Tyler Graham

Final Project Report

CS5201 Microservice & Cloud Computing

12/15/2024

# Adopt-A-Pet Service!

Table of Contents

[Adopt-A-Pet Service! 1](#_Toc185180261)

[Motivation 2](#_Toc185180262)

[Architecture of the Application 2](#_Toc185180263)

[Diagram of Architecture 3](#_Toc185180264)

[Explanation of Each Service 3](#_Toc185180265)

[Admin Service: 3](#_Toc185180266)

[Pet Search Service 4](#_Toc185180267)

[User Management Service 5](#_Toc185180268)

[Demonstration 6](#_Toc185180269)

[Performance 10](#_Toc185180270)

[Load Testing - Locust 10](#_Toc185180271)

[Setup 10](#_Toc185180272)

[Results 11](#_Toc185180273)

[Discussion 11](#_Toc185180274)

[Digital Ocean Implementation: 12](#_Toc185180275)

[Droplet Overview 12](#_Toc185180276)

[Testing Setup and Configuration 14](#_Toc185180277)

[Python Path Configuration: 14](#_Toc185180278)

[Flask Test Client Setup: 15](#_Toc185180279)

[Test Cases: 15](#_Toc185180280)

[Results 16](#_Toc185180281)

[Project Post-Mortem 16](#_Toc185180282)

[Project Feedback 16](#_Toc185180283)

# Motivation

The Pet Search Application was developed to address a growing need for an easy-to-use platform that connects pet adopters with shelters and available pets. According to recent estimates, there are over 3 million dogs in the United States available for adoption annually, highlighting the critical demand for efficient adoption platforms. By integrating with the Petfinder API, the application simplifies the process of finding pets for adoption, enhances user engagement through personalized search and favoriting features, and allows for robust user management and activity tracking. The project also served as an opportunity to explore multi-service architecture and evaluate system performance under simulated loads.

# Architecture of the Application

The application is structured as a microservices-based system consisting of three core services:

1. Admin Service: Handles user authentication, activity tracking, and event logging.
2. Pet Search Service: Integrates with the Petfinder API to provide pet search functionality and supports favoriting pets.
3. User Management Service: Manages user registration and communicates with the Admin Service to synchronize user data.

The services communicate using RESTful APIs and share event data where required. Each service is independently deployable, ensuring modularity and scalability.

## Diagram of Architecture

A screenshot of a computer

Description automatically generated

Figure 1: Architecture Diagram

## Explanation of Each Service

### Admin Service:

Purpose: Manages user sessions, tracks user-specific activities (e.g., searches and favorites), and logs events from other services.

#### Key Features:

* User authentication via Flask-Login.
* Endpoint for adding users (`/add\_user`).
* Dashboard displaying user activities (`/dashboard`).

#### Flowchart:

A diagram of a software application

Description automatically generated

Figure 2: User Management Flowchart

### Pet Search Service

Purpose: Enables users to search for pets by criteria (e.g., breed, age) and mark pets as favorites.

#### Key Features:

* Integrates with the Petfinder API.
* Tracks and logs search events.
* User-friendly endpoints for search (`/search`) and favoriting pets (`/favorite`).

#### Flowchart: A diagram of a website Description automatically generated

Figure 3: Search Flowchart

### User Management Service

Purpose: Handles user registration and communicates with the Admin Service to ensure user data synchronization.

#### Key Features:

* Registration form (`/register\_page` and `/register`).
* Tracks registration events and forwards them to the Admin Service.
* Simple in-memory user store for testing purposes.

## Demonstration

1. User Registration
   1. Visit <http://localhost:5003/login>
   2. Click “Register”
   3. A screenshot of a login page

      Description automatically generated
   4. Fill out desired email and password
   5. A screenshot of a user registration form

      Description automatically generated
2. Login
   1. Navigate to <http://localhost:5003/login>
   2. Enter in credentials to access search
   3. A screenshot of a login page

      Description automatically generated
3. Pet Search
   1. Use search form to specify pet criteria
   2. A screenshot of a search bar

      Description automatically generated
4. Favorite (optional)
   1. Click Favorite Icon to Save Pet
   2. A screenshot of a search results page

      Description automatically generated
5. Dashboard
   1. Click “Go to dashboard”
   2. A screenshot of a dog

      Description automatically generated
   3. A screenshot of a search engine

      Description automatically generated

## Performance

### Load Testing - Locust

### Setup

* Tests were conducted on the `/register`, `/search`, and `/favorite` endpoints across all services.
* Simulated 100 concurrent users with request rates of up to 150 requests per second.

