

E-Commerce Project User Guide

Team: *Claudia Chen, Thomas Le, Garland Yee, Alex Hong*

Project mentors: *Travis Heppe from Google, Naji Dmeiri from Google, Professor Xin Liu, TA Ethan Wang*

Table of Contents/Glossary

Preface	4
Introduction	4
Motivation	4
Project Overview	4
Frontend	4
Backend	5
Hosting	5
Functionalities and Features	5
Standard E-commerce Website Features	5
Secure Payment Transactions through Paypal	5
Admin Account System	5
Secure Data Communication between Server and Browser	5
Website Uptime Prober	6
Installation	6
TroubleShooting	7
Error Messages in General	7
Localhost Not Making Requests Properly	7
Paypal	7
CSS	8
Mocking Backend API Requests Locally through Postman	8
Frequently Asked Questions	8
Contact Information	8
Taking Over the Project	9
Backend	9
Firestore Database	9
Giving yourself Access to the Database	10
Users	10

User Document Properties	10
Cart subcollection	11
Admins	11
Vendors	12
Vendor Document Properties	13
adminCode Subcollection	13
admins Subcollection	13
Products	13
Products Properties	14
Orders	14
Orders Document Properties	15
Routes	16
app	16
signup	16
POST "/"	16
GET "/confirmEmail"	16
login	17
POST "/"	17
POST "/gmail"	17
logout	17
resetPass	17
POST "/"	17
GET "/checkToken"	17
POST "/updatePass"	17
getAllProducts	17
getProductInfo	18
getUserCart	18
getVendorInfo	18
getVendorProducts	18
adminUser	18
adminProducts	18
adminVendor	18
Cookies and JWTs	18
Creating a New Vendor	19
Image Storage	19
Website Prober	19
Database Backups	21
Frontend	21

Redux	22
Components List Below	24
About	24
AboutClub	25
AddProduct	25
ButtonAppBar	26
Cart	27
CartItem	28
CartView	29
Checkout	29
Clubs	30
EditClubInfo	30
EditItem	31
EditItemView	32
EmailConfirmation	35
FAQ	35
Footer	35
GenericPage	36
Home	36
InputRecoveryPassword	36
Login	36
OrderHistory	36
OrderHistoryAdmin	37
OrderHistoryItem	37
PaypalButton	38
Privacy	38
RecoverPassword	38
ScrollToTop	39
ShopView	39
ShopItem	39
ShopItemDetailed	39
Signup	39
Terms and Services	40
VendorSignup	40
VendorView	40
Frontend Styling	41
Logo	41
Appendix	42

Preface

This is a user guide meant for whoever takes over the project and wishes to make updates to it. There are summaries of features and functionalities of this project, as well as more detailed information about how the technologies and systems of our application work, for future developers wishing to make larger changes to the project. Please consult the glossary/table of contents if one wishes to only see specific information.

Introduction

We are doing an ecommerce website for clubs at UC Davis. The site aims to help facilitate the buying and selling of club merchandise. It will provide an easy way for clubs to display their products, as well as obtain secure online payments. This project utilizes modern web technologies for both learning and performance.

Motivation

UC Davis clubs typically have an unofficial way to sell their merchandise, usually through Facebook and word of mouth. There is no easy way to display, buy and sell goods. Our site is to provide a method to streamline this process. Clubs can post their items onto our site, customers purchase what they like, and the appropriate profits will go towards the participating clubs. Those interested in becoming vendors can apply on our site also.

Project Overview

Below is a brief overview and description of our website and the technologies used. The website domain is: <http://193ecommerce.com>. More detailed descriptions of the technologies can be found in the “Taking Over the Project” section and its subsections.

Frontend

The frontend is the client-side of our website that user can interact with directly. We use the React framework for the frontend, as well as standard HTML, CSS, and Javascript.

Backend

In this project, when we refer to the “backend”, we are referring to our NodeJS/ExpressJS (both javascript frameworks) server that handles more code heavy tasks. We use Firebase as our database for storage of data.

Hosting

We use google app engine to host our website, google domain for our domain name, and AWS for our prober.

Functionalities and Features

Standard E-commerce Website Features

The website contains standard e-commerce features such as:

- Account system for users with signup and login either through our website or through google OAuth
- Dynamic email system for account confirmation, password reset, and order confirmation
- Pages for club information
- Shop pages for items and apparel
- Cart system

Secure Payment Transactions through Paypal

Payment transactions are handled securely through Paypal. Paypal’s API allows the user to pay either through Paypal’s money balance or through credit card.

Admin Account System

Clubs or vendors who wish to sell their items on our website can make an admin account that allows them to easily update their information and their product information such as amount of stock, product price, product pickup location and time, product pictures etc.

Secure Data Communication between Server and Browser

The website implements HTTPS to ensure that when sensitive information is communicated over an HTTPS connection, that no one can eavesdrop on the information while it is in transit.

Likewise, the website's server makes use of RESTful API calls to send and store information on the database. To ensure that these API calls can only be made by authenticated users, the project implements HTTPOnly cookies that store JSON Web Tokens (JWTs) that store the authenticated user info needed to access certain parts of the website. Please see "Cookies and JWTs" section for more detailed information.

Website Uptime Prober

There is a prober on an AWS server that checks the site for any potential outages by querying every minute to see if the website is up. If the website is down a text message will be sent to a phone number indicating when the site is down. The prober also has its own website that displays a live status of the website for an hour window. Below will be two graphs displaying the uptime and downtime of the website in a week timeframe. In addition this prober also does backups of the firestore database which stores the data for the website.

Installation

The website may be accessed through <http://193ecommerce.com>.

The below steps are for future developers who wish to take over the project and make updates to the project.

1. Download the code from the repo <https://github.com/clauidiaschen/ecommerce>
2. Make sure to have NodeJS installed. NodeJS also installs NPM, which is a popular library package manager.
3. Once downloaded, within the main "ecommerce" root folder, run "sudo npm install" to install the libraries used for the project.
 - a. Then, please change directory into the "frontend" folder and again run "sudo npm install" to install the libraries used for the frontend version of the website.
4. Then, the user has to create ".env" files in two locations. These ".env" files contain sensitive information that is gitignored for security reasons. Please email one of our project members for the information, if we have not given it to you already, since simply listing the info in plain text in a google doc is unsafe. This comes from experience.
 - a. First, create a ".env" file in the root "ecommerce" folder
 - b. Then, another ".env" file should be made inside of the "frontend" folder.
5. Next, the user needs make "config" folders in the following folders. Again, please email one of our members for the information, if you have not already received this info.
 - a. Make a config folder in the root ecommerce folder.
 - b. Make a config folder in "frontend" folder as well.

6. Once done installing, one can run a local version of our website for testing by running “npm run dev”. This script starts up the backend server in the root folder, as well as the running the frontend of our website in the frontend folder.
7. If future developers wish to have admin access rights to the database and server, please contact one of the members so that we made give your email and account access rights. This task cannot be automated and one of our members must do it manually.

TroubleShooting

Error Messages in General

Code on the backend server generally have “catch”es to catch any errors. Those errors are generally console logged. Future developers may see what those errors are locally through their terminal. If future developers wish to see those errors on the live site, they should go to the app engine

Localhost Not Making Requests Properly

If you are running this locally and the local version is not making requests properly, you may have missed a step in the installation process. Please see the Installation section of this guide, and especially make note not to forget to run “npm install” and remember to include appropriate data within the “.env” files.

Paypal

If you are attempting to make changes with how the website displays the Paypal checkout button then it might because there is a problem with how the 3rd party Paypal API scripts are loaded with React. It has to do with attempting to change unmounting the Paypal button and mounting a different component after it in React.

We have already ensured that the current working version of the site deals with that problem by passing a value into the PayPalButton component to un-display the button before switching to a different component. This section was added in case future developers run into this problem when trying to make changes to how the Checkout process works.

CSS

We have made separate CSS files that correspond to the different components. This was done to make it easier on the developer to know what CSS is being interacted with for which component. However, future developers should note that when these CSS files are loaded into the application, any and all CSS ID's, classes, variables etc. are global, That is, for example, if you have a class name called "header" in "Home.CSS", inside of the "Home" component folder, any other component can use that "header" class name, so future developers should be careful when naming CSS classes in case of overlap.

