From Monolith to Micro-services with Kubernetes

16 Mar 2019, FOSS Asia, Singapore

Michael Bright, @mjbright

Slides & source code at https://mjbright.github.io/Talks
Michael Bright, @mjbright

Freelance Consultant & Trainer on CloudNative Solutions

Past researcher, dev, team lead, dev advocate

British, living in France for 27-years

Docker Community Lead, Python User Group

linkedin.com/in/mjbright  github.com/mjbright
Outline

- [Why?] Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary
Outline

• [Why?] Monoliths to Micro-services
• Orchestration: Kubernetes
• Deployment Strategies
• Architecture Design patterns
• Summary
First ... a bit of history

Baremetal Servers → Virtual Machines → Cloud: IaaS, PaaS, SaaS ... → Containers: LXC, Docker, rkt → Serverless: AWS Lambda

Toward smaller, faster, cheaper solutions with easier management enabling faster time to market
First ... a bit of history

Monolithic apps → N-tier apps → Micro-services → Nano-services

Baremetal Servers → Virtual Machines → Cloud: IaaS, PaaS, SaaS → Containers: LXC, Docker, rkt → Serverless: AWS Lambda

Toward smaller, faster, cheaper solutions with easier management enabling faster time to market

Note: The future will be hybrid ... (technologies, providers, on-prem/cloud ...)

@mjbright
[Why?] Monoliths to Micro-services

Traditionally software has been delivered as large packages which can only be deployed, scaled, upgraded, reimplemented as a whole.

Problem: A paradigm ill-adapted to enterprise or web-scale
[Why?] Monoliths to Micro-services

Traditionally software has been delivered as large packages which can only be deployed, scaled, upgraded, reimplemented as a whole.

Problem: A paradigm ill-adapted to enterprise or web-scale

- Tightly-coupled components exist as a unit, are difficult to reuse
- Waterfall release cycles make software difficult to patch
- Difficult to innovate due to slow release cycles
Monoliths to Micro-services

Micro-services use small loosely-coupled software components

Individual components can be **deployed, scaled, upgraded, replaced** ...

Micro-service architecture components are lightly-coupled

- interconnected by network
- can be scaled independently
- can be deployed/upgraded independently
Advantages of Micro-services

Separation of Concerns: "do one thing well"
Advantages of Micro-services

Separation of Concerns: "do one thing well"

Smaller *focussed* Projects/teams
Advantages of Micro-services

Separation of Concerns: "do one thing well"

Smaller *focussed* Projects/teams

Ease Scaling, Deployment, Testing, Evolution
Advantages of Micro-services

Separation of Concerns: "do one thing well"

Smaller *focussed* Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Allow for composition of new services
Advantages of Micro-services

Separation of Concerns: "do one thing well"

Smaller *focussed* Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Allow for composition of new services

Can be re-implemented with "Best in class" tech
Advantages of Micro-services

Separation of Concerns: "do one thing well"

Smaller *focussed* Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Allow for composition of new services

Can be re-implemented with "*Best in class*" tech

So are they a panacea?
Disadvantages

Greater complexity

- Require orchestration, and rigorous component version management
- Need to *evolve* to greater *organizational* complexity
- Monitoring, debugging, end-2-end test are more difficult
Disadvantages

Greater complexity

- Require orchestration, and rigorous component version management
- Need to evolve to greater organizational complexity
- Monitoring, debugging, end-2-end test are more difficult

Network communication is critical

- Need good error handling, Performance, Circuit-breakers
Disadvantages

Greater complexity
- Require orchestration, and rigorous component version management
- Need to evolve to greater organizational complexity
- Monitoring, debugging, end-2-end test are more difficult

Network communication is critical
- Need good error handling, Performance, Circuit-breakers

Useless without adopting best practices
- Behaviour and Test-Driven Development, CI/CD
- Require rigorous documentation of interfaces/APIs
- Stable APIs and backward-compatibility support
Outline

- [Why?] Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary
Orchestration: Kubernetes

**Problem:** As our systems scale it becomes impossible to manage 1000's of diverse containers running across a data center of 100's of nodes.

