

Zero to Cloud Native with IBM Cloud

Part 9B: Creating a Tekton Pipeline

Kevin Collins

kevincollins@us.ibm.com

Technical Sales Leader

IBM Cloud Enterprise Containers – Americas

Kunal Malhotra

kunal.malhotra3@ibm.com

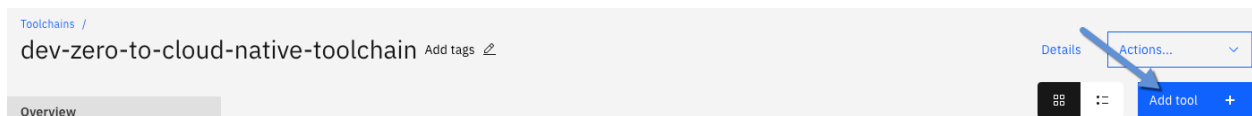
Cloud Platform Engineer

IBM Cloud MEA

1 – IBM Cloud Continuous Delivery – Tekton Pipeline

Note: This tutorial will assume you have already setup as described in Part 9 – Section 4.

This tutorial will go step by step on how to create a tekton pipeline for the Web frontend microservice. To get start, navigate to the zero-to-cloud-native-toolchain you created in part 9 for this series.



1 – 1 GitHub Integration

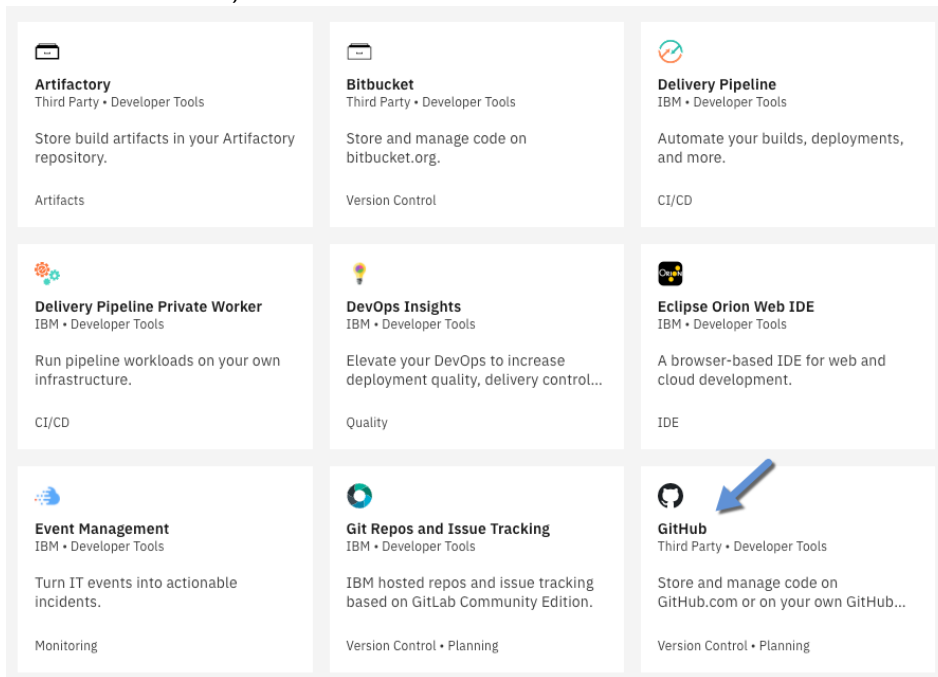
The first thing you need to do is connect your SCM (Source Code Management) repository with the toolchain. You do this by clicking add tool and then selecting SCM (GitHub, Bitbucket, GitLab, or GitHub Enterprise Whitewater).

Note: Repositories can be Public or Private. In this tutorial, I will be the public repository that I've shared and that you clone in part 8. By default, I recommend setting your repositories to private unless you plan to share them publicly. This way, if you make a mistake and upload something like an API key, you are still protected.

Note: if this is the first time you are creating a toolchain with a GitHub repository, you will first need to Authorize IBM Cloud to access GitHub. You can follow the instructions here if needed:

<https://cloud.ibm.com/docs/ContinuousDelivery?topic=ContinuousDelivery-integrations#github>

On the next screen, select **GitHub**.



Select **Existing** as the repository type and enter the repository URL for your git repository that you setup in Part 8 of this tutorial series. Click **Create Integration** after entering these settings.

The screenshot shows the GitHub integration configuration form. The fields are:

- GitHub Server:** GitHub (https://github.com)
- Authorized as:** kmcolli with access granted to zero GitHub organization(s) [Manage Authorization](#)
- Repository type:** Existing
- Link to the repository that is specified in the Repository URL field.**
- Repository URL:** https://github.com/kmcolli/web-frontend-02cn
- Integration Owner:** kmcolli
- ☒ **Enable GitHub Issues**
- ☐ **Track deployment of code changes**

A blue arrow points to the **Create Integration** button.

1 – 2 Create Delivery Pipeline

The next tool we need to add is a Delivery Pipeline. Click on **add tool** and then click on the **Delivery Pipeline** tile.



Delivery Pipeline
IBM • Developer Tools

Automate your builds, deployments, and more.

CI/CD

On the next screen, give your delivery pipeline a name, I will use **dev-tekton-frontend-web-pipeline**. For this part of the tutorial, keep **tekton** selected as the Pipeline type, and click **Create Integration**.

