Spring on Kubernetes!

Workshop Materials: https://hackmd.io/@ryanjbaxter/spring-on-k8s-workshop
(https://hackmd.io/@ryanjbaxter/spring-on-k8s-workshop)
Ryan Baxter, Spring Cloud Engineer, VMware
Dave Syer, Spring Engineer, VMware

What You Will Do

- Create a basic Spring Boot app
- Build a Docker image for the app
- Push the app to a Docker repo
- Create deployment and service descriptors for Kubernetes
- Deploy and run the app on Kubernetes
- External configuration and service discovery
- Deploy the Spring PetClinic App with MySQL

Prerequisites

- Basic knowledge of Spring and Kubernetes (we will not be giving an introduction to either)
- JDK 8 or higher (https://openjdk.java.net/install/index.html)
  - **Please ensure you have a JDK installed and not just a JRE**
- Docker (https://docs.docker.com/install/) installed
- Kubectl (https://kubernetes.io/docs/tasks/tools/install-kubectl/) installed
- Kustomize (https://github.com/kubernetes-sigs/kustomize/blob/master/docs/INSTALL.md) installed
- Skaffold (https://skaffold.dev/docs/install/) installed
- An IDE (IntelliJ, Eclipse, VSCode)
  - **Optional** Cloud Code (https://cloud.google.com/code/) for IntelliJ and VSCode is nice. For VSCode the Microsoft Kubernetes Extension (https://github.com/Azure/vscode-kubernetes-tools) is almost the same.

Configure kubectl

- Depending on how you are doing this workshop will determine how you configure `kubectl`

Doing The Workshop On Your Own

- If you are doing this workshop on your own you will need to have your own Kubernetes cluster and Docker repo that the cluster can access
  - **Docker Desktop and Docker Hub** - Docker Desktop allows you to easily setup a local Kubernetes cluster (Mac (https://docs.docker.com/docker-for-mac/#kubernetes), Windows (https://docs.docker.com/docker-for-windows/#kubernetes)). This in combination with Docker Hub (https://hub.docker.com/) should allow you to easily run through this workshop.
  - **Hosted Kubernetes Clusters and Repos** - Various cloud providers such as Google and Amazon offer options for running Kubernetes clusters and repos in the cloud. You will need to follow instructions from the cloud provider to provision the cluster and repo as well configuring `kubectl` to work with these clusters.

Doing The Workshop In Person

- Login To https://gangway.workshop.demo.ryanjbaxter.com/ (https://gangway.workshop.demo.ryanjbaxter.com/)
  - Use provided username and password
- Configuring `kubectl`
  - If you are on a unix based OS, you can copy the commands from the browser window and execute them in your terminal window to configure `kubectl`
  - If you are on Windows and using PowerShell copy and pasting the commands does not work well. You can copy the commands into a text file and name is `configkubectl.sh` and run `./configkubectl.sh` in PowerShell to configure everything
- Run the below command to verify kubectl is configured correctly

```
$kubectl cluster-info
Kubernetes master is running at https://k8s.workshop.demo.ryanjbaxter.com:8443
CoreDNS is running at https://k8s.workshop.demo.ryanjbaxter.com:8443/api/v1/namesp

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Create A Spring Boot App

- Click here (https://start.spring.io/starter.zip?type=maven-project&language=java&bootVersion=2.3.0.M4&packaging=jar&jvmVersion=1.8&groupId=com.example&artifactId=k8s-demo-app&name=k8s-demo-

app&description=Demo%20project%20for%20Spring%20Boot%20and%20Kubernetes&packageName=com.example.k8s-demo-app&dependencies=web,actuator) to download a zip of the Spring Boot app

- Unzip the project to your desired workspace and open in your favorie IDE

Add A RestController

Modify `K8sDemoApplication.java` and add a `@RestController`

> Be sure to add the `@RestController` annotation and not just the `@GetMapping`

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class K8sDemoAppApplication {

	public static void main(String[] args) {
		SpringApplication.run(K8sDemoAppApplication.class, args);
	}

	@GetMapping("/")
	public String hello() {
		return "Hello World";
	}
}
```

Run The App

In a terminal window run

```
$ ./mvnw clean package && java -jar ./target/*.jar
```

The app will start on port `8080`

Test The App

Make an HTTP request to http://localhost:8080 (http://localhost:8080)

```
$ curl http://localhost:8080; echo
Hello World
```

Test Spring Boot Actuator

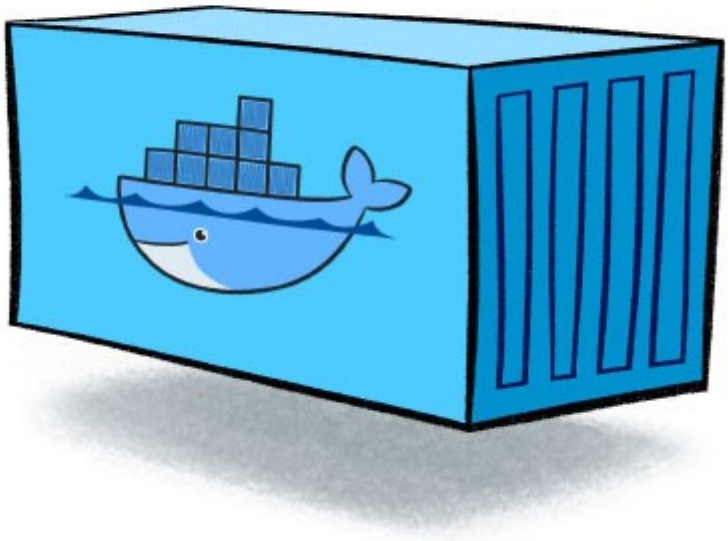
Spring Boot Actuator adds several other endpoints to our app as well

```
$ curl localhost:8080/actuator; echo
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
}
```