Mocking Backend API Requests Locally through Postman

Postman is a useful application for mocking and testing API requests. If future developers would like to tests requests through Postman, please first go into the root folder's "config" folder and update "config.json" (see Installation section for more info). Then, set the "secure" variable to false for testing. This is originally set to true, because we want the server should send cookies to the frontend and a user's browser securely. If testing locally, Postman does not make secure requests and thus all requests are denied by the server. Setting "secure" to false fixes this problem. Don't forget to set "secure" back to true, before pushing any new changes onto the live site.

Frequently Asked Questions

Here is a link to the website's FAQ: <https://193ecommerce.com/faq>

Contact Information

- Thomas Le
 - Database, Backend, and Partial Frontend Developer
 - Email: thele@ucdavis.edu
- Claudia Chen
 - Point of Contact, Frontend Styling Designer and Developer
 - Email: cschen@ucdavis.edu
- Garland Yee
 - Server Manager, Prober Developer
 - Email: gnyee@ucdavis.edu
- Alex Hong

- Main Frontend Developer
- Email: ahong@ucdavis.edu

Taking Over the Project

Below is a more detailed explanation of components and systems of our website. The below is intended to guide understanding for any programmer wishing to take over this project. Please see the “Installation” section of this project first.

Backend

In this project, when we refer to the “backend”, we are referring to our NodeJS/ExpressJS server that handles more code heavy tasks. The frontend makes REST API calls to the backend to send and receive data to our database.

Firestore Database

The backend makes use of firebase for its database. Specifically, we use the newer Cloud Firestore or Firebase’s realtime database. Please see Cloud Firestore documentation on how to perform queries, add, update, and delete data. The below explanation will be more detailed and is assumed that the reader has a knowledge of how Firestore works.

The database schema was made following advice online on how to structure a NoSQL database to improve querying speed. The design of the database will be explained below. We use terms from Firestore, reproduced below for convenience:

“Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in documents, which are organized into collections.

Each document contains a set of key-value pairs. Cloud Firestore is optimized for storing large collections of small documents.

All documents must be stored in collections. Documents can contain subcollections and nested objects, both of which can include primitive fields like strings or complex objects like lists.

Collections and documents are created implicitly in Cloud Firestore. Simply assign data to a document within a collection. If either the collection or document does not exist, Cloud Firestore creates it.”

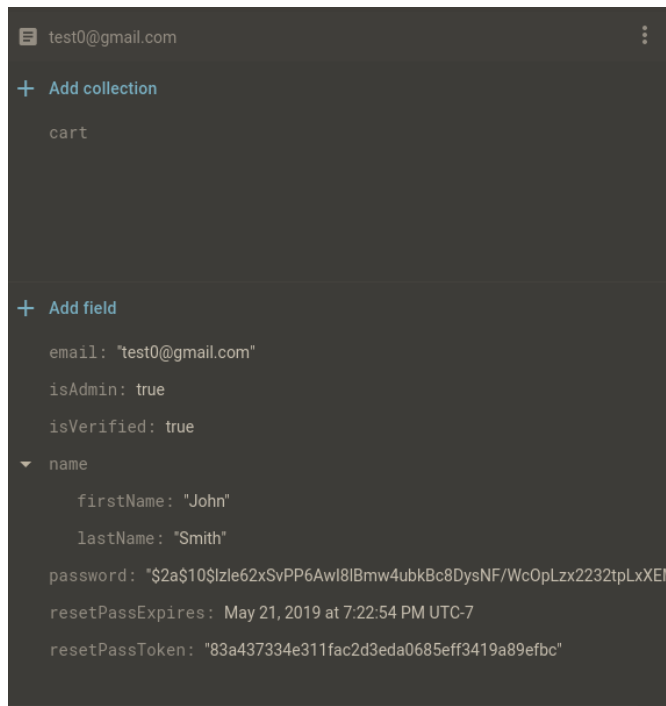
<https://firebase.google.com/docs/firestore/data-model>

Giving yourself Access to the Database

If you want access rights to our database, please email one of us and we can add your email to the database.

Users

The users collection holds user information. Each document holds document for an individual user. A user's unique document ID is their email.



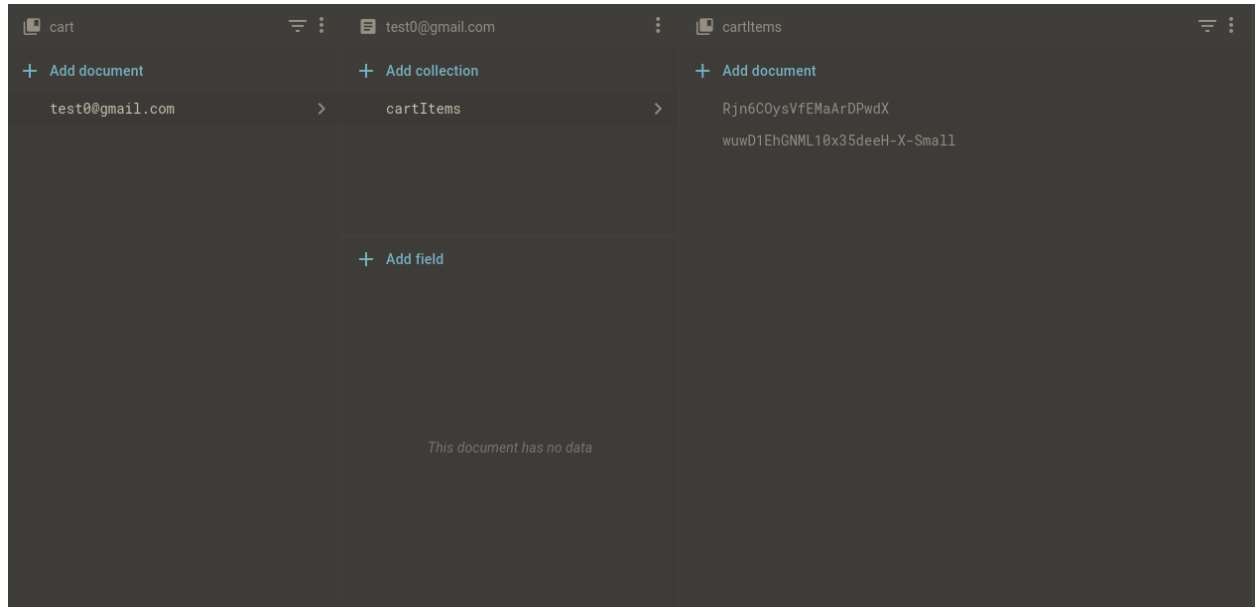
User Document Properties

- **email:** user's email; also the document's unique ID
- **isAdmin:** boolean for whether user is an admin or not
- **isVerified:** boolean for whether a user has verified their account or not
- **name:** user's first and last name
- **password:** user's hashed password
- **resetPassExpires:** timestamp for when a user's reset password token expires
- **resetPassToken:** a unique token for the server to identify which email reset email belongs to which user

Cart subcollection

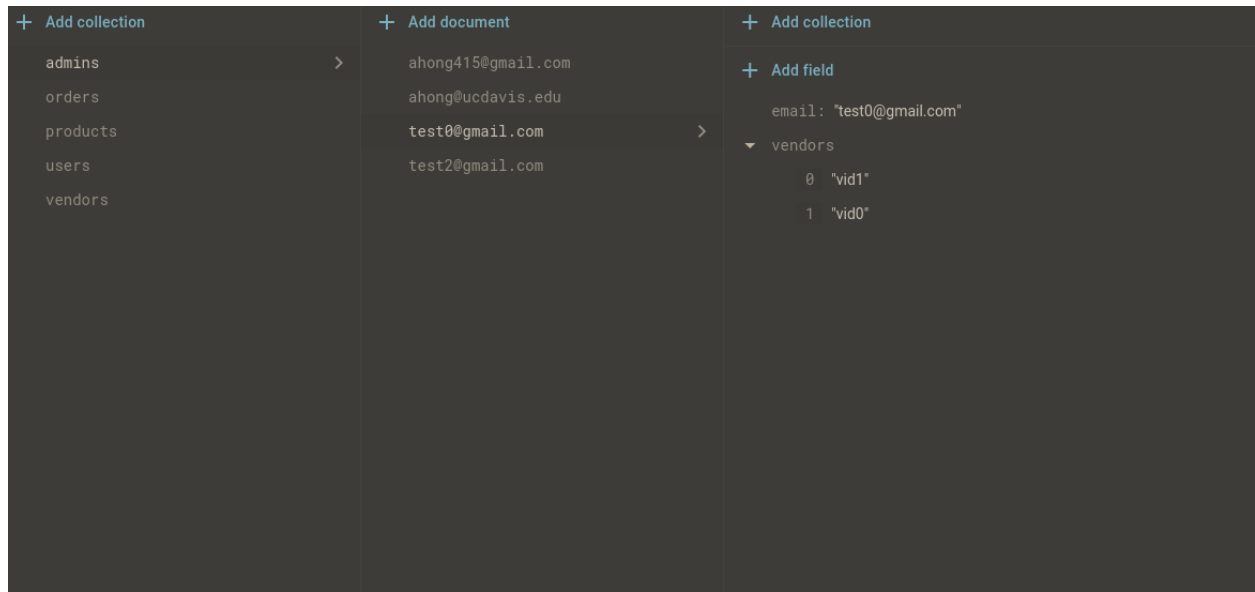
Each user has a cart subcollection, with a single document ID'd by the user's email, with a subcollection called cartItems, whose documents are the products currently in the user's cart.