Pipeline name:

Pipeline type: Tekton


Tekton itself is under active development, and is currently pre-release technology. There is no guarantee of backwards compatibility between Tekton versions, and you should thus expect to adapt your pipeline definitions until Tekton achieves a level of maturity which guarantees backward compatibility. [Learn more about Tekton roadmap.](#)


☒ Show apps in the View app menu ⓘ

1 – 2 – 1 Add the Continuous Delivery Service

Note: If you are following the tutorial and created the continuous delivery service as you created a classic pipeline, you can skip this part and proceed to part 1-3. You only have to create one instance of the continuous delivery service.

On the next screen, you will see all the tools we have added to our toolchain. The next step is to configure the delivery pipeline we just created. You will also see the following warning notice.

 **Continuous Delivery service required:**
A service instance was not found in this resource group and region us-south. If you do not take action before 2020-09-07, 14:02 UTC, Delivery Pipeline jobs will not run, pushes to Git Repos and Issue Tracking will fail, and DevOps Insights will be disabled. [Add the service](#) to the resource group to ensure uninterrupted use of the service. [Learn more](#). For more information about usage and billing, see the [blog](#).



You have 5 days to use the service for free. If you want to use the service more than 5 days, click on **Add the service**.

The first step on the next page is to select a region and plan. Keep the region as **Dallas**, like all our other resources, and you can also keep the plan selected as **Lite**.

Catalog / Services /

Continuous Delivery

Author: IBM • Date of last update: 05/01/2020 • Docs

Create About

Select a region

Select a region

Dallas

Select a pricing plan

Displayed prices do not include tax. Monthly prices shown are for country or region: United States

Plan	Features	Pricing
Lite	Continuous Delivery for organizations (orgs) or resource groups of up to 5 users 5 users per organization or resource group 500 Delivery Pipeline jobs run per organization per month or per resource group per month 500 MB of private Git Repos storage per organization or resource group	Free

The Lite plan offers the full capabilities of Continuous Delivery, with usage limits, to small teams at no cost.

Lite plan services are deleted after 30 days of inactivity.

Scrolling down, you need to give you service a name and indicate which resource group to place it in. I will use **Continuous Delivery-zero-to-cloud-native** as the name and select **zero-to-cloud-native** as the resource group.

After entering those settings, click on **Create**.

Configure your resource

Service name

Continuous Delivery-zero-to-cloud-native

Select a resource group ⓘ

zero-to-cloud-native

Tags ⓘ

Examples: env:dev, version-1

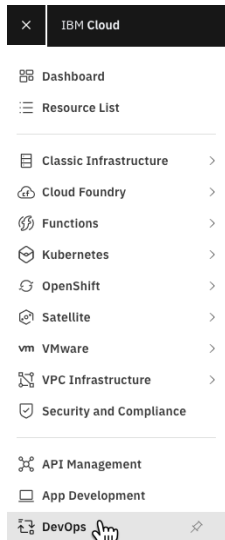
Create

Add to estimate

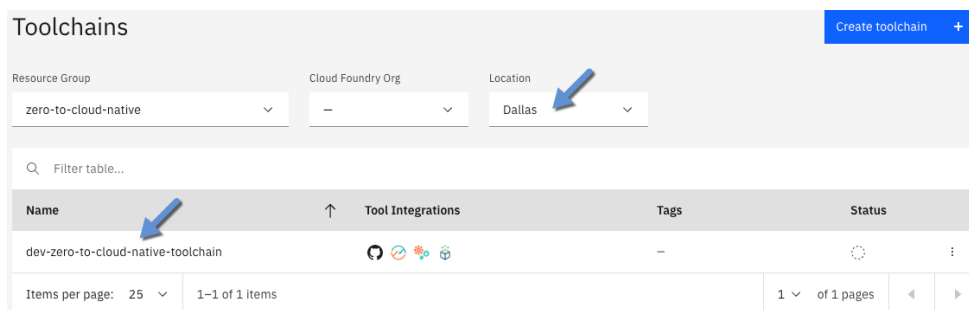
View terms

1 – 3 Configure your Delivery Pipeline

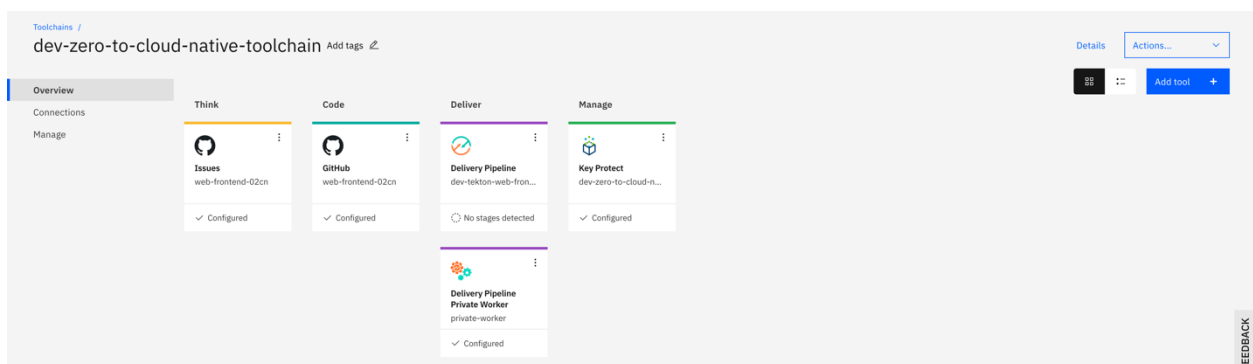
Navigate back to the toolchain you just created by clicking on the IBM Cloud hamburger menu and **select DevOps**.