> Be sure to stop the Java process before continuing on or else you might get port binding issues since Java is using port `8080`

Containerize The App

The first step in running the app on Kubernetes is producing a conainer for the app we can then deploy to Kubernetes



Building A Container

- Spring Boot 2.3.x can build a container for you without the need for any additional plugins or files
- To do this use the Spring Boot Build plugin goal `build-image`

```
$ ./mvnw spring-boot:build-image
```

- Running `docker images` will allow you to see the built container

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREA
k8s-demo-app                        0.0.1-SNAPSHOT      ab449be57b9d        5 mi
```

Run The Container

```
$ docker run --name k8s-demo-app -p 8080:8080 k8s-demo-app:0.0.1-SNAPSHOT
```

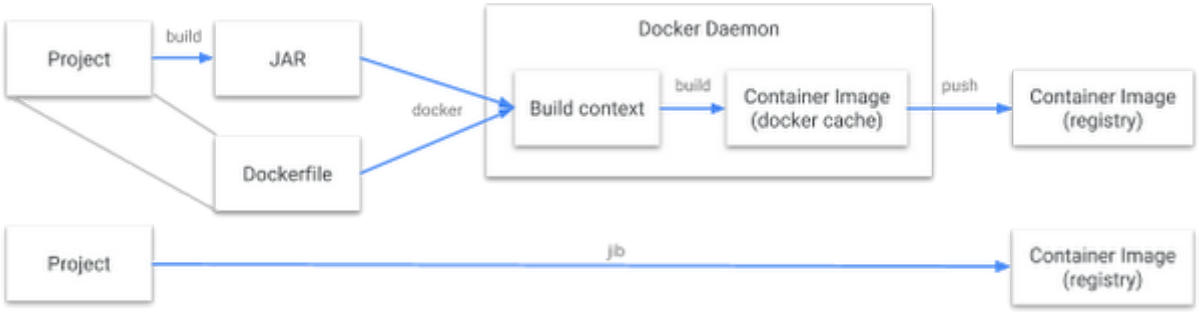
Test The App Responds

```
$ curl http://localhost:8080; echo
Hello World
```

> Be sure to stop the docker container before continuing. You can stop the container and remove it by running `$ docker rm -f k8s-demo-app`

Jib It!

- Jib (https://github.com/GoogleContainerTools/jib) is a tool from by Google to make building Docker containers easier
- Runs as a Maven or Gradle plugin
- Does not require a Docker deamon
- Implements many Docker best practices automatically



Add Jib To Your POM

- Open the POM file of your app and add the following `plugin` to the `build` section

```
...
<build>
   <plugins>
      ...
      <plugin>
          <groupId>com.google.cloud.tools</groupId>
          <artifactId>jib-maven-plugin</artifactId>
          <version>1.8.0</version>
          <configuration>
             <from>
                <image>openjdk:8u222-slim</image>
             </from>
             <to>
                <image>k8s-demo-app</image>
             </to>
             <container>
                <user>nobody:nogroup</user>
             </container>
          </configuration>
       </plugin>
       ...
   </plugins>
</build>
...
```

Using Jib To Build A Container

- Now building a container is as simple as building our app

```
$ ./mvnw clean compile jib:dockerBuild
```

- NOTE: This still uses your local Docker dameon to build the container

Putting The Container In A Registry

- Up until this point the container only lives on your machine
- It is useful to instead place the container in a registry
  - Allows others to use the container
- Docker Hub (https://hub.docker.com/) is a popular public registry
- Private registries exist as well

Using Jib To Push To A Registry

- By default Jib will try and push the image to Docker Hub (https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#using-docker-hub-registry)
- We will push our container to a registry associated with our Kubernetes cluster
- Login to https://harbor.workshop.demo.ryanjbaxter.com/ (https://harbor.workshop.demo.ryanjbaxter.com/) with your supplied username and password
- Once logged in you will see two projects, one called `library` and the other with the same name as your user, we will use the one with the same name as your user

Authenticate With The Registry

- Before Jib can push the container image to the registry we must authenticate

```
$ docker login harbor.workshop.demo.ryanjbaxter.com
```

- Enter your username and password when prompted

Modify Jib Plugin Configuration

- Back in your POM file, modify the Jib plugin, specifically the image name
  - We also add an execution goal in order to do the container build and push whenever we run our Maven build
- **Be sure to replace `USERNAME` in the image name with your supplied user name**

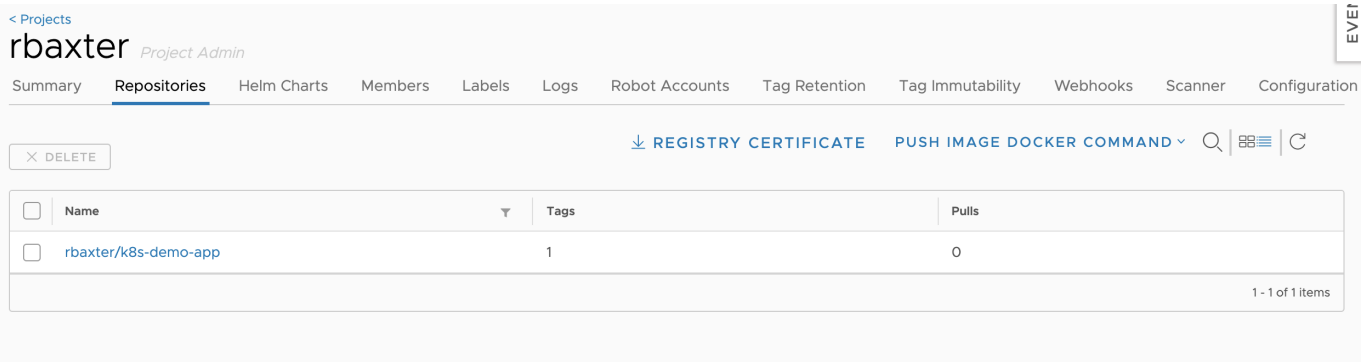
```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>1.8.0</version>
    <configuration>
        <from>
            <image>openjdk:8u222-slim</image>
        </from>
        <to>
            <image>harbor.workshop.demo.ryanjbaxter.com/[USERNAME]/k8s-demo-app</i
        </to>
        <container>
            <user>nobody:nogroup</user>
        </container>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Run The Build And Deploy The Container

