

# Docker for Enterprise Developers Exercises

Some of these exercises ask you to run things on your local machine. In order to do so, you will need Docker as well as some other bash utilities installed; in case this is a problem, the 'elk' node (username 'ubuntu') has everything you'll need all pre-configured. **When the instructions refer to 'your' machine, this means either your laptop or the 'elk' node - whichever you have chosen to use.**

Be aware that copy-pasting of commands or code snippets from this PDF may apply changes to some characters e.g. quotes, tabs, which may leads to errors. Please consider typing suggested commands and code snippets in case you encounter any issues.

**Windows Users:** Please note that in all exercises we will use Unix style paths using forward slashes ('/') instead of backslashes ('\'). On Windows you can work directly with such paths by either using a **Bash** terminal or a **Powershell** terminal. Powershell can work with both Windows and Unix style paths.

## Contents

<b>1</b>	<b>Workshop Sample Application</b>	<b>4</b>
1.1	Using Docker compose to build & run the application . . . . .	4
1.2	Running the application on Kubernetes . . . . .	7
1.2.1	Prerequisites . . . . .	7
1.2.2	Tagging and pushing images to Docker Hub . . . . .	8
1.2.3	Creating the application in Kubernetes on Minikube . . . . .	8
1.3	Exploring the application . . . . .	9
1.3.1	The Database . . . . .	9
1.3.2	The API . . . . .	10
1.3.3	The UI . . . . .	10
1.4	Conclusion . . . . .	10
<b>2</b>	<b>Edit and Continue</b>	<b>10</b>
2.1	Modifying and re-running the app . . . . .	10
2.2	Mounting source code . . . . .	11
2.3	Automatically restart the application . . . . .	12
2.4	Cleaning up . . . . .	12
2.5	Optional Challenge . . . . .	13
2.6	Conclusion . . . . .	13
<b>3</b>	<b>Debugging in a Container</b>	<b>13</b>
3.1	Debug the Java API . . . . .	13
3.2	Optional Bonus: Debugging a Node JS application . . . . .	16
3.3	Conclusion . . . . .	19
<b>4</b>	<b>Docker Compose</b>	<b>19</b>
4.1	Back-end Development . . . . .	19
4.2	Front-end Development . . . . .	21
4.3	Conclusion . . . . .	23
<b>5</b>	<b>Unit Tests</b>	<b>23</b>
5.1	Extending the Node JS UI . . . . .	23

5.2	Adding Unit Tests . . . . .	24
5.3	Not Shipping Unit Tests . . . . .	25
5.3.1	Mounting test code . . . . .	25
5.4	Creating a Test Image . . . . .	26
5.5	Edit and Continue when writing tests . . . . .	26
5.6	Optional Challenge: Unit Testing a .NET Core Application . . . . .	27
5.6.1	Creating a .NET Core application . . . . .	27
5.6.2	Containerizing the .NET Application . . . . .	30
5.6.3	Adding Edit and Continue Experience . . . . .	31
5.6.4	Freeing the application image from test code . . . . .	31
5.7	Conclusion . . . . .	32
<b>6</b>	<b>API Tests</b>	<b>32</b>
6.1	Adding Postgres as a backing DB . . . . .	32
6.2	Writing API tests . . . . .	33
6.3	Optional Challenge: Exercise with Python/Flask API backed by Mongo DB . . . . .	35
6.3.1	Creating a Python Flask API . . . . .	35
6.3.2	Adding Mongo DB as a backing DB . . . . .	36
6.3.3	Writing API tests . . . . .	37
6.4	Conclusion . . . . .	40
<b>7</b>	<b>End-to-End Tests</b>	<b>40</b>
7.1	Writing the Tests . . . . .	40
7.2	Conclusion . . . . .	42
<b>8</b>	<b>Service Discovery</b>	<b>43</b>
8.1	Creating an Application . . . . .	43
8.2	Using Aliases in Service Discovery . . . . .	44
8.3	Network Partitioning . . . . .	45
8.4	Kubernetes and Service Discovery . . . . .	45
8.5	Conclusion . . . . .	46
<b>9</b>	<b>Health Checks</b>	<b>46</b>
9.1	Health Checks in Swarm . . . . .	47
9.1.1	Implementing a Health Endpoint . . . . .	47
9.1.2	Adjusting the Dockerfile and building the Images . . . . .	47
9.1.3	Running the Application with Health Checks in a Swarm . . . . .	47
9.1.4	Responding to Health Failure . . . . .	49
9.2	Health Checks in Kubernetes . . . . .	50
9.2.1	Implementing a Health Endpoint . . . . .	50
9.3	Optional Challenge: Health Check Endpoint in Python . . . . .	51
9.3.1	Creating the application . . . . .	51
9.3.2	Defining the Health Check Command . . . . .	52
9.3.3	Running a Service with Health Check . . . . .	52
9.4	Conclusion . . . . .	53
<b>10</b>	<b>Defensive Programming</b>	<b>54</b>
10.1	Defining an Entrypoint Script . . . . .	54
10.2	Conclusion . . . . .	57
<b>11</b>	<b>Logging</b>	<b>58</b>
11.1	Logging in Java . . . . .	58
11.2	Optional: Logging in .NET Core . . . . .	60
11.3	Optional: Logging in Python/Flask . . . . .	62
11.4	Conclusion . . . . .	63
<b>12</b>	<b>Error Handling</b>	<b>63</b>

12.1 Implementing the Hasher . . . . .	63
12.2 Implementing the Worker . . . . .	64
12.3 Running the application . . . . .	65
12.4 Conclusion . . . . .	65
<b>13 Builder</b>	<b>66</b>
13.1 Optimize the Dockerfile . . . . .	66
13.1.1 Collapse layers and clean up . . . . .	67
13.1.2 Decrease Build Times . . . . .	67
13.2 Defining a multi-stage build . . . . .	68
13.3 Conclusion . . . . .	69
<b>14 Creating a Docker Swarm</b>	<b>70</b>
14.1 Creating a clean Slate . . . . .	70
14.2 Creating a new Swarm . . . . .	70
14.3 Conclusion . . . . .	71
<b>15 Routing Mesh</b>	<b>71</b>
15.1 Publishing a Service . . . . .	71
15.2 Scaling the Service . . . . .	72
15.3 Cleanup . . . . .	72
15.4 Conclusion . . . . .	72
<b>16 Secrets</b>	<b>72</b>
16.1 Setting up a demo . . . . .	73
16.2 Getting GitHub access credentials . . . . .	73
16.3 Creating and using a Secret . . . . .	73
16.4 Mocking Secrets . . . . .	74
16.5 Using Secrets in Kubernetes . . . . .	74
16.6 Conclusion . . . . .	75
<b>17 Configuration Management</b>	<b>75</b>
17.1 Planning Environment-Specific API Config . . . . .	76
17.2 Preparing Deployment to other Environments . . . . .	77
17.3 Configuration Management in Kubernetes . . . . .	78
17.4 Optional Challenge: Packaging and Shipping . . . . .	79
17.5 Conclusion . . . . .	79
<b>18 Installing UCP</b>	<b>79</b>
18.1 Prerequisites: . . . . .	79
18.2 Installing UCP . . . . .	79
18.3 Uninstalling UCP . . . . .	80
18.4 Conclusion . . . . .	80
<b>19 Layer 7 Load Balancing</b>	<b>80</b>
19.1 Pre-requisites: . . . . .	80
19.2 Enabling HTTP Routing Mesh . . . . .	81
19.3 Deploy Services with Routing Labels . . . . .	81
19.3.1 Creating the Service from UCP . . . . .	81
19.3.2 Creating the Service from the CLI . . . . .	81
19.4 Conclusion . . . . .	82
<b>20 Installing Docker Trusted Registry</b>	<b>82</b>
20.1 Prerequisites: . . . . .	82
20.2 Installing DTR . . . . .	82
20.3 Trusting DTR . . . . .	82
20.4 Testing DTR . . . . .	83
20.5 Uninstalling DTR . . . . .	83

20.6 Conclusion . . . . .	83
<b>21 Content Trust</b>	<b>84</b>
21.1 Pulling Images by Tag . . . . .	84
21.2 Pulling Images by Digest . . . . .	84
21.3 Using Docker Content Trust . . . . .	84
21.3.1 Understanding The Update Framework (TUF) . . . . .	85
21.4 Exploring Content Trust Metadata . . . . .	86
21.5 Conclusion . . . . .	87
<b>22 Image Scanning in DTR</b>	<b>87</b>
22.1 Enabling Image Scanning . . . . .	87
22.2 Creating a repo . . . . .	87
22.3 Investigating layers & components . . . . .	87
22.4 Conclusion . . . . .	88
<b>23 Image Promotion &amp; Webhooks</b>	<b>88</b>
23.1 Pre-requisites: . . . . .	88
23.2 Using Webhooks . . . . .	88
23.2.1 Creating the Service . . . . .	88
23.2.2 Registering the Webhook . . . . .	91
23.2.3 Triggering the Webhook . . . . .	91
23.3 Configuring Image Promotion . . . . .	92
23.4 Conclusion . . . . .	93
<b>24 Build Server</b>	<b>93</b>
24.1 Prerequisites . . . . .	93
24.2 Creating and Shipping the Jenkins Image . . . . .	94
24.3 Preparing the Repository in DTR . . . . .	94
24.4 Preparing a Source Repo . . . . .	95
24.5 Running Jenkins in the Swarm . . . . .	95
24.5.1 Preparing the Swarm Node . . . . .	95
24.5.2 Creating the Jenkins Service . . . . .	95
24.5.3 Finalizing the Jenkins Configuration . . . . .	96
24.6 Configuring Jenkins Jobs . . . . .	96
24.7 Conclusion . . . . .	96
<b>25 Appendix: Connecting to the Remote Desktop</b>	<b>97</b>
25.1 Windows Users . . . . .	97
25.2 Mac Users . . . . .	97

# 1 Workshop Sample Application

In this lab we are going to prepare a sample application that will serve us as a basis for most subsequent exercises. The backend is implemented in Java whilst the frontend is based on Node JS. Furthermore we use a Postgres database as our data store.

