

ATLAN TASK

Application Architecture

The e-commerce application consists of the following components:

1. **Frontend:** A React-based web application served by an Nginx server.
2. **Backend API:** A set of Node.js microservices handling various business operations.
3. **Database:** A MongoDB cluster for data persistence.
4. **Message Queue:** RabbitMQ for handling order processing asynchronously.

The Above application architecture is deployed in an kubernetes/Openshift environment. I have created a custom url to access the application

Clone the repo ecommerce-app

Change the working directory to ecommerce-app

```
cd ecommerce-app
```

Note: Change the image-registry of your convenience

Eg: Image-registry: hub.docker.io/repo_name

Create a project namespace in kubernetes/Openshift

```
kubectl apply -f ecommerce-k8s/namespace.yaml
```

Backend:

Backend server is a node js rest application running on server 5000.

To build the backend server:

```
make build-backend
```

Push the backend to the container registry:

```
make push-backend
```

Deploy the backend server in kubernetes/Openshift

```
kubectl apply -f ecommerce-k8s/backend
```

Frontend:

A react js frontend application serving on 3000 port locally and running on cluster ip service in kubernetes

To build the fronted server

`make build-frontend`

Push the frontend to the container registry:

`make push-frontend`

Deploy the frontend server in kubernetes/Openshift

`kubectl apply -f ecommerce-k8s/frontend`

Message-Broker:

A node js server which runs and connects to both mongo db and rabbitmq instances

To build the message server

`make build-message`

Push the message to the container registry:

`make push-message`

Deploy the message server in kubernetes/Openshift

`kubectl apply -f ecommerce-k8s/message`

Database: MongoDB Deployment

`Kubectl apply -f ecommerce-k8s/mongodb`

Message Queue: RabbitMQ Deployment

`Kubectl apply -f ecommerce-k8s/rabbitmq`

Phase 1: Deployment

Marked-green – done

Marked-red — not done

1. Configuration and Deployment:

- Create Kubernetes manifests (YAML files) for each component, ensuring proper configuration, interconnection, and deployment strategies (e.g., RollingUpdate).
- Use ConfigMaps and Secrets to manage application configurations and sensitive data.

2. Networking:

- Configure the Ingress controller to expose the frontend service with proper URL routing.
- Set up internal networking with appropriate service discovery mechanisms for inter-service communication.

3. Persistence:

- Implement Persistent Volumes and Claims for MongoDB and RabbitMQ to ensure data persistence and resilience.

4. Security and Compliance:

- Define and enforce security policies using Pod Security Policies or OPA/Gatekeeper to ensure least privilege access and other security best practices.

The Pod Security policies are configured in kubernetes api server yaml definition. A custom admission-controller webhook runs in the cluster which check the pods and validates on every request.

Phase 2: Observability and Scaling

1. Monitoring and Logging:

- Integrate a monitoring solution (e.g., Prometheus and Grafana) to track the performance and resource usage of all components.

To track the performance and resource usage of resources in the namespace we

Can check with the below command.

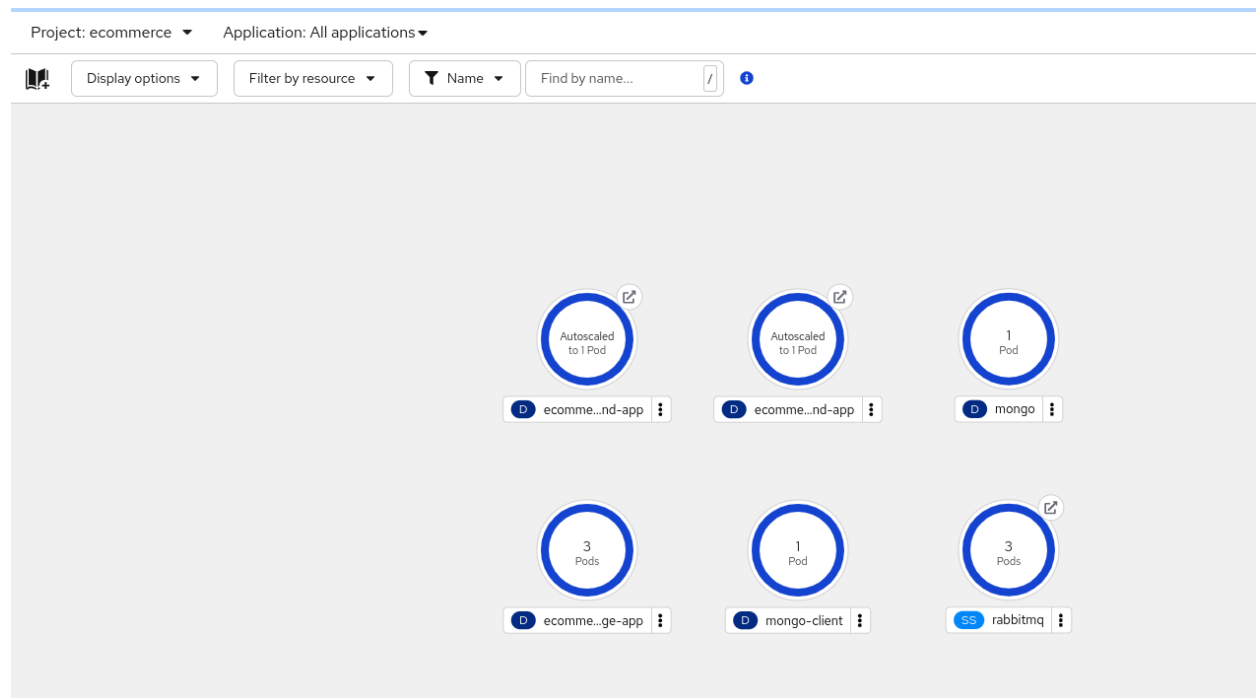
```
kubectl top pods -n ecommerce
```

```
kubectl top pods -n ecommerce
```

