



---

## Exploring the Adapter Container Pattern in Kubernetes

**Bhargav Bachina**

---

**Abstract** This paper presents a comprehensive guide for deploying MERN (MongoDB, Express.js, React.js, Node.js) stack applications on Amazon Web Services (AWS) Elastic Container Service (ECS). With the increasing popularity of MERN stack for web development and the growing adoption of cloud computing, efficient deployment strategies are essential. Leveraging AWS ECS offers scalability, reliability, and ease of management. The paper provides an overview of MERN stack components, AWS ECS features, and a step-by-step deployment process covering containerization, task definition creation, cluster setup, service configuration, and load balancing. Practical examples, code snippets, and best practices are discussed to empower developers and DevOps engineers in efficiently deploying MERN stack applications on AWS ECS, facilitating streamlined development workflows and robust, cloud-native solutions.

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

---

Kubernetes is an open-source container orchestration engine for automating deployment, scaling, and management of containerized applications. A pod is the basic building block of Kubernetes application. Kubernetes manages pods instead of containers and pods encapsulate containers. A pod may contain one or more containers, storage, IP addresses, and options that govern how containers should run inside the pod.

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. There are a few patterns for multi-container pods one of them is the adapter container pattern. In this post, we will see this pattern in detail with an example project.

- What is Adapter Container
- Other Patterns
- Example Project
- Test With Deployment Object
- How to Configure Resource Limits
- When should we use this pattern?
- Summary
- Conclusion

### 1. What are adapter containers

There are so many applications that are heterogeneous in nature which means they don't contain the same interface or are not consistent with other systems. This pattern extends and enhances the functionality of current containers without changing it as the sidecar container pattern. Nowadays, we know that we use container technology to wrap all the dependencies for the application to run anywhere. A container does only one thing and does that thing very well.

Imagine that you have a pod with a single container working very well but, it doesn't have the same interface as other systems to integrate or work with it. How can you make this container have a unified interface with a



standardized format so that other systems can to your container? This adapter container pattern really helps exactly in that situation.

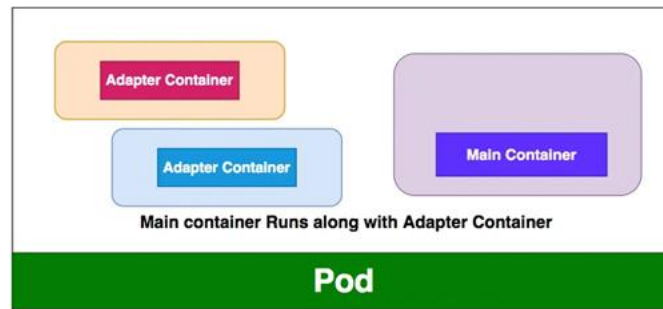


Figure 1: Adapter Container Pattern

If you look at 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 adapter 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
- Adapter Container Pattern
- Ambassador Container Pattern

## 3. Example Project

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

<https://github.com/bbachi/k8s-adaptor-container-pattern.git>

Imagine your main container generates logs in text format and external systems need to consume these logs in a JSON format. You need to have some mechanism to convert text files to JSON format.

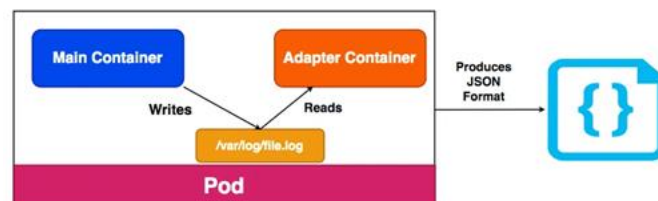


Figure 2: Adapter Pattern

The Adapter container is a simple express node server that reads from the location `/var/log/file.log` and produces JSON format. Let's implement a simple project to understand this pattern. The Adapter container is a simple express API that serves these logs as a JSON response. Here is the `file-server.js`

<https://gist.github.com/bbachi/c00c6e6a1a2e74a97f634ce7106863ba#file-server-js>

Here is the Dockerfile that converts this app into a Docker image.