- on which nodes should you schedule?
  - to ensure availability
  - to satisfy affinity, non-affinity constraints
  - to take advantage of specialized h/w
Orchestration: Kubernetes

**Problem:** As our systems scale, it becomes impossible to manage 1000's of diverse containers running across a data center of 100's of nodes.

- on which nodes should you schedule?
  - to ensure availability
  - to satisfy affinity, non-affinity constraints
  - to take advantage of specialized h/w
- which containers are malfunctioning?
Problem: As our systems scale it becomes impossible to manage 1000's of diverse containers running across a data center of 100's of nodes.

- on which nodes should you schedule?
  - to ensure availability
  - to satisfy affinity, non-affinity constraints
  - to take advantage of specialized h/w
- which containers are malfunctioning?
- which are started and ready to go?
Orchestration: Kubernetes

**Problem:** As our systems scale it becomes impossible to manage 1000's of diverse containers running across a data center of 100's of nodes.

- on which nodes should you schedule?
  - to ensure availability
  - to satisfy affinity, non-affinity constraints
  - to take advantage of specialized h/w
- which containers are malfunctioning?
- which are started and ready to go?
- how to easily upgrade applications?
Orchestration: Kubernetes

**Problem:** As our systems scale it becomes impossible to manage 1000's of diverse containers running across a data center of 100's of nodes.

- on which nodes should you schedule?
  - to ensure availability
  - to satisfy affinity, non-affinity constraints
  - to take advantage of specialized h/w
- which containers are malfunctioning?
- which are started and ready to go?
- how to easily upgrade applications?
- how to auto-scale applications?
We need Orchestration
Orchestration Feature Wish-list

- **Health checks** - to Verify when a task is ready to accept traffic
- **Dynamic port-mapping** - Ports are assigned dynamically when a new container is spun up
- **Zero-downtime deployments** - Deployments do not disrupt end users
- **Service discovery** - Automatic detection of new containers and services
- **Auto scaling** - Automatically scale resources up or down based on the load

- **Provisioning** - New containers should select hosts based on resources and configuration

- **Other** - Load balancing, logging, monitoring, authentication and authorization, security... predictability, scalability, and high availability...
Kubernetes - Architecture

- Master Node
- Worker Node 1
- Worker Node 2
- Worker Node 3
Kubernetes - Master Nodes

- API
- etcd
- GUI (dashboard)
- Controller
- Scheduler
- CLI (kubectl)
Kubernetes - Worker Nodes

Kubelet

Container Engine

Pod
Pod
Pod
Pod
Pod

kube-proxy

kube-dns
dashboard

Add-ons

flat network
Kubernetes - Pods

Containers share some namespaces:
- PID, IPC, network, time sharing

Main container  Sidecar  Sidecar

same ip, e.g. 192.168.1.20

A pod houses one or more containers
Kubernetes Demo

Master Node
"Worker"

Docker Desktop tainted single-node

Flask1
Flask2
Flask3

Redis
Kubernetes - Deploying Redis

```bash
kubectl create -f redis-deployment.yaml
```

- **deployment**
  - **ReplicaSet**
    - **Pod1**
      - `2e76: redis`
Kubernetes - Deploying Redis
Kubernetes - Deploying Redis (yaml)
Kubernetes - Deploying Flask

```
kubectl create -f flask-deployment.yaml
```

**Deployment**

**ReplicaSet**

**Pod1**

2e76: flask:v1

**Pod2**

1f3d: flask:v1
Kubernetes - Deploying Flask

# kubectl run flask-app --image=$IMAGE --port=5000

$ kubectl apply -f flask-deployment.yaml
deployment.extensions "flask-app" created

