Developing Micro-services with Kubernetes

24 April 2018, CodeEurope.pl

Michael Bright, @mjbright

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

Cloud Native Solution Architect

Trainer: Kubernetes, Serverless, Docker, CloudNative

Past researcher, dev, team lead, dev advocate

British, living in France for 25-years

Docker Community Lead, Python User Group

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

• Monoliths to Micro-services
• Micro-service design patterns
• Kubernetes
• Operations
• Demo
• Tools
Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
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
First ... a bit of history

Note: But the future is hybrid ...
Monoliths to Micro-services

Monoliths are **deployed, scaled, upgraded, reimplemented** as complete units.

Individual µ-service components can be **deployed, scaled, upgraded, reimplemented** ...
Advantages of Micro-services

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

Separation of Concerns - "do one thing well"

Smaller Projects/teams
Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

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

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components
Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components

Allow for composition of new services
Advantages of Micro-services

Separation of Concerns - "do one thing well"

Smaller Projects/teams

Ease Scaling, Deployment, Testing, Evolution

Loosely coupled components

Allow for composition of new services

So are they a panacea?
Disadvantages

Greater complexity

- Requires more orchestration
- Greater organizational complexity
- Monitoring, debugging is more difficult
Disadvantages

Greater complexity

- Requires more orchestration
- Greater organizational complexity
- Monitoring, debugging is more difficult

More network communication

- Network error handling
- Performance
Disadvantages

Greater complexity

- Requires more orchestration
- Greater organizational complexity
- Monitoring, debugging is more difficult

More network communication

- Network error handling
- Performance

Still requires best practices

- Behaviour and Test-Driven Development
- CI/CD
- Documentation of interfaces/APIs
- Stable interfaces/APIs
Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
Architecture Design Patterns

Standard Component Patterns
Architecture Design Patterns

Standard Component Patterns

Fine-grained SOA - Micro-services(!)
Architecture Design Patterns

Standard Component Patterns

Fine-grained SOA - Micro-services(!)

Strangler
Architecture Design Patterns

Standard Component Patterns

Fine-grained SOA - Micro-services(!)

Strangler

API Gateway
Architecture Design Patterns

Standard Component Patterns

Fine-grained SOA - Micro-services(!)

Strangler

API Gateway

Service Mesh
Architecture Design Patterns

Standard Component Patterns

Fine-grained SOA - Micro-services(!)

Strangler

API Gateway

Service Mesh

Hybrid Apps
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces μ-service complexity
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces μ-service complexity

- Hides internal infrastructure detail
  - service routing, load-balancing
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces µ-service complexity

- Hides internal infrastructure detail
  - service routing, load-balancing

- Prevents general access to internal infrastructure
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces µ-service complexity

- Hides internal infrastructure detail
  - service routing, load-balancing

- Prevents general access to internal infrastructure

- Allows to refactor/scale/mock internal implementation
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces µ-service complexity

- Hides internal infrastructure detail
  - service routing, load-balancing

- Prevents general access to internal infrastructure

- Allows to refactor/scale/mock internal implementation

- Protocol version translation, e.g. REST/https to REST or SOAP/http, *-RPC ...
Design Pattern - API Gateway

Exposes internal APIs via single external entry point.

- Offload common functions
  - rate limiting, security, authorisation
  - protection against DDoS
  - reduces μ-service complexity

- Hides internal infrastructure detail
  - service routing, load-balancing

- Prevents general access to internal infrastructure

- Allows to refactor/scale/mock internal implementation

- Protocol version translation, e.g. REST/https to REST or SOAP/http, *-RPC ...

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

https://api.com/....

API Gateway

Service 1 (REST/http)
Service 2 (JSON-RPC)
Service 3 (WAMP)
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.
Design Pattern - Service Mesh

Abstraction above TCP/IP, secure reliable inter-service connectivity.
Offloads functionality from services in a distributed way.
Design Pattern - Service Mesh

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

Offloads functionality from services in a distributed way.
Design Pattern - Hybrid Apps

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

Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
We need Orchestration
Kubernetes - Architecture

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

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

Minikube single-node "tainted"

Ingress

Load Balancer

Flask1

Flask2

Flask3

Redis
Kubernetes - Deploying Redis

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

```
deployment
```
Kubernetes - Deploying Redis

```
# kubectl run redis --image=redis:latest --port=6379

$ kubectl apply -f redis-deployment.yaml
deployment.extensions "redis" created

$ kubectl get pods
NAME             READY   STATUS              RESTARTS   AGE
redis-68595c4d95-rr4pr  0/1       ContainerCreating   0          1s
```
Kubernetes - Deploying Redis (yaml)

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: redis
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      run: redis
  template:
    metadata:
      labels:
        run: redis
    spec:
      containers:
        - image: redis:latest
          name: redis
          ports:
            - containerPort: 6379
```
Kubernetes - Deploying Flask

cubectl create -f flask-deployment.yaml

deployment

ReplicaSet

Pod1
2e76: flask:v1

Pod2
1f3d: flask:v1
# 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)

