



REPORT

IT-460

CLOUD COMPUTING

Multi-Container App Management with Openshift

Authors:

Rayen Nasraoui
Rayen Chtioui

Submitted to:

Prof. Manel Abdelkader

Contents

1	Introduction	1
2	Implementation	1
2.1	Architecture	1
2.2	Dockerfiles setup	2
2.2.1	Base Image	2
2.2.2	Working Directory	2
2.2.3	Environment Variables	2
2.2.4	Copy Files	3
2.2.5	Install Dependencies	3
2.2.6	Generate Prisma Client	3
2.2.7	Expose Port	3
2.2.8	Default Command	3
2.3	Deployment Startup	3
2.3.1	Environment Variable Retrieval Function	3
2.3.2	Deployment Cleanup	3
2.3.3	Application Deployment	4
2.3.4	Setting Environment Variables from Secret	4
2.3.5	Email Service Deployment	4
2.3.6	Role and Role Binding Deployment	4
2.3.7	Expose API Gateway Service	4
2.4	Database Data Persistance	5
2.5	Continuous Integration	6
2.6	Container Deployment in OpenShift	6
2.7	Email Monitoring Service for Pod Failures	7
3	Conclusion	8

List of Figures

1	Architecture	2
2	Database Template	5
3	Github actions	6
4	Example of a Failure in a Pod	8

1 Introduction

In this project, we showcase the transformative power of OpenShift in redefining cloud management. Through a multi-container architecture, OpenShift seamlessly integrates diverse components, laying the groundwork for robust Continuous Integration and Continuous Deployment (CI/CD) pipelines. With automated scaling, load balancing, and fault tolerance, our project highlights how applications can dynamically adapt to varying workloads.

2 Implementation

2.1 Architecture

- **API Gateway:**
 - Located in the `api-gateway` directory.
 - Serves as the entry point for all client requests.
 - Includes its own Dockerfile and configuration.
- **Services:**
 - The application is divided into microservices, each responsible for specific tasks.
 - Located in the `services` directory.
 - Each service has its own environment configuration, Dockerfile, and source code.
 - **Client:** Handles client-related operations.
 - **Product:** Manages product-related operations.
 - **User:** Manages user-related operations.
 - **Email:** Sends emails to the admin about incidents happening in the architecture.
- **PostgreSQL Database:**
 - a PostgreSQL database instance is created by configuring the necessary information in the dashboard
 - It ensures data-persistence when the database client's pods are rolled-out (restarted)

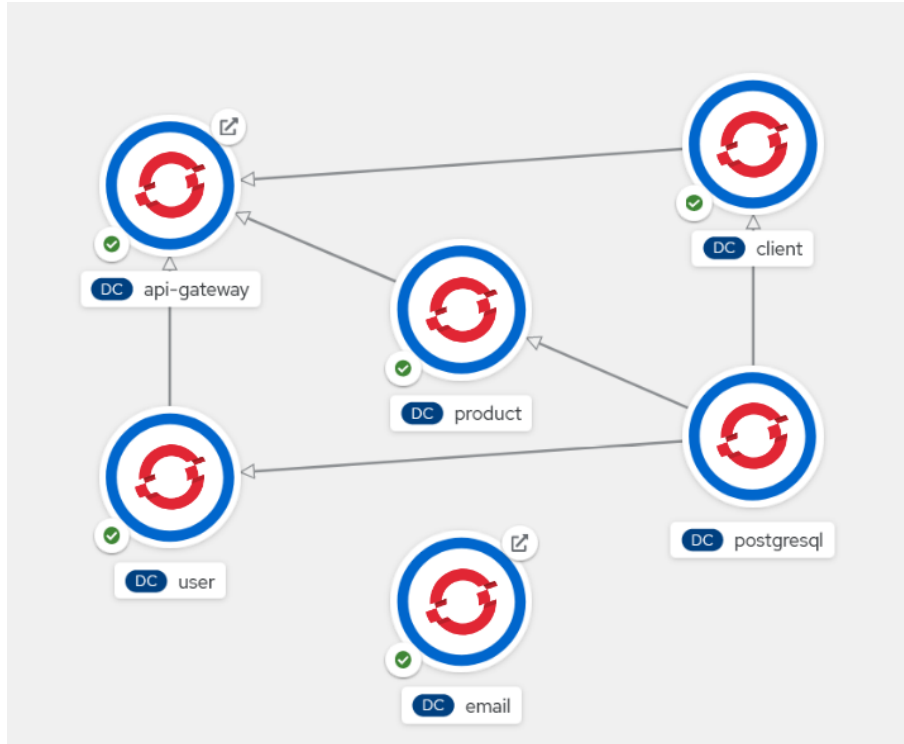


Figure 1: Architecture

2.2 Dockerfiles setup

In this section we will explain the setup for the Dockerfiles. We take the example of the Client API

