SW Engineering CSC 648/848 Fall 2020
Team 4

Milestone 4

GatorTrader

Niall Healy (Team Lead, nhealy@mail.sfsu.edu)
Aaron Lander
Dale Armstrong
Lukas Pettersson
Joseph Babel
Vern Saeteurn

Date submitted	Date(s) Revised
December 8, 2020	

Table of Contents:

1. Product Summary:

GatorTrader is a unique website that allows San Francisco State University students to search for, purchase, sell, and trade essential materials for their courses. Our free-to-use website provides users with core functions that will allow them to make use of excellent features such as search by category or class, post item listings, and a messaging system in addition to a clean user interface. **GatorTrader**'s core functions include:

1. **Unregistered Users**
    - 1.1. Shall be able to search for items by item name, class, category
    - 1.2. Shall be able to use their SFSU email to create an account
2. **Registered Users**
    - 2.1. Shall be able to post item/service **listings**
    - 2.2. Shall be able to send **messages** to sellers
    - 2.3. Shall log in with a username and password

GatorTrader is unique due to the need for a medium that provides SFSU students the ability to make transactions with other students. With **GatorTrader**, students are able to message owners of post listings so that they have all the information they need before committing to a purchase.

URL: http://ec2-3-21-104-38.us-east-2.compute.amazonaws.com/

2. Usability Test Plan:

Test Objectives

We have decided to test the search capabilities of the GatorTrader website. Search was chosen because we feel it is one of the most important aspects of our website. Without search, GatorTrader is nearly useless.

Many criteria will be used to test search capabilities, including:

- Ease of finding and using the search bar
- The quickness of the results being loaded
- The satisfaction with the results received

With the feedback provided, we are hoping to gain insights into how we can provide the optimal experience for GatorTrader users.

Test Background and Setup

- System Setup:
    - Users will test the GatorTrader website using both Google Chrome and Firefox browsers.
- Starting Point:
    - The starting point of all tests will be on GatorTrader's home page.
- Intended Users:
    - Students and Faculty of San Francisco State University
- URL to be tested:
    - http://ec2-3-21-104-38.us-east-2.compute.amazonaws.com/html/index.html
- What is being tested:
    - Ease in finding and using the search bar
    - Quickness of results being returned
    - Receiving expected results from search

Usability Task Description

- Task
    - Search for a textbook on GatorTrader
- Success criteria
    - User finds an expected result from search query
- Benchmark:
    - Users are able to locate the search bar, enter a search query, and receive a result all within 10 seconds.

Effectiveness:

Effectiveness will be determined by:

- User's response to ease of finding and using search bar
- User's response to satisfaction of search results
- Any additional comments left by user

Efficiency:

Efficiency is measured by:

- Average time it takes for user to complete task
- User's response to results loading speed.
- Any additional comments left by user

Survey

	Strongly Disagree	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**	**Comment (Optional)**
The search bar was easy to find and use.						
The results loaded quickly.						
The results I received from my search were expected and satisfactory.						

3. QA Test Plan:

Test Objectives

Ensure the website displays the correct content based on the behavior of the search bar.

HW and SW Setup

Using Windows 10 with Google Chrome running version 87.0.4280.88, as well as Firefox running version 83.0.

Start with Google Chrome and go to the homepage. Each test should start on the homepage and after all page elements are finished loading. Once all tests are completed with Google Chrome, repeat the process with Firefox until the table is filled out.

