Synchronizing Code from a Laptop into a DevPod

This lesson gives instructions to synchronize code from the DevPod to your local system and then build the application.

We'll cover the following

I hope that you liked the idea of using a browser-based IDE like Visual Studio Code. On the other hand, the chances are that you believe that it might be useful in some scenarios, but that the bulk of your development will be done using desktop-based IDE. In other words, I bet that you prefer to work with local files. If that's the case, we need to figure out how to sync them with DevPod. But, before we do that, we'll add a critical component to our development process. We're missing tests, and that is, as I'm sure you already know, *unacceptable*.

Adding unit tests in `Makefile`

Given that we are using `Makefile` to specify our targets (at least when working with Go), that's the place where we'll add unit tests. I assume that you want to run unit tests every time you change your code and that you'll leave slower types of tests (e.g., functional and integration tests) to Jenkins. If that's not the case, you should have no problem extending our examples to run a broader set of validations.

> ⚠ Remember what we said before about `Makefile`. It expects tabs as indentation. Please make sure that the command that follows is indeed using tabs and not spaces if you're typing the commands instead of copying and pasting from the Gist.

```
echo 'unittest:
	CGO_ENABLED=$(CGO_ENABLED) $(GO) \\
	test --run UnitTest -v
' | tee -a Makefile
```

We added a `unittest` target with `go test` command limited to functions that contain `UnitTest` in their names.

Modifying `watch.sh`

Next, we need to modify `watch.sh` so that it executes the new target.

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

Now that we added unit tests to `Makefile` and `watch.sh`, we can go back to our original objective and figure out how to synchronize local files with those in a DevPod.

Synchronizing local files with DevPod

We'll use ksync. It transparently updates containers running inside a cluster from a local checkout. That will enable us to use our favorite IDE to work with local files that will be synchronized with those inside the cluster.

To make things simpler, `jx` has its own implementation of ksync that will connect it with a DevPod. Let's fire it up.

> ⚠ At the time of this writing (April 2020), there is an open issue that

> prevents the command that follows from working correctly. If it fails, please consult the issue 7015.

```
jx sync --daemon
```

We executed `jx sync` in `daemon` mode, allowing us to run it in the background instead of blocking a terminal session.

It'll take a few moments to get everything up and running. The final message should state that `it looks like 'ksync watch' is already running so we don't need to run it yet...` When you see it, you'll know that it is fully operational, and all that's left is to press *ctrl+c* to go back to the terminal session. Since we specified `--daemon`, `ksync` will continue running in the background.

Creating a new DevPod

Now we can create yet another DevPod. This time, however, we'll add `--sync` argument. That will give it a signal that we want to use `ksync` to synchronize our local file system with the files in the DevPod.

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

Now we need to repeat the same commands as before to start the watcher inside the DevPod. However, this time we will not run it in the background since it might be useful to see the output in case one of our tests fail and we might need to apply a fix before we proceed with the development. For that reason, we'll open a second terminal. I recommend that you resize two terminals so you can see them both.

Running the watcher

Open a second terminal session.

> ⚠ If you are using EKS, you'll need to recreate the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION`. Otherwise, your second terminal will not be able to authenticate against Kube API.

Next, we'll enter the DevPod and execute the same commands, ending with running the `watch.sh` script.

```
jx rsh --devpod

unset GOPATH

go mod init

helm init --client-only

chmod +x watch.sh

./watch.sh
```

Now that `watch.sh` is running in the foreground, we can see the results of building, testing, and deploying development releases created every time we change our source code.

Confirming the deployment of the latest release

The last time we modified the files in the DevPod, we did not push them to Git. Since we did not have synchronization, they were lost when we deleted the Pod. Let's confirm that we are still at square one by sending a request to the application.

Please return to the first terminal.

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

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

The output is `hello, world!` thus confirming that our source code is indeed intact and that the watcher did its job by deploying a new release based on the original code. If the output is `hello, devpod!`, the new deployment did not yet roll out. In that case, wait for a few moments and repeat the `curl` command.

Modifying the files locally

Next, we'll make a few changes to the files on our laptop.

```
cat main.go | sed -e \
    's@hello, world@hello, devpod with tests@g' \
    | tee main.go

cat main_test.go | sed -e \
    's@hello, world@hello, devpod with tests@g' \
    | tee main_test.go
```

Since we changed the file, and if you are quick, we should see the result of the new iteration of `watch.sh`. Go back to the second terminal; you should see that the binary is built, that the unit tests are executed, and that skaffold built a new image and upgraded the development release using Helm.