NAME	CPU(cores)	MEMORY(bytes)
ecommerce-backend-app-6f8fb549c4-pfs8j	1m	34Mi
ecommerce-frontend-app-5ddb54cc4c-ktc56	1m	283Mi

ecommerce-frontend-app-5ddb54cc4c-w9kkn	1m	288Mi
ecommerce-message-app-78788f6fc6-8dpcq	1m	22Mi
ecommerce-message-app-78788f6fc6-sdwzs	1m	22Mi
ecommerce-message-app-78788f6fc6-x7v7w	1m	22Mi
mongo-6cd74b648d-z7qwj	82m	144Mi
mongo-client-7c77b5f795-tnmbw	5m	130Mi
rabbitmq-0	3m	129Mi
rabbitmq-1	14m	131Mi
rabbitmq-2	4m	130Mi

- Set up a centralized logging solution (e.g., ELK stack) to aggregate and analyze logs from all components.



2. Autoscaling:

- Implement Horizontal Pod Autoscaling for the frontend and backend services based on CPU/Memory usage metrics.

Project: ecommerce ▼

HorizontalPodAutoscalers

Name ▼
Search by name...

Name ↓	Labels ↓	Scale target ↓	Min pods ↓	Max pods ↓
HPA backend-autoscaler	No labels	ecommerce-backend-app	1	5
HPA frontend-autoscaler	No labels	ecommerce-frontend-app	1	5

- Discuss the possibility and strategy for implementing Cluster Autoscaling.

The autoscalers increases the pod count based on the memory utilization in the frontend and backend pods. The average Utilization parameter in autoscalers is set to 100 which is configurable based on the traffic .

```
averageUtilization: 100
```

Phase 3: Debugging & Troubleshooting Scenarios

1. Frontend Accessibility:

- Issue: The frontend service is not accessible externally post-deployment.
- Task: Diagnose the networking and Ingress configuration, ensuring the frontend is reachable and performs as expected.

Answer:

Check the logs of the frontend pods by sing the command

```
kubectl -n ecommerce logs pod/ecommerce-frontend-app-5ddb54cc4c-ktc56
```

```
> ecommerce-backend@1.0.0 start
```

```
> node server.js
```

```
amqp://admin
```

:admin

@rabbitmq.ecommerce.svc.cluster.local:5672

Server running on 0.0.0.0:5000

MongoDB connected

The above output says the frontend pod is successfully connected to all the services in the cluster. I observe no issues in the cluster.

If any case we have to check the connection parameter of React js application with the backend service.

Check the network-policy whether the traffic from react js pod to backend pod is configured correctly

2. Intermittent Backend-Database Connectivity:

- Issue: The backend services occasionally lose connection to the MongoDB cluster, causing request failures.

Check the database parameter are correctly configured in the configmaps and secrets.

Check the rabbitmq connection parameters are correctly configured. Try to ping from backend pod to mongoclient and rabbitmq client.