2.2.1 Base Image

```
FROM node:14
```

This line sets the base image for the Docker container. Here, it's using the official Docker image for Node.js version 14.

2.2.2 Working Directory

```
WORKDIR /usr/src/app
```

This line sets the working directory inside the Docker container to `/usr/src/app`. All subsequent commands in the Dockerfile will be run from this directory.

2.2.3 Environment Variables

```
# Service port
ENV PORT=8001

# Other services
ENV PRODUCT_SERVICE_URL='http://product:8002'
ENV USER_SERVICE_URL='http://user:8003'
```

These lines set environment variables inside the Docker container. `PORT` is the port on which the service will run. `PRODUCT_SERVICE_URL` and `USER_SERVICE_URL` are the URLs of the product and user services in Kubernetes, respectively.

2.2.4 Copy Files

```
COPY . .
```

This line copies the entire contents of the current directory on the host machine into the working directory in the Docker container.

2.2.5 Install Dependencies

```
RUN npm install
```

This line runs the `npm install` command, which installs all the dependencies listed in the `package.json` file.

2.2.6 Generate Prisma Client

```
RUN npx prisma generate
```

This line runs the `npx prisma generate` command, which generates the Prisma client based on the Prisma schema.

2.2.7 Expose Port

```
EXPOSE 8001
```

This line tells Docker that the container will listen on the specified network ports at runtime. Here, it's exposing port 8001.

2.2.8 Default Command

```
CMD [ "npm", "run", "start" ]
```

This line sets the default command for the Docker container. When the container is run, it will execute the `npm run start` command, which typically starts the Node.js application.

2.3 Deployment Startup

In this section, we will describe the script that's used to configure the setup at the start of the project

2.3.1 Environment Variable Retrieval Function

```
function getenv() {  
  oc get secret postgresql -o jsonpath="{.data.$1}" | base64 -d  
}
```

This function, `getenv`, is defined to retrieve a specific environment variable from a Kubernetes Secret named `postgresql`. It uses the `oc get` command to extract the base64-encoded data corresponding to the specified key (`$1`) in the Secret and then decodes it with `base64 -d`.

2.3.2 Deployment Cleanup

```
oc delete imagestream,buildconfig,deploymentconfig,service,routes  
  {product,user,client,api-gateway,email}  
oc delete template {product,user,client,api-gateway,email}-template  
oc delete rolebinding pod-watcher-rolebinding  
oc delete role pod-watcher-role  
oc delete secret email-secret
```

This section deletes various OpenShift resources such as image streams, build configurations, deployment configurations, services, routes, templates, role bindings, roles, and secrets related to products, users, clients, API gateways, and emails.

2.3.3 Application Deployment

```
oc new-app https://github.com/rayenchtioui/microservices-base \
  --as-deployment-config=true \
  --build-env=DATABASE_URL="postgresql://$(getenv database-user):$(getenv
    database-password)@postgresql:5432/$(getenv database-name)?schema=public" \
  --name api-gateway --context-dir api-gateway
```

This command deploys the `api-gateway` service from the specified GitHub repository. It sets the `DATABASE_URL` environment variable using the values retrieved from the `postgresql` Secret. The `-context-dir` flag specifies the context directory within the repository.

Similar commands deploy services for `product`, `client`, and `user`, each with their respective context directories.

2.3.4 Setting Environment Variables from Secret

```
oc set env --from secret/postgresql dc/product
oc set env --from secret/postgresql dc/client
oc set env --from secret/postgresql dc/user
```

These commands set environment variables for the deployed services (`product`, `client`, `user`) from the `postgresql` Secret.

