Zero to Cloud-Native with IBM Cloud

Part 9: Deploying the Application

Kevin Collins
kevincollins@us.ibm.com
Technical Sales Leader
IBM Cloud Enterprise Containers – Americas

Kunal Malhotra
kunal.malhotra3@ibm.com
Cloud Platform Engineer
IBM Cloud MEA

1 - Configuring Secrets

Before we can deploy our application, or at least deploy our application successfully, we need to populate a create a secrets yaml file and apply it our namespace. There are two things you will need to create the updated secrets file 1) the temporary file that you used to copy all the various parameters such as the RabbitMQ settings 2) the secrets template that you can find in the secrets-02cn Git Repository.

> **IMPORTANT:** after populating your secrets file, do not upload it to a public Git repository. If you do, you will be exposing all your API keys to the Internet.

In Visual Studio Code, open the file named **zero-to-cloud-native-secrets** in the **secrets-02cn** repository. Before editing the file, save the file locally. I recommend saving the file outside of GitHub to avoid publishing your API keys.

These are the entries that you will need to update:

```
LOGDNA_APIKEY: '<LOGDNA – APIKEY>'
```

LogDNA ingestion key that you obtained when you created your instance of LogDNA. This is required to send log message from our application directly to LogDNA. This should be in the temporary file with important parameters that you created in Part 7.

```
LOGDNA_LOGHOST: 'https://logs.us-south.logging.cloud.ibm.com/logs/ingest'
```

LogDNA ingestion host. If you followed the directions of the tutorial and deployed your application in Dallas, you can leave this setting as is. If you deployed LogDNA to another region, you will need to update this to match the region you deployed LogDNA to.

```
RABBITMQ_HOST: '<Rabbit MQ Host>'
```

Hostname of Messages for RabbitMQ. This should be in the temporary file with important parameters that you created in Part 7.

```
RABBITMQ_PORT: '<Rabbit MQ Port>'
```

Port of Messages for RabbitMQ. This should be in the temporary file with important parameters that you created in Part 7.

```
RABBITMQ_USER: 'admin'
```

For this tutorial, you can keep the user as '**admin'**.

```
  RABBITMQ_PASSWORD: '<RabbitMQ Admin Password'
```
Admin password for RabbitMQ. This should be in the temporary file with important parameters that you created in Part 7.

```
RABBITMQ_QUEUE: '<RabbitMQ Queue Name>'
```
Queue name for RabbitMQ. This should be in the temporary file with important parameters that you created in Part 7.

```
RABBITMQ_CERT_CRN: '<Certificate Manager CRN of the RabbitMQ Cert>'
```
Certificate Manager ID / CRN for the RabbitMQ TLS certificate. This should be in the temporary file with important parameters that you created in Part 7.

```
API_SERVERNAME: '<api.zero-to-cloud-native.com>'
```
Server name for the API application. In the example above, I will name the API server api.<the custom domain> that I created in Part 3.

```
WEB_SERVERNAME: '<web.zero-to-cloud-native.com>'
```
Server name for the web application. In the example above, I will use web.<the custom domain> that I created in Part 3.

```
ENVIRONMENT: 'dev'
```
Environment represents the build application environment such as dev or prod. In this tutorial, we will be using **dev**.

```
IAM_ENDPOINT: 'https://iam.cloud.ibm.com'
```
IAM endpoint to authenticate with IBM Cloud's IAM service.

```
FLASK_DEBUG: 'True'
```
Flask debug variable that will turn detailed logging on off. For this development tutorial, we will keep logging turned on.

```
REDIS_HOST: '<REDIS HOST>'
```
Hostname of IBM Cloud Databases for Redis. This should be in the temporary file with important parameters that you created in Part 7.

```
REDIS_PORT: '<REDIS PORT>'
```
Port of IBM Cloud Databases for Redis. This should be in the temporary file with important parameters that you created in Part 7.

```
REDIS_PASSWORD: '<REDIS_PASSWORD>'
```
Admin password of IBM Cloud Databases for Redis. This should be in the temporary file with important parameters that you created in Part 7.

```
REDIS_CERT_CRN: '<Certificate Manager CRN of the RabbitMQ Cert>'
```
Certificate Manager ID / CRN for the Redis TLS certificate. This should be in the temporary file with important parameters that you created in Part 7.

```
CERT_MANAGER_ENDPOINT: 'https://us-south.certificate-manager.cloud.ibm.com'
```
Endpoint for your **certificate manager endpoint**. If you followed the tutorial and created your certificate managed instance in Dallas, you can leave this setting as-is. If you create certificate manager in a different region, then you will need to update it.

```
IBMCLOUD_APIKEY: '<IBM CLOUD API KEY>'
```
Your **IBM Cloud API Key**. This should be in the temporary file with important parameters that you created in Part 7.

