## Step-by-Step Guide: Supplier Service with MySQL, Docker, and Kubernetes

This guide provides a detailed, step-by-step process for deploying the Supplier Service application using MySQL, Docker, and a Kubernetes cluster.

### Prerequisites

Before you begin, ensure you have the following installed and configured:

* **Docker:** For containerizing the application.
* **kubectl:** The Kubernetes command-line tool.
* **A Kubernetes Cluster:** This could be Minikube (for local development), or a cloud-based Kubernetes service (like Google Kubernetes Engine, Amazon EKS, or Azure Kubernetes Service).
* **Python 3.x:** For running the Supplier Service application.
* **MySQL:** (If you plan to run MySQL outside of Kubernetes, which is not covered in this guide.)

### Step 1: Create the Supplier Service Application (Python/Flask)

If you haven't already, create the supplier\_service.py file with the following code:

# supplier\_service.py  
from flask import Flask, jsonify, request  
from flask\_mysqldb import MySQL  
import logging  
  
app = Flask(\_\_name\_\_)  
  
# Configuration - Load from environment variables, good for Docker/K8s  
app.config['MYSQL\_HOST'] = 'mysql' # IMPORTANT: Use 'mysql' service name in Kubernetes  
app.config['MYSQL\_USER'] = 'root'  
app.config['MYSQL\_PASSWORD'] = 'password' # Change this! Use a Kubernetes secret.  
app.config['MYSQL\_DB'] = 'suppliers\_db'  
app.config['MYSQL\_CURSORCLASS'] = 'DictCursor'  
  
mysql = MySQL(app)  
  
# Set up logging  
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')  
logger = logging.getLogger(\_\_name\_\_)  
  
# Function to initialize database - create table if not exists.  
def init\_db():  
 try:  
 conn = mysql.connection  
 cursor = conn.cursor()  
 cursor.execute("""  
 CREATE TABLE IF NOT EXISTS suppliers (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 name VARCHAR(255) NOT NULL,  
 contact\_person VARCHAR(255),  
 email VARCHAR(255),  
 phone VARCHAR(20),  
 address VARCHAR(255)  
 )  
 """)  
 conn.commit()  
 cursor.close()  
 logger.info("Database table 'suppliers' created or already exists.")  
 except Exception as e:  
 logger.error(f"Error initializing database: {e}")  
  
# Initialize the database when the app starts  
with app.app\_context():  
 init\_db()  
  
def get\_db\_connection():  
 """  
 Helper function to get a database connection.  
 """  
 try:  
 if not mysql.connection.is\_connected():  
 mysql.connection.connect() # Re-establish the connection  
 return mysql.connection  
 except Exception as e:  
 logger.error(f"Error getting database connection: {e}")  
 return None  
  
def handle\_db\_error(e):  
 """  
 Helper function to handle database errors.  
 """  
 logger.error(f"Database error: {e}")  
 return jsonify({"error": "Database error"}), 500  
  
  
@app.route('/suppliers', methods=['GET'])  
def get\_suppliers():  
 try:  
 conn = get\_db\_connection()  
 if not conn:  
 return jsonify({"error": "Failed to connect to database"}), 500  
 cursor = conn.cursor()  
 cursor.execute("SELECT \* FROM suppliers")  
 suppliers = cursor.fetchall()  
 cursor.close()  
 return jsonify(suppliers), 200  
 except Exception as e:  
 return handle\_db\_error(e)  
  
@app.route('/suppliers/<int:id>', methods=['GET'])  
def get\_supplier(id):  
 try:  
 conn = get\_db\_connection()  
 if not conn:  
 return jsonify({"error": "Failed to connect to database"}), 500  
 cursor = conn.cursor()  
 cursor.execute("SELECT \* FROM suppliers WHERE id = %s", (id,))  
 supplier = cursor.fetchone()  
 cursor.close()  
 if supplier:  
 return jsonify(supplier), 200  
 else:  
 return jsonify({"error": "Supplier not found"}), 404  
 except Exception as e:  
 return handle\_db\_error(e)  
  