- Now you should be able to run the Maven build which will deploy the container to your container registry

```
$ ./mvnw clean package
```

- Go back to Harbor and you should now see the image in your registry



Deploying To Kubernetes

- With our container build and deployed to a registry we can now run this container on Kubernetes

Deployment Descriptor

- Kubernetes uses YAML files to provide a way of describing how the app will be deployed to the platform
- You can write these by hand using the Kubernetes documentation as a reference
- Or you can have Kubernetes generate it for you using kubectl
- The `--dry-run` flag allows us to generate the YAML without actually deploying anything to Kubernetes

> Be sure to replace [USERNAME] with your username in the below `kubectl` command

```
$ mkdir k8s
$ kubectl create deployment k8s-demo-app --image harbor.workshop.demo.ryanjbaxter.
```

- The resulting `deployment.yaml` should look similar to this

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: k8s-demo-app
  name: k8s-demo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: k8s-demo-app
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: k8s-demo-app
    spec:
      containers:
      - image: harbor.workshop.demo.ryanjbaxter.com/user1/k8s-demo-app
        name: k8s-demo-app
        resources: {}
status: {}
```

Service Descriptor

- A service acts as a load balancer for the pod created by the deployment descriptor
- If we want to be able to scale pods than we want to create a service for those pods

```
$ kubectl create service clusterip k8s-demo-app --tcp 80:8080 -o yaml --dry-run >
```

- The resulting `service.yaml` should look similar to this

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: k8s-demo-app
  name: k8s-demo-app
spec:
  ports:
  - name: 80-8080
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: k8s-demo-app
  type: ClusterIP
status:
  loadBalancer: {}
```

Apply The Deployment and Service YAML

- The deployment and service descriptors have been created in the `/k8s` directory
- Apply these to get everything running
- If you have `watch` installed you can watch as the pods and services get created

```
$ watch -n 1 kubectl get all
```

- In a separate terminal window run

```
$ kubectl apply -f ./k8s
```

```
Every 1.0s: kubectl get all                                  Ryans-MacBook-Pro.loca

NAME                                 READY   STATUS    RESTARTS   AGE
pod/k8s-demo-app-d6dd4c4d4-7t8q5     1/1     Running   0          68m

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/k8s-demo-app    ClusterIP   10.100.200.243   <none>        80/TCP     68m

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/k8s-demo-app    1/1     1            1           68m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/k8s-demo-app-d6dd4c4d4     1         1         1       68m
```

`watch` is a useful command line tool that you can install on Linux (https://www.2daygeek.com/linux-watch-command-to-monitor-a-command/) and OSX (https://osxdaily.com/2010/08/22/install-watch-command-on-os-x/). All it does is continuously executes the command you pass it. You can just run the `kubectl` command specified after the `watch` command but the output will be static as opposed to updating constantly.

Testing The App

- The service is assigned a cluster IP, which is only accessible from inside the cluster

```
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/k8s-demo-app    ClusterIP   10.100.200.243   <none>        80/TCP     68m
```

- To access the app we can use `kubectl port-forward`

```
$ kubectl port-forward service/k8s-demo-app 8080:80
```

- Now we can `curl` localhost:8080 and it will be forwarded to the service in the cluster

```
$ curl http://localhost:8080; echo
Hello World
```

Congrats you have deployed your first app to Kubernetes 🎉

via GIPHY (https://giphy.com/gifs/day-subreddit-msKNSs8rmJ5m)

Be sure to stop the `kubectl port-forward` process before moving on

Exposing The Service

> NOTE: `LoadBalancer` features are platform specific. The visibility of your app after changing the service type might depend a lot on where it is deployed (e.g. per cloud provider).