3. Order Processing Delays:

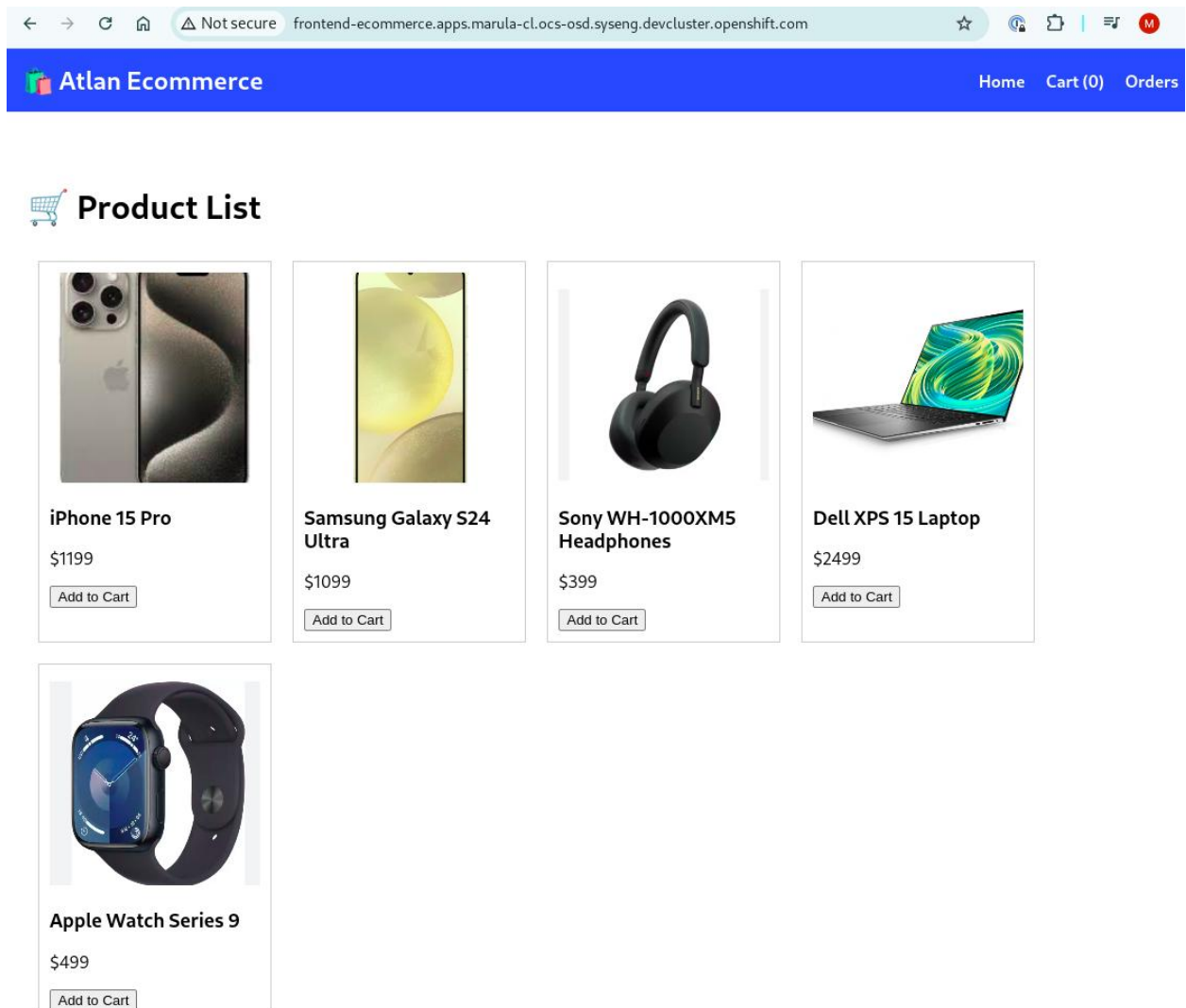
- Issue: Users report delays in order processing, suspecting issues with the RabbitMQ message queue.
- Task: Analyze and optimize the message queue setup, ensuring efficient message processing and minimal latency.

Check the rabbitmq connection parameters are correctly configured. Try to ping from backend pod to mongoclient and rabbitmq client.