@app.route('/suppliers', methods=['POST'])  
def create\_supplier():  
 try:  
 data = request.get\_json()  
 if not data:  
 return jsonify({"error": "Invalid input. Request body must be JSON."}), 400  
 required\_fields = ['name']  
 for field in required\_fields:  
 if field not in data:  
 return jsonify({"error": f"Missing required field: {field}"}), 400  
 conn = get\_db\_connection()  
 if not conn:  
 return jsonify({"error": "Failed to connect to database"}), 500  
 cursor = conn.cursor()  
 cursor.execute("""  
 INSERT INTO suppliers (name, contact\_person, email, phone, address)  
 VALUES (%s, %s, %s, %s, %s)  
 """, (data['name'], data.get('contact\_person'), data.get('email'), data.get('phone'), data.get('address')))  
 conn.commit()  
 cursor.execute("SELECT LAST\_INSERT\_ID()")  
 new\_supplier\_id = cursor.fetchone()['LAST\_INSERT\_ID()']  
 cursor.close()  
 return jsonify({"id": new\_supplier\_id, "message": "Supplier created successfully"}), 201  
 except Exception as e:  
 return handle\_db\_error(e)  
  
@app.route('/suppliers/<int:id>', methods=['PUT'])  
def update\_supplier(id):  
 try:  
 data = request.get\_json()  
 if not data:  
 return jsonify({"error": "Invalid input. Request body must be JSON."}), 400  
 conn = get\_db\_connection()  
 if not conn:  
 return jsonify({"error": "Failed to connect to database"}), 500  
 cursor = conn.cursor()  
 cursor.execute("""  
 UPDATE suppliers SET name=%s, contact\_person=%s, email=%s, phone=%s, address=%s  
 WHERE id=%s  
 """, (data.get('name'), data.get('contact\_person'), data.get('email'), data.get('phone'), data.get('address'), id))  
 conn.commit()  
 cursor.close()  
 return jsonify({"message": "Supplier updated successfully"}), 200  
 except Exception as e:  
 return handle\_db\_error(e)  
  
@app.route('/suppliers/<int:id>', methods=['DELETE'])  
def delete\_supplier(id):  
 try:  
 conn = get\_db\_connection()  
 if not conn:  
 return jsonify({"error": "Failed to connect to database"}), 500  
 cursor = conn.cursor()  
 cursor.execute("DELETE FROM suppliers WHERE id=%s", (id,))  
 conn.commit()  
 cursor.close()  
 return jsonify({"message": "Supplier deleted successfully"}), 200  
 except Exception as e:  
 return handle\_db\_error(e)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True, host='0.0.0.0', port=5000)

### Step 2: Create a Dockerfile

Create a file named Dockerfile in the same directory as supplier\_service.py with the following content:

# Dockerfile  
FROM python:3.9-slim-buster  
  
WORKDIR /app  
  
# Copy requirements.txt first to leverage Docker cache  
COPY requirements.txt requirements.txt  
RUN pip install -r requirements.txt  
  
COPY . /app  
  
# Set environment variables for MySQL connection  
ENV MYSQL\_HOST=mysql # IMPORTANT: Use the service name  
ENV MYSQL\_USER=root  
ENV MYSQL\_PASSWORD=password # Change this! Use a Kubernetes secret.  
ENV MYSQL\_DB=suppliers\_db  
  
CMD ["python", "supplier\_service.py"]

* **requirements.txt**: Create a requirements.txt file:  
  Flask  
  Flask-MySQLdb

### Step 3: Create Kubernetes Deployment and Service Files

Create a file named k8s-all-in-one.yaml (or any name you prefer) with the following Kubernetes configurations:

