COLUMBUS LIST

Group H:
Nguyen, Trung Minh
Nguyen, Justin
Oberhelman, Andres William
Opial, Blake
Pardo, Devin Jakob

Table Of Content

Overview

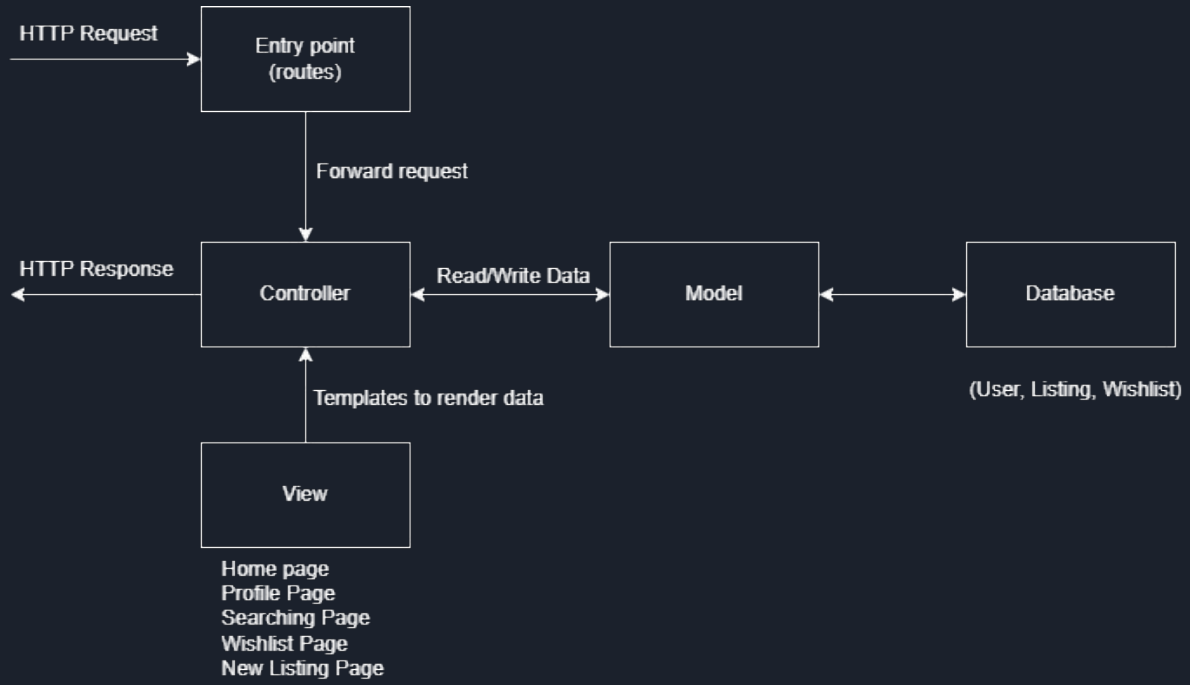
The Columbus University administration needs an online website called ColumbusList for its students.

ColumbusList is a classified advertisement website that should be designed to assist users in buying and selling products and services exclusively within the University. The website's goal is to provide an exclusive marketplace for university students to purchase and sell products while ensuring that all users are university students and that transactions are secure.