The cartItems are a subcollection because the DB was originally created with other higher order cart info in mind. However, as the quarter progressed, that was no longer needed.



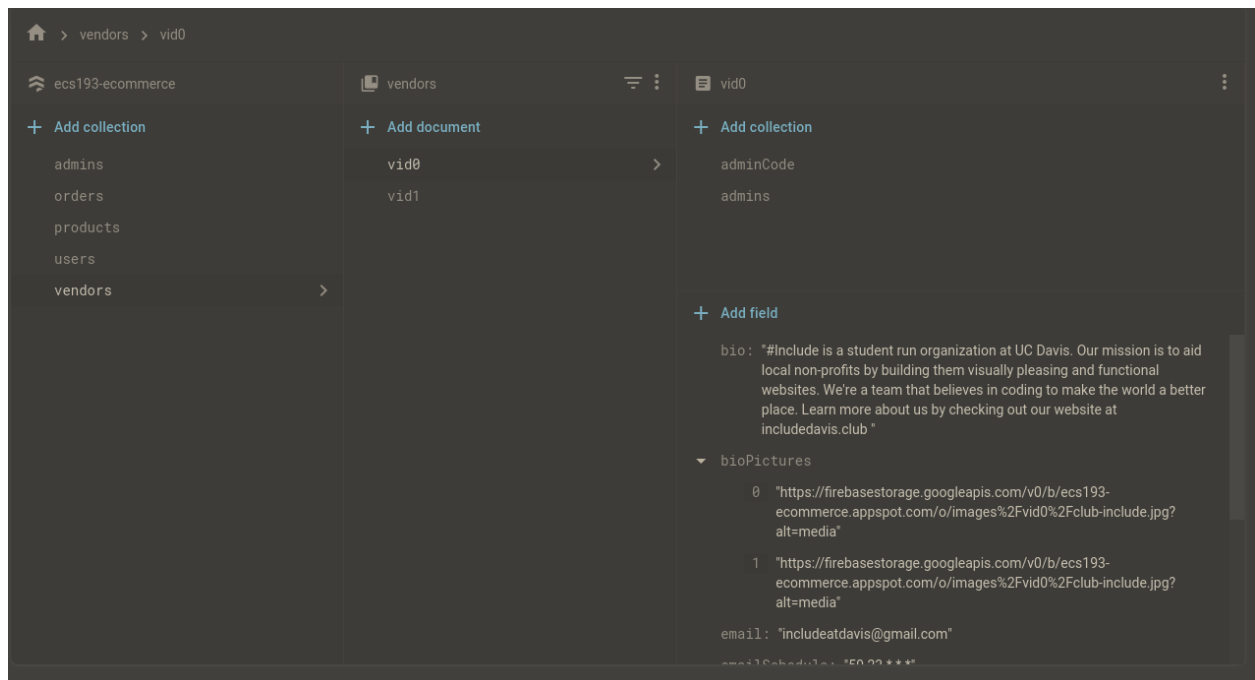
Admins

This collection is a list of users who are admins, with an array of vendors who they are admins of. The admin collection with its info was made separate from the user's info so that when Firestore does a query on user info, it also doesn't grab admin info, which is sensitive information.



Vendors

The vendors collection contains information about all vendors/clubs.



Vendor Document Properties

- **vid** - a vendor's ID; it is the unique identifier for a vendor
- **bio** - a vendor's bio displayed in their about information page
- **bioPictures** - the images displayed in the vendor's about pages
 - bioPicture[0] - main image in their about page
 - bioPicture[1] - box image found in clubs listing page
- **email** - main contact email for a vendor, displayed in their about page; also the email that gets notified periodically about new purchases
- **emailSchedule** - the schedule for often a club is to be emailed about new purchases; the format is used for node-schedule library here:
<https://www.npmjs.com/package/node-schedule>
- **lastUpdate**: last time a vendor's about information has been updated
- **lastUpdateUser**: admin user who last made an update
- **pickupInfo**: information on where and when to pickup a vendor's products; this information will be displayed in a vendor's about page and in any order confirmation emails for their products
- **vendorName** - name of the vendor

adminCode Subcollection

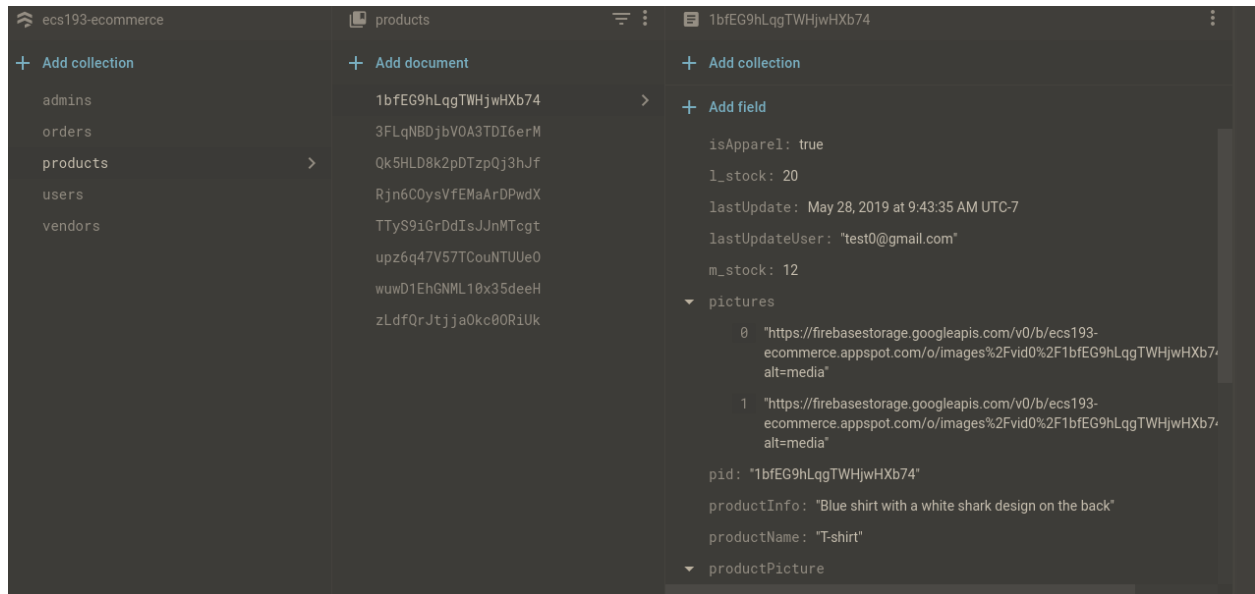
This subcollection has only one document, also called "adminCode". The document has a field called "adminCode" which is the code needed to upgrade a user to an admin for this particular vendor. Please see frontend sub-section called "VendorSignup" for more information.

admins Subcollection

This subcollection contains all users who are admins for this vendor.

Products

The products collection contains documents and information for each product in the shop.

