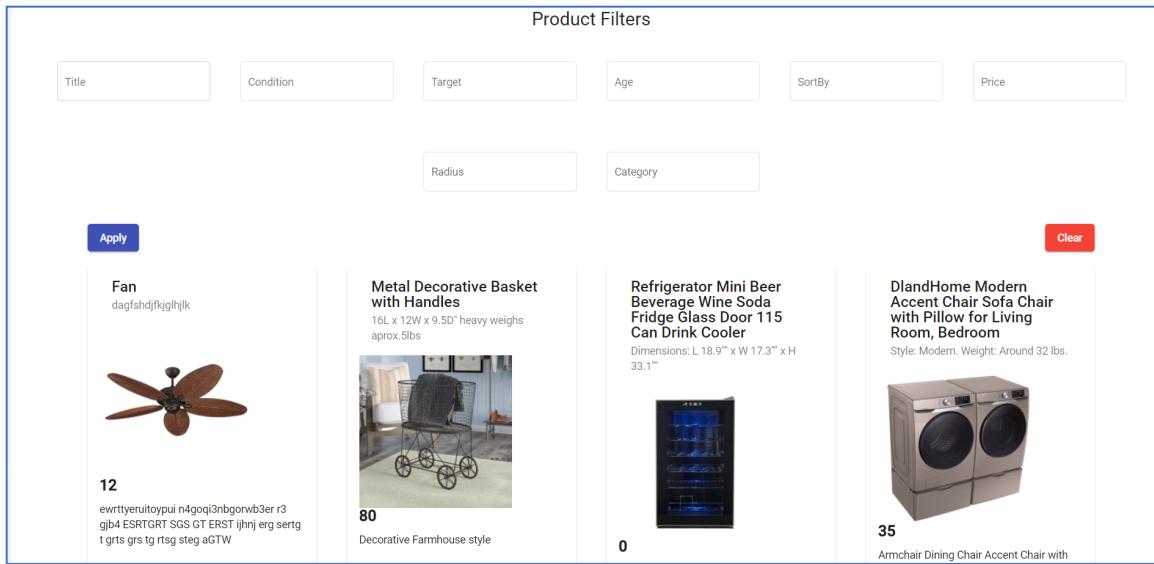


Image: Product Filters

The List view function also contains the product filters, with options such as title, condition, target, age etc. This allows the users to segregate the products and only list those that are relevant to the search parameters.

4) Detailed View:

The Detailed View is presented when the user clicks on a product in the List view. This view, as the name suggests, gives more detailed information regarding the selected product such as:

- A carousel of images of the product
- Age of the product
- Detailed description
- Price etc.
- Option to Edit/Remove the Add
- Map indicating the location of the seller