Project objective

Create a simple, secure web application

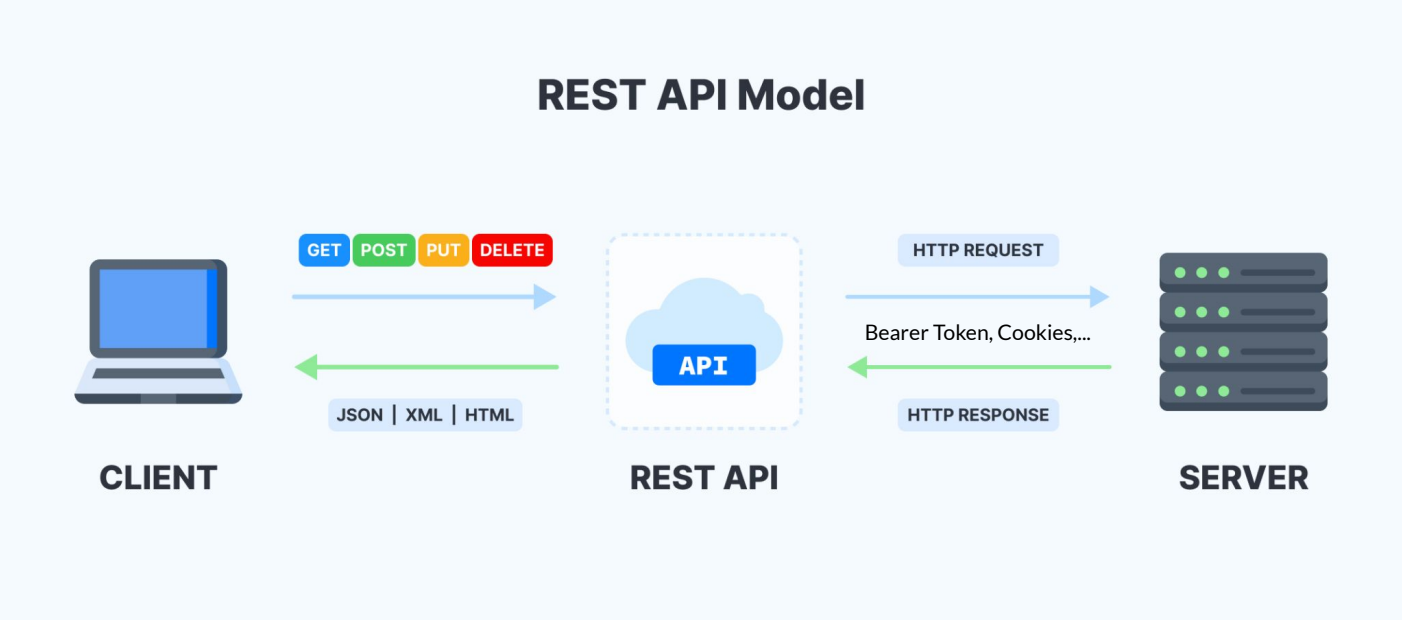
System Architecture



Database Schema

Back End Overview

Back End Overview



Credit: https://appmaster.io/blog/how-create-api-no-code

Authorization

POST **Sign Up**

{{URL}}api/v1/user/signup

User's name is only a field in the document. So if you have first and last name just concatenate them into single string.

BODY raw

```
{
    "name": "blake",
    "email": "blake@columbus.edu",
    "password": "pass1234",
    "passwordConfirm": "pass1234"
}
```

POST **Login**

{{URL}}api/v1/user/login

BODY raw

```
{
    "email": "trung@columbus.edu",
    "password": "pass1234"
}
```

Authorization

PATCH **Update Password**

{{URL}}api/v1/user/updatepassword

User must be logged in to change the password. So if you want to test this API, make sure you're already logged in

AUTHORIZATION

Bearer Token

Token

{{jwt}}

BODY raw

```
{
    "currentPassword" : "pass1234",
    "newPassword": "newpass123",
    "passwordConfirm": "newpass123"
}
```

GET **Log Out**

{{URL}}api/v1/user/logout

User

GET **Get User**

{{URL}}api/v1/user/

Given the user has been looged in, the API return the info of that user

Wishlist

POST **Create Wishlist**

{{URL}}api/v1/wishlist

User must be logged in to add a listing to wishlist.

The body of the request must have the _id of the listing.

Owner cannot add his listing to wishlist.

Not valid listing _id will result an error

AUTHORIZATION

Bearer Token

Token

{{jwt}}

BODY raw

```
{
    "listing_id":"6266f83dbfce521c3c51ba65"
}
```

GET **Get User Wishlist**

{{URL}}api/v1/wishlist

User must be logged to get the array of his wishlist listing.