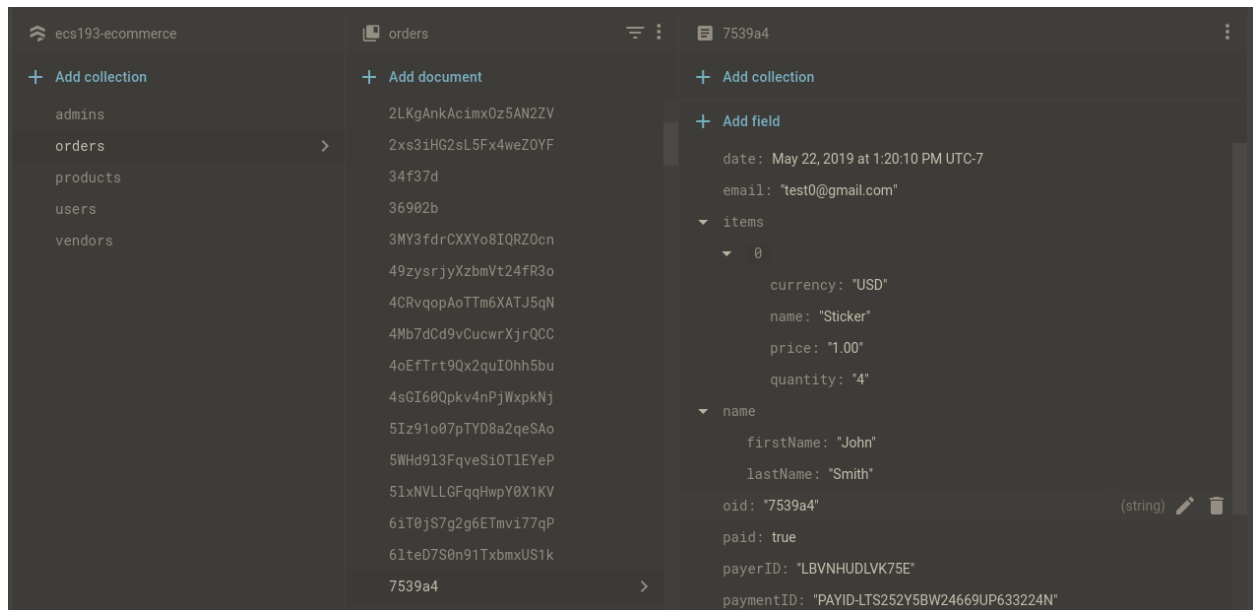


Products Properties

- **pid** - the product ID is the unique identifier to each product.
- **isApparel** - products' can be regular items or apparel items. Apparel items have stock for different sizes, so this property is needed to differentiate between the product type
- **lastUpdate**: last time a product's information has been updated
- **lastUpdateUser**: admin user who last made an update
- **productInfo** - a product's information that is displayed on its item page.
- **productPicture** - Array of pictures that are displayed on a product's item page. The first picture in the array is the main image displayed.
- **productPrice** - the product's price in USD
- **purchasedStock** - number of items that have been purchased
- **stock** - The amount of stock a product has left.
 - If the product is an item, then it just has this stock property.
 - If the product is an apparel item, then it also has stock amounts for different sizes for the apparel.
- **xs_stock, s_stock, m_stock, l_stock, xl_stock** - stocks for differing sizes of apparel items

Orders

The orders collection holds documents containing information about individual orders. Note in the below image, newer orders have shorter order ID's to make it easier for user and vendor to find orders.



Orders Document Properties

- **oid** - Unique order ID.
- **date** - The date the order was made.
- **email** - The email of the user who made the order
- **items** - array of items in the order; this is the specific info saved in a Paypal order
 - **currency** - string of currency type, typically USD
 - **name** - name of the item
 - **price** - price of an individual item
 - **quantity** - quantity of that item purchased
- **name** - name of the user
- **paid** - boolean for whether the order has been paid for yet. Currently, if purchasing through Paypal, then all orders are paid for. Made for future work if users want to pay by cash in person.
- **payerID** - paypal account payer ID; not used but saved just in case
- **paymentID** - paypal transaction payment ID; not used by saved just in case
- **pickedUp** - boolean signifying if order has been picked up yet.
- **seenByVendor** - boolean signifying if the server has counted the order as a new order for emailing the vendor about new orders
- **totalPrice** - total summed price for all items in the order
- **vid** - id of vendor for items in the order

Routes

Routes describe different routes on the server e.g. /signup, /login, /orders etc. The frontend of the app sends a RESTful API call to different routes on the backend, which do different things. The list of routes used can be found in the app.js. The below sections summarize what the below routes and sub-routes do.

Below will provide brief descriptions of what the routes do. Please view the code and the comments in the code for more detail.

app

The app.js is where the main server code is setup. This file also contains the setup for the below routes. Please see the code for the actual routes. For example, the signup route would be "/api/signup".

NOTE: for example the signup section and its sub-routes e.g. signup section has route "api/signup/" and "api/signup/confirmEmail"

Also, the app.js file contains setup for the global node-schedule library, which is the library used to make the server periodically do a certain task. Please see the code and the comments for more information. NOTE: for example the signup section and its sub-routes e.g. signup section has route "/signup/" and "/signup/confirmEmail"

signup

POST "/"

The "/" route (i.e. /signup/) is the route that makes a user document in the database. The route sends an email to the given email with a confirmation link. The server will also apply a hash to the user's password, and save that, alongside all the other info, into the DB

GET "/confirmEmail"

Each email confirmation link is associated with a token and a token expiration time. The user has one hour to confirm the account before the link expires. The link uses the randomly generated token to figure out which email is being verified.

When the user clicks the confirmation link on, a call is made to this route. The route gets the token, finds the user in the database with said token, and then verifies that account if the token has not yet expired.

login

POST “/”

Route used to login a user. When a user logs in, the server will attach a cookie containing a json web token (JWT) to the response sent back to the frontend. The JWT holds user information. Once the cookie with the JWT is sent back to the frontend, the frontend can then attach that cookie alongside all requests made to the server backend.

This process is done to protect our server routes for security reasons.

POST “/gmail”

Route used to signup or login in a user through OAuth gmail. Also attaches a cookie to the response back to frontend.

logout

Route used to clear out a user’s cookie when they logout.

resetPass

Routes used to reset a user’s password.

POST “/”

This route is called when the user requests a password reset link. The route generates a token with an expiration, and sends a password reset link to the requested email.

GET “/checkToken”

This route receives and checks the token, and sends back to the frontend, the email associated with the password.

POST “/updatePass”

This route updates the given email with the given new password. It also deletes the associated reset pass token, so that token can no longer be used.

getAllProducts

Route used to get all product info to display on the Shop page on frontend.

getProductInfo

Route used to get the more detailed info of a product.

getUserCart

Routes used to get a user's cart, as well as to add and remove items to their cart.

getVendorInfo

Routes used to get information about a vendor/club , such as bio, vendorName, vid, etc.

getVendorProducts

Route used to get a given vendor's products.

adminUser

Routes to:

- Get clubs the user is an admin of
- Check if a user is an admin
- Upgrade a user account to an admin account for a club

adminProducts

Routes to:

- Add new products
- Get product information
- Edit existing product information

adminVendor

Routes to:

- Get a vendor's information
- Edit a vendor's information

Cookies and JWTs

Because our website uses different routes on the server for RESTful API calls, technically, anybody with the route can use the API. That is why we implemented cookies and json web tokens (JWTs) for security purposes.

A JWT is a token that has information and is associated with an authenticated user. Protected routes on the server will check for, and decode tokens, before allowing the user to access certain parts of the website, which make certain API calls.

To make sure the JWT themselves are secure, they are attached inside of HTTPOnly cookies. These cookies, with these tokens, are sent to the frontend when the user logs in. Following that, any requests that user makes will send back the cookie with the token that gets decoded. HTTPOnly cookies are safer in that these cookies cannot be accessed by client-side code. This protects the website against malicious XSS (cross-site scripting) attacks.

Creating a New Vendor

If any future developers want to add a new vendor or club, they will need to do the following steps.

1. Create a new document for the vendor in the “vendors” collection
2. Set up basic database information for them such as email, bio, bioPictures, emailSchedule, etc. Admins for that vendor can update that information later.
3. Have the vendor create an account on the website.
4. Create a secret admin code in the database for the vendor and send that to them, and get them to go to the following route to upgrade their user account into an admin account: see the “VendorSignup” section for more information. Please also see the above section on the “vendors” collection for more information on adminCode.
5. Then, the admin can use that account to make updates to their information, and add and edit their products.