- If we want to expose the service publically we can change the service type to `LoadBalancer`
- Open `k8s/service.yaml` and change `ClusterIp` to `LoadBalancer`

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: k8s-demo-app
  name: k8s-demo-app
spec:
...
  type: LoadBalancer
...
```

- Now apply the updated `service.yaml`

```
$ kubectl apply -f ./k8s
```

Testing The Public LoadBalancer

- Kubernetes will assign the service an external ip
  - It may take a minute or so for Kubernetes to assign the service an external IP, until it is assigned you might see `<pending>` in the `EXTERNAL-IP` column

```
$ kubectl get service k8s-demo-app -w
NAME           TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)        AGE
k8s-demo-app   LoadBalancer   10.100.200.243   35.202.115.69   80:31428/TCP   85m
$ curl http://35.202.115.69; echo
Hello World
```

> The `-w` option of `kubectl` lets you watch a single Kubernetes resource.

Best Practices

Liveness and Readiness Probes

- Kubernetes levages two probes to determine if the app is ready to accept traffic and whether the app is alive
- If the readiness probe does not return a `200` no trafic will be routed to it
- If the liveness probe does not return a `200` kubernetes will restart the Pod
- Spring Boot has a build in set of endpoints from the Actuator (https://spring.io/blog/2020/03/25/liveness-and-readiness-probes-with-spring-boot) module that fit nicely into these use cases
  - The `/health/readiness` endpoint indicates if the application is healthy, this fits with the readiness proble
  - The `/health/liveness` endpoint serves application info, we can use this to make sure the application is “alive”

The `/health/readiness` and `/health/liveness` endpoints are only available in Spring Boot 2.3.x.

Add The Readiness Probe

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
  name: k8s-demo-app
spec:
...
  template:
    ...
    spec:
      containers:
        ...
        readinessProbe:
          httpGet:
            port: 8080
            path: /actuator/health/readiness
```

Add The Liveness Probe

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
  name: k8s-demo-app
spec:
...
  template:
    ...
    spec:
      containers:
        ...
        livenessProbe:
          httpGet:
            port: 8080
            path: /actuator/health/liveness
```

Graceful Shutdown

- Due to the asyncronous way Kubnernetes shuts down applications there is a period of time when requests can be sent to the application while an application is being terminated.
- To deal with this we can configure a pre-stop sleep to allow enough time for requests to stop being routed to the application before it is terminated.
- Add a `preStop` command to the `podspec` of your `deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
  name: k8s-demo-app
spec:
...
  template:
    ...
    spec:
      containers:
        ...
        lifecycle:
          preStop:
            exec:
              command: ["sh", "-c", "sleep 10"]
```

Handling In Flight Requests

- Our application could also be handling requests when it receives the notification that it need to shut down.
- In order for us to finish processing those requests before the applicaiton shuts down we can configure a “grace period” in our Spring Boot applicaiton.
- Open `application.properties` in `/src/main/resources` and add

```
server.shutdown.grace-period=30s
```

> `server.shutdown.grace-period` is only available begining in Spring Boot 2.3.x

Update The Container & Apply The Updated Deployment YAML

```
$ ./mvnw clean package
$ kubectl apply -f ./k8s
```

- An updated Pod will be created and started and the old one will be terminated
- If you use `watch -n 1 kubectl get all` to see all the Kubernetes resources you will be able to see this appen in real time

Cleaning Up

- Before we move on to the next section lets clean up everything we deployed

```
$ kubectl delete -f ./k8s
```

Skaffold

- Skaffold (https://github.com/GoogleContainerTools/skaffold) is a command line tool that facilitates continuous development for Kubernetes applications
- Simplifies the development process by combining multiple steps into one easy command
- Provides the building blocks for a CI/CD process
- Make sure you have Skaffold installed before continuing

```
$ skaffold version
v1.3.1
```

Adding Skaffold YAML

- Skaffold is configured using...you guessed it...another YAML file
- We can easily initialize a YAML file using the `skaffold` command line

```
$ skaffold init --XXenableJibInit
apiVersion: skaffold/v2alpha1
kind: Config
metadata:
  name: k-s-demo-app--
build:
  artifacts:
  - image: harbor.workshop.demo.ryanjbaxter.com/rbaxter/k8s-demo-app
deploy:
  kubectl:
    manifests:
    - k8s/deployment.yaml
    - k8s/service.yaml

Do you want to write this configuration to skaffold.yaml? [y/n]: y
```

- This will create `skaffold.yaml` in the root of our project

Development With Skaffold

- Skaffold makes some enhacements to our development workflow when using Kubernetes
- Skaffold will
  - Build the app (Maven)
  - Create the container (Jib)
  - Push the container to the registry (Jib)
  - Apply the deployment and service YAMLs
  - Stream the logs from the Pod to your terminal
  - Automatically setup port forwarding

```
$ skaffold dev --port-forward
```

Testing Everything Out

- If you are `watch` ing your Kubernetes resources you will see the same resources created as before
- When running `skaffold dev --port-forward` you will see a line in your console that looks like

```
Port forwarding service/k8s-demo-app in namespace rbaxter, remote port 80 -> addre
```

- In this case port `4503` will be forwarded to port `80` of the service

```
$ curl localhost:4503; echo
Hello World
```

Make Changes To The Controller

- Skaffold is watching the project files for changes
- Open `K8sDemoApplication.java` and change the `hello` method to return `Hola World`
- Once you save the file you will notice Skaffold rebuild and redeploy everthing with the new change

```
$ curl localhost:4503; echo
Hola World
```

Cleaning Everything Up

- Once we are done, if we kill the `skaffold` process, skaffold will remove the deployment and service resources

