JEDD HOPKINS  
CAPGEMINI  
  
  
3. Concourse Pipeline

**\*\*Please note, many method, class and variable names have been changed to generic terms such as ‘thing’ to maintain anonymity\*\***

**Contents**

**Concourse Pipeline**

Page 2 – User Story/Sub-Tasks

Page 3 – Building my Pipeline

Page 7 – The Final Pipeline

Page 9 – Conclusion

**Please refer to submission:**

1. **Introduction & Selenium Testing**

**For an introduction to the company, how we operate and how I pick up tickets**

**User Story/Sub-tasks**

During one morning stand up I explained to my team that I was looking for a new ticket to work on. It is company standard to involve the whole dev team when deciding which new tickets any single developer should pick up. My team lead needs to have the confidence that I can complete the ticket to a high standard and by involving the whole team in such decisions it enables us to have a conversation on how the sprint is currently going and which tickets may take priority.

One story I picked up from the backlog was a ticket that would enable us to build docker images of our applications and push them to a docker registry using a tool called Concourse. This required me to learn about Continuous Integration/Continuous Deployment (CI/CD) pipelines and then put what I had learnt into practice by building out an initial pipeline. This pipeline would go on to become an evolving template that we, as a team would use for all applications moving forward. Due to the nature of our project having multiple development teams across multiple companies, it was vital that we try to collaborate as much as possible. The end product of this ticket would mean that any development team on the project will be able to create a docker image of their micro-service and push it to one, unified docker registry. These images could then be used by any of the 6 development teams in their own micro-services where they have dependencies on them.

The pipeline had to focus on two initial steps:

1. Running a maven build on the source code to initiate all tests and generate a build output subject to all the tests passing.
2. Push the generated docker image to our docker repository in Amazon Web Services (AWS).

This ticket had previously been broken up into 5 sub-tasks during a backlog refinement session which I played an active role in. The sub-tasks are as follows:

1. Learn about Concourse
2. Set up a test pipeline using the Concourse documentation
3. Understand how Concourse gets its resources (access to my source code)
4. Understand how it does something with those resources (tasks & jobs)
5. Transfer all this knowledge into my own project

Breaking up the ticket in this way gave me more clarity on what needed to be achieved and also split the ticket up into more manageable chunks, enabling me to work in a more agile manner.

**Building my Pipeline**

I began reading the Concourse documents which do a great job in explaining all the elements needed to create and run pipelines. I also came across a ‘Stark & Wayne Concourse Tutorial’ which was a brilliant website and walked me through many of the beginning steps.

One of the first things I had to do was to download the Fly Command Line Interface (CLI). Downloading this gave me the language I needed to talk to Concourse through terminal commands, doing things such as logging into my Concourse environment and setting up the relationship between my git repo and the Concourse UI that I see in my web browser.

Using the Stark and Wayne tutorial, I started to build a simple ‘hello world’ pipeline using a docker image that the tutorial provides. This pipeline only had one job, to run the image. This image, once ran, would then print ‘hello world’ to the concourse UI in the same way it would. The final passing pipeline can be seen in Figure 1.1.

A screenshot of a cell phone

Description automatically generated

Figure 1.1 – Passing pipeline

Following the tutorials further allowed me to work my way through the sub-tasks numbered 2-4 which were associated with the ticket.

I found that the resources are the heart and soul of Concourse. They represent all external inputs and outputs of jobs in the pipeline. Each resource represents a versioned artifact. The most common type of resource, which I would go on to include in my pipeline, is a git resource. This resource pulls in the git repository which houses the pipeline.yaml in which Concourse needs to use for configuration. The other aspect of pipelines is the jobs. Jobs determine the actions of your pipeline. They determine how resources progress through it, and how the pipeline is visualized.

Once I had a basic pipeline.yaml set up I needed to use the fly CLI to set up my connection between Concourse and my micro-service. To do this I navigated to the root of my project then I ran a command that will set up my pipeline, tell Concourse where to find my pipeline.yaml, name the pipeline and provide any variables that may be needed in set up.

A screenshot of a cell phone

Description automatically generatedThis can be seen in Figure 1.2

Figure 1.2 – Terminal command

This command to set up my pipeline reads:

**fly -t example-env set-pipeline -c ci/pipeline.yaml -p my-example-app -v gitlab-username=\*\*\*\*\*\*\*\* -v gitlab-repository-password=\*\*\*\*\*\*\*\***

This can look confusing at first, but it is actually quite simple.