The overall application after deployment in kubernetes clusters looks like below. Access the URL using the link

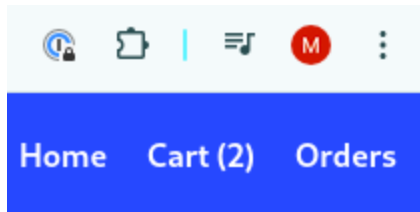
<http://frontend-ecommerce.apps.marula-cl.ocs-osd.syseng.devcluster.openshift.com/>

Frontend Page:

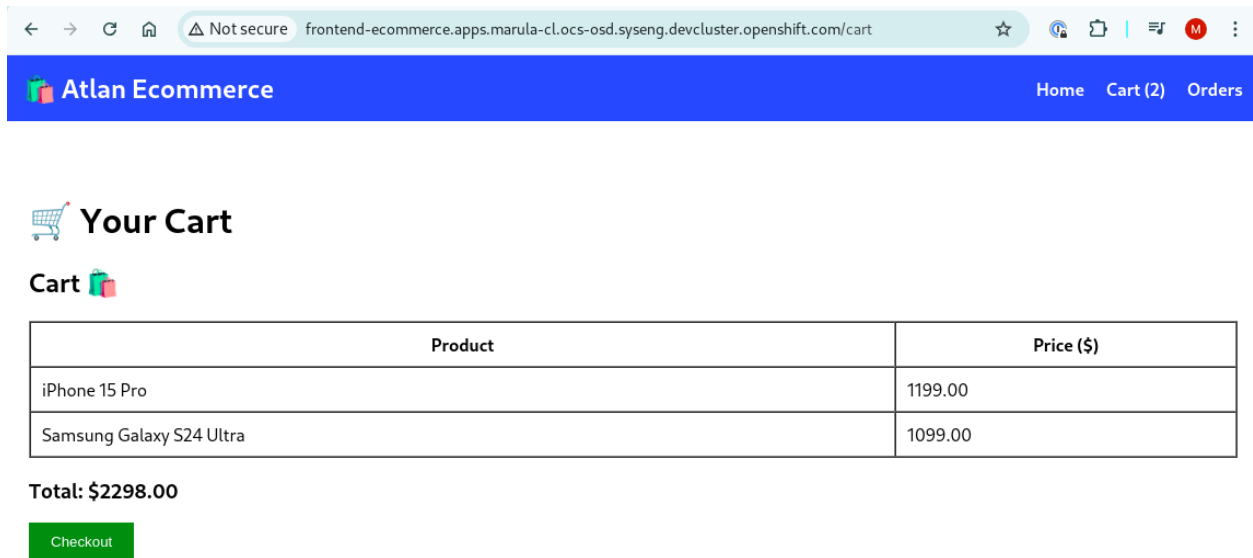


I will do a transaction of creating an order of 2 items.

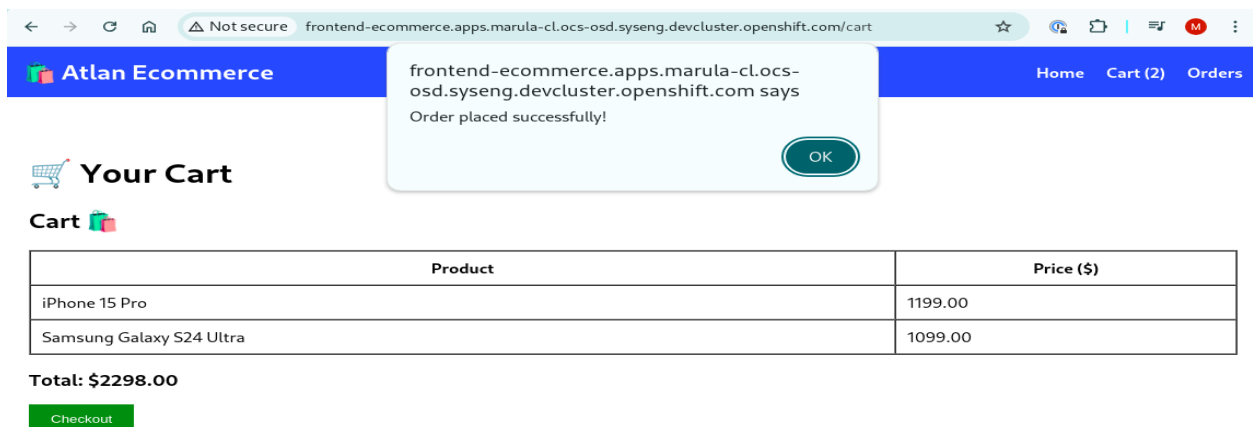
When we click on add to cart button, the count on the Cart increases, which you can see in the below screenshot.