The homepage is at the following URL:
http://ec2-3-21-104-38.us-east-2.compute.amazonaws.com/html/index.html

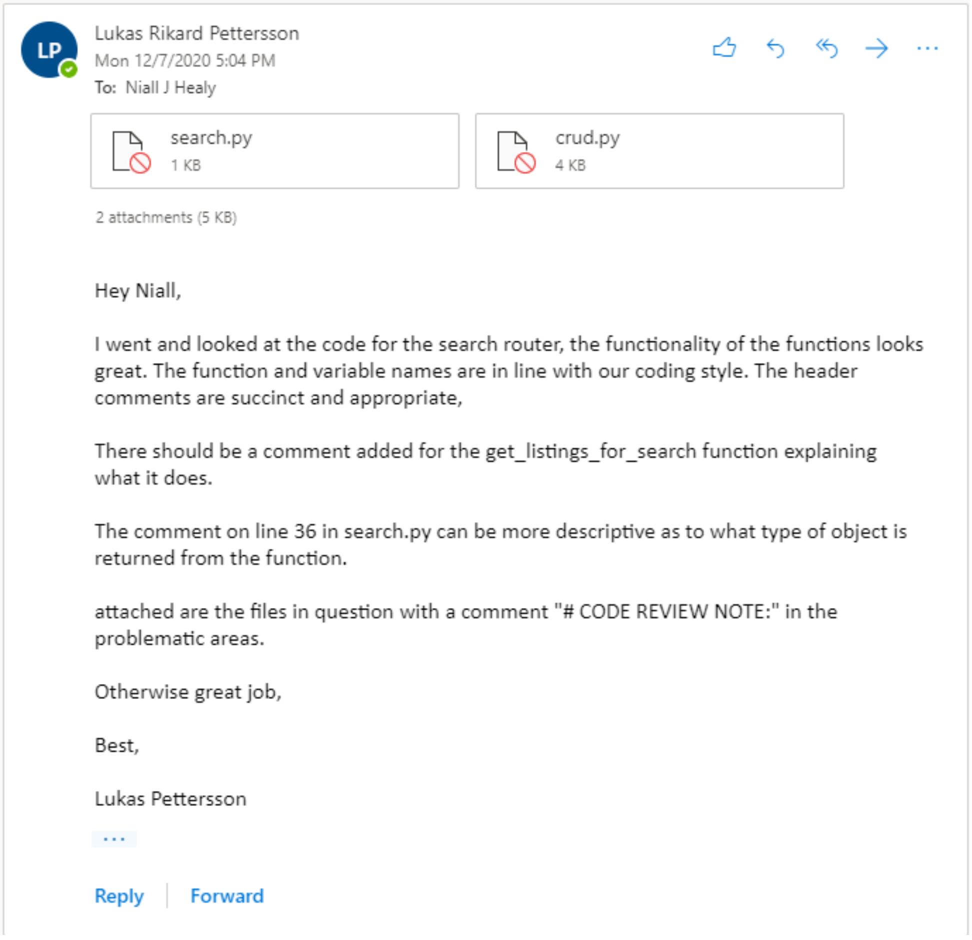
Testing Feature

The following test plan is to ensure expected behavior for the search bar.

QA Test Plan

Number	**Description**	**Test Input**	**Expected Output**	**Chrome (PASS/FAIL)**	**Firefox (PASS/FAIL)**
1	Test search persistence after a search	Type “book” in the search bar and click the search button	“book” will persist on the following results page	PASS	PASS
2	Test % like when searching by name	Ensure category is set to “Any” and type “calculus” into the search bar, then click the search button	1 result with the title: “precalculus with limits”	PASS	PASS
3	Test search by category	Click on category and set it to “Housing”. Ensure the search bar is clear and click on the search button	7 results showing all available housing	PASS	PASS

4. Code Review:



Lukas Rikard Pettersson
Mon 12/7/2020 5:04 PM
To: Niall J Healy

search.py
1 KB

crud.py
4 KB

2 attachments (5 KB)

Hey Niall,

I went and looked at the code for the search router, the functionality of the functions looks great. The function and variable names are in line with our coding style. The header comments are succinct and appropriate,

There should be a comment added for the get_listings_for_search function explaining what it does.

The comment on line 36 in search.py can be more descriptive as to what type of object is returned from the function.

attached are the files in question with a comment "# CODE REVIEW NOTE:" in the problematic areas.