Image Storage

Images are stored through Firebase storage. The AddProduct and EditProduct components on the frontend and AddProduct.js in the backend routes deal with images. Admins can upload images through the AddProduct and EditProduct frontend components, which save the images in the firebase storage, which is connected to the database. The backend routes will then find save links to those images into the DB so that they can later be rendered to the frontend.

Website Prober

To set up the server one must first create an AWS account and create an EC2 instance which can be done by following the instructions given when first creating an instance.

In order to ensure that one can access the webpage it is important to change the settings in the Security Group for the instance. To do so go into the EC2 tab of AWS and scroll down the left sidebar and locate the "Security Groups" section. Once there create a Security Group and make sure for "Outbound" all traffic is allowed for all port ranges and protocols with destination 0.0.0.0/0. For the "Inbound" tab, make sure to add in a SSH rule with TCP on port range 22 in order to be able to ssh into the server and access the code as needed.

In addition, one must also allow HTTP TCP at port range 80 as well as a custom TCp at port range 3000 (or any port range) as this is how one will be able to access the web page for the prober. To access the web page using a browser one would copy and paste the Public DNS address listed in the EC2 section, it should follow a format to something like ec2.compute.amazonaws.com. To connect to the website one would copy this address and add a :3000 at the end of it so it would look like ec2.compute.amazonaws.com:3000. Keep in mind that the address for the website will change as it is unless one creates an Elastic IP where the address will be set, this can also be found on the left sidebar.

To be able to ssh into the server to edit/change code a key pair is needed which should be generated by default. If not one can create one by scrolling to the left side bar and finding the "key pair" section. From there a key pair will be generated that can be used to ssh into the server.

This prober uses AWS SNS which must be set up and can be found by searching for SNS in AWS. From there one must create a Topic. When creating a Topic it is important to create a JSON for the Access Policy. An example of the access policy is as follows:

```
"Action": [
    "SNS:GetTopicAttributes",
    "SNS:SetTopicAttributes",
    "SNS:AddPermission",
    "SNS:RemovePermission",
    "SNS:DeleteTopic",
    "SNS:Subscribe",
    "SNS:ListSubscriptionsByTopic",
    "SNS:Publish",
    "SNS:Receive"
],
```

Once created one must also create a Subscription that is connected to this Topic. It is important that for the Topic and Subscription an Endpoint (which is one's phone number) is inputted as a way of connection.

The server code is broken up into two sections, the frontend which consists of the index.html and css files and the backend which is the app.js file. Inside of the app.js file, one would have to change the variables "AWS.config.region", "AWS.config.accessKeyId", and "AWS.config.secretAccessKey" to match one's EC2 instance.

The multiple “params” variables consist of the messages and phone numbers that the AWS SNS will send a text message to. Nodecron is used to schedule the frequency of the get request to check for site outages which can be changed.

To run the prober one must first download all of the necessary npm packages using npm install package.json. The library pm2 is used to continuously run the prober service. To use type in the command: pm2 start app.js in the console (Note: this app.js is not the one found in the my-app folder but the one outside of it). The prober will then run even when one does not have an ssh connection to the server. To stop running the prober type in the command: pm2 stop app.js.

Database Backups

Firestore backups are run using a program called "FirestoreRestore" which is located in the server directory. To utilize the firestore backup feature, one must edit the variable "cmdString" and place inside of the server folder their key for their firestore access which should take the form of a json. One must also have a bucket created on GCloud that the program will store the backup data to. To create a bucket click on the Storage tab located on the Google Cloud Platform's left sidebar and from there create permissions in the permissions tab.

To restore to a backup one can run the command in the cmdString in the console. Simply copy and paste the command and remove the "-wait" flag and specify the name of the backup which is the date that it was backed up.

The command is broken down into:

```
export GOOGLE_APPLICATION_CREDENTIALS="[Path to the firebase access key] &&  
./FirestoreRestore -backup -wait -p [name of the google project] -b [name of the bucket] -n"
```

One can change the node cron job to detail how often backups occur (it is currently set to backup at the end of each day at 11:59 pm)

Frontend

The frontend is the client-side of our website that the user can interact with directly. We use the React framework for the frontend. React uses components, which are essentially modularized classes that use and display different UI to the user.

React components render HTML/CSS elements and communicate with backend API endpoints to retrieve data from our database.

The below “Redux” section describes Redux, which is global state management system.

After that section, we describe the components used on the frontend in more detail.

Redux

- State management system for components to access global state values
- Required imports
 - `import { connect } from react-redux`
 - actions file containing list of actions
- Must use `connect()` function from `react-redux` for components to communicate with store.
 - Eg. `export default connect(mapStateToProps, mapDispatchToProps)(ShopItemDetailed)`
- Global store contains state values that are able to be passed as props to components to use via the function `mapStateToProps()`.
- Components are also able to update values in global store by passing values via matching actions by calling `mapDispatchToProps()`.
- Store state values are updated by calling actions, in which frontend components call appropriate actions to update state values
- Store state values are split up in groups known as reducers
- **Actions** - Frontend component call a specific action to update certain state values in the Redux store. Depending on the action called, a reducer function handles the matching action.
- **Reducer** - Functions that contain state values and modify state values based on certain actions called.
- **Persist Reducer**- Saves state values in local storage, if user refreshes/reloads page, previous state values are loaded once again
- **Login Reducer**:
 - State Values:
 - **login** - boolean to determine if user is logged in, default is false
 - **text** - string to display on navbar for user, toggles between “Login” and “Logout”, default is “Login”
 - **isAdmin** - boolean to determine if logged in user is an admin or not, default is false
 - **vendorID** - string, if the user is an admin, determine which vid user belongs to, default is “
 - **adminsOf** - array, if the user is an admin contains array of vendors user is an admin of, default is []
 - **currentVendor** - string, set to name of vendor passed in from frontend component, default is ‘ ‘
 - Actions:
 - **LOGGED_IN**
 - Sets login boolean to true and text to “Logout”
 - **ADMIN_LOGGED_IN**
 - Sets login boolean to true and text to “Logout”

- Sets isAdmin boolean to true
 - Sets vendorID to passed in vendorID from frontend component as action.vid
 - Sets adminsOf to array of vendors passed in from frontend component as action.admins
 - Sets currentVendor to vid passed in from frontend component as action.currentVendor
- **LOGGED_OUT**
 - Sets all state values to default values
- **UPDATE_VENDOR_ID**
 - Sets vendorID to vid passed in from frontend component as action.vid
 - Sets currentVendor to vendor name passed in from frontend component action.vendor
- **Get Products Reducer:**
 - State Values:
 - **products** - array, array of items to display on shop view
 - Actions:
 - **GET_PRODUCTS**
 - Sets value of products to be array of shop items passed in from frontend component as action.items
- **Cart Reducer:**
 - State Values:
 - **items**- array, array of item objects stored user's cart
 - **total**- number, total price of items summed from user's cart
 - **itemsPurchased**- number, total amount of items purchased, summation of amountPurchased for each item in user's cart
 - Actions:
 - **GET_CART**
 - Sets items array to be array of items passed in from frontend component as action.cart
 - **ADD_CART**
 - Appends item passed in from frontend component as action.item to items array
 - **REMOVE_CART**
 - Not used
 - **EMPTY_CART**
 - Sets items array to be an empty array, clear items when user logs out
 - **UPDATE_TOTAL**
 - Sets total amount to be value passed in from frontend component as action.total
 - **UPDATE_CART**

- Sets items array to be array of items passed in from frontend component as action.cart
 - **UPDATE_AMOUNT_PURCHASED**
 - Sets amountPurchased to be value passed in from frontend component as action.amountPurchased
- **Vendor Reducer:**
 - State Values:
 - vendor - string, vendor ID of selected vendor
 - vendors- array, list of vendors user is an admin
 - Actions:
 - GET_VENDOR_PRODUCTS
 - Sets vendor to be vid passed in from frontend component as action.vendor
 - GET_VENDORS
 - Sets array of vendors passed in from frontend component as action.vendors
- **Shop Item Reducer:**
 - State Values:
 - **selectedItemID** - string, string to store product ID of selected item to view
 - Actions:
 - **UPDATE_SELECTED_ITEM**
 - Set value of selectedItemID to be product ID passed in from frontend component as action.itemID

Components List Below

Here we describe each component in detail with its functionality and design choices. We feature screenshots of pages that would require more overhead (admin pages, general account pages, etc) to access, but pages easily accessible do not. Our website is accessible through <https://193ecommerce.com/>

About

This component holds general information that the user may be interested in. It explains who the website's authors are (us), our motivations, the technology stack, and our contact information. We use a parallax scrolling as an aesthetic way to make this static web page more dynamic.