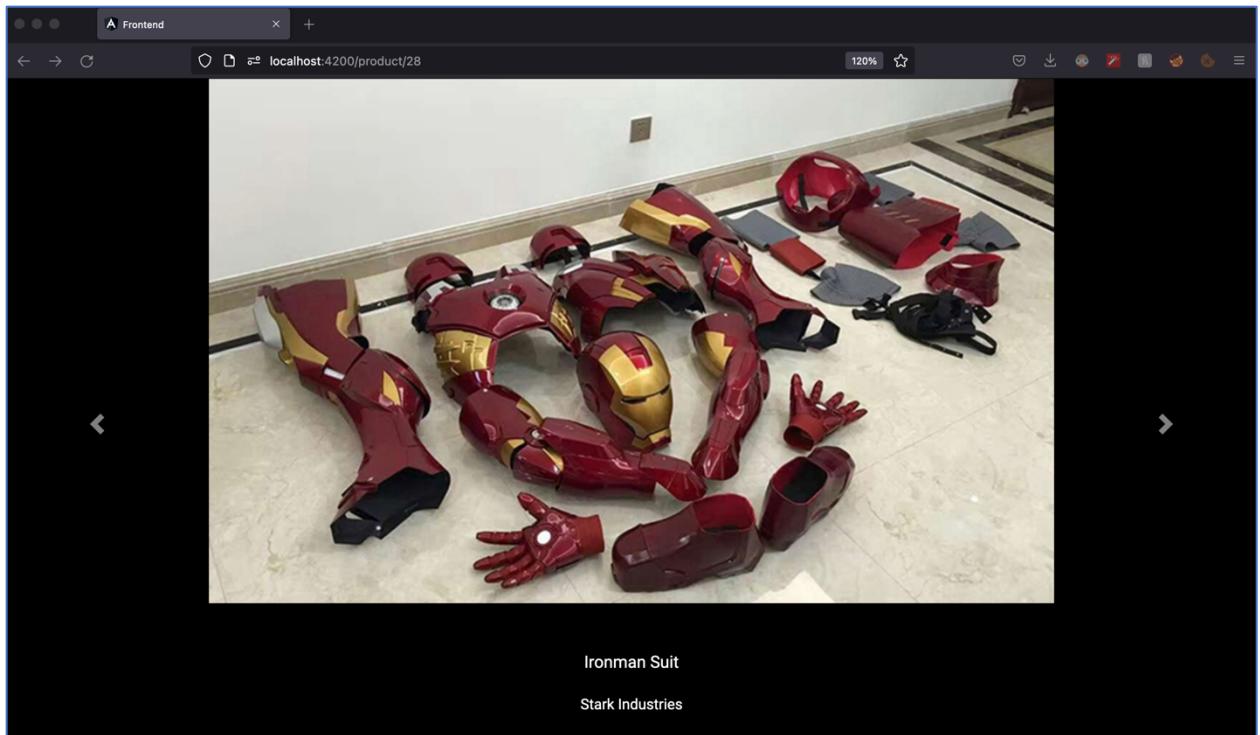


Image: Image carousel

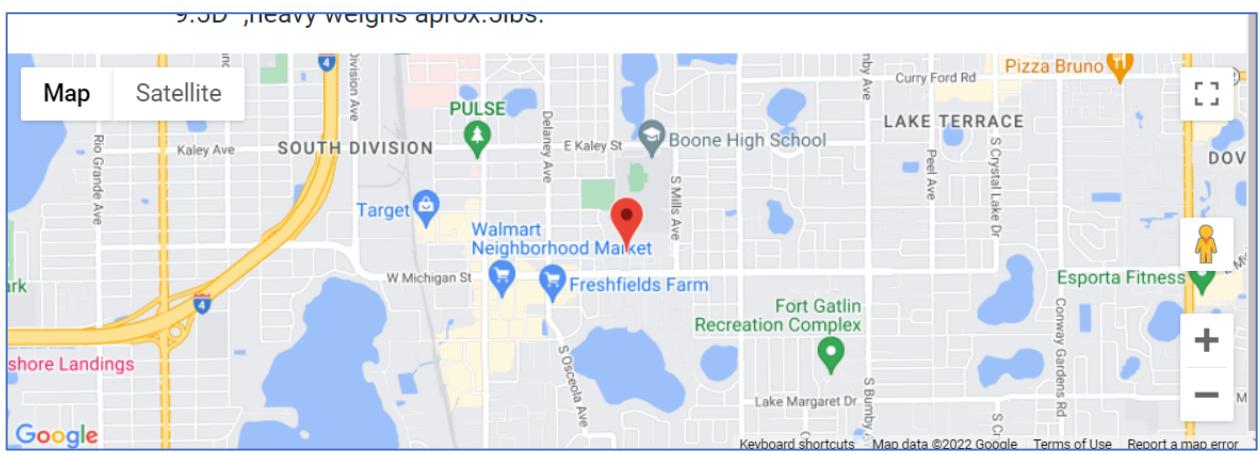


Image: Detailed View

All details are conveniently placed in one area with multiple images to give a better overall understanding of the product.