### Results

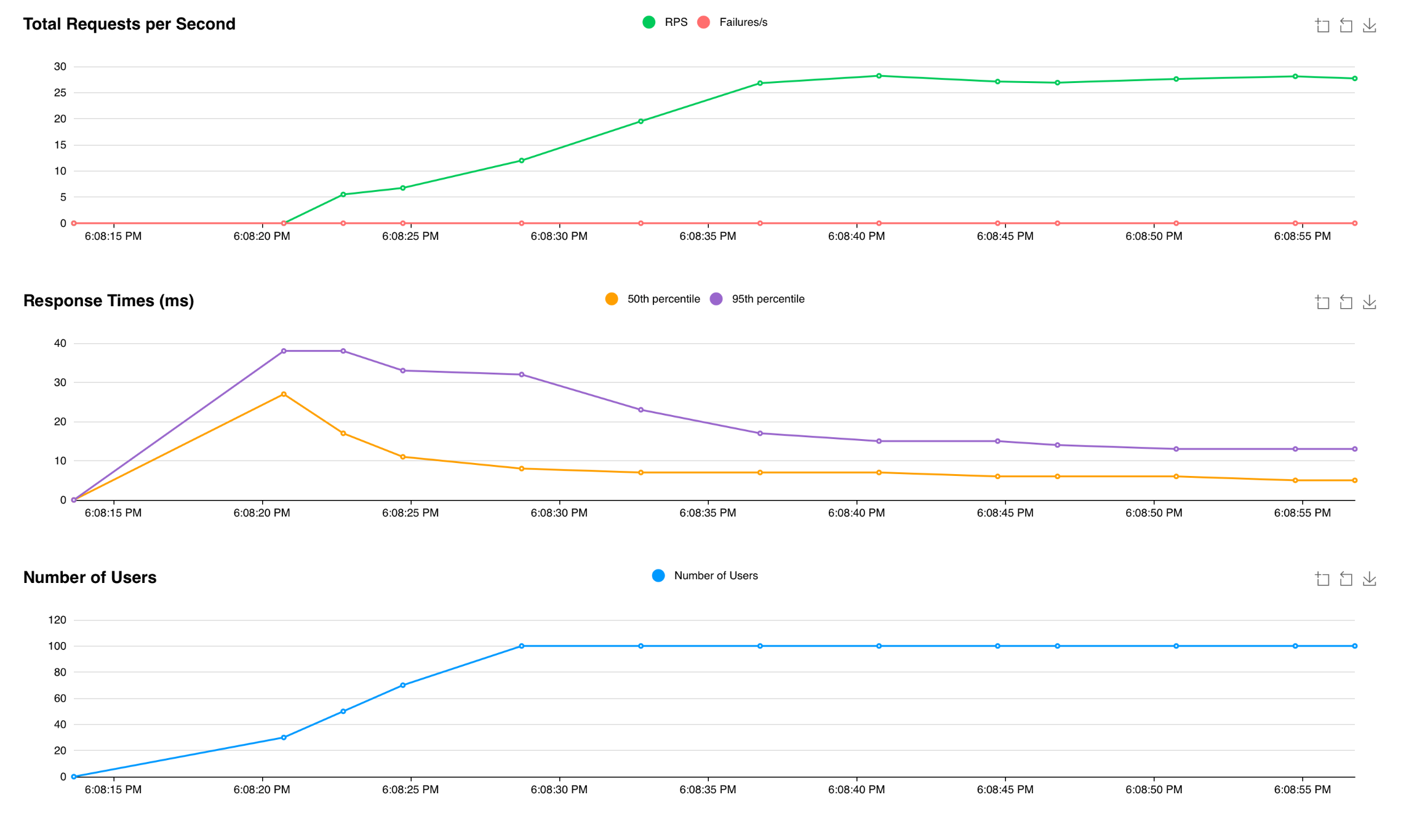


Figure 4: Locust Graphs

A screenshot of a graph

Description automatically generated

Figure 5: Locust Table

### Discussion

The performance test was conducted to evaluate the scalability and response latency of the microservices under increasing concurrent user load. The test results show that as the number of users scaled to approximately 100, the system maintained a consistent **Requests Per Second (RPS),** peaking at **27.7 RPS**. Importantly, there were **no failures recorded** throughout the test, as indicated by the absence of failure spikes in the graph. The number of users scaled steadily, and the system effectively handled the load without disruptions, showcasing its stability and scalability.

