Tainerz – Minimal Docker Manager Web Application

Developed by Shuba S and Designed by Sanjana M

Overview

Tainerz is a lightweight web application built with Flask that provides a user-friendly interface for managing Docker containers and images. The application features user authentication using MySQL database, real-time Docker engine integration, and a responsive web interface for monitoring and controlling Docker resources.

Features

- **User Authentication**: Secure signup/login system with MySQL database
- **Docker Images Management**: View all Docker images with details (ID, tags, size)
- **Container Management**: List all containers with start/stop functionality
- **Real-time Status**: Live container status monitoring (running/exited)
- **Responsive Design**: Mobile-friendly interface with modern CSS styling
- **Session Management**: Secure user sessions with Flask sessions

Project Structure

```
Tainerz/
├── app.py                  # Main Flask application with authentication
├── templates/              # HTML templates
│   ├── layout.html         # Base template
│   ├── login.html          # Login/signup page
│   ├── home.html           # Dashboard home
│   ├── docker_images.html     # Images listing page
│   └── docker_containers.html # Containers management page
├── static/                 # Static assets
│   ├── css/
│   │   └── static.css      # Application styles
│   └── js/
│       └── script.js       # Frontend JavaScript
└── MySQL Database          # Remote MySQL database
```

Technologies Used

Backend

- **Python 3.x**: Core programming language for server-side logic
- **Flask**: Lightweight web framework for routing and templating
- **MySQL**: Relational database for user management and authentication
- **Docker SDK for Python**: Docker engine integration for container management

Frontend

- **HTML5**: Semantic markup with Jinja2 templating
- **CSS3**: Modern styling with flexbox, gradients, and responsive design
- **JavaScript**: Dynamic UI interactions and form toggles

Installation & Setup

Prerequisites

- Python 3.7+
- Docker installed and running
- MySQL server running
- pip package manager

Required Dependencies

```
Flask==2.3.2
mysql-connector-python==8.1.0
docker==6.1.3
```

Installation Steps

1. Clone the repository

```
git clone <repository-url>
cd Tainerz
```

2. Install dependencies

```
pip install flask mysql-connector-python docker
```

3. Configure MySQL database connection in app.py

```
# Update MySQL connection parameters
```

```
        mysql_config = {
        'host': 'localhost',
        'user': 'your_username',
        'password': 'your_password',
        'database': 'tainerz_db'
        }
```

4. Initialize the database

```
CREATE DATABASE tainerz_db;
USE tainerz_db;
CREATE TABLE users (
 id INT AUTO_INCREMENT PRIMARY KEY,
 email VARCHAR(255) NOT NULL,
 username VARCHAR(100) UNIQUE NOT NULL,
 password VARCHAR(255) NOT NULL
  ;
```

5. Run the application

```
python app.py
```

Access the application Open your browser and navigate to http://localhost:5000

Code Explanation

Authentication System (in app.py)

```
@app.route('/login', methods=['GET', 'POST'])
def login():
   # Handles both signup and login functionality
   # Uses MySQL to verify credentials and create new users
   # Stores username in Flask session upon successful authentication
```

This function manages user authentication by connecting to MySQL database, validating credentials for login, and creating new user accounts for signup. It uses parameterized queries to prevent SQL injection and maintains secure session management.

Docker Images Listing

```
@app.route('/images')
def list_docker_images():
   client = docker.from_env()
   images = client.images.list()
```

```
    image_data = [{
        'id': img.id[:12],
        'tags': img.tags or ['<none>'],
        'size': f"{img.attrs['Size'] / (1024 * 1024):.2f} MB"
    } for img in images]
    return render_template('docker_images.html', images=image_data)
```

This endpoint connects to the local Docker daemon using Docker SDK, fetches all available images with their metadata, and formats the size in MB. It extracts essential information like shortened IDs, tags, and file sizes for display in a clean table format.

Container Management

```
@app.route('/containers')
def list_containers():
    client = docker.from_env()
    containers = client.containers.list(all=True)
    container_data = [{
        'id': c.id[:12],
        'name': c.name,
        'status': c.status
    } for c in containers]
    return render_template('docker_containers.html', containers=container_data)
```

This function retrieves all Docker containers regardless of their current state (running, stopped, exited), extracts key information including container ID, name, and status. It passes this data to the template which renders interactive buttons for container management operations.