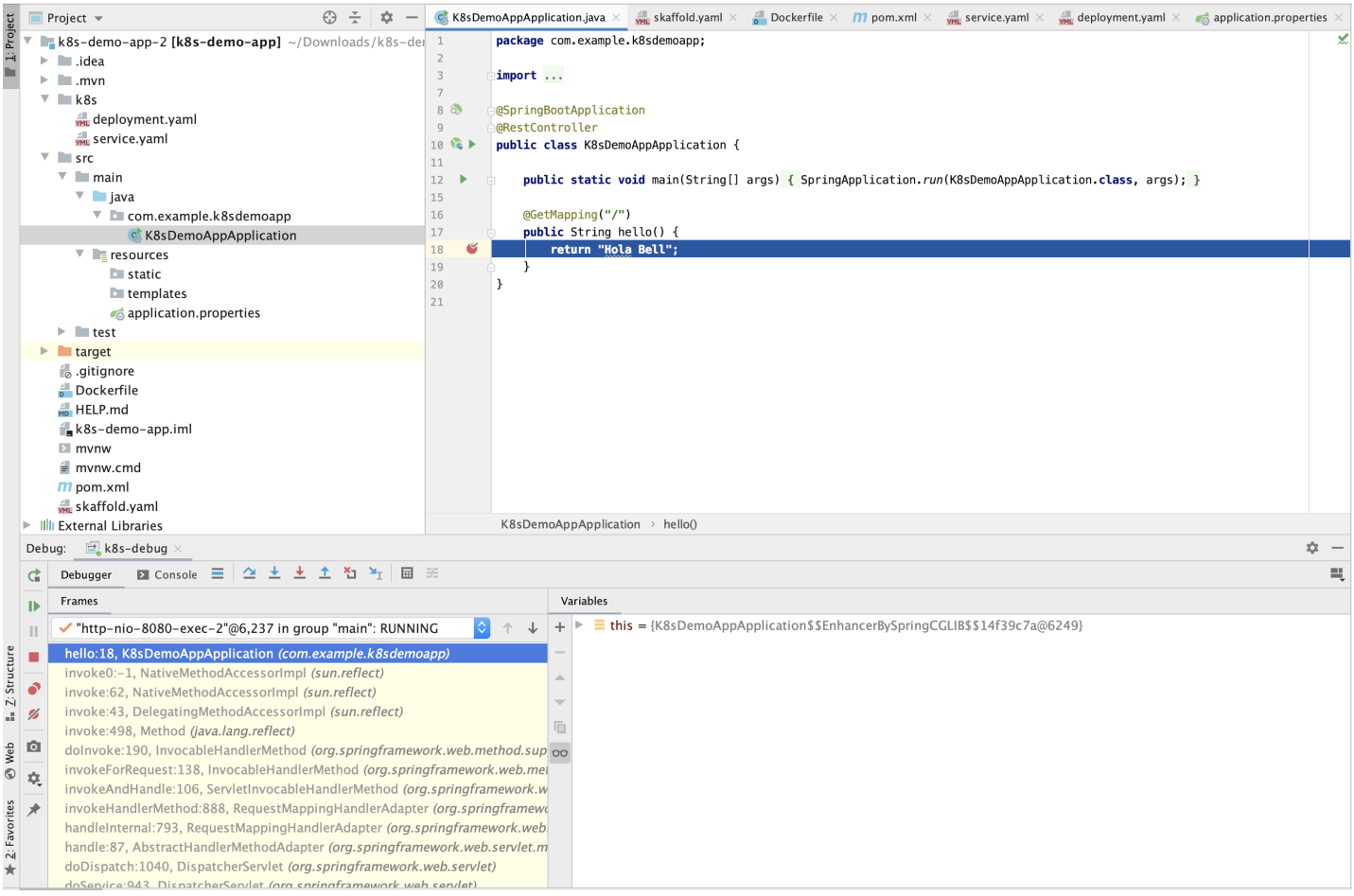
```
WARN[2086] exit status 1
Cleaning up...
 - deployment.apps "k8s-demo-app" deleted
 - service "k8s-demo-app" deleted
```

Debugging With Skaffold

- Skaffold also makes it easy to attach a debugger to the container running in Kubernetes

```
$ skaffold debug --port-forward
...
Port forwarding service/k8s-demo-app in namespace rbaxter, remote port 80 -> addre
Watching for changes...
Port forwarding pod/k8s-demo-app-75d4f4b664-2jqvx in namespace rbaxter, remote por
...
```

- The `debug` command results in two ports being forwarded
  - The http port, `4503` in the above example
  - The remote debug port `5005` in the above example
- You can then setup the remote debug configuration in your IDE to attach to the process and set breakpoints just like you would if the app was running locally
- If you set a breakpoint where we return `Hola World` from the `hello` method in `K8sDemoApplication.java` and then issue our `curl` command to hit the endpoint you should be able to step through the code



Be sure to detach the debugger and kill the `skaffold` process before continuing

Kustomize

- Kustomize (https://kustomize.io/) another tool we can use in our Kubernetes toolbox that allows us to customize deployments to different environments
- We can start with a base set of resources and then apply customizations on top of those
- Features
  - Allows easier deployments to different environments/providers
  - Allows you to keep all the common properties in one place
  - Generate configuration for specific environments
  - No templates, no placeholder spaghetti, no environment variable overload

Getting Started With Kustomize

- Create a new directory in the root of our project called `kustomize/base`
- Move the `deployment.yaml` and `service.yaml` from the `k8s` directory into `kustomize/base`
- Delete the `k8s` directory

```
$ mkdir -p kustomize/base
$ mv k8s/* kustomize/base
$ rm -Rf k8s
```

kustomize.yaml

- In `kustomize/base` create a new file called `kustomization.yaml` and add the following to it

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

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

> NOTE: Optionally, you can now remove all the labels and annotations in the metadata of both objects and specs inside objects. Kustomize adds default values that link a service to a deployment. If there is only one of each in your manifest then it will pick something sensible.