On the next screen, select **Dallas** as the location your toolchain is in and then click on the **toolchain name**.



On the next screen, click on the **Delivery Pipeline** tile that you just created.

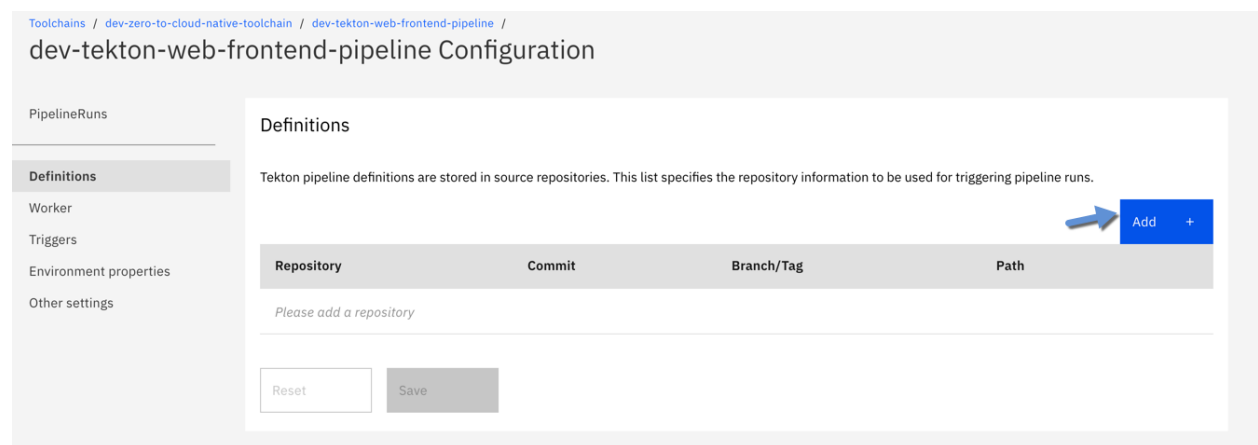


1 – 3 - 1 Pipeline Definitions

Add the tekton pipeline definition by clicking on the Add button, selecting the repository, branch and the path to the tekton pipeline definition files.

Note: Pipeline definition should be in the directory named as **.tekton**. You will find the .tekton folder with all the pipeline definition files already in a .tekton folder in all the repositories.

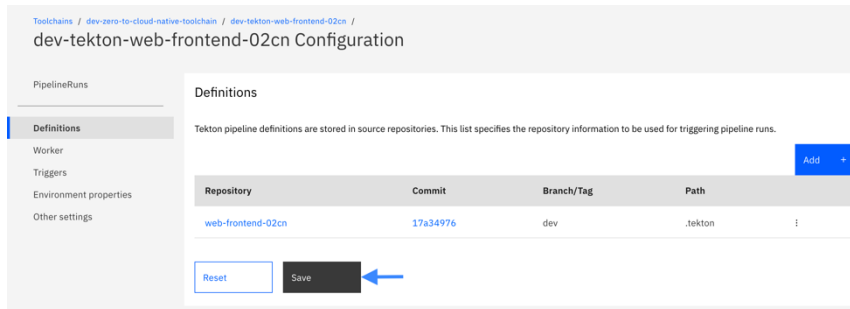
Pipeline definition can be defined as Tekton Resource definitions that create a Tekton PipelineRun. It "runs" a pipeline that builds the application into an image, scans the image for vulnerabilities, and then deploys the application with IBM Cloud Redhat OpenShift Service. Tekton pipeline definitions are stored in source repositories.



On the definitions tab, click on add to add a new definition. The definition will indicate which git repository we want to use along with specifying the branch.



Enter the repository name that you clone in part 8 of this series. In this example, we are using the web-frontend repository and the dev branch. Then click Add.



On the next screen, click Save.

1 – 3 - 2 Tekton Definitions

Before we go through the next steps, let's review the pipeline definitions we will be using.

In our case we have three files (listener.yaml, pipeline.yaml and tasks.yaml) that describe our Tekton pipeline.

- **listener.yaml:** In a nutshell listener.yaml file contains all the configurations needed to trigger a pipeline. Usually a listener.yaml file contains three different Tekton objects known as TriggerTemplate, TriggerBinding and EventListener described in them. (Note: Details about TriggerTemplate, TriggerBinding and EventListener are provided in detail in Section 1-3-4)
- **pipeline.yaml** contains information regarding the task and in which sequence tasks should be executed. It has a Tekton object known as a pipeline which allow us to define what tasks will be executed when the pipeline runs.
- **tasks.yaml** contains information about each task, steps that build that task and their execution. For example in our case task.yaml has 3 tasks, Build-task, validate-task and deploy-task. Build-task is divided into two different steps know as clone-repo and build-docker-image. The clone-repo step pulls the git repository containers, all the application code, and Dockerfile from the SCM and then the build-docker-image step builds the container image and pushes it to IBM Cloud Container Registry.

1 – 3 – 2 Worker Configuration

When it comes to worker configuration, we have two options to either use the Managed worker provided by IBM or private workers i.e. run the pipeline server and runner in your own OpenShift cluster.