$ kubectl get pods

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>flask-app-8577b44db-96cht</td>
<td>0/1</td>
<td>Pending</td>
<td>0</td>
<td>1s</td>
</tr>
<tr>
<td>redis-68595c4d95-rr4pr</td>
<td>0/1</td>
<td>ContainerCreating</td>
<td>0</td>
<td>1s</td>
</tr>
</tbody>
</table>
Kubernetes - Deploying Flask (yaml)

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: flask-app
name: flask-app
spec:
  replicas: 1
selector:
  matchLabels:
    run: flask-app
template:
  metadata:
    labels:
      run: flask-app
spec:
  containers:
  - image: mjbright/flask-web:v1
    name: flask-app
    ports:
    - containerPort: 5000
Operations - Scaling

```
# kubectl scale deploy flask-app --replicas=4
$ kubectl edit -f flask-deploy.yaml

...  
spec:  
  replicas: 4
```
Kubernetes - Scaling Flask (yaml)

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: flask-app
  name: flask-app
spec:
  replicas: 4
  selector:
    matchLabels:
      run: flask-app
  template:
    metadata:
      labels:
        run: flask-app
    spec:
      containers:
        - image: mjbright/flask-web:v1
          name: flask-app
          ports:
            - containerPort: 5000
```
$ kubectl apply -f flask-deployment-r4-v1.yaml
deployment.extensions "flask-app" created

$ kubectl get pods

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>flask-app-8577b44db-96cht</td>
<td>1/4</td>
<td>Pending</td>
<td>0</td>
<td>1h</td>
</tr>
<tr>
<td>redis-68595c4d95-rr4pr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1h</td>
</tr>
</tbody>
</table>
Outline

- [Why?] Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary
Deployment Strategies

**Problem:** How can we simply/automatically upgrade micro-services?

- across a data center
- in the cloud
Deployment Strategies

**Problem:** How can we simply/automatically upgrade micro-services?

- across a data center
- in the cloud

**Solution:** Several deployment strategies exist

- Some strategies can be implemented by Kubernetes alone
- Some strategies must be handled by external routing
Micro-service Deployment Strategies

Service Upgrade Strategies

Health Checks

Strangler Pattern - migration pattern
Operations - Service Upgrade Strategies

Several strategies exist

Ref: Kubernetes deployment strategies, Container Solutions, github

**recreate** - terminate old version before releasing new one
Operations - Service Upgrade Strategies

Several strategies exist

Ref: *Kubernetes deployment strategies, Container Solutions, github*

**recreate** - terminate old version before releasing new one

**ramped** - gradually release a new version on a rolling update fashion
Operations - Service Upgrade Strategies

Several strategies exist

Ref: *Kubernetes deployment strategies, Container Solutions, github*

- **recreate** - terminate old version before releasing new one
- **ramped** - gradually release a new version on a rolling update fashion
- **blue/green** - release new version alongside old version then switch
Operations - Service Upgrade Strategies

Several strategies exist

Ref: *Kubernetes deployment strategies, Container Solutions, github*

- **recreate**: terminate old version before releasing new one
- **ramped**: gradually release a new version on a rolling update fashion
- **blue/green**: release new version alongside old version then switch
- **canary**: release new version to subset of users, proceed to full rollout
Operations - Service Upgrade Strategies

Several strategies exist

Ref: *Kubernetes deployment strategies, Container Solutions, github*

**recreate** - terminate old version before releasing new one

**ramped** - gradually release a new version on a rolling update fashion

**blue/green** - release new version alongside old version then switch

**canary** - release new version to subset of users, proceed to full rollout

**a/b testing** - release new version to subset of users in a precise way (HTTP headers, cookie, weight, etc.).
Operations - Service Upgrade Strategies

Ramped

# kubectl set image deploy flask-app flask-app=mjbright/flask-web:v2

$ kubectl edit -f flask-deploy.yaml
$ kubectl rollout status deployment/flask-app

...

spec:
  containers:
    - image: mjbright/flask-web:v2
Master Node
"Worker"
Docker Desktop tainted single-node

Flask1
Flask2
Flask3

Redis
Containers - Are you healthy, ready?

**Problem:** But how can the system determine if a Service is healthy and available

We'd like the system to not route traffic to unhealthy service instances.
Containers - Are you healthy, ready?

**Problem:** But how can the system determine if a Service is healthy and available?

We'd like the system to not route traffic to unhealthy service instances.

**Kubernetes Healthchecks (Liveness and Readiness probes)** provide a solution.

Ref: Kubernetes Liveness, Readiness Probes Documentation

- Liveness probe can be used to force re-creation of blocked image
- Readiness probe can be used to await startup
Operations - Healthchecks

Liveness probes

- This probe is used to establish if the container is healthy (or blocked, unable to progress).
- The probe can specify
  - A command to execute
  - An http request to try
  - A TCP request to try
Operations - Healthchecks

Liveness probes

- This probe is used to establish if the container is healthy (or blocked, unable to progress).
- The probe can specify
  - A command to execute
  - An http request to try
  - A TCP request to try

Readiness probes

- Once started the container still needs time before being able to accept traffic
- This probe tests the readiness to receive and process requests
- Probe types are as for Liveness probes
Operations - Liveness probes

