ROSA DEVOPS-IN-A-BOX MANUAL

INSTALLATION

PRE-REQUISITES

Note: This accelerator is free to use, but standard AWS charges apply (EC2, ROSA, CodeCommit)

Note: You can get support at rosa-support@perficient.com

You need to be an AWS admin

You need to create a RedHat account

You need to enable ROSA in your AWS account – look for "Red Hat OpenShift Service on AWS" in the console menu and find the "Enable Red Hat OpenShift" button on the service landing page

DEPLOYMENT

Get your ROSA token from RedHat at https://console.redhat.com/openshift/token/rosa

Launch the CloudFormation template:

https://us-east-1.console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/quickcreate?templateURL=https://devopsinabox-bootstrap.s3.amazonaws.com/cf-stack.yml&stackName=rosa1

Or download the template first and launch it from your AWS CloudFormation console:

https://devopsinabox-bootstrap.s3.amazonaws.com/cf-stack.yml

When getting to the CloudFormation form:

- Enter the name of your cluster
- Enter your ROSA token

The provisioning of the cluster might take between 15 and 30 minutes. You can follow the provisioning progress in the Cloudwatch logs:

In the AWS console, go to Cloudwatch > Log Groups > perficient-[your-cluser-name] > launcher and wait until you see "dev cluster ready!". For more visibility, check the View as text box at the top. You might need to click the "Load older events" link to see the entire output.

QUICK START

- In the AWS console, go to Cloudwatch > Log Groups > perficient-[your-cluser-name] > launcher locate the following information:
  - Console URL (ex: INFO `Console URL: https://console-openshift-console.apps.test-4191-prod.dab8.p1.openshiftapps.com)`

    - Admin username (cluster-admin) and password (ex: `INFO oc login https://api.test-4191-prod.dab8.p1.openshiftapps.com:6443 --username cluster-admin --password UaBcD$-IQiBw-mZm5U-i56dZ --insecure-skip-tls-verify=true`)
- Navigate to the console URL and login with the admin credentials
- In the left menu, go to pipelines and at the top of the page, switch to the “sample-ns” project
- Click the (only) pipeline name and switch to the PipelineRun tab. You should see the Tekton CI/CD pipeline being executed for the sample app
- When everything is green, open the networking menu on the left and go to routes
- Find the sample app route and click the link. The service URL should be echo-service-spring-boot-sample-ns.[cluster-host-url]. Add echo?message=test to see the sample page.

Congrats! Your first spring boot app is deployed in your new cluster.

USING THE ACCELERATOR

GITOPS SETUP

PHILOSOPHY

As a platform-as-a-service solution, Kubernetes cluster management is entirely driven by REST APIs. Every CLI commands and actions taken in the UI is ultimately translated into an API call which transfers specific resources to the Kubernetes API server. Modifications of these resources trigger the creation or update of internal objects in the cluster that describe the desired state of the cluster.

For example, you’re not “deploying an application” but instead you’re “creating a deployment” by creating a Deployment resource which describes how you want your application to be deployed on the cluster.

This mean the entire cluster state at any given time is described as a collection of configuration objects, which can be materialized as yml or json files. You can then store these files in a git repository in order to track cluster configuration changes over time, revert changes, use git flow to create configuration changes gates, etc

Perficient’s Point-of-View

We feel strongly that all changes made to a cluster should be managed with manifest files in git, as opposed to interactively using the Openshift UI or CLI. This applies to cluster settings, application deployments, scaling operations, etc. EVERYTHING IS IN GIT. Changes made interactively are not tracked, peer reviewed, and subject to human errors. Changes tracked in Git can be easily reversed, reviewed, analyzed, audited and replayed.

It also allows for the restoration of entire clusters at any point in time in case of infrastructure problems such as cloud region failures, cyberattacks, etc. It can also be used to keep multiple clusters in-sync over different datacenters or clouds.

