Services for All

How to Empower Engineers (with Kubernetes)
1. A brief history of configuration at Airbnb
2. Generating Kubernetes files
3. Creating a kubectl wrapper
4. Extending and customizing Kubernetes
5. Beyond Kubernetes
What is Airbnb?

AN ONLINE MARKETPLACE FOR SHARING HOMES AND EXPERIENCES

81k cities
191+ countries
5 mil homes
Who am I?
A BRIEF HISTORY
Why Microservices?

MONOLITH LOC
Why Microservices?
Why Microservices?

SCALING CONTINUOUS DELIVERY
Why Microservices?

Deploys per week (all apps, all environments)
Why Microservices?

Monolith production deploys per week

COUNT(*)
120,000 production deploys per year
Splitting the Monolith

Monolith

UI
Load Balancer
Server Side Rendering
API Gateway

Data services

Global services
Splitting the Monolith

UI
Load Balancer
Server Side Rendering
API Gateway

Data services
Global services

@MELANIECEBULA
Standardize Service Creation

NO SNOWFLAKES
OKAY BUT HOW?
CONFIGURATION IS KEY
Evolution of Configuration Management

Manually configuring boxes

Automating configuration of applications with Chef

Automating configuration and orchestration of containerized applications with Kubernetes
KUBERNETES OUT-OF-THE-BOX
What Makes Kubernetes Awesome

- immutable, reproducible containers
- designed for a microservices architecture
- human-readable format
- declarative
- efficient scheduling
- extensible API
What Makes Kubernetes NOT Awesome

- significant set up cost
- complex configuration and concepts
- tooling is not developer friendly
- open issues
GENERATING KUBERNETES FILES
kubectl apply
lots of boilerplate repetitive by environment

kubernetes config files

Production Deployment
Canary Deployment
Production ConfigMap
Canary ConfigMap
Production Service
Canary Service

kubectl apply

kubernetes cluster
kube-gen!

kube-gen generate

Production Deployment -> Canary Deployment
Production ConfigMap -> Canary ConfigMap
Production Service -> Canary Service

kubectl apply

kubernetes cluster
What are the basic attributes of my project?
# under _infra/kube/kube-gen.yml

version: 5.1.0

project:
  name: bonk

environments:
  production:
    params:
      replicas: 10
      port: 6585
  staging:
    params:
      replicas: 2
      port: 6586

users:
  melanie_cebula
# under _infra/kube/kube-gen.yml

version: 5.1.0

project:
  name: bonk

environments:
  production:
    params:
      replicas: 10
      port: 6585
  staging:
    params:
      replicas: 2
      port: 6586

users:
  melanie_cebula

becomes an admin role binding file!
# under generated/bonk/bonk-production/admin-role-binding.yml

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: null
  name: bonk-production-admin-role-binding
  namespace: bonk-production
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: User
  name: melanie_cebula
# under _infra/kube/kube-gen.yml

version: 5.1.0

project:
  name: bonk

environments:
  production:
    params:
      replicas: 10
      port: 6585

  staging:
    params:
      replicas: 2
      port: 6586

users:
  melanie_cebula

environments can be accessed by all the other files!
- different concepts map to different yaml files under `_infra/kube`
- environment params defined in the project are accessed with go templating:
  - ex: `{.Env.Params.replicas }`
kube-gen will generate all the files for the different environments, and replace each `.Env.Params.replicas` with the appropriate value from the project file.

ex: staging has 2 replicas, production has 10 replicas

```yaml
# under _infra/kube/apps/bonk.yml
workload:
  deployment:
    replicas: {{ .Env.Params.replicas }}
    strategy:
      rollingUpdate
```

{{ }} signals go templating

from the project file!
What kind of workload?

example workloads

- bonk-web
- bonk-worker
- bonk-cron
# under _infra/kube/apps/bonk-web.yml

workload:

  deployment:

  {{ if eq .Env.Name "development" }}

    strategy:

      rollingUpdate:

        maxUnavailable: 1

  {{ else }}

    autoscaling:

      minReplicas: {{ .Env.Params.minReplicas }}

      maxReplicas: {{ .Env.Params.maxReplicas }}

  {{ end }}
Which shared components to use?