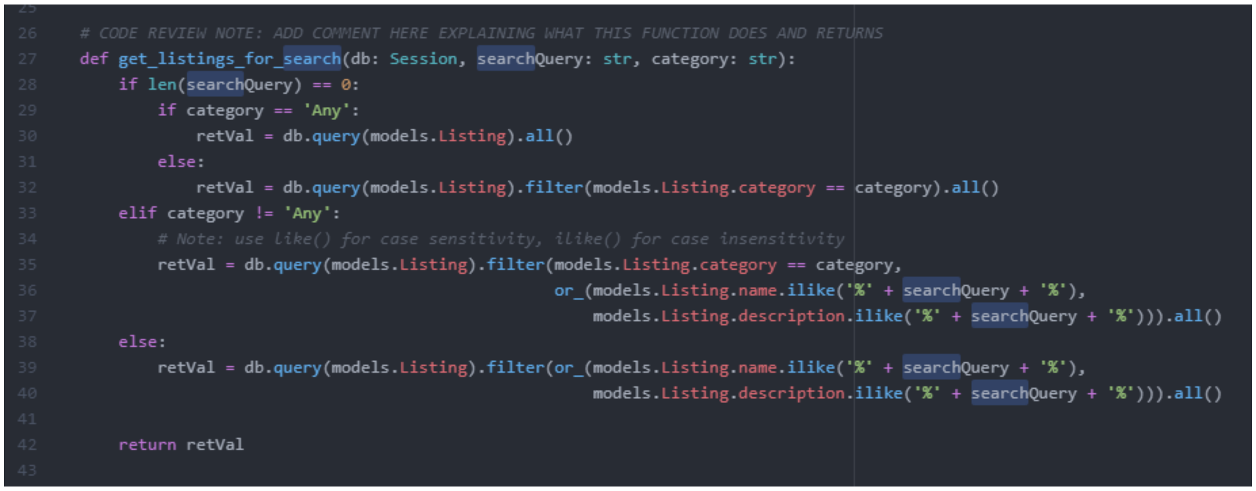
Otherwise great job,

Best,

Lukas Pettersson

Reply | Forward

Figure 1: Code review email

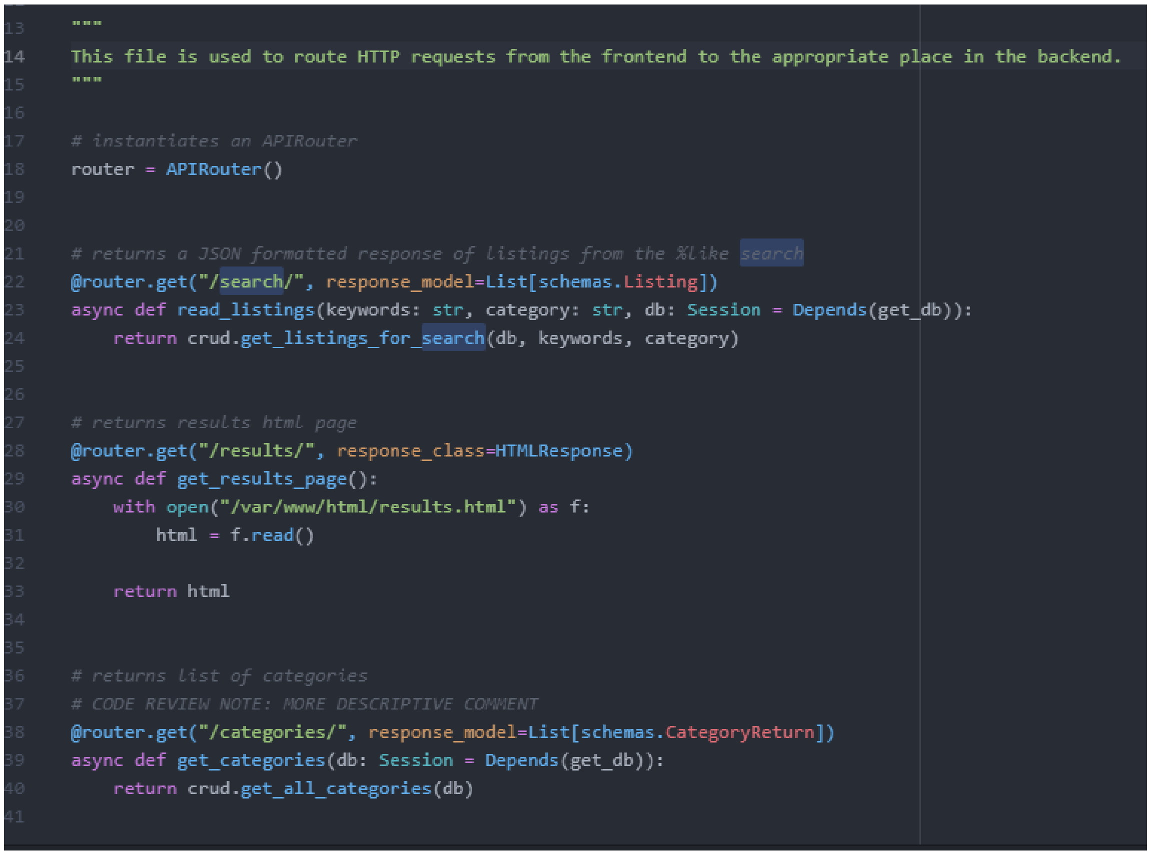


```python
# CODE REVIEW NOTE: ADD COMMENT HERE EXPLAINING WHAT THIS FUNCTION DOES AND RETURNS
def get_listings_for_search(db: Session, searchQuery: str, category: str):
    if len(searchQuery) == 0:
        if category == 'Any':
            retVal = db.query(models.Listing).all()
        else:
            retVal = db.query(models.Listing).filter(models.Listing.category == category).all()
    elif category != 'Any':
        # Note: use like() for case sensitivity, ilike() for case insensitivity
        retVal = db.query(models.Listing).filter(models.Listing.category == category,
                                                 or_(models.Listing.name.ilike('%' + searchQuery + '%'),
                                                     models.Listing.description.ilike('%' + searchQuery + '%'))).all()

    else:
        retVal = db.query(models.Listing).filter(or_(models.Listing.name.ilike('%' + searchQuery + '%'),
                                                     models.Listing.description.ilike('%' + searchQuery + '%'))).all()

    return retVal
```

Figure 2: Part of crud.py pertaining to search with comments on suggested fixes



```python
"""
This file is used to route HTTP requests from the frontend to the appropriate place in the backend.
"""


# instantiates an APIRouter
router = APIRouter()


# returns a JSON formatted response of listings from the %like search
@router.get("/search/", response_model=List[schemas.Listing])
async def read_listings(keywords: str, category: str, db: Session = Depends(get_db)):
    return crud.get_listings_for_search(db, keywords, category)


# returns results html page
@router.get("/results/", response_class=HTMLResponse)
async def get_results_page():
    with open("/var/www/html/results.html") as f:
        html = f.read()