IN PRACTICE

We use ArgoCD (Openshift GitOps) to synchronize the configuration repositories with the cluster, i.e. detect changes to yml manifests and call the Kubernetes API on your behalf. In order to do that, several ArgoCD “applications” are created to connect to various repositories.

***TIP**: ArgoCD "Application" objects are a type of Kubernetes resource that represents a subscription to a single git repository or a subdirectory inside a single git repository. The content of that repository is totally arbitrary and could be a single service, multiple services, entire stacks, etc depending on several factors though it will typically hold manifests that are logically tied together.*

Our solution installer creates 4 ArgoCD Applications which map to the following AWS CodeCommit repositories:

- **$SOLUTION_NAME-sample-app**: The application code only, no docker or Kubernetes configuration
- **$SOLUTION_NAME-cluster-config**: The manifests (yml files) for cluster-wide configuration, reserved for cluster admins only. This repository is split into 2 ArgoCD apps, one which installs custom object definitions (*main* directory) and another which manages objects using those definitions (*resources*). There is only one cluster config repo per cluster, and multiple clusters can share the same repo
- **$SOLUTION_NAME-sample-ns-config**: The manifests for the sample namespace configuration, which is seeded with the sample app configuration manifests. There will be one such repo per namespace/team/project

We recommend using one namespace - one config git repository – one ArgoCD app per team or project in the organization. Without going into further details, this allows clean separation of permissions and resource allocation in the cluster.

DEVOPS-IN-A-BOX OPERATOR

The Devops-in-a-box operator is our custom Kubernetes extension which manages 3 new kinds of objects:

- **DevSecOpsManager**: the top level resource which manages shared resources for the entire cluster like logging, monitoring, gitops, etc. There is only one instance of this object per cluster.
- **ManagedNamespace**: a resource which declares an Openshift namespace/project as managed by our operator. It manages things that are shared by a team such as CI/CD pipelines and access to private team resources like passwords to vaults, git, etc
- **SpringBootApp**: an abstraction of a Spring Boot application which declares things like source code location, app configuration, autoscaling instructions, etc and is in charge of deploying a service into a ManagedNamespace, creating a load balancer and DNS, registering the service with the monitoring and logging system, setting up distributed tracing, and a lot more.

Perficient's Point-of-View

There are many reasons we chose to use an operator to manage our objects is that operator upgrades propagate to all existing resources in a cluster. This is a huge advantage compared to typically used helm charts which require an explicit update of each individual chart that is already deployed on a cluster.

This means that an enhancement to the SpringBootApp resource such as a security tweak, will not only be immediately available to every existing Spring Boot services, but it can also be enforced across an entire organization.

These objects are manifested in Git as yaml files, like any other Kubernetes resource. Example:

```yaml
apiVersion: apimc.com.perficient/v1alpha1
kind: SpringBootApp
metadata:
  name: echo-service
spec:
  autoscaling:
    enabled: false
    maxReplicas: 200
    minReplicas: 1
    targetCPUUtilizationPercentage: 80
  config:
    repo: ${SOLUTION_NAME}-sample-ns-config
```

(…)

BASIC FLOW

1. Admin creates a namespace/team repo
2. Admin creates a Namespace resource yaml file in the cluster config repo
3. Admin creates a ManagedNamesoace resource yaml file in that cluster config repo
4. Developer creates a SpringBootApp manifest in the namespace/team repo
5. The application gets built and deployed

IMPORTANT: notice how this approach doesn't require installing any extra tools since it's only using text files and git. It also means a developer doesn't require any prior knowledge of containers or kubernetes, nor does it require access to any cluster. Everything is driven by git, which developers are already familiar with. This also means you can use git flow to control deployments of applications, i.e. peer review and pull-request any kind of changes to the cluster.