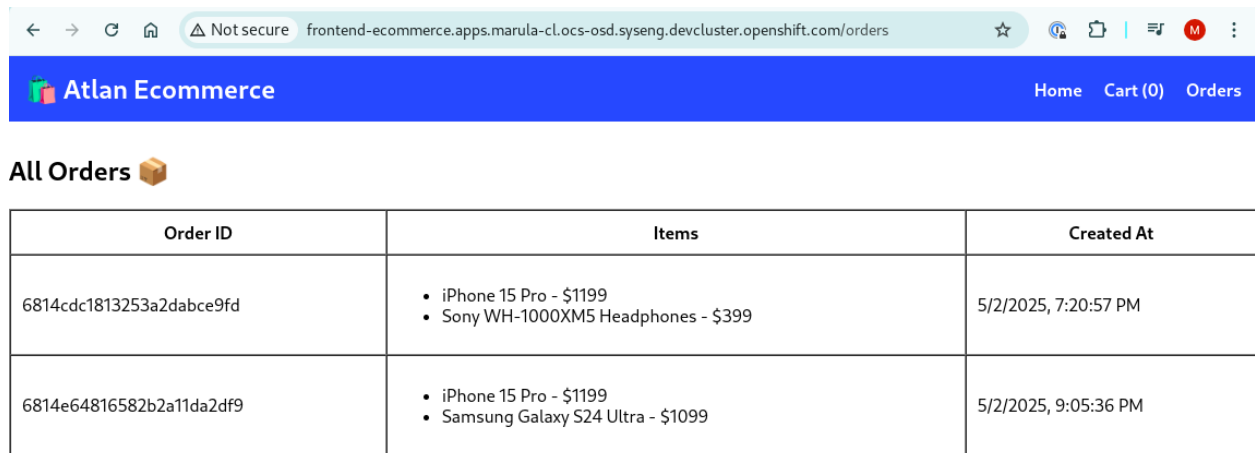
The cart page shows the items added into the cart



When we click on the checkout the rabbitmq princess the request and updates the orders tables