Note: Private workers currently are not supported with OpenShift so we will be using IBM Managed Workers.

Click on worker on the left side and select the IBM Managed workers (Tekton Pipelines)

Toolchains / dev-zero-to-cloud-native-toolchain / dev-tekton-web-frontend-pipeline /

dev-tekton-web-frontend-pipeline Configuration

PipelineRuns

Definitions

Worker

Triggers

Environment properties

Other settings

Worker

Tekton pipelines are executed by Workers, which are made available either by adding and configuring a Private Worker tile on your toolchain, or by using the shared pool of IBM Managed Public Workers. If you intend to run your pipelines on a Private Worker, and you have not added one to this toolchain yet, [click here](#).

Logs and status results from pipeline runs are stored in the cloud, so that they will be available even if the worker that ran the pipeline can not be accessed.

[Learn more.](#)

Worker

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

✓ IBM Managed workers (Tekton Pipelines v0.14.1) in DALLAS

Reset

Save

Once the worker is selected click on save button.

1 – 3 – 3 Trigger Configuration

Tekton triggers allow users to create resource templates that get instantiated when an event is received. Additionally, fields from event payloads can be injected into these resource templates as runtime information. This enables users to automatically create templated PipelineRun or TaskRun resources when an event is received.

Tekton Triggers introduce three Kubernetes custom resources to configure triggers:

1. TriggerTemplates

A [TriggerTemplate](#) declares a blueprint for each Kubernetes resource you want to create when an event is received. Each TriggerTemplate has parameters that can be substituted anywhere within the blueprint you define. In general, you will have one TriggerTemplate for each of your Tekton Pipelines. In this tutorial, you create a TriggerTemplate for your build-and-deploy PipelineRun because you want to create a build-and-deploy PipelineRun every time you receive a pull request event.

2. TriggerBindings

A [TriggerBinding](#) describes what information you want to extract from an event to pass to your TriggerTemplate. Each TriggerBinding essentially declares the parameters that get passed to the TriggerTemplate at runtime (when an event is received). In general, you will have one TriggerBinding for each type of event that you receive. In this tutorial, you will create a TriggerBinding for the GitHub pull request event in order to build and deploy the code in the pull request.

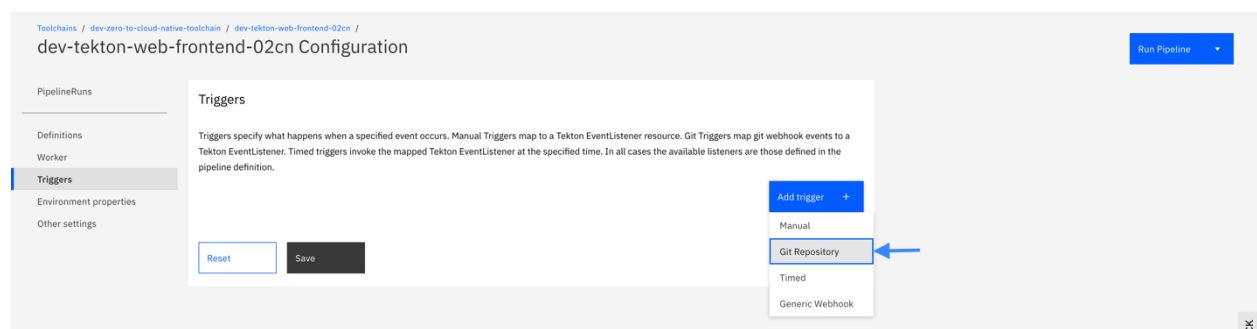
3. EventListeners

An [EventListener](#) creates a Deployment and Service that listen for events. When the EventListener receives an event, it executes a specified TriggerBinding and TriggerTemplate. In this tutorial, the EventListener will receive pull request events from GitHub and execute the TriggerBinding and TriggerTemplate to create a build-and-deploy PipelineRun.

There are multiple options of trigger you can have with tekton pipeline in the toolchains. For example **Manual**, **Git Repository**, **Timed**, and **Generic Webhook**. Depending upon your use case and preference you can use one of them.

In our case we are going to use Git Repository i.e. whenever a commit is pushed, the pipeline runs.

Click on **Triggers**, **Add trigger** button, and then select **Git Repository**.



Next, edit maximum concurrent runs to **1**.

Select the web-frontend-02cn repository.

Keep the dev Branch selected and select **When a commit is pushed**.

The screenshot shows the 'Triggers' configuration page in a web interface. On the left is a sidebar with links: PipelineRuns, Definitions, Worker, Triggers (highlighted), Environment properties, and Other settings. The main area is titled 'Triggers' and contains a description: 'Triggers specify what happens when a specified event occurs. Manual Triggers map to a Tekton EventListener resource. Git Triggers map git webhook events to a Tekton EventListener. Timed triggers invoke the mapped Tekton EventListener at the specified time. In all cases the available listeners are those defined in the pipeline definition.' Below this is a configuration form for a 'Git Trigger - 0'. The form includes: a toggle for 'Enable concurrent runs by this trigger' (checked), a 'Max Concurrent Runs' field set to '1' with a blue arrow pointing to it; a 'Repository' dropdown set to 'web-frontend-02cn (https://github.com/sudoalgorithm/web-frontend-02cn.git)' with a blue arrow pointing to it; a 'Branch' dropdown set to 'dev' with a blue arrow pointing to it; a section 'Run jobs automatically for Git events on the chosen branch' with three radio buttons: 'When a commit is pushed' (checked, with a blue arrow), 'When a pull request is opened or updated', and 'When a pull request is closed'; an 'EventListener' dropdown set to 'listener' with a blue arrow pointing to it; and a 'Worker' dropdown set to 'Inherited from Pipeline Configuration (private-worker)'. At the bottom are 'Reset' and 'Save' buttons, with a blue arrow pointing to the 'Save' button. A vertical sidebar on the right contains 'ASK A QUESTION' and 'FEEDBACK' links.