Container Control Actions

```
@app.route('/start_container/<container_id>')
@app.route('/stop_container/<container_id>')
def start/stop_container():
    # Uses Docker SDK to control container lifecycle
    # Provides feedback on operation success/failure
    # Redirects back to containers page with updated status
```

These endpoints handle container lifecycle operations by accepting container ID as parameter, executing the requested action via Docker SDK, and providing user feedback. They include error handling and redirect users back to the containers page with updated status information.

API Endpoints

Method	Endpoint	Description	Authentication Required
GET/POST	/login	User authentication (signup/login)	No
GET	/home	Dashboard home	Yes
GET	/images	List Docker images	Yes
GET	/containers	List containers	Yes
GET	/start_container/<id>	Start a container	Yes
GET	/stop_container/<id>	Stop a container	Yes
POST	/logout	User logout	Yes

Database Schema

Users Table (MySQL)

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   email VARCHAR(255) NOT NULL,
   username VARCHAR(100) UNIQUE NOT NULL,
   password VARCHAR(255) NOT NULL
);
```

Fields Description:

- id: Auto-incrementing primary key for unique user identification
- email: User's email address for account recovery and notifications

- username: Unique identifier for login authentication
- password: User password (consider implementing hashing for security)

Frontend Components

Docker Images Table (docker_images.html)

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Tags</th>
      <th>Size</th>
    </tr>
  </thead>
  <tbody>
    {% for image in images %}
    <tr>
      <td>{{ image.id }}</td>
      <td>{{ image.tags|join(', ') }}</td>
      <td>{{ image.size }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

This template renders Docker images in a clean table format, displaying truncated image IDs, comma-separated tags, and formatted file sizes. It uses Jinja2 templating to iterate through image data passed from the backend.

Docker Containers Table (docker_containers.html)

```
<table class="docker-container-table">
  <thead>
    <tr>
      <th>Container ID</th>
      <th>Name</th>
      <th>Status</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {% for container in containers %}
    <tr>
      <td>{{ container.id }}</td>
      <td>{{ container.name }}</td>
      <td>{{ container.status }}</td>
      <td>
        <a href="{{ url_for('start_container', container_id=container.name) }}">
          <button>Start</button>
        </a>
        <a href="{{ url_for('stop_container', container_id=container.name) }}">
          <button>Stop</button>
        </a>
      </td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

This component displays containers with their current status and provides interactive Start/Stop buttons. It uses Flask's url_for function to generate proper routes for container control actions and applies CSS styling for visual appeal.

Application Flow & Output

Step 1: Initial Access - Signup/Login Page

Input: Navigate to http://localhost:5000 **Output**: Login page with toggle between Sign In and Sign Up forms **Screenshot Reference**: [Image 1 - Login/Signup Interface]

The initial page presents a clean authentication interface with JavaScript-powered form switching. Users can toggle between existing account login and new account registration without page refresh.