Example components:
- nginx
- secrets
- statsd
nginx component

- common patterns are abstracted into a component
- apps enable components
- component yaml merged into project
- may require params to be set
- may set default params
What does the container need?
- image to use
- command and args to run
- ENV variables to pass in
- resource requests and limits for container
- file and volume mounts used by this container
These can be mounted into your containers
The main Dockerfile for the project (installs needed dependencies)
KUBECKL WRAPPER
kubernetes config files

- Production Deployment
- Canary Deployment
- Production ConfigMap
- Canary ConfigMap
- Production Service
- Canary Service

kubectl apply

kubernetes cluster
kubectl IS VERbose

kubernetes config files

- Production Deployment
- Canary Deployment
- Production ConfigMap
- Canary ConfigMap
- Production Service
- Canary Service

kubectl apply

kubernetes cluster

verbose repetitive by namespace
k tool
THE ALL PURPOSE CLI WRAPPER
• Runs in the project home directory:
  $ cd /path/to/bonk
  $ k status

• Environment variables for arguments:
  $ k status ENV=staging
  $ export ENV=staging
  $ k status

• Prints the command that it will execute:
  $ k status ENV=staging
  kubectl get pods --namespace=bonk-staging
k tool
SIMPLIFIES BUILD AND DEPLOYS

- k generate transforms kube-gen files to kubernetes files
- k build performs docker build and docker push with tags
- k deploy creates namespace, applies/replaces kubernetes files, sleeps and checks deployment status
- can chain commands; ex: k all
• defaults to random pod, main container:
  $ k ssh ENV=staging

• specify particular pod, specific container:
  $ k logs ENV=staging POD=... CONTAINER=statsd-proxy

• k diagnose: shows status and logs for each failing container of your pod, and shows failure events
Extending Kubernetes with Custom Controllers

• **admission controller:**
  - require project ownership (via annotations)
  - prevent namespace creation under incorrect cluster
  - prevent deployment of objects with known security holes

• **deployment pruner:**
  - automatically rotate old pods
  - rotates over-provisioned availability zones
- it is a common anti-pattern for clients of a service to not retry on connection failures
- as a k8s service deploys, requests may be sent to terminating pods and cause a spike in 500s
- use graceful termination to mitigate this
Extending Kubernetes with Graceful Termination

# under _infra/kube/apps/bonk.yml
# wait up to 180 seconds before “kill -9” the pods
terminationGracePeriodSeconds: 180

# under _infra/kube/containers/container.yml
# wait up to 120 seconds before shutting down
lifecycle:
  preStop:
    exec:
      command:
      - /bin/sleep
      - “120”
Extending Kubernetes WITH GRACEFUL TERMINATION

# under _infra/kube/apps/bonk.yml
# wait up to 180 seconds before “kill -9” the pods
terminationGracePeriodSeconds: 180

# under _infra/kube/containers/container.yml
# wait up to 120 seconds before shutting down
lifecycle:
  preStop:
    exec:
      command:
      - /bin/sleep
      - "120"

this gives our service discovery container time to mark this pod as unhealthy.
Extending Kubernetes with Graceful Termination

# under _infra/kube/apps/bonk.yml
# wait up to 180 seconds before “kill -9” the pods
terminationGracePeriodSeconds: 180

# under _infra/kube/containers/container.yml
# wait up to 120 seconds before shutting down
lifecycle:
  preStop:
    exec:
      command:
      - /bin/sleep
      - “120”

you can do more sophisticated server shutdown here too!
Extending Kubernetes with Graceful Termination

# under _infra/kube/apps/bonk.yml
# wait up to 180 seconds before “kill
terminationGracePeriodSeconds: 180

# under _infra/kube/containers/container.yml
# wait up to 120 seconds before shutting down
lifecycle:
  preStop:
    exec:
      command:
      - /bin/sleep
      - “120”

after the preStop command finishes, the containers have an additional 60sec to exit gracefully
BEYOND KUBERNETES
Creating a new service (before)
Everything about a service should be in one place, and managed with one process.
Everything you need to know about your service can be found in one place

- All configuration lives in _infra alongside project code
- Edit code and configuration with one pull request
- Easy to add new configuration
- Configuration statically validated as part of CI
What we support:

- kube-gen files
- continuous integration files (with containers)
- documentation (markdown)
- databag secrets and keys
- development (legacy)
- AWS IAM roles
- internal tool configuration (UI/UX)
- legacy service discovery configuration
- project ownership and metadata
- .. and more!
A single deploy process for every change

Develop
Write code and config under your project

Merge
Open a PR and merge your code to master

Deploy
Deploy all changes atomically
GENERATING SERVICE
BOILERPLATE
Reduce service boilerplate
WITH GENERATORS!

• generators make best practices the default
• collection of config generators and language-specific generators
• uses thor for better control (update, review, commit)
• set up a "hello world" service with just one command
Reduce time to “hello world”

2 weeks → 2 minutes
Takeaways

- Store configuration alongside project code
- Abstract away your infrastructure with generators
- Abstract away complex tooling with a wrapper CLI
- Configuration and tools should set defaults
- Standardize on one process for storing and applying configuration changes