

Student Information

Name: Kabelo Dike

Student ID: 20391493

E2E Testing Approach

To validate the functionality and user experience of the Library Management System, end-to-end testing was performed using pytest and Playwright. Playwright enabled full browser-based automation, allowing tests to interact with the application exactly like a real user by navigating pages, submitting forms, clicking buttons, and reading rendered HTML content. Four core user workflows were tested: adding a book to the catalog, borrowing a book, returning a book, and searching the library collection. Each flow included UI-focused assertions, such as verifying success flash messages, confirming table entries appear, and ensuring navigation links and form elements render correctly. Because the tests operate through the Flask UI and use the live SQLite database, they validate the entire system stack including frontend, backend routing, business logic, and persistence, providing strong confidence in overall application reliability and usability.

Execution Instructions

Running app with docker:

Pull the published image:

```
docker pull dike12/library-app:v1
```

Run the container:

```
docker run -d -p 5000:5000 --name library-app dike12/library-app:v1
```

Verify the container is running:

```
docker ps
```

Open the application in a browser:

Visit:

<http://localhost:5000> (note: this goes in the browser search bar)

Running test:

Create and activate virtual environment:

```
python3 -m venv venv  
source venv/bin/activate
```

Install dependencies:

```
pip install -r requirements.txt  
playwright install
```

Run the test suite:

```
pytest tests/test_e2e.py
```

Test Case Summary

Test Name	User Actions (Flow)	Expected Results
test_add_book_appears_in_catalog	<ul style="list-style-type: none">Open app homepageClick Add Book in navbarFill in Title, Author, ISBN, CopiesSubmit form	<ul style="list-style-type: none">Redirects to Catalog pageNew book appears in tableTitle, author, ISBN visibleAvailability shows correct count
test_borrow_book_and_patron_status	<ul style="list-style-type: none">Add a new bookEnter valid 6-digit patron IDClick Borrow buttonNavigate to Patron Status pageSearch using same patron ID	<ul style="list-style-type: none">Borrow success flash message visibleRedirect to updated CatalogPatron Status page loadsBorrowed book appears under patron's current records
test_return_book_flow	<ul style="list-style-type: none">Add a new bookBorrow it with patron IDNavigate to Return Book pageEnter Patron ID + Book	<ul style="list-style-type: none">Borrow confirmation flash message appearsAfter return, success flash message visibleBook no longer appears

	ID • Submit return form	as borrowed • Available copies increase
test_search_finds_added_book	<ul style="list-style-type: none"> • Add a new book • Navigate to Search page • Enter partial title • Select search type: Title • Click Search 	<ul style="list-style-type: none"> • Search Results table appears • Book with matching title displayed • Author & ISBN visible • Availability shown correctly

Dockerization Process

Step 1:

Ensures project dependencies are defined in requirements.txt

```
Ξ requirements.txt
1  Flask==2.3.3
2  pytest==7.4.2
3  playwright==1.56.0
4
5  |
```

Step 2:

created a Dockerfile which defines the base image, copying files into the container, installing dependencies, exposing port 5000, and starting the flask server

```
👉 Dockerfile > ...
1  FROM python:3.11
2
3  WORKDIR /app
4
5  COPY requirements.txt .
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY . .
9
10 ENV FLASK_APP=app.py
11 ENV FLASK_RUN_HOST=0.0.0.0
12 ENV FLASK_RUN_PORT=5000
13
14 EXPOSE 5000
15
16 CMD ["flask", "run", "--host=0.0.0.0"]
17 |
```

Step 3:

To simplify running the container, I created a docker-compose.yml file. With this addition we can use Docker Compose to start the application with a single command. It also centralizes configuration such as ports and environment variables, which makes the setup easier to maintain. This file allows the project to scale later because new services like a database or test runner can be added and started together without changing any code.

```
docker-compose.yml > {} services > {} web > [] ports
      docker-compose.yml - The Compose specification establishes a standard for the definition of services.
1   services:
  |>Run Service
2     web:
3       build: .
4       ports:
5         - "5000:5000"
6
```

Step 4:

I build and run the container with the command:

Docker compuse up - -build

```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker compose up --build
[+] Building 2.1s (12/12) FINISHED
  => [internal] load local bake definitions
  => => reading from stdin 605B
  => [internal] load build definition from Dockerfile
  => => transferring dockerfile: 341B
  => [internal] load metadata for docker.io/library/python:3.11
  => [internal] load .dockerignore
  => => transferring context: 2B
  => [1/5] FROM docker.io/library/python:3.11@sha256:a3f963d22cc2bd1b0e9a62c8e6f31b3bd3340d554fae08cc312c3359b61a6d7e
  => [internal] load build context
  => => transferring context: 314.75kB
  => CACHED [2/5] WORKDIR /app
  => CACHED [3/5] COPY requirements.txt .
  => CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt
  => [5/5] COPY . .
  => exporting to image
  => => exporting layers
  => => writing image sha256:4f8a8ebc49f717fecf9f7658bde7024a82e6f7e72f059bd444ac4e1715c08f6c
  => => naming to docker.io/library/software_quality_assurance-web
  => resolving provenance for metadata file
[+] Running 2/2
  ✓ software_quality_assurance-web           Built
  ✓ Container software_quality_assurance-web-1 Recreated
Attaching to web-1
web-1 | * Serving Flask app 'app.py'
web-1 | * Debug mode: off
web-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
web-1 | * Running on all addresses (0.0.0.0)
web-1 | * Running on http://127.0.0.1:5000
web-1 | * Running on http://172.19.0.2:5000
web-1 | Press CTRL+C to quit
web-1 | 172.19.0.1 - - [24/Nov/2025 05:35:21] "GET / HTTP/1.1" 302 -
web-1 | 172.19.0.1 - - [24/Nov/2025 05:35:22] "GET /catalog HTTP/1.1" 200 -
web-1 | 172.19.0.1 - - [24/Nov/2025 05:35:22] "GET /favicon.ico HTTP/1.1" 404 -
[ ]
```

Enable Watch

Docker Hub Deployment

Push:

```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker images


| IMAGE                                 | ID           | DISK USAGE | CONTENT SIZE | EXTRA |
|---------------------------------------|--------------|------------|--------------|-------|
| software_quality_assurance-web:latest | c2075c25a123 | 1.88GB     | 486MB        | U     |


(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker tag software_quality_assurance-web:latest dike12/library-app:v1
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker push dike12/library-app:v1
The push refers to repository [docker.io/dike12/library-app]
505fac1b5c3a: Pushed
eae668646f44: Pushed
ff2e6687b6c: Pushed
7c40a3faff76: Pushed
7d045536fc2b: Pushed
6637c0e10955: Pushed
b933556e36b5: Pushed
70bbc8515263: Pushed
b64c4eb60d79: Pushed
20a3f69f2611: Pushed
53c88f1dfcb7: Pushed
0022a4b71fe9: Pushed
v1: digest: sha256:c2075c25a1232556b17f3f6a572c6a5a0b16ab3173da5e32da61873b94cb290c size: 856
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ [ ]
```

Remove local image:

```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker images


| IMAGE                                 | ID           | DISK USAGE | CONTENT SIZE | EXTRA |
|---------------------------------------|--------------|------------|--------------|-------|
| dike12/library-app:v1                 | c2075c25a123 | 1.88GB     | 486MB        | U     |
| software_quality_assurance-web:latest | c2075c25a123 | 1.88GB     | 486MB        | U     |

  


```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker rmi dike12/library-app:v1
Untagged: dike12/library-app:v1
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker images

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
software_quality_assurance-web:latest	c2075c25a123	1.88GB	486MB	U


```


```

Pull:

```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker images


| IMAGE                                 | ID           | DISK USAGE | CONTENT SIZE | EXTRA |
|---------------------------------------|--------------|------------|--------------|-------|
| software_quality_assurance-web:latest | c2075c25a123 | 1.88GB     | 486MB        | U     |

  


```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker pull dike12/library-app:v1
v1: Pulling from dike12/library-app
Digest: sha256:c2075c25a1232556b17f3f6a572c6a5a0b16ab3173da5e32da61873b94cb290c
Status: Downloaded newer image for dike12/library-app:v1
docker.io/dike12/library-app:v1
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker images

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
dike12/library-app:v1	c2075c25a123	1.88GB	486MB	U
software_quality_assurance-web:latest	c2075c25a123	1.88GB	486MB	U


```


```

Run:

```
(venv) kabelodike@fedora:~/Desktop/School_shit/Software_Quality_Assurance$ docker run -p 5000:5000 --name library-app dike12/library-app:v1
 * Serving Flask app 'app.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
172.17.0.1 - - [24/Nov/2025 06:47:04] "GET /catalog HTTP/1.1" 200 -

```

Challenges and Reflections

Throughout this course, I gained a much deeper appreciation for software quality and the discipline required to measure it effectively. One of my biggest challenges was getting the passing badge in the CI/CD pipeline, even though my tests were consistently scoring around 80 percent. It pushed me to refine my assertions, increase coverage, and better understand how automated pipelines evaluate reliability. I also really enjoyed learning about non-functional requirements, especially security-related ones, since cybersecurity is something I'm passionate about. Testing functional requirements was equally valuable because it demonstrated how clear specifications translate into verifiable system behavior. What surprised me most is how directly this knowledge helped in my other course, CISC 322 (Software Architecture), since architectural decisions heavily depend on well-defined functional requirements and NFRs. Overall, the course strengthened both my practical testing skills and my understanding of what high-quality software looks like in real development contexts.