HCS COMPANY

KUSTOMIZE

Hello...

```
$ oc adm new-project HCS-Company \
  --admin="Vincent van Dam"      \
  --display-name="joyrex2001"    \
  --admin-role="Open Source Architect"
```

Delivering software

- Developed a service
- Think of the invariants for each deployment target (dev/acc/prd)
- Use some templating system (openshift templates, helm)
- Populate these variables in a delivery pipeline... and deploy...

TEMPLATING

```
---
kind: Template
apiVersion: v1
metadata:
parameters:
- name: NAME
  displayName: Name
  description: The name assigned to all of the frontend objects defined
  required: true
  value: game
- name: MEMORY_LIMIT
  displayName: Memory Limit
  description: Maximum amount of memory the frontend container can use.
  required: true
  value: 256Mi
- name: MEMORY_LIMIT_REDIS
  displayName: Memory Limit
  description: Maximum amount of memory the redis container can use.
  required: true
  value: 128Mi
- kind: DeploymentConfig
  apiVersion: v1
  metadata:
    name: "${NAME}"
    annotations:
livenessProbe:
  timeoutSeconds: 3
  initialDelaySeconds: 30
  httpGet:
    path: "/healthz"
    port: 8080
resources:
  limits:
    memory: "${MEMORY_LIMIT}"
```

POPULATE IN A PIPELINE

```
stage(”Apply template in dev project") {
  steps {
    script {
      openshift.withCluster() {
        openshift.withProject("${myproject}") {
          openshift.apply(openshift.process("-f", template,
             "-p NAME=${appname}",
             "-p MEMORY_LIMIT=${mem_limit}",
             "-p MEMORY_LIMIT_REDIS=${mem_limit_redis}” ))
        }
      }
    }
  }
}
```

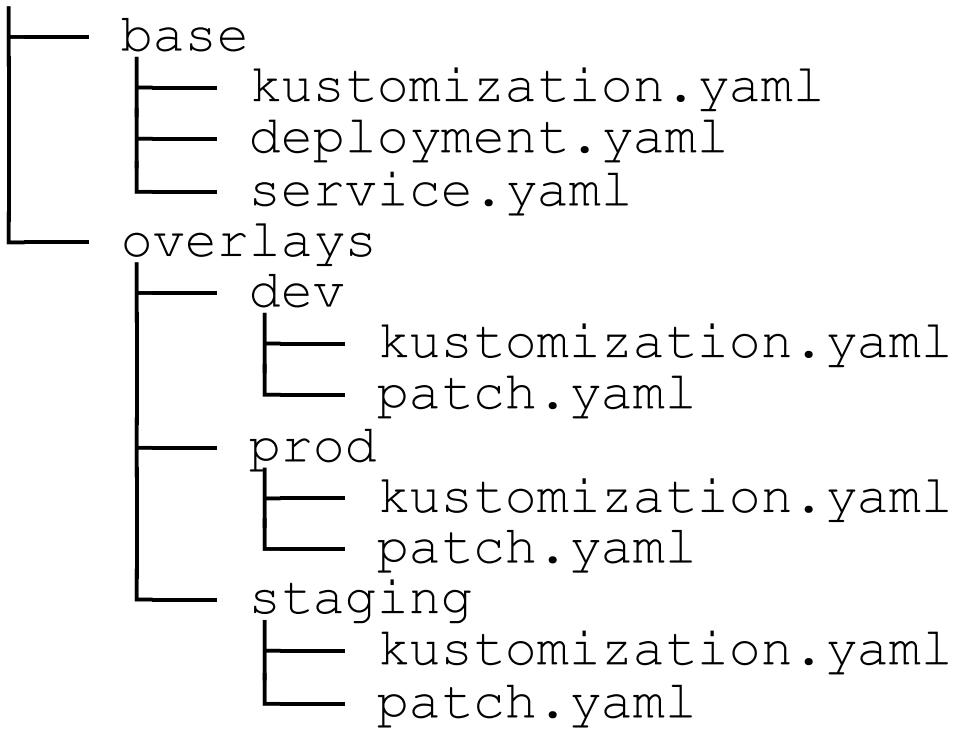
Many moons later…

- The parameters are populated at various places for the deployments…
- new insights, and a new parameter is required, for prod only…
- Changes everywhere…

Then comes kustomize

- Different approach
- Create the deployment as regular resource definitions
- And patch it…
- …integrated in kubectl, but also available stand-alone

THEN COMES KUSTOMIZE



Default – base configuration

Overlays with patches for each specific environment

THEN COMES KUSTOMIZE

```
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   └── service.yaml
```

kustomization.yaml

```
commonLabels:
  app: nginx

resources:
- deployment.yaml
- service.yaml
```

THEN COMES KUSTOMIZE

```
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   └── service.yaml
```

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx

      <<< CUT FOR SIMPLICITY >>>
```

THEN COMES KUSTOMIZE

```
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   └── service.yaml
```

service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    deployment: nginx
  type: LoadBalancer
  ports:

      <<< CUT FOR SIMPLICITY >>>
```

THEN COMES KUSTOMIZE

```
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   └── service.yaml
└── overlays
    ├── dev
    │   ├── kustomization.yaml
    │   └── patch.yaml
```