```yaml
apiVersion: v1
type: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```
Operations - Readiness probes

It is sufficient to replace 'livenessProbe:' by 'readinessProbe:' in the yaml

```
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```
How to Migrate to Micro-services?

**Problem:** We may not have the luxury of a *Greenfield* deployment!!

So how can we migrate an existing Monolith to Micro-services?
How to Migrate to Micro-services?

**Problem:** We may not have the luxury of a *Greenfield* deployment!!

So how can we migrate an existing Monolith to Micro-services?

It's a monolith after all!
How to Migrate to Micro-services?

**Problem**: We may not have the luxury of a *Greenfield* deployment!!

So how can we migrate an existing Monolith to Micro-services?

It's a monolith after all!

Do we wait 6 months before having a new implementation

(*with no extra features!*?)
How to Migrate to Micro-services?

**Problem:** We may not have the luxury of a *Greenfield* deployment!!

So how can we migrate an existing Monolith to Micro-services?

It's a monolith after all!

Do we wait 6 months before having a new implementation

(*with no extra features!*?)

The *Strangler* Pattern provides a possible solution.
Migration - Strangler Pattern

The Strangler is a pattern used in the initial migration from a Monolithic architecture to a Micro-services architecture

Ref: Azure Docs - "Strangler pattern"
Micro-service - Architecture Design Patterns

Here, we are not concerned with:

Standard Component *Design Patterns*

Micro-services themselves (!) - Fine-grained SOA

Sidecar
Micro-service - Architecture Design Patterns

We are concerned with:

Exposing Services

Ingress

providing access to the Kubernetes cluster ...
Micro-service - Architecture Design Patterns

We are concerned with:

**Exposing Services**

**Ingress**

providing access to the Kubernetes cluster ...

and ways of providing offload-functionality

**API Gateway**

**Service Mesh**

**Hybrid Apps - "API Gateway Pattern"**
Micro-service - Architecture Design Patterns

We are concerned with:

**Exposing Services**

**Ingress**

providing access to the Kubernetes cluster ...

and ways of providing offload-functionality

**API Gateway**

**Service Mesh**

**Hybrid Apps - "API Gateway Pattern"**

**Note:** This is the new war-zone as API Gateways battle it out, Service Meshes battle it out and both battle it out!
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster
But how do we access those services?
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster

But how do we access those services?

- We can access the Pods/containers directly at their IP and port addresses
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster
But how do we access those services?

- We can access the Pods/containers directly at their **IP** and **port** addresses

**Don't !!** - they are *ephemereal*
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster

But how do we access those services?

- We can access the Pods/containers directly at their **IP** and **port** addresses

**Don't !!** - they are *ephemereal*

- What happens if a Pod dies ... *it just might happen ;-)"
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster
But how do we access those services?

- We can access the Pods/containers directly at their IP and port addresses

**Don't !! - they are ephemereal**

- What happens if a Pod dies ... *it just might happen* ;-)

*(it's a joke: it will happen)*
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster

But how do we access those services?

- We can access the Pods/containers directly at their **IP** and **port** addresses

**Don't !!** - they are *ephemeral*

- What happens if a Pod dies ... *it just might happen ;-)*

*(it's a joke: it will happen)*

- Also - we don't want to **expose our infrastructure details !!**
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster

But how do we access those services?

- We can access the Pods/containers directly at their IP and port addresses

**Don't !!** - they are *ephemeral*

- What happens if a Pod dies ... *it just might happen ;-)*

*(it's a joke: it will happen)*

- Also - we don't want to **expose our infrastructure details !!**

- Also - they should be on isolated networks
Accessing our Services

**Problem:** We've deployed, scaled & upgraded Services across our Cluster

But how do we access those services?

- We can access the Pods/containers directly at their **IP** and **port** addresses

**Don't !!** - they are *ephemeral*

- What happens if a Pod dies ... *it just might happen ;-)*

(it's a joke: it *will* happen)

- Also - we don't want to **expose our infrastructure details !!**
- Also - they should be on isolated networks

So we provide *well-known endpoints* to reliably/safely **expose services**
Kubernetes - Exposing Services

The general pattern is to provide a *cluster-wide, well-known endpoint* which remains available as Pods come and go.
Design Pattern - Services

Services can be exposed via

NodePort
HostPort
ClusterIP
LoadBalancer
Exposing Services (NodePort)

User connects to IP/port of one of the Nodes
Exposing Services (LoadBalancer)

Master

User

Service

Worker

pod

pod

Worker

pod

pod

External Load Balancer

IP:port
Exposing Services (IngressController)
Exposing Redis Service (LoadBalancer)

