

Creating a Remote Development Environment

This lesson covers the steps for creating a devpod and configuring it for our application.

We'll cover the following



- Browse to go-demo-6 directory
- Creating a DevPod
- Exploring the new development environment
 - Inspecting the project files
- Confirming the installations
 - Compiling the binary
 - Inspecting skaffold.yaml
 - Explanation of skaffold.yaml
 - The build section
 - The deploy section
 - The DIGEST_HEX variable
 - The deploy section of the dev profile
 - The DOCKER_REGISTRY variable
- Checking the env variables
- Initializing the Helm client
 - Generating UUID and running skaffold using the dev profile
 - Inspecting the output
 - Confirming the installation of application
- Inspecting watch.sh
 - Adding the missing UUID variable
- Exiting the DevPod
- Running the application

Let's say that we want to change the *go-demo-6* code.

Once we finish making changes to the code, we'll need to compile it and test it, and

Since we make making changes to the code, we'll need to compile it and test it, and that might require additional installations and configurations. Once our code is compiled, we'll need to run the application to confirm that it behaves as expected.

- For that, we need Docker to build a container image, and we need a **Helm** to deploy it.
- We'd also need to create a personal Namespace in our cluster, or create a local one.
- Given that we adopted **Skaffold** as a tool that builds images and deploys them, we'd need to install that as well. But that's not the end of our troubles.
- Before we deploy our application, we'll need to push the new container image and the **Helm** chart, to the registries running inside our cluster. To do that, we need to know their addresses, and we need credentials with sufficient permissions.

Even when we do all that, we need a repeatable process that will build, test, release, and deploy our work whenever we make a significant change. Better yet, whenever we make any change.



Here's the task for you.

Change anything in the code, compile it, build an image, release it together with the **Helm** chart, deploy it inside the cluster, and confirm that the result is what you expect it to be. I urge you to stop reading and do all those things.

Come back when you're done.

How much did it take you to get a new feature up-and-running? Chances are that you failed. Maybe you used Jenkins X and the knowledge from the previous chapters to do all that. If that's the case, was it efficient? I bet it took more than a few minutes to set up the whole environment. Now, make another change to the code and don't touch anything else. Did your change compile? Were your tests executed? Was your change rolled out? Did that take more than a few seconds?

If you were not successful, we'll explore how to make you succeed the next time. If you didn't, we'll explore how to make the processes much easier, more efficient, and faster. We'll use Jenkins X, but not in the way you expect.

Browse to `go demo 6 directory #`

Browse to go-demo-6 directory

Please enter into the *go-demo-6* directory, if you're not already there.

```
cd go-demo-6
```



Creating a DevPod

We'll create a whole development environment that will be custom tailored for the *go-demo-6* project. It will be the environment for a single user and it will run inside our cluster. We'll do that through a single command.

```
jx create devpod --label go --batch-mode
```



Right now, a Jenkins X DevPod is being created in the batch mode (no questions asked), and we can deduce what's happening from the output.

First, Jenkins X created the `jx-edit-YOUR_USER` Namespace in your cluster. That'll be your personal piece of the cluster where we'll deploy your changes to *go-demo-6*. You'll see instructions on how to access Web IDE as well as the DevPod itself.

Exploring the new development environment

Many things happened in the background that we'll explore in due time. For now, assuming that the process has finished, we'll enter the Pod and explore the newly created development environment.

```
jx rsh --devpod
```



The `jx rsh` command opens a terminal inside a Pod. The `--devpod` argument indicated that we want to connect to the DevPod we just created.

Inspecting the project files

Before we proceed, we'll confirm that the *go-demo-6* code was indeed cloned inside the Pod.

```
cd go-demo-6
```

```
ls -l
```



The output is as follows.

```
charts
Dockerfile
functional_test.go
go.mod
jenkins-x.yml
main.go
main_test.go
Makefile
OWNERS
OWNERS_ALIASES
production_test.go
README.md
skaffold.yaml
vendor
watch.sh
```

Since we created the DevPod while inside our local copy of the *go-demo-6* repository, Jenkins X knew that we wanted to work with that code, so it cloned it for us.

Confirming the installations

Next, we should check whether our development environment indeed contains everything we need. *Was Jenkins X intelligent enough to figure out needs just by knowing the repository we're using to develop our application?* We can (partly) check that by trying to compile it.

Compiling the binary

```
make linux
```

We executed `make linux` to compile the binary. It was a success and we proved that, as a minimum, our new environment contains the Go compiler.