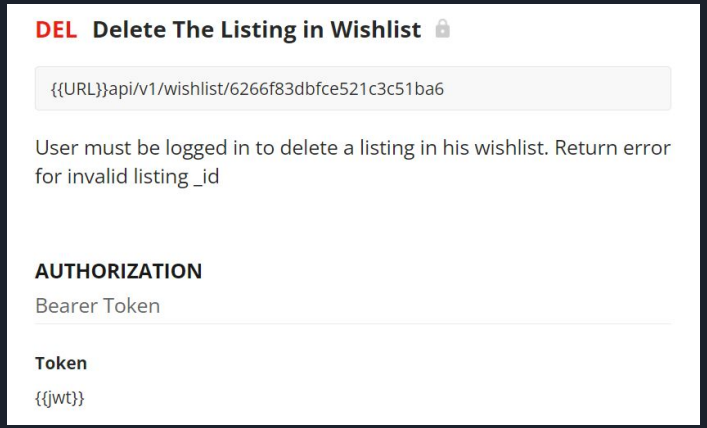
AUTHORIZATION

Bearer Token

Token

{{jwt}}

Wishlist


DEL Delete The Listing in Wishlist
{{URL}}api/v1/wishlist/6266f83dbfce521c3c51ba6
User must be logged in to delete a listing in his wishlist. Return error for invalid listing _id
AUTHORIZATION
Bearer Token
Token
{{jwt}}

Listing Model

String: title

Number: price

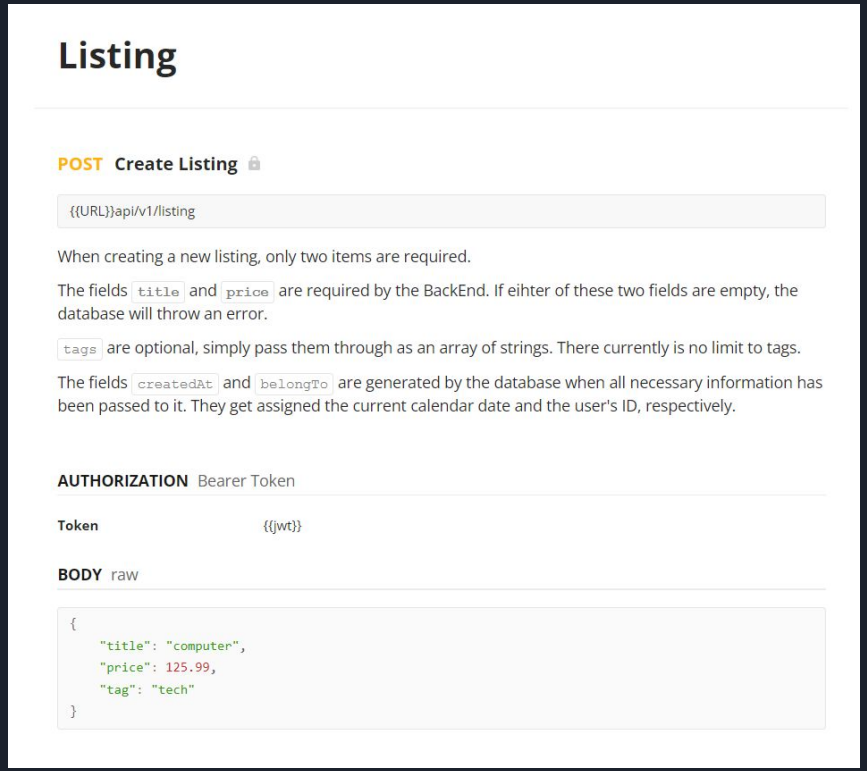
String Array: Tags

Date: createdAt*

String: belongTo

*Not directly passed into the model, but rather generated by the server itself

Create Listing



Listing

POST **Create Listing**

{{URL}}api/v1/listing

When creating a new listing, only two items are required.

The fields `title` and `price` are required by the BackEnd. If eihter of these two fields are empty, the database will throw an error.

`tags` are optional, simply pass them through as an array of strings. There currently is no limit to tags.

The fields `createdAt` and `belongTo` are generated by the database when all necessary information has been passed to it. They get assigned the current calendar date and the user's ID, respectively.

