Chapter 3: CI/CD with Tekton Pipelines

This chapter discusses how to integrate complex tasks, such as building and deploying applications, into Kubernetes using Tekton. Continuous integration and continuous development (CI/CD) are represented in Tekton as *pipelines* that combine all the steps you need to accomplish what you want. And Tekton makes it easy to write a general pipeline that you can adapt to many related tasks.

Tekton and OpenShift Pipelines

Tekton is an open source framework to create pipelines for Kubernetes and the cloud. This means that there is no central tool you need to maintain, such as Jenkins. You just have to install a Kubernetes Operator into your Kubernetes cluster to provide some custom resource definitions (CRDs). Based on those CRDs, you can create tasks and pipelines to compile, test, deploy, and maintain your application.

OpenShift Pipelines is based on Tekton and adds a nice GUI to the OpenShift developer console. The Pipelines Operator is free to use for every OpenShift user.

Tekton Concepts

Tekton has numerous objects, but the architecture is quite easy to understand. The key concepts are:

- Step: A process that runs in its own container and can execute whatever the container image provides. A step does not stand on its own, but must be embedded in a task.
- Task: A set of steps running in separate containers (known as "pods" in Kubernetes). A task could be, for example, a compilation process using Maven. One step would be to check the Maven `settings.xml` file. The second step could be to execute the Maven goals (compile, package etc.).
- Pipeline: A set of tasks that are executed either in parallel or (in a simpler case) one after another. A pipeline can be customized through *parameters*.
- PipelineRun: A collection of parameters to submit to a pipeline. For instance, a build-and-deploy pipeline might refer to a PipelineRun that contains technical input (for example, a `ConfigMap` and `PersistentVolumeClaim`) as well as non-technical parameters (for example, the URL for the Git repository to clone, the name of the target image, etc.)

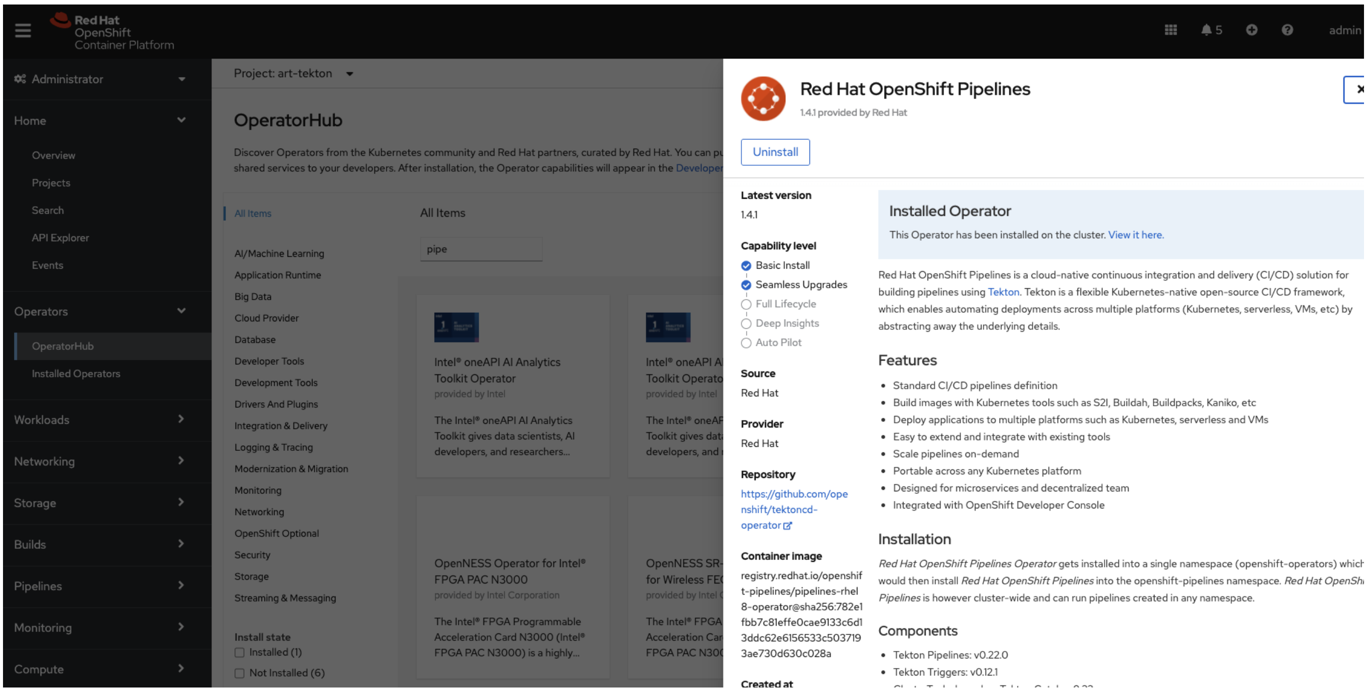
Internally, Tekton creates a TaskRun object for each task it finds in a PipelineRun.

To summarize: A pipeline contains a list of tasks, each of which contain a list of steps. One of the benefits of Tekton is that tasks and pipelines can be shared with other people, because a pipeline just specifies what to do in a given order. So if most of your projects have a similar pipeline, share and reuse it.

