Supercharging Kubernetes

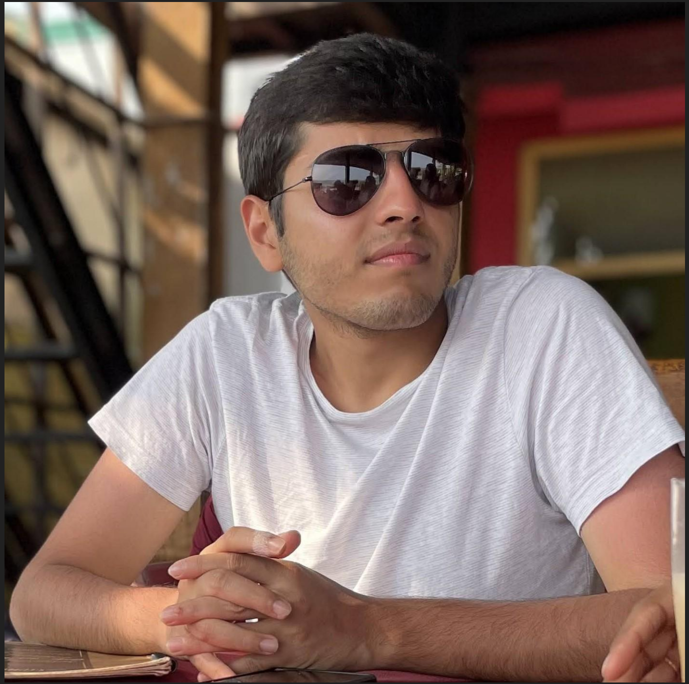
Writing controllers in python

About Me: Yash Mehrotra

Backend, Infrastructure & Platform

Software Engineer @ Flanksource

Prev: BlinkIt, MindTickle



DevOps is a software engineering culture of putting horrors into containers and then talking about Kubernetes at conferences.

The Philosophy of Kubernetes

Small detour: Kubernetes 101

- A container-orchestration system for automating application deployment, scaling, and management.
- Works on declarative configuration principle
- You apply YAML or JSON manifests and the system ensures the desired state is achieved

Kubernetes 101: Terminology

Namespace: A mechanism to provide virtual isolation for resources

Pods: Group of one or more containers

Deployment: Maintain a set of replica pods

Service: Route requests to a group of pods via labels

Secret: Store key-value pairs which can be used by pods

Our imaginary story

- We work at TicketSellers
- They handle the sales for almost all concerts and events
- Existing infrastructure was not scalable
- Someone suggested to migrate to Kubernetes

Our imaginary story

- Thought kubernetes would solve all their problems
- Planned a very long and painful migration
- Finally everything was migrated to Kubernetes
- … the problems still existed

When you take a trash pile and deploy that to Kubernetes, all you get is a containerized orchestrated trash pile