Response latency measurements revealed efficient performance across the endpoints. The **median response times** remained low, consistently under **10 ms**, with the **95th percentile** peaking briefly at **40 ms** during the ramp-up phase before stabilizing to approximately **15-20 ms**. Endpoint-specific metrics confirm this trend: key endpoints like /login, /dashboard, and /register\_page maintained low latencies even under load, with no failures observed. The aggregated metrics reflect this efficiency, with a total of **934 requests**, a median latency of **6 ms**, and an average response time of **7.81 ms**.

Overall, the microservices demonstrated excellent scalability, reliability, and performance under load. The system effectively handled concurrent requests with no failures and minimal response latency degradation. This highlights its robustness and readiness for production use. Further optimization efforts could focus on minimizing the minor latency spikes observed during the initial load ramp-up phase.

## Digital Ocean Implementation:

**A DigitalOcean Droplet,** which is a virtual private server used to host the application. Here's a breakdown of the DigitalOcean droplet to host the service:

### **Droplet Overview**

#### Droplet Name: ubuntu-s-1vcpu-1gb-35gb-intel-nyc1-01

* **1 vCPU**: One virtual CPU core.
* **1 GB Memory**: RAM available for running applications.
* **35 GB Disk**: Storage allocated for your service.
* **Region**: NYC1 data center (New York City).
* **OS**: Ubuntu 24.10 x64.
* **IPv4**: 192.241.128.137 — Public IP for accessing your service from the internet.