Edit function : The product details can be edited by the users via the edit button, where the updated details is entered and reflected in the site.

Thumbnail Image No file chosen Select only one image

Display Images No file chosen You can select multiple images

Title
Enter your product title here

Secondary Title
Give important product specs

ImageURL
Please upload images from the choose file above

Price
Provide the product listing price

Simple Description
Short but highly informative description

Description
Please provide some detailed information

Latitude
Locate me will come to your rescue

Image: Edit View

Delete function : The product can be deleted in the product details view. This will verify the deletion function with the user and proceed to delete the product.

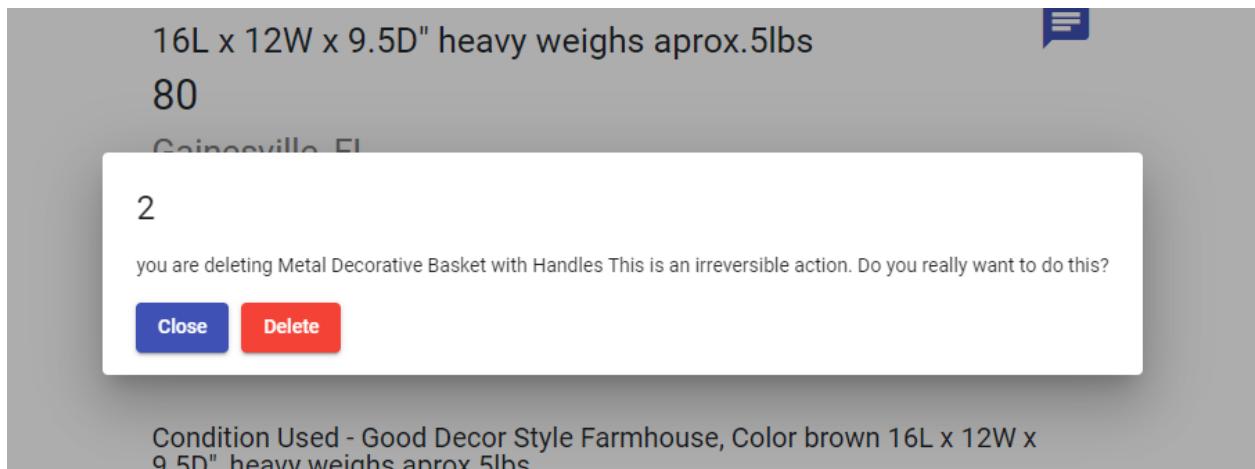


Image: Delete Ad

5) Create Product View:

The create product view confirses of a form where the user can add all relevant details and set the product up for listing. This includes the main image, carousel images and device specific data.