2.3.5 Email Service Deployment

```
oc create -f k8s/email.yaml
oc new-app --template email-template
oc create secret generic email-secret --from-env-file=services/email/.env
oc set env --from=secret/email-secret dc/email
```

These commands deploy an email service. They create Kubernetes resources from YAML files, instantiate a template, create a generic secret for email configuration, and set environment variables for the `email` service.

2.3.6 Role and Role Binding Deployment

```
oc create -f k8s/role.yaml
oc create -f k8s/rolebinding.yaml
```

These commands create role and role binding Kubernetes resources.

2.3.7 Expose API Gateway Service

```
oc expose svc/api-gateway
```

This command exposes the `api-gateway` service to make it accessible externally.

2.4 Database Data Persistence

In OpenShift, when creating a PostgreSQL database using a template, several resources are generated to facilitate deployment, management, and persistence. Here's an overview of the purpose of each resource:

1. DeploymentConfig:

- *Purpose:* Defines deployment settings for the PostgreSQL database, specifying deployment, replication, and update strategies.
- *Details:* Configurations include the number of replicas, update strategies, and image specifications. Note that scaling to more than one replica is not supported.

2. PersistentVolumeClaim:

- *Purpose:* Ensures persistent storage for the PostgreSQL database, allowing data to persist across pod restarts or redeployments.
- *Details:* Requests storage from the cluster, and the associated PersistentVolumeClaim ensures data integrity across pod lifecycle events.

3. Secret:

- *Purpose:* Manages sensitive information and credentials for PostgreSQL, such as usernames and passwords.
- *Details:* Securely stores and provides access to sensitive credentials, enhancing security by separating them from configuration.

4. Service:

- *Purpose:* Defines a Kubernetes service to expose the PostgreSQL database within the cluster.
- *Details:* Enables communication with the PostgreSQL database for other applications or services within the same OpenShift project.

Instantiate Template

Namespace *
test

Memory Limit *
512Mi
Maximum amount of memory the container can use.

Namespace
openshift
The OpenShift Namespace where the ImageStream resides.

Database Service Name *
postgresql
The name of the OpenShift Service exposed for the database.

PostgreSQL Connection Username
(generated if empty)
Username for PostgreSQL user that will be used for accessing the database.

PostgreSQL Connection Password
(generated if empty)

PostgreSQL
DATABASE: POSTGRES
[View documentation](#) [Get support](#)

PostgreSQL database service, with persistent storage. For more information about using this template, including OpenShift considerations, see <https://github.com/sclorg/postgresql-container/>.

NOTE: Scaling to more than one replica is not supported. You must have persistent volumes available in your cluster to use this template.

The following resources will be created:

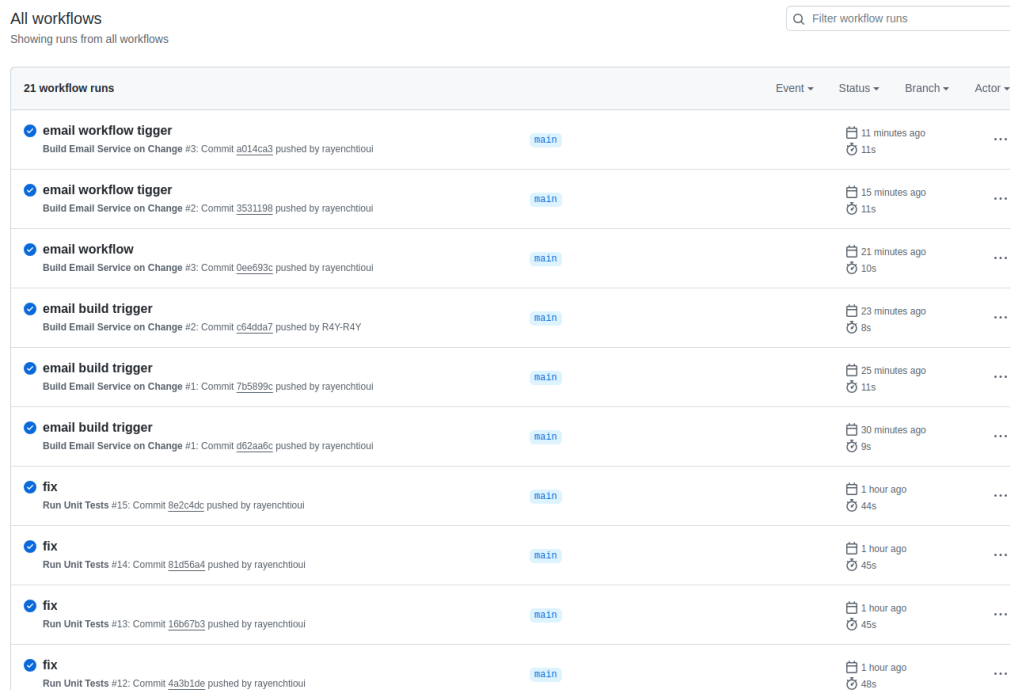
- DeploymentConfig
- PersistentVolumeClaim
- Secret
- Service