    return html


# returns list of categories
# CODE REVIEW NOTE: MORE DESCRIPTIVE COMMENT
@router.get("/categories/", response_model=List[schemas.CategoryReturn])
async def get_categories(db: Session = Depends(get_db)):
    return crud.get_all_categories(db)
```

Figure 3:search.py with comments on suggested fixes

5. Self-check on best practices for security:

Asset	Threat	Control
Users	Bad Actor uses XSS (Cross Site Scripting) to inject malicious code into a User's html.	1. Validate form inputs. Including character limit, size limit, file upload types, and limiting types of characters to only alphanumeric and dashes.
User Information	Bad Actor gains access to data through Buffer Overflow.	1. Validate all input fields. 2. Check and Limit the number of characters that the user may submit per input field.
User Information	Bad Actor gains access to a user account through brute force password.	1. Force a minimum number of characters for password to ensure the password is decently strong. 2. Include Captcha when signing in to limit bots capabilities to brute force.
User Information	Unauthorized users gain access to another user's data through a GET or POST request.	1. Use authorization tokens generated by JWT and a strong Secret Key to ensure that only the user with the correct token is able to access their data.
User Information	Unauthorized users gain access to another user's data through a man in the middle attack, listening to data being sent to and from the User and Database.	1. Will attempt to implement HTTPS utilizing a free certificate from the CA (Certificate Authority) Let's Encrypt.
User Passwords	If someone gains access to the database, they would now have the user's passwords that may be used on other sites.	1. Hash and Salt the passwords using bcrypt. One way encryption makes it harder to decrypt.
Database	Corruption through faulty data or mishandling	1. Create a backup and restore option. Backup every time the