# k8s-all-in-one.yaml  
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: mysql-pv-claim  
spec:  
 storageClassName: standard # Or your desired storage class  
 accessModes:  
 - ReadWriteOnce  
 resources:  
 requests:  
 storage: 1Gi # Adjust as needed  
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: mysql-deployment  
 labels:  
 app: mysql  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: mysql  
 template:  
 metadata:  
 labels:  
 app: mysql  
 spec:  
 containers:  
 - name: mysql  
 image: mysql:5.7 # Or a specific version  
 env:  
 - name: MYSQL\_ROOT\_PASSWORD  
 value: password # Change this! Use a secret in production.  
 - name: MYSQL\_DATABASE  
 value: suppliers\_db  
 ports:  
 - containerPort: 3306  
 volumeMounts: # Persist data  
 - name: mysql-data  
 mountPath: /var/lib/mysql  
 volumes:  
 - name: mysql-data  
 persistentVolumeClaim:  
 claimName: mysql-pv-claim  
---  
apiVersion: v1  
kind: Service  
metadata:  
 name: mysql  
 labels:  
 app: mysql  
spec:  
 ports:  
 - port: 3306  
 name: mysql  
 selector:  
 app: mysql  
 type: ClusterIP # Important: ClusterIP for internal access  
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: supplier-service-deployment  
 labels:  
 app: supplier-service  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: supplier-service  
 template:  
 metadata:  
 labels:  
 app: supplier-service  
 spec:  
 containers:  
 - name: supplier-service  
 image: supplier-service:latest # Change this after building  
 ports:  
 - containerPort: 5000  
 env:  
 - name: MYSQL\_HOST  
 value: mysql # Use the service name of the MySQL service.  
 - name: MYSQL\_USER  
 value: root  
 - name: MYSQL\_PASSWORD  
 value: password # Change this! Use a Kubernetes secret.  
 - name: MYSQL\_DB  
 value: suppliers\_db  
 livenessProbe: # Add these probes  
 httpGet:  
 path: /suppliers # Change this if you have a health check  
 port: 5000  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 readinessProbe:  
 httpGet:  
 path: /suppliers  
 port: 5000  
 initialDelaySeconds: 5  
 periodSeconds: 10  
---  
apiVersion: v1  
kind: Service  
metadata:  
 name: supplier-service-service  
spec:  
 selector:  
 app: supplier-service  
 ports:  
 - protocol: TCP  
 port: 5000  
 targetPort: 5000  
 type: ClusterIP # ClusterIP for internal

### Step 4: Build the Docker Image

1. Open a terminal and navigate to the directory containing the Dockerfile and supplier\_service.py.
2. Build the Docker image:  
   docker build -t supplier-service:latest .

### Step 5: Load the Docker Image into your Kubernetes Cluster

* **If using Minikube:**  
  minikube image load supplier-service:latest
* **If using a different Kubernetes cluster (e.g., GKE, EKS, AKS):**
  + You need to push the image to a container registry (like Docker Hub, Google Container Registry, etc.).  
    docker tag supplier-service:latest <your-registry>/supplier-service:latest  
    docker push <your-registry>/supplier-service:latest
  + Then, in the k8s-all-in-one.yaml file, change the image field in the supplier-service-deployment section to:  
    image: <your-registry>/supplier-service:latest

### Step 6: Deploy to Kubernetes

1. Ensure your kubectl is configured to connect to your Kubernetes cluster.
2. Deploy the application using the k8s-all-in-one.yaml file:  
   kubectl apply -f deployment.yaml

### Step 7: Access the Supplier Service

* **If using ClusterIP (the default):** The service is only accessible from within the Kubernetes cluster. To access it from your host machine (for testing with Minikube), use:  
  minikube service supplier-service-service --url  
    
  This will print a URL that you can use with curl or a browser.
* **If you need external access on a different Kubernetes cluster:**
  + Change the type in the supplier-service-service definition in k8s-all-in-one.yaml to NodePort or LoadBalancer. LoadBalancer is preferred in cloud environments, but NodePort works in Minikube.
  + If you change to NodePort, get the node port:  
    kubectl get service supplier-service-service -o jsonpath='{.spec.ports[0].nodePort}'  
      
    Then, access the service at http://<your-node-ip>:<node-port>. To get your Minikube IP: minikube ip
  + If you change to LoadBalancer (and your cloud provider supports it), get the external IP:  
    kubectl get service supplier-service-service  
      
    Look for the External IP in the output.