2 - Deploying Secrets

Now that you have created and updated you secrets file, you need to apply the secrets to your OpenShift cluster.

Following the same instructions as in *Part 7 – Preparing to Deploy – Section 8 –* log into your OpenShift cluster through the terminal.

First you will need to switch to the zero to cloud native project. On your terminal run:

oc project zero-to-cloud-native

Next, navigate to your local directory where your secrets file is located and the run the oc command to create the secrets in your OpenShift cluster.

oc create -f zero-to-cloud-native-secrets.yaml

IMPORTANT: make sure you are applying the secrets file you created with all the parameters for your environment.

All of these relevant secrets are loaded into your container as operating system environment variables by passing them in the **deployments/deployment.yaml** file. If you look at the **deployments/deployment.yaml** file you will see how we reference the secrets.

```
    containers:
    - env:
      - name: LOGDNA_APIKEY
        valueFrom:
          secretKeyRef:
            name:  zero-to-cloud-native-secrets
            key: LOGDNA_APIKEY
```

Looking at the snipped above, this shows how we are accessing the **LogDNA API Key**. We simply reference which secret contains the value we want to use and Kubernetes makes the value available to the container as an environment variable.
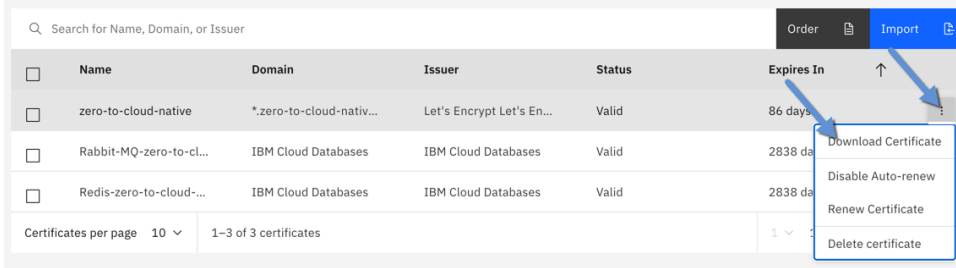
3 – Add Certificates

Before we can deploy our microservices to the cluster, we will need to add the certificates for our website to our web frontend. To do so, navigate to your list of resources in IBM Cloud and click on your certificate manage instance.
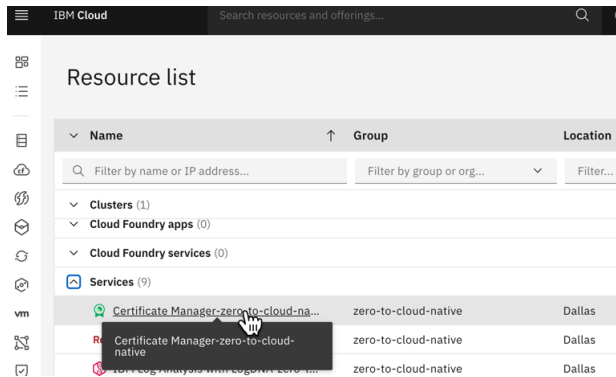
Next, click on the options on the right hand side of the screen next to the certificate for the domain you created in part 3. Click on Download Certificate.

This will download your certificate to your downloads directory as a zipfile.
Double click on the zip file to expand the certificate zip file.

You will see that there are three files after you unzip the file.