CI/CD

Devops-in-a-box uses Tekton (Openshift Pipelines) as a container-native CI/CD tool. This allows us to treat ci/cd pipelines like any other resource in Openshift and leverage the declarative setup to maximize re-usability of tasks. And because our pipelines are resources managed by our operator, enhancements made overtime will propagate to all clusters and allow every teams to benefit instantly.

IMPORTANT: The AWS solution assumes CodeCommit as the Git provider. If you need to use a different provider, please contact Perficient for a custom build.

BASIC FLOW

Upon triggering a build (typically a commit to the application code main branch)

1. **Tekton**
   a. Pull application code from git
   b. Compile code
   c. Containerize application
   d. Push container

    e. Update SpringBootApp resource with new container tag
    f. Push updated SpringBootApp resource to namespace repo
2. **ArgoCD**
    a. Detects the change in the SpringBootApp yaml file
    b. Applies the change to the cluster
3. **Devops-in-a-box operator**
    a. Detects the change in the SpringBootApp resource
    b. Creates or updates the underlying Kubernetes objects to deploy the new version

IMPORTANT: The CI/CD process is a collaboration between 3 autonomous systems. It is critical to understand that with this model, application deployment is asynchronous, i.e. the last step of the CI pipeline is simply to declare that a new version is available, not the actual deployment itself. It is an eventually consistent, event-driven system in which each component is triggered by an event generated by an upstream component. This is in contrast to a classic CI/CD tool like Jenkins in which a single pipeline usually drives the entire CI/CD process in a synchronous fashion.

IMPORTANT: Naming convention is that the name of the yaml file containing the SpringBootApp resource is the same as the SpringBootApp name inside the manifest with the "-spring-boot" suffix. Ex: echo-service-spring-boot.yaml

```
apiVersion: apimc.com.perficient/v1alpha1
kind: SpringBootApp
metadata:
  name: echo-service
```

This name is used by the CI/CD pipeline to know which manifest file to update with the new container image version.

You can follow the progress of each component by going to their respective UI:

- **Tekton:** go to the Openshift UI and navigate to the Pipelines menu
- **ArgoCD**: go to the Openshift UI and click the little squares menu on the top. You will see the ArgoCD link. After signing in with your Openshift credentials, open the sample-ns application to see the deployment status of your services

- **SpringBootApp operator**: there is no UI for that at the moment but you can go to the Openshift UI > Administration > Custom Resources > SpringBootApp and find the resource app corresponding to your service. The resource instance page will show the current sync status of the resource.

Perficient's Point-of-View

We treat the organization as a global system and we use CI as a way to declare a desired change in the global system. Using this essentially event-driven approach to deploying applications allows us to separate desired system state from physical implementation. Each cluster is then responsible to make sure its state is in compliance with the organization requirements. This pull vs push model affords us more flexibility to add and remove clusters at will, removes coupling between systems, and introduces a single source of truth for the entire organization.

RELEASE MANAGEMENT

With this model, application builds are immutable, which means the same container image is deployed to multiple environments, and configured through environment variables. Using immutable images is now industry standard and well documented.

For dev clusters

The CI pipeline default behavior is to update the SpringBootApp manifest with the new container version in the targeted namespace configuration repository. It does so by pushing the commit directly into the main branch of the ManagedNamespace configuration repository. Out of the box, the dev cluster ArgoCD app for that managed namespace is mapped to the main branch so the update is immediately applied without human intervention.

For staging and production clusters

After committing the SpringBootApp change to the main branch, the CI pipeline will automatically create a branch with the same commit and start a pull-request for the staging and/or prod branch, which has a corresponding ArgoCD application on the staging and/or prod clusters.

Upgrading to the latest version of the service on the staging/production cluster(s) is simply a matter of approving and merging that pull-request. You can then use git permissions to decide who is allowed to promote builds to higher environments.

NOTE: The free version of the solution doesn't come with a production cluster, but you can still simulate version promotion by creating an additional fake production managed namespace, which points to the prod branch. See documentation further down to learn how to do that.