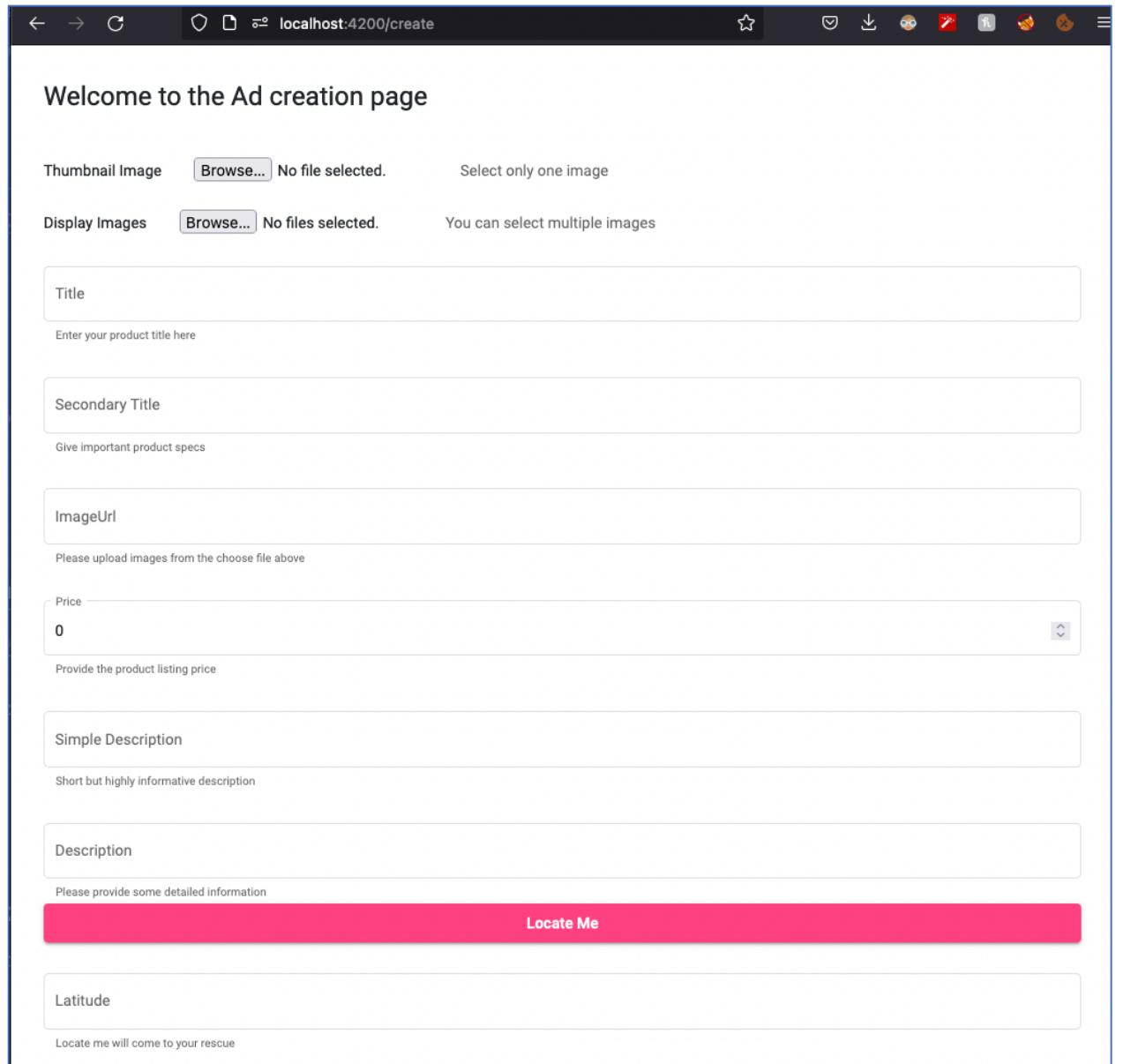


Image: Create Product View

Backend:

1) Register:

Added new fields and updated the struct of golang, some fields firstname, lastname, email, Password, Profession and DateofBirth which helps us to focus on the targeted audience for selling and buying of products.

The screenshot shows a Postman interface for a POST request to `localhost:8000/register`. The request body is a JSON object containing the following fields:

```
1 {  
2     "firstname": "firtnname",  
3     "lastname": "latnaoe",  
4     "email": "fiaime.lannnaoe@gmail.com",  
5     "password": "Passwnrd@001",  
6     "profession": "Student",  
7     "DOB": "1999-12-12"  
8 }
```

The response status is 200 OK with a response time of 188 ms and a size of 410 B. The response body is also a JSON object:

```
1 {  
2     "ID": 5,  
3     "CreatedAt": "2022-04-01T18:27:11.979-04:00",  
4     "UpdatedAt": "2022-04-01T18:27:11.979-04:00",  
5     "DeletedAt": null,  
6     "userId": 0,  
7     "firstname": "firtnname",  
8     "lastname": "latnaoe",  
9     "email": "fiaime.lannnaoe@gmail.com",  
10    "profession": "Student",  
11    "DOB": "1999-12-12T00:00:00Z"
```

2) Login:

Upadted login end point to generate JWT token based on userid, useremail and username.

