Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2022, 9(12):148-153

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Implementing and Understanding the Ambassador Container Pattern in Kubernetes Ecosystems

Bhargav Bachina

Abstract This concise paper explores the Kubernetes platform, a premier solution for automating, scaling, and managing containerized applications, with a particular focus on the concept of pods as the fundamental operational units. It distinguishes the practical applications and advantages between single-container and multi-container pods, with an in-depth look at the Ambassador container pattern within the multi-container context. This pattern facilitates enhanced network communication and service integration, proving critical for complex Kubernetes deployments. By presenting a comparison with other pod patterns and offering a detailed case study, this study aims to underscore the significance and operational benefits of adopting the Ambassador pattern in Kubernetes environments. The objective is to furnish developers and IT professionals with strategic insights into leveraging Kubernetes for improved application performance and efficiency in cloud-native ecosystems.

Keywords Kubernetes, Docker, DevOps, Software Development, Software Engineering

Kubernetes stands out as a leading open-source engine for container orchestration, specifically designed to streamline the deployment, scaling, and management processes for applications encapsulated in containers. Within the Kubernetes ecosystem, the primary structural element is the pod, which serves as the cornerstone for orchestrating application instances. Unlike traditional container management that focuses on individual containers, Kubernetes introduces a higher level of abstraction by orchestrating pods. These pods act as encapsulated environments that can house one or more containers, along with associated storage resources, IP addresses, and configuration details that dictate the operational parameters for the containers contained within.

The architecture of Kubernetes pods is designed to support various deployment scenarios. The simplest form is the single-container pod, which, as the name suggests, contains only one container. This setup represents the most straightforward and commonly encountered use case in Kubernetes deployments, ideal for managing singular, isolated tasks. On the other hand, Kubernetes also supports multi-container pods, which are designed to contain multiple interrelated containers that share the same pod resources and lifecycle. These multi-container pods are essential for applications that require closely coupled service components to operate within the same network space and share common resources.

Among the strategies for managing multi-container pods, the Ambassador container pattern stands out. This pattern is specifically designed to simplify the communication between containers and the external environment. It acts as a proxy, routing requests from the outside world to the appropriate container within the pod. In the forthcoming sections, we will delve deeper into the Ambassador container pattern, providing a comprehensive exploration accompanied by an illustrative example project. This detailed examination will shed light on how the pattern functions, its advantages, and the scenarios in which it is most effectively employed, thereby offering valuable insights into advanced Kubernetes application design and deployment strategies.

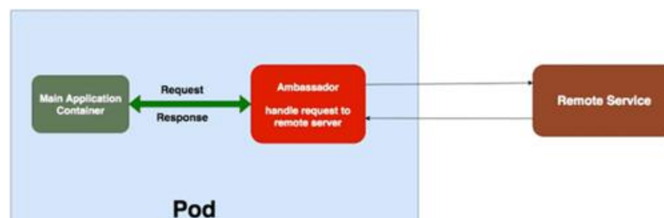
- What is an Ambassador Container
- Other Patterns

- Example Project
- Test With Deployment Object
- How to Configure Resource Limits
- When should we use this pattern?
- Summary
- Conclusion

1. What is an ambassador container?

The Ambassador container is a special type of sidecar container which simplifies accessing services outside the Pod. When you are running applications on kubernetes it's a high chance that you will access the data from the external services. The Ambassador container hides the complexity and provides the uniform interface to access these external services.

Imagine that you have the pod with one container running successfully but, you need to access external services. But these external services are dynamic in nature or difficult to access. Sometimes there is a different format that external service returns. There are some other reasons as well and you don't want to handle this complexity in the main container. So, we use the Ambassador containers to handle these kinds of scenarios.



One more thing we need to notice in the above diagram, you can define any number of containers for Adapter containers and your main container works along with it successfully. All the Containers will be executed parallelly and the whole functionality works only if both types of containers are running successfully. Most of the time these ambassador containers are simple and small and consume fewer resources than the main container.

2. Other Patterns

There are other patterns that are useful for everyday Kubernetes workloads.

- Init Container Pattern
- Sidecar Container Pattern
- Ambassador Container Pattern

3. Example Project

Here is an example project you can clone and run it on your machine. You need to install Minikube as a prerequisite.

https://github.com/bbachi/k8s-ambassador-container-pattern.git

For simplicity, imagine you have the public REST API and you have a main container that needs this data but you don't want to handle calling this API so you have to use the Ambassador container which handles this for the main container. With this approach, your main container doesn't need to have all the external service details. Let's take a mock public API here https://api.mocki.io/v1/b043df5a which returns the following JSON response.

