Zero to Cloud-Native with IBM Cloud

Part 2: Microservices Architecture and Design

Kevin Collins
Technical Sales Leader
IBM Cloud Enterprise Containers – Americas

Kunal Malhotra
Cloud Platform Engineer
IBM Cloud MEA

1 - Introduction

When developing a cloud native application, it is very important to take the time upfront to architect your application properly from the beginning. If you don't, you may end up creating a monolith that you will end up having to re-factor or you will have inefficient macro-services. While developing CloudPak Provisioner that I describe in Part 1 of this series, I ended up falling into both traps. While this did give me the experience of modernizing a monolith application, I could have saved myself a lot of pain and suffering if I spent the time upfront designing and architecting the application.

2 - Microservices Patterns

There are many microservices patterns that you can leverage for your cloud native application. For a deeper dive into microservices and microservices patterns, I recommend the following two articles:

Design Patterns for Microservices by Rajesh Bhojwani:
https://dzone.com/articles/design-patterns-for-microservices

Microservices by Martin Fowler.
https://martinfowler.com/articles/microservices.html

For an API Application like the one I will discuss in this blog series and we will implement in the accompanying tutorial is based on, there are a variety of patterns you can follow to ensure you are starting with good design. I will be used the concept of the API Gateway Pattern and the Command Query Responsibility Segregation (CQRS) pattern. You can read more about these patterns in the links above.

3 - Getting Started With a Monolith

Don't repeat my mistake! When I started developing the CloudPak Provisioner application, I started by simply creating a frontend and backend. I knew it was very important to separate the frontend from the backend so they were not dependent on one another. This is the absolute minimum microservices separation to consider. However, there is much more to good microservices design than separating the frontend from the backend.

The backend 'microservice' that I started with, was a single microservice that implemented the API Gateway pattern. I leveraged message queues to have the frontend communicate with the backend through messaging using IBM Cloud's RabbitMQ service. The front-end would simply take an API request and put the request data as a message onto a single queue. The backend application would then pick up the message and process the request. I quickly realized a couple of things. For some of the APIs, I needed them to return immediately such as functions that would drive fields on a webpage. Secondly, every time I had to make a change to the backend this required deploying the entire backend code, even for APIs that were not changed. This was not very efficient.

While developing the application and adding new 'real-time' functionality, I ended up adding these real-time APIs to the frontend microservice. This violated one of the core principals of developing a 12-factor app. I was mixing the frontend functionality with 'backend' logic. This resulted in a bad experience where I had a dependency on the frontend with the real-time backend API logic.

I had a similar experience with having single backend monolith 'microservice'. After about a month of different teams in IBM using part of my CloudPak Provisioner I thought of new APIs, features and functions to add. Having a single backend service meant that adding a new API or feature resulted in having to update and re-deploy the entire backend code base. This resulted in bad development experience by having so many dependencies. I learned an important lesson, take the time to properly architect your cloud-native application before developing it.

4 - Breaking Down the Monolith

After experiencing the pain of not taking the time in designing my microservices properly, I went back to the architecting drawing board. I ended up creating a hybrid pattern of leveraging both the API Gateway and CQRS pattern. I ended up categorizing the APIs into two ways:

1) Realtime APIs that need to return quickly and immediately. You can view these APIs as the query part of CQRS.
2) Command APIs that did not need to return data immediately to a user. These typically were longer running tasks that performed in the background and leveraging messaging for communication.

4.1 - Realtime APIs:

How can you call one microservice from another in real-time? There are many techniques, but one that I like to use is service discovery which is already built into Kubernetes. When you create a microservice one of the most common ways to implement and deploy the microservice in Kubernetes is through a deployment. With a deployment, you indicate how many instances (i.e. pods) of the microservice that you want running. To then expose the microservice so that it can be called from other microservices you create a service. One of the types of services that you can create is called Cluster IP. A Cluster IP service only exposes itself on the cluster through an internal IP which makes the service only reachable from within the cluster.

For my real-time APIs, I only wanted these backend real-time APIs to be called from the Frontend. I didn't want the backend to be called directly from the end user. To enable this, I exposed each real-time backend API as a Cluster IP service.

But how can you then call the service since it is only visible in the cluster? The answer is … automatically with Kubernetes. One of the many great features of Kubernetes is service discovery. With Kubernetes, you can run commands directly on the container. With each container being built on Kubernetes, one of the things you can do is view the container environment variables with the `env` command.

This will return you a dynamic list of all environment variables including host and port information of all the services you have in the cluster.

For example, after running the env command on any of the pods I have running in my cluster and grepping for one of the real-time services I have, I get the following result.

```
# env | grep OPENSHIFT_REALTIME_SERVICE
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_PORT_8220_TCP_ADDR=172.21.131.40
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_PORT_8220_TCP_PORT=8220
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_PORT_8220_TCP_PROTO=tcp
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_SERVICE_HOST=172.21.131.40
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_PORT=tcp://172.21.131.40:8220
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_SERVICE_PORT=8220
CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_PORT_8220_TCP=tcp://172.21.131.40:8220
#
```

You can tell from this screenshot that Kubernetes knows all the information about the internal service that I have exposed. But what good is this, how can I programmatically use this and what happens if the IP address changes? Kubernetes helps with this as well by using OS environment variables.