Figure 2: Database Template

By utilizing this template, OpenShift simplifies the process of setting up a PostgreSQL database, ensuring persistent storage, secure credentials, and accessibility within the cluster. These resources collectively contribute to the reliability, security, and ease of use of the PostgreSQL database in the OpenShift environment.

2.5 Continuous Integration

We used Github Webhooks and Actions to trigger builds when there is push in the remote repository.



The screenshot shows the 'All workflows' page on GitHub. It displays a list of 21 workflow runs. The table includes columns for Event, Status, Branch, and Actor. The runs are categorized into 'email workflow tigger', 'email workflow', 'email build trigger', and 'fix' jobs. Each run shows the commit hash, the user who pushed, and the time since the run completed.

Event	Status	Branch	Actor
email workflow tigger	Completed	main	rayenchtioui
email workflow tigger	Completed	main	rayenchtioui
email workflow	Completed	main	rayenchtioui
email build trigger	Completed	main	R4Y-R4Y
email build trigger	Completed	main	rayenchtioui
email build trigger	Completed	main	rayenchtioui
fix	Completed	main	rayenchtioui
fix	Completed	main	rayenchtioui
fix	Completed	main	rayenchtioui
fix	Completed	main	rayenchtioui

Figure 3: Github actions

2.6 Container Deployment in OpenShift

This section outlines the configuration for the objects that are used in OpenShift for deployment, focusing on load balancing and replication management.

- **Template Metadata:** Defines the template using 'template.openshift.io/v1' API version, categorized as a 'Template' with the name 'api-gateway-template'.
- **ImageStream Object:** Establishes an ImageStream for storing Docker images of the API gateway.
- **BuildConfig Object:** Configures the build process, with source code from 'https://github.com/rayenchtioui/microservices-base'. It employs a Docker build strategy and triggers (GitHub, Generic, ImageChange) for automated builds.
- **DeploymentConfig Object:** Manages deployment settings. Key features include:
 - **Replicas:** Set to 3, enabling replication for load balancing and high availability.
 - **Triggers:** Includes ConfigChange and ImageChange for dynamic updates.
 - **Strategy:** Uses a 'Rolling' strategy for continuous deployment without downtime.
- **Service Object:** Defines a service with 'ClusterIP' type, mapping port 8000 internally to an external port (8000), facilitating network traffic management and load balancing.
- **Route Object:** Creates an external route to the API gateway service, specifying the service port as '8080-tcp'.

This configuration ensures that the API Gateway is efficiently load balanced across multiple replicas, enhancing reliability and scalability in the OpenShift environment.

2.7 Email Monitoring Service for Pod Failures

This section describes the deployment and functionality of an email notification service in a FastAPI application on OpenShift. The service monitors Kubernetes pods and sends email alerts upon pod failures.