Name
▼ 📁 zero-to-cloud-native
    📄 zero-to-cloud-native_intermediate.pem
    📄 zero-to-cloud-native.key
    📄 zero-to-cloud-native.pem

```
<domain_name>_intermediate.pem
<domain_name>.key
<domain_name>.pem
```

Next, we need to copy the **domain_name.key** and **domain_name.pem** file to the location where you are storing the code for you GitHub repository.
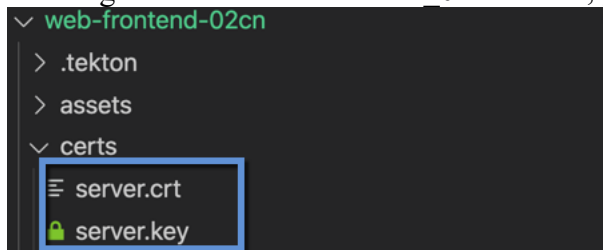
Copy the files to
`<GitHub Location>/web_frontend_02cn/certs`

The last step we will need to do is rename the two files.

Rename domain_name.key -> server.crt
Rename domain_name.pem -> server.key

Looking under the web-frontend_02cn folder, you should see the following:

∨ web-frontend-02cn
  > .tekton
  > assets
  ∨ certs
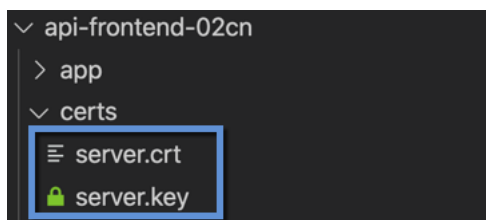   ≡ server.crt
    🔒 server.key

Next, we need to do the same thing for the API frontend microservice.

Copy the same server.crt and server.key files to
`<GitHub Location>/api_frontend_02cn/certs`
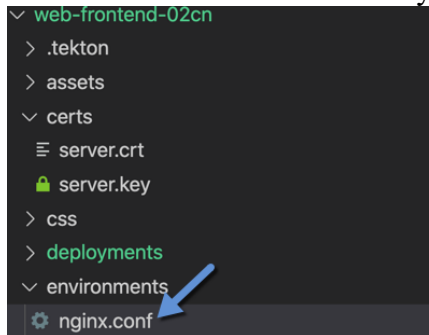
The api-frontend-02cn directory should look like this:

∨ api-frontend-02cn
  > app
  ∨ certs
    ≡ server.crt
    🔒 server.key

4 – Customizing the Web Frontend

Before deploying the Web Frontend microservice, you will need to update the code to use the domain that you create in step 3 of this tutorial series.

Open your visual studio code editor and navigate to your web-frontend-02cn microservice and under the environments directory, click on the nginx.conf file.

```
∨ web-frontend-02cn
  > .tekton
  > assets
  ∨ certs
    ☰ server.crt
    🔒 server.key
  > css
  > deployments
  ∨ environments
    ⚙ nginx.conf
```

There are lines the file that you will need to update with your domain. Simply replace the zero-to-cloud-native.com domain with the domain you created.

```
server {
  listen 443 ssl;

  server_name zero-to-cloud-native.com;
  ssl_certificate /etc/nginx/certs/server.crt;
  ssl_certificate_key /etc/nginx/certs/server.key;

  #ssl_verify_client on;

  # static file 404's aren't logged and expires header is set to maximum age
  location / {
    include  /etc/nginx/mime.types;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Host $server_name;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-Port 443;
    root /etc/nginx/html;
    index index.html;
  }
}

server {
    listen 80;
    server_name zero-to-cloud-native.com;
```

Next, we will need to update the javascript file that calls the APIs. Navigate to the js directory and click on script.js.
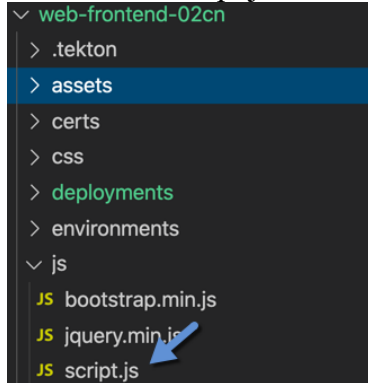
There are three locations that references zero-to-cloud-native.com that you will need to update with your domain.

getOpenshiftVersion() function

```
 9        xhr.open("GET", "https://api.zero-to-cloud-native.com:8000/api/v1/getOCPVersions/", true);
10        xhr.send();
```

getOpenshiftToken() function

```
28            xhr.open("POST", "https://api.zero-to-cloud-native.com:8000/api/v1/getOCPToken/");
29            xhr.setRequestHeader('Content-Type', 'application/json');
```

enableSSH() function

```
56            xhr.open("POST", "https://api.zero-to-cloud-native.com:8000/api/v1/enableSSH/");
57            xhr.setRequestHeader('Content-Type', 'application/json');
```

5 – Manual Deployment

Typically, I would recommend to deploy each microservice with a CI/CD pipeline which I will go through in the next section. However, it is useful to understand how to deploy manually.