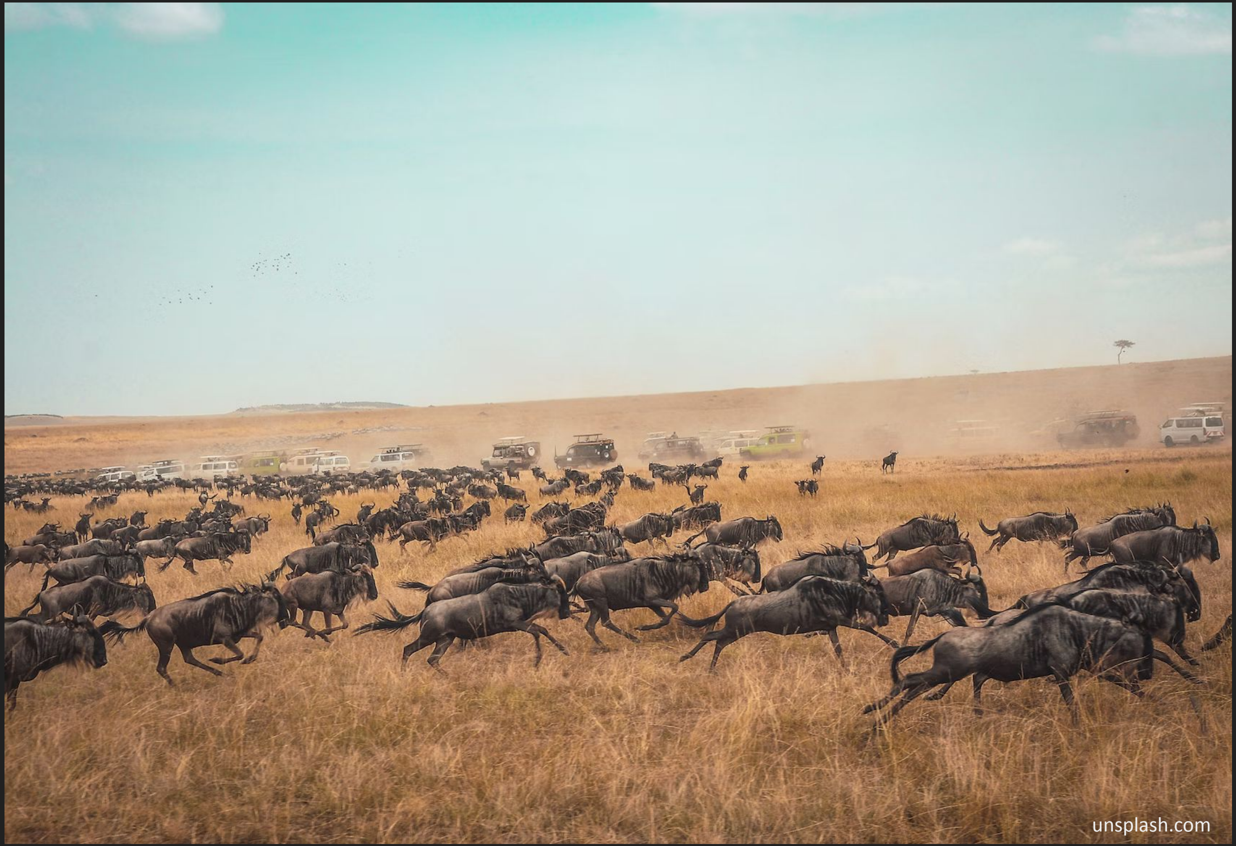
New Platformer in the house

- Danny joins TicketSellers
- He is not impressed by the processes and decision-making
- Since Danny is in platform team, they ask him to solve the scaling problems

The problems

- Previously, new VMs were spawned but they took time to bootstrap and start up
- On kubernetes, they setup and start fast
- But when and how to spawn the extra ones ?
- Can't they do autoscaling on metrics like CPU, Memory or Latency

The thundering herd problem



The thundering herd problem

- System designed for a set amount of load
- Too many requests for certain entities
- Usually simpler to scale up on gradual load
- Not easy for generic auto-scaling to handle these scenarios

In the midst of chaos, there is also opportunity

Brainstorming

- What if we try to forecast the load ?
- What if we spawn dedicated servers for those scenarios ?
- What if we launch a new database at runtime ?

A suggested solution

- Forecast the load based on social media followers
- Before the start of the selling window:
  - Spawn new servers just to handle that load
  - Create a new database for that event
  - Update URL routing to direct those requests

More problems ...

- This is like deploying a new application for every major event
- Will we write new manifests everytime ?
- How will we launch new databases and point apps to them ?
- Different routes for so many different events ? That's not easy to manage !
- Who will clean all of it up after its done ?