## 1.1 Using Docker compose to build & run the application

1. Navigate to your home folder on your development machine:

```
$ cd ~
```

2. Clone the repository containing seed code for all labs of the workshop application. We will use this folder for all our projects during the whole course:

```
$ git clone -b ee2.0 https://github.com/docker-training/ddev-labs
```

3. Build the sample application:

```
$ cd ddev-labs/ddev-seed
$ docker-compose build
```

Please be patient since the above command will build the images that back the 3 services that makes up our application. You should see something similar to this (abbreviated output shown):

```
$ docker-compose build
Building api
Step 1/12 : FROM maven:latest AS appserver
latest: Pulling from library/maven
9f0706ba7422: Already exists
d3942a742d22: Pull complete
62b1123c88f6: Pull complete
cd20aef20918: Pull complete
a45121f36b26: Pull complete
62ab37d06041: Pull complete
03c611b2838d: Pull complete
8c3c1f561032: Pull complete
463554ba15a0: Pull complete
3ef0fdd588bb: Pull complete
6a6871006876: Pull complete
74bd4e62771d: Pull complete
Digest: sha256:85987e6a9ad9847b65fd0b0cf78aa5542b9f84efd66c89d21b25ee39ffef1e7
Status: Downloaded newer image for maven:latest
---> d089198872b5
Step 2/12 : WORKDIR /usr/src/atsea
...
Building ui
Step 1/8 : FROM node:8-alpine
8-alpine: Pulling from library/node
88286f41530e: Pull complete
9d3e88382b55: Pull complete
b9798bb8846b: Pull complete
Digest: sha256:9640cfe623ffe57bafb29763a82fb04eb774337cb656c172938056e58e0afd2b
Status: Downloaded newer image for node:8-alpine
...
Building database
Step 1/7 : FROM postgres
latest: Pulling from library/postgres
9f0706ba7422: Already exists
df3070b9fd62: Pull complete
945954562465: Pull complete
...
Removing intermediate container ae6a96c51439
Successfully built 3ca762a69639
Successfully tagged ddev_db:latest
```

4. Run the application:

```
$ docker-compose up
```

Once again, be patient, it takes a few seconds to start up all services. Specifically the database takes a moment to initialize the very first time. You should see something similar to this (shortened):

```
Creating network "ddevsample_front-tier" with the default driver
Creating network "ddevsample_back-tier" with driver "overlay"
Creating ddevsample_ui_1 ...
Creating ddevsample_api_1 ...
Creating ddevsample_database_1 ...
Creating ddevsample_ui_1
Creating ddevsample_database_1
Creating ddevsample_api_1 ... done
Attaching to ddevsample_ui_1, ddevsample_database_1, ddevsample_api_1
ui_1      | Listening at 0.0.0.0:3000
database_1 | The files belonging to this database system will be owned by user "postgres".
database_1 | This user must also own the server process.
...
database_1 | performing post-bootstrap initialization ... ok
api_1     |
api_1     | .
api_1     | /\ / ____'--___-( )__--___\ \ \ \ \
api_1     | ( ( ) \ ___| '_ \| | | |'_ \ \ \ \ \
api_1     | \ \ / ___| |_) | | | | | | |_) | | | |
api_1     | ' | ___| . _ \| | | | | | | | | | | |
api_1     | =====|_|=====|___/=/_/_/_/
api_1     | :: Spring Boot ::          (v1.5.3.RELEASE)
api_1     |
database_1 | syncing data to disk ... ok
database_1 |
...
api_1     | 2017-07-06 13:59:49.054 INFO 1 --- [           main] com.docker.ddev.DdevApp
api_1     | 2017-07-06 13:59:49.057 INFO 1 --- [           main] com.docker.ddev.DdevApp
api_1     | 2017-07-06 13:59:49.166 INFO 1 --- [           main] ationConfigEmbeddedWebApplication
database_1 | done
database_1 | server started
database_1 | CREATE DATABASE
database_1 |
database_1 | CREATE ROLE
database_1 |
database_1 |
database_1 | /usr/local/bin/docker-entrypoint.sh: running /docker-entrypoint-initdb.d/init-db.sql
database_1 | CREATE TABLE
database_1 | ALTER TABLE
database_1 | ALTER ROLE
database_1 | INSERT 0 1
...
database_1 | INSERT 0 1
database_1 |
database_1 |
database_1 | LOG:  received fast shutdown request
database_1 | waiting for server to shut down...LOG:  aborting any active transactions
database_1 | LOG:  autovacuum launcher shutting down
database_1 | .LOG:  shutting down
database_1 | LOG:  database system is shut down
api_1     | 2017-07-06 13:59:51.011 INFO 1 --- [           main] s.b.c.e.t.TomcatEmbeddedServlet
api_1     | 2017-07-06 13:59:51.026 INFO 1 --- [           main] o.apache.catalina.core.Standard
api_1     | 2017-07-06 13:59:51.028 INFO 1 --- [           main] org.apache.catalina.core.Standa
api_1     | 2017-07-06 13:59:51.143 INFO 1 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].
...
database_1 | LOG:  database system is ready to accept connections
```

```

database_1 | LOG: autovacuum launcher started
api_1      | 2017-07-06 13:59:51.559 INFO 1 --- [          main] j.LocalContainerEntityManagerFa
...
api_1      | 2017-07-06 13:59:54.236 INFO 1 --- [          main] s.b.c.e.t.TomcatEmbeddedServlet
api_1      | 2017-07-06 13:59:54.244 INFO 1 --- [          main] com.docker.ddev.DdevApp

```

**Note:** If the very first time you start the application, it fails with an error message similar to this:

```
api_1      | 2017-07-06 14:21:37.577 ERROR 1 --- [          main] o.s.boot.SpringApplication      : Ap
```

then just stop the application with CTRL-C and start again with `docker-compose up`. This is a timing problem where the API tries to access the DB that is not yet ready. In a later exercise we will show how one could prevent such situations from happening.

5. Ensure that the application is running by opening an additional terminal window, navigating to the application folder and listing the running services:

```
$ cd ~/ddev-labs/ddev-seed
$ docker-compose ps
```

you should see something like this:

```
$ docker-compose ps
```

Name	Command	State	Ports
ddevsample_api_1	java -jar /app/ddev-0.0.1- ...	Up	0.0.0.0:5005->5005/tcp, 0.0.0.0:80
ddevsample_database_1	docker-entrypoint.sh postgres	Up	0.0.0.0:5432->5432/tcp
ddevsample_ui_1	/bin/sh -c node src/server.js	Up	0.0.0.0:3000->3000/tcp

6. Test the application by opening a browser window and navigating to `http://<publicIP>:3000/pet`. You should be seeing cat pictures.
7. Refresh the view a few times and make sure you see different cat images.
8. When done, stop the application by pressing CTRL-C in the terminal you launched from.
9. Remove the application assets (stopped containers, networks etc):

```
$ docker-compose down
```

## 1.2 Running the application on Kubernetes

### 1.2.1 Prerequisites

For this exercise you need to have `kubectl` and `minikube` installed on your development machine.

1. Check that Minikube is running:

```
$ minikube version
```

If it isn't (and **only** if it isn't), start it via:

```
$ minikube start --vm-driver=none --apiserver-ips 127.0.0.1 --apiserver-name localhost
```

2. You have built the application images as described in the previous part with `docker-compose build`.
3. You're logged in to Docker Hub. If not use the following command to do so:

```
$ docker login
```

enter your credentials when asked to do so.

### 1.2.2 Tagging and pushing images to Docker Hub

To be able to run this application in Kubernetes on Minikube we need to have the images (publicly) available on Docker Hub.

1. Define an environment variable with your Docker ID:

```
$ export MY_DOCKER_ID=<your Docker ID>
```

2. Run the script tag-and-push.sh to tag and push the images that we had previously built:

```
$ ./tag-and-push.sh
```

### 1.2.3 Creating the application in Kubernetes on Minikube

Kubernetes does not (yet) have something similar to a docker-compose or stack file. Thus we have to use a collection of YAML (or JSON) snippets to define the various objects. Luckily we can put all into one single file.

1. Open the file kube.yaml and at each image name, replace user with your Docker ID.
2. Create all the Kubernetes objects that make up the application using the YAML file kube.yaml:

```
$ kubectl apply -f kube.yaml
```

you should see this:

```
namespace "ddev" created
secret "db-secret" created
deployment "database" created
service "database" created
deployment "api" created
service "api" created
deployment "ui" created
service "ui" created
```

3. Use kubectl to list all objects created in the namespace ddev:

```
$ kubectl -n ddev get all
```

you should see something similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/api	1	1	1	1	1m
deploy/database	1	1	1	1	1m
deploy/ui	1	1	1	1	1m

NAME	DESIRED	CURRENT	READY	AGE
rs/api-5c677587f5	1	1	1	1m
rs/database-5d5cb5488f	1	1	1	1m
rs/ui-65d77678c9	1	1	1	1m

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/api	1	1	1	1	1m
deploy/database	1	1	1	1	1m
deploy/ui	1	1	1	1	1m

NAME	DESIRED	CURRENT	READY	AGE
rs/api-5c677587f5	1	1	1	1m
rs/database-5d5cb5488f	1	1	1	1m
rs/ui-65d77678c9	1	1	1	1m

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----



po/api-5c677587f5-26wdl	1/1	Running	0	1m
po/database-5d5cb5488f-zs6pm	1/1	Running	0	1m
po/ui-65d77678c9-dl6fg	1/1	Running	0	1m

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/api	ClusterIP	10.101.203.18	<none>	8080/TCP	1m
svc/database	ClusterIP	10.106.109.107	<none>	5432/TCP	1m
svc/ui	NodePort	10.102.15.64	<none>	3000:32030/TCP	1m

Note the NodePort line above. It gives us the public port through which we can access the UI of the application. In my case it is port 32030. It has been automatically assigned by Kubernetes.

4. Open a browser window and navigate to `http://<public IP>:<NodePort>/pet`, where <public IP> is a reachable IP for whatever host you're running your app on (the public IP for the `elk` node, for example, or `127.0.0.1` if you're running locally). You should see a nice cat GIF.
5. Refresh the browser a few times to load additional cat GIFs.
6. Delete the application from Minikube:

```
$ kubectl delete namespace/ddev
```

## 1.3 Exploring the application

Open the application with your favorite editor. You should see this structure:

```
| -api
| -database
| -devsecrets
| -ui
| -.dockerignore
| -docker-compose-api.yml
| -docker-compose.yml
| -kube.yaml
| -tag-and-push.sh
```

We just used the `docker-compose.yml` file to build and run the application. Open the file and take a moment to analyze its content. Discuss your findings with your peers. Make sure you can explain what each and every key in the file is for. If any are unclear, have a look at the Compose file docs at <https://docs.docker.com/compose/compose-file/>, or ask the instructor.

Also open the `kube.yaml` file. We used it to run the application on Kubernetes. We have been using Minikube but it runs unmodified on any Kubernetes cluster. Make sure you can explain what each and every part in the file is for. If any are unclear, have a look at the Kubernetes documentation for Namespaces, Secrets, Deployments and Services.

### 1.3.1 The Database

The source code for the database service is in the subfolder `database`. Inside this folder we have the following 4 files

- `Dockerfile`
- `pg_hba.conf`
- `postgresql.conf`
- `docker-entrypoint-initdb.d/init-db.sql`

The latter file contains the SQL to initialize and seed the database with some data. The `*.conf` files contain data to configure the Postgres database and the `Dockerfile` contains the instructions on how to build the database image.