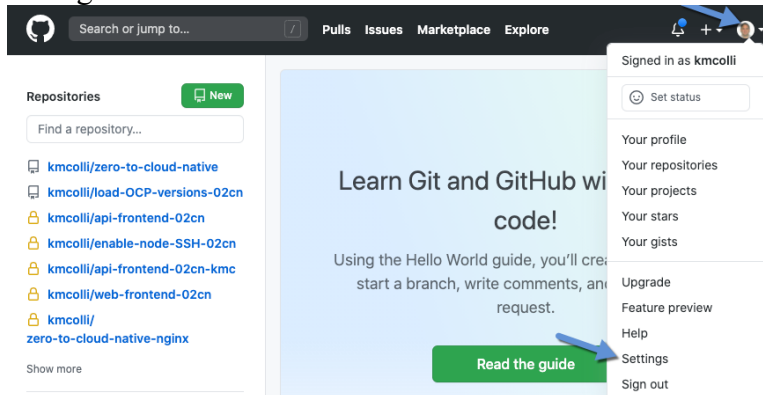
The Event listener and Worker option will be automatically populated with the values as per Tekton definition configured in Pipeline Definitions step and worker configuration step.

After entering all of these details, click on **Save**.

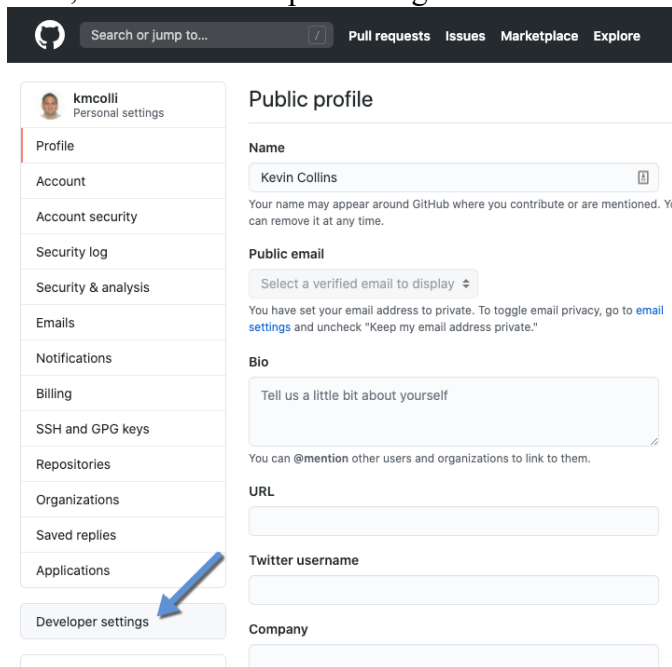
1 – 3 – 4 GitHub Token

If you are following the tutorial and using private GitHub repositories, you will need to get a GitHub token. Best practice is to use a unique GitHub token per project so that you can revoke the token if it becomes compromised without effecting your other repositories.

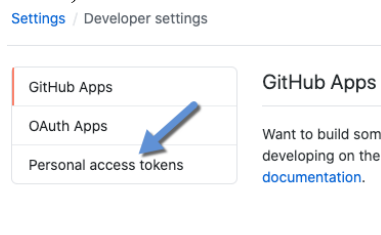
To create a GitHub token, go to GitHub.com. After logging in, click on your ID and then select settings.



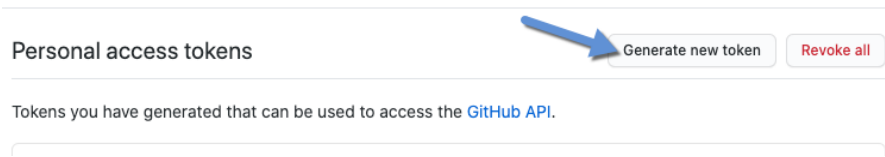
Next, click on Developer Settings.



Next, click on Personal access tokens.



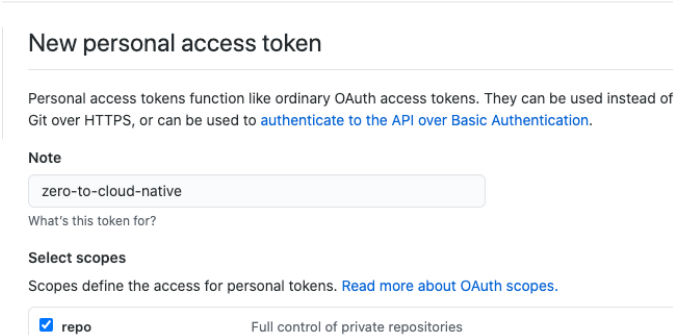
Next, click on Generate new token.



Personal access tokens

Tokens you have generated that can be used to access the [GitHub API](#).