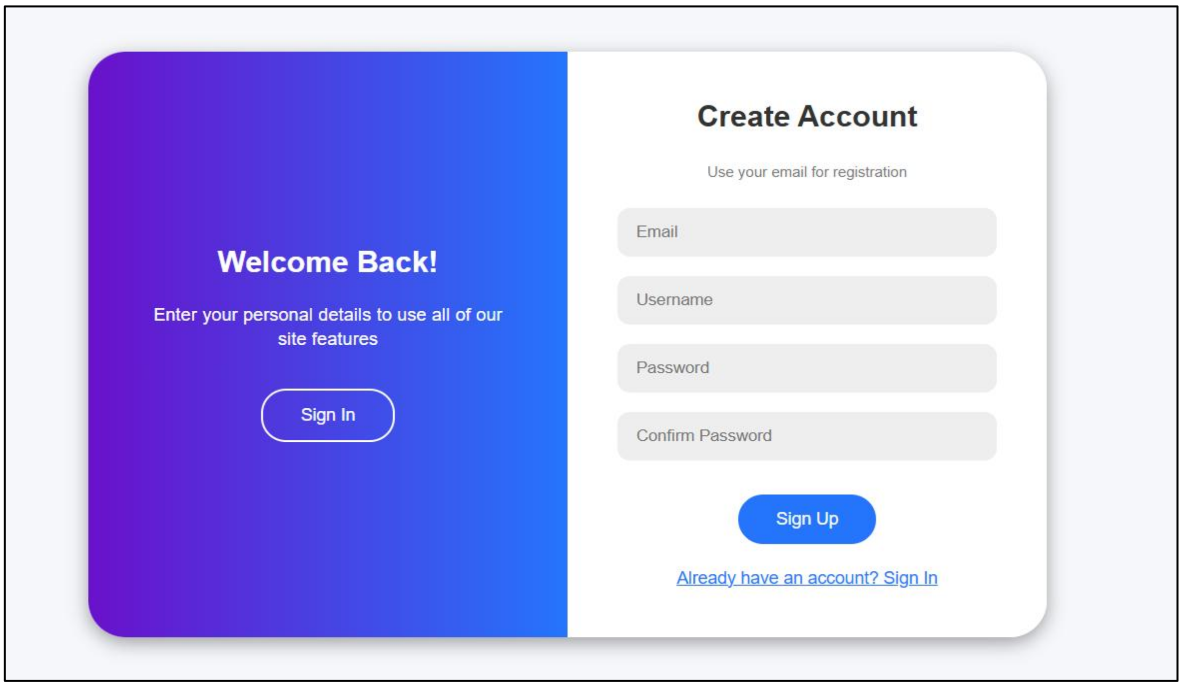

Welcome Back!
Enter your personal details to use all of our site features
Sign In
Create Account
Use your email for registration
Email
Username
Password
Confirm Password
Sign Up
Already have an account? Sign In

Image 1 - Signup Interface

Step 2: User Registration (Signup)

Input: Fill signup form (email, username, password) and submit **Output**: User account created in MySQL database, automatic redirect to login.

When users complete the signup form, the application validates input, creates a new record in the MySQL users table, and provides feedback on successful registration before redirecting to login.

Step 3: User Authentication (Sign-in)

Input: Enter valid username and password credentials **Output**: Session established, redirect to home dashboard **Screenshot Reference**: [Image 2 – Sign-in Interface]

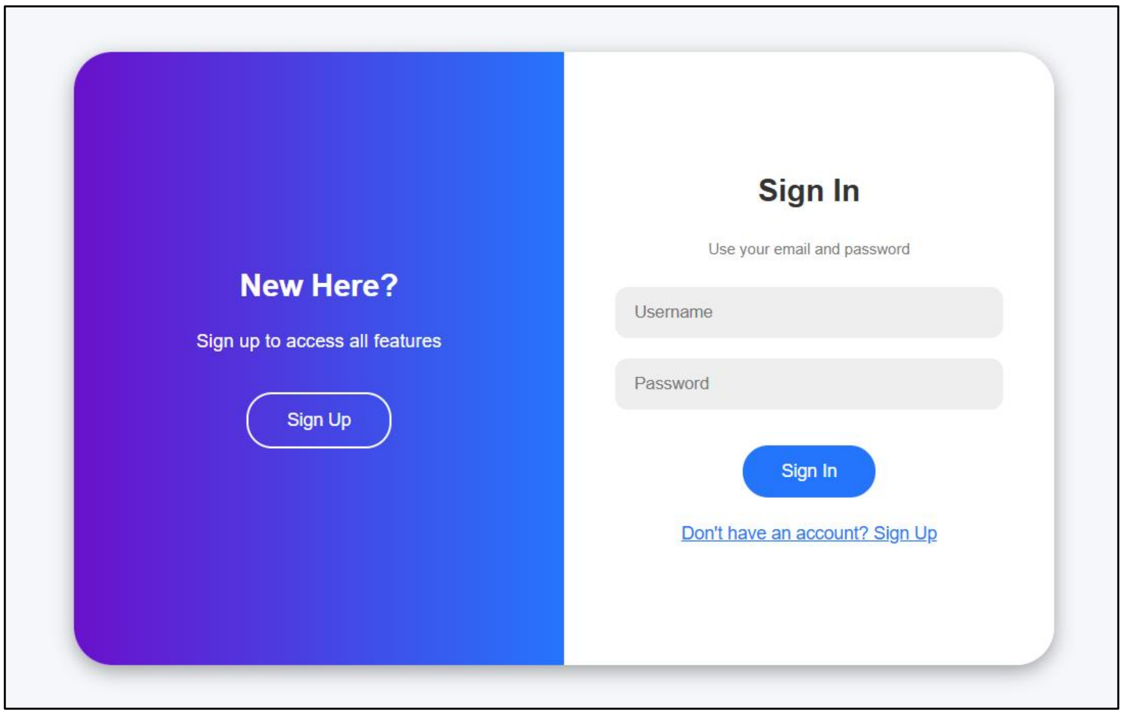

New Here?
Sign up to access all features
Sign Up
Sign In
Use your email and password
Username
Password
Sign In
Don't have an account? Sign Up