The screenshot shows a Postman interface with the following details:

- Request URL: `localhost:8000/login`
- Method: `POST`
- Body content:

```
1
2
3     "email": "fiaime.lastname@gmail.com",
4     "password": "Passwprd@001"
```
- Response status: `200 OK`
- Response body (Pretty):

```
{ "message": "login success", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJFeHBpcmVzQXQi0jE2NDg5NDc10TEsIk1zc3VlZEF0IjoxNjQ40DYxMTkxLCJJc3N1ZXIi0iJHYXRvck1hcnQiLCJTdWJqZWN0IjoiVG9rZW4gZm9yIEhdhdG9yTWFydCBmcn9udGVuZCIsImF1dGhvcml6ZWQiOnRydwUsInVzZXJlbWFpbCI6ImZpYW11Lmxhc3RuYW1lQGdtYWlsLmNvbSIsInVzZXJpZCI6MiwidXNlc5hbWUiOjmaXJ0bmFtZSJ9.pP0G1Lp-q9YMKGmEnykbaioLN6wbwnDakXpRtLj0GNg" }
```

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJFeHBpcmVzQXQi0jE2NDg5NDc10TEsIk1zc3VlZEF0IjoxNjQ40DYxMTkxLCJJc3N1ZXIi0iJHYXRvck1hcnQiLCJTdWJqZWN0IjoiVG9rZW4gZm9yIEhdhdG9yTWFydCBmcn9udGVuZCIsImF1dGhvcml6ZWQiOnRydwUsInVzZXJlbWFpbCI6ImZpYW11Lmxhc3RuYW1lQGdtYWlsLmNvbSIsInVzZXJpZCI6MiwidXNlc5hbWUiOjmaXJ0bmFtZSJ9.pP0G1Lp-q9YMKGmEnykbaioLN6wbwnDakXpRtLj0GNg
```

Decoded EDIT THE PAYLOAD AND SECRET

The jwt.io interface shows the following decoded token details:

HEADER: ALGORITHM & TOKEN TYPE

```
{ "alg": "HS256", "typ": "JWT" }
```

PAYOUT: DATA

```
{ "ExpiresAt": 1648947591, "IssuedAt": 1648861191, "Issuer": "GatorMart", "Subject": "Token for GatorMart frontend", "authorized": true, "useremail": "fiaime.lastname@gmail.com", "userid": 2, "username": "firtnname" }
```

3) Productsave:

Updated save method to automatically update postedby field by taking user details from JWT token.

The screenshot shows a Postman request to `localhost:8000/product` using the `POST` method. The `Body` tab is selected, showing the following JSON payload:

```
8  "city": "gainesville fl",
9  "state": "florida",
10 "location_lat": "maps",
11 "location_long": "maps",
12 "target": "student",
13 "category": "Mobile",
14
15 "posted_date": "3rdFeb",
```

The response tab shows a successful `200 OK` status with a response time of `12 ms` and a size of `669 B`. The response body is identical to the request body, but includes an additional `posted_by` field with the value `2`.

Line Number	Field	Value
8	city	"gainesville fl"
9	state	"florida"
10	location_lat	"maps"
11	location_long	"maps"
12	target	"student"
13	category	"Mobile"
14	posted_by	2
15	posted_date	"3rdFeb"

4) Category Dropdown:

Added categories dropdown where users can select the category type only from the dropdown. Updated save product and update product end point to restrict the user selection of categories from the dropdown. It throws an error if the value does not present in the dropdown.

localhost:8000/categories

GET localhost:8000/categories Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description	...	

Body Cookies Headers (5) Test Results 200 OK 5 ms 237 B Save Response

Pretty Raw Preview Visualize JSON