Give your key or token a name, such as zero-to-cloud-native. Since this is a token that I'm not going to share with anyone, I will select every option under select scopes.



New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

zero-to-cloud-native

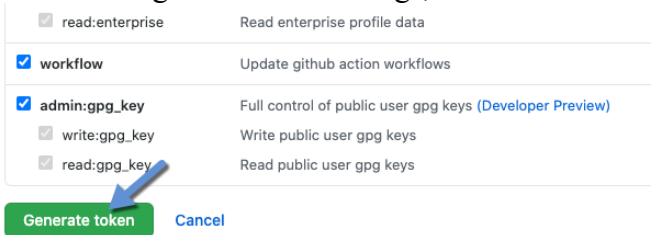
What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

☒ repo Full control of private repositories

After entering all of these settings, click on Generate Token.



☐ read:enterprise Read enterprise profile data

☒ workflow Update github action workflows

☒ admin:gpg_key Full control of public user gpg keys ([Developer Preview](#))

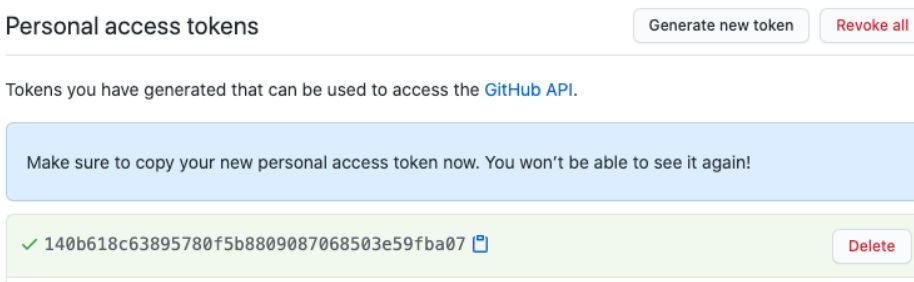
☒ write:gpg_key Write public user gpg keys

☒ read:gpg_key Read public user gpg keys

Generate token Cancel

On the next screen, you will see your token. Store this in a save place, we will be using this token in the next step.

Important: This is your only time to copy the token, after you navigate away from this screen you will not have access to view the token.



Personal access tokens

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your new personal access token now. You won't be able to see it again!

✓ 140b618c63895780f5b8809087068503e59fba07 [Copy](#) Delete

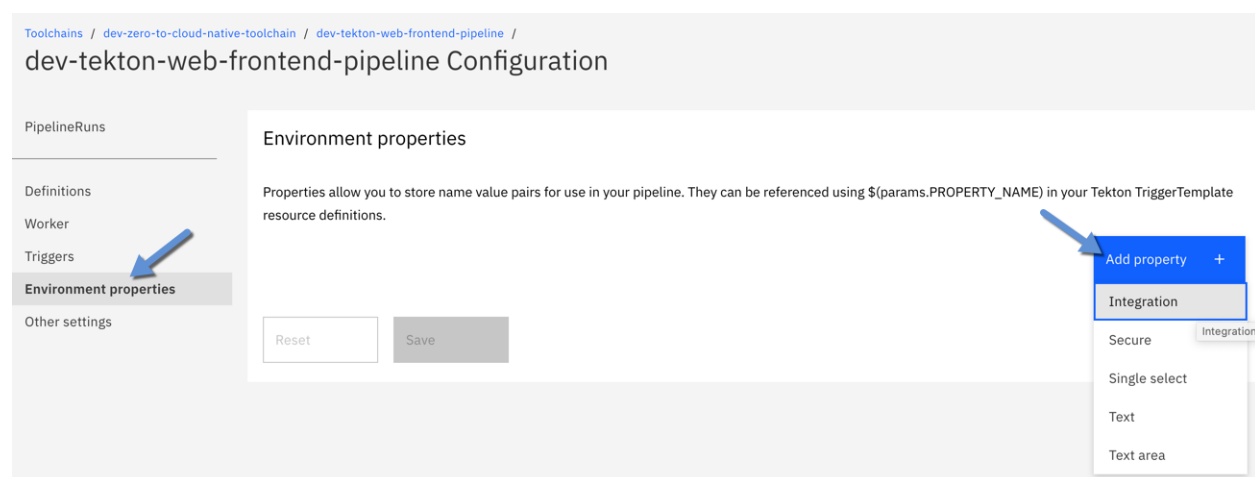
1 – 3 – 5 Environment Properties

If you are following the tutorial and using private github repositories, you will need to get a github token. Best practice is to use a unique github token per project so that you can revoke the token if it becomes compromised without effecting your other repoes.

In order to build and deploy to your own cluster we need to configure some parameter so that we can get access to Container registry and the OpenShift Cluster. In our case we have 10 parameters to define the above access. The 10 parameters are as follows.

- **apikey:** An IBM Cloud API Key.
- **cluster:** The name of your OpenShift cluster where you will be deploying the application. I will be using **zero-to-cloud-native**.
- **clusterNamespace:** The namespace in your cluster where the app will be deployed. I will be using **zero-to-cloud-native**.
- **clusterRegion:** The region where your OpenShift cluster is located – I will be using **us-south**.
- **registryNamespace:** The IBM Cloud Container Registry namespace where the app image will be built and stored. I will be using **zero-to-cloud-native**, make sure to select the namespace you created.
- **registryRegion:** The region where your IBM Cloud Container Registry is located. I will be using **us-south**.
- **repository:** The source git repository where your resources are cloned. Enter the name of your Github Repository that you created in Part 8. The repository must also contain the Git Token you created in the previous step. `https://<token>@github.com/owner/repo.git`
- **revision:** The branch of the source git repository where your resources are cloned. We are simulating a dev branch, so enter **dev**.
- **imageName:** name of the image including the registry. In this example, we will be using: **us.icr.io/zero-to-cloud-native/web-frontend-02cn**. Note, you will need to replace the zero-to-cloud-native with the registry namespace you created.
- **deploymentFile:** location and name of the deployment file. In this case, we will be using `deployments/deployment.yaml` as this is where the deployment file is located in all of our repositories.