Image 2 – Sign-in Interface

Successful authentication creates a secure Flask session, stores the username for future requests, and redirects to the main dashboard interface with navigation options.

Step 4: Dashboard Home Page

Input: Authenticated user accesses /home **Output**: Navigation options for Images and Containers management **Screenshot Reference**: [Image 3 - Home Dashboard Navigation]

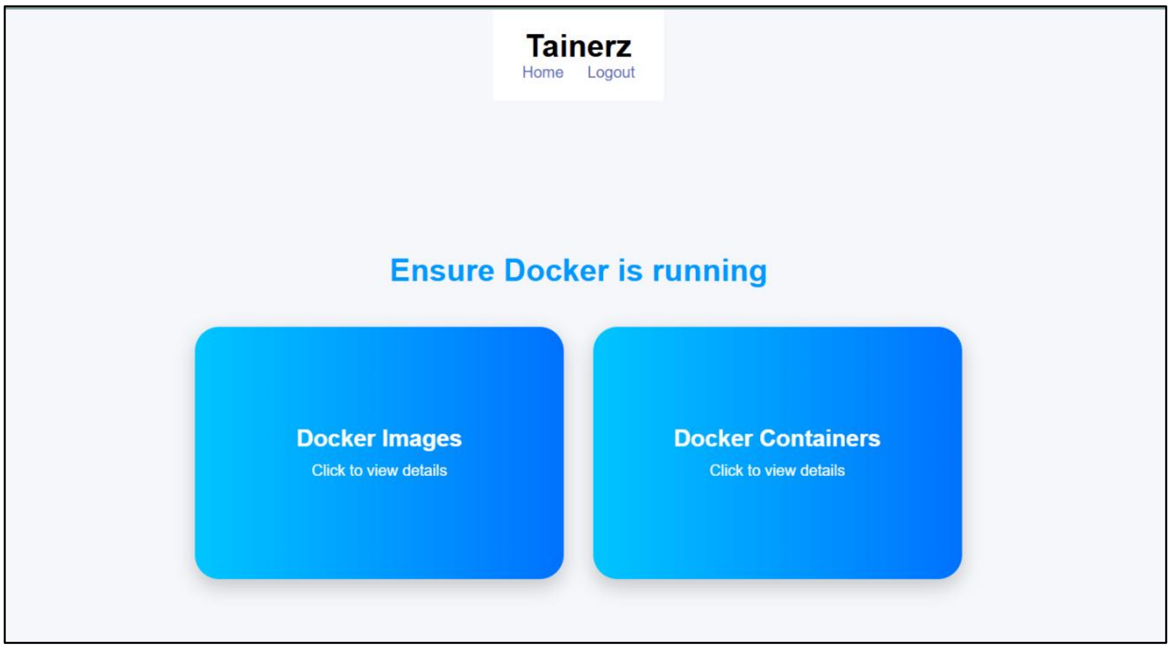
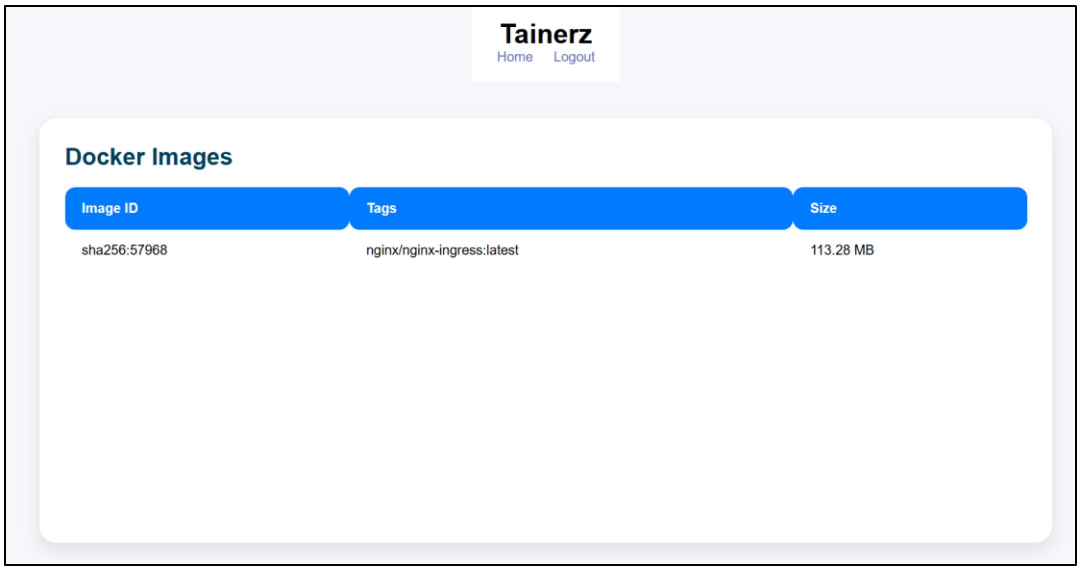