* **fly** – This has to be run before every fly command. It lets the terminal know that the following command runs using fly. Much in the same way that using the keyword `git` works before running any git command in the terminal.
* **-t example-env** – This sets the target. This is the target environment that I want my pipeline to be set in. I have called this ‘example-env’.
* **set-pipeline -c ci/pipeline.yaml** – This is how I tell Concourse where to find the configuration file. From the root of my project it needs to look in the ci folder to find the pipeline.yaml.
* **-p my-example-app** – The -p denotes the name I want to call my pipeline. For this I have called it ‘my-example-app’.
* **-v (optional)**– This allows me to inject variables into the config. Good for sensitive information such as usernames and passwords. I am injecting my GitLab username and GitLab password as I do not want these floating around in the code for security reasons.

As you can see from the full pipeline.yaml screenshot at the end of this piece, these are injected into the pipeline.yaml using a double bracket notation.

Figure 1.3 shows a screenshot of what is shown to me in the IDE terminal when I run a fly command like the one in Figure 1.2. In this example I have just made a small change to my git-repo-name to show how the fly CLI formats this. It initially asks me to log in and get a token to access the pipeline. It then provides me with an easy to read comparison of what has been changed. This side by side comparison is my first line of defence in debugging (or catching the bug before it happens). When checking over this information it can be clear that I have made a typo like the one below, thus giving me chance to stop the update and go back and fix my error, saving my pipeline from potentially failing. In the penultimate line it asks me to confirm the changes are correct and final, which for the purpose of this screenshot, I do.

A screenshot of a cell phone

Description automatically generated

Figure 1.3 – Fly command confirmation

A screenshot of a cell phone

Description automatically generatedOne small problem I ran into pretty soon was figuring out how to debug my pipeline.yaml past the point of Figure 1.3. There were times that the changes that I made looked good but there may have been an underlying error that I did not quite see at first. Sometimes the Concourse UI would not display much information as to what has gone wrong.

Figure 1.4 – Infinite build

Figure 1.4 is the result of my git-repo-name change. As you can see there are no error messages, instead just an infinite loop of trying to discover any new versions of the source-code.

The ticks at the side, however, do give some hint as to what has happened. My build is currently stuck discovering any new versions of source code. It will stay on this task indefinitely. This is because it is looking for a git-repo-name that doesn’t exist, therefore it will never find a new version.

This is quite a niche example of a time where I would not receive an error message from concourse. For all the other times I can use the window that serves as an extension of my IDE terminal. Normally I would do my debugging through my IntelliJ IDE as there are tools at my disposal which allow me to take a closer look at certain classes and components. This allows me to check whether objects have been instantiated or if information is being passed between objects corrected.

Luckily for me the concourse UI contains error messages just like any IDE would, these messages were really helpful in determining what part of the pipeline was wrong. Examples of this would be failing tests.

The debugging feature that I could not access was the ability to stop the program and inspect code at certain points, however seeing as concourse was taking my pipeline.yaml and building the project, it was not necessary for me to check the inner functionality of the components within.

**The Final pipeline.yaml**

Figure 2.1 – Full pipeline.yaml

Running through the above code, from the top, we have two resources and three jobs.

* The first resource holds the Gitlab uri location of the source code, the branch in which the changes will be made and the necessary credentials to access it. It also tells Concourse to skip verification. This is because we are working within a protected environment which needs certain certificates to access it. This command negates the need for the certificates.
* The second resource holds the location of the AWS Docker image repository that we push Docker images to.
* The first part of the job gets the source code from the repository using the source code resource
* The part of the second job executes the maven build by locating the build-mvn.yaml file in which the config for the maven build is held.
* Finally, the third part of the job takes the maven build output, signs it with the person who made the commits, creates the docker image and pushes the image to the location held in the docker build resource.
* The result of all of this is that the code above builds, and all tasks are successfully executed without the need for human interaction. The pipeline will recognise when a change has been made to a Gitlab repository, and in turn it rebuilds the Docker image using the new code and deploys it to the Docker registry.

**Conclusion**

This piece of work gave me the chance to learn a completely new technology, exploring CI/CD automation.

It also helped me build on the debugging skills I had, by branching out to other ways of working through problems within my code when my usual go to method of debugging was not necessarily available.

I also believe that this piece of work has given me more confidence when exploring new technology whilst working alone, granted the documentation was very detailed and easy to follow. I feel that if there were any more tickets around this topic, expanding on the pipeline and what I have achieved here, I would be very confident in picking the task up.

The business outcome of this piece of work will not only help our lot, but also others across the whole project. The pipeline I have produced will now serve as a template for every other microservice within our environment that needs CI/CD. This means that human error associated with setting up these pipelines can be eradicated, thus streamlining this particular process whilst increasing the velocity of not only our own team, but also the other suppliers on the lot.