```sh
# kubectl expose deployment redis --type=LoadBalancer

$ kubectl apply -f redis-service.yaml
service "redis" created

$ kubectl get svc
  NAME   TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)       AGE
  kubernetes  ClusterIP  10.96.0.1      <none>        443/TCP        5h
  redis      LoadBalancer 10.101.158.201 <pending>    6379:31218/TCP 1s
```
Exposing Redis Service (LoadBalancer)

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: redis
  name: redis
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    run: redis
  type: LoadBalancer
```
Exposing Flask Service (LoadBalancer)

```plaintext
# kubectl expose deployment flask-app --type=LoadBalancer

$ kubectl apply -f flask-service.yaml
service "flask-app" created

$ kubectl get svc
<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>flask-app</td>
<td>LoadBalancer</td>
<td>10.103.154.19</td>
<td>&lt;pending&gt;</td>
<td>5000:32201/TCP</td>
<td>1s</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.96.0.1</td>
<td>&lt;none&gt;</td>
<td>443/TCP</td>
<td>5h</td>
</tr>
<tr>
<td>redis</td>
<td>LoadBalancer</td>
<td>10.101.158.201</td>
<td>&lt;pending&gt;</td>
<td>6379:31218/TCP</td>
<td>2s</td>
</tr>
</tbody>
</table>
```
Exposing Flask Service (LoadBalancer)