Image 3 - Home Dashboard Navigation

The home page serves as the main navigation hub, presenting clean buttons or links to access Docker images listing and container management sections with consistent styling.

Step 5: Docker Images Management

Input: Click on "Images" navigation option **Output**: Table displaying all Docker images with ID, tags, and size information **Screenshot Reference**: [Image 4 - Docker Images List Table]



Docker Images

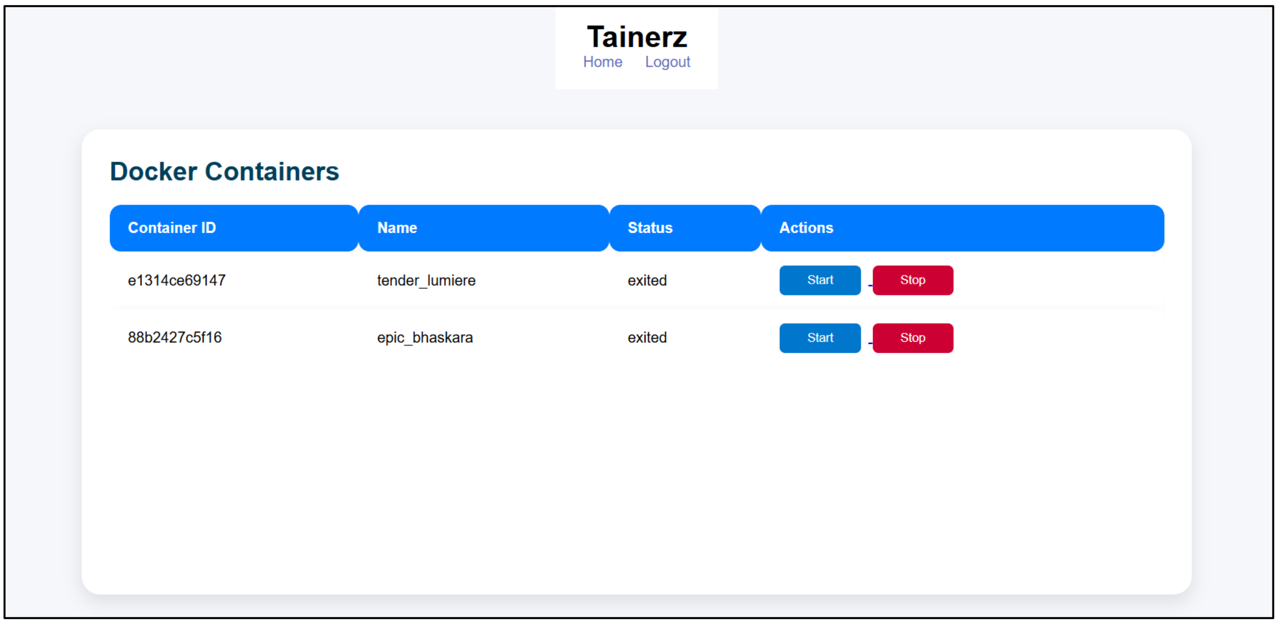
Image ID	Tags	Size
sha256:57968	nginx/nginx-ingress:latest	113.28 MB

Image 4 - Docker Images List Table

The images page connects to Docker daemon, retrieves all available images, and displays them in a formatted table showing shortened IDs, associated tags, and file sizes in MB for easy comparison.