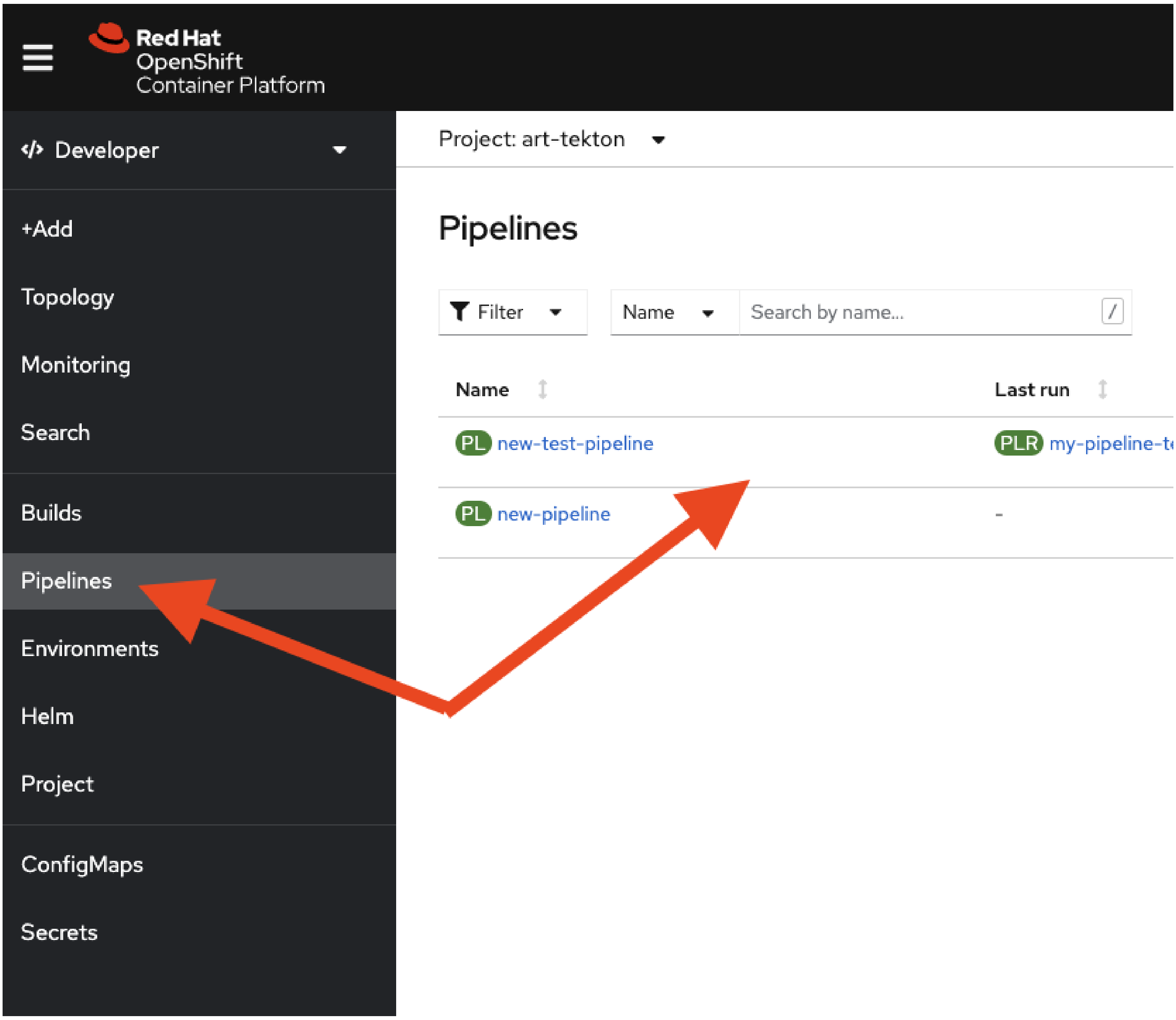
Installing OpenShift Pipelines on Red Hat OpenShift

The process in this chapter requires version 1.4.1 or higher of the OpenShift Pipelines Operator. To install that version, you also need a recent 4.7 OpenShift cluster, which you could install for example via CodeReady Containers. Without these tools, you won't have access to workspaces (which you need to define).

To install OpenShift Pipelines, you must be cluster-admin. Go to the OperatorHub, search for "pipelines," and click the **Install** button. There is nothing more to do for now, as the Operator maintains everything for you (Figure 1).



After a while you'll notice a new GUI entry in both the Administrator and the Developer UI (Figure 2).



This chapter's example: Create a pipeline for quarkus-simple

For our person-service, we are going to create a Tekton pipeline for a simple deployment task. The pipeline compiles the source, creates a Docker image based on Jib, pushes the image to Quay.io, and uses kustomize to apply that image to an OpenShift project called `book-tekton`.

Sounds easy?

It is. Well, mostly.

First of all, why are we going to use Jib for building the container image? Well, that’s easily explained: Right now, there are three different container image build strategies available with Quarkus:

- Docker
- Source-to-Image (S2I)
- Jib The Docker strategy uses the `docker` binary to build the container image. But the `docker` binary is not available inside a Kubernetes cluster (as mentioned in Chapter 3), because Docker is too heavyweight and requires root privileges to run the daemon.

S2I requires creating `BuildConfig,`DeploymentConfig`, and`ImageStream` objects specific to OpenShift, but these are not available in vanilla Kubernetes clusters.

So in order to stay vendor-independent, we have to use Jib for this use case.

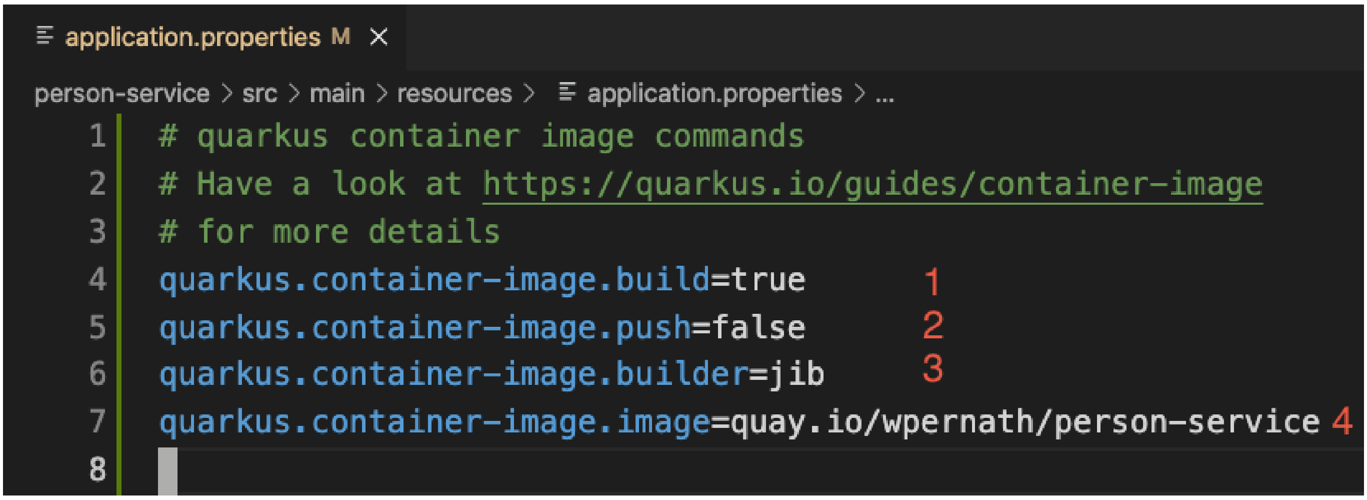
Of course, you could also use other tools to create your container image inside Kubernetes. But in order to keep this Tekton example clean and simple, we are reusing what Quarkus provides. So we are able to simply set a few Quarkus properties in `application.properties` to define how Quarkus should package the application. Then we'll be able to use exactly *one* Tekton task to compile, package, and push the application to an external registry.