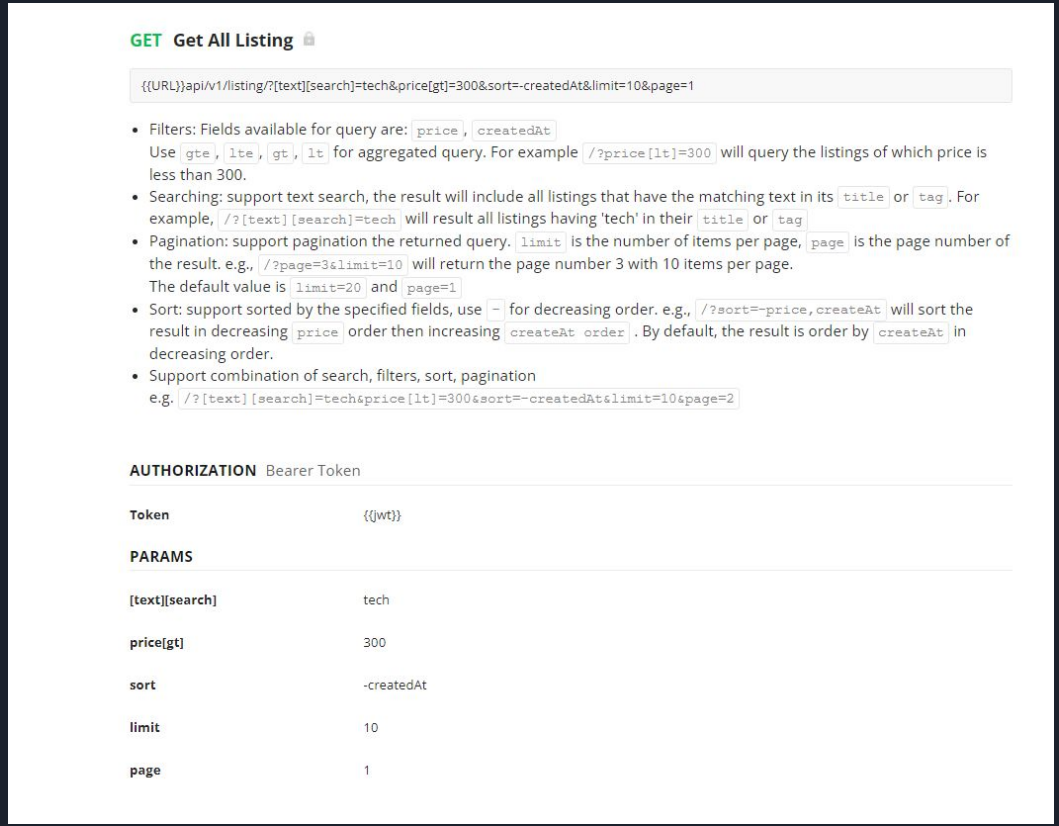
AUTHORIZATION Bearer Token

Token	{{jwt}}

BODY raw

```
{
    "title": "computer",
    "price": 125.99,
    "tag": "tech"
}
```

Get Listings



GET **Get All Listing**

{{URL}}api/v1/listing/?[text][search]=tech&price[gt]=300&sort=-createdAt&limit=10&page=1

- Filters: Fields available for query are: `price` , `createdAt`
  Use `gte` , `lte` , `gt` , `lt` for aggregated query. For example `/?price[lt]=300` will query the listings of which price is less than 300.
- Searching: support text search, the result will include all listings that have the matching text in its `title` or `tag` . For example, `/?[text][search]=tech` will result all listings having 'tech' in their `title` or `tag`
- Pagination: support pagination the returned query. `limit` is the number of items per page, `page` is the page number of the result. e.g., `/?page=3&limit=10` will return the page number 3 with 10 items per page.
  The default value is `limit=20` and `page=1`
- Sort: support sorted by the specified fields, use `-` for decreasing order. e.g., `/?sort=-price,createAt` will sort the result in decreasing `price` order then increasing `createAt order` . By default, the result is order by `createAt` in decreasing order.
- Support combination of search, filters, sort, pagination
  e.g. `/?[text][search]=tech&price[lt]=300&sort=-createdAt&limit=10&page=2`

AUTHORIZATION Bearer Token

Token	{{jwt}}

PARAMS

[text][search]	tech
price[gt]	300
sort	-createdAt
limit	10
page	1

Update Listing

PATCH **Update A Listing**

{{URL}}api/v1/listing/6261a3ca79cd8f30a846e672

`api/v1/listing/id`

when creating a request to update a listing, it's imperative that the specific listing `:id` be passed as a parameter.

On the server side, the `user_id` is compared to the `belongTo` varible to verify user ownership. If it fails an error is thrown.

Once verified, simply pass whatever variables are being changed (ie, `price, title, tags`). The database will only update the variables that need to be changed, the rest will be the same.