Step 6: Container Management Interface

Input: Click on "Containers" navigation option **Output**: Interactive table with container details and action buttons **Screenshot Reference**: [Image 5 - Containers Management Interface]



Docker Containers

Container ID	Name	Status	Actions
e1314ce69147	tender_lumiere	exited	Start Stop
88b2427c5f16	epic_bhaskara	exited	Start Stop

Image 5 - Containers Management Interface

This page shows all containers (running and stopped) with their current status, names, and provides Start/Stop buttons for lifecycle management with gradient styling.

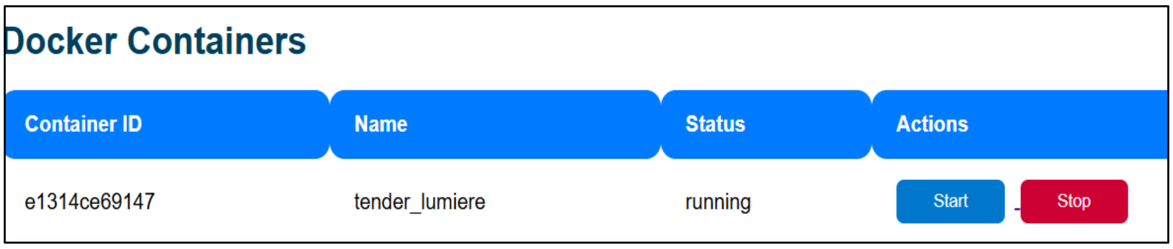
Step 7: Container Status Display

Input: View containers list **Output**: Real-time status indicators (Running/Exited/Created)

Each container row displays current status with color-coded indicators - different colors for running containers, exited containers, and other states for quick visual identification.

Step 8: Container Control Actions

Input: Click Start button on stopped container **Output**: Container starts, status updates to "Running", page refreshes **Screenshot Reference**: [Image 6 - Container Start Action Result]



Docker Containers

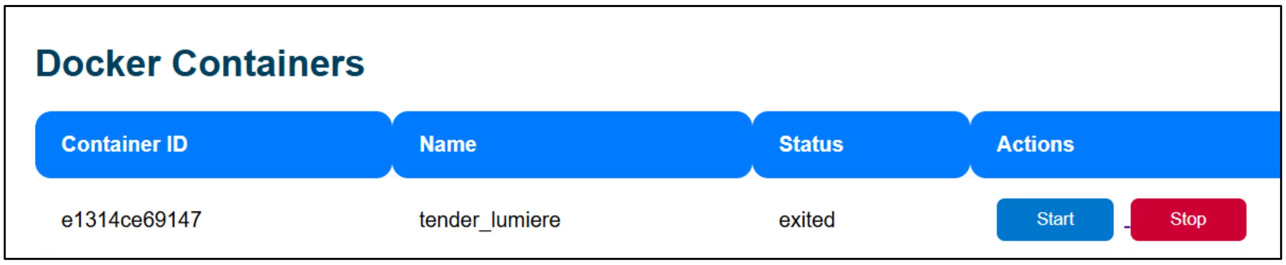
Container ID	Name	Status	Actions
e1314ce69147	tender_lumiere	running	Start Stop

Image 6 - Container Start Action Result

When users click action buttons, the application sends commands to Docker daemon, executes the operation, and refreshes the page to show updated container status with immediate feedback.

Step 9: Stop Container Operation

Input: Click Stop button on running container **Output**: Container stops, status changes to "Exited", confirmation message **Screenshot Reference**: [Image 7 - Container Stop Confirmation]



Docker Containers

Container ID	Name	Status	Actions
e1314ce69147	tender_lumiere	exited	Start Stop

Image 7 - Container Stop Confirmation

The stop operation immediately terminates the selected container, updates the status display, and provides visual feedback confirming the successful operation completion.