These instructions will show how to manually deploy the API Frontend microservice and then you will create a CI/CD pipeline to automate the build and deployment of your microservices.

Before you can manually deploy an image, you will need to update the **image** line in the **deployments/deployment.yaml** file. You will need to update the image name to refer to the container registry namespace that you create in Part 7 of this series.

The snipped above from **deployments/deployment.yaml** shows the default image name. You will need to change the text in the bluebox to the container registry namespace you created in Part 7 of this series.

Start **iTerm2** and navigate to the directory where your **api-frontend-02cn** is located.
The first step we will need to do is build the docker container. Run the following docker command which will build a docker image for the **api-frontend-02cn** microservice with a tag of v1.

```
docker build -t us.icr.io/zero-to-cloud-native/api-frontend-02cn:v1 .
```
Change the bolded text *zero-to-cloud-native* to your container registry namespace. Don't forget the period at the end which indicates to build the docker image from the current directory.

Next, we need to push the docker image we just created to the container registry service. To do so, run the following command:

```
docker push us.icr.io/zero-to-cloud-native/api-frontend-02cn:v1
```
Again, change the bolded text *zero-to-cloud-native* to your container registry namespace.

Now that we have created our docker image and pushed it to the container registry, we can deploy it to OpenShift. To deploy the image to our OpenShift Cluster, run the following command:

oc create -f deployments/deployment.yaml

This will create a new deployment in your OpenShift cluster. To make sure the deployment and pod were successfully deployed, run the following command:

```
oc get pods
```

and you should see that your api frontend pod is running:

```
→ oc get pods
NAME                                    READY   STATUS    RESTARTS   AGE
api-frontend-02cn-7b668884f6-5jpxw      1/1     Running   0          2d18h
```

Before we can test out the application, we will need to deploy the remaining microservices. While you can deploy the remaining microservices manually, let's follow best practices and automate the deployment with a CI/CD toolchain.

When developing cloud native applications, regardless of size, one of the first and most important things you can do is create a CI/CD pipeline. By starting with a CI/CD pipeline you will save considerable amounts of time doing repetitive tasks such as building containers and deploying them to your RedHat OpenShift/Kubernetes Environment. Moreover, when working with multiple code bases (dev, staging and production) having a consistent way of building, deploying and testing will save you from hours of frustrating debugging.

IBM Cloud Continuous Delivery provides two types of delivery pipelines that you can use to build, test, and deploy your applications.

- **Classic**: Classic delivery pipelines are created graphically, with the status embedded in the pipeline diagram. These pipelines can run on shared workers in the cloud or on private workers that run on your own Kubernetes cluster.
- **Tekton**: Tekton delivery pipelines are created within yaml files that define pipelines as a set of Kubernetes resources. You can edit those yaml files to change the behaviour of a pipeline. Tekton pipelines can run on private workers that run on your own cluster. They can also run on IBM-managed workers on the public cloud. The Tekton integration provides a dashboard that you can use to view the output of Tekton pipeline runs. It also provides mechanisms for identifying the pipeline definitions repo, the pipeline triggers, where the pipeline runs, and the storage and retrieval of properties.

For this tutorial, we will go through both creating Classic and Tekton Pipelines. Classic toolchains are traditional toolchains are time test proven technique for building and deploying images. Typically, classic pipelines leverage shell scripts and are hard to transition from one environment to another. Tekton on the other hand, is a more modern approach. Tekton Pipelines project provides Kubernetes-style resources for declaring CI/CD-style pipelines. These resources are naturally described in yam and stored in a code repository. This pipeline-as-code approach provides the benefits of versioning and source control.

Tekton is become more and more mature and most of the growing pains have been addressed. I'm including both Classic, which is the safe choice, and Tekton which is certainly the choice of the future (if not present).