AUTHORIZATION Bearer Token

Token	{{jwt}}

PARAMS

_id	The specific listing ID of the document to be changed.

BODY raw

```
{
    "price": 30000
}
```

Delete Listing

DEL **Delete Listing**

```
{{URL}}api/v1/listing/6261a3ca79cd8f30a846e672
```

`api/v1/listing/id`

when creating a request to delete a listing, it's imperative that the specific listing `:id` be passed as a parameter.

On the server side, the `user_id` is compared to the `belongTo` varible to verify user ownership. If it fails an error is thrown.

Otherwise, the listing is removed from the server.

Front End Overview

Login Page

The login in page takes two user inputs. The email and the password.

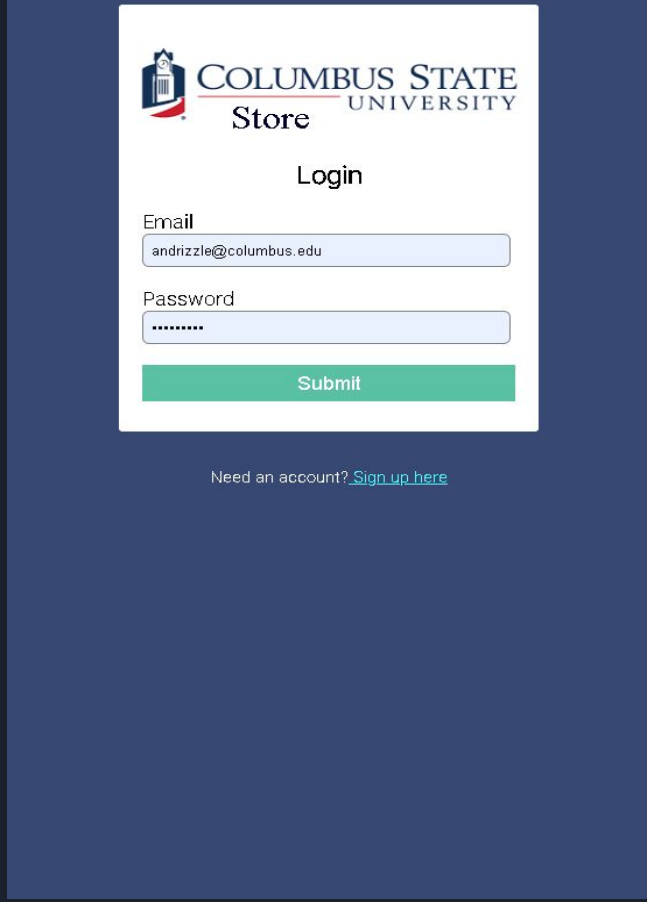
The data is then taken when the submit button is pressed by using an event listener connected to the submit button.

An axios POST request is then used to send the info to the database.
If an error occurs where error the user info does match the info within the database then an error will display saying that the user is submitting an invalid email or password.

If the users info matches with the user info in the database then the login was a success and the user will be rerouted to the homepage.

```
const login = async (email, password) => {
  try {
    const result = await axios({
      method: "POST",
      url: "http://127.0.0.1:3000/api/v1/user/login",
      data: {
        email,
        password,
      },
    });

    if (result.data.status === "success") {
      window.setTimeout(() => {
        location.assign("/homepage");
      }, 1500);
    }
  } catch (err) {
    console.log(err.response);
    alert("Email or password is invalid. Please try again")
  }
};

document.querySelector(".form").addEventListener("submit", (e) => {
  e.preventDefault();
  const email = document.getElementById("email").value;
  const password = document.getElementById("password").value;
  login(email, password);
});
```

Login Post request and event listener



Signup Page

The signup page accepts 4 elements: The name, email, password, and confirm password.