NOTE: Make sure that your Quarkus application is using the required Quarkus extension called `container-image-jib`. If your `pom.xml` file does not include the `quarkus-container-image-jib` dependency, add it by executing:

```
$ cd ~/GitOps-Workshop/artifacts/person-service/
$ mvn quarkus:add-extension -Dextensions="container-image-jib"
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< org.wanja.book:person-service >---------------
-----
[INFO] Building person-service 1.0.0
[INFO] --------------------------------[ jar ]-----------------------------
-----
[INFO]
[INFO] --- quarkus-maven-plugin:2.4.2.Final:add-extension (default-cli) @
person-service ---
[INFO] Looking for the newly published extensions in registry.quarkus.io
[INFO] [SUCCESS] ✅  Extension io.quarkus:quarkus-container-image-jib has
been installed
[INFO] --------------------------------------------------------------------
-----
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.817 s
[INFO] Finished at: 2021-11-22T09:34:05+01:00
[INFO] ------------------------------------------------------------------------
```

Then have a look at Figure 3 to see what properties need to be set to let Quarkus build, package, and push the image. Basically, the following properties need to be set in `application.properties`:



1. `quarkus.container-image.build`: Set this to `true` to ensure that a `mvn package` command builds a container image.
2. `quarkus.container-image.push`: This is optional and required only if you want to push the image directly to the registry. I don't intend to do so, so I set the value to `false`.
3. `quarkus.container-image.builder`: This property selects the method of building the container image. We set the value to `jib` to use Jib.
4. `quarkus.container-image.image`: Set this to the complete name of the image to be built, including the domain name.

Now check out the source code, have a look at `~/GitOps-Workshop/artifacts/person-service/src/main/resources/application.properties`, change the image property to meet your needs, and issue:

```
$ mvn clean package -DskipTests
```

This command compiles the sources and builds the container image. If you want to push the resulting image to your registry, simply call:

```
$ mvn package -DskipTests -Dquarkus.container-image.push=true
```

After a while, Quarkus will generate and push your image to your registry. In my case, it's `quay.io/wpernath/person-service`.

Inventory check: What do we need?

To create our use case, you need the following tools:

- `git`: To fetch the source from GitHub.
- `maven`: To reuse most of what Quarkus provides.
- `kustomize`: To change our Deployment to point to the new image.
- OpenShift client: To apply the changes we've made in the previous steps.

Some of them can be set up for you by OpenShift. So now log into your OpenShift cluster, create a new project, and list all the available ClusterTasks:

```
$ oc login .....
$ oc new-project book-tekton
$ tkn ct list
```

> **Note**: What is the difference between a task and a ClusterTask? A ClusterTask is available globally in all projects, whereas a task is available only locally per project and must be installed into each project where you want to use it.

Figure 4 shows all the available ClusterTasks created after you install the OpenShift Pipelines Operator. It seems you have most of what we need:

- `git-clone`
- `maven`
- `openshift-client`

```
NAME                       DESCRIPTION              AGE
buildah                    Buildah task builds...   1 week ago
buildah-v0-22-0            Buildah task builds...   1 week ago
git-cli                    This task can be us...   1 week ago
git-clone                  These Tasks are Git...   1 week ago
git-clone-v0-22-0          These Tasks are Git...   1 week ago
helm-upgrade-from-repo     These tasks will in...   1 week ago
helm-upgrade-from-source   These tasks will in...   1 week ago
jib-maven                  This Task builds Ja...   1 week ago
kn                         This Task performs ...   1 week ago
kn-apply                   This task deploys a...   1 week ago
kn-apply-v0-22-0           This task deploys a...   1 week ago
kn-v0-22-0                 This Task performs ...   1 week ago
kubeconfig-creator         This Task do a simi...   1 week ago
maven                      This Task can be us...   1 week ago
openshift-client           This task runs comm...   1 week ago
openshift-client-v0-22-0   This task runs comm...   1 week ago
pull-request               This Task allows a ...   1 week ago
s2i-dotnet                 s2i-dotnet task fet...   1 week ago
s2i-dotnet-v0-22-0         s2i-dotnet task fet...   1 week ago
s2i-go                     s2i-go task clones ...   1 week ago
s2i-go-v0-22-0             s2i-go task clones ...   1 week ago
s2i-java                   s2i-java task clone...   1 week ago
s2i-java-v0-22-0           s2i-java task clone...   1 week ago
s2i-nodejs                 s2i-nodejs task clo...   1 week ago
s2i-nodejs-v0-22-0         s2i-nodejs task clo...   1 week ago
s2i-perl                   s2i-perl task clone...   1 week ago
s2i-perl-v0-22-0           s2i-perl task clone...   1 week ago
s2i-php                    s2i-php task clones...   1 week ago
s2i-php-v0-22-0            s2i-php task clones...   1 week ago
s2i-python                 s2i-python task clo...   1 week ago
s2i-python-v0-22-0         s2i-python task clo...   1 week ago
s2i-ruby                   s2i-ruby task clone...   1 week ago
s2i-ruby-v0-22-0           s2i-ruby task clone...   1 week ago
skopeo-copy                Skopeo is a command...   1 week ago
skopeo-copy-v0-22-0        Skopeo is a command...   1 week ago
tkn                        This task performs ...   1 week ago
trigger-jenkins-job        The following task ...   1 week ago
```

You're missing just the `kustomize` task. You'll create one later, but first we want to take care of the rest of the tasks.