AboutClub

This component provides general information about the clubs featured on the site that the user may be interested in. This information includes a picture of the club, a description of it, their social media and shop page links, and their pickup and contact information. This page is primarily for advertising and promoting clubs at UC Davis.

AddProduct

This component allows admin users to create/add a new product to their inventory. User inputs product name, product info, product price, amount of stock, whether the item is an apparel or not, and pictures of the item to upload. If the item is an apparel item, another form is displayed that includes stock for each size. After user inputs appropriate info, it makes an API request to the `/api/adminProducts/addNewProduct` to add item to the vendor's collection in our database.

Add Product

Product Name *

Product Info *

Product Price *

\$

Select Product Type

☐ Item

☐ Apparel

Product Stock *

Upload Images

Choose Files	No file chosen
Choose Files	No file chosen
Choose Files	No file chosen
Choose Files	No file chosen

ADD PRODUCT

How the add item form looks like for admin users.

ButtonAppBar

The Button App Bar (our nav bar) is an app bar API provided from React Material UI. This app bar uses an svg that is the navigation bar for all UCD websites. It was used to match UCD's visual identity and was taken from their marketing toolbox. We feature links to important components such as Shop, Clubs, About, Sign Up, and Login when not logged in. The order is from left to right the most important features of our website.

ECS193 E-Commerce

SHOP CLUBS ABOUT SIGN UP LOGIN



The Button App Bar view for non-logged in users (the default view).

For users logged in as non-admins, the app bar features the Shop, Clubs, About, Logout, and Order History links.



The Button App Bar view for logged in users without admin privileges.

Admin users' app bar will feature the club a Select Club Menu (in the case that they are an officer of more than one club) and an Admin Menu where the user can select whether to Edit Club Info (the information featured on their Club page), Edit Product Info (where users can fill out a form that edits the details of their current items), and Add Items (where users can add new items to their Shop Page). More about these menu items will be in their component section.



The Button App Bar view for logged-in admin users.

Edit Club Info



Add Items



Edit Items

The on-click view of the admin menu.

Cart

Component to store user's items purchased. Items within a Cart component are separated by club/vendorID. Each cart consists of an array of CartItem components which contain the information of a product within a user's cart. Total price of a Cart is also summed based on the summation of price for each CartItem within a Cart. Depending on the items in the cart, it separates the items by club for each PayPal checkout.


#Include Cart (1)		Price	Qty	Total
	Mug	\$12	1	\$12
	REMOVE ITEM			
			Total	\$12.00
				

WICS Cart (2)		Price	Qty	Total
	Another Shirt	\$30	3	\$90
	REMOVE ITEM			
	Plant Mug	\$6	7	\$42
	REMOVE ITEM			
			Total	\$132.00

Example of Cart components separated by vendor.

CartItem

Component to store the information of a product to be purchased. Each CartItem contains properties such as productID, vendorID, itemID, amount purchased, individual price, total price based on amount purchased, stock of item, and whether the item is an apparel or not.

#Include Cart (1)		Price	Qty	Total
	Mug	\$12	1	\$12
	REMOVE ITEM			



Example of CartItem component.



CartView

Component that stores/renders multiple Carts based on vendor/club. On rendering it separates items based on vendorID and creates a Cart component for each vendorID to store each list of separated items. The cart items are simply centered on the page as unlike other e-commerce websites that utilizes a two column grid for different forms of payment, we only have one form of payment, thus one column to complete a checkout.

ECSt193 E-Commerce

SHOPCLUBSABOUTLOGOUTORDER HISTORY



#Include Cart (1)	Price	Qty	Total
<div><div>Mug</div><div>REMOVE ITEM</div></div>	\$12	1	\$12
Total			\$12.00
<div></div>			


WICS Cart (2)	Price	Qty	Total
<div><div>Another Shirt</div><div>REMOVE ITEM</div></div>	\$30	3	\$90
<div><div>Plant Mug</div><div>REMOVE ITEM</div></div>	\$6	7	\$42

Example of CartView component showing all Carts and CartItems

Checkout

Component that contains a PayPal checkout button to allow a user to proceed with purchasing items. Each Cart component consists of a Checkout component to allow a user to make a purchase for that specific club. Payment parameters are passed from the Cart component to the Checkout component to be used with the PayPal checkout button. Payment parameters include total price of cart, list of items, name of each item, the price of each item, and amount purchased for each item. When a payment is complete, stock is removed from the server based on the amount purchased and the items are removed from the user's cart.

WICS Cart (2)		Price	Qty	Total
	Panda Sticker	\$1	1	\$1
	REMOVE ITEM			
	Tree Sweater	\$24	4	\$96
	Size: X-Small			
	REMOVE ITEM			
			Total	\$97.00



Example of Checkout component below user's Cart.

Clubs

This component features an image of each club that is a quick link to their shop page. Below is a link to their about page. When the component renders it makes an API request to `/api/getVendorInfo` to obtain list of clubs registered with our site. For each club in the list, it will render a `DisplayClub` component which features the image and link to their shop page. Since we do not have that many clubs displayed, we used a two grid layout. In the future, a different grid layout may be necessary to display more clubs in one screen view.

EditClubInfo

Component that allows admin user to edit club information. Admin user is able to edit club name, biography, item pickup information, Facebook link, Instagram link, contact email, club pictures, and email preferences.

Edit Club Info

Last Updated: 2019-05-28T17:42:46.986Z
Last Edited By: test0@gmail.com

Club Name *

WICS

Biography *

test

Item Pickup Info *

Pickup your items at Wellman Hall during our weekly meetings from: Wednesday: 8:00 - 10:00 PM

Facebook Link *

https://www.facebook.com/DavisWICS/

Instagram Link

https://www.instagram.com/wicsdavis/

Contact Email *

wicsdavis@gmail.com

UPDATE CLUB INFO

Upload club pictures to display on your club's about page.

Choose Files No file chosen

UPDATE PICTURE 1

Choose Files No file chosen

UPDATE PICTURE 2

Select your email preference ▼

UPDATE EMAIL PREFERENCES

EditClubInfo forms that allow user to update club info.

EditItem

Component to select/update item information based on item clicked. When user clicks the name of an item, it populates the EditItemView form with the information of the clicked item.

Select Item To Edit

[Graphic T-Shirt](#)
Another Shirt
Coffee Mug
Generic Basketball
WICS T-Shirt

Clicking an item's name will populate EditItemView forms with item info.

EditItemView

Component to display item information to edit. When an admin user selects an item to be edited, it populates the forms with the matching information. Admin user is able to edit product name, product info, product price, stock, and whether to upload new images. Stock value of form changes depending if the item is an apparel or item.

Select Item To Edit

- Graphic T-Shirt
- Another Shirt
- Coffee Mug
- Generic Basketball
- WICS T-Shirt

Product Name *

Coffee Mug

Product Info *

Mug labelled with a picture of a cup between the "<coffee> </coffee>" label

Product Price *

\$ 15

Stock *

79

To edit any of the pictures for an item please click below.

Choose Files

No file chosen

Choose Files

No file chosen

Choose Files

No file chosen

Choose Files

No file chosen

UPDATE ITEM

Non-Apparel Edit View

Select Item To Edit

Graphic T-Shirt
Another Shirt
Coffee Mug
Generic Basketball
WICS T-Shirt

Product Name *

Graphic T-Shirt

Product Info *

Cute Shirt

Product Price *

\$ 10

Product Stock

7

Small Stock *

2

Medium Stock *

2

Large Stock *

3

X-Small Stock *

0

X-Large Stock *

0

To edit any of the pictures for an item please click below.

Choose Files	No file chosen
Choose Files	No file chosen
Choose Files	No file chosen
Choose Files	No file chosen

UPDATE ITEM

Apparel Edit View

EmailConfirmation

Component that displays an email confirmation is in process and checks if user has validated email during the signup process. An email is sent to the user with a link containing a unique token. After user clicks the link in one's email, it extracts the signup token as a param from URL and makes an API request to /api/signup/confirmEmail. If the email confirmation is complete, the user is then redirected to the login page.