		database is updated. 2. Protect the images folder
Database Integrity	Unauthorized users may attempt to use SQL Injection to manipulate the database or retrieve user’s data.	1. SQLAlchemy classes and functions provide automatic escaping of special characters that may be used for SQL Injection.
Database Integrity	Spam bots create accounts to spam and upload faulty data.	1. Include Captcha when registering for an account to limit the ability for bots to create accounts.
Server	Bad Actor attempts to SSH into the server to gain access to information and manipulate the server files.	1. Utilize a strong key-pair to protect the server from unauthorized access. This key-pair is provided by Amazon.
Server	Bad Actors attempt to perform a DDoS (Distributed Denial of Service) attack to attempt to take down the server.	1. Utilize Amazon services to host the site. Amazon AWS Shield is automatically enabled for sites hosted on AWS. This provides protection against most common DDoS attacks.
Records Integrity	Users may upload malicious data or images to harm the reputation of the site.	1. Moderation - All posts must be pre approved before they can be seen by other users.

Password Encryption

- Stored user passwords are hashed and salted using bcrypt.

Input Validation

- All form input fields have a character limit along with limiting characters to alphanumeric and dashes. 40 max length for search is implemented.
  - maxlength and pattern attributes in html to control search input.
- Ensure the user emails start with proper names and end with @sfsu.edu or @mail.sfsu.edu
  - Regex in Javascript
- Ensure the email is not already used.

    - Query the database and check if the email is already in use.
- File upload - Number of files, file size, and file type limits.
    - Javascript to check all the variables
- Password 6 characters minimum and 40 characters maximum.
    - Javascript checks character length and notifies user
- Password confirmation matches.
    - Javascript to check if both passwords are the same

6. Self-check: Adherence to original Non-functional specs:

1. 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). **DONE**
2. Application shall be optimized for standard desktop/laptop browsers e.g. must render correctly on the two latest versions of two major browsers. **DONE**
3. All or selected application functions must render well on mobile devices. **DONE**
4. Data shall be stored in the database on the team's deployment server. **DONE**
5. No more than 50 concurrent users shall be accessing the application at any time. **ON TRACK**
6. Privacy of users shall be protected and all privacy policies will be appropriately communicated to the users. **DONE**
7. The language used shall be English (no localization needed). **DONE**
8. Application shall be very easy to use and intuitive. **DONE**
9. Application should follow established architecture patterns. **DONE**
10. Application code and its repository shall be easy to inspect and maintain. **DONE**
11. Google analytics shall be used. **ON TRACK**
12. No e-mail clients shall be allowed. Interested users can only message to sellers via in-site messaging. One round of messaging (from user to seller) is enough for this application. **DONE**
13. Pay functionality, if any (e.g. paying for goods and services) shall not be implemented nor simulated in UI. **DONE**
14. Site security: basic best practices shall be applied (as covered in the class) for main data items. **DONE**
15. Media formats shall be standard as used in the market today. **DONE**
16. Modern SE processes and practices shall be used as specified in the class, including collaborative and continuous SW development. **DONE**
17. The application UI (WWW and mobile) shall <u>prominently</u> display the following <u>exact</u> text on all pages *"SFSU Software Engineering Project CSC 648-848, Fall 2020. For Demonstration Only"* at the top of the WWW page. (Important so as to not confuse this with a real application). **DONE**