```
1 ["Automobile",
2 "Mobile",
3 "ElectronicsAppliances",
4 "Furniture",
5 "Books",
6 "Sports",
7 "Pets"]
```

localhost:8000/product

POST localhost:8000/product Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

Body Cookies Headers (5) Test Results 400 Bad Request 7 ms 181 B Save Response

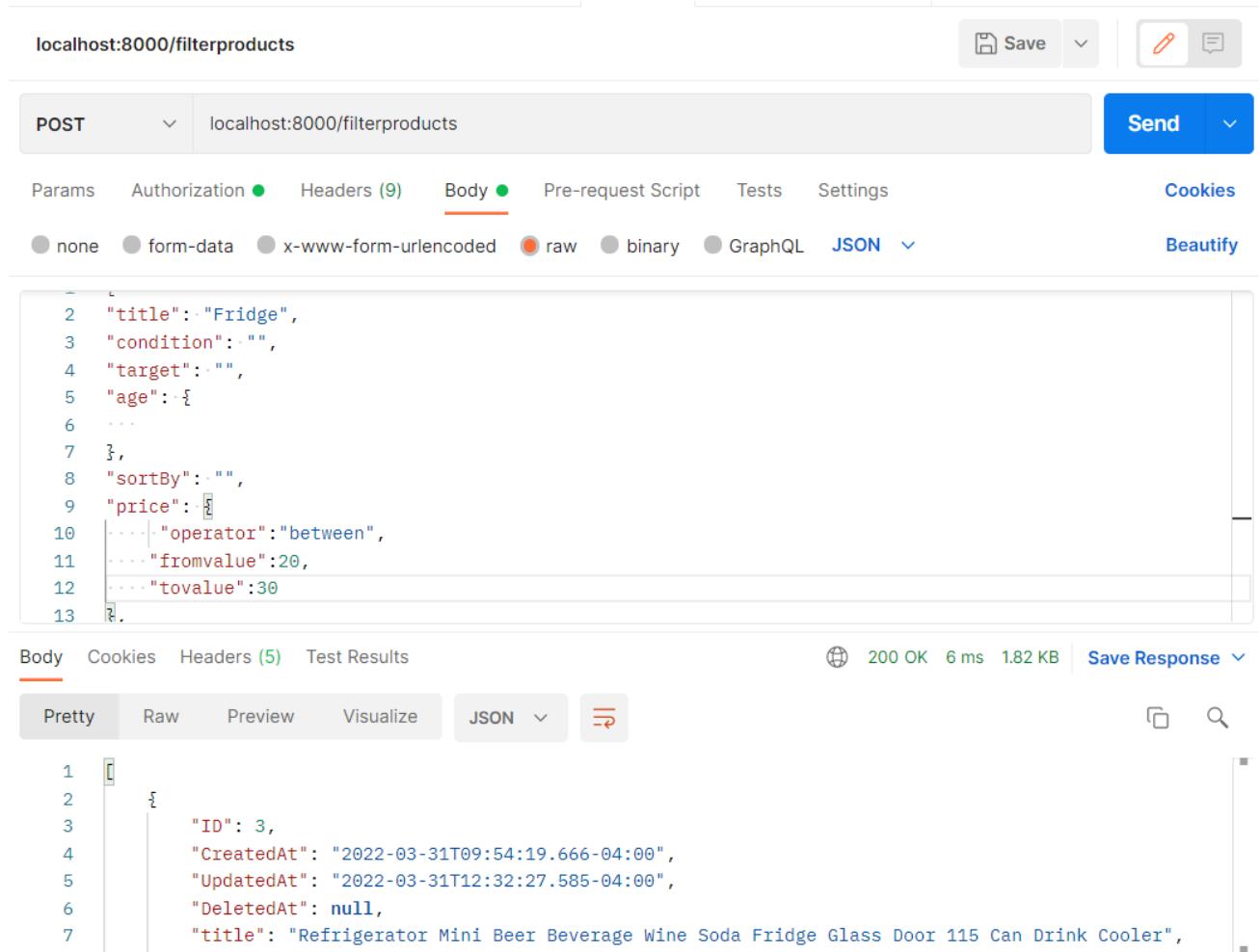
Pretty Raw Preview Visualize JSON