### 1.3.2 The API

The source code for the backend API written in Java is in the subfolder `api`. We're using SpringBoot to create a simple REST API that is serving as the basis for our backend in our sample application. The API serves a random URL from a predefined set. We're using Maven as our build lifecycle tool. Some files worth mentioning:

- `pom.xml`: this is the Maven project file containing all instruction on how to build (and run) the Java service.
- `Dockerfile`: this contains the declarative definition of a multi-step build of the Docker image for the API service.
- `src/main/resources/application.yml`: contains application configuration details
- `src/main/java/com/docker/ddev/DdevApp.java`: The start class of the Java service
- `src/main/java/com/docker/ddev/controller/PetController.java`: controller that handles HTTP GET requests at the URL `/api/pet`

### 1.3.3 The UI

All source code for the Node JS and Express JS based UI service is in the subfolder `ui`. It will use the API that we just described above as its backend. Let's look at the individual parts:

- `Dockerfile`: used to build the Docker image for this service. It is based on a slim version of the official Node image.
- `package.json`: The node definition file for the project.
- `src/server.js`: Contains all the server side code for the UI service
- `src/index.html`: Contains the Mustache template for the view to be served to the browser when requesting cat images

## 1.4 Conclusion

In this lab we have cloned and tested the workshop sample application consisting of a Java based backend that serves URLs to cat images and a frontend based on Node JS, Express JS and Mustache that uses the backend to retrieve image URLs and display them as a HTML page in a browser. The backend storage is provided by a Postgres database.

## 2 Edit and Continue

As developers we seek a frictionless development process. The introduction of containers should make things smoother and not add additional friction. This lab shows you how to establish an **edit and continue** workflow when running code in a container.

The project folder for this exercise is `~/ddev-labs/edit-and-continue`.

### 2.1 Modifying and re-running the app

1. Run the application:

```
$ cd ~/ddev-labs/edit-and-continue
$ docker-compose up --build
```

2. Open the browser at `localhost:3000`. You should see a greeting message **Pets Demo Application**.
3. Edit the `server.js` in the folder `ui/src` and change the message returned by the `/` [GET] request:

```
app.get('/',function(req,res){
    res.status(200).send('Pets Demo Application (changed...)');
});
```

Don't forget to save your change.

4. Refresh the browser. What do you see?

Right, you still see the original message “Pets Demo Application”. The code change is only on your host filesystem and not propagated into the container.

5. Stop the application by pressing CTRL-c in the terminal you started it in.

6. Re-build and run the application again:

```
$ docker-compose up --build
```

7. Test the change in the browser. Do you see the change? Note: You might need to do a hard refresh in the browser.

8. Stop the application by pressing CTRL-c, and clean up after it with:

```
$ docker-compose down
```

## 2.2 Mounting source code

The above procedure to see changes made in the application is a bit tedious. If we always need to stop the container, build the new image and run the container after each change we waste a lot of time and create a lot of frustration. Luckily we can do better.

1. Open the file `docker-compose.dev.yml` and note the `volumes:` entry for the `ui` service:

```
version: '3.1'
services:
  ...
  ui:
    build:
      context: ui
    image: ddev_ui
    volumes:
      - ./ui/src:/app/src
    ports:
      - "3000:3000"
    networks:
      - front-tier
  ...
```

This stanza indicates you are mounting the folder `./ui/src` from our host to the target folder `/app/src` inside the `ui` container.

2. Run the application using this new modified Docker compose file:

```
$ docker-compose -f docker-compose.dev.yml up --build
```

Note how we can explicitly define which compose file to use with the `-f` parameter.

3. Open the browser at `localhost:3000` and make sure you can see the changed message.

4. Now change the greeting message in `server.js` again and save.

5. Refresh the browser. Do you see the change? No. Why not?

6. Make sure that the change actually made it into the container:

1. Open another terminal and exec into the `ui` container: `docker container exec -it <container-id> sh`
2. Show the content of the file: `cat /app/src/server.js`
3. Make sure the changes are there
4. Type `exit` to exit the container

7. Stop the application by hitting CTRL-c.

## 2.3 Automatically restart the application

Apparently just mounting the source code into the container wasn't good enough. We also need to make sure that the application running inside the container monitors the filesystem for changes and restarts the application if any are discovered.

1. Open the file `edit-and-continue/ui/Dockerfile.dev` and note that (compared to `Dockerfile`) we have added the following line just after the `FROM` statement:

```
RUN npm install nodemon -g
```

2. Compared to `Dockerfile` we have also changed the last line in the file to look like this:

```
CMD nodemon src/server.js 0.0.0.0:3000
```

Instead of running the app using Node we're using nodemon. **If you're running this on a Windows machine**, please also add a `-L` flag after nodemon here, to get the desired behavior.

3. Modify the `docker-compose.dev.yml` file to use our new `Dockerfile.dev` to build the `ui` service:

```
version: '3.1'
services:
  ...
  ui:
    build:
      context: ui
      dockerfile: Dockerfile.dev
    image: ddev_ui
    volumes:
      - ./ui/src:/app/src
    ports:
      - "3000:3000"
    networks:
      - front-tier
  ...
```

Note the `dockerfile:` entry in the `build:` block of the `ui` service.

4. Build and run the application using the modified `Dockerfile.dev`:

```
$ docker-compose -f docker-compose.dev.yml up --build
```

5. Now change the message in `server.js` again. As you save your changes you should observe output similar to the following in your terminal:

```
ui_1 | [nodemon] restarting due to changes...
ui_1 | [nodemon] starting `node src/server.js 0.0.0.0:3000`
ui_1 | Listening at 0.0.0.0:3000
```

Nodemon has detected the changes and automatically restarted the node server.

6. Make sure you can see the changes without restarting the application by browsing to `localhost:3000`.

## 2.4 Cleaning up

Do not forget to clean up after yourself when you're done with the exercise.

1. Use Docker compose to stop the application and remove all unused resources.

```
$ docker-compose -f docker-compose.dev.yml down
```

## 2.5 Optional Challenge

If you finish early and like to experiment, try to achieve similar edit-and-continue behavior in the following scenarios:

- Do a similar exercise for a static website consisting of a single file `index.html` with a simple greeting message. Use a simple web server to serve the page such as `http-server`. For this you don't need Docker compose and can directly run the (single) container using `docker container run...`
- Do a similar exercise for a Python/Flex or a .NET Core web application.

## 2.6 Conclusion

In this lab we have learned how to use volume mapping together with an auto-start tool to achieve an **edit and continue** experience for an application running in a container. Without these techniques we would have to rebuild images and restart our containers (or applications) each time we modified our code.

Note that this technique works out of the box with interpreted languages & frameworks such as Node JS, Python, Ruby etc., or languages that compile just-in-time like ASP.NET Core. With languages like Java using Spring Boot a similar technique is also possible although a bit more involved. Please refer to the following link for additional details: <https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-devtools.html>.

## 3 Debugging in a Container

In this exercise we are going to explore how applications can be debugged line by line even when running in a container. The project folder for this exercise is `~/ddev-labs/debugging`.

### 3.1 Debug the Java API

For this you need an IDE capable of remote debugging Java applications. In this sample we'll use Eclipse.

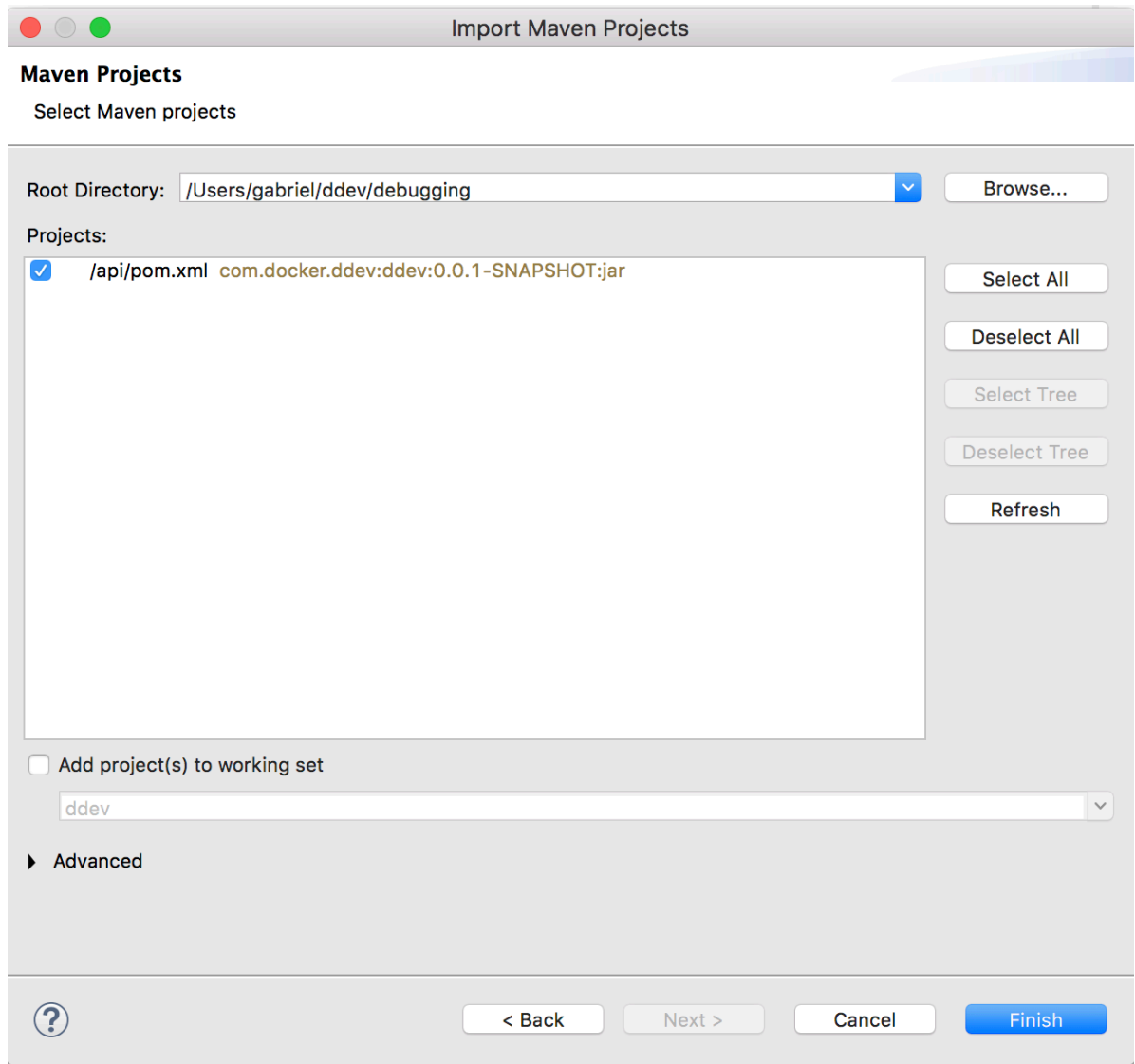
1. Eclipse is pre-installed on your remote desktop; if you'd like to try this exercise on your own machine and don't already have Eclipse, download it from <https://www.eclipse.org/downloads/> and install it.
2. Run the application with:

```
$ docker-compose -f docker-compose-api.yml up --build
```