Styling & UI

The application uses modern CSS with the following design elements:

Container Table Styling

```
.docker-container-table tbody tr {
  background: linear-gradient(to right, #00c9ff, #92fe9d);
  color: white;
  border-radius: 12px;
}
```

This CSS creates visually appealing gradient backgrounds for container rows, uses modern border-radius for rounded corners, and ensures good contrast with white text on gradient backgrounds.

Key Design Features

- **Gradient Backgrounds**: Modern blue-to-green gradients for visual appeal
- **Rounded Corners**: 12px border-radius for contemporary look
- **Responsive Layout**: Flexbox-based responsive design for mobile compatibility
- **Color Scheme**: Consistent blue and green theme throughout the application
- **Interactive Elements**: Hover effects and button styling for better user experience
- **Typography**: Clean, readable fonts with proper spacing and hierarchy

Current Security Issues

- **Plain Text Passwords**: Passwords are stored without hashing in MySQL
- **SQL Injection**: Basic parameterized queries used, but limited validation
- **Session Security**: Basic Flask sessions without additional security measures
- **Docker Access**: Direct Docker daemon access without permission restrictions

Recommended Security Improvements

1. **Password Hashing**: Implement bcrypt or similar for secure password storage
2. **Input Validation**: Add comprehensive server-side validation and sanitization
3. **CSRF Protection**: Implement CSRF tokens for form submissions
4. **HTTPS Deployment**: Use SSL/TLS in production environments
5. **Rate Limiting**: Add rate limiting for login attempts and API calls
6. **Docker Security**: Run with limited Docker permissions and user isolation
7. **Session Security**: Implement secure session configuration with timeout
8. **SQL Security**: Add additional input validation and error handling

Future Enhancements

Planned Features

- **Container Logs**: Real-time log viewing with streaming updates and search functionality
- **Resource Monitoring**: CPU and memory usage graphs, statistics, and historical data
- **User Roles**: Admin privileges, user access control, and permission management
- **Activity Logs**: Comprehensive audit trail for all user actions and system events
- **Search & Filter**: Advanced filtering capabilities for large container and image lists
- **Image Management**: Pull new images, push to registries, and remove unused images
- **Network Management**: Docker network configuration and management interface
- **Volume Management**: Docker volume administration and data persistence
- **Container Creation**: Web interface for creating new containers with custom configurations
- **Backup & Restore**: Database backup functionality and container state snapshots

Technical Improvements

- **API Documentation**: Swagger/OpenAPI documentation for RESTful endpoints
- **Unit Tests**: Comprehensive test coverage with automated testing pipeline
- **Docker Compose**: Containerized deployment with multi-service configuration
- **Environment Configuration**: Environment-based configuration management for different deployments
- **Structured Logging**: Application logging with log levels and rotation

- **Error Handling**: Improved error handling with user-friendly messages and graceful failure recovery
- **Performance Optimization**: Caching mechanisms and database query optimization
- **Monitoring**: Application performance monitoring and health checks

Troubleshooting

Common Issues

Docker Connection Error

Error: Could not connect to Docker daemon

- Ensure Docker is running: sudo systemctl start docker
- Check Docker socket permissions: sudo chmod 666 /var/run/docker.sock
- Verify Docker SDK installation: pip install docker
- Add user to docker group: sudo usermod -aG docker $USER

MySQL Connection Error

Error: Unable to connect to MySQL database

- Ensure MySQL server is running: sudo systemctl start mysql
- Check database credentials in app.py configuration
- Verify database exists: SHOW DATABASES;
- Test connection: mysql -u username -p database_name

Permission Denied

Error: Permission denied accessing Docker

- Add user to docker group: sudo usermod -aG docker $USER
- Restart terminal session or logout/login
- Verify Docker permissions: docker ps
- Check Docker service status: sudo systemctl status docker

Module Import Errors

Error: No module named 'flask' or 'docker'

- Install required dependencies: pip install flask mysql-connector-python docker
- Check Python environment and virtual environment activation
- Verify pip installation: pip --version

MySQL Authentication Errors

Error: Access denied for user

- Check MySQL user credentials and permissions
- Grant necessary privileges: GRANT ALL PRIVILEGES ON tainerz_db.* TO 'username'@'localhost';
- Flush privileges: FLUSH PRIVILEGES;

Debug Mode

Enable Flask debug mode for development:

```
if __name__ == '__main__':
    app.run(debug=True)
```