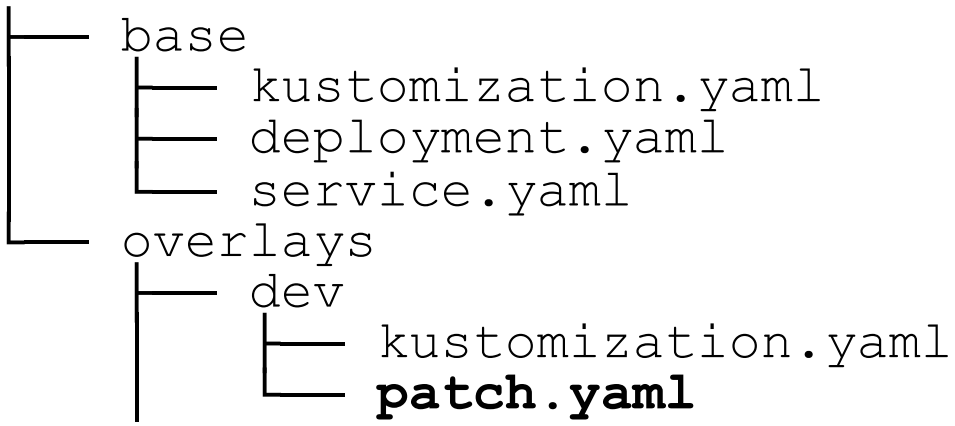
kustomization.yaml

```
namespace: myservice-dev

resources:
- ../../base

patches:
- patch.yaml
```

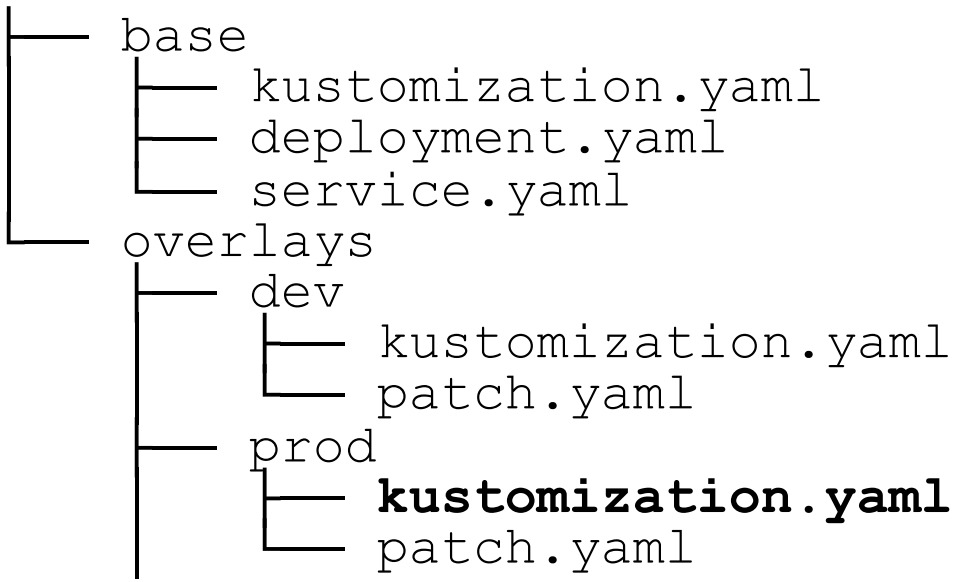
THEN COMES KUSTOMIZE



patch.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
```

THEN COMES KUSTOMIZE



kustomization.yaml

```
namespace: myservice-prod

resources:
- ../../base

patches:
- patch.yaml
```

THEN COMES KUSTOMIZE

```
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   └── service.yaml
└── overlays
    ├── dev
    │   ├── kustomization.yaml
    │   └── patch.yaml
    └── prod
        ├── kustomization.yaml
        └── patch.yaml
```

patch.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    spec:
      containers:
      - name: fluentd
        image: fluentd:latest
```

THEN COMES KUSTOMIZE

service1

```
deploy
├── kustomization.yaml
├── deployment.yaml
└── service.yaml
```

service2

```
deploy
├── kustomization.yaml
├── deployment.yaml
└── service.yaml
```

kustomization.yaml

```
namespace: staging
resources:
- git::ssh://git@mygit.local/service1//deploy
- git::ssh://git@mygit.local/service2//deploy
```

Transforming and generating

- Transformers – update, change existing resources (patching)
- Generators – create resources

GENERATORS

- EXAMPLES:
  - CONFIGMAPGENERATOR
  - SECRETGENERATOR

mysecret.yaml

```
secretGenerator:
- name: app-tls
  files:
    - secret/tls.cert
    - secret/tls.key
  type: "kubernetes.io/tls"
```

Plug ins

- Custom transformers or generators, for example:
  - Creating secrets with custom encryption
  - Custom validators (e.g. test if default values overwritten)
  - Rewriting configurations

Plug ins

- Can be implemented as:
  - Native go plugin **NOT A GOOD IDEA**
  - Exec plugin

Exec plug ins

- Install in well know place (~/.config/kustomize/plugin/hcs-company.com/example)
- Define config (get this as argv[1] in the plugin)
- Get processed resources in yaml via stdin (transformer)
- Output result to stdout

Exec plug ins

- Custom config

myplugin.yaml

```
apiVersion: hcs-company.com/v1
kind: Example

mysecret:
  - key: username
    value: WB4HBKtOyfQx4+Ds15======
  - key: password
    value: WB4HBKtOyfQx4+Ds15======
```

Examples (python): https://github.com/Agilicus/kustomize-plugins

Why kustomize?

- Use regular Kubernetes resource manifest
- No need for planning up-front what settings to 'template'
- Ability to write custom plugins to tackle specific use cases
- Part of kubectl

FIN!