6 -1 Image Naming Convention

One of my favorite side benefits for deploying code with a CI/CD toolchain is ensuring that your images follow the same naming convention. In this tutorial, we will be creating toolchains (both Classic and Tekton) following the same image naming convention.

The image naming convention we will use will have the image name, environment (git branch), date the image was built, and finally the git commit number.


The reason for this naming convention is so we can easily identify a specific image and be able to quickly find the source code in GitHub. While developing CloudPak Provisioner, I started deploying images using tags such as :v1 and the next image :v2, and the third being :v3. I ran into a situation where other developers were also submitting and pushing code and one of the images failed. It took a very long time in debugging and looking at the history in GitHub to find the last image that worked. Following a naming convention like the one above will save you considerable amounts of time if you run into a similar situation.

7 – IBM Cloud Continuous Delivery – Toolchain

The first step is to login into IBM Cloud and create a toolchain at
https://cloud.ibm.com/devops/toolchains. Toolchain is a service on IBM Cloud which provides
set of tools to securely integrate, build and deploy applications. The toolchain that we will create
will contain all of the delivery pipelines for all of our microservices.

From the toolchain landing page, select the **zero-to-cloud-native** resource group where we will
deploy the toolchain along with setting the region to **Dallas** keeping all our resources in the same
geographic region.
Next, click on **Create toolchain**.
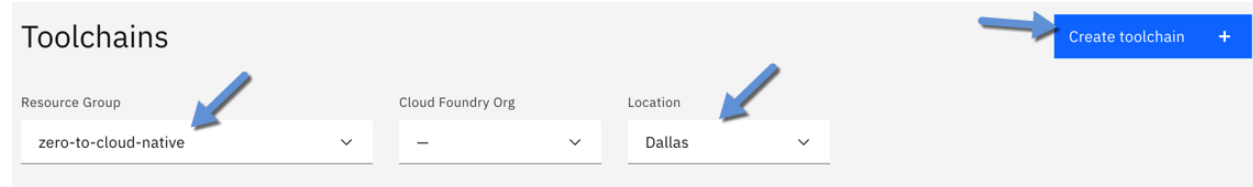
The next page describes various templates that are available to quickly build a CI/CD pipeline.
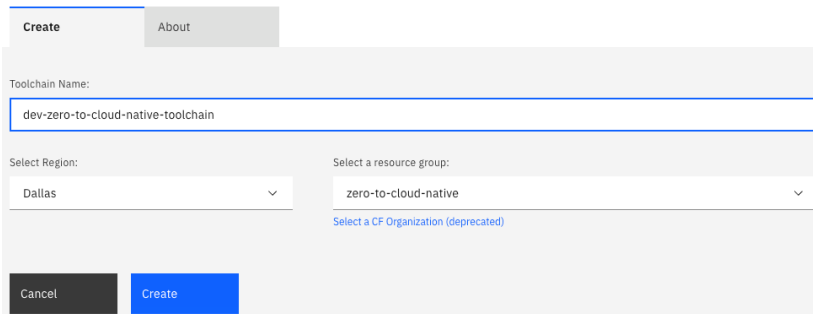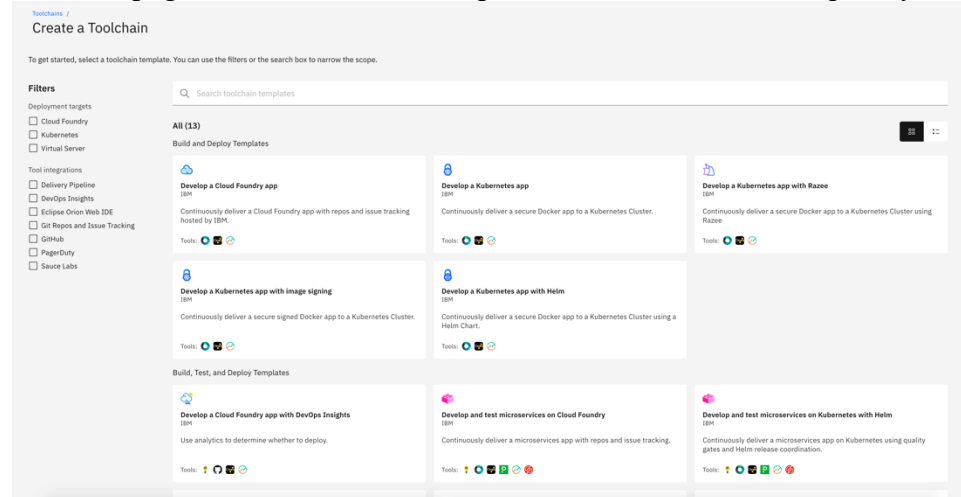