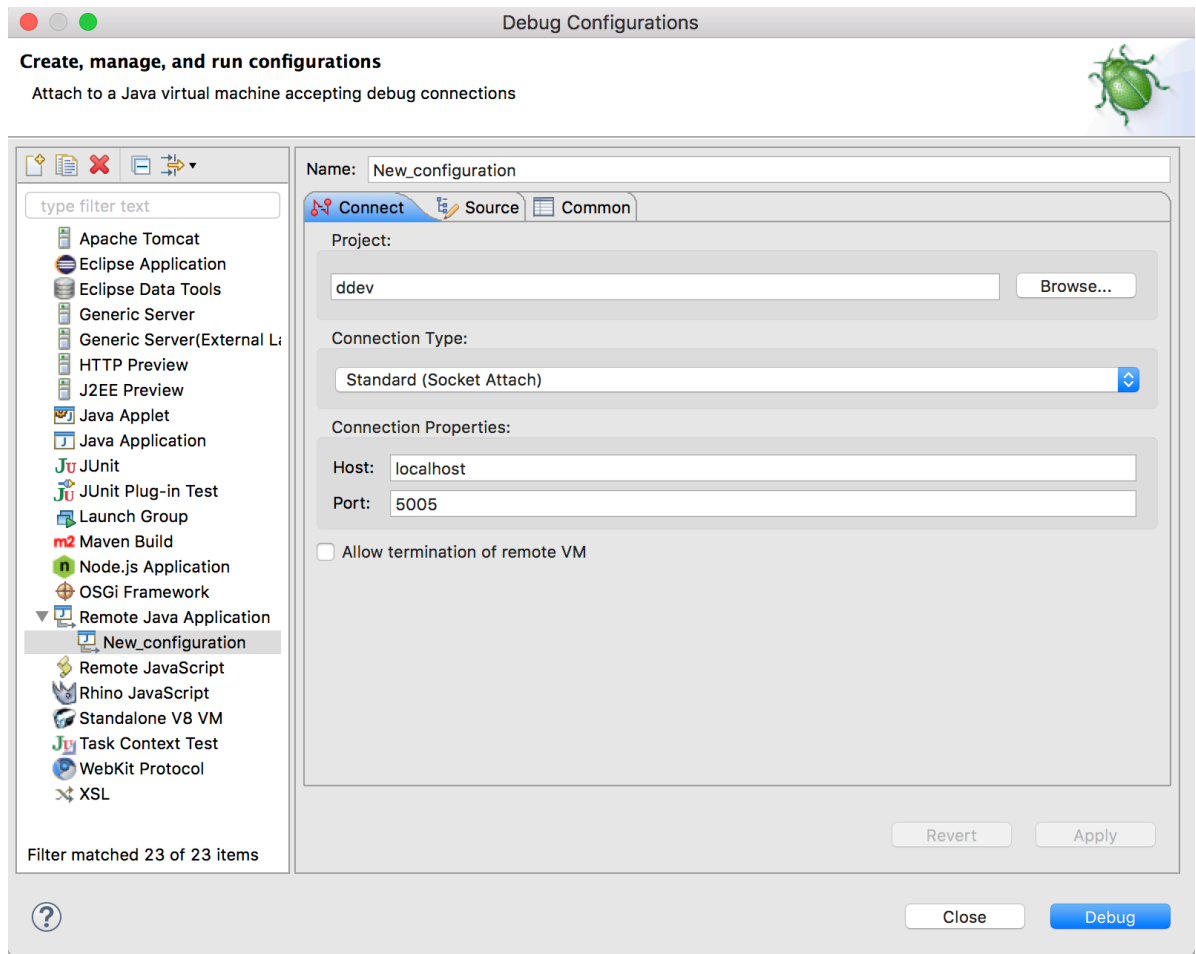
Note how we're using the `docker-compose-api.yml` file to describe the app which in turn uses `api/Dockerfile-dev` to build the API image. This Dockerfile contains a special `ENTRYPOINT` to configure Java and SpringBoot for debugging:

```
ENTRYPOINT ["java", \  
    "-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005", "-jar", \  
    "/app/ddev-0.0.1-SNAPSHOT.jar"]
```

3. Import the Java/Maven project to Eclipse: select **File->Import** and then select **Maven->Existing Maven Project** and hit **Next**.
4. In the "Import Maven Projects" dialog box navigate to the folder `~/ddev/debugging` and click **OK**. Make sure the project `/api/pom.xml` is selected and click **Finish**.



5. Create a new **Debug configuration** by selecting **Run->Debug Configurations...**. In the dialog box "Debug Configurations\*\* select **Remote Java Application** and click the **New launch configuration** button (white rectangular icon with a tiny yellow + above the list containing 'Remote Java Application') to create such a new configuration.
6. Fill in the fields as follows:



7. Click **Apply** and then **Debug**. Notice in the terminal how the Java application gets initialized. You should see something along the line of this (shortened for brevity):

```

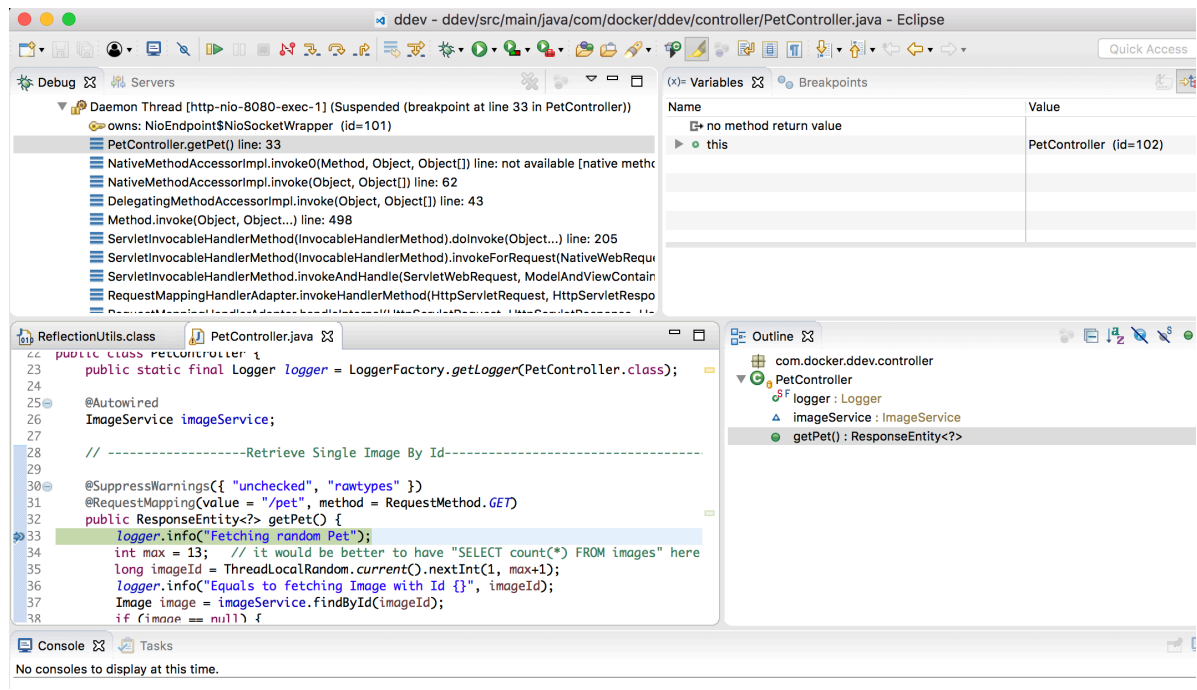
apiserver_1 |
apiserver_1 |
apiserver_1 |  . ____ _ _ _
apiserver_1 |  / \ / ____ \ (_ _ _ \ \ \ \
apiserver_1 |  ( ( )\___ | ' | ' | ' \ \ \ \
apiserver_1 |  \ \ / ____ | | | | | | | | ( | | ) ) )
apiserver_1 |  ' | ____ | . _ | | | | | \__, | / / / /
apiserver_1 |  =====|_|=====|__/_/ _/_/_/
apiserver_1 |  :: Spring Boot ::                (v1.5.3.RELEASE)
apiserver_1 |
apiserver_1 | 2017-07-06 20:34:05.836 INFO 1 --- [           main] com.docker.ddev.DdevApp
apiserver_1 | 2017-07-06 20:34:05.847 INFO 1 --- [           main] com.docker.ddev.DdevApp
...
apiserver_1 | 2017-07-06 20:34:22.005 INFO 1 --- [           main] o.s.j.e.a.AnnotationMBeanExpor
apiserver_1 | 2017-07-06 20:34:22.007 INFO 1 --- [           main] o.s.j.e.a.AnnotationMBeanExpor
apiserver_1 | 2017-07-06 20:34:22.016 INFO 1 --- [           main] o.s.j.e.a.AnnotationMBeanExpor
apiserver_1 | 2017-07-06 20:34:22.164 INFO 1 --- [           main] s.b.c.e.t.TomcatEmbeddedServe
apiserver_1 | 2017-07-06 20:34:22.180 INFO 1 --- [           main] com.docker.ddev.DdevApp

```

8. Navigate **Window -> Show View -> Navigator**. Set a breakpoint in the class `api/src/main/java/com/docker/ddev/contr` at the line `logger.info("Fetching random Pet");` by doubleclicking on the corresponding line number.
9. In a terminal window use `curl` to target the endpoint `/api/pet`:

```
$ curl localhost:8080/api/pet
```

Notice how the breakpoint in Eclipse is hit and we're able to debug the code line by line now. (You may be asked to open a new view in Eclipse - go ahead and click Yes, and it'll take you to the debug terminal).



- When done exit the application with CTRL-c and remove it with:

```
$ docker-compose -f docker-compose.debug.yml down
```

## 3.2 Optional Bonus: Debugging a Node JS application

In this exercise we are going to configure our ui service so that we can debug it line by line inside the container using **Visual Studio Code** (go ahead and install this on your machine or your remote desktop if necessary). Note that other editors might also support line by line debugging of Node applications in containers.

Note: for this exercise to work you need Node installed on your development machine. We're using version 6.9.4.

- Navigate to the project folder:

```
$ cd ~/ddev-labs/debugging
```

- Compare the two files ui/Dockerfile and ui/Dockerfile.debug and note the difference in the first and last two lines of the latter:

```
FROM node:6.9.4-alpine
...
EXPOSE 3000 5858
CMD node --inspect=5858 src/server.js
```

Note:

- how we're using an older version of Node here for debugging. For some reason debugging with Node 8 does not work as expected at this time.
- we're opening port 5858 for the debugger
- we're instructing Node to run in debug mode using the `--inspect` flag



3. Open the file `docker-compose.debug.yml` and modify the section for the `ui` service such as that it looks like this:

```
...
ui:
  build:
    context: ui
    dockerfile: Dockerfile.debug
  image: ddev_ui
  ports:
    - "3000:3000"
    - "5858:5858"
  networks:
    - front-tier
...
```

what exactly changed (compared to the original `docker-compose.yml`) and why?