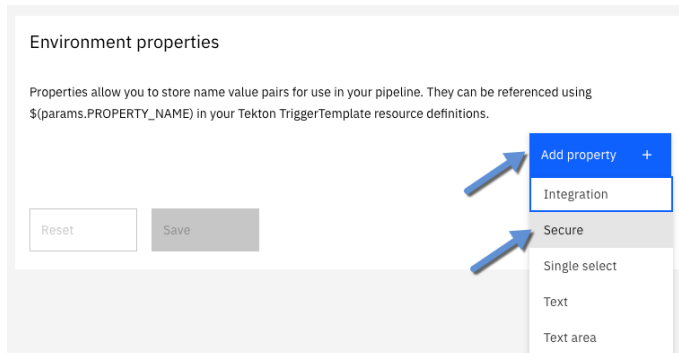
Click on the Environment tab and enter the following parameters by click on **Add property** button.



Note: You will get different options when you click the add property button such **Integration, Secure, Single Select, Text, Text Area**. Functionality of each option is different such **integration** provides

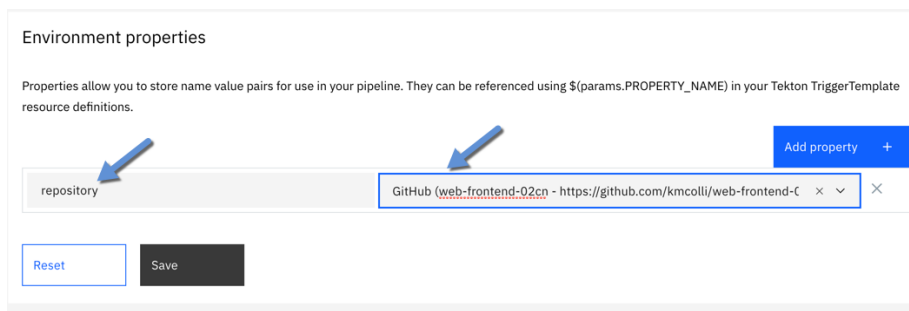
access to values stored in different tools inside toolchains, a **secure** option will mask the entered values, **text area** will provide a text box with 3-4 which can be used to add scripts etc.

The first type we are going to enter is Secure Field. In this case I will be using the **Secure** as the GitHub API will be used in repository specification.

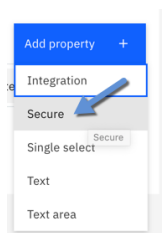


Next, enter repository as the name and then your github repository name. In my example, I will use <https://8d5472cea0cc588f62efed4237119@github.com/kmcolli/web-frontend-02cn.git>

The convention is `https://<github-token>@github.com/<owner>/repo-name`

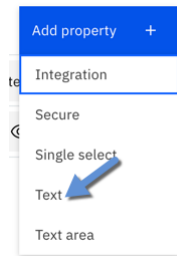


Next, we are going to enter a secured text property to store our IBM Cloud API Key. Click on Add property and then select Secure.



On the next screen, enter apikey as the name and paste you apikey as the secured property.

The remaining properties are all text properties. For each property in the table below, click on Add property, select Text and then enter the property field name and value:



cluster	Name of your cluster. e.g. zero-to-cloud-native
clusterNamespace	Kubernetes Namespace / OpenShift Project where you are deploying the tutorial application. e.g. zero-to-cloud-native
clusterRegion	Region your cluster is in. e.g. us-south
registryNamespace	Namespace of your IBM Cloud Container Registry service where you are storing your images. e.g. zero-to-cloud-native
registryRegion	Region of your IBM Cloud Container Registry. e.g. us-south
revision	Git branch. e.g. dev
imageName	Name of the complete image name such as: web-frontend-02cn
deploymentFile	deployments/deployment.yaml

Note: the imageName can only contain lowercase characters.

Environment properties

Properties allow you to store name value pairs for use in your pipeline. They can be referenced using `$(params.PROPERTY_NAME)` in your Tekton TriggerTemplate resource definitions.