```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
```
Kubernetes - Exposing Services

Master

Service

Worker
- pod
- pod

Worker
- pod
- pod

User

IP:port

@mjbright
Exposing Services (LoadBalancer)

Master

Service

Worker
pod
pod

Worker
pod
pod

User

External Load Balancer

IP:port
Exposing Services (NodePort)

User connects to IP/port of one of the Nodes

- Master
- Service
- User
- Worker
  - pod
  - pod
Exposing Services (IngressController)

- Master
- Service
- Worker
- User
- pod
- pod
- pod
- pod

Ingress (IngressController)
Exposing Redis Service (LoadBalancer)

# 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)

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

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

$k 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)

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
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
class: 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>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>

$ curl http://192.168.99.100:31280

$ curl http://minikube.test/k8sdemo
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>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
Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
Operations

- H.A.
- Scaling
- Rolling Upgrade
- Strategies
- Health Checks
Operations - achieving High Availability

Achieved through running multiple instances across multiple nodes of the data center

- resilience to node outages
- resilience to pod outages or poor response times
Operations - Scaling

```bash
# kubectl scale deploy flask-app --replicas=4

$ kubectl edit -f flask-deploy.yaml

...
spec:
  replicas: 4
```
Operations - Rolling Upgrades

Several strategies exist

recreate - terminate old version before releasing new one
Operations - Rolling Upgrades

Several strategies exist

- **recreate** - terminate old version before releasing new one
- **ramped** - gradually release a new version on a rolling update fashion
Operations - Rolling Upgrades

Several strategies exist

**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 - Rolling Upgrades

Several strategies exist

**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 - Rolling Upgrades

Several strategies exist

- **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 - Rolling Upgrade

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
Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
Demo

Master Node
"Worker"

Minikube single-node "tainted"

Load Balancer

Ingress

Flask1

Flask2

Flask3

Redis
Outline

- Monoliths to Micro-services
- Micro-service design patterns
- Kubernetes
- Operations
- Demo
- Tools
Tools

- Tools
  - Helm (use to install tools)
  - Prometheus
  - Squash
  - Gloo
  - Istio / Service Meshes / Envoy
Summary

Getting started with Micro-services
Summary

Getting started with Micro-services

If migrating monolith, take small steps
Summary

Getting started with Micro-services

If migrating monolith, take small steps

Secure your services behind firewall/API gw
Summary

Getting started with Micro-services

- If migrating monolith, take small steps
- Secure your services behind firewall/API gw
- Services must use public APIs only
Summary

Getting started with Micro-services

- If migrating monolith, take small steps
- Secure your services behind firewall/API gw
- Services must use public APIs only
- Choose "best" technology for each component
Summary

Getting started with Micro-services

- If migrating monolith, take small steps
- Secure your services behind firewall/API gw
- Services must use public APIs only
- Choose "best" technology for each component
- Transform technology and your organization
Summary

Getting started with Micro-services

- If migrating monolith, take small steps
- Secure your services behind firewall/API gw
- Services must use public APIs only
- Choose "best" technology for each component
- Transform technology and your organization
- Automate, automate, automate ...
Summary

Getting started with Kubernetes
Summary

Getting started with Kubernetes

Start by learning Docker principles
Summary

Getting started with Kubernetes

Start by learning Docker principles

Experiment by Dockerizing some applications
Summary

Getting started with Kubernetes

- Start by learning Docker principles
- Experiment by Dockerizing some applications
- Learn about Container Orchestration
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(*)
Summary

Micro-services offer new deployment possibilities
Summary

Micro-services offer new deployment possibilities

- ease of deployment
Summary

Micro-services offer new deployment possibilities

- ease of deployment
- ease of scaling
Summary

Micro-services offer new deployment possibilities

- ease of deployment
- ease of scaling
- ease of upgrades
Summary

Micro-services offer new deployment possibilities

- ease of deployment
- ease of scaling
- ease of upgrades
- "Best in Class" polyglot implementation
Summary

**Micro-services offer new deployment possibilities**

- ease of deployment
- ease of scaling
- ease of upgrades
- "Best in Class" polyglot implementation

**Hybrid approaches will be adopted**
Summary

**Micro-services offer new deployment possibilities**

- ease of deployment
- ease of scaling
- ease of upgrades
- "Best in Class" polyglot implementation

**Hybrid approaches will be adopted**

- combining container-based micro-services, VMs, Serverless ...

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

Questions?

Michael Bright, 🐦@mjbright

linkedin.com/in/mjbright  🐱github.com/mjbright

Training classes available

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

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

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

- Martin Fowler
  - https://martinfowler.com/articles/microservices.html
- MuleSoft, "The top 6 Microservices Patterns"
- FullStack Python
  - https://www.fullstackpython.com/microservices.html
- Idit Levine
  - https://medium.com/@ssola/building-microservices-with-python-part-i-5240a8dcc2fb
- SSola

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>
</tbody>
</table>

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