<https://gist.github.com/bbachi/e5ab5e0cd85a5c7ff7fc9f32e0fe40ed#file-dockerfile>

The following are the commands creating a Docker image and pushing into Docker Hub.

```
// build the Docker image docker build -t bbachin1/adapter-node-server
```

```
// list the images docker images
```

```
// push the image docker push bbachin1/adapter-node-server
```



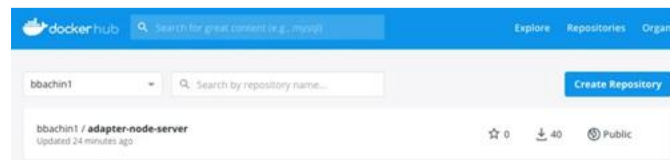


Figure 3: Docker Hub

Once the Docker image is ready and pushed into Docker Hub and it can be available publicly. You can pull it when creating a pod. Here is a simple pod that has a main and adapter container. The main container is busybox generating some logs to the path `/var/log/file.log` from the volume mount `workdir` location. Since the Adapter container and main container runs parallel node express API will display the new log information every time you hit in the browser.

<https://gist.github.com/bbachi/75318bd1f3648a6dbe0b6371bfb1b6bf#file-pod.yml>

```
// create the podkubectl create -f pod.yml
```

```
// list the podskubectl get po
```

```
// exec into podkubectl exec -it adapter-container-demo -c adapter-container -- /bin/sh# apt-get update && apt-get install -y curl# curl localhost
```

You can install curl and query the local host and check the response.

```
Bhargavs-MacBook-Pro:k8s-adapter-container-pattern bhargavbachina$ kubectl create -f pod.yml
pod/adapter-container-demo created
Bhargavs-MacBook-Pro:k8s-adapter-container-pattern bhargavbachina$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
adapter-container-demo  2/2     Running   0           11s

done.
# curl localhost:3080/logs
{"time":"Sun Sep 13 03:20:09 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:14 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:19 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:24 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:29 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:34 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:39 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:44 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:49 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:54 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:20:59 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:21:04 UTC 2020","message":"This is log"}, {"time":"Sun Sep 13 03:21:09 UTC 2020","message":"This is log"}

```

Figure 4: Testing Adapter 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 to port **3080** based on the selectors. You will see that in action in a while.

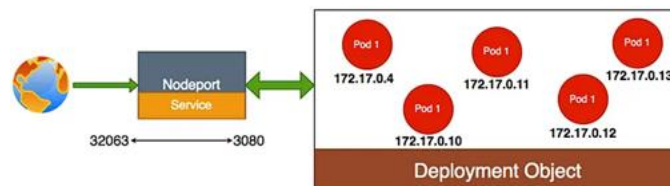


Figure 5: Deployment

Let's look at the below deployment object where we define one main container and one adapter container. All the containers run in parallel. The main container creates logs in the location `/var/log/file.log`. The adapter container serves those log files in a JSON format on the REST API on port **3080**. You will see that in action in a while.

<https://gist.github.com/bbachi/18acf3a40950adf1514d39cecd46760#file-manifest.yml>