NOTE: there could be new environment variables for that version of the service in which case you will need to update your application configuration manifest accordingly on that branch first. See next chapter to learn how to use external configuration.

Perficient's Point-of-View

The process of promoting applications through git though is our personal opinion based on experience. Having that level of traceability afforded by git allows us to quickly roll-back problematic builds, know what version of every applications are/were deployed at any point in time and do sophisticated auditing and analysis using standard tools.

APPLICATION CONFIGURATION

Since our service builds are immutable, you will need to ensure your applications are built to be environment-agnostic which means externalizing configuration parameters. Spring Boot configuration management system is very mature so we are taking full advantage of that capability in our solution.

Environment specific variables fall into two categories:

- **Non-sensitive**: logging levels, database host names, etc
- **Sensitive**: passwords, certificates, etc

There are multiple ways to pass configuration properties in Kubernetes. One is through ConfigMaps for non-sensitive information, and one is through Secrets for the others. Both types of resources can be then passed to the application by mapping them to Environment Properties or mounting them as text file.

Perficient's Point-of-View

Spring Boot supports both environment variables override or property files but we opted for the file option in our operator. This allows us to use typical application.properties or application.yml files as a starting point to create our ConfigMaps and Secrets manifests and make it easier for developers to keep them in sync.

ADDING CONFIGURATION PROPERTIES

The Spring Boot operator will expect a ConfigMap or Secret resource to be available in the same namespace as the application with the same name as the SpringBootApp resource. It will then mount that resource as a file onto the service pods at a well-known location and set an environment variable to tell Spring Boot where to find that file.

1. Create a ConfigMap or Secret manifest in the managed namespace config repo, in the same location as your SpringBootApp manifest
2. Make sure the name of the resource (inside the file, the file name itself doesn't matter) matches the name of the SpringBootApp resource (again, inside the SpringBootApp manifest) plus the "-config" suffix
3. Copy the content of the your application.yml or application.properties in that resource (see sample app for syntax example) and change according to the environment
4. Repeat for each additional environment branches (staging, prod, etc)

Example:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: echo-service-spring-boot
data:
  application.properties: |
    logging.level.org.springframework.web=DEBUG
    spring.zipkin.baseUrl=http://jaeger-all-in-one-inmemory-collector.perficient-
operators.svc:9411
```

Notice the name of the ConfigMap is the name of the SpringBootApp + “-spring-boot”:

```
apiVersion: apimc.com.perficient/v1alpha1
kind: SpringBootApp
metadata:
  name: echo-service
```

IMPORTANT: Make sure the data: field contains the application.properties as the key, as the operator will expect a file with that name mounted on the pod.

MODIFYING A CONFIGURATION PROPERTY

At the moment, the operator doesn’t support automatic rollout of configuration changes, i.e. modifications of a ConfigMap or Secret resource in git will not trigger an application update without an explicit intervention.

There are different ways to achieve the desired result, including just pushing an empty commit to the application code repository, which will trigger the CI/CD pipeline BUT… This will start a new build of your application without a code change which is a waste. The better way is to add an annotation with a new unique value in your SpringBootApp manifest. Changes to annotations will always cause Kubernetes to initiate a rollout and reload the configuration resources.

HANDLING OF SENSITIVE INFORMATION

Typically you want to avoid storing passwords and certificates in Git, so adding a plain secret directly into the managed namespace repository as explained above is not necessarily a safe option. Note that if your organization controls its own git server and your security is tight, that might not be a problem, but for git-as-a-service users, it’s definitely not recommended.

You have multiple options in that case and you will need to evaluate the pros and cons of each of them for your organization specific requirements. Reach out to Perficient for recommendations and custom implementations.

Sealed secrets: in that scenario, Secret resources are first encrypted by a cluster administrator with a private certificate, then committed to Git. The Secret is then decrypted by the cluster which has a copy of the certificate. This option allows you to use the familiar application.properties or application.yml format but requires an extra step by a privileged user with access to the certificate. It also requires admins to keep the clear version somewhere so it can be re-encrypted if the certificate is revoked and other cases. This is the only supported option in devops-in-a-box in the free version and you will have to come up with your own encryption flow.

Follow the same instructions as for the ConfigMap above but create a Secret object instead. You also need to rename the data field to stringData. **Do not commit that file directly.** You still need to encrypt the secret as a next step: https://github.com/bitnami-labs/sealed-secrets.