Add property +

apikey	🔑	✕
cluster	zero-to-cloud-native		✕
clusterNamespace	zero-to-cloud-native		✕
clusterRegion	us-south		✕
deploymentFile	deployments/deployment.yaml		✕
imageName	web-frontend-02cn		✕
registryNamespace	zero-to-cloud-native		✕
registryRegion	us-south		✕
repository	🔑	✕
revision	dev		✕

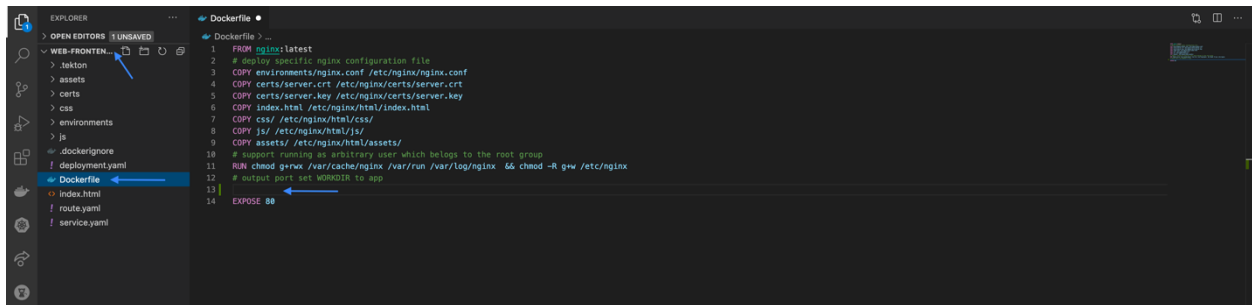
Reset
Save

After entering all of these settings, click on **Save**.

1 – 4 Testing the Tekton Pipeline.

To test our pipeline, all we have to do is make a change to the source code and commit it to GitHub as we have configured our pipeline to automatically start after a GitHub commit to our repository.

To make a change, start Visual Studio Code, and open the workspace you created in step 5. Navigate to the web-frontend-02cn folder and click on the Dockerfile. Simply add a new line anywhere in the file.



Save the changes and push the 'change' to GitHub. You can either use GitHub desktop to push the changes as we did in part 9A. Or you can use the following command line commands.

Open your terminal and follow these commands

- **git add -A**
- **git commit -m "Changes to the dockerfile"** (You will notice the highlighted changes which should only be adding the newline.)
- **git push --set-upstream origin dev**

```
→ web-frontend-02cn git:(dev) ✕ git add -A
→ web-frontend-02cn git:(dev) ✕ git commit -m "Changes to the dockerfile"
[dev df85880] Changes to the dockerfile
 1 file changed, 1 insertion(+)
→ web-frontend-02cn git:(dev) git push --set-upstream origin dev
Enumerating objects: 11, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 12 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 875 bytes | 875.00 KiB/s, done.
Total 7 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/sudoalgorithm/web-frontend-02cn.git
 17a3497..df85880 dev -> dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
→ web-frontend-02cn git:(dev) █
```

Now, navigate back to your browser, click on the pipeline name and watch it run!

Toolchains / dev-zero-to-cloud-native-toolchain / dev-tekton-web-frontend-pipeline /

dev-tekton-web-frontend-pipeline Configuration

PipelineRuns

Definitions

Worker

Triggers

Environment properties

Other settings

Environment properties

Properties allow you to store name value pairs for use in your pipeline. They can be referenced using \$(params.PROPERTY_NAME) in your Tekton TriggerTemplate resource definitions.

Add property +

apikey	✕
cluster	zero-to-cloud-native	✕
clusterNamespace	zero-to-cloud-native	✕
clusterRegion	us-south	✕
registryNamespace	zero-to-cloud-native	✕
registryRegion	us-south	✕
repository	GitHub (web-frontend-02cn - https://github.com/kmcoll/web-frontend-C	✕
revision	dev	✕

Reset Save

After a few minutes, you should see that the pipeline was built successfully. If you click on the pipeline run name, you can view the status of each step.

Toolchains / dev-zero-to-cloud-native-toolchain / dev-tekton-web-frontend-pipeline /

dev-tekton-web-frontend-pipeline Dashboard

Run Pipeline ▾

PipelineRuns

Status: All ▾ Trigger: All ▾

<input type="checkbox"/>	#	Status	Name	Pipeline	Trigger ⓘ	Created	Duration	
<input type="checkbox"/>	2	✓	pipelinerun-fb298181-9fd9-4...	pipeline	Git Trigger - 0	September 24, 2020, 9:00 AM	3 minutes 7 seconds	:

Items per page: 30 ▾ 1 - 1 items

1 ▾ page 1 ◀ ▶

This shows the parameters used, status and logs for each step in the pipeline.

pipelinerun-fb298181-9fd9-4bde-87af-9e0c6162db57 2 minutes ago

Succeeded Tasks Completed: 3 (Failed: 0, Cancelled 0), Skipped: 0

#2 Triggered by kmcoll Git Trigger - 0 [Create index.html](#) [Show Context](#)

✓ pipeline-build-task

✓ clone-repo Completed

✓ pre-build-check Completed

✓ build-docker-image Completed

✓ pipeline-validate-task

✓ pipeline-deploy-task

✓ pipeline-build-task Succeeded

Parameters Status

Name	Value
repository	'****'
revision	dev
apikey	'****'
registryNamespace	zero-to-cloud-native
registryRegion	us-south
imageName	us.icr.io/namespace/image

1 – 5 Next Steps

If this is first time you are going through this section in creating a tekton pipeline, you should now have two of our 6 microservices deployed.

In part 9B we created a classic pipeline for the **api-frontend-02cn** microservice.

In this section, we just created a tekton pipeline for the **web-frontend-02cn** microservice.

To complete the deployment of the tutorial application, you will need to create CI/CD pipelines for the remaining microservices:

enable-node-ssh-02cn

utility-02cn

load-ocp-versions-02cn

ocp-realtime-02cn

Choose your favorite method of using either a classic or Tekton pipeline. After you deploy these microservices, continue to part 9C where we will test the application!