Account Confirmation

Thanks for signing up!. Please click the following link to activate your account:

<https://193ecommerce.com/emailConfirmation/590d1cf14638c3675cfd387cfe91c92047dd8b99>

UC Davis ECS193 E-commerce Team
About Us

Account confirmation view

FAQ

Component that features a list of frequently asked questions and their answers. These questions includes pickup location, refund policy, product issues, how to become a vendor, and contact information. This component is simply left justified and centered due to its simplicity and purpose.

Footer

The Footer follows the visual identity of UCD by using UC Davis Blue for digital branding. It includes the title of our website (ECS 193 E-Commerce) with links to our About, FAQ, About Clubs, Privacy Policy, and Terms and Services components.

GenericPage

Component to display general messages onto screen for information that is not major enough to need its own specific component.

Home

Component that welcomes user to site. Contains an image slideshow and links to the shop page, about page, and clubs page. This slideshow and the images used as quick links are to depict daily life at UC Davis as mentioned in the Styling section.

InputRecoveryPassword

Component that allows user to input email and new password for changing new passwords. The user inputs their email, password, and confirmation password and makes an API request to `/api/resetPass/updatePass`. If the password update process is successful, the user is informed of their new password update for use in the login process.

Login

Component that allows user to login to site. User enters their email and password and makes an API request to `/api/login`. If the inputted data is valid, current user is successfully logged in as a regular or admin user. User is able to login via Google if the user has a Gmail account as well.

OrderHistory

Component that displays the order history of the logged in user. Each order contains the date, email, order ID, picked up status, and total. For every user, the orders can be listed in descending (most recent orders) or ascending (least recent orders), along with a way to display only items that have/have not been picked up. Users can also search by item name.

User Orders

Date: **Ascending** ▾ Picked Up: **Display All** ▾ Search by Item Name

Date	Email	Order ID	Picked Up	Total
Fri Mar 08 2019 4:02 PM	cschen@ucdavis.edu	4oEfTt9Qx2quIOhh5bu	No	\$23.00
Item	Price	Quantity	Total	
Very Cool T-Shirt	\$10.00	1	\$10.00	
Test Product	\$1.00	1	\$1.00	
test	\$12.00	1	\$12.00	

Order History view for non-admin users.

OrderHistoryAdmin

For Admins, all the components in a regular order history still holds, but also allow the admin user to see orders by club. Orders by club show all the orders made to that club and by whom (through their email). This is so clubs can manage their orders. Admin users can also see the personal orders they have made for themselves.

WICS Orders

USER ORDERS **WICS ORDERS** **#INCLUDE ORDERS**

Date: **Ascending** ▾ Picked Up: **Display All** ▾ Search by Item Name

Date	Email	Order ID	Picked Up	Total
Sun Mar 03 2019 9:58 PM	ahong415@gmail.com	090KbDo06g6zZLQpewP4	Yes <input checked="" type="checkbox"/>	\$5.00
Item	Price	Quantity	Total	
mug0	\$5.00	1	\$5.00	

Order History View by club for admin users.

OrderHistoryItem

For each order, a table of each item purchased is listed, along with its item name, price, quantity, and total for that item.

Date	Email	Order ID	Picked Up	Total
Fri Mar 08 2019 4:02 PM	cschen@ucdavis.edu	4oEFtr9Qx2quOhh5bu	No	\$23.00

Item	Price	Quantity	Total
Very Cool T-Shirt	\$10.00	1	\$10.00
Test Product	\$1.00	1	\$1.00
test	\$12.00	1	\$12.00

The organization for each item per order.

PaypalButton

Component that processes payments via PayPal. Payment process is called by loading PayPal's script. Stock check before payment is processed is also handled to ensure enough items are available before payment proceeds.



Privacy

Component that displays the privacy policy of our website. This information includes how personal information is collected/used on our website, along with data collection, minors, and contact information. Like the FAQ of our site, it is simply left-justified and centered.

RecoverPassword

Component that allows user to enter email to request a password update. An API request to `/api/resetPass` is made and an email is sent to the user's inputted email to verify password update process.

ScrollToTop

Component to bring user's view of webpage to the top of the page when a page is switched.

ShopView

Component that displays ShopItem components for a user to browse/purchase. All products stored in our database are retrieved by making an API request to /api/getAllProducts. For each item in the result, a ShopItem component is created and rendered onto the screen. This page displays a grid of each item with a hero image of the UCD Farmer's market on top to make the page seem less plain.

ShopItem

Component that displays a ShopItem for a user to browse and add to cart in the ShopView component. An image of the product is displayed along with the price and a more from vendor option to view more items from selling club. If user clicks the image of the item, it proceeds to the ShopItemDetailed component of the selected item.

ShopItemDetailed

Component that displays more detailed information about a selected ShopItem component and the option to add the item to a user's cart. Information displayed includes product name, product price, stock availability, product description, slideshow of magnifiable images. User has an option to select amount of items to be added to cart and size if the item is an apparel. A check if the user selects an amount that exceeds stock available is present as well. This page is done in a two column set up to fill up the screen. Depending on the stock, the color will either be blue or red depending if stock is low/out.

Signup

Component that allows user to register an account with our site. User inputs their first name, last name, email, password, and confirmation password. User also has an option to signup via Gmail.

Terms and Services

Component that displays our Terms and Services policy as part of the legality of our website. It includes an overview, along with sections that describe the legal aspects of our website. Some of these sections include Online Store Terms, General Conditions, and Third Party Links. Like the FAQ and Privacy Policy page, this is also left justified and centered.

VendorSignup

Component that allows a user to become an admin for one of the clubs listed within our site. This process allows the user to have admin privileges to add products, edit club information, and more. User inputs their email, verification code and selects which club to be an admin of.



Admin Verification

Email *

Access Code *

Select Vendor

Select Club Name

VERIFY

VendorView

Component that displays items provided by a specific vendor. When a user selects the “More From Vendor” tab on a ShopItem component, the VendorView component is rendered based on the vendorID. ShopItem components are rendered based on if the vendor sells the item, along with brief information about the vendor is displayed as well. The component functions similarly

to the ShopView component, where the difference is items only specific to a vendor are rendered/displayed. There is no hero image for a club's page because the images club's typically tend to provide are not suited as hero images.

Frontend Styling

The styling in the frontend is done to match the visual identity of UC Davis. We use the fonts, colors, and header depicted in marketingtoolbox.ucdavis.edu. The design choices for each component can be read in their respective section in the list above. The photos and overall aesthetic of the site is to give the user the feeling of exploring life at UC Davis.

All photos are taken from UC Davis's official photo sharing source here <https://ucdavis.photoshelter.com>. For this site, an official UC Davis keroberos must be used to create an account and gain access to the photos.

If the images are too large, thus slowing down the loading time of the site, use this website to lower the resolution of the images <https://compressimage.toolur.com/>.

We use HTML and CSS (mainly div class names and IDs) to structure the styling of our code. We currently have all our CSS pages divided per component, although each component's CSS may override each other.

This was done at the beginning of the project, and should be refactored into one main CSS file.

We use React Material UI for many components in the frontend (such as for the ButtonAppBar, the Paper containers for login/signup, etc). For styling those pages, in line styling in the react code must be done to override any CSS that the library provides. Below is an example on how to do in-line styling. You CANNOT write CSS code and wrap the Material UI with a div. Please see <https://material-ui.com/> for more information.

```
<AppBar
  position="static"
  style={{ backgroundImage: `url(${header})`, backgroundSize: "cover", boxShadow: "none"}}
>
```

For responsive styling, we made a basic view for mobile and tablet. We use 40em and 767px for mobile and tablet respectively. They are at the bottom of each CSS page. There is less responsive code for tablet as much of the component's default desktop view looks fitting for tablet. Some pages, use the tag Hidden from Material UI. This hides or shows certain views depending on the screen size. More can be read on that here.

<https://material-ui.com/components/hidden/>

Logo

Here is a png of our logo/favicon. It was done in Adobe Illustrator and can be edited there. It depicts a cow shopping and holding a laptop, a reference to e-commerce.



Appendix

Glossary of Terms

Domain: The name of our site (193ecommerce)

Https: Form of security to encrypt our communication

Client: What user sees on the webpage, frontend aspect of our project.

Server: How our site handles data logic between frontend and database interactions.

Database: Where we are storing our data and information for users, products, etc.