4. Run the application by using this command:

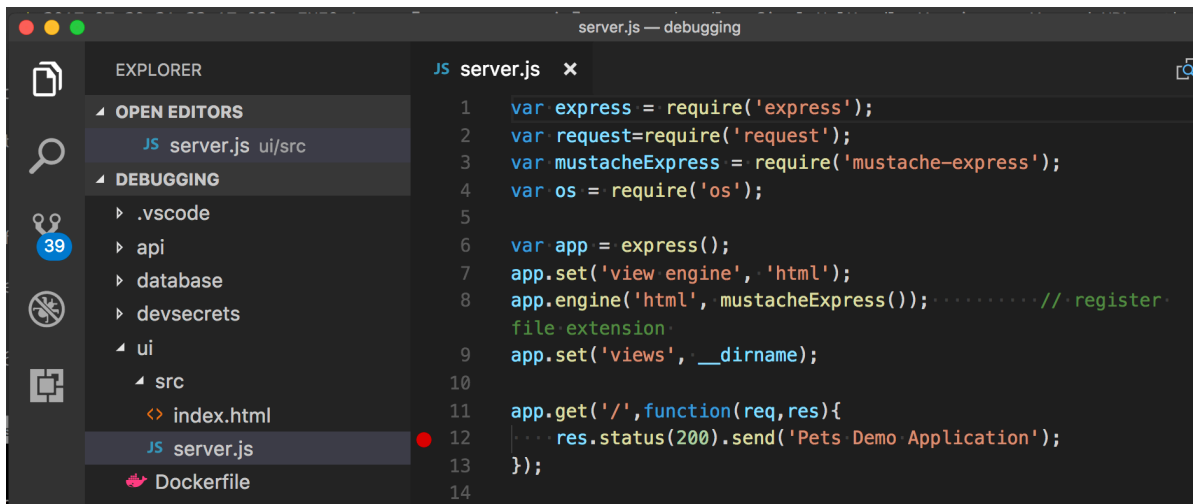
```
$ docker-compose -f docker-compose.debug.yml up --build
```

5. Note that in order to configure VS Code such as that it can attach to the node application running in the container, we had to do the following:

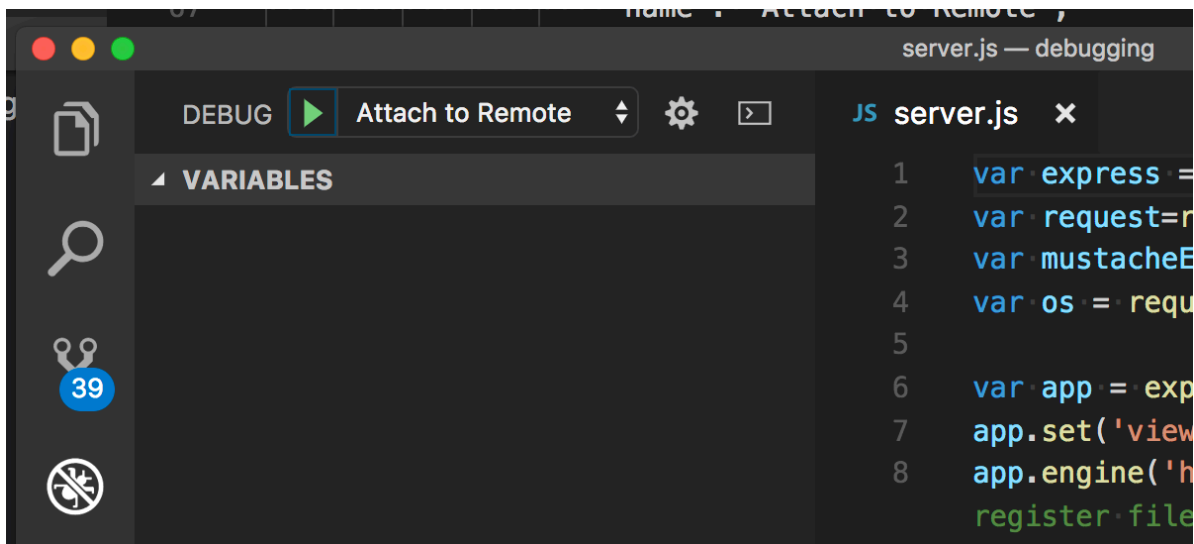
1. Add a folder `.vscode` to the project folder
2. Add a file `launch.json` to this folder
3. Add the following content to this file:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "attach",
      "name": "Attach to Remote",
      "address": "127.0.0.1",
      "port": 5858,
      "localRoot": "${workspaceRoot}/ui",
      "remoteRoot": "/app"
    }
  ]
}
```

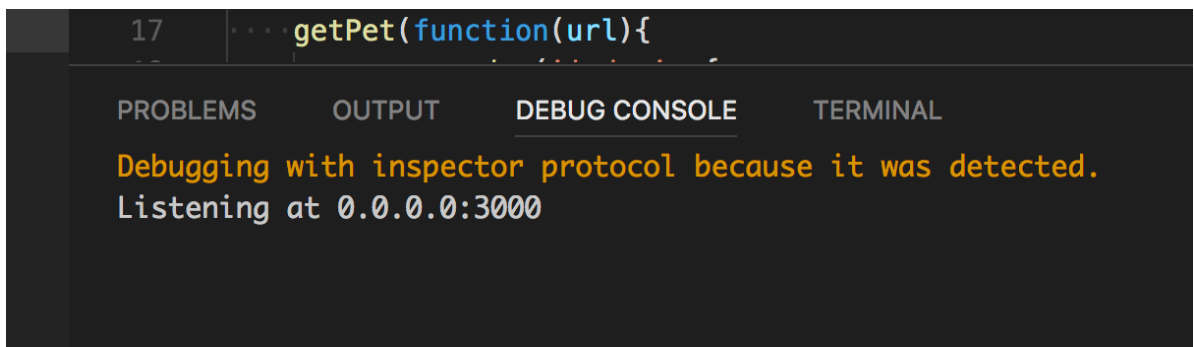
6. Open the folder `~/ddev-labs/debugging` in VS Code.
7. Set a break point in the file `ui/src/server.js` where the output message is generated (line 12).



8. Select **Debug** icon in the left hand toolbar in VS Code and then select the **Attach to Remote** launch configuration. Finally hit the green arrow next to **DEBUG**.



you should see the following message in the **DEBUG CONSOLE** of VS Code:



9. In the browser navigate to localhost:3000 and notice that code execution is stopped at the breakpoint in ui/src/server.js.
10. Use the navigation buttons in the debug toolbar to proceed past the breakpoint.
11. When done exit the application with CTRL-c and remove it with:

```
$ docker-compose -f docker-compose.debug.yml down
```

### 3.3 Conclusion

In this lab we have demonstrated line by line debugging of code running in a container. For this lab we have used **Eclipse** and **Visual Studio Code**, but other IDEs such as Visual Studio 2015/2017, IntelliJ or PyCharm offer similar possibilities.

## 4 Docker Compose

When developing different parts of an application, you'll often want different Docker Compose files that describe different configurations or priorities for developing each part of the project. In this exercise, we'll customize our Docker Compose file twice: once for backend development, and once for frontend development.

The project folder for this exercise is `~/ddev-labs/docker-compose`.

### 4.1 Back-end Development

When working on data and API tiers of an application, we:

- don't care about the frontend, and leave it off. We can use things like curl or Postman to hit API endpoints without a fully-featured frontend.
- still want edit and continue cycles like always.

Auto-reloading a Spring Boot based Java API is a bit more involved than the Node JS API we set up in a previous exercise. Here we're going to implement one possible solution using the Spring Boot devtools, detailed here: <https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-devtools.html>.

1. Change the `api/pom.xml` file and add this block to the `<plugins>` node:

```
<plugin>
  <groupId>com.fizzed</groupId>
  <artifactId>fizzed-watcher-maven-plugin</artifactId>
  <version>1.0.6</version>
  <configuration>
    <touchFile>target/classes/watcher.txt</touchFile>
    <watches>
      <watch>
        <directory>src/main/java/com/docker/ddev</directory>
      </watch>
    </watches>
    <goals>
      <goal>package</goal>
    </goals>
  </configuration>
</plugin>
```

Gist: <http://bit.ly/2uZj09D>

This will use the `fizzed-watcher` Maven plugin to watch for changes in any file in the folder `src/main/java/com/docker/ddev` folder and in this case run the goal `package`.

For more info please refer to the following URL: <https://github.com/fizzed/maven-plugins>.

2. Create a file `api/Dockerfile.api` with this content:

```
FROM maven:3.5.0-jdk-8-alpine AS appserver
WORKDIR /usr/src/ddev
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests
EXPOSE 8080
CMD mvn spring-boot:run
```

Note how we're using the `spring-boot:run` goal to run the API in auto-reload mode. Any changes in `target/classes` will trigger a reload of the API.

3. Open the `docker-compose-api.yml` file, and replace the `api` service with:

```
apiserver:
  build:
    context: api
    dockerfile: Dockerfile.api
  image: ddev_api
  volumes:
    - ./api:/usr/src/ddev
  ports:
    - "8080:8080"
    - "5005:5005"
  networks:
    - back-tier
  secrets:
    - postgres_password
```

Here we're mounting our source code for edit-and-continue, and we aren't bothering with the `front-tier` network since we won't be running a UI container while working on the backend. Remove `front-tier` from the list of networks at the bottom of the compose file, too. Ports are exposed on the API service so we can hit them directly with `curl`.

4. In the same compose file, add another service that recompiles the project whenever a `*.java` file changes:

```
watcher:
  build:
    context: api
    dockerfile: Dockerfile.api
  volumes:
    - ./api:/usr/src/ddev
  command: mvn fizzed-watcher:run
  networks:
    - back-tier
```

The complete `docker-compose-api.yml` file should now look like this gist: <http://bit.ly/2h48tnT>

5. Change the `api/src/main/resources/application.yml` file and change the default profile to use the same settings as the `postgres` profile:

- Remove the default profile in the first part of the file as shown here:

```
...
spring:
  profiles: local, default
...
```

- and instead add it further down after `postgres` as shown here:

```
...
spring:
```

```
profiles: postgres, default
...
```

6. Run the application:

```
$ docker-compose -f docker-compose-api.yml up --build
```

7. Test the API using curl:

```
$ curl localhost:8080/api/pet
```

you should be served a URL for a cat image.

8. Change some of the API's java source; for example, in `api/src/main/java/com/docker/ddev/controller/PetController` try setting `int max = 1` instead of 13; in the window you started the compose app from, you should be able to see the watcher container successfully rebuild the API. Once it's done, hit `localhost:8080/api/pet` again and make sure you can see the change.

9. When done terminate the application with CTRL-c and remove it:

```
$ docker-compose -f docker-compose-api.yml down
```

## 4.2 Front-end Development

In the case of front-end development, we want:

- A fixed backend; since (of course) your backend and frontend code is loosely coupled, there is no need to rebuild the database and API every time. Instead we will use fixed images for both.
- edit and continue cycles for our frontend code

1. From the `ddev-labs/docker-compose` directory, build the images for the API and the database:

```
$ docker image build -t pets-api:1.0 api
$ docker image build -t pets-db:1.0 database
```

2. Since we need nodemon we create a special Dockerfile for our UI only used during development. Add a file `Dockerfile-dev` to the `ui` folder with this content:

```
FROM node:8-alpine

# This is only needed during development
# to auto-reload the app on src change
RUN npm install -g nodemon

RUN mkdir /app
WORKDIR /app
COPY package.json /app/
RUN npm install
COPY ./src /app/src
EXPOSE 3000
CMD nodemon src/server.js -e html,js 0.0.0.0 3000
```

The only difference between the `Dockerfile` and the `Dockerfile-dev` is the installation instruction for nodemon which is used to auto-reload the app in case of code changes and the `CMD` which starts nodemon instead of node. **If you're trying this on a Windows machine**, add the `-L` flag after nodemon to poll for changes in legacy mode to see the desired behavior.

3. Add a file `docker-compose/docker-compose-ui.yml` with this content:

```
version: "3.1"

services:
```

```

database:
  image: pets-db:1.0
  environment:
    POSTGRES_USER: gordonuser
    POSTGRES_DB: ddev
  ports:
    - "5432:5432"
  networks:
    - back-tier
  secrets:
    - postgres_password

api:
  image: pets-api:1.0
  ports:
    - "8080:8080"
  networks:
    - front-tier
    - back-tier
  secrets:
    - postgres_password

ui:
  build:
    context: ui
    dockerfile: Dockerfile-dev
  image: ddev_ui
  volumes:
    - ./ui/src:/app/src
    - ./ui/package.json:/app/package.json
  ports:
    - "3000:3000"
  networks:
    - front-tier

secrets:
  postgres_password:
    file: ./devsecrets/postgres_password

networks:
  front-tier:
  back-tier:

```

Gist: <https://bit.ly/2J81JyT>

Note that we use a pre-built image of the backend and we achieve edit-and-continue experience through:

- Volume mapping of our source code directory into the container
- Choosing a CMD so that we run nodemon which auto-reloads the UI in case of file changes

#### 4. Run the application:

```
$ docker-compose -f docker-compose-ui.yml up --build
```

5. Test the application by opening a browser window and navigating to <development machine public IP>:3000/pet. You should see a cat image. Refresh a few times and make sure you get served different images.
6. Change some UI code, e.g. change the background color in ui/src/index.html to darkblue and make sure

the application auto-reloads (visible in the console) and when refreshing the browser you get the new color displayed.

7. When done stop the application with CTRL-c and remove it:

```
$ docker-compose -f docker-compose-ui.yml down
```

## 4.3 Conclusion

Application configuration is often best captured in a docker-compose file; as such, we will often write different compose files for different environments or even different parts of the development cycle. As we saw above, some common case-specific modifications are:

- automatic re-compile tools during development (similarly, automatic re-test tools during testing)
- port exposures for hitting APIs under development (would never do this in production)
- volume mounts for edit-and-continue experience
- fixing other loosely coupled components while we iterate on the code of interest

We also saw some cases where we used specialized Dockerfiles that added extra tooling for the development process. One common pattern here is to use a multi-stage build, where the first stage is a fully featured dev environment and the second is a minimal production environment. Another is to go the other way: build a minimal production environment image, and use that as the base image when constructing a development image with more tooling. Which is preferable depends on whether tooling (usually compilers or SDKs) is needed inside the image to build the production binaries; if so, we'll usually go the multi-stage build route, so we can build our binaries and then cherry-pick them in the final production image. If we're building containers for purely interpreted code, it can be easier to create a minimal production image right away, and then base development images with extra tooling on top of these.

## 5 Unit Tests

Unit tests are by their nature tightly coupled to the code under test. Thus they need to run in the same container as the code we want to test.

The project folder for this exercise is `~/ddev-labs/unit-tests`.

### 5.1 Extending the Node JS UI

We want to add some functionality to our Node based UI for which we can write unit tests. We also want to add some helper code to smooth the unit-test experience.

1. Open `ui/src/server.js` in your editor and replace the last two lines:

```
app.listen(3000, '0.0.0.0');
console.log("Listening at 0.0.0.0:3000");
```

with this:

```
var server = app.listen(3000, '0.0.0.0', function(){
  console.log("Express server listening on 0.0.0.0:3000");
});
```

2. Add a function `stop` to the `exports` object as follows:

```
exports.stop = function(){
  server.close();
}
```

This is a pure helper function that will be used later by our unit-test code.

3. Add a file `ui/src/primes.js` which contains the definition of the prime numbers algorithm:

```
exports.isPrime = function(number){
  for(var i = 2; i < number/2; i++) {
    if(number % i === 0) {
      return false;
    }
  }
  return number > 1;
}
```

4. Add an endpoint to the ui/src/server.js file which uses the above function isPrime to test for prime numbers:

```
app.get('/isPrime/:number', function(req, res){
  res.status(200).send(primes.isPrime(req.params.number));
});
```

and also add an import statement at the top of the file:

```
var primes = require('./primes.js');
```

5. Build the image for this app:

```
$ cd ~/ddev-labs/unit-tests/ui
$ docker image build -t pets-ui .
```

6. Run the app:

```
$ docker container run --rm -it -p 3000:3000 pets-ui
```

7. In another terminal window use curl to find out whether 7 is a prime number:

```
$ curl localhost:3000/isPrime/7
```

What does it say?

8. Stop the container by pressing CTRL-c.

## 5.2 Adding Unit Tests

1. In the folder unit-tests/ui we have added a subfolder specs.
2. Open the file ui/specs/primes-spec.js and add this content:

```
var sut = require('../src/primes.js');
describe("when evaluating if a given number is a prime", function(){
  describe("when the number is a prime", function(){
    it("should return 'true'", function(){
      expect(sut.isPrime(7)).toBe(true);
    });
  });
});
```

We're using Jasmin (<https://jasmine.github.io/>) as our test framework. The variable name sut stands for **system under test** and is often used to refer to the component or functionality to be teste.

3. Open the file ui/Dockerfile-unittests and note that (compared to ui/Dockerfile) we have added the following:

```
RUN npm install -g jasmine-node
```

as the second line. This installs jasmine-node which we need to be able to execute the tests.

4. We also added a line to copy the specs into the image (line 8):



```
COPY ./specs /app/specs
```

- Open the file `ui/package.json` and note that we extended the **script** block with a new **test** script:

```
...
"scripts": {
  "start": "node src/server.js",
  "test": "jasmine-node specs"
},
...
```

- Build the image:

```
$ cd ~/ddev-labs/unit-tests/ui
$ docker image build -t pets-ui-unittests -f Dockerfile-unittests .
```

- Run the container with the instruction to execute the unit tests instead of just running the app:

```
$ docker container run --rm -it pets-ui-unittests npm run test
```

you should see something along the line of:

```
$ docker container run --rm -it pets-ui-unittests npm run test

> pets@1.0.0 test /app
> jasmine-node specs

.

Finished in 0.008 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

## 5.3 Not Shipping Unit Tests

In the example above, we built our unit test code right into our image. Of course we don't want testing code (or tooling) going into our production images; in what follows we explore how to avoid this.

### 5.3.1 Mounting test code

One options is to mount test code during the testing phase, rather than including it in our image. To make things simpler we need to adhere to some standards in where we put our files containing test code and how we name them. In our case tests will always reside in a (sub-) folder called `specs`.

- In the `ui` folder create a file `.dockerignore` with the following content:

```
node_modules
specs
```

This will instruct Docker to not include folders named `node_modules` and `specs` (including all their content) in the context when building the image.

- Change the two copy statements that copy the `src` and `specs` folders in `Dockerfile-unittests` into a single one like this:

```
COPY . /app
```

- Build the image.
- Run an interactive container based on this image:

```
$ docker container run --rm -it pets-ui-unittests sh
```

5. While inside the container make sure the specs folder was not copied into the container, despite the `COPY . /app` command you added to your Dockerfile; the `.dockerignore` rule prevented specs from being copied in.
6. Quit the container via `exit`.
7. Now run the tests by mounting them into the container at runtime:

```
$ docker container run --rm -it \
  -v $(pwd)/specs:/app/specs \
  pets-ui-unittests npm run test
```

The tests should execute and produce the same output as in the previous exercise.

## 5.4 Creating a Test Image

If we also want to avoid that our production image contains any libraries that are only needed for testing (e.g. `jasmine-node` in our case) then we can build a special test image that inherits from the production image.

1. Build the production image, based on `ui/Dockerfile`:

```
$ docker image build -t pets-ui .
```

2. Modify our `Dockerfile-unittests` to look like this:

```
FROM pets-ui
RUN npm install jasmine-node -g
CMD ["npm", "run", "test"]
```

3. Build the test image using this new Dockerfile (note the `-f` parameter):

```
$ docker image build -t pets-ui-unittests -f Dockerfile-unittests .
```

4. Now run an instance of this image to execute the tests (don't forget to mount the volume with the tests):

```
$ docker container run --rm -it \
  -v $(pwd)/specs:/app/specs \
  pets-ui-unittests
```

We have the production image `pets-ui` which only contains production code and dependencies and a separate image `pets-ui-unittests` which inherits from the former and additionally contains the necessary dependencies that are needed to execute tests. We also re-defined the `CMD` to execute `npm run test` when running a container.

## 5.5 Edit and Continue when writing tests

Currently we have to re-run the test container each time we modify code or add a new test. To improve the process a bit we can run the container with a slightly modified entrypoint to configure `jasmine-node` to auto-test on change (see <https://github.com/mhevery/jasmine-node> for details):

1. Add a `test-watch` script to the 'package.json' file:

```
...
"scripts": {
  "start": "node src/server.js",
  "test-watch": "jasmine-node specs --autotest --watch ./**/*.js",
  "test": "jasmine-node specs"
},
...
```

2. Rebuild the container:

```
$ docker image build -t pets-ui-unittests -f Dockerfile-unittests .
```

- Execute the test container with the test-watch script:

```
$ docker container run --rm -it \
  -v $(pwd):/app \
  pets-ui-unittests npm run test-watch
```

Note that since test-watch is not a standard script we need to run it with the command `npm run test-watch` and not just `npm test-watch`.