Customizing Our Deployment

- Lets imagine we want to deploy our app to a QA environment, but in this environment we want to have two instances of our app running
- Create a new directory called `qa` under the `kustomize` directory
- Create a new file in `kustomize/qa` called `update-replicas.yaml`, this is where we will provide customizations for our QA environment
- Add the following content to `kustomize/qa/update-replicas.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-demo-app
spec:
  replicas: 2
```

- Create a new file called `kustomization.yaml` in `kustomize/qa` and add the following to it

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ../base

patchesStrategicMerge:
- update-replicas.yaml
```

- Here we tell Kustomize that we want to patch the resources from the `base` directory with the `update-replicas.yaml` file
- Notice that in `update-replicas.yaml` we are just updating the properties we care about, in this case the `replicas`

Running Kustomize

- You will need to have Kustomize installed (https://github.com/kubernetes-sigs/kustomize/blob/master/docs/INSTALL.md)

```
$ kustomize build ./kustomize/base
```

- This is our base deployment and service resources

```
$ kustomize build ./kustomize/qa
...
spec:
  replicas: 2
...
```

- Notice when we build the QA customization that the replicas property is updated to `2`

Piping Kustomize Into Kubectl

- We can pipe the output from `kustomize` into `kubectl` in order to use the generated YAML to deploy the app to Kubernetes

```
$ kustomize build kustomize/qa | kubectl apply -f -
```

- If you are watching the pods in your Kubernetes namespace you will now see two pods created instead of one

```
Every 1.0s: kubectl get all                                  Ryans-MacBook-Pro.loca

NAME                                 READY   STATUS    RESTARTS   AGE
pod/k8s-demo-app-647b8d5b7b-r2999   1/1     Running   0          83s
pod/k8s-demo-app-647b8d5b7b-x4t54   1/1     Running   0          83s

NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/k8s-demo-app   ClusterIP   10.100.200.200   <none>        80/TCP     84s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/k8s-demo-app   2/2     2            2           84s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/k8s-demo-app-647b8d5b7b   2         2         2       84s
```

Our service `k8s-demo-app` will load balance requests between these two pods

Clean Up

- Before continuing clean up your Kubernetes environment

```
$ kustomize build kustomize/qa | kubectl delete -f -
```

Using Kustomize With Skaffold

- Currently our Skaffold configuration uses `kubectl` to deploy our artifacts, but we can change that to use `kustomize`
- Change your `skaffold.yaml` to the following

> Make sure you replace `[USERNAME]` in the image property with your own username

```
apiVersion: skaffold/v2alpha3
kind: Config
metadata:
  name: k-s-demo-app
build:
  artifacts:
  - image: harbor.workshop.demo.ryanjbaxter.com/[USERNAME]/k8s-demo-app
    jib: {}
deploy:
  kustomize:
    paths: ["kustomize/base"]
profiles:
  - name: qa
    deploy:
      kustomize:
        paths: ["kustomize/qa"]
```

- Notice now the `deploy` property has been changed from `kubectl` to now use Kustomize
- Also notice that we have a new profiles section allowing us to deploy our QA configuration using Skaffold

Testing Skaffold + Kustomize

- If you run your normal `skaffold` commands it will use the deployment configuration from `kustomize/base`

```
$ skaffold dev --port-forward
```

- If you want to test out the QA deployment run the following command to activate the QA profile

```
$ skaffold dev -p qa --port-forward
```

> Be sure to kill the `skaffold` process before continuing

Externalized Configuration

- One of the 12 factors for cloud native apps (https://12factor.net/config) is to externalize configuration
- Kubernetes provides support for externalizing configuration via config maps and secrets
- We can create a config map or secret easily using `kubectl`

```
$ kubectl create configmap log-level --from-literal=LOGGING_LEVEL_ORG_SPRINGFRAMEV
$ kubectl get configmap log-level -o yaml
apiVersion: v1
data:
  LOGGING_LEVEL_ORG_SPRINGFRAMEWORK: DEBUG
kind: ConfigMap
metadata:
  creationTimestamp: "2020-02-04T15:51:03Z"
  name: log-level
  namespace: rbaxter
  resourceVersion: "2145271"
  selfLink: /api/v1/namespaces/rbaxter/configmaps/log-level
  uid: 742f3d2a-ccd6-4de1-b2ba-d1514b223868
```

Using Config Maps In Our Apps

- There are a number of ways to consume the data from config maps in our apps
- Perhaps the easiest is to use the data as environment variables
- To do this we need to change our `deployment.yaml` in `kustomize/base`

```
apiVersion: apps/v1
kind: Deployment
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - image: harbor.workshop.demo.ryanjbaxter.com/rbaxter/k8s-demo-app
        name: k8s-demo-app
        envFrom:
          - configMapRef:
              name: log-level
        ...
```

- Add the `envFrom` properties above which reference our config map `log-level`
- Update the deployment by running `skaffold dev` (so we can stream the logs)
- If everything worked correctly you should see much more verbose logging in your console

Removing The Config Map and Reverting The Deployment

- Before continuing lets remove our config map and revert the changes we made to `deployment.yaml`
- To delete the config map run the following command

```
$ kubectl delete configmap log-level
```

- In `kustomize/base/deployment.yaml` remove the `envFrom` properties we added
- Next we will use Kustomize to make generating config maps easier

Config Maps and Spring Boot Application Configuration

- In Spring Boot we usually place our configuration values in application properties or YAML
- Config Maps in Kubernetes can be populated with values from files, like properties or YAML files
- We can do this via `kubectl`

```
$ kubectl create configmap k8s-demo-app-config --from-file ./path/to/application.y
```