Analyzing the necessary tasks

If you want to have a look at the structure of a task, you can easily do so by executing the following command:

```
$ tkn ct describe <task-name>
```

The output explains all the parameters of the task, together with other necessary information such as its inputs and outputs.

By specifying the `-o yaml` parameter, you can view the YAML source definition of the task.

The `git-clone` task allows a large number of parameters, but most of them are optional. You just have to specify `git-url` and `git-revision`. And you have to specify a workspace for the task.

What are workspaces?

Remember that Tekton is running each and every task (and all steps inside a task) as a separate pod. If the application running on the pod writes to some random folder, nothing gets really stored. So if we want (and yes, we do want) one step of the pipeline to read and write data that is shared with other steps, we have to find a way to do that.

This is what workspaces are for. They could be a persistent volume claim, a config map, etc. A task that either requires a place to store data (such as git-clone) or needs to have access to data coming from a previous step (such as Maven), defines a workspace. If the task is embedded into a pipeline, the workspace is defined for every task in the pipeline. The PipelineRun (or in case of a single running task, the TaskRun) finally creates the mapping between the defined workspace and a corresponding storage.

In our example, we need two workspaces:

- A PersistentVolumeClaim (PVC) where the git-clone task is cloning the source code to and from the place where the Maven task is compiling the source
- A ConfigMap with the `maven-settings` file you need in your environment

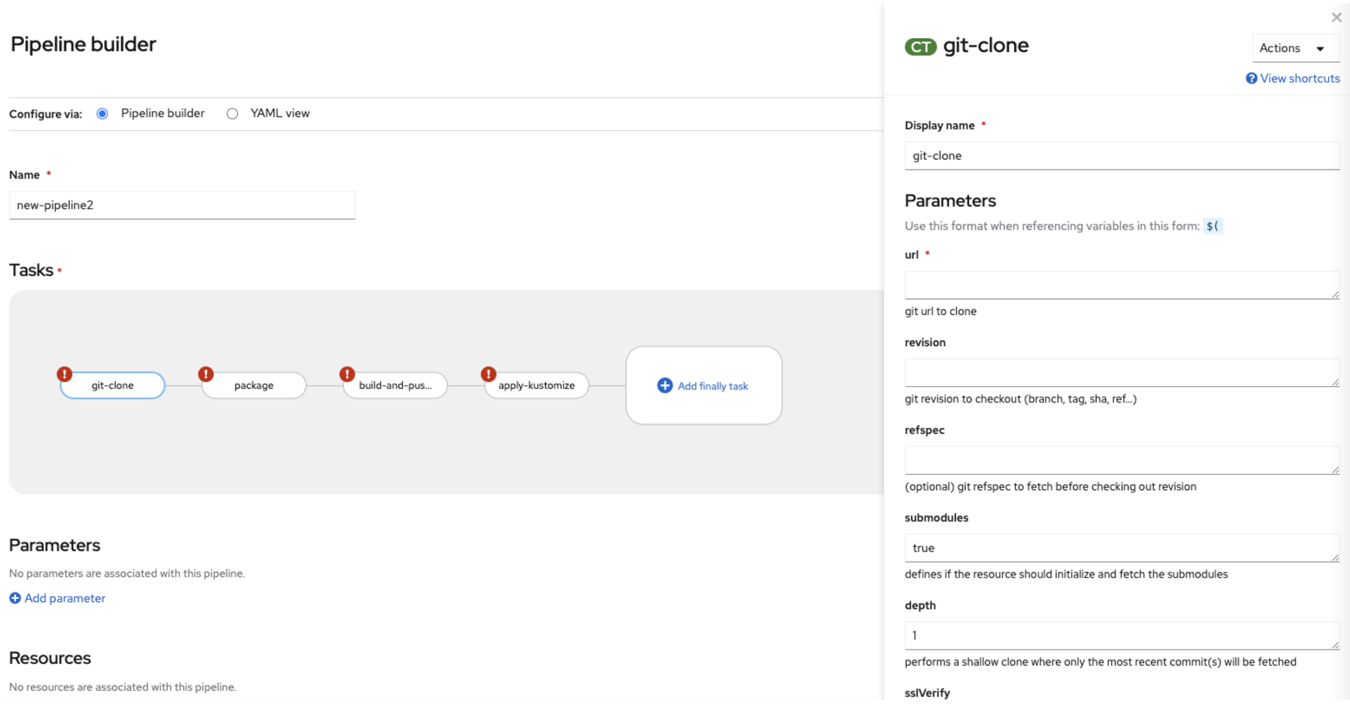
Ways of building the pipeline

Once you know what tasks you need in order to build your pipeline, you can start creating it. There are two ways of doing so:

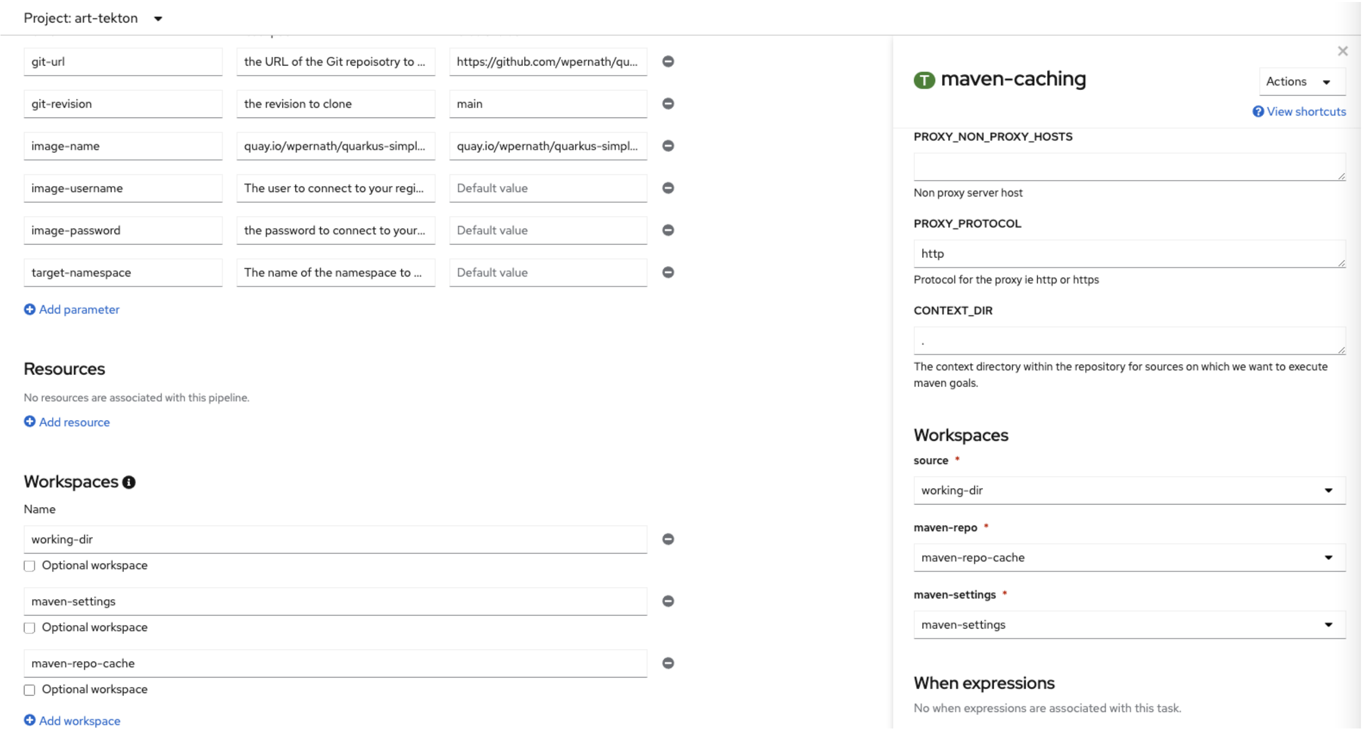
- Build your pipeline via a code editor as a YAML file.
- Build your pipeline in the OpenShift Developer Console.

As a first try, I recommend building the pipeline via the graphical Developer Console of OpenShift (Figure 5). Then export and see it what it looks like. The rest of this section focuses on that activity.

> **Note**: Remember that you should have at least version 1.4.1 of the OpenShift Pipelines Operator installed.



You have to provide parameters to each task, and link the required workspaces to the tasks. You can easily do that by using the GUI (Figure 6).



You need to use the `maven` task twice, using the `package` goal:

1. To simply compile the source code.
2. To execute the `package` goal with the following parameters that instruct quarkus to build and push the image:
    - `-Dquarkus.container-image.push=true`
    - `-Dquarkus.container-image.builder=jib`
    - `-Dquarkus.container-image.image=$(params.image-name)`
    - `-Dquarkus.container-image.username=$(params.image-username)`
    - `-Dquarkus.container-image.password=$(params.image-password)`

Once you've done all that and have clicked on the **Save** button, you're able to export the YAML file by executing:

```
$ oc get pipeline/build-and-push-image -o yaml > tekton/pipelines/build-and-push-image.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-push-image
spec:
  params:
  - default: https://github.com/wpernath/book-example.git
    description: the URL of the Git repoisotry to build
    name: git-url
    type: string
....
```

You can easily re-import the pipeline file by executing:

```
$ oc apply -f tekton/pipelines/build-and-push-image.yaml
```

Placement of task parameters

One of the goals of Tekton has always been to provide tasks and pipelines that are as reusable as possible. This means making each task as general-purpose as possible.

If you're providing the necessary parameters directly to each task, you might repeat the settings over and over again. For example, in our case, we are using the Maven task for compiling, packaging, image generation, and pushing. In this case it makes sense to take the parameters out of the specification of each task. Instead, put them on the pipeline level under a property called `params` (as shown in the following listing) and refer to them inside the corresponding task by specifying them by their name in the syntax `$(params.parameter-name)`.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-push-image
spec:
  params:
    - default: 'https://github.com/wpernath/book-example.git'
      description: Source to the GIT
      name: git-url
      type: string
    - default: main
      description: revision to be used
      name: git-revision
      type: string
[...]
  tasks:
    - name: git-clone
      params:
        - name: url
          value: $(params.git-url)
        - name: revision
          value: $(params.git-revision)
[...]
      taskRef:
        kind: ClusterTask
        name: git-clone
      workspaces:
        - name: output
          workspace: shared-workspace
[...]
```

Creating a new task: kustomize

Remember that our default OpenShift Pipelines Operator installation didn't include Kustomize. Because we want to use it to apply the new image to our Deployment, we have to look for a proper task in Tekton Hub. Unfortunately, there doesn’t seem to be one available, so we have to create our own.

For this, we first need to have a proper image that contains the `kustomize` executable. The `Dockerfile` for this project is available in the kustomize-ubi repository on GitHub and the image is available in its repository on Quay.io.

Now let’s create a new Tekton task:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: kustomize
  labels:
    app.kubernetes.io/version: "0.4"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: build-tool
spec:
  description: >-
    This task can be used to execute kustomze build scripts and to apply
the changes via oc apply -f
  workspaces:
    - name: source
      description: The workspace holding the cloned and compiled quarkus
source.
  params:
    - name: kustomize-dir
      description: Where should kustomize look for kustomization in
source?
    - name: target-namespace
      description: Where to apply the kustomization to
    - name: image-name
      description: Which image to use. Kustomize is taking care of it
  steps:
    - name: build
      image: quay.io/wpernath/kustomize-ubi:latest
      workingDir: $(workspaces.source.path)
      script: |

        cd $(workspaces.source.path)/$(params.kustomize-dir)

        DIGEST=$(cat $(workspaces.source.path)/target/jib-image.digest)

        kustomize edit set image quay.io/wpernath/simple-
quarkus:latest=$(params.image-name)@$DIGEST

        kustomize build $(workspaces.source.path)/$(params.kustomize-dir)
> $(workspaces.source.path)/target/kustomized.yaml

    - name: apply
```

```
      image: 'image-registry.openshift-image-
registry.svc:5000/openshift/cli:latest'
      workingDir: $(workspaces.source.path)
      script: |
        oc apply -f $(workspaces.source.path)/target/kustomized.yaml -n
$(params.target-namespace)
```