- **FastAPI Application:** Utilizes FastAPI, a high-performance web framework for building APIs with Python.
- **Periodic Monitoring:** Implements an asynchronous function, `periodic_monitoring()`, which periodically invokes `monitor_pods()` every 100 seconds to check the status of Kubernetes pods.
- **Background Loop:** During the application's startup, a new asynchronous event loop is initiated in a background thread, allowing the monitoring function to operate independently of the main application thread.
- **Pod Monitoring:** The `monitor_pods()` function leverages Kubernetes client libraries to continuously watch for pod events within the 'test' namespace. It identifies pods with a "Failed" status and triggers an email notification process.
- **Email Notification:** The `send_mail_logs()` function is tasked with sending emails in response to pod failures. It constructs an email with relevant details about the failed pod and dispatches it to a predefined recipient.
- **OpenShift Template:** The `email.yaml` file outlines an OpenShift template for deploying this email service. It encompasses several key components:
 - **ImageStream:** Manages Docker images for the service.
 - **BuildConfig:** Automates the building of the service from a Git repository.
 - **DeploymentConfig:** Configures the deployment with three replicas for resilience.
 - **Service:** Defines the internal cluster exposure of the service.
 - **Route:** Establishes an external access route to the service.
 - **Role (pod-watcher-role):**

```

* apiVersion: rbac.authorization.k8s.io/v1
* Kind: Role
* Namespace: test
* Permissions:
  · Resources: Pods
  · Verbs: get, watch, list

```
 - **RoleBinding (pod-watcher-rolebinding):**

```

* apiVersion: rbac.authorization.k8s.io/v1
* Kind: RoleBinding
* Namespace: test
* Subject: Default Service Account in test namespace
* Role Reference: pod-watcher-role

```

This sets up a Role with permissions for pod-related operations (get, watch, list) and binds it to the default Service Account in the `test` namespace for execution.

This service exemplifies an automated, real-time approach to monitoring and alerting in Kubernetes environments, ensuring quick response to pod failures and enhancing overall system reliability.

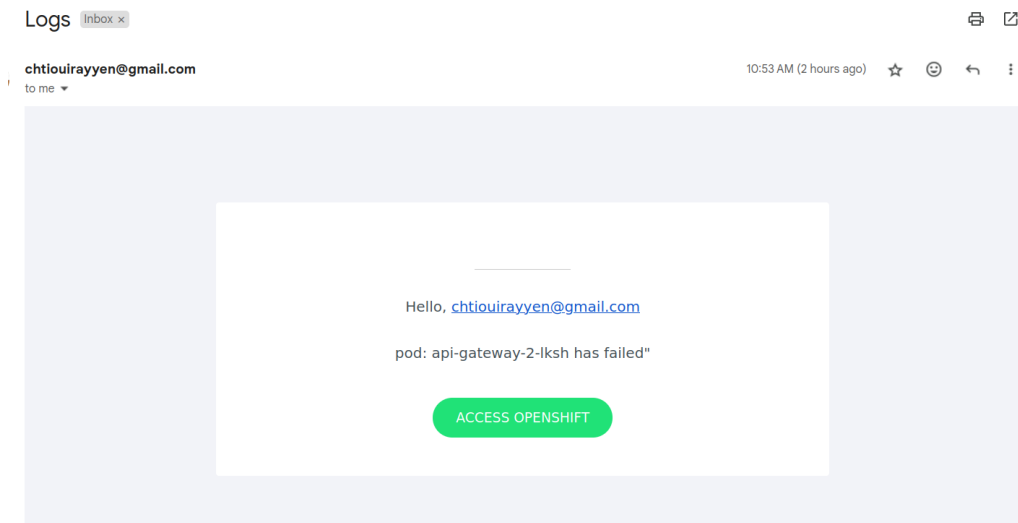


Figure 4: Example of a Failure in a Pod

3 Conclusion

In this report, we explored the use of OpenShift in cloud management. The multi-container architecture, orchestrated by OpenShift, seamlessly integrated diverse components, showcasing efficiency and agility.

The implementation section detailed the architecture, deployment startup, continuous integration, and container deployment in OpenShift. The PostgreSQL database ensured data persistence, and the deployment script provided a clear, automated process for application deployment on OpenShift.

Continuous integration was achieved through GitHub Webhooks and Actions, automating builds upon repository pushes. The container deployment configuration in OpenShift, with load balancing and replication management, highlighted the application's robustness and scalability.

The email monitoring service for pod failures demonstrated real-time monitoring and alerting in Kubernetes environments. Utilizing FastAPI, it periodically checked pod statuses, sending email notifications upon failures, contributing to enhanced system monitoring.

The overall project illustrates OpenShift's power in efficient application management, continuous integration, and real-time monitoring. This report serves as a concise guide for deploying a multi-container application on OpenShift, emphasizing its role in fostering efficiency and agility in software development.