No need to execute the above command, it is just an example, the following sections will show a better way

- We can then mount this config map as a volume in our container at a directory Spring Boot knows about and Spring Boot will automatically recognize the file and use it

Creating A Config Map With Kustomize

- Kustomize offers a way of generating config maps and secrets as part of our customizations
- Create a file called `application.yaml` in `kustomize/base` and add the following content

```
logging:
  level:
    org:
      springframework: INFO
```

- We can now tell Kustomize to generate a config map from this file, in `kustomize/base/kustomization.yaml` by adding the following snippet to the end of the file

```
configMapGenerator:
  - name: k8s-demo-app-config
    files:
      - application.yaml
```

- If you now run `$ kustomize build` you will see a config map resource is produced

```
$ kustomize build kustomize/base
apiVersion: v1
data:
  application.yaml: |-
    logging:
      level:
        org:
          springframework: INFO
kind: ConfigMap
metadata:
  name: k8s-demo-app-config-fcc4c2fmcd
```

By default `kustomize` generates a random name suffix for the `ConfigMap`. Kustomize will take care of reconciling this when the `ConfigMap` is referenced in other places (ie in volumes). It does this to force a change to the `Deployment` and in turn force the app to be restarted by Kubernetes. This isn't always what you want, for instance if the `ConfigMap` and the `Deployment` are not in the same `Kustomization`. If you want to omit the random suffix, you can set `behavior=merge` (or `replace`) in the `configMapGenerator`.

- Now edit `deployment.yaml` in `kustomize/base` to have kubernetes create a volume for that config map and mount that volume in the container

```
apiVersion: apps/v1
kind: Deployment
...
spec:
   ...
  template:
    ...
    spec:
      containers:
      - image: harbor.workshop.demo.ryanjbaxter.com/rbaxter/k8s-demo-app
        name: k8s-demo-app
        volumeMounts:
          - name: config-volume
            mountPath: /config
        ...
      volumes:
        - name: config-volume
          configMap:
            name: k8s-demo-app-config
```

- In the above `deployment.yaml` we are creating a volume named `config-volume` from the config map named `k8s-demo-app-config`
- In the container we are mounting the volume named `config-volume` within the container at the path `/config`
- Spring Boot automatically looks in `./config` for application configuration and if present will use it (because the app is running in /)

Testing Our New Deployment

- If you run `$ skaffold dev --port-forward` everything should deploy as normal
- Check that the config map was generated

```
$ kubectl get configmap
NAME                              DATA   AGE
k8s-demo-app-config-fcc4c2fmcd    1      18s
```

- Skaffold is watching our files for changes, go ahead and change `logging.level.org.springframework` from `INFO` to `DEBUG` and Skaffold will automatically create a new config map and restart the pod
- You should see a lot more logging in your terminal once the new pod starts

> Be sure to kill the `skaffold` process before continuing

Service Discovery

- Kubernetes makes it easy to make requests to other services
- Each service has a DNS entry in the container of the other services allowing you to make requests to that service using the service name
- For example, if there is a service called `k8s-workshop-name-service` deployed we could make a request from the `k8s-demo-app` just by making an HTTP request to `http://k8s-workshop-name-service`

Deploying Another App

- In order to save time we will use an existing app (https://github.com/ryanjbaxter/k8s-spring-workshop/tree/master/name-service) that returns a random name
- The container for this service resides in Docker Hub (https://hub.docker.com/repository/docker/ryanjbaxter/k8s-workshop-name-service) (a public container registery)
- To make things easier we placed a Kustomization file (https://github.com/ryanjbaxter/k8s-spring-workshop/blob/master/name-service/kustomize/base/kustomization.yaml) in the GitHub repo that we can reference from our own Kustomization file to deploy the app to our cluster

Modify `kustomization.yaml`

- In your k8s-demo-app's `kustomize/base/kustomization.yaml` add a new resource pointing to the new app's `kustomize` directory

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - service.yaml
  - deployment.yaml
  - https://github.com/ryanjbaxter/k8s-spring-workshop/name-service/kustomize/base

configMapGenerator:
  - name: k8s-demo-app-config
    files:
      - application.yaml
```

Making A Request To The Service

- Modify the `hello` method of `K8sDemoApplication.java` to make a request to the new service

```
1  @SpringBootApplication
2  @RestController
3  public class K8sDemoAppApplication {
4
5          private RestTemplate rest = new RestTemplateBuilder().build();
6
7          public static void main(String[] args) {
8                  SpringApplication.run(K8sDemoAppApplication.class, args);
9          }
10
11         @GetMapping("/")
12         public String hello() {
13                 String name = rest.getForObject("http://k8s-workshop-name-ser
14                 return "Hola " + name;
15         }
16 }
```

- Notice the hostname of the request we are making matches the service name in our `service.yaml` file (https://github.com/ryanjbaxter/k8s-spring-workshop/blob/master/name-

service/kustomize/base/service.yaml#L7)

Testing The App

- Deploy the apps using Skaffold

```
$ skaffold dev --port-forward
```

- This should deploy both the k8s-demo-app and the name-service app

```
$ kubectl get all
NAME                                            READY   STATUS    RESTARTS   AGE
pod/k8s-demo-app-5b957cf66d-w7r9d               1/1     Running   0          172m
pod/k8s-workshop-name-service-79475f456d-4lqgl  1/1     Running   0          173m

NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP
service/k8s-demo-app                LoadBalancer   10.100.200.102   35.238.231.79
service/k8s-workshop-name-service   ClusterIP      10.100.200.16    <none>