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: flask-app
  name: flask-app
spec:
  ports:
    - port: 5000
      protocol: TCP
      targetPort: 5000
  selector:
    run: flask-app
  type: LoadBalancer
```
Design Pattern - Ingress

**Ingress** is the general term for controlling *incoming* traffic

(and *Egress* is the term for *outgoing* traffic)
Design Pattern - Ingress

**Ingress** is the general term for controlling *incoming* traffic

(and *Egress* is the term for *outgoing* traffic)

In the context of Kubernetes it refers to the ability (limited feature set) to control incoming traffic. See [Kubernetes Docs - Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/)
**Design Pattern - Ingress**

**Ingress** is the general term for controlling *incoming* traffic

(and *Egress* is the term for *outgoing* traffic)

In the context of Kubernetes it refers to the ability (limited feature set) to control incoming traffic. See [Kubernetes Docs - Ingress](https://kubernetes.io/docs/concepts/services/ingress/)

A set of **Ingress Rules** is specified to be implemented by a **Kubernetes Controller** which typically implements Load Balancer, Gateway features.

There are many projects providing such controller functionality such as *Nginx, HAproxy, Ambassador, Gloo, Traefik*
Exposing Services (Ingress)

$ minikube addons enable ingress
  ingress was successfully enabled

$ kubectl apply -f misc/ingress-definition.yaml
  ingress.extensions "ingress-definitions" created

$ sudo vi /etc/hosts
  ...
  192.168.99.100  minikube.test  flaskapp.test
Exposing Services (Ingress)

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-definitions
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  backend:
    serviceName: default-http-backend
    servicePort: 80
  rules:
  - host: minikube.test
    http:
      paths:
      - path: /
        backend:
          serviceName: k8sdemo
          servicePort: 8080
  - host: flaskapp.test
    http:
      paths:
      - path: /flask
        backend:
          serviceName: flask-app
          servicePort: 5000
```
**Exposing Services (Ingress)**

```
$ minikube service list

<table>
<thead>
<tr>
<th>NAMESPACE</th>
<th>NAME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>k8s-demo</td>
<td><a href="http://192.168.99.100:31280">http://192.168.99.100:31280</a></td>
</tr>
<tr>
<td>default</td>
<td>redis</td>
<td><a href="http://192.168.99.100:31218">http://192.168.99.100:31218</a></td>
</tr>
</tbody>
</table>

$ curl http://192.168.99.100:31280

$ curl http://minikube.test/k8s-demo
```
## Exposing Services (Ingress)

```bash
$ minikube service list

<table>
<thead>
<tr>
<th>NAMESPACE</th>
<th>NAME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>k8sdemo</td>
<td><a href="http://192.168.99.100:31280">http://192.168.99.100:31280</a></td>
</tr>
<tr>
<td>default</td>
<td>redis</td>
<td><a href="http://192.168.99.100:31218">http://192.168.99.100:31218</a></td>
</tr>
</tbody>
</table>

[flask-app-8577b44db-kbwpn] Redis counter value=214

$ curl http://flaskapp.test/flask
[flask-app-8577b44db-kbwpn] Redis counter value=215
```
Design Pattern - API Gateway

Ref: "What is an API Gateway?"

Classic API Gateways date back to Web Service (SOAP APIs) which offloaded Ingress functions into a single system.

API Gateways are API proxies between the client (API consumer) and server (API Provider).

- API Security
- API Control and governance
- API Monitoring
- API Administration
- API Transformation: See "API Gateway Pattern"
Design Pattern - API Gateway

External entrypoint exposes APIs

- Offloads common Ingress functions => reduces microservice complexity.
  - rate limiting, security, authorisation, DDOS protection
  - Protocol version translation, e.g. REST to SOAP, *-RPC ...
  - TLS decryption/encryption

- Hides internal infrastructure detail => controls access
  - service routing, load-balancing
  - Allows to refactor/scale/mock internal implementation
Design Pattern - API Gateway

External entrypoint exposes APIs

- Offloads common Ingress functions => reduces µ-service complexity.
  - rate limiting, security, authorisation, DDOS protection
  - Protocol version translation, e.g. REST to SOAP, *-RPC ...
  - TLS decryption/encryption

- Hides internal infrastructure detail => controls access
  - service routing, load-balancing
  - Allows to refactor/scale/mock internal implementation

Needs to scale, be H.A.
Design Pattern - API Gateway

There are many API Gateways including

- NGInx, HA-Proxy,
- Newer generation: Envoy-based such as Ambassador, Gloo
Design Pattern - API Gateway

There are many API Gateways including

- NGInx, HA-Proxy,
- Newer generation: Envoy-based such as Ambassador, Gloo

But can API Gateways resist the pressure coming from the next contender ...
Design Pattern - Service Mesh