Technology Survey

Summary

We plan to host our site using Google's App Engine and integrating Firebase as our database. Due to all of us being familiar with JavaScript, we decided to use Node/Express as our backend language, in which App Engine also supports. For our frontend framework, we decided to use React also due to our experience and interest in it. To minimize the complexity of payments, we are leaning towards using Paypal.

Name	Pros	Cons
Server: Google App Engine	<ul style="list-style-type: none"> - Server to store our code, host our site - Connect to database hosted on Firebase - Can host Node.js, store our code/files 	<ul style="list-style-type: none"> - SSL setup with custom domain name
Database: Firebase (NoSQL)	<ul style="list-style-type: none"> - Entire package - Easy deployment server - NoSQL - flexible data object storage - JSON Storage - Google Cloud Platform integration with Google App Engine - Experience with NoSQL (MongoDB) 	<ul style="list-style-type: none"> - NoSQL: possibly limited database transactional capabilities - Restricted to what Firebase can do
Backend: Node/Express	<ul style="list-style-type: none"> - All familiar with JavaScript - Developed using NPM libraries - Fullstack JS, frontend and backend in same JavaScript language - Most comfortable backend language 	<ul style="list-style-type: none"> - Async troubles
Frontend: React	<ul style="list-style-type: none"> - Virtual DOM, render only when components update - Previous experience/familiarity - Interested in practicing more with library 	<ul style="list-style-type: none"> - No defined methodologies among community for development - Learn JSX

Payment: Paypal	- Easy to implement, minimize complexity	- Not everyone has PayPal/would want to make a PayPal

Technology Survey Results

We have decided to go with Google App Engine, Firebase (noSQL), Node/Express, React, and PayPal. We decided to use Google App Engine naturally because this is a Google sponsored project. We chose Firebase because it's easy to integrate with our application being deployed onto Google App Engine. We have team members who have experience using NoSQL databases as well. We went with Node/Express because of our prior experience with Node/Express and wanting to have a consistent stack using all JavaScript. We chose React because we have experience with React and familiarity with JavaScript. Modern trends for development among companies use React, so we want to practice more with it. Lastly we chose PayPal as it is an existing, free, secure way to process transactions, as we do not want to be directly liable for our users' money.

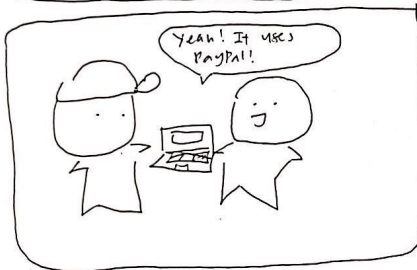
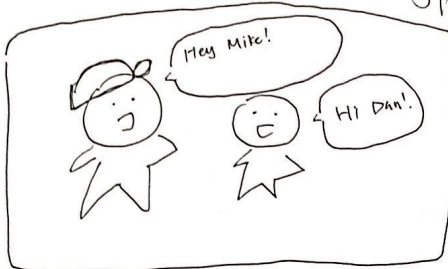
Real-World Constraints

We cannot provide shipping to our clients, and many UC Davis clubs/vendors are satisfied with the commerce system they have already. Also, some clubs only sell items during a specific part of the school year and not all clubs sell merchandise year round. Informing when and where to pick up products between a customer and vendor will depend on the vendor. Also, we will likely not maintain this website after we graduate. Another constraint we encountered was PayPal's non-partner restrictions. In regards to vendors, some distributors have declined participation in our website as they sell copyrighted material, and would like their products to remain under the spotlight. As partners, when someone makes a checkout to multiple vendors, PayPal will handle distributing the correct profits to the appropriate vendor. However, as we are not an official business, we cannot get PayPal partnership, thus when a consumer checks out with merchandise from multiple clubs, we have to handle the distribution of funds ourselves.

Storyboard

The first is how a buyer would be introduced to the website, the second is how a seller would be introduced.

STORY BOARD





Requirements

User Stories/Cases

User Story	Description	
	User can get a brief overview of the website	The landing page will be a slideshow with images of life at UC Davis. There are images with quick links to the key features of our site. This is so users can immediately see what our site is for.
	Users can read the About page for the website	This page displays general information about the site including our technology stack, motivation, and contact information.
	Users can see a list of clubs	Users can see which clubs are featured to spark

	featured	interest in how they want to consume their time/money. This can be a fun way to explore/navigate the website.
	Users can select which club to shop/learn from	If users are interested in supporting a specific club, they can immediately jump to having solely that clubs' merchandise displayed to browse what products they may be interested in. They can also read more about the club.
	Users can view About section of clubs	Users can access the general information regarding a club. Using this section, users can learn more about a club and what their products entail. Users are naturally curious, and this section may answer some unexpected questions they may have.
	Users can add items	Vendors who have come up with new merchandise can add it to their page. Another easy and simple way for users to respond to any changing inventory or incorporate their new ideas/products.
	Users can use the navigation bar	Users can use the navigation bar to access key features of the website such as the shop, club, about, and account information depending on the type of privileges the user has.
	Users can view their cart	Users can view the cart to manage what items they are/aren't considering purchasing. This provides a quick way for buyers to get at a glance all their interests on one page.
	Users can add items to a cart	Buyers can add items to their cart as a quick way to save the items they are interested in. They can also view the cart to manage what items they are/aren't considering. As a buyer, I would want some easy way to keep track of items so I don't have to constantly remember what products I enjoy.
	Users can buy products	Users can securely purchase items through an existing payment API (PayPal). They will receive a confirmation email that their purchase has been received. As a buyer, I would want a secure existing payment method for my purchases, and confirmation that my transaction has gone through.

	Users can sell products	Vendors can securely receive payment and receive requests that a payment/order has been processed. They will also receive email confirmation. As a vendor, I would want a uniform way to sell my products, and a receipt of what of my inventory has been purchased.
	Users can edit club info	Club vendors can edit basic information about their club so consumers can understand more of where their purchases is towards or learn how to participate. This makes it easy for users to respond to changes that may come about in regards to their club.
	Users can edit items	Vendors can edit their items such as their product names, descriptions, etc. This allows for users to react to changes in regards to their products.
	Users will get email confirmation upon a purchase	When users complete a purchase, they will get an email confirming the transaction, along with details of it. These details include basic item information, order ID, and pick up location.
	Users can access a FAQ	Users can read an FAQ in order to have a simple way to access basic questions. As a user of an ecommerce website, I would want a quick way to have my questions answered before I contacted an official.
	Users can navigate the footer	Users can use the footer to access quick links such as the About page and the legal aspects of our club.
	Users can have account recovery	Users can recover their accounts if they forget their information. Users can be forgetful, and we want to be considerate of that.
	Users can log in and to sign up	There will be a login and signup button in our navbar. We plan to have vendors to have admin accounts so that they can update their products/information. Buyers can manage their order/cart history through their account. This is to provide an easy way for both vendors and buyers to access important transactions.
	Users can view their order history	Users can view their order history to see their recent transactions. Users can see if they have picked up their orders yet. Vendors can see their

		orders by clubs also.
	Users can view Privacy Policy	Here users can understand what information of theirs is and isn't protected. Again as a buyer/vendor, I'd like to know how my information is being used.
	Users can view a shop page	Users can shop by all items or by a club's specific merchandise. This is for users who are simply interested in browsing inventory for leisure or for a club they are specifically interested in.
	Users can view a shop item page	Users can view a product's description and use this page to add it to a cart. This is for users to understand more about a product.
	Users can view our Terms & Services	Here is where users can learn what we are/aren't responsible for regarding their transactions. As a buyer/vendor, I would want clear documentation of the website's responsibilities.
	Users can apply to become a vendor	Form where users can apply to become a vendor and sell their products. A potential vendor should have a clear way on how to put their products on our market.

Prototyping Code

<https://github.com/clauidiaschen/ecommerce>

Technologies Employed

We are using Firebase, Google App Platform, React, and Node/Express, and the PayPal API.

Social/Legal

The social aspect of our project is that it will provide easier access for UC Davis vendors to sell and buy merchandise to a wider audience. It will also make actions (buying/selling) more manageable since there are confirmation and online receipts.

In order to make our website as secure as possible, we will be utilizing existing payment APIs (PayPal). We also plan to examine existing ecommerce websites terms and services and privacy policies to draft up our own. We will use a template website that will take into account the features of our website and return the legal documentation of our website for us.