```
kind: Secret
apiVersion: v1
metadata:
  name: echo-service-spring-boot
stringData:
  application.properties: |
    logging.level.org.springframework.web=DEBUG

    spring.zipkin.baseUrl=http://jaeger-all-in-one-inmemory-collector.perficient-
operators.svc:9411
```

Perficient's Point-of-View

Don't let this process block you in a development environment, it can be a little bit harder to work out at the beginning. In most cases having passwords in clear in a ConfigMap for development databases is acceptable, but you will certainly have to deal with it at some point.

Vaults: this approach doesn't use files for environment variables, but instead specifically lists environment variables in the SpringBootApp manifest. The values for the properties are loaded from an external vault system like AWS Secrets Manager, Hashicorp, etc. and injected in the environment on pod startup. Some teams find that approach easier because it doesn't require encryption but there is a security pre-requisite that is specific to each vendor. The most commonly used Kubernetes extension to implement this option is Kubernetes Secrets Store CSI Driver. This is not supported in the free version of our operator but we can customize the build to your specific requirements.

APPLICATION HEATH PROBES

In order to monitor the health of your application and enable advanced release management features such as canary deployments, Kubernetes requires your application to expose liveness and readiness probes. While this can mean different things to different applications, Spring Boot has a handy extension called Spring Boot actuator which makes that job really easy if you don't need to customize anything.

Make sure you add the actuator dependency to your application to enable that feature. The application will not be able to start properly if you don't, but you do not have to do anything else besides adding the dependency.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
```

```
</dependency>
```

APPLICATION METRICS

While our operator automatically registers your application with the monitoring system, you need to make sure your service is exposing its metrics in a Prometheus-compliant format. With Spring Boot the easiest way to do that is by adding Spring Boot actuator and the Prometheus libraries to your dependencies.

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
	<groupId>io.micrometer</groupId>
	<artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

If present, the operator will automatically configure the actuator on your behalf. If your application is overriding the default actuator settings, you will need to customize the operator to comply with your specific requirements. Contact Perficient for help.

VISUALIZATION

While Openshift comes with an instance of Grafana for metrics visualization, the provided dashboards are container centric and very generic in nature. You cannot alter the Grafana instance without risking breaking Redhat support. In order to provided Spring Boot specific dashboards and the ability to create custom application dashboards as well, our operator installs its own Grafana instance and configures the SpringBootApp instances automatically.

To access the Grafana dashboard, go to the Openshift UI and navigate to Networking -> Routes. Switch to the perficient-operators project at the top, and find the Grafana route. You will see a Spring Boot dashboard on that page which lists all your currently deployed applications.

APPLICATION LOGS

Openshift uses the FluentD + Elasticsearch stack to store application logs. If you're using standard Spring Boot logging, your application will log messages in plain text to the standard out (console logs) at the level specified in your application.properties.

```
<dependency>
	<groupId>net.logstash.logback</groupId>
	<artifactId>logstash-logback-encoder</artifactId>
	<version>6.3</version>
</dependency>
```

In order to filter and search through logs more efficiently though, Openshift now supports structured logging which means JSON logs can be parsed and indexed so individual fields can be used for analysis purposes.

Our operator will automatically configure Openshift to understand Spring Boot logs. All you have to do is to switch your application to produce JSON logs, and in Spring Boot, this can be achieved by adding the logstash-logback-encoder dependency to your application and configure a jsonConsoleAppender in your logback-spring.xml file (see sample app example for syntax).