The output should now look like this:

```
$ docker container run --rm -it \
> -v $(pwd):/app \
> pets-ui-unittests npm run test-watch

> pets@1.0.0 test-watch /app
> jasmine-node specs --autotest --watch ./**/*.js

.

Finished in 0.007 seconds
1 test, 1 assertion, 0 failures, 0 skipped

Watching for changes in ./specs/primes-spec.js
Watching for changes in ./src/primes.js
Watching for changes in ./src/server.js
Watching for changes in /app/specs
```

- Now let the container run and add a new test to the file `primes-spec.js`, just after the first one, which makes sure that if we provide a non-prime number the answer will be false. The result should look like this:

```
var sut = require('../src/primes.js');
describe("when evaluating if a given number is a prime", function(){
  describe("when the number is a prime", function(){
    it("should return 'true'", function(){
      expect(sut.isPrime(11)).toBe(true);
    });
  });
  describe("when the number is a not prime", function(){
    it("should return 'false'", function(){
      expect(sut.isPrime(4)).toBe(false);
    });
  });
});
```

- Notice that immediately after you saved the above changes the tests were executed again - but this one failed!
- Apparently there is a bug in `primes.js`. Find it, fix it, and save the fix.
- Notice that again upon saving the file the tests are re-executed, this time hopefully all passing.

## 5.6 Optional Challenge: Unit Testing a .NET Core Application

This exercise is optional; give it a try if you have some extra time after finishing the examples above.

### 5.6.1 Creating a .NET Core application

In the following few exercises we're going to create a simple .NET Core service with unit tests and containerize them similarly to what we have done for the Node Express application above.

1. Create a folder ~/ddev-labs/unit-test-dot-net.

2. Inside this folder run the following command:

```
$ docker container run --rm -it -v $(pwd):/app -w /app microsoft/dotnet bash
```

3. Inside the container create a .NET class library as follows:

```
$ mkdir PrimeService
$ cd PrimeService
$ dotnet new classlib
```

Notice that this creates a folder PrimeService containing files Class1.cs and PrimeService.csproj also on the host since we have volume mounted the host directory.

4. In the host file system rename the file Class1.cs to PrimeService.cs.

5. Replace its content with the following one:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first");
        }
    }
}
```

6. Back in the running .NET container create a folder /app/PrimeService.Tests and navigate into this folder:

```
$ mkdir /app/PrimeService.Tests
$ cd /app/PrimeService.Tests
```

7. Make a new (test) project in this folder:

```
$ dotnet new classlib
```

This should have added two files Class1.cs and PrimeService.Tests.csproj to the folder.

8. Now we need to add the PrimeService project as a reference to the test project:

```
$ dotnet add reference ../PrimeService/PrimeService.csproj
```

9. Open the file PrimeService.Tests.csproj and change it to look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="xunit" Version="2.3.0-beta2-build3683" />
    <DotNetCliToolReference Include="dotnet-xunit" Version="2.3.0-beta2-build3683" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\PrimeService\PrimeService.csproj" />
  </ItemGroup>
</Project>
```

Note: Please refer to this link for more info: <https://xunit.github.io/docs/getting-started-dotnet-core>

10. Rename the class `Class1.cs` to `PrimeService_IsPrimeShould.cs` and add the following content:

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Fact]
        public void ReturnFalseGivenValueOf1()
        {
            var result = _primeService.IsPrime(1);

            Assert.False(result, $"1 should not be prime");
        }
    }
}
```

Gist: <http://bit.ly/2v9B3ua>

11. Back in the container run the restore command to download all dependencies:

```
$ dotnet restore
```

The output should look like this:

```
Restoring packages for /app/PrimeService.Tests/PrimeService.Tests.csproj...
Restoring packages for /app/PrimeService/PrimeService.csproj...
Lock file has not changed. Skipping lock file write. Path: /app/PrimeService/obj/project.assets.json
Restore completed in 199.38 ms for /app/PrimeService/PrimeService.csproj.
Restoring packages for /app/PrimeService.Tests/PrimeService.Tests.csproj...
Restore completed in 184.76 ms for /app/PrimeService.Tests/PrimeService.Tests.csproj.
Generating MSBuild file /app/PrimeService.Tests/obj/PrimeService.Tests.csproj.nuget.g.props.
Generating MSBuild file /app/PrimeService.Tests/obj/PrimeService.Tests.csproj.nuget.g.targets.
Writing lock file to disk. Path: /app/PrimeService.Tests/obj/project.assets.json
Restore completed in 536.08 ms for /app/PrimeService.Tests/PrimeService.Tests.csproj.

NuGet Config files used:
  /root/.nuget/NuGet/NuGet.Config

Feeds used:
  https://api.nuget.org/v3/index.json

Installed:
  20 package(s) to /app/PrimeService/PrimeService.Tests/PrimeService.Tests.csproj
```

12. Finally we can try to run the tests:

```
$ dotnet xunit
```

This should result in a failure with a message [snippet]:

```
...
xUnit.net Console Runner (64-bit .NET Core 4.6.25211.01)
Discovering: PrimeService.Tests
Discovered: PrimeService.Tests
Starting: PrimeService.Tests
    Prime.UnitTests.Services.PrimeService_IsPrimeShould.ReturnFalseGivenValueOf1 [FAIL]
    System.NotImplementedException : Please create a test first
...
```

13. Implement code in the class `PrimeService.cs` to handle the test correctly:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first");
}
```

14. Run the tests again. This time it should succeed. Here's a snippet from the output:

```
...
xUnit.net Console Runner (64-bit .NET Core 4.6.25211.01)
    Discovering: PrimeService.Tests
    Discovered: PrimeService.Tests
    Starting: PrimeService.Tests
    Finished: PrimeService.Tests
=== TEST EXECUTION SUMMARY ===
    PrimeService.Tests Total: 1, Errors: 0, Failed: 0, Skipped: 0, Time: 0.162s
```

15. Exit the .NET container by entering `exit` in the shell.

### 5.6.2 Containerizing the .NET Application

Now that we have interactively created the code for our .NET project we want to containerize it.

1. Add a `Dockerfile` to the project folder `unit-test-dot-net` with this content:

```
FROM microsoft/dotnet
RUN mkdir -p /app
COPY ./PrimeService/PrimeService.csproj /app/PrimeService/
COPY ./PrimeService.Tests/PrimeService.Tests.csproj /app/PrimeService.Tests/
WORKDIR /app/PrimeService.Tests
RUN dotnet restore
COPY . /app
ENTRYPOINT dotnet xunit
```

2. Build the app:

```
$ docker image build -t dot-net-app .
```

3. And run a container from this image:

```
$ docker container run --rm -it dot-net-app
```

The tests should run and the output should be the same as when we ran the tests interactively in the .NET container earlier on.

### 5.6.3 Adding Edit and Continue Experience

Instead of always restarting the container when we have added or changed some code we can do better and use the **watcher** tool provided by Microsoft.

1. Add the following snippet to the file `PrimeService.Tests.csproj`:

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="1.0.0" />
</ItemGroup>
```

2. Change the `ENTRYPOINT` in the `Dockerfile` as follows:

```
ENTRYPOINT dotnet watch xunit
```

3. Build the image.

4. Run the container using volume mounting:

```
$ docker container run --rm -it -v $(pwd):/app dot-net-app
```

we should see the following output:

```
bash watch : Started Detecting target frameworks in PrimeService.Tests.csproj...
Building for framework netcoreapp1.1... PrimeService -> /app/PrimeService/bin/Debug/netstandard1.4/PrimeService.dll
PrimeService.Tests -> /app/PrimeService.Tests/bin/Debug/netcoreapp1.1/PrimeService.Tests.dll
Running .NET Core tests for framework netcoreapp1.1... xUnit.net Console Runner
(64-bit .NET Core 4.6.25211.01) Discovering: PrimeService.Tests Discovered:
PrimeService.Tests Starting: PrimeService.Tests Finished: PrimeService.Tests
=== TEST EXECUTION SUMMARY === PrimeService.Tests Total: 1, Errors: 0, Failed: 0,
Skipped: 0, Time: 0.160s
```

5. Add another test to the file `PrimeService_IsPrimeShould.cs` and save:

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void ReturnFalseGivenValuesLessThan2(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

6. Observe that the tests are run again upon saving of the changes. Of course the test will fail since our production code is not yet handling those scenarios.
7. Add the code to make the test succeed. In the class `PrimeService.cs` change the first line of the `IsPrime` function to:

```
if (candidate < 2)
```

After a short timeout the tests should rerun the tests. This time they should succeed (4 Passed).

### 5.6.4 Freeing the application image from test code

Once again, if we don't want to ship test code with the production image we can use the same technique that we used in the Node Express sample and create a special test image that inherits from the production image.

1. Move the `Dockerfile` into the subfolder `PrimeService`.
2. Change the `Dockerfile` to look like this:

```
FROM microsoft/dotnet
RUN mkdir -p /app/PrimeService
WORKDIR /app/PrimeService
COPY ./PrimeService.csproj /app/PrimeService/
RUN dotnet restore
COPY . /app/PrimeService
```

3. Make sure in the terminal you're in the project folder `unit-test-dot-net`.

4. Build the production image:

```
$ docker image build -t dot-net-app PrimeService
```

5. To the project folder `unit-test-dot-net` add a file `Dockerfile-unit` with this content:

```
FROM dot-net-app
RUN mkdir /app/PrimeService.Tests
WORKDIR /app/PrimeService.Tests
COPY ./PrimeService.Tests/PrimeService.Tests.csproj /app/PrimeService.Tests/
RUN dotnet restore
ENTRYPOINT dotnet watch xunit
```

6. Build the test image:

```
$ docker image build -t unit-test-dot-net-app -f Dockerfile-unit .
```

7. Run the test container:

```
$ docker container run --rm -it -v $(pwd):/app unit-test-dot-net-app
```

The 4 tests (we have so far) should be executed and pass.

8. Add some additional tests and make sure they automatically run when you save.

## 5.7 Conclusion

In this exercise we have learned how we can write unit tests against the UI part of our workshop sample. We also have discussed how we can package and run those tests.

We also have shown techniques on how to improve the workflow when writing code and authoring tests for it such as that a typical “edit and continue” experience is achieved.

Of course we cannot just write unit tests in JavaScript using e.g. Jasmin but in any of the popular languages like C#/.NET, Java or Python. The test code ideally is written in the same language as the code under test.

## 6 API Tests

In the following exercises we will use our simple Java API and then write some tests for this API. Unlike the unit tests we did previously, the API tests will run in their own container during execution.

The project folder for this exercise is `~/ddev-labs/api-tests`.

### 6.1 Adding Postgres as a backing DB

In this exercise we will execute tests against the Java API which in turn uses the Postgres database as a backing data store.

1. Build the API and database containers we want to test:



```
$ cd ~/ddev-labs/api-tests
$ docker image build -t ddev-api:1.0 api
$ docker image build -t ddev-db:1.0 database
```

2. From the project folder `api-tests` open the file `docker-compose-api.yml`. Note how we use pre-built images for the `api` and `database` services.

3. Now let's run the application:

```
$ docker-compose -f docker-compose-api.yml up
```

4. And test it:

```
$ curl localhost:8080/api/pet
```

which should return something along the lines of:

```
{ "imageId": 11, "url": "https://...jpg", "description": "cat image" }
```

5. Stop the application by hitting CTRL-C, and tidy up as usual:

```
$ docker-compose -f docker-compose-api.yml down
```

## 6.2 Writing API tests

Now that we have our API up and running, we want to write some **API tests** for it. For this we're going to use Node JS and Jasmine, but we could just as easily use any other language and framework we like, since API testing treats the API like a black box.

1. In the project folder `api-tests` we have added a folder `tests`.
2. From there open the file `package.json`. It should have this content:

```
{
  "name": "api-tests-java",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "jasmine-node *.js",
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.15.2",
    "jasmine-node": "^1.14.5",
    "request": "^2.81.0"
  }
}
```

Note especially the `test` key under `scripts`, indicating the script that will run our tests.

3. We also have a file `api-spec.js` in the same folder that contains the following (API) test, which expects to hit an API at `http://api:8080`. This is what we meant by testing the API like a black box: anything could be sitting behind that endpoint; we only care that it gives a response we expect:

```
var request = require('request');
var base_url = "http://api:8080/";

describe("When testing 'api/pet'", function(){
  it("should respond with the URL of a cat GIF", function(done) {
    request(base_url + 'api/pet', function(error, response, body){
      expect(body).toMatch(/\{"imageId":\d+,"url":"http:/);
      done();
    });
  });
});
```

```
    });
  });
});
```

4. And finally we have a Dockerfile in the tests folder with:

```
FROM node:8-alpine
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD npm run test
```

which we use to build the image for running our tests. Notice the CMD calling the test script defined above.

5. Finally, have a look at our application definition at `api-tests/docker-compose-api-tests.yml`:

```
version: '3.1'
services:
  tests:
    build: tests
    networks:
      - back-tier
networks:
  back-tier:
```

Note how we put the tests container on the same network as the api container (defined in `docker-compose-api.yml`) to enable service discovery.

6. Re-start the API, and wait until it is initialized:

```
$ cd ~/ddev-labs/api-tests
$ docker-compose -f docker-compose-api.yml up
```

7. Now, in another terminal run the API tests:

```
$ cd ~/ddev-labs/api-tests
$ docker-compose -f docker-compose-api-tests.yml up --build
```

you should see something like this:

```
...
tests_1 |
tests_1 | > api-tests-java@1.0.0 test /app
tests_1 | > jasmine-node *.js
tests_1 |
tests_1 | .
tests_1 |
tests_1 | Finished in 0.053 seconds
tests_1 | 1 test, 1 assertion, 0 failures, 0 skipped
tests_1 |
tests_1 |
apitests_tests_1 exited with code 0
```

8. Clean up:

```
$ docker-compose -f docker-compose-api.yml down
$ docker-compose -f docker-compose-api-tests.yml down
```

We have an API that is using a Postgres DB as storage backend and runs in containers. Then in another container we run our tests against that API.

## 6.3 Optional Challenge: Exercise with Python/Flask API backed by Mongo DB

This exercise is optional. It shows an alternative implementation of an API which can be tested using API tests.

1. In the workshop folder ~/ddev-labs create a project folder api-tests-flask and navigate to it:

```
$ cd ~/ddev
$ mkdir api-tests-flask
$ cd api-tests-flask
```

### 6.3.1 Creating a Python Flask API

In this exercise we'll create a simple Flask based API. The API will be backed by Mongo DB as a database.

1. Create a folder api as well as a folder tests in the api-tests-flask folder:

```
$ mkdir api
$ mkdir tests
```

2. Add a file api/requirements.txt with this content:

```
flask
```

3. Add a file api/main.py with this content:

```
import json
from flask import Flask, Response, request
app = Flask(__name__)

@app.route("/")
def description():
    return "Products API!"

products = [
    {"id": 1, "name": "Samsung Galaxy S8"},
    {"id": 2, "name": "Apple iPhone 6s"},
    {"id": 3, "name": "Google Nexus 6P"},
    {"id": 4, "name": "LG G6"}
]

@app.route("/products")
def list_products():
    return Response(json.dumps(products), mimetype='application/json')

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

4. Add a file api/Dockerfile which is used to build the API:

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . /app
EXPOSE 8000
ENTRYPOINT python main.py
```

5. Build the image for the API:

```
$ docker image build -t flask-api api
```

6. Run the API:

```
$ docker container run --rm -it -p 8000:8000 flask-api
```

7. Manually test the API:

```
$ curl localhost:8000/products
```

should return a response similar to this:

```
[{"id": 1, "name": "Samsung Galaxy S8"}, {"id": 2, "name": "Apple iPhone 6s"}, {"id": 3, "name": "G
```

8. Kill the container once you've seen it running correctly.

### 6.3.2 Adding Mongo DB as a backing DB

In this step we will add Mongo DB to the picture and extend the API such as that it uses Mongo DB as a backing data store.

1. Add flask\_pymongo as a dependency to the file api/requirements.txt.
2. Add the following import statements to api/main.py:

```
from bson.json_util import dumps
from flask_pymongo import PyMongo
```

3. Just after the line `app = Flask(__name__)` add the following lines to initialize a connection to the Mongo DB:

```
app.config['MONGO_HOST'] = 'mongo'
app.config['MONGO_PORT'] = 27017
app.config['MONGO_DBNAME'] = 'inventory'
app.config['MONGO_URI'] = 'mongodb://mongo:27017/inventory'
mongo = PyMongo(app)
```

4. Change the function `list_products` as follows:

```
@app.route("/products", methods=['GET'])
def list_products():
    products = mongo.db.products.find(
        {"$or": [
            {"discontinued":{"$exists":False}},
            {"discontinued":False}
        ]}
    )
    return Response(dumps(products), mimetype='application/json')
```

to load all the not discontinued products from Mongo DB.

5. Add a method `add_product` as follows:

```
@app.route("/products", methods=['POST'])
def add_product():
    prod = request.json
    prod_id = mongo.db.products.insert(prod)
    return Response(response=str(prod_id), status=201)
```

6. Create a file `api-tests-flask/docker-compose.yml` with the following content:

```
version: '3.1'
services:
```

```
api:
  build: api
  ports:
    - 8000:8000
  volumes:
    - ./api:/app
mongo:
  image: mongo:3.4
  volumes:
    - ./data:/data/db
  ports:
    - 27017:27017  # for debugging purposes only
```

Gist: <http://bit.ly/2uDKGxQ>

7. Run the application using Docker Compose:

```
$ docker-compose up --build
```

8. In a new terminal window test the application using curl:

- Add a few new products e.g.:

```
$ curl -X POST -H "Content-Type: application/json" -d \
'{"company": "Samsung", "name": "Galaxy S8", "price": 755.99}' localhost:8000/products
$ curl -X POST -H "Content-Type: application/json" -d \
'{"company": "Apple", "name": "iPhone 7S", "price": 795.05}' localhost:8000/products
$ curl -X POST -H "Content-Type: application/json" -d \
'{"company": "Google", "name": "Nexus 6P", "discontinued": false}' localhost:8000/products
$ curl -X POST -H "Content-Type: application/json" -d \
'{"company": "Google", "name": "Nexus 5", "discontinued": true}' localhost:8000/products
```

- List all products:

```
$ curl localhost:8000/products
```

9. Shut down the API services. This time, we'll stand them up again as part of our testing docker-compose app.

### 6.3.3 Writing API tests

Now that we have a working API we want to write some API tests for it. For this we're going to use Node JS and Jasmine.

1. Add a file tests/package.json with this content:

```
{
  "name": "api-tests-flask",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "jasmine-node *.js",
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.15.2",
    "jasmine-node": "^1.14.5",
    "request": "^2.81.0"
  }
}
```

Gist: <http://bit.ly/2tKNkEW>

2. Add a file `tests/api-spec.js` with this content:

```
var request = require('request');
var base_url = "http://api:8000/";

describe("When testing api root", function(){
  it("should respond with description", function(done) {
    request(base_url, function(error, response, body){
      expect(body).toEqual("Products API!");
      done();
    });
  });
});
```

3. Add a Dockerfile to the tests folder:

```
FROM node:8-alpine
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD npm run test
```

4. Add a file `api-tests-flask/docker-compose.api-tests.yml` with this content:

```
version: '3.1'
services:
  tests:
    build: tests

  api:
    build: api

  mongo:
    image: mongo:3.4
```

Gist: <http://bit.ly/2tEEbJR>

5. Run the tests with:

```
$ docker-compose -f docker-compose.api-tests.yml up -d --build
$ docker-compose -f docker-compose.api-tests.yml logs -f tests
```

you should see something like this:

```
...
tests_1 | > api-tests-flask@1.0.0 test /app
tests_1 | > jasmine-node *.js
tests_1 |
tests_1 | .
tests_1 |
tests_1 | Finished in 0.032 seconds
tests_1 | 1 test, 1 assertion, 0 failures, 0 skipped
tests_1 |
tests_1 |
tests_1 | npm info lifecycle api-tests-flask@1.0.0~posttest: api-tests-flask@1.0.0
tests_1 | npm info ok
apitestsflask_tests_1 exited with code 0
```

6. Add another test series to `api-spec.js`:

```

describe("When testing post to /products", function(){
    var prod_id;

    it("should respond with status 201 (CREATED)", function(done) {
        var options = {
            url: base_url + "products",
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                "company": "LG",
                "name": "G6",
                "price": 499.99
            }),
            dataType: 'json'
        }
        request.post(options, function(error, response, body){
            expect(response.statusCode).toEqual(201);
            prod_id = body;
            console.log("Prod ID: " + body);
            done();
        });
    });

    it("should have created a new product", function(done){
        request(base_url + "products", function(error, response, body){
            data = JSON.parse(body);
            expect(data.length).toBeGreaterThan(0);
            done();
        })
    });

    it("should have created a new product with correct id", function(done){
        request(base_url + "products", function(error, response, body){
            data = JSON.parse(body);
            var found = false;
            for(var i in data){
                if(data[i]._id["$oid"] === prod_id){
                    found = true;
                }
            }
            expect(found).toBe(true);
            done();
        })
    });
});

```

Gist: <http://bit.ly/2u0M7EO>

these tests make sure we can actually add a new product to the DB.

7. Build and run the test suite again. Now you should see 4 tests, 4 assertions, 0 failures, 0 skipped.
8. Clean up:

```
$ docker-compose -f docker-compose.api-tests.yml down
```

## 6.4 Conclusion

When testing containerized code, a key consideration is where to place your tests. In this exercise, we explored placing tests in a fully independent container from the containers we were interested in testing. This is usually an appropriate strategy for testing *behavior*, like the API behavior we examined here. Compare this to what we did in the last chapter, where we were interested in testing *logic* - that level of introspection was best done from within the container of interest.

## 7 End-to-End Tests

In this exercise we are going to test our workshop sample application as a whole, end-to-end. The tests will run in their own container during execution.

The project folder for this exercise is `~/ddev-labs/end-2-end-tests`.

### 7.1 