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/k8s-demo-app               1/1     1            1           173m
deployment.apps/k8s-workshop-name-service  1/1     1            1           173m

NAME                                                  DESIRED   CURRENT   READY
replicaset.apps/k8s-demo-app-5b957cf66d               1         1         1
replicaset.apps/k8s-demo-app-fd497cdfd                0         0         0
replicaset.apps/k8s-workshop-name-service-79475f456d  1         1         1
```

- Because we deployed two services and supplied the `--port-forward` flag Skaffold will forward two ports

```
Port forwarding service/k8s-demo-app in namespace user1, remote port 80 -> address
Port forwarding service/k8s-workshop-name-service in namespace user1, remote port
```

- Test the name service

```
$ curl localhost:4504; echo
John
```

Hitting the service multiple times will return a different name

- Test the k8s-demo-app which should now make a request to the name-service

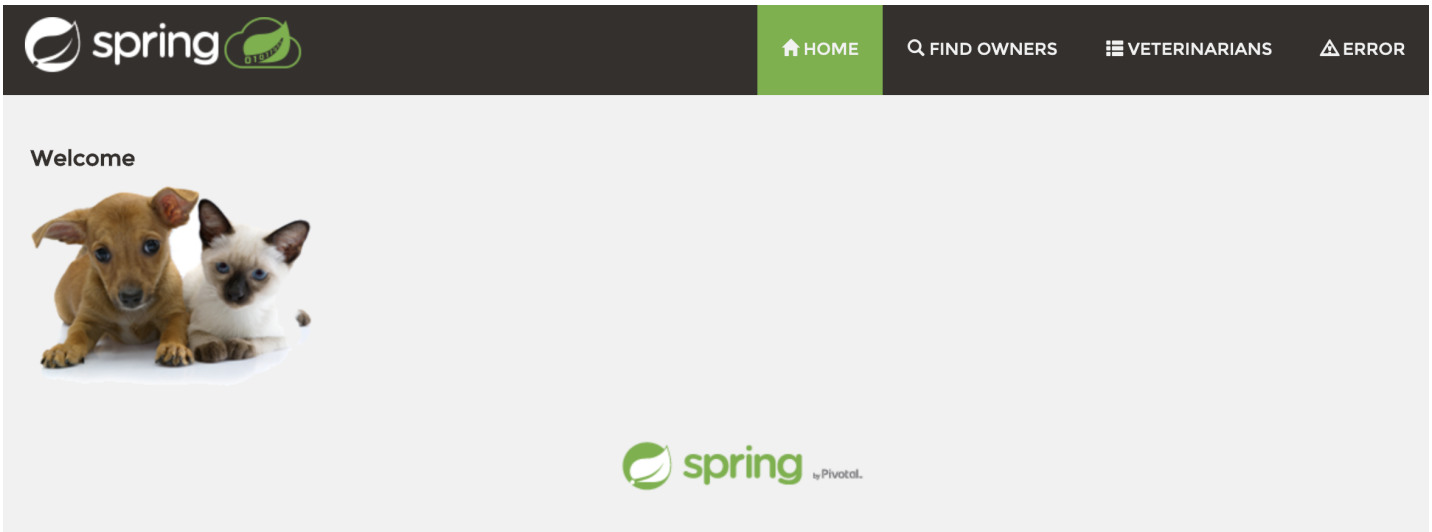
```
$ curl localhost:4503; echo
Hola John
```

> Making multiple requests should result in different names coming from the name-service

- Stop the Skaffold process to clean everything up before moving to the next step

Running The PetClinic App

- The PetClinic app (https://github.com/spring-projects/spring-petclinic) is a popular demo app which leverages several Spring technologies
  - Spring Data (using MySQL)
  - Spring Caching
  - Spring WebMVC



Deploying PetClinic

- We have a Kustomization that we can use to easily get it up and running

```
tps://github.com/dsyer/docker-services/layers/samples/petclinic | kubectl apply -f -
rd service/petclinic-app 8080:80
```

The above `kustomize build` command may take some time to complete

- Head to `http://localhost:8080` and you should see the "Welcome" page
- To use the app you can go to "Find Owners", add yourself, and add your pets
- All this data will be stored in the MYSQL database

Dissecting PetClinic

- Here's the `kustomization.yaml` that you just deployed:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- ../../base
- ../../mysql
namePrefix: petclinic-
transformers:
- ../../mysql/transformer
- ../../actuator
- ../../prometheus
images:
  - name: dsyer/template
    newName: dsyer/petclinic
configMapGenerator:
  - name: env-config
    behavior: merge
    literals:
      - SPRING_CONFIG_LOCATION=classpath:/,file:///config/bindings/mysql/meta/
      - MANAGEMENT_ENDPOINTS_WEB_BASEPATH=/actuator
      - DATABASE=mysql
```

- The relative paths `../../*` are all relative to this file. Clone the repository to look at those: `git clone https://github.com/dsyer/docker-services` and look at `layers/samples`.
- Important features:
  - `base`: a generic `Deployment` and `Service` with a `Pod` listening on port 8080
  - `mysql`: a local MySQL `Deployment` and `Service`. Needs a `PersistentVolume` so only works on Kubernetes clusters that have a default volume provider
  - `transformers`: patches to the basic application deployment. The patches are generic and could be shared by multiple different applications.
  - `env-config`: the base layer uses this `ConfigMap` to expose environment variables for the application container. These entries are used to adapt the PetClinic to the Kubernetes

environment.