To access the application logs in Kibana (Elasticsearch UI), go to the Openshift UI and navigate to Networking -> Routes. Switch to the openshift-logging project (you might need to toggle the “show default projects” button) at the top and find the Kibana route.

Once in Kibana, you need to create an index pattern that our operator created in Elasticsearch automatically. Go to Stack Management > Index Patterns and add an index with the pattern app-spring-boot-*. Then on the left menu, go to Discover to see the log entries. You can now filter by specific fields on the left side, search by keywords, build log graphs, etc

IMPORTANT: the free version uses an ephemeral log storage which might be destroyed if the Elasticsearch pods need to be re-located to different VMs. Do not rely on that for production, as long-term storage requires careful planning.

SECURITY BASICS

When deploying the solution, a default cluster-admin user is created. For evaluation purposes, you’re welcome to use that account and distribute to developers so they can access the various tools outlined in this document.

NOTE: As explained previously, our recommendation is to never make changes directly on the cluster and instead use Git and let the GitOps tool apply the update. While this is crucial on a production cluster, we tend to treat our development clusters as production systems for the organization. A breaking change to a development cluster can cripple one or several development teams for a few hours and updates made outside of Git are hard to reverse or even identify. It’s important that you enforce that process really early on to avoid forming bad habits.

NOTE: This solution doesn’t setup an identity provider out of the box because each organization has a different ways to manage their users. We recommend that you setup an OIDC provider as soon as possible so you don’t distribute the admin account to inexperienced users. You can then use read-only groups to give developers access to CI/CD, logging, monitoring tools. Contact Perficient for help.

DEPLOYING YOUR OWN APPLICATION

Let’s now use the default configuration to deploy your own code in place of the sample application

- In the AWS console, go to CodeCommit
- Locate the sample app repository created automatically during provisioning
- Clone the repository to your local machine (CodeCommit authentication required, see AWS doc)
- Replace the sample code with your own and push to the repository
- Go back to the Openshift console Pipeline page and see the new PipelineRun executing
- When everything is green, wait a few minutes and refresh the sample app page we opened before

You should now see your application output instead

APPLICATION PROPERTIES

To pass environment variables to your new application, follow these steps:

1. In the AWS console, go to CodeCommit
2. Locate the sample namespace repository created automatically during provisioning
3. Clone the repository to your local machine (CodeCommit authentication required, see AWS doc)
4. Notice the echo-service-spring-boot.yaml file which is the SpringBootApp resource for this application
5. Update the echo-service-config.yaml
6. In the echo-service-spring-boot.yaml file, add an annotation with the current timestamp
7. Commit your change to the repo and wait a few minutes for ArgoCD and the operator to sync

Your application should now be updated with the new configuration

TO CREATE A BRAND NEW APPLICATION IN THE SAMPLE NAMESPACE

Pre-requisites:

1. Your app code needs to be in CodeCommit. Create a new repository and make sure you give read access to it to the $SOLUTION_NAME-build-bot IAM user that was automatically created for you. You can see the existing policy for the sample-app repository in that user config in IAM
2. You need to create an ECR repository where the pipeline will store the container image build. Go to ECR and create a repository with your app name, with all default settings. Also make sure you give read/write access to the $SOLUTION_NAME-build-bot IAM user that was automatically created for you. You can see the existing policy for the sample-app ECR repository in that user config in IAM

Steps:

1. In the AWS console, go to CodeCommit
2. Locate the sample namespace repository created automatically during provisioning
3. Clone the repository to your local machine (CodeCommit authentication required, see AWS doc)
4. Copy both echo-service-spring-boot.yaml and echo-service-config.yaml into new files. The file names need to comply with the convention so for example: my-service-spring-boot.yaml and my-service-config.yaml if your SpringBootApp resource name is “my-service”
5. Update the content of the SpringBootApp manifest to match your new app details. You should be able to leave everything intact except for the fields that explicitly reference “echo-service”. Replace those with “my-service”.
6. Update the –config file with your environment properties if needed
7. Commit to the sample namespace config repo
8. Wait up to 5 minutes until the application is configured by ArgoCD. You can go to the ArgoCD UI sample application and watch the progression status at the top

Once ArgoCD sync is complete (you should see your commit message in the status section), you can trigger the CI/CD pipeline by pushing an empty commit to your application repository. From there just follow the same steps as outlined previously for the sample app to watch the progress of the pipeline and ArgoCD deployment.