Inspecting `skaffold.yaml`

Jenkins X pipelines use `skaffold` to create container images and deploy our applications. We won't go into all the details behind `skaffold` just yet, but only through the parts that matter for our current goals. So, let's take a quick look at `skaffold.yaml`.

```
cat skaffold.yaml
```

The output is as follows

The output is as follows.

```
apiVersion: skaffold/v1beta2
kind: Config
build:
  artifacts:
    - image: vfarcic/go-demo-6
      context: .
      docker: {}
  tagPolicy:
    envTemplate:
      template: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}:{{.VERSION}}'
    local: {}
deploy:
  kubectl: {}
profiles:
- name: dev
  build:
    tagPolicy:
      envTemplate:
        template: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}:{{.DIGEST_HEX}}'
      local: {}
  deploy:
    helm:
      releases:
        - name: go-demo-6
          chartPath: charts/go-demo-6
          setValueTemplates:
            image.repository: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}'
            image.tag: '{{.DIGEST_HEX}}'
```

Explanation of `skaffold.yaml`

This might be the first time you're going through `skaffold.yaml`, so let us briefly describe what we have in front of us.

The two main sections of `skaffold.yaml` are:

- `build`: Defines how we build container images.
- `deploy`: Describes how we deploy the containers.

The `build` section #

We can see that images are built with `docker`. The interesting part of the `build` section is `tagPolicy.envTemplate.template` that defines the naming scheme for our images. It expects environment variables `DOCKER_REGISTRY` and `VERSION`.

The `deploy` section #

The `deploy` section is uneventful and only defines `kubectl` as the deployment mechanism.

We won't be using the root `build` and `deploy` sections of the config. They are reserved for the Jenkins pipeline. There are slight differences when building and deploying images to permanent and development environments. Our target is the `dev` profile which overwrites the `build` and the `deploy` targets.

The `DIGEST_HEX` variable #

During the development builds, we won't be passing `VERSION`, but let it be autogenerated through `scaffold`'s "special" variable `DIGEST_HEX`. Through it, every build will be tagged using a unique string (e.g., `27ffc7f...`). Unfortunately, as of this writing, this variable is now deprecated and removed. We will have to replace it as there is no fix for this in Jenkins X's buildpacks yet.

```
cat scaffold.yaml \  
| sed -e 's@DIGEST_HEX@UUID@g' \  
| tee scaffold.yaml
```

We replaced the `DIGEST_HEX` with a `UUID`, which will have the same function and a similar format.

The `deploy` section of the `dev` profile #

The `deploy` section of the `dev` profile changes the type from `kubectl` to `helm`. It sets the `chartPath` to `charts/go-demo-6` so that it deploys whatever we defined as the application chart in the repository. Further down, it overwrites `image.repository` and `image.tag` **Helm** values so that they match those of the image we just built.

The `DOCKER_REGISTRY` variable #

The only unknown left in that YAML is the `DOCKER_REGISTRY`. It should point to the registry where we store container images. *Should we define it ourselves? If so, what is the address of the registry?*

I already stated that DevPods contains everything we need to develop applications. So, it should come as no surprise that `DOCKER_REGISTRY` is already defined for us. We can confirm that by outputting its value.

```
echo $DOCKER_REGISTRY
```

The output will vary depending on the type of the Kubernetes cluster you're running and whether the Docker registry is inside it or you're using a service. In GKE it should be `gcr.io`, in AKS it should be the name of the cluster followed with `azurecr.io` (e.g., `THE_NAME_OF_YOUR_CLUSTER.azurecr.io`), and in EKS it is a combination of a unique ID ending with `amazonws.com` (e.g., `036548781187.dkr.ecr.us-east-1.amazonaws.com`). The exact address doesn't matter since all we need to know is that it is stored in the environment variable `DOCKER_REGISTRY`.

Speaking of variables, it might be useful to know that many others were created for us.

Checking the `env` variables

```
env
```

The output should display over a hundred variables. Some of them were created through the Kubernetes services, while others were injected through the process of creating the DevPod. I'll let you explore them yourself. We'll be using some of them soon.

Initializing the Helm client

Next, we should initialize the **Helm** client residing in the DevPod so that we can use it to deploy charts.

```
kubectl create \
  -f https://raw.githubusercontent.com/vfarcic/k8s-specs/master/helm/tiller-rbac.yml \
  --record --save-config

helm init --service-account tiller
```



We are using `tiller` only to simplify the development. For a more secure cluster, you should consider using **Helm** with `tiller` (server-side **Helm**) by executing `helm template` command.

Now we're ready to build and deploy our application in the personal development environment

environment.