Time for controllers

- Danny looks at the team and asks that since we are already on kubernetes, why not just write a controller
- It will take care of the bootstrapping and cleanup
- Only have to write the glue code
- But … no one in the team understood what he meant

Who wishes to fight must first count the cost

Forecasting load

1. We already have social media profiles of all the events
2. Based on our collected data and models, we can estimate the load and how many servers (pods) to run for all of them
3. Before the window opens we deploy the new pods, the database and update the rules

Forecasting load

```
def forecast():
    upcoming_events = event_db.GetUpcomingEvents(days=1)
    for event in upcoming_events:
        follower_count = get_follower_count(event.social_media_urls)
        server_count = calculate_server_count(follower_count)
        tenant_db.StoreNewTenant(
            id=event.id,
            name=event.name,
            path=event.path,
            server_count=server_count,
        )
```

Spawning new workload

```python
def create_deployment(tenant):
    # Generate DB credentials and store in kubernetes secret
    username = generate_random_string()
    password = generate_random_string()
    create_kubernetes_secret(
        tenant.name,
        data={'username': username, 'password': password}
    )

    deployment_spec = TENANT_DEPLOYMENT_TEMPLATE.format(
        name=tenant.name,
        replicas=tenant.server_count,
    )
    client.AppsV1Api().create_namespaced_deployment(
        body=deployment_spec,
        namespace='production',
    )

    service_spec = TENANT_SERVICE_TEMPLATE.format(name=tenant.name)
    client.CoreV1Api().create_namespaced_service(
        body=service_spec,
        namespace='production',
    )
```

Spawning new workload

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ticket-seller-tenant-{name}
  labels:
    app: ticket-seller-tenant-{name}
```

```yaml
spec:
  replicas: {replicas}
  selector:
    matchLabels:
      app: ticket-seller-tenant-{name}
  template:
    metadata:
      labels:
        app: ticket-seller-tenant-{name}
    spec:
      containers:
      - name: ticket-seller
        image: registry.example.com/ticket-seller-app
        envFrom:
          secretRef:
            name: {name}
        ports:
        - containerPort: 80
```

Spawning new workload

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ticket-seller-tenant-{name}
spec:
  selector:
    app: ticket-seller-tenant-{name}
  ports:
    - port: 80
      targetPort: 80
```

Creating a new database

```
def create_database(tenant):
    secret = get_kubernetes_secret(tenant.name)
    cloud_provider.CreateDatabase(
        name=tenant.name,
        username=secret.data.username,
        password=secret.data.password,
    )
```

How Kubernetes Ingress works ?

- Ingress object has a hostname and path directives binded to services
- Each service has multiple pods behind it
- Traffic gets routed via an ingress controller (nginx, traefik etc.)
- They route it to the pods based on the request params and the ingress object specification

How Kubernetes Ingress works ?

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api
```

```yaml
spec:
  ingressClassName: nginx
  rules:
  - host: api.example.com
    http:
      paths:
      - path: /v1
        pathType: Prefix
        backend:
          service:
            name: api-v1
            port:
              number: 80
      - path: /v2
        pathType: Prefix
        backend:
          service:
            name: api-v2
            port:
              number: 80
```

Updating routing

```python
def update_routes(tenant):
    ingress_lock.acquire()

    ingress_obj = client.NetworkingV1Api().read_namespaced_ingress(
        name='ticket-seller-ingress',
        namespace='production',
    )
    ingress_obj.spec.rules[0].http.paths.append(
        client.V1HTTPIngressPath(
            path=tenant.path,
            path_type='Prefix',
            backend=client.V1IngressBackend(
                service=client.V1IngressServiceBackend(
                    port=client.V1ServiceBackendPort(number=80),
                    name='ticket-seller-tenant'+tenant.name,
                )
            )
        )
    )
```

Updating routing

```
def update_routes(tenant): # Continued ...

    client.NetworkingV1Api().patch_namespaced_ingress(
        name='ticket-seller-ingress',
        namespace='production',
        body=ingress_obj,
    )

    ingress_lock.release()
```

Other benefits

- Each tenant scales individually
- Isolated Telemetry
- One tenant per event, no noisy neighbour problem
- Easy to cleanup

Demo Time

Takeaways

- Kubernetes can be extended in anyway you like
- Writing python controllers for kubernetes is a breeze
- Finding the correct problem to solve is very important
- There is always a better way

Thank you

yashmehrotra.com

@yashm95