The orders page on the applications shows the total orders

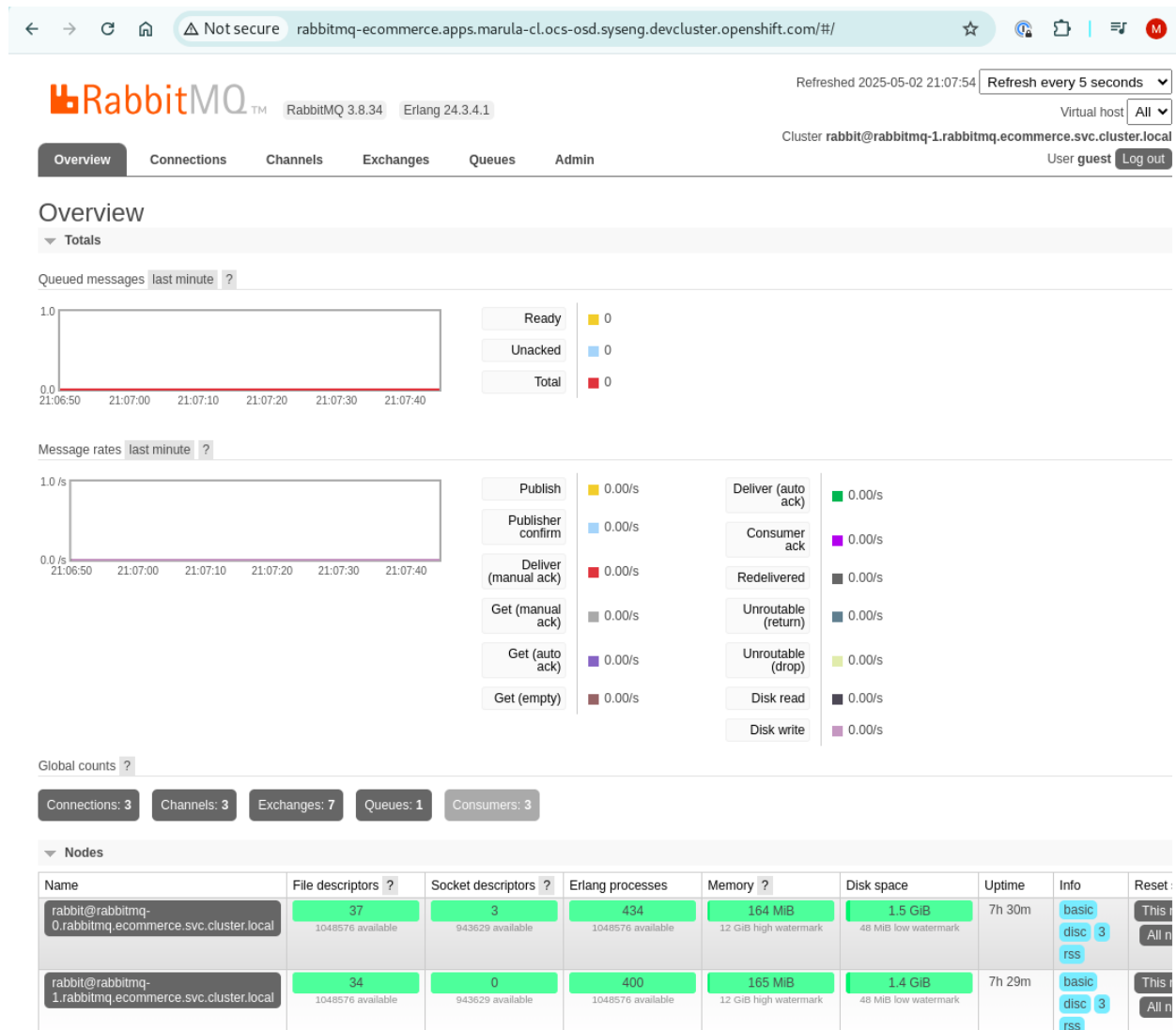


Order ID	Items	Created At
6814cdc1813253a2dabce9fd	<ul style="list-style-type: none">• iPhone 15 Pro - \$1199• Sony WH-1000XM5 Headphones - \$399	5/2/2025, 7:20:57 PM
6814e64816582b2a11da2df9	<ul style="list-style-type: none">• iPhone 15 Pro - \$1199• Samsung Galaxy S24 Ultra - \$1099	5/2/2025, 9:05:36 PM

The message queue RabbitMQ client , where we can monitor the order queues in the client.

To access the rabbitmq service

<http://rabbitmq-ecommerce.apps.marula-cl.ocs-osd.syseng.devcluster.openshift.com/#/>



To check the Database tables and schema in the Mongo db pods we can run

```
kubectl -n ecommerce exec deployment/mongo-client -it -- /bin/bash
```

root@mongo-client-7c77b5f795-tnmbw:/#

```

mrudraia:ecommerce-k8s$ kubectl -n ecommerce exec deployment/mongo-client -it -- /bin/bash
root@mongo-client-7c77b5f795-tnmbw:/# mongosh --host mongo-svc --port 27017 -u adminuser -p password123
Current Mongosh Log ID: 6814e7b1bf17227391d861df
Connecting to:      mongodb://<credentials>@mongo-svc:27017/?directConnection=true&appName=mongosh+2.5.0
Using MongoDB:      8.0.9
Using Mongosh:      2.5.0

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

-----
  The server generated these startup warnings when booting
2025-05-02T11:27:47.569+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb
system
2025-05-02T11:27:48.615+00:00: For customers running the current memory allocator, we suggest changing the contents of the following sysfsFil
2025-05-02T11:27:48.615+00:00: We suggest setting the contents of sysfsFile to 0.
2025-05-02T11:27:48.615+00:00: vm.max_map_count is too low
-----

test> use ecommerce
switched to db ecommerce
ecommerce> db.products.all()
TypeError: db.products.all is not a function
ecommerce> db.products.find().pretty()
[
  {
    _id: ObjectId('6814c0a5dbc622a587d861e0'),
    name: 'iPhone 15 Pro',
    description: 'Latest Apple smartphone with A17 Pro chip.',
    price: 1199,
    image: 'images/iphone15pro.jpg'
  },
  {
    _id: ObjectId('6814c0a5dbc622a587d861e1'),
    name: 'Samsung Galaxy S24 Ultra',
    description: 'Flagship Samsung device with best-in-class camera.',
    price: 1099,
    image: 'images/galaxys24ultra.jpg'
  },
  {
    _id: ObjectId('6814c0a5dbc622a587d861e2'),
    name: 'Sony WH-1000XM5 Headphones',
    description: 'Top-rated noise-cancelling wireless headphones.',
    price: 399,
    image: 'images/sonywh1000xm5.jpg'
  },
  {
    _id: ObjectId('6814c0a5dbc622a587d861e3'),
    name: 'Dell XPS 15 Laptop',
    description: 'Powerful laptop with Intel i9 and RTX graphics.',
    price: 2499,
  }
]

```