SW Engineering CSC648/848 Fall 2019

Gator Trader

Team 10

Ibraheem Chaudry - Team Lead (ichaudry@mail.sfsu.edu)

Tom Sechrist - GitHub Master

Alexander Beers - Back-end Lead

Lance Santos - Front-end Lead

Paul Lueng - Front-end

Saleh Zahran - Back-end

Milestone 4

12.12.19

Features

Gator Trader provides an online e-commerce platform personalized for SF State students. The main functions of the product are as follows:

User features:

- shall be able to register for an account.
- shall be able to browse through the website.
- shall be able to use the search bar to search for items.
- shall be able to view all item details.
- shall be able to fill out a form to post an item for sale.
- shall be able to log into the web application.
- shall be able to post items for sale.
- shall be able to contact other registered users about products.
- shall be able to receive messages from other registered users.
- shall be able to delete their post.

Administrator features:

- shall be able to approve pending posts before they go live.
- shall be able to reject post proposals from registered users.
- shall be able to send messages to users.
- shall be able to see the info of a registered user.
- shall be able to ban users.
- shall be able to remove a post.

Our product is unique in that its items are tailored to students at SFSU and will allow for a quick and easy way to meet up with buyers and sellers.

Link to website: http://gatortrader.ga/

Usability Test

Test objectives:

We are testing search page usability. The usability of the search page is paramount to the success of our product as if no one can search properly then no contacts could be made and there is no purpose for our product.

Test background and setup:

System setup: The setup will be a simple single room setup. One monitor and one input device needed for the test. This will allow for a simple observation of the results of the test.

Starting point: We will start the testers at the home landing page of our website. We will want to see how quickly and easily our testers will be able to search on our site when trying it out for the first time.

Who are the intended users: We will select average SFSU students for our test. We will especially looking for students who have a minimal to novice level of computer experience as we will want this product to be easily accessible to all students, regardless of computer experience.

http://gatortrader.ga/ is the URL of our landing page. http://gatortrader.ga/results is the page after the tester has tried to search for an item.

What is to be measured: We will measure the effectiveness of the website by the percent of cases successfully and we will measure the efficiency by comparing the average time of the successful test over the average time of all users.

Usability Task description:

We will ask the testers to go to our webpage, search for an item and attempt to contact the seller.

- Measuring Effectiveness
    - Take the percent of cases completed.
    - Number of errors made
    - Additional comments.
- Measuring Efficiency
    - Average time to complete the task.
    - Average time of those who completed the task/average time of all users.
    - Number of clicks.
    - Number of web-pages visited.

Lickert subjective test:

We will focus on our testers satisfaction with the searching process. We will have a questionnaire for the testers with the following questions:

“I found it easy to search for an item”

Strongly disagree - Disagree - Unsure - Agree - Strongly agree

Additional Comments: _____

“I found the webpage necessarily complex.”

Strongly disagree - Disagree - Unsure - Agree - Strongly agree

Additional Comments: _____

“I would use this website often”

Strongly disagree - Disagree - Unsure - Agree - Strongly agree

Additional Comments: _____

QA Testing

Test objectives:

User Search. We are focusing on the effectiveness of our %LIKE database search function.

Hardware and Software setup:

- URL: http://gatortrader.ga/
- Hardware setup
    - The server is set up on AWS EC2 instance and the operating system is Amazon Linux AMI
    - Any device that has a Google Chrome or Safari can run our application without any bugs

Feature to be tested:

In this QA test, we are testing how accurate our search is.