///////

Okay, let's walk through how to create tables and query the MySQL database you've deployed in your Kubernetes cluster using the k8s-all-in-one.yaml configuration.

To create tables and query the MySQL database running in your Kubernetes cluster, you'll need to connect to the MySQL pod. Here's how you can do it:

**Step 1: Get the MySQL Pod Name**

First, you need to identify the name of the MySQL pod. Use the following command:

Bash

kubectl get pods -l app=mysql

This will list all pods with the label app=mysql. The output will look something like this:

NAME READY STATUS RESTARTS AGE

mysql-deployment-6b8d8b9b5f-r42p8 1/1 Running 0 20h

Copy the name of the MySQL pod (e.g., mysql-deployment-6b8d8b9b5f-r42p8).

**Step 2: Connect to the MySQL Pod**

Next, use kubectl exec to get a shell inside the MySQL pod:

Bash

kubectl exec -it <your-mysql-pod-name> -- /bin/bash

Replace <your-mysql-pod-name> with the actual name of your MySQL pod.

**Step 3: Connect to the MySQL Server**

Once you're inside the pod, you can connect to the MySQL server using the mysql client. You'll need the root password, which you set in the k8s-all-in-one.yaml file.

Bash

mysql -u root -p

When prompted, enter the password you specified for MYSQL\_ROOT\_PASSWORD in the mysql-deployment.yaml file (e.g., "password").

**Step 4: Create a Database (If Necessary)**

If you haven't already, create the suppliers\_db database:

Code snippet

CREATE DATABASE suppliers\_db;

USE suppliers\_db;

**Step 5: Create the suppliers Table (If Necessary)**

You can create the suppliers table as defined in your supplier\_service.py file.

Code snippet

CREATE TABLE suppliers (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

contact\_person VARCHAR(255),

email VARCHAR(255),

phone VARCHAR(20),

address VARCHAR(255)

);

**Step 6: Perform Queries**

Now you can perform standard SQL queries. Here are some examples:

* Insert data:

Code snippet

INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Acme Corp', 'John Doe', 'john.doe@example.com', '555-1234', '123 Main St');

* Select data:

Code snippet

SELECT \* FROM suppliers;

* Update data:

Code snippet

UPDATE suppliers SET email='new.email@example.com' WHERE name='Acme Corp';

* Delete data:

Code snippet

DELETE FROM suppliers WHERE name='Acme Corp';

**Step 7: Exit**

To exit the MySQL client, type:

Code snippet

exit

Then, to exit the pod's shell, type:

Bash

Exit

1. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Global Tech', 'Sarah Johnson', 's.johnson@globaltech.com', '555-9876', '450 Tech Park Blvd');
2. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Summit Supplies', 'Michael Chen', 'michael.chen@summitsupplies.net', '555-2345', '789 Industrial Way');
3. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Oceanic Foods', 'Lisa Wong', 'lisa.wong@oceanicfoods.org', '555-8765', '321 Harbor Drive');
4. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Pinnacle Manufacturing', 'Robert Davis', 'rdavis@pinnacle-mfg.com', '555-3456', '100 Factory Lane');
5. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Green Earth Organics', 'Emily Wilson', 'emily@greenearth.org', '555-6543', '55 Farm Road');
6. INSERT INTO suppliers (name, contact\_person, email, phone, address) VALUES ('Metro Office Solutions', 'James Rodriguez', 'j.rodriguez@metrosolutions.com', '555-7890', '200 Business Center Ave');

+++

+++ DeepSeek

# Connecting to Local MySQL Using Flask-MySQLdb

Here's a step-by-step guide to connect your Flask application to a local MySQL database using Flask-MySQLdb:

## Prerequisites

1. MySQL server installed and running locally

2. Python installed

3. A database created in MySQL

## Installation

First, install the required packages:

```bash

pip install flask flask-mysqldb

```

## Basic Setup

```python

from flask import Flask

from flask\_mysqldb import MySQL

app = Flask(\_\_name\_\_)

# Configure MySQL

app.config['MYSQL\_HOST'] = 'localhost'

app.config['MYSQL\_USER'] = 'your\_username' # Default is often 'root'

app.config['MYSQL\_PASSWORD'] = 'your\_password'

app.config['MYSQL\_DB'] = 'your\_database\_name'

mysql = MySQL(app)

@app.route('/')

def index():

# Create a cursor

cur = mysql.connection.cursor()

# Execute query

cur.execute("SELECT \* FROM your\_table")

# Fetch all rows

data = cur.fetchall()

# Close connection

cur.close()

return str(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

```

## Complete Example with CRUD Operations

```python

from flask import Flask, render\_template, request, redirect, url\_for, flash

from flask\_mysqldb import MySQL

app = Flask(\_\_name\_\_)

app.secret\_key = 'your\_secret\_key'

# MySQL Configuration

app.config['MYSQL\_HOST'] = 'localhost'

app.config['MYSQL\_USER'] = 'root'

app.config['MYSQL\_PASSWORD'] = 'yourpassword'

app.config['MYSQL\_DB'] = 'testdb'

app.config['MYSQL\_CURSORCLASS'] = 'DictCursor' # To get results as dictionaries

mysql = MySQL(app)

@app.route('/')

def index():

cur = mysql.connection.cursor()

cur.execute("SELECT \* FROM users")

users = cur.fetchall()

cur.close()

return render\_template('index.html', users=users)

@app.route('/add', methods=['POST'])

def add\_user():

if request.method == 'POST':

name = request.form['name']

email = request.form['email']

cur = mysql.connection.cursor()

cur.execute("INSERT INTO users (name, email) VALUES (%s, %s)", (name, email))

mysql.connection.commit()

cur.close()

flash('User added successfully!')

return redirect(url\_for('index'))

@app.route('/edit/<int:id>', methods=['POST', 'GET'])

def edit\_user(id):

cur = mysql.connection.cursor()

if request.method == 'POST':

name = request.form['name']

email = request.form['email']

cur.execute("UPDATE users SET name=%s, email=%s WHERE id=%s", (name, email, id))

mysql.connection.commit()

cur.close()

flash('User updated successfully!')

return redirect(url\_for('index'))

cur.execute("SELECT \* FROM users WHERE id = %s", (id,))

user = cur.fetchone()

cur.close()

return render\_template('edit.html', user=user)

@app.route('/delete/<int:id>')

def delete\_user(id):

cur = mysql.connection.cursor()

cur.execute("DELETE FROM users WHERE id = %s", (id,))

mysql.connection.commit()

cur.close()

flash('User deleted successfully!')

return redirect(url\_for('index'))

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

```

## HTML Templates

Create a `templates` folder and add these files:

`index.html`:

```html

<!DOCTYPE html>

<html>

<head>

<title>Flask MySQL App</title>

</head>

<body>

{% with messages = get\_flashed\_messages() %}

{% if messages %}

{% for message in messages %}

<div>{{ message }}</div>

{% endfor %}

{% endif %}

{% endwith %}

<h1>Users</h1>

<table>

<tr>

<th>ID</th>

<th>Name</th>

<th>Email</th>

<th>Actions</th>

</tr>

{% for user in users %}

<tr>

<td>{{ user.id }}</td>

<td>{{ user.name }}</td>

<td>{{ user.email }}</td>

<td>

<a href="/edit/{{ user.id }}">Edit</a>

<a href="/delete/{{ user.id }}">Delete</a>

</td>

</tr>

{% endfor %}

</table>

<h2>Add User</h2>

<form action="/add" method="POST">

<input type="text" name="name" placeholder="Name" required>

<input type="email" name="email" placeholder="Email" required>

<button type="submit">Add</button>

</form>

</body>

</html>

```

`edit.html`:

```html

<!DOCTYPE html>

<html>

<head>

<title>Edit User</title>

</head>

<body>

<h1>Edit User</h1>

<form action="/edit/{{ user.id }}" method="POST">

<input type="text" name="name" value="{{ user.name }}" required>

<input type="email" name="email" value="{{ user.email }}" required>

<button type="submit">Update</button>

</form>

</body>

</html>