**Problem:** Micro-services are fine, but we see the need for common functions

- Logging and tracing
- Reliable network communication
- Encryption between components
Design Pattern - Service Mesh

**Problem:** Micro-services are fine, but we see the need for common functions

- Logging and tracing
- Reliable network communication
- Encryption between components

**BUT** if every micro-service reimplements the same functionalities we will get **micro-monoliths**!!
Design Pattern - Service Mesh

**Problem:** Micro-services are fine, but we see the need for common functions

- Logging and tracing
- Reliable network communication
- Encryption between components

**BUT** if every micro-service reimplements the same functionalities we will get **micro-monoliths**!!

- The problem is compounded by the polyglot nature of micro-services, requiring good library support for functions
Design Pattern - Service Mesh

**Problem:** Micro-services are fine, but we see the need for common functions

- Logging and tracing
- Reliable network communication
- Encryption between components

**BUT** if every micro-service reimplements the same functionalities we will get micro-monoliths !!

- The problem is compounded by the polyglot nature of micro-services, requiring good library support for functions

**Service Mesh** helps to address this issue by offloading such functionality

This keeps our micro-services small and simple.

Offload-functionality is provided through **Sidebar** containers - *not libraries.*
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.

Platforms such as Linkerd (v2) and Istio (v1) provide offload for μ--services

Offloads functionality from services in a distributed way.
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.
Platforms such as Linkerd (v2) and Istio (v1) provide offload for μ--services
Offloads functionality from services in a distributed way.
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.
Platforms such as Linkerd (v2) and Istio (v1) provide offload for µ--services
Offloads functionality from services in a distributed way.
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.
Platforms such as Linkerd (v2) and Istio (v1) provide offload for μ-services
Offloads functionality from services in a distributed way.
Hybrid Apps - *API Gateway Pattern*

**Problem:** But wouldn't it be better if we could mix legacy and new paradigms?

The Strangler pattern is an option but requires being able to rebuild the original monolith to extract functionality.

It would be useful to be able to add new functionality in a less invasive way.
Hybrid Apps - *API Gateway Pattern*

There is a "API Gateway" pattern whereby the gateway has the ability to understand the API protocols.

It may also understand the underlying Infrastructure and Platform APIs.

This allows to perform API translation and routing and really take advantage of the orchestration platforms.

"Microservices Patterns" Book
Hybrid Apps - *API Gateway Pattern*

Gloo allows to route between legacy apps, micro-services and serverless incrementally adding new functionality.

Hybrid Apps - *API Gateway Pattern*

Gloo understands the infrastructure on which it is running and the APIs being used.

Gloo is one of several open source projects from Solo.io to facilitate the adoption of modern paradigms such as Micro-services

- Gloo: API Gateway
- Sqoop: Tool for modelling API interactions
- Squash: Micro-service debugging tool
Hybrid Apps - API Gateway Pattern

Gloo understands the infrastructure on which it is running and the APIs being used.

Gloo is one of several open source projects from Solo.io to facilitate the adoption of modern paradigms such as Micro-services

- Gloo: API Gateway
- Sqoop: Tool for modelling API interactions
- Squash: Micro-service debugging tool
So API Gateways or Service Mesh?

Service Mesh and API Gateways provide similar functionality

- Service Mesh control *mainly* E-W traffic between micro-services
- API Gateway control N-S (Ingress) traffic
So API Gateways or Service Mesh?

Service Mesh and API Gateways provide similar functionality

- Service Mesh control *mainly* E-W traffic between micro-services
- API Gateway control N-S (Ingress) traffic

Service Mesh technology is quickly advancing

- May be overkill for some use cases
- Istio now includes basic Gateway (N-S) functionality
- Service Mesh Vendors say we still need API Gateways for the moment.
- Linkerd just received new VC funding
So API Gateways or Service Mesh?