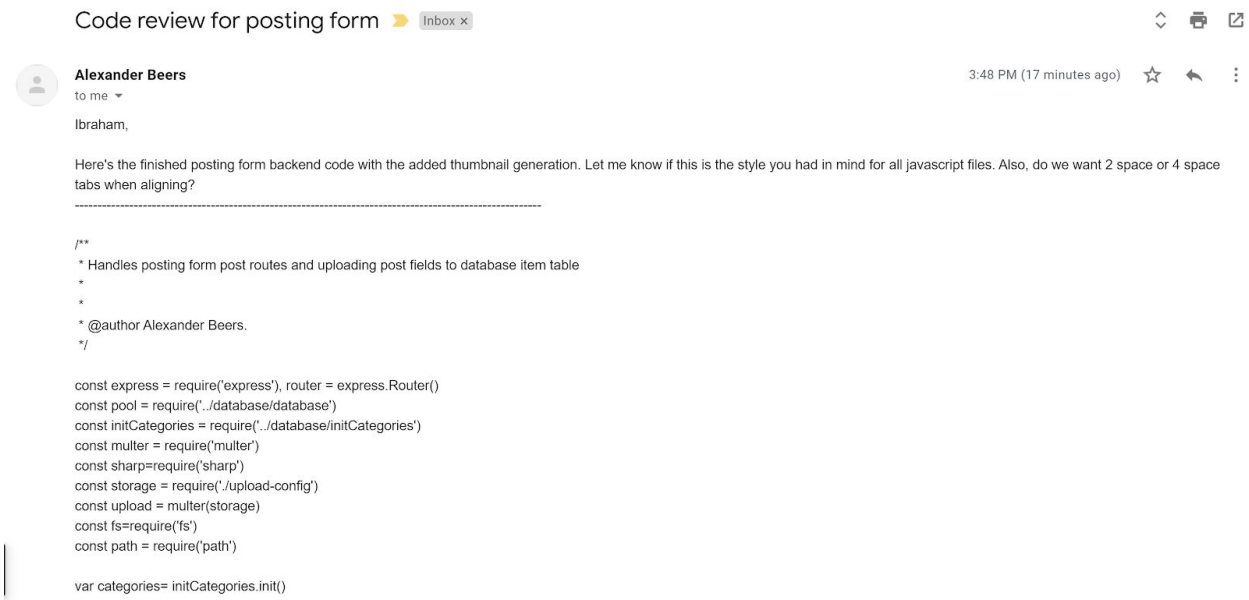
QA Test plan:

Number	Description	Test Input	Expected Output	Pass/Fail Chrome	Pass/Fail Firefox
1	Test %LIKE on search	"test" in the top search bar	Get 4 results with "test" in the title	Pass	Pass
2	Test %LIKE on search	"" in the top search bar	Get all 8 results	Pass	Pass
3	Test %LIKE on search	"item"in the top search bar	Get 4 results with "item" in the title	Pass	Pass

Code Review

We have used best practiced coding styles to make our code easily readable. Here are some rules our team uses to maintain good quality code:

- Using inline comments where needed to briefly explain the function of particular line/lines of code.
- Using header comments which include the author's name of the person/persons who wrote the piece of code along with a brief description of the function performed by a module.
- Using good variable names that are descriptive while being easy to remember at the same time.
- Using camel case with a lower case first character to write names in javascript.
- Using kebab_case to write names in css.
- The applications directory structure has been designed to follow best practices MVC templates.
- Recycling of code is done where possible to avoid hard coding and copy-pasting similar code.