```

## Notes:

1. Make sure your MySQL server is running before starting the Flask app

2. Create the database and table in MySQL first

3. For production, don't use root user and don't store passwords in code

4. Consider using environment variables for sensitive data

5. Always close your database cursors after use

6. Use parameterized queries (with %s) to prevent SQL injection

To create the database and table, you can run these SQL commands in MySQL:

```sql

CREATE DATABASE testdb;

USE testdb;

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100)

);

```

Postman JSON

{

    "name": "John Doe",

    "email": 30,

    "phone": "New York",

    "contact\_person": 1234,

    "address": "lala"

}

RABBIT MQ

Okay, I'll update the code to publish newly added suppliers to RabbitMQ. This will involve adding RabbitMQ libraries, connection setup, and message publishing to your Flask application.

Supplier Service with RabbitMQ Integration

May 4, 6:34 PM

Open

**Key Changes:**

* **RabbitMQ Libraries:**
  + Imported the pika library for RabbitMQ interaction.
* **RabbitMQ Connection:**
  + The connect\_rabbitmq function establishes a connection to the RabbitMQ server. It includes retry logic to handle potential connection failures.
  + The connection and channel are initialized globally to be reused across requests.
* **Queue Declaration:**
  + The queue new\_suppliers is declared in RabbitMQ. The durable=True setting ensures that the queue will survive a RabbitMQ server restart.
* **Message Publishing:**
  + The publish\_to\_rabbitmq function now handles publishing messages to the new\_suppliers queue. It converts the supplier data to JSON before sending it. The message is also made persistent. It also attempts to reconnect to RabbitMQ if the connection is lost.
* **New Supplier Publication:**
  + In the create\_supplier function, after a new supplier is successfully added to the database, the supplier's data (including the newly generated ID) is extracted, converted to JSON, and published to the new\_suppliers RabbitMQ queue.
* **Error Handling:**
  + Added more robust error handling for RabbitMQ connection and publishing. The application will attempt to reconnect to RabbitMQ if the connection is lost.

**Additional Notes:**

* **RabbitMQ Service:** You'll need to deploy a RabbitMQ service in your Kubernetes cluster. A basic RabbitMQ deployment YAML is provided below.
* **Dependencies:** Make sure to add pika to your requirements.txt file:
* Flask
* Flask-MySQLdb
* pika
* **Message Format:** The supplier data is published to RabbitMQ as a JSON string. This is a common and flexible format for message exchange.
* **Queue Durability:** The RabbitMQ queue is declared as durable, meaning that the queue itself will survive a RabbitMQ server restart. You can also make the messages persistent by setting the delivery mode to 2 in the message properties. This ensures that the messages are written to disk and will not be lost if the RabbitMQ server restarts.
* **Error Handling:** The code includes error handling for the RabbitMQ connection. If the connection fails, it will retry a few times before giving up. This helps to make the application more resilient to temporary network issues.
* **Idempotency:** In a production environment, you should consider the idempotency of message processing. This means that your message consumer should be able to handle duplicate messages without causing problems. One way to achieve this is to include a unique message ID in the message payload and have the consumer check for duplicates.

**RabbitMQ Deployment YAML (k8s-rabbitmq.yaml):**

Here's a basic Kubernetes YAML file to deploy RabbitMQ. You can save this as k8s-rabbitmq.yaml and apply it to your cluster. This is suitable for development; you might want to explore more robust, production-ready RabbitMQ deployments.