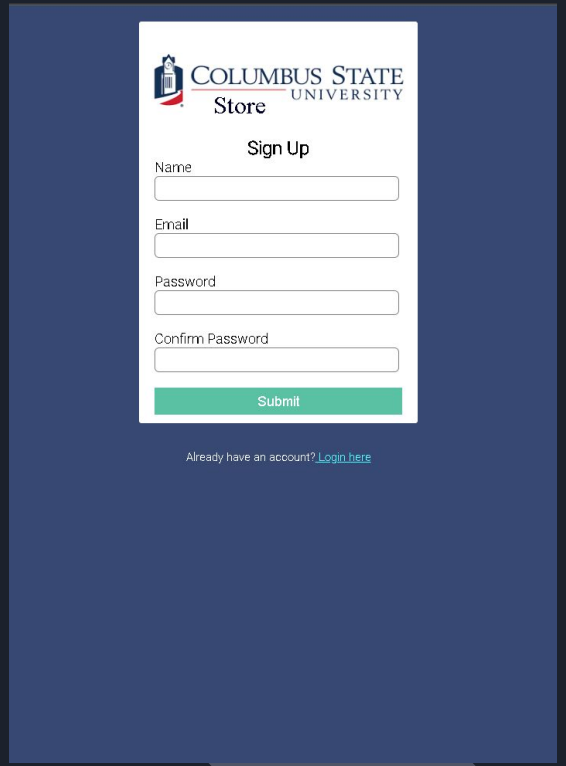
The signup uses an event listener to take the user input data once the submit button has been pressed by the user.

If an error occurs with the user info when it is checked by the backend an alert will appear that ask the user to check their info to make sure its correct.

Possible errors in user input include:

- Password mismatch
- Email not including @columbus.edu
- Email not having an @ symbol

If the user successfully logins then an alert will appear notifying the user that their sign up was successful and the user will then be rerouted to the homepage



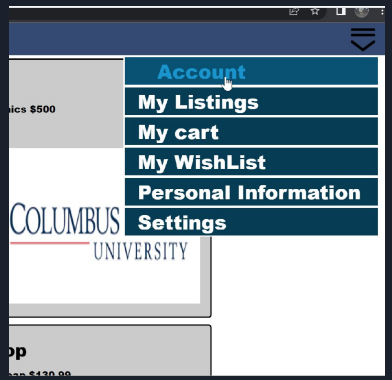

```
const signup = async (name,email,password,passwordConfirm) => {
  try {
    const result = await axios({
      method: "POST",
      url: "http://127.0.0.1:3000/api/v1/user/signup",
      data: {
        name,
        email,
        password,
        passwordConfirm,
      },
    });

    if (result.data.status === "success") {
      alert("Successful signup");
      window.setTimeout(() => {
        location.assign("/login");
      }, 1500);
    }
  } catch (err) {
    console.log(err.response);
    alert("One of the fields was invalid. Please check both passwords are the same and email is correct.");
    }
};
```

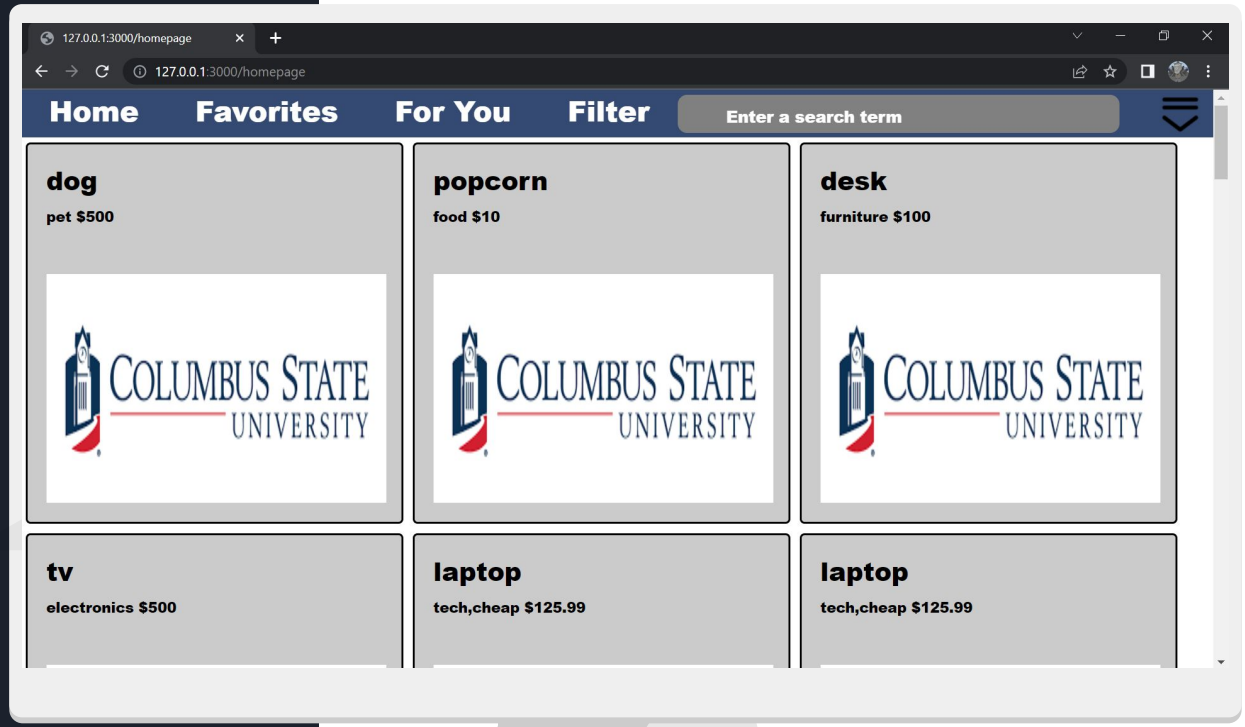
Axios POST request to send the data to the MongoDB database

```
document.querySelector(".signup").addEventListener("submit", (e) => {
  e.preventDefault();
  const name = document.getElementById("name").value;
  const email = document.getElementById("email").value;
  const password = document.getElementById("password").value;
  const passwordConfirm = document.getElementById("passwordConfirm").value;
  signup(name,email,password,passwordConfirm);
});
```