A screenshot of a graph

Description automatically generated

Figure 6: Digital Ocean Graphs

A black screen with white text

Description automatically generated

Figure 7: SSH into Digital Ocean Droplet

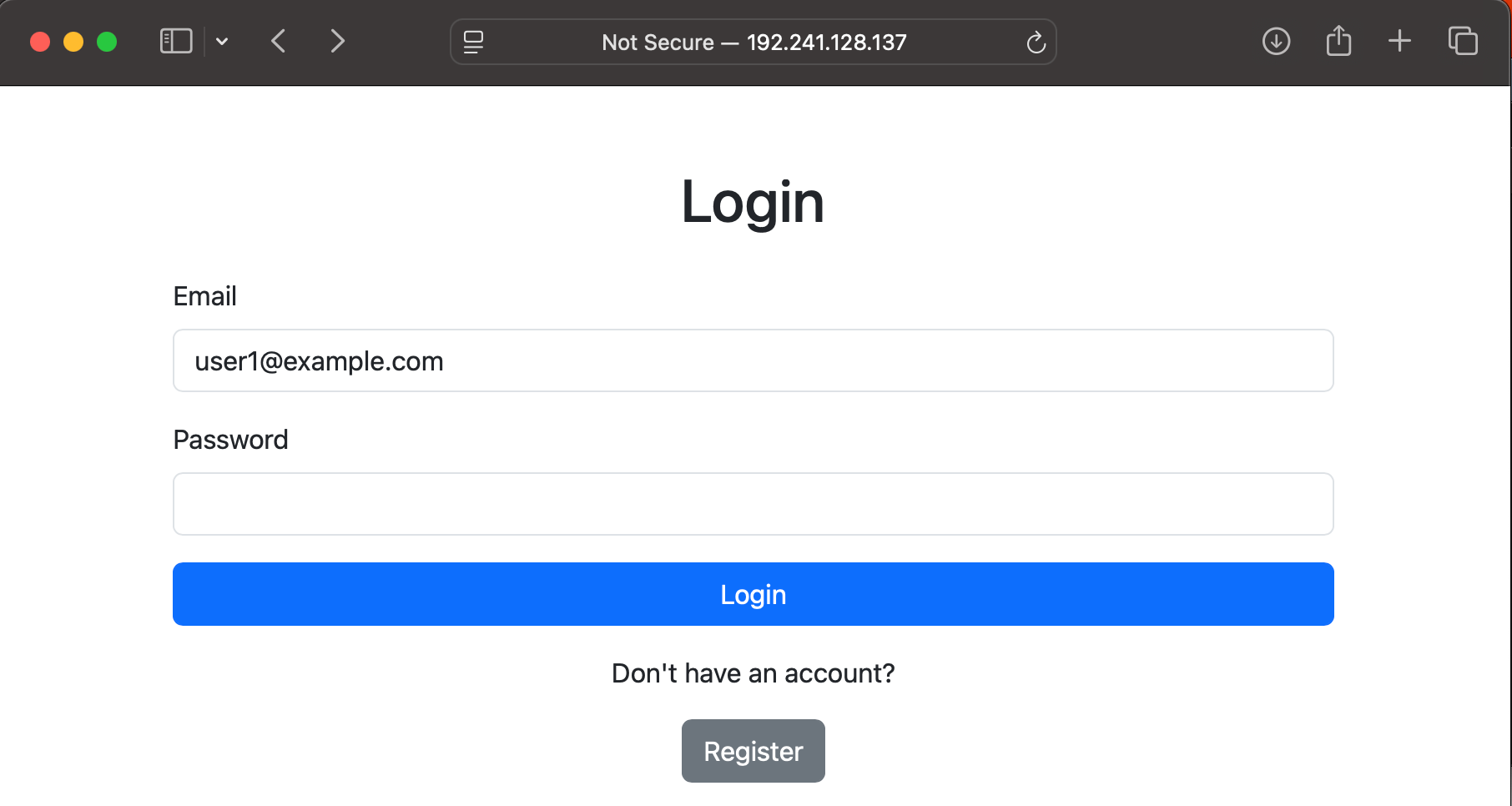


Figure 8: Login From External IP/Digital Ocean Droplet

## Testing Setup and Configuration

To ensure proper testing of the pet\_search\_service application, the following setup was implemented. The sys.path manipulation ensures that the correct project modules are imported, avoiding conflicts with similarly named packages in the virtual environment.

### Python Path Configuration:

The following lines add the project root directory to the Python path. This step ensures that the test file can access the correct pet\_search\_service module located two directories above the test file.

A screen shot of a computer code

Description automatically generated

Here, os.path.abspath resolves the absolute path of the project root, and sys.path.insert prioritizes this path for module resolution. This prevents conflicts with installed packages that may share the same name as project modules.

### Flask Test Client Setup:

A pytest fixture named client was created to initialize Flask's test\_client(). This provides a lightweight testing environment for simulating HTTP requests without needing to run the full application server:

A screen shot of a computer code

Description automatically generated

### Test Cases:

#### test\_search\_menu:

This test verifies that the root endpoint (/) returns a 200 OK status code and contains the expected content, such as the keyword "Search":

A computer screen shot of text

Description automatically generated

#### test\_search\_pets:

This test mocks the get\_petfinder\_token function using pytest’s monkeypatch utility. The mock ensures that the API token retrieval does not depend on external factors during testing. The test then validates the /search route for correct behavior:

A computer code with text

Description automatically generated

By combining path manipulation, the Flask test client, and monkeypatching, the test suite ensures accurate and isolated testing of the application's core functionality. This structure allows for scalable and maintainable tests as the application grows.

### Results

A screen shot of a computer

Description automatically generated

## Project Post-Mortem

The project successfully implemented a modular microservices architecture that integrates with the Petfinder API. Basic scalability and performance benchmarks were achieved, and the application provided a user-friendly interface for pet searches and user management. This demonstrated the viability of a distributed system design for the application’s objectives.

I enjoyed working with Petfinder API and spending time seeing how other companies use their API and reading about all the features available. If given more time, I would love to implement notifications to user with Kafka.

I spent a lot of time re-architecting my microservices to be compatible with each other as I added more features. It really showed the benefits of mocking out and designing your entire system with all the features in mind and the challenges of scope creep and continuous improvement on the microservice architectures.

## Project Feedback

I faced significant challenges implementing Kafka into my codebase and deploying it on DigitalOcean. Despite spending numerous hours debugging and configuring it, I ultimately decided to remove it from the project to focus on completing a functional and deliverable system. While disappointing, this decision allowed me to meet the core requirements and ensure a stable application.

The base 85% of the project requirements were relatively straightforward to implement, thanks to the foundation provided by the labs and my prior experience with similar concepts. However, the additional 15%—including advanced features and optimizations—proved to be disproportionately time-consuming, requiring more effort than the rest of the project combined.