YAML

apiVersion: apps/v1

kind: Deployment

metadata:

name: rabbitmq-deployment

labels:

app: rabbitmq

spec:

replicas: 1

selector:

matchLabels:

app: rabbitmq

template:

metadata:

labels:

app: rabbitmq

spec:

containers:

- name: rabbitmq

image: rabbitmq:3.9-management # Or a specific version

ports:

- containerPort: 5672 # AMQP port

- containerPort: 15672 # Management UI port

env:

- name: RABBITMQ\_DEFAULT\_USER

value: myuser

- name: RABBITMQ\_DEFAULT\_PASS

value: mypassword

# livenessProbe: # Add livenessProbe

# tcpSocket:

# port: 5672

# initialDelaySeconds: 15

# periodSeconds: 10

# readinessProbe: # Add readinessProbe

# tcpSocket:

# port: 5672

# initialDelaySeconds: 5

# periodSeconds: 5

---

apiVersion: v1

kind: Service

metadata:

name: rabbitmq

labels:

app: rabbitmq

spec:

selector:

app: rabbitmq

ports:

- protocol: TCP

port: 5672

targetPort: 5672

name: amqp

- protocol: TCP

port: 15672

targetPort: 15672

name: management

type: ClusterIP # Use ClusterIP for internal access within the cluster

**To Deploy RabbitMQ:**

1. Save the YAML above to k8s-rabbitmq.yaml.
2. Apply it to your Kubernetes cluster:

Bash

kubectl apply -f k8s-rabbitmq.yaml

**Connecting to RabbitMQ Management UI (Optional):**

If you want to access the RabbitMQ management UI:

* **Port Forwarding (for development):**

Bash

kubectl port-forward svc/rabbitmq 15672:15672

Then, you can access the UI at http://localhost:15672. Use the username myuser and password mypassword.

* **Ingress (for production):** For production use, you'd typically set up an Ingress to expose the RabbitMQ management UI.

Don't forget to create the RabbitMQ service.

**Setting up RabbitMQ on Windows for Testing**

Here's how to set up RabbitMQ on your Windows machine for local development and testing:

**1. Install Erlang**

* RabbitMQ is written in Erlang, so you need to install it first.
* Go to the Erlang Solutions website: <https://www.erlang.org/downloads>
* Download the Windows installer. Choose the latest stable version.
* Run the installer and follow the on-screen instructions. Make sure to add Erlang to your system's PATH environment variable during the installation.

**2. Install RabbitMQ**

* Go to the RabbitMQ website: <https://www.rabbitmq.com/download.html>
* Under "RabbitMQ Server," find the link for "Windows Installer."
* Download the installer (it will be an .exe file).
* Run the installer and follow the on-screen instructions. The default settings are usually fine for development.

**3. Start the RabbitMQ Service**

* RabbitMQ should start automatically after installation. If not, you can start it manually:
* **Using Services:**
  + Open the Services application (search for "Services" in the Windows search bar).
  + Find "RabbitMQ" in the list of services.
  + If it's not running, right-click on it and select "Start."
* **Using the Command Line:**
  + Open a command prompt as an administrator.
  + Run the following command:
  + net start RabbitMQ

**4. Enable the RabbitMQ Management Plugin (Recommended)**

* The RabbitMQ Management Plugin provides a web-based UI for managing and monitoring your RabbitMQ server. It's very helpful for development and testing.
* Open a command prompt as an administrator.
* Run the following command:
* rabbitmq-plugins enable rabbitmq\_management
* Restart the RabbitMQ service:
* net stop RabbitMQ
* net start RabbitMQ

**5. Access the RabbitMQ Management UI**

* Open your web browser and go to: http://localhost:15672
* The default username is "guest," and the default password is "guest."

**6. Test the Setup**

* In the RabbitMQ Management UI, you should see the overview of your RabbitMQ server. You can view queues, exchanges, connections, and other information.
* You can create a queue in the UI to test if your Python code can connect and publish messages.

**Important Notes:**

* **Firewall:** Ensure that RabbitMQ's ports (5672 for AMQP, 15672 for the management UI) are not blocked by your Windows Firewall.
* **Credentials:** For production, change the default "guest" username and password to more secure credentials.
* **Erlang Version:** RabbitMQ has specific Erlang version requirements. The installers usually handle this, but if you encounter issues, check the RabbitMQ documentation for compatibility.
* **Service Startup Type:** By default, the RabbitMQ service is set to start automatically. You can change this in the Services application if needed.