Templates are a great way to get up and running quickly, but in many cases, you will want more
control over how to architect your CI/CD pipeline. For custom toolchains where you want more
control over how the toolchain is implement, you need to build your own toolchain. To do that
select **Build your own toolchain** under the other template category as shown in the picture
below.
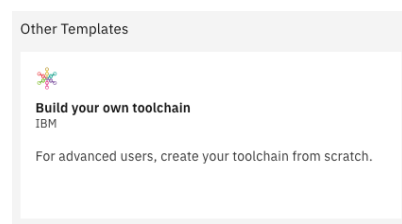
Next you need to give your toolchain a name, select **Dallas** as the region the toolchain will be created in and a the **zero-to-cloud-native** resource group. In this example, we will be creating a classic toolchain for the frontend API microservice. So we can easily find and identify which microservice and environment the toolchain is for, name the toolchain **dev-zero-to-cloud-native-toolchain**

Build your own toolchain

Create	About

Toolchain Name:

dev-zero-to-cloud-native-toolchain

Select Region:

Dallas ⌄

Select a resource group:

zero-to-cloud-native ⌄

Select a CF Organization (deprecated)

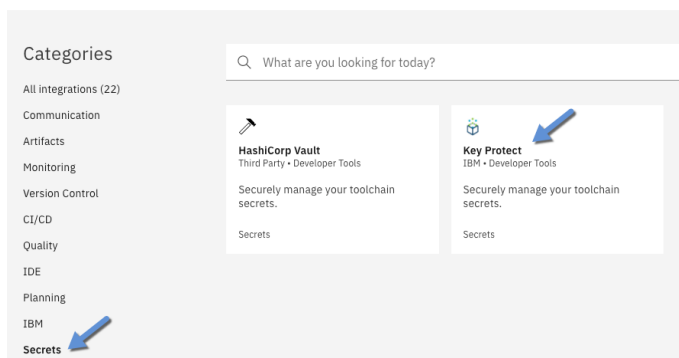Cancel   Create

After the toolchain is created, you will need to add the various tools that will comprise your toolchain.

7 − 1 Key Protect Integration

One of the great features of IBM cloud is the ability to manage the lifecycle of encryption keys used in IBM Cloud Services and for your applications. To facilitate the management of the API Keys we will be using in creating the toolchain, we will leverage IBM Key Protect to store our keys. To use Key Protect, click on **Add Tool** and on the next screen click on **Secrets** and then select **Key Protect**.

Add tool integration

Categories

All integrations (22)
Communication
Artifacts
Monitoring
Version Control
CI/CD
Quality
IDE
Planning
IBM
Secrets

🔍 What are you looking for today?

HashiCorp Vault
Third Party • Developer Tools

Securely manage your toolchain secrets.

Secrets

Key Protect
IBM • Developer Tools

Securely manage your toolchain secrets.

Secrets

On the next screen, start by giving the key protect integration a name. So I can easily find the tool later, I will use the name **dev-zero-to-cloud-native-key-protect**. Select the region you have been using throughout the tutorial, I will be using **Dallas**, select the **zero-to-cloud-native** resource group, and then finally select the **Key Protect service name** we created in Part 3.

🔮 Configure Key Protect

Create Integration

Use Key Protect to securely store and apply secrets like API keys that are part of your toolchain. Literal secret values will be stored, rotation is not yet enabled.

IBM

View Docs

TOOLCHAIN   dev-zero-to-cloud-native-toolchain

Name: ⓘ

dev-zero-to-cloud-native-key-protect

Region

Dallas   ∨

Resource group

zero-to-cloud-native   ∨

Service name

Key Protect-zero-to-cloud-native   ✕ ∨

7 − 2 Private Worker

Being rather impatient, I want my pipelines to build as quickly as possible so we will create a private worker. A private worker allows us to run the pipeline runner directly in our OpenShift/Kubernetes environment resulting in better performance of the pipeline and without the need of any inbound network connectivity.