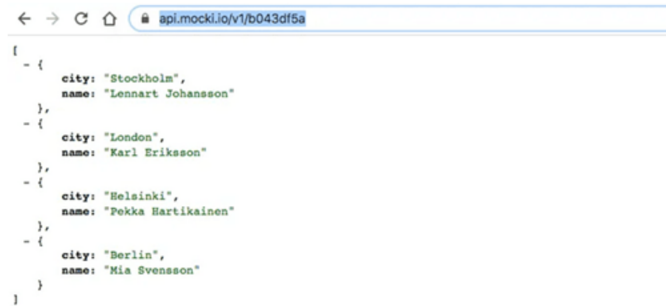


Figure 1: JSON response

A.Ambassador Container

Let's look at the Ambassador container which is a simple NGINX server that acts as a reverse proxy. Here is the simple nginx.conf file.

https://gist.github.com/bbachi/777bafb5994c428533c51f6e9ca9d4b7#file-nginx-conf

Here is the Dockerfile that builds the image.

https://gist.github.com/bbachi/e8583b2789e5b3eed9f75809308ded4b#file-dockerfile

Let's build this image and push it to Docker Hub.

// build the imagedocker build -t nginx-server-proxy

// tag the imagedocker tag nginx-server-proxy bbachin1/nginx-server-proxy

// push the imagedocker push bbachin1/nginx-server-proxy

B. Main Container

The main container is the nodejs express server that serves port 9000. This server calls the reverse proxy, and this proxy intern calls the actual API. Here is the server.js file.

https://gist.github.com/bbachi/45479563add9510099b25e1bca7380e2#file-server-js

Here is the Dockerfile that builds the image.

https://gist.github.com/bbachi/c0363d05d0e3dd695d2c740134054cc0#file-dockerfile

Let's build this image and push it to Docker Hub.

// build the imagedocker build -t main-container

// tag the imagedocker tag main-container bbachin1/main-container

// push the imagedocker push bbachin1/main-container

Here are the two images in the DockerHub.

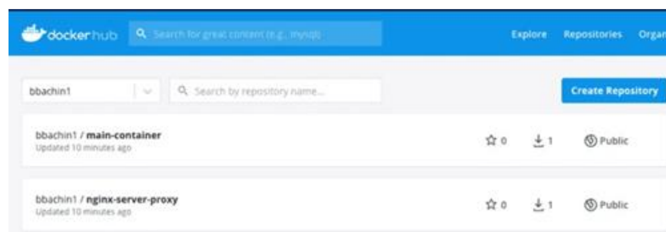


Figure 2: Docker Hub

Now the both Docker images are ready let's create a Pod with Ambassador container pattern with nginx-server-proxy acts as an ambassador container here which hides the complexity or details of the external services. The main container accesses the API as it is communicating on the same network since all the containers in the pod share the localhost and port space. Here is the pod.yml file.

https://gist.github.com/bbachi/060051c6c0e4e5d227bec494a36d0940#file-pod-yml

// create the podkubectl create -f pod.yml

// list the podskubectl get po

// exec into podkubectl exec -it ambassador-container-demo -c ambassador-container -- /bin/sh# curl localhost: 9000

You can install curl and query the local host: 9000 and check the response. Notice that the main container is running on the port **9000**.

```
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl create -f pod.yml
pod/ambassador-container-demo created
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl get po
NAME                        READY   STATUS    RESTARTS   AGE
ambassador-container-demo   2/2     Running   0          26s
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl exec -it ambassador-container-demo -c am
bassador-container -- /bin/sh
/ # curl localhost:9000
[{"city":"Stockholm","name":"Lennart Johansson"},{"city":"London","name":"Karl Eriksson"},{"city":"Helsinki","name":"P
ekka Hartikainen"},{"city":"Berlin","name":"Mia Svensson"}]/ #
/ #
```

Figure 3: Testing Ambassador Container

4. Test With Deployment Object

Let's create a deployment object with the same pod specification with 5 replicas. I have created a service with the port type NodePort so that we can access the deployment from the browser. Pods are dynamic here and the deployment controller always tries to maintain the desired state. That's why you can't have one static IP Address to access the pods so that you must create a service that exposes the static port to the outside world. Internally service maps port **9000** based on the selectors. You will see that in action in a while.

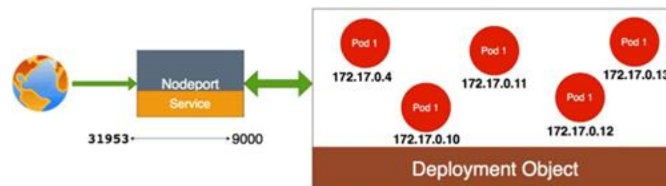


Figure 4: Deployment

Let's look at the below deployment object where we define one main container and one ambassador container. All the containers run in parallel. The main container is running on port **9000** which maps to the outside world through the service NodePort **31953**. The main container internally calls the Ambassador container running on **3000** that calls the external endpoint.

https://gist.github.com/bbachi/e7a70e54018aab52f9296514d33e1e9b#file-manifest-yml

Let's follow these commands to test the deployment.