Paste this text into a new file called `kustomize-task.yaml`. As you can see from the contents of the file, this task requires a workspace called `source` and three parameters: `kustomize-dir`, `target-namespace`, and `image-name`. The task contains two steps: `build` and `apply`.

The build step uses the Kustomize image to set the new image and digest. The apply step uses the internal OpenShift CLI image to apply the Kustomize-created files in the `target-namespace` namespace.

To load the `kustomize-task.yaml` file into your current OpenShift project, simply execute:

```
$ oc apply -f kustomize-task.yaml
task.tekton.dev/kustomize configured
```

Putting it all together

We have now created a pipeline that contains four tasks: `git-clone`, `package`, `build-and-push-image`, and `apply-kustomize`. We have provided the necessary parameters to each task and to the pipeline and we have connected workspaces to it.

Now we have to create the PersistentVolumeClaim (PVC) and a ConfigMap named `maven-settings`, which will then be used by the corresponding PipelineRun.

Creating a maven-settings ConfigMap

If you have a working `maven-settings` file, you can easily reuse it with the Maven task. Simply create it via:

```
$ oc create cm maven-settings --from-file=/your-maven-settings --dry-
run=client -o yaml > maven-settings-cm.yaml
```

If you need to edit the ConfigMap, feel free to do it right now and then execute to import the ConfigMap into your current project:

```
$ oc apply -f maven-settings-cm.yaml
```

Creating a PersistentVolumeClaim

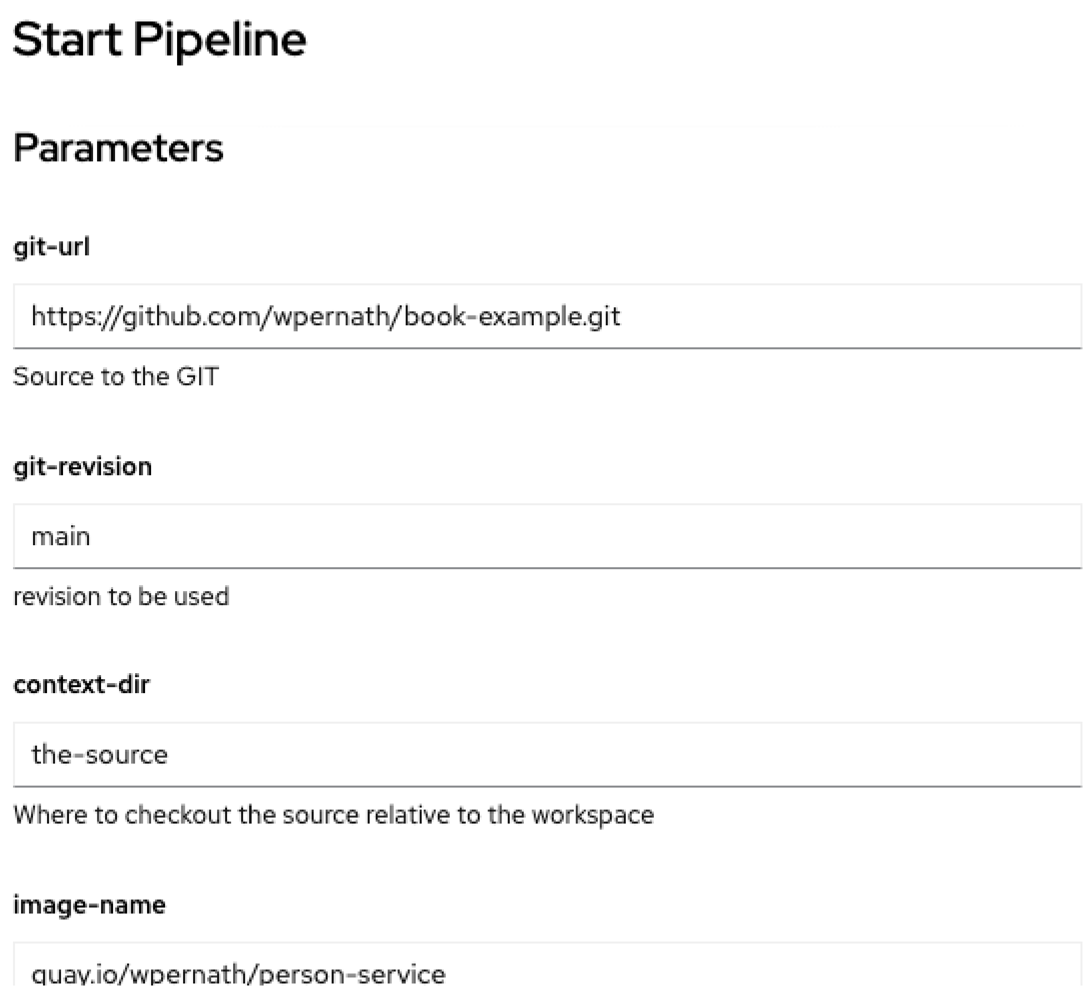
Create a new file with the following content and execute `oc apply -f` to import it into your project:

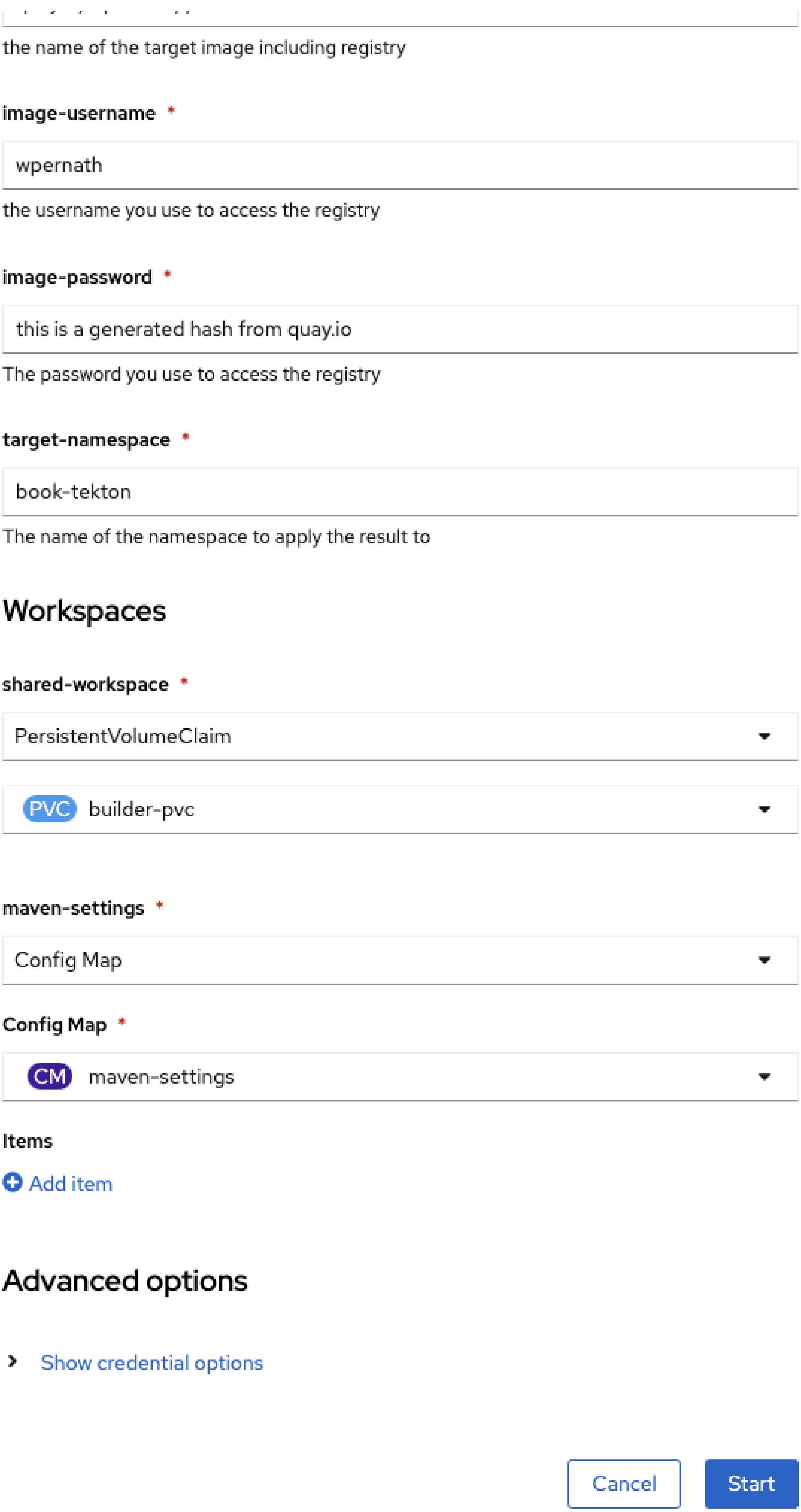
```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: builder-pvc
spec:
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
```

This file reserves a PVC with the name `builder-pvc` and a requested storage of 10GB. It's important to use `persistentVolumeReclaimPolicy: Retain` here, as we want to reuse build artifacts from the previous builds. More on this requirement later in this chapter.

Running the pipeline

Once you have imported all your artifacts into your current project, you can run the pipeline. To do so, click on the **Pipelines** entry on the left side of the Developer Perspective of OpenShift, choose your created pipeline, and select **Start** from the **Actions** menu on the right side. After you've filled in all necessary parameters (Figure 7), you're able to start the PipelineRun.




the name of the target image including registry
image-username *
wpernath
the username you use to access the registry
image-password *
this is a generated hash from quay.io
The password you use to access the registry
target-namespace *
book-tekton
The name of the namespace to apply the result to
Workspaces
shared-workspace *
PersistentVolumeClaim
PVC builder-pvc
maven-settings *
Config Map
Config Map *
CM maven-settings
Items
Add item
Advanced options
Show credential options
Cancel
Start

The **Logs** and **Events** cards of the OpenShift Pipeline Editor show, well, all the logs and events. If you prefer to view these things from the command line, use `tkn` to follow the logs of the PipelineRun:

```
$ tkn pr
```

The output shows the available actions for PipelineRuns.

To list each PipelineRun and its status, enter:

```
$ tkn pr list
NAME                                        STARTED         DURATION
STATUS
build-and-push-image-run-20211123-091039   1 minute ago    54 seconds
Succeeded
build-and-push-image-run-20211122-200911   13 hours ago    2 minutes
Succeeded
build-and-push-image-ru0vni                13 hours ago    8 minutes
Failed
```

To follow the logs of the last run, execute:

```
$ tkn pr logs -f -L
```

If you omit the `-L` option, `tkn` lets you choose from the list of PipelineRuns.

You can also log, list, cancel, and delete PipelineRuns.

Visual Code has a Tekton Pipeline extension that you can also use to edit, build, and execute pipelines.