Code review for posting form Inbox ×

Alexander Beers 3:48 PM (17 minutes ago)
to me

Ibraham,

Here's the finished posting form backend code with the added thumbnail generation. Let me know if this is the style you had in mind for all javascript files. Also, do we want 2 space or 4 space tabs when aligning?

```
/**
 * Handles posting form post routes and uploading post fields to database item table
 *
 *
 * @author Alexander Beers.
 */

const express = require('express'), router = express.Router()
const pool = require('../database/database')
const initCategories = require('../database/initCategories')
const multer = require('multer')
const sharp=require('sharp')
const storage = require('./upload-config')
const upload = multer(storage)
const fs=require('fs')
const path = require('path')

var categories= initCategories.init()
```

```
//Route for posting form page
router.get("/postingForm", (req, res) => {
    //timeout necessary to get categories to appear before page is refreshed
    res.setTimeout(200, () => {
        res.render('postingForm', {
            itemName: "",
            description: "",
            price: 0.0,
            image: "",
            categories: categories,
            category: "",
            isLogin: req.session.loggedin
        })
    })
})
```

```
//Post function to make an upload item request to the database
router.post('/postingform', upload.single('image'), async (req, res) => {
    const { filename: image } = req.file

    //Used for thumbnail generation
    await sharp(req.file.path)
    .resize(500)
    .jpeg({quality: 50})
    .toFile(
        path.resolve(req.file.destination,'resized',image)
    )
    fs.unlinkSync(req.file.path)

    console.log("--- Post Entry --- ");
    console.log("itemName: " + req.body.itemName);
    console.log("description: " + req.body.description);
    console.log("price: " + req.body.price);
    console.log("category: " + req.body.category);
    console.log("image filename: " + req.file.filename)
    console.log("-----------------");

    if (req.fileValidationError) {
        return res.end(req.fileValidationError);
    }

    //synchronously gets userID of user that posted and then creates a database record from all the postform values
    postUpload(req.session.email, req.body.itemName, req.body.description, req.body.price, req.body.category, req.file.filename)
    res.redirect('back')
})

//Query for a userID using user email address
function getUserID(email) {
    return new Promise(resolve => {
        pool.query("SELECT userID FROM User WHERE email = ?", [email], (err, rows, result) => {
            var userID = 0
            if (rows[0]) {
                var userID = rows[0].userID
                resolve(userID)
            }
        })
    })
}
```

```
    //Upload item to the database
    async function postUpload(email, name, description, price, category, file) {
        const userID = await getUserID(email)
        pool.query("INSERT INTO item (userID, name, description, price, category, picture) VALUES (?,?,?,?,?,?)",
            [userID, name, description, price, category, file], (err, rows, result) => {
                if (err) console.log(err)
            })
    }
```

Ibraheem Chaudry <ibraheem27496@gmail.com> 4:06 PM (7 minutes ago)

to Alexander

Hello Alexander

Thanks for sending me the code for review. You have done an excellent job on the implementation. The code styling looks great and I don't see any portions that look like spaghetti code.

There are some lines where indentation is a little off and the code gets hard to read like in the post upload function. Other than that the code looks good. Please correct the indentations where needed.

For indentation please stick to 4 spaces as it is good practice and we have been using 4 spaces in all our other modules.

Thanks!
Best,
Ibraheem Chaudry

Self-check on best practices for security

Major assets being protected

- User Data
    - User Data is being protected through preventing SQL injection form inputs. This is being done through client side form validation.
        - All forms and input bars only allow up to 40 alphanumeric characters.
    - Passwords are encrypted before storing in the database
- Defense against cross scripting attacks
    - Search field and input forms will be verified for valid inputs only.
- Images of items posted
    - The uploaded images are being protected as they are stored locally on the server rather than as BLOBS in the database.

Self-check: Adherence to original Non-functional specs – performed by team leads

1. **DONE** - Application shall be developed, tested and deployed using tools and servers approved by Class CTO and as agreed in M0 (some may be provided in the class, some may be chosen by the student team but all tools and servers have to be approved by class CTO).
2. **ON TRACK** - Application shall be optimized for standard desktop/laptop browsers e.g. must render correctly on the two latest versions of two major browsers
3. **ON TRACK** - Selected application functions must render well on mobile devices
4. **DONE** - Data shall be stored in the team's chosen database technology on the team's deployment server.
5. **DONE** - No more than 50 concurrent users shall be accessing the application at any time
6. **DONE** - Privacy of users shall be protected and all privacy policies will be appropriately communicated to the users.
7. **DONE** - The language used shall be English.
8. **ON TRACK** - Application shall be very easy to use and intuitive.
9. **DONE** - Google analytics shall be added
10. **DONE** - No e-mail clients shall be allowed
11. **DONE** - Pay functionality, if any (e.g. paying for goods and services) shall not be implemented nor simulated in UI.
12. **DONE** - Site security: basic best practices shall be applied (as covered in the class)
13. **DONE** - Modern SE processes and practices shall be used as specified in the class, including collaborative and continuous SW development
14. **DONE** - The website shall <u>prominently</u> display the following <u>exact</u> text on all pages *"SFSU Software Engineering Project CSC 648-848, Fall 2019. For Demonstration Only"* at the top of the WWW page. (Important so as to not confuse this with a real application).