Service Mesh and API Gateways provide similar functionality

- Service Mesh control *mainly* E-W traffic between micro-services
- API Gateway control N-S (Ingress) traffic

Service Mesh technology is quickly advancing

- May be overkill for some use cases
- Istio now includes basic Gateway (N-S) functionality
- Service Mesh Vendors say we still need API Gateways for the moment.
- Linkerd just received new VC funding

But, API Gateways will continue to offer advanced functionality for Ingress control.
So API Gateways or Service Mesh?

Service Mesh and API Gateways provide similar functionality

- Service Mesh control *mainly* E-W traffic between micro-services
- API Gateway control N-S (Ingress) traffic

Service Mesh technology is quickly advancing

- May be overkill for some use cases
- Istio now includes basic Gateway (N-S) functionality
- Service Mesh Vendors say we still need API Gateways for the moment.
- Linkerd just received new VC funding

But, API Gateways will continue to offer advanced functionality for Ingress control.

Going forward we can expect to see Service Mesh incorporating more and more Gateway functionality
Outline

- [Why?] Monoliths to Micro-services
- Orchestration: Kubernetes
- Deployment Strategies
- Architecture Design patterns
- Summary
Summary

Micro-services offer new deployment possibilities

- with ease of deployment, scaling, upgrading
- facilitate "Best in Class" technology choices/replacements
Summary

Micro-services offer new deployment possibilities
- with ease of deployment, scaling, upgrading
- facilitate "Best in Class" technology choices/replacements

**BUT** moving to µ-services requires
- organizational changes and best practices!
- incremental rollout - small steps / Strangler
- hybrid approaches - old/new, cloud/on-premise, VM/container/µ-service
- offload via API Gateway and/or Service Mesh
Thank you!

From Monologue to Discussions ... ?

Questions?

Michael Bright, @mjbright

Cloud Native Training (Docker, Kubernetes, Serverless)

Slides & source code at https://mjbright.github.io/Talks
Summary

Getting started with Kubernetes

- Start by learning Docker principles
- Experiment by Dockerizing some applications
- Learn about Container Orchestration
- Hands-on with Kubernetes online or Minikube(*)
- Kubernetes Visualization with KubeView
- https://github.com/mjbright/kubeview
Resources

**minikube**

- Download: [https://github.com/kubernetes/minikube/releases](https://github.com/kubernetes/minikube/releases)
- Documentation: [https://kubernetes.io/docs/getting-started-guides/minikube/](https://kubernetes.io/docs/getting-started-guides/minikube/)
- Hello Minikube: [https://kubernetes.io/docs/tutorials/stateless-application/hello-minikube/](https://kubernetes.io/docs/tutorials/stateless-application/hello-minikube/)

Slides & source code at [https://mjbright.github.io/Talks](https://mjbright.github.io/Talks)
Resources - Articles

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

MuleSoft, "The top 6 Microservices Patterns"  
https://www.mulesoft.com/lp/whitepaper/api/top-microservices-patterns

FullStack Python  
https://www.fullstackpython.com/microservices.html

Idit Levine  

SSola  
https://medium.com/@ssola/building-microservices-with-python-part-i-5240a8dcc2fb

Deployment  
http://container-solutions.com/kubernetes-deployment-strategies/

Slides & source code at https://mjbright.github.io/Talks
## Resources - Books

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Title, Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>O'Reilly</td>
<td>&quot;Building Microservices&quot;, Sam Newman, July 2015</td>
</tr>
<tr>
<td>PacktPub</td>
<td>&quot;Python Microservices Development&quot;, Tarek Ziade, July 2017</td>
</tr>
<tr>
<td>kNative - O'Reilly</td>
<td></td>
</tr>
<tr>
<td>Istio - Manning</td>
<td></td>
</tr>
<tr>
<td>Istio - O'Reilly</td>
<td></td>
</tr>
<tr>
<td>Testdriven.io</td>
<td></td>
</tr>
</tbody>
</table>

Slides & source code at [https://mjbright.github.io/Talks](https://mjbright.github.io/Talks)