Event Listener to get input values

Homepage





The homepage is an interactive listing site with an interactive navigation bar, and dynamic scrolling list page. It was designed primarily with html and CSS.

Homepage (cont.)

```
doctype html
.Header_box
  a(href='#') Home
  a(href='#') Favorites
  a(href='#') For You
  a(href='#') Filter
  input.search-box(type='text' placeholder='  Enter a search term')
  .Account-Menu
    img(src='./img/Dropdown.png')
    ol.Drop-Button
      li
       a(href="/profile") Account
      li My Listings
      li My cart
      li My WishList
      li Personal Information
      li Settings
.Listing-Zone
  each listing in listings
    button(type='button')
      h1 #{listing.title}
      p #{listing.tag} $#{listing.price}
      img(src='./img/CSU_Logo.png' alt='Image Place')

link(rel='stylesheet' href='./css/homestyle.css')
```

To access the listings the html document retrieves the title, tags, and price from the listing database.

Profile Page

The profile page contains the users info, their current listings, and their current wishlist items.

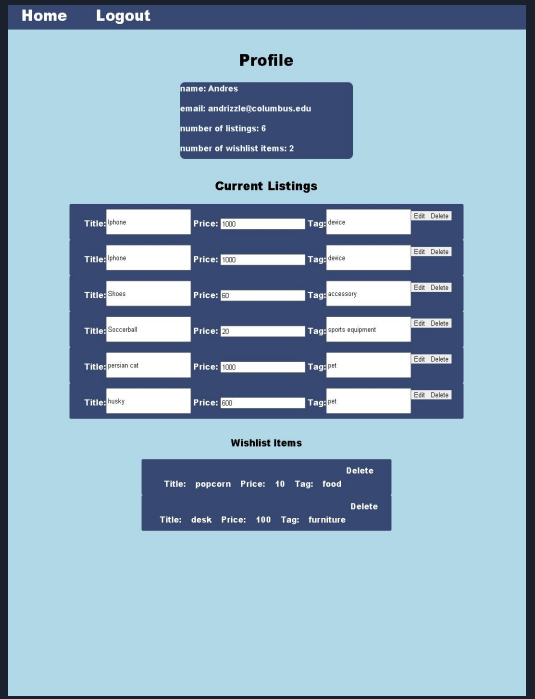
Listings and wishlist items can be deleted on this page by using an axios delete request. The listings can be edited using a PATCH request.

Pages are automatically reloaded after making these requests so that the user can see their changes immediately after editing or deleting an item.