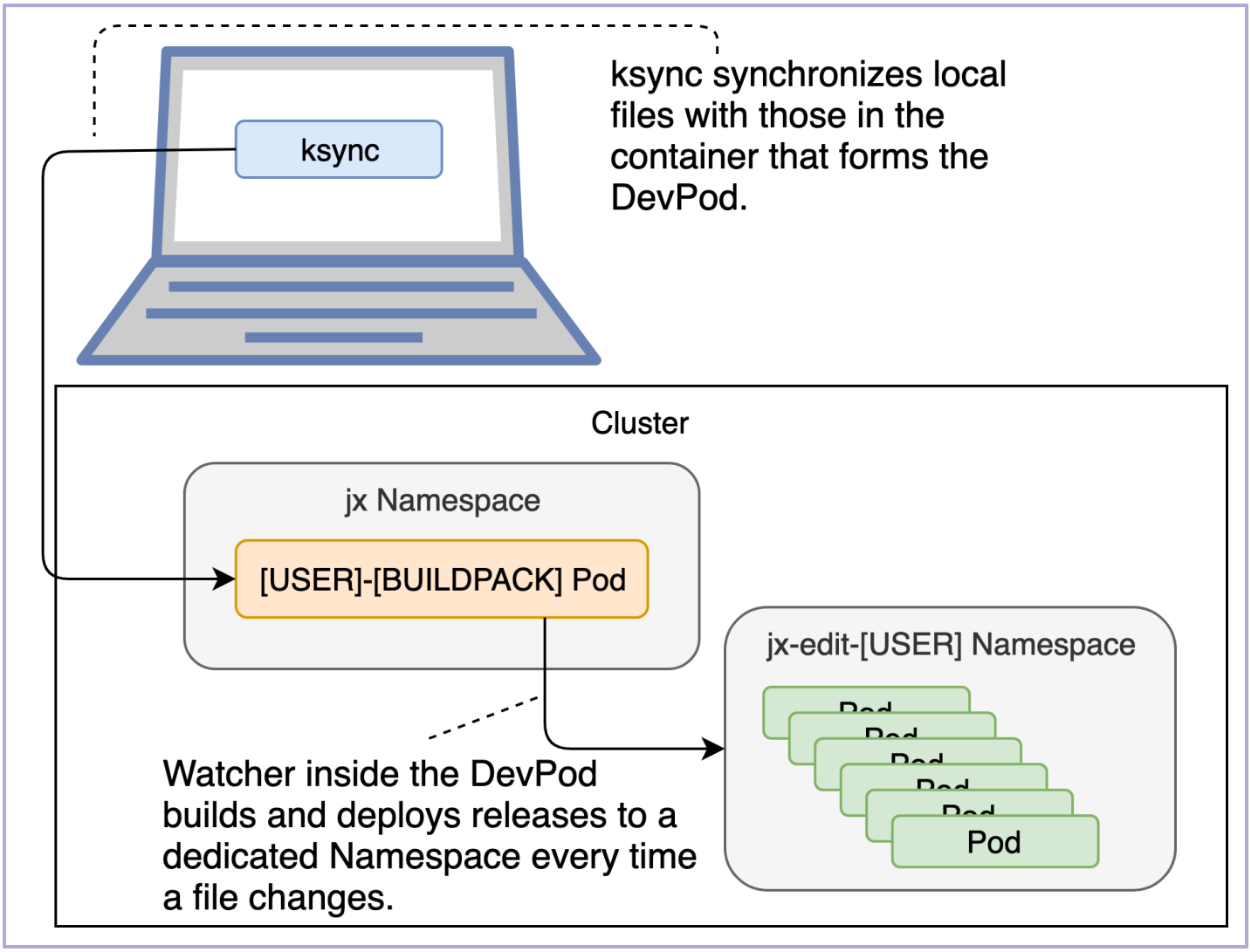
Confirming the synchronization of files

Now that we observed that the process run through yet another iteration, we can send a request to the application and confirm that the new release indeed rolled out. Please go to the first terminal to execute the `curl` command that follows.

```
kubectl --namespace jx-edit-$GH_USER \
    port-forward service/go-demo-6 \
    8086:80 &

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

This time, the output is `hello, devpod with tests!`. From now on, every time we change any of the local Go files, the process will repeat. We will be notified if something (e.g., tests) fails by the output from the second terminal. Otherwise, the application running in our personal environment (Namespace) will always be up-to-date.



The process of using ksync to synchronize local files with those in the container that forms the DevPod

Deploying the new release on staging

Next, we'll imagine that we continued making changes to the code until the new feature is done. The only thing left is to push them back to the GitHub repository. We'll ignore the fact that we should probably make a pull request (explanation is coming in the next chapter), and push directly to the master branch.

```
git add .

git commit \
    --message "devpod"

git push
```

You should be familiar with the rest of the process. Since we pushed a change to the master branch, Jenkins will pick it up and run all the steps defined in the pipeline. As a result, it will deploy a new release to the staging environment. As

always, we can monitor the activity of the Jenkins build.

```
jx get activity \
    --filter go-demo-6 \
    --watchjx get activity -f go-demo-6 -w
```

The new release should be available in the staging environment once all the steps `Succeeded` and we can cancel the activity watcher by pressing *ctrl+c*.

Let's take another look at the available applications.

```
jx get applications
```

The output should be the same as before. However, this time we're interested in the URL of the staging environment since that's where the new release was rolled out after we pushed the changes to the master branch.

Please copy the URL of the staging release (the second one) and paste it instead of `[...]` in the commands that follow.

```
STAGING_URL=[...]

curl "$STAGING_URL/demo/hello"
```

As expected, the output is `hello, devpod with tests!` thus confirming that the new release (the same one we have in the private development environment) is now rolled out to staging.

Deleting the DevPod

We're done with our DevPod combined with ksync synchronization so we can delete it from the system. We're not going to make any more changes to the code, so there is no need for us to waste resources on the DevPod.

```
jx delete devpod
```

Please press `y` followed by the enter key to confirm the deletion.

There's still one more development-related topic we should explore.