// create a deployment

kubectl create -f manifest.yml

// list the deployment, pods, and service

kubectl get deploy -o wide

kubectl get po -o wide

kubectl get svc -o wide

```
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl create -f manifest.yml
deployment.apps/node-server created
service/node-server created
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl get deploy -o wide
NAME          READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS                                IMAGES
  SELECTOR
node-server   5/5     5            5           26s   main-container,ambassador-container   bbachin1/main-container,bbachin1/nginx-server-prox
  app=node-server
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl get po -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED NODE   READINESS GATES
node-server-65fc7775df-9hn6n   2/2     Running   0          33s   172.17.0.7   minikube   <none>           <none>
node-server-65fc7775df-k9nl5   2/2     Running   0          33s   172.17.0.6   minikube   <none>           <none>
node-server-65fc7775df-nxq9x   2/2     Running   0          33s   172.17.0.5   minikube   <none>           <none>
node-server-65fc7775df-t6bvj   2/2     Running   0          33s   172.17.0.8   minikube   <none>           <none>
node-server-65fc7775df-v6jbv   2/2     Running   0          33s   172.17.0.2   minikube   <none>           <none>
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl get svc -o wide
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE   SELECTOR
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP          95d   <none>
node-server   NodePort    10.110.83.137   <none>        9000:31953/TCP   41s   app=node-server
vue-webapp    NodePort    10.99.91.168    <none>        8080:32358/TCP   8d    app=vue-webapp
Bhargavs-MacBook-Pro:k8s-ambassador-container-pattern bhargavbachina$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.2:8443
KubeDNS is running at https://192.168.64.2:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 5: Deployment in action

In the above diagram, you can see **5** pods running in different IP addresses and the service object maps the port **31953** to the port **9000**. You can access this deployment from the browser from the kubernetes master IP address **192.168.64.2** and the service port **31953**.

http://192.168.64.2:31953/

Figure 6: Adapter container pattern

You can even test the pod with the following commands

```
// exec into main container of the pod
kubectl exec -it ambassador-container-demo -c ambassador-container -- /bin/sh
# curl localhost:9000
```

5. How To Configure Resource Limits

Configuring resource limits is very important when it comes to Ambassador containers. The main point we need to understand here is All the containers run in parallel so when you configure resource limits for the pod you have to take that into consideration.

- The sum of all the resource limits of the main containers as well as ambassador containers (Since all the containers run in parallel)

6. When Should We Use This Pattern

These are some of the scenarios where you can use this pattern.

- Whenever you want to hide the complexity from the main container such as service discovery.
- Whenever your containerized services want to talk to external services you can use this pattern to handle the request and response for these services.
- Whenever there is a need to convert or standardize the format of external services responses.

7. Summary

- A pod that contains one container refers to a single container pod and it is the most common Kubernetes use case.
- A pod that contains Multiple co-related containers refers to a multi-container pod.
- The Ambassador container pattern is a specialization of sidecar containers.
- Ambassador containers run in parallel with the main container. So that you need to consider resource limits of ambassador containers while defining request/resource limits for the pod.
- The application containers and Ambassador containers run in parallel which means all the containers run at the same time. So that you need to sum up all the request/resource limits of the containers while defining request/resource limits for the pod.
- You should configure health checks for Ambassador containers as main containers to make sure they are healthy.
- All the pods in the deployment object don't have static IP addresses so that you need a service object to expose to the outside world.
- The service object internally maps to the port container port based on the selectors.
- You can use this pattern where you want to hide the complexity such as service discovery from the main container.

8. Conclusion

In conclusion, this paper has systematically explored the architecture and management of Kubernetes pods, distinguishing between the single-container and multi-container configurations, with a focus on the latter's increased complexity and versatility. The Ambassador container pattern, a subset of the sidecar container

approach, has been highlighted for its ability to run parallel to the main container, necessitating careful consideration of resource limits and health checks to ensure system stability and efficiency. Furthermore, we have underscored the importance of configuring health checks for Ambassador containers, akin to main containers, to maintain overall system health. The dynamic nature of pod IP addressing in Kubernetes deployments necessitates the use of service objects for external exposure, highlighting the intricate internal mechanisms that facilitate communication and service discovery within Kubernetes clusters.

The Ambassador pattern proves invaluable for abstracting complex networking functionalities like service discovery away from the main application container, thereby simplifying application logic and enhancing maintainability. This pattern is particularly beneficial in scenarios requiring the seamless integration of services while maintaining clear separation of concerns within containerized environments. Ultimately, this paper provides a foundational understanding and practical insights into the deployment and management of multi-container pods in Kubernetes, with specific emphasis on the Ambassador container pattern, offering a strategic approach to enhancing container orchestration and application scalability in modern cloud-native environments.

References

[1]. Kubernetes Documentation https://kubernetes.io/docs/home/

[2]. Docker Documentation https://www.docker.com/blog/

[3]. Container Patterns https://www.xenonstack.com/insights/kubernetes-container-design-patterns