Each request will show a alert once the action has occurred notifying the user that their action has occurred.

```
script.
    const deleteListing = async(id) => {
            axios.delete(`http://127.0.0.1:3000/api/v1/listing/${id}`).then(response => {
            console.log(response);
            window.location.reload();
            alert("Listing deleted")
        })
    }
script.
    const editListing = async(id) => {
        var title = document.getElementById(`title.${id}`).value;
        var price = document.getElementById(`price.${id}`).value;
        var tag = document.getElementById(`tag.${id}`).value;
        axios.patch(`http://127.0.0.1:3000/api/v1/listing/${id}`,{ "title" : title, "price" : price, "tag" : tag}).then(response => {
            console.log(response);
            window.location.reload();
            alert("Listing updated");
        })
    }
script.
    const deleteWishlist = async(id) => {
        axios.delete(`http://127.0.0.1:3000/api/v1/wishlist/${id}`).then(response => {
            console.log(response);
            window.location.reload();
            alert("wishlist item deleted")
        })

    }
```

Delete listing, edit listing, and delete wishlist request using axios two DELETE requests and one PATCH requests



Testing Framework

Chai is a BDD/TDD assertion library for node.js and easily works well with other testing frameworks for java code. The BDD styles of “should” and “expect” provide expressive language, and the TDD style of “assert” keeps things simple but still effective.

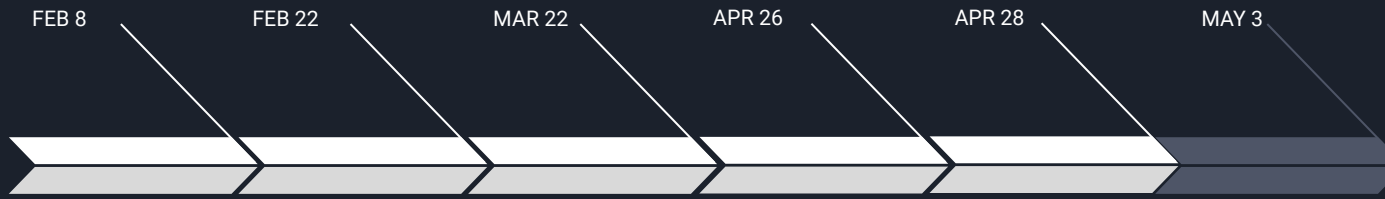
Mocha is another testing framework used within node.js, and it’s truly effective at making asynchronous testing easy and simple. It will run each test in a sequence so that everything is accurate and flexible. Additionally, mocha will state the invalid parts of tests so they could be corrected at each specific area.

```
"dependencies": {
        "axios": "^0.26.1",
        "bcryptjs": "^2.4.3",
        "chai": "^4.3.6",
        "cookie-parser": "^1.4.6",
        "cors": "^2.8.5",
        "dotenv": "^16.0.0",
        "ejs": "^3.1.6",
        "express": "^4.17.3",
        "git": "^0.1.5",
        "helmet": "^3.23.3",
        "jest": "^27.5.1",
        "jsonwebtoken": "^8.5.1",
        "mocha": "^9.2.2",
        "mongoose": "^5.13.14",
        "node": "^17.7.2",
        "pug": "^3.0.2",
        "react": "^18.0.0",
        "react-dom": "^18.0.0",
        "requirejs": "^2.3.6"
    },
```

```
"scripts": {
        "start": "nodemonserver.js",
        "test": "mocha"
```

```
npm test

> csc4330projectgrouph@1.0.0 test
> mocha
  Login
    ✔ login should allow you to fill in your created
email and password
    ✔ login should return type string
    ✔ login should notify the user when email and/or
password are wrong
    ✔ The button at the bottom of the login page
should bring the user to the signup
  Signup
    ✔ Signup should allow you to fill in name, email,
password, passwordConfimed
    ✔ Signup should return type string
    ✔ Signup should have be a student email
    ✔ Password should have 8 characters at most in
order to be accepted
    ✔ Confirmed password should completely relate to
password
    ✔ The button at the bottom of the signup page
should bring the user to the login

  10 passing (31ms)
```

Project Timeline



Introduction

Project information revealed and explained. Groups are formed

Sprint 1

User stories, system architecture, and project backlog were created.

Sprint 2

Use case diagrams, skeleton codes, database schema, design documents, and feature descriptions were created.

Sprint 3

Further progress on the database schema, design documents, site visuals and links, user model, the api, test cases, and coding the backend for listings.

Final

Testing framework and code, api documentation, and the frontend and backend coding were completed.

Presentation

Thank you!