Let's follow these commands to test the deployment.

```
// create a deploymentkubectl create -f manifest.yml
```

```
// list the deployment, pods, and servicekubectl get deploy -o widekubectl get po -o widekubectl get svc -o wide
```



```

Bhargava-MacBook-Pro:~$ kubectl create -f manifest.yml
deployment.apps/node-webapp created
service/node-webapp created
Bhargava-MacBook-Pro:~$ kubectl get deploy -o wide
NAME          READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS
node-webapp   5/5     5             5           18s   main-container,adapter-container
Bhargava-MacBook-Pro:~$ kubectl get pod -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP              NODE          NOMINATED NODE   READINESS GATES
node-webapp-6d5f9f5d4-71jxr  2/2     Running   0           38s   172.17.0.13     minikube     <none>            <none>
node-webapp-6d5f9f5d4-7kxvj  2/2     Running   0           38s   172.17.0.12     minikube     <none>            <none>
node-webapp-6d5f9f5d4-1pmet  2/2     Running   0           38s   172.17.0.18     minikube     <none>            <none>
node-webapp-6d5f9f5d4-wf2qg  2/2     Running   0           38s   172.17.0.11     minikube     <none>            <none>
node-webapp-6d5f9f5d4-q7k7b  2/2     Running   0           38s   172.17.0.14     minikube     <none>            <none>
Bhargava-MacBook-Pro:~$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.2:32063/api/v1/
KubeDNS is running at https://192.168.64.2:32063/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

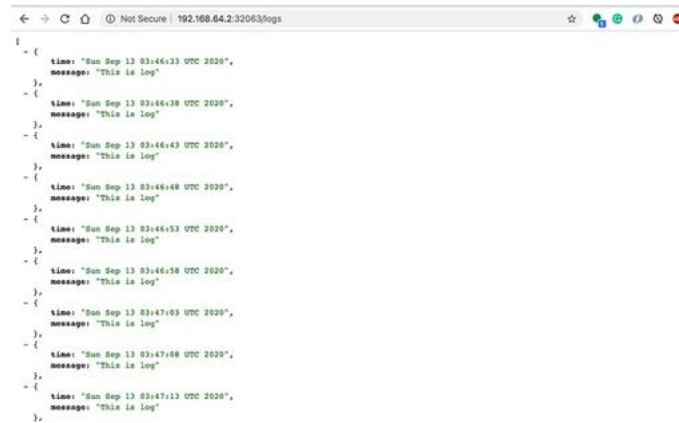
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
Bhargava-MacBook-Pro:~$ kubectl cluster-info dump
NAME          TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE   SELECTOR
kubernetes    ClusterIP    10.96.0.1    <none>        443/TCP    66d   <none>
node-webapp   NodePort     10.99.8.85   <none>        32063/TCP  2m28s app=node-webapp

```

Figure 6: Deployment in action

In the above diagram, you can see 5 pods running in different IP addresses and the service object maps the port **32063** to port 3080. You can access this deployment from the browser from the Kubernetes master IP address **192.168.64.2** and the service port **32063**.

<http://192.168.64.2:32063/logs>



```

{
  "time": "Sun Sep 13 03:46:33 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:46:38 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:46:43 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:46:48 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:46:53 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:46:58 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:47:03 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:47:08 UTC 2020",
  "message": "This is log"
},
{
  "time": "Sun Sep 13 03:47:13 UTC 2020",
  "message": "This is log"
},
}

```

Figure 7: Adapter container pattern

You can even test the pod with the following commands.

```

// exec into main container of the pod
kubectl exec -it <pod name> -c adapter-container -- /bin/sh
// install curl
apt-get update && apt-get install -y curl
curl localhost

```

## 5. How to Configure Resource Limits

Configuring resource limits is very important when it comes to Adapter 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 adapter 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 extend the functionality of the existing single container pod without touching the existing one.
- Whenever you want to enhance the functionality of the existing single container pod without touching the existing one.
- Whenever there is a need to convert or standardize the format for the rest of the systems.

## 7. Summary

- A pod that contains one container refers to a single container pod and it is the most common Kubernetes use case.
- The Adapter container pattern is a specialization of sidecar containers.



- The application containers and Adapter 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.
- All the pods in the deployment object don't have static IP addresses so that you need a service object to expose yourself to the outside world.
- The service object internally maps to the port container port based on the selectors.
- You can use this pattern where your application or main containers need to standardize some format for the external systems.

## **8. Conclusion**

In conclusion, Kubernetes stands as a powerful open-source tool for orchestrating containerized applications, offering automation and scalability benefits. Understanding the concept of pods, the fundamental units managed by Kubernetes, is crucial for effective application deployment. While single-container pods are common, multi-container pods present diverse architectural possibilities, including the adapter container pattern explored in this paper. Through the detailed examination of this pattern and its application in an example project, we've underscored its significance in facilitating efficient communication and collaboration between co-related containers within a pod. Furthermore, this paper has provided insights into other multi-container pod patterns, emphasizing the importance of choosing the right architecture based on specific use cases. Practical considerations such as configuring resource limits and determining the appropriate timing for pattern implementation have also been addressed, contributing to a comprehensive understanding of Kubernetes best practices.

## **References**

- [1]. Official Docker Guides <https://docs.docker.com/get-started/overview/>
- [2]. Official Kubernetes Docs <https://kubernetes.io/docs/home/>
- [3]. Container Design Patterns <https://kubernetes.io/blog/2016/06/container-design-patterns/>