```
8 {"city":"gainesville fl",
9 "state":"florida",
10 "location_lat":"maps",
11 "location_long":"maps",
12 "target":"student",
13 "category":"Sample",
14
15 "posted_date":"3rdFeb",}
```

1 "Invalid Category"

5) Filter Products:

User can filter products based on partial title match, target, category and price based upon operator, fromvalue, tovalue.



The screenshot shows the Postman application interface. At the top, it displays the URL "localhost:8000/filterproducts". Below the URL, there are buttons for "Save" and "Edit". The main area shows a POST request to "localhost:8000/filterproducts". The "Body" tab is selected, showing the following JSON payload:

```
2   "title": "Fridge",
3   "condition": "",
4   "target": "",
5   "age": {
6     ...
7   },
8   "sortBy": "",
9   "price": {
10    ...
11    "operator": "between",
12    "fromvalue": 20,
13    "tovalue": 30
14  }.
```

Below the JSON editor, there are tabs for "Body", "Cookies", "Headers (5)", and "Test Results". The "Test Results" tab is currently active, showing a response status of 200 OK with a response time of 6 ms and a size of 1.82 KB. The response body is displayed in "Pretty" format:

```
1  [
2    {
3      "ID": 3,
4      "CreatedAt": "2022-03-31T09:54:19.666-04:00",
5      "UpdatedAt": "2022-03-31T12:32:27.585-04:00",
6      "DeletedAt": null,
7      "title": "Refrigerator Mini Beer Beverage Wine Soda Fridge Glass Door 115 Can Drink Cooler",
```

6) Unit test Cases:

Implemented unit test cases has been implemented for all the functions.

The screenshot shows a code editor interface with a dark theme. At the top, there is a status bar with tabs for PROBLEMS (8), OUTPUT, DEBUG CONSOLE, and TERMINAL. Below the status bar is a code editor window containing Go test code. The code defines a test function `TestLoginWhenPassWordInCorrect` that sends a POST request to "/api/login" with specific data and asserts the response status code. Below the code editor is a terminal window showing the execution of the test command. The terminal output indicates a failure for the `TestRegisterWhenSuccess` test and a pass for the `TestLoginWhenPassWordInCorrect` test. The total execution time is 0.465s.

```
run test | debug test
227 func TestLoginWhenPassWordInCorrect(t *testing.T) {
228     var data = []byte(`{
229         "email": "gatormart1@ufl.edu",
230         "password": "gatormart1@A1"
231     }`)
232
233     app := fiber.New()
234
235     req, _ := http.NewRequest("POST", "/api/login", bytes.NewBuffer(data))
236
237     response, err := app.Test(req)
238
239     if err != nil {
240         t.Errorf("Handler Returned a wrong status code")
241     }
242
243     assert.Equal(t, fiber.StatusNotFound, response.StatusCode)
244 }
```

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL

```
Test:           TestRegisterWhenSuccess
--- FAIL: TestRegisterWhenSuccess (0.00s)
FAIL
FAIL    main/products  0.465s

> Test run finished at 4/1/2022, 8:39:21 PM <

Running tool: /usr/local/go/bin/go test -timeout 30s -run ^TestLoginWhenPassWordInCorrect$ main/products
==== RUN  TestLoginWhenPassWordInCorrect
==== PASS: TestLoginWhenPassWordInCorrect (0.00s)
PASS
ok    main/products  0.538s

> Test run finished at 4/1/2022, 9:33:12 PM <
```