Below is sample Python code the constructs a REST API url to call the service that will retrieve an OpenShift Login Token:

```
port = os.environ.get("CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_SERVICE_PORT")
url = "http://"+os.environ.get("CLOUDPAK_PROVISIONER_OPENSHIFT_REALTIME_SERVICE_SERVICE_HOST")
openshift_realtime_url = url+":"+port+"/api/v1/getRoksToken/"
```

I can now call the backend microservice that returns an OpenShift login token by only using the name of the service I exposed. If the IP of service would change it would be no problem as the operating system environment variable is automatically updated by Kubernetes.

4 – 2 Command APIs:

For APIs that take a longer time to run or ones that you don't necessarily need to do anything with their output, a great design technique is to leverage messaging. In this design pattern, a backend process listens on a queue. When a message lands on the queue, the message is picked up and then processed independently. The benefit of this, is that the frontend API returns very quickly indicating that the message request was received and then the work is being done on the backend.

This sounds great, but how can we track if a frontend request actually completes successfully on the backend. One technique that I used to solve this was leveraging request IDs. I simply created a random string of 6 characters for each frontend request and then in each message that is sent to the backend, I added this request id to the backend message. Then to track that status of each request, I prepended every log message with the request ID. As you will see in the IBM Log Analysis with LogDNA session, I can quickly see the logs across all microservices from the request id. I can also easily share the status of each request with the end user.

5 - Microservices Granularity

How do you know if you need to break apart microservices into more granular microservice and how can you best get started breaking apart a monolith? In my case, after thinking about how to break apart my monolith, I turned to Domain Driven Design. Thinking about the categories of each API, I treated each as its own domain. I started by having each domain equaling one microservice for real-time backend APIs and another microservice for longer running command APIs. The categories, or domains, I had were for IBM Cloud Object Storage, Portworx, Classic Infrastructure, VPC, OpenShift, CloudPak For Data, and general utility. You can view each as its own domain that are completely independent of the other domains. Refer to part one of this series to read more about each of the categories of microservices.

Breaking down the microservices was a great start. I saw a tremendous improvement in overall application performance as each domain was independent of each other. However, there was one domain still causing performance concerns with end users. The cloud object storage category had several long running tasks which resulted with users expressing concern on how long the APIs were taking. After debugging the issue, the performance problems occurred when there were multiple users calling different Cloud Object Storage APIs. At this point, I had two choices. I could either scale the pods of the deployment to handle more concurrent requests or refactor the microservice into more granular microservices. In this case, adding more pods would have been the easy way out, not using resources (compute) wisely, and not really addressing the issue of proper microservices design. After learning from my experience at the beginning, I decide to follow good cloud-native design. Looking at the APIs that were in the Cloud Object Storage domain, I had APIs for:

- Retrieving a list of buckets with metadata
- Deleting a bucket and all objects
- Add metadata to a bucket
- Retrieve a list of bucket metadata
- Copy one COS bucket and all objects to another bucket
- Creating a COS PVC on a Kubernetes cluster
- Load COS Buckets in Redis for caching

Following the CQRS pattern, I created a single real-time API for retrieving the list of COS buckets and for retrieving a list a metadata. Why did I decide on a single microservice for handling these two functions? There is a certain amount of overhead for each microservice. Since these two APIs were very short living and return quickly the best design is to combine these APIs into one microservice and then scale if needed. I also don't expect these APIs to be hit 1000s of times per minute, so usage wasn't a concern.

The rest of the microservices were more independent of each other and were longer running tasks. If someone was using the API to create new buckets and attaching the buckets to a PVC, I didn't want that longer running task preventing someone else from being able to add metadata to a bucket. Because I had already implemented queuing, refactoring the microservice to be

more granular turned out to be very easy. All I had to do was create a new queue for each microservice! The rest of the code stayed the same.

6- Tutorial

If you read part 1 – introduction to this series, you will remember that the tutorial application will comprise of three APIs:

1) **getOCPVersions** – will return a list of OpenShift versions supported by IBM Cloud Managed OpenShift
2) **getOCPToken** – will return an OCP token to login into your cluster.
3) **enableNodeSSH** – will enable your IBM Cloud Managed OpenShift worker nodes to be SSH'd into.

Other key features of the application will be:
Web and API frontends for the application.
Cronjob to load OCP Versions into Redis.

Pop Quiz - How would you architect this simple cloud native application from a microservices perspective? Which microservices patterns would you recommend?

See the suggest design on the next page.

You will create the following microservices in this tutorial.

- **frontend-web** – HTML front-end to invoke the three public facing APIs
- **frontend-api** – API interface to call the public facing APIs
  * notice that each frontend user face has it's own micro-service, following good application design using the Backend for Frontend (BFF) design pattern.

- **ocp-realtime** – real-time API which will have two endpoints. The first one will quickly return a list of OCP Versions leveraging a Redis Cache. The second one will return a login token for OCP.
- **loadOCPVersions** – Kubernetes cronjob that will load current versions of OCP supported by IBM Cloud into a Redis database.
- **enableNodeSSH** – command API for the 'long' running task to enable SSH on an IBM Cloud Managed OpenShift worker node.