Creating a PipelineRun object

In order to start the pipeline via a shell (or from any other application you're using for CI/CD), you need to create a PipelineRun object, which looks like the following:

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: $PIPELINE-run-$(date "+%Y%m%d-%H%M%S")
spec:
  params:
    - name: git-url
      value: https://github.com/wpernath/book-example.git
    - name: git-revision
      value: main
    - name: context-dir
      value: the-source
```

```
      - name: image-name
        value: quay.io/wpernath/person-service
      - name: image-username
        value: wpernath
      - name: image-password
        value: *****
      - name: target-namespace
        value: book-tekton
    workspaces:
      - name: shared-workspace
        persistentVolumeClaim:
          claimName: builder-pvc
      - configMap:
          name: maven-settings
        name: maven-settings
    pipelineRef:
      name: build-and-push-image
    serviceAccountName: pipeline
```

Most of the properties of this object are self-explanatory. Just one word on the `serviceAccountName` property: Each PipelineRun runs under a given service account, which means that all pods started along the pipeline run inside this security context.

OpenShift Pipelines creates a default service account for you called `pipeline`. If you have secrets that you want to make available to your PipelineRun, you have to connect them with the service account name. But this requirement is out of scope for this chapter of the book; we'll return to secrets in the next chapter.

The `tekton/pipeline.sh` shell script creates a full version of this PipelineRun based on input parameters.

Optimizing the pipeline

As earlier output from the logs showed, the first pipeline run takes quite a long time to finish: In my case, approximately 8 minutes. The second pipeline still took 2 minutes. I was running the pipelines on a home server, which has modest resources. When you use Tekton on your build farms, run times should be much lower because you're running on dedicated server hardware.

But still, the pipelines at this point take way too long.

If you're looking at the logs, you can see that the `maven` task is taking a long time to finish. This is because Maven is downloading the necessary artifacts again and again on every run. Depending on your Internet connection, this takes some time, even if you're using a local Maven mirror.

On your developer machine, Maven uses the `$HOME/.m2` folder as a cache for the artifacts. The same will be done when you're running Maven from a task. However, because each PipelineRun runs on a separate set of pods, `$HOME/.m2` is not properly defined, which means the whole cache gets invalidated once the PipelineRun is finished.

Maven allows you to specify `-Dmaven.repo.local` to provide a different path to a local cache. This option is what you can use to solve the problem.

I have created a new Maven task (`maven-caching`), which you can find in the book's example repository. The file was originally just a copy of the one that came from Tekton Hub. But then I decided to remove the init step, which was building a `maven-settings.xml` file based on some input parameters. Instead, I removed most of the parameters and added a ConfigMap with must-have `maven-settings`. I believe this makes everything much easier.

As Figure 8 shows, you now have only two parameters: `GOALS` and `CONTEXT_DIR`.

maven-task.yaml

```yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: maven-caching
  labels:
    app.kubernetes.io/version: "0.4"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: build-tool
spec:
  description: >-
    This Task can be used to run a Maven build. The only difference to
    the original one from github.com/tektoncd/catalog is that it uses an
    optional local maven repo path to cache all the dependencies. And it requires a
    a maven-settings workspace.
  workspaces:
    - name: source
      description: The workspace consisting of maven project.
      optional: false
    - name: maven-settings
      description: >-
        The workspace consisting of the custom maven settings
        provided by the user.
      optional: false
  params:
    - name: GOALS
      description: maven goals to run
      type: array
      default:
        - "package"
    - name: CONTEXT_DIR
      type: string
      description: >-
        The context directory within the repository for sources on
        which we want to execute maven goals.
      default: "."
  steps:
    - name: mvn-goals
      image: gcr.io/cloud-builders/mvn@sha256:57523fc43394d6d9d2414ee8d1c85ed7a13460cbb268c3cd16d28cfb3859e641
      workingDir: $(workspaces.source.path)/$(params.CONTEXT_DIR)
      command: ["/usr/bin/mvn"]
      args:
        - -B
        - -s
        - $(workspaces.maven-settings.path)/settings.xml
        - -Dmaven.repo.local=$(workspaces.source.path)/MAVEN_MIRROR
        - "$(params.GOALS)"
```

The important properties for the `maven` call is shown in the second red box of Figure 8. These properties call `maven` with the `maven-settings` property and with the parameter indicating where to store the downloaded artifacts.

One note on artifact storage: During my tests of this example, I realized that if the `git-clone` task clones the source to the root directory of the PVC (when no `subdirectory` parameter is given on task execution), the next start of the pipeline will delete everything from the PVC again. And in that case, we once again have no artifact cache.

So you have to provide a `subdirectory` parameter (in my case, I used a global property called `the-source`) and provide exactly the same value to the `CONTEXT_DIR` parameter in the `maven` calls.

The changes discussed in this section reduce our maven calls dramatically, in my case from 8 minutes to 54 seconds:

```
$ tkn pr list
NAME                         STARTED         DURATION     STATUS
build-and-push-image-123     1 minute ago    54 seconds   Succeeded
build-and-push-image-ru0     13 hours ago    8 minutes    Succeeded
```

Summary of using Tekton pipelines

Tekton is a powerful tool for creating CI/CD pipelines. Because it is based on Kubernetes, it uses extensive concepts from Kubernetes and reduces the maintenance of the tool itself. If you want to start your first pipeline quickly, try to use the OpenShift Developer UI, which you get for free if you're installing the Operator. This gives you a nice base to start your tests. However, at some point—especially when it comes to optimizations—you need a proper editor to code your pipelines.

One of the biggest advantages of Tekton over other CI/CD tools such as Jenkins is that you can reuse all your work for other projects and applications. If you want to standardize the way your pipelines work, build one pipeline and simply specify different sets of parameters for different situations. PipelineRun objects make this possible. The pipeline we have just created in this chapter can easily be reused for all Quarkus-generated applications. Just change the `git-url` and `image-name` parameters. Isn't this great?

And even if you're not satisfied with all the tasks you get from Tekton Hub, use them as bases and build your own iterations out of them, as we did with the optimized Maven task and the Kustomize task in this chapter.

I would not say that Tekton is the easiest technology available to do CI/CD pipelines, but it is definitely one of the most flexible.

However, we have not even talked about Tekton security and how we are able to provide, for example, secrets to access your Git repository or the image repository. And we have cheated a little bit about image generation, because we were using the mechanism Quarkus provides. There are other ways of creating images using a dedicated Buildah task.

The next chapter of this book discusses Tekton security, as well as GitOps and Argo CD.