Generating **UUID** and running **scaffold** using the **dev** profile

Since **Scaffold** does not generate the **DIGEST_HEX** nor our replacement **UUID**, we will have to create one before we run the dev profile. So we will prefix our **scaffold** run with **export UUID=\$(uuidgen)**.

```
export UUID=$(uuidgen)
scaffold run --profile dev
```

We run scaffold using the **dev** profile.

Inspecting the output

If you inspect the output, you'll see that quite a few things happened. It built a new container image using the *go-demo-6* binary we built earlier. Afterward, it installed the application chart. But, where was that chart installed? We could find that out from the output, but there is an easier way. The Namespace where Scaffold installs applications is also defined as an environment variable.

```
echo $SKAFFOLD_DEPLOY_NAMESPACE
```

In my case, the output is **jx-edit-vfarcic** (yours will use a different user). That's the personal Namespace dedicated to my development of the *go-demo-6* application. If I'd work on multiple projects at the same time, I would have a Namespace for each. So, there will be as many **jx-edit-*** Namespaces as there are developers working in parallel, multiplied with the number of projects they are working on. Of course, those Namespaces are temporary, and we should delete them together with DevPods once we're finished working on a project (or when we're ready to go home). It would be a waste to keep them running permanently. We'll keep our DevPod for a while longer so that we can explore a few other goodies it gives us.

Confirming the installation of application

Let's confirm that the application and the associated database were indeed installed when we executed **scaffold run**.

```
kubectl -n $SKAFFOLD_DEPLOY_NAMESPACE \
get pods
```


The output is as follows.

```
NAME                                READY STATUS  RESTARTS  AGE
go-demo-6-go-demo-6-...            1/1   Running  3         4m
go-demo-6-go-demo-6-db-arbiter-0    1/1   Running  0         4m
go-demo-6-go-demo-6-db-primary-0    1/1   Running  0         4m
go-demo-6-go-demo-6-db-secondary-0  1/1   Running  0         4m
```

I hope that you can already see the benefits of using DevPod. If you are, you'll be pleased that there's quite a lot left to discover and quite a few benefits we haven't yet explored.

Inspecting `watch.sh`

When we import a project, or when we create a new one using one of the quickstarts, one of the files created for us is `watch.sh`. It is a simple yet handy script that combines the commands we run so far, and it adds a twist you can probably guess from its name. Let's take a look at what's inside.

```
cat watch.sh
```

The output is as follows.

```
#!/usr/bin/env bash

# watch the java files and continuously deploy the service
make linux
skaffold run -p dev
reflex -r ".go$" -- bash -c 'make linux && skaffold run -p dev'
```

We can see that the first two lines (excluding misplaced comments) are the same as those we executed. It's building the binary (`make linux`) and executing the same `skaffold` we run. The last line is the one that matters.

Adding the missing `UUID` variable

As you can see, it is missing the `UUID` variable, so let's add that to the `watch.sh`

```
cat watch.sh | sed -e \
's@skaffold@UUID=$(uuidgen) skaffold@g' \
| tee watch.sh
```

[Reflex](#) is a nifty tool that watches a directory and reruns commands when specific files change. In our case, it'll rerun `make linux && skaffold run -p dev` whenever

any of the `.go` files are changed. That way, we'll build a binary and a container image, and we'll deploy a **Helm** chart with that image every time we change any of the source code files ending with `.go`. In other words, we'll always have the application running the latest code we're working on. Isn't that nifty?

Let's try it out.

```
chmod +x watch.sh  
nohup ./watch.sh &
```

To be on the safe side, we've assigned executable permissions to the script before we executed it. Since we used `nohup` in the second command, it'll run even if we end our session, and `&` will make sure that the watcher is running in the background. Please press the enter key to go back to the terminal.

We run the script in the background so that we can execute a few commands that will validate whether or not everything works as expected. The downside is that we won't see the output of the script. We'll fix that later when we start working with multiple terminals. For now, please note that the script will run `make linux` and `scaffold run -p dev` commands every time we change any of the `.go` files.

Exiting the DevPod

Now, let's exit the DevPod and confirm that what's happening inside it indeed results in new builds and installations every time we change our Go source code.

```
exit
```

Running the application

Before we change our code, we'll confirm that the application indeed runs. To do that, we'll create port forwarding to our app, instead of bothering to set up "proper" **Ingress**.


Please replace `[...]` in the command that follows with your GitHub user.