Private workers also provide the flexibility to create pipeline jobs that can access resources outside of the public network and are not limited to a 60 minutes run time per job that you get with a shared worker.

To create a private worker, click **Add Tool** and then **Delivery Pipeline Private Worker**.

🌸🌸

Delivery Pipeline Private Worker
IBM • Developer Tools

Run pipeline workloads on your own infrastructure.

CI/CD
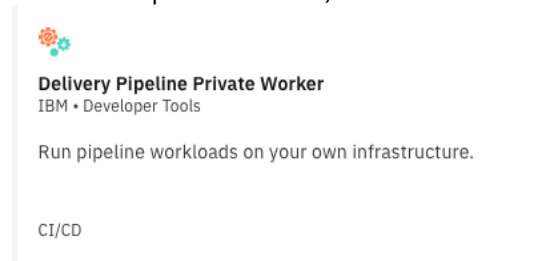
Give your private pipeline worker a name. I will use **dev-zero-to-cloud-native-private-worker** as the name of my private worker so I can easily find it.

Best Practice: For an enterprise application you will most like have different code bases such as dev, staging and production environments. A best practice is to have a private pipeline worker for each environment. We want to be able to quickly tell which pipeline worker is associated for each code base so we will start the name of the private worker with the branch name.

When you create a private pipeline worker, the worker will need access to a Service ID API Key for the service in your account. This is a key than we can store safely with our Key Protect instance we created in part 3.

7-2-1 Adding a Service ID API Key to Key Protect

Since this is the first time we are accessing the Toolchain service, we don't have a **Service ID API Key** in our Key Protect instance, so we will need to add one. Following best practices, we will create and store the Service ID API Key in our instance of Key Protect. Start by clicking on the **New Button**.

Note: as you go through the tutorial and create toolchains for the remaining service, rather than creating a new key, you can simply click on the Key Icon and select the Service ID API Key you already created.

Service ID API Key   ×

Select a secret for this field from a secrets store using the options below.

Provider

Key Protect: dev-zero-to-cloud-native-private-worker   ⌄

Secret name

zero-to-cloud-native-toolchain-key   ⌄

Cancel   OK

On the next page, I'll keep the defaults which already give good names for the **Name** and **Secret Name** that will be stored in Key Protect. Make sure to check the box '**Save this key in a secrets store for re-use**', and select the Key Protect instance you created in Part 3 under the Provider. Click **OK**.

Create a new Service ID API key   ×

This will create a Service ID and associated API key that has no access to your resources but enables access to a private worker queue. For more information on Service IDs and access see the IAM documentation.

⚠ **Warning:** You will not be able to see the value of the key again once you complete this integration. However you may copy the key to the clipboard in order to view it later.

Name   Description

Service ID for dev-zero-to-cloud-native-priva

☑ Save this key in a secrets store for reuse

Provider

Key Protect: dev-zero-to-cloud-native-private-worker   ⌄

Secret name

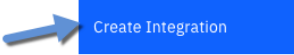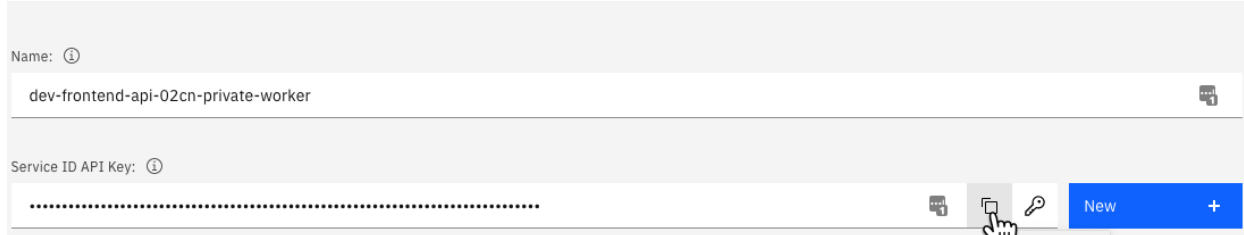dev-zero-to-cloud-native-private-worker

Cancel   OK

On next screen, your **Service ID API Key** will automatically be populated.
Important – the only chance you have to find the details of the key is right now! Make sure to click the icon to copy the key to the clipboard, we will need this value in the next step. After you **copy the key to the clipboard**, create the private worker integration by clicking **Create Integration.**