Once everything is synced you can see the status of your service pods by going to the Openshift UI and navigate to Workloads > DeploymentConfigs. Click the link for “my-service” and it should say 1 pod ready.

If you don't see the ready pod after a few minutes, go to the Elasticsearch UI as outlined earlier in this doc and search your application logs for startup problems. Fix the issue and commit again to trigger a new build and deploy.

Your service URL should be available by going to the Openshift UI and navigate to Networking -> Routes. Look for the route named after your service and copy the URL.

CREATING A NEW NAMESPACE/PROJECT

Properly setup a new team requires a number of security pre-requisite that are out of scope for this documentation but you can easily create a clone of the sample namespace.

Pre-requisites:

1. Create a new CodeCommit repository to hold your new namespace configuration and make sure you give full access to it to the $SOLUTION_NAME-build-bot IAM user that was automatically created for you. You can see the existing policy for the sample namespace repository in that user config in IAM

Steps:

1. In the AWS console, go to CodeCommit
2. Locate the cluster config repository created automatically during provisioning
3. Clone the repository to your local machine (CodeCommit authentication required, see AWS doc)
4. Copy the resources/projects/sample-ns directory into a new directory under resources/projects
5. Open the project.yml file in the new directory and change the namespace name
6. Open the managed-ns.yml file and update the following:
    a. name and namespace fields (same value for both)
    b. repoUrl with the URL of the new namespace repo you created in the previous step
    c. keep everything the same
7. Commit

ArgoCD will detect the new namespace manifests and create a new project on the cluster. The operator will also detect the new ManagedNamespace resource and create a new ArgoCD app pointing to the config.repoBranch specified in the ManagedNamespace file.

NOTE: This namespace will use the sample AWS IAM build bot account that was created automatically during provisioning.

IMPORTANT: In a properly secured environment, each team must only have access to specific resources (application repos, container registries, etc). This is controlled through AWS IAM permissions and specified in appropriate ManagedNamespace fields. Contact Perficient for help with designing a security model tailored to your organization.

NEXT STEPS

The free version of this solution will get you up and running on Openshift on day 1 and give you a clear methodology for developing and operating services for Kubernetes. That being said, each organization is unique and there are nuances in the way you want to implement the solution depending on the needs to integrate existing CI/CD tools, security constraints, team maturity, programing language, nature of workloads, etc

These are just some of the other aspects you will need to address as you mature and get ready for production:

- Sizing
- Storage
- Security (auth for both Openshift & your own applications)
- Testing
- Versioning
- Serverless
- Cost management
- Autoscaling
- Resource quotas

Perficient already solved these problems for a lot of prestigious customers and this solution has a lot more capabilities than outlined in this document. Feel free to discover them at your own pace or contact us to fast-track adoption. Email rosa-support@perficient.com or find me online, Matthieu Rethers at your service.

We can also customize the solution for your organization and establish an Openshift Center of Excellence so you can start providing Openshift-as-a-service to all your teams.

GOING (EVEN) FURTHER

This accelerator can be used as a base for the development of more complex or focused solutions. We have leveraged this methodology for our middleware accelerator build on top of RedHat Integration which includes Kafka, Camel, K-Native, all very well suited for Spring Boot. These technologies have been successfully applied to IoT and other event-driven, streaming architecture as well as Machine Learning and AI.

Good Luck!