The output is as follows.

```
export GH_USER=[...]
```

```
kubectl --namespace jx-edit-$GH_USER \  
port-forward service/go-demo-6 \  
8085:80 &
```

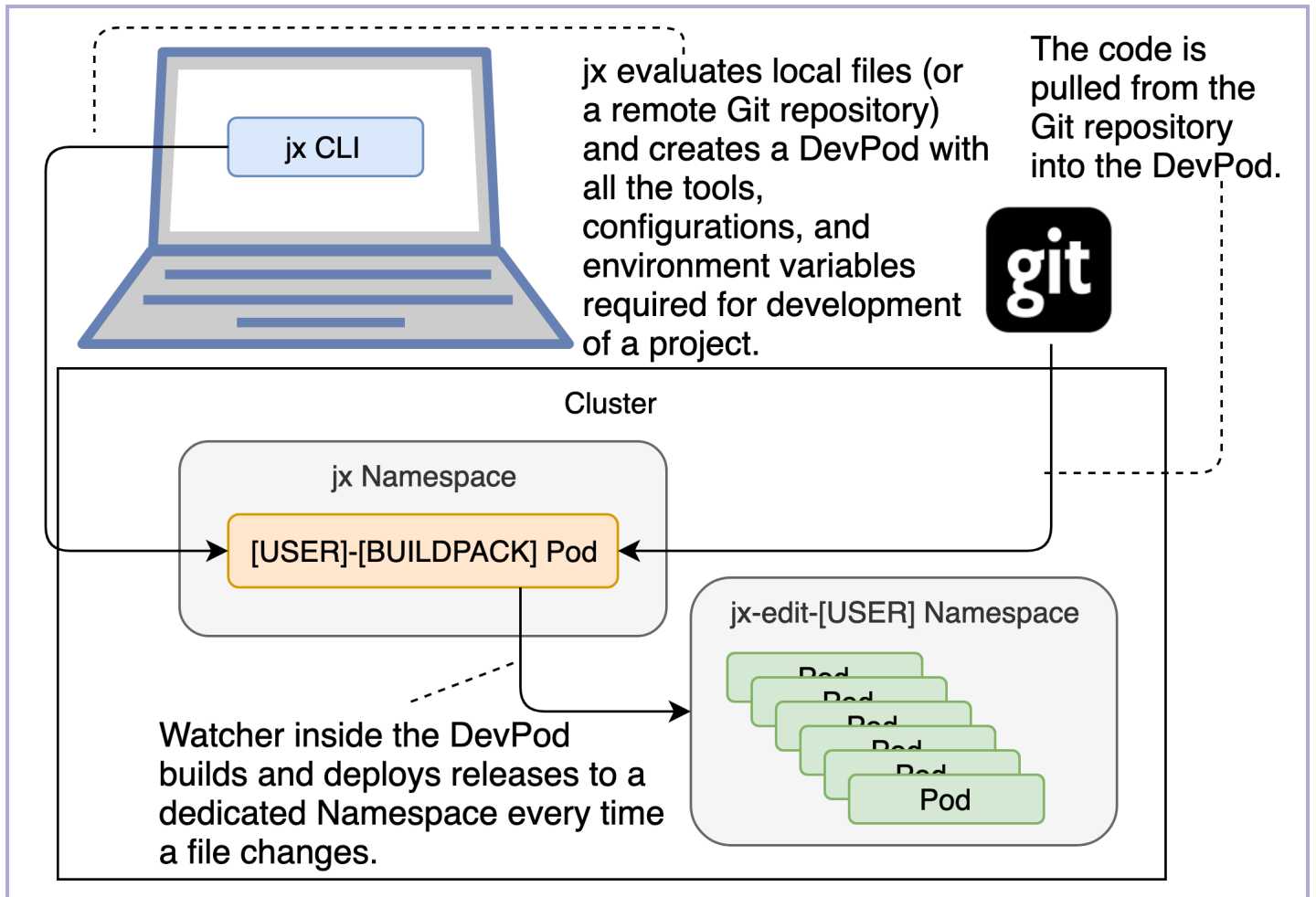
We created port forwarding from localhost on port `8085` to the Pod of our application accessible through the Service `go-demo-6`. Now we should be able to access the application in the personal development Namespace.

 We're running port forwarding as a background process. Please press the enter key if the terminal input is not released.

```
curl "http://localhost:8085/demo/hello"
```

The result of the `curl` command should output `hello, world!`, thus confirming that we are still running the initial version of the application. Our next mission is to change the source code and confirm that the new release gets deployed in the personal development environment.

 If the output is HTML with `503 Service Temporarily Unavailable`, you were too fast; you didn't give the process enough time. Please wait for a few moments until the application is up-and-running, and repeat the `curl` command.



The process of creating a DevPod and building and deploying applications from inside it

Now we need to figure out how to change the source code running inside the DevPod so that the process of building and deploying is repeated. As you will soon see, there is more than one way to do that.