Next, we need to configure the delivery pipeline private worker and register it with our cluster. To do so, click on the **Delivery Pipeline Private Worker** tile that we just created.
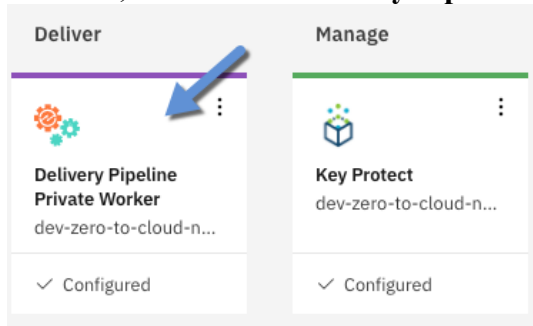
On the next screen, click on **Getting Started**.

On the next page, **paste the Service ID API Key** you just copied in the previous step. Name the Worker Name something meaningful. I will choose **dev-zero-to-cloud-native**. Then click **Generate**.

Setup and Configure Private Workers

Add private worker support and configure your worker to run jobs in your Kubernetes cluster.
Learn more about installing Delivery Pipeline Private Workers and troubleshooting general problems.

Prerequisites

- Ensure that you have cluster-admin privileges in your cluster
- Configure kubectl to point at your cluster and verify that your connection is working

Provide Worker Inputs

To enable us to display the kubectl commands needed to configure the worker in your Kubernetes cluster, please enter the Service ID API Key (which was used to create this worker tool), enter the worker agent name and then click Generate. Choose Skip to get generic kubectl commands, which you will have to modify to include the API Key & worker agent name before running.

Service ID API Key:

`.........................................`

Worker Name:

`dev-zero-to-cloud-native`

[Generate]   [Skip]

Next, you will now see instructions on how to add private worker to your cluster. The first thing you need to do is **log into your cluster** with the command line. Following the same instructions as in *Part 7 – Preparing to Deploy – Section 8 –* log into your OpenShift cluster through the terminal.

Add a new Private Worker

☑ Enable auto update ⓘ

1. Install the Delivery Pipeline Kubernetes Private Worker support

```
$ kubectl apply --filename "https://private-worker-service.us-south.devops.cloud.ibm.com/install?autoupdate=true"
```

2. Register a new worker in your cluster

```
$ kubectl apply --filename "https://private-worker-service.us-south.devops.cloud.ibm.com/install/worker?serviceId=ServiceId-9b94c286-cf5d-4b7d-9dc5-baf34c6a243d&apikey..."
```

3. Verify the worker was created on the cluster

```
$ kubectl get workeragent
```

Make sure to select **Enable auto update** so you always have the latest version of the private worker node agent on your cluster.
Copy the kubectl command in step 1 and paste it into your terminal.
Copy the kubectl command in step 2 and paste it into your terminal.
Copy the kubectl command in step 3 and paste it into your terminal.

Next, we need to update the OpenShift Security Context Constraints to allow the private worker to write to the container filesystem. To do so, you will need to enter the following command.

```
oc adm policy add-scc-to-user anyuid system:serviceaccount:tekton-
pipelines:tekton-pipelines-controller
```


After running the commands, you should see that the agent was successfully registered.

Click back to the **Overview** tab and you will see that your private worker is registered.

Registered Worker Pool ⓘ

This list shows all workers that have registered using an API key associated with the Service ID: "ServiceId-d56a117e-ac11-4999-a49b-a892cc282a12". The Runs column shows agent activity in the last hour.

Name	Status	Version	Tekton Version	Cluster	Last Active	⟳	✓	✕	↻
dev-zero-to-cloud-native	● Active	0....	0.14.1	zero-to-cloud-native ⓘ	Today at 2:53 PM	0	0	0	

Want to add or remove workers? Don't see what you expect? Check out the guide in Getting Started

Now that we have our toolchain create, the next step is to create pipelines for each microservice.
The next two parts:
9A will go creating a 'classic' pipeline for the api frontend microservice
9B will go through creating a 'tekton' pipeline for the web frontend microservice

For the remain microservices you will have your choice in creating whichever type of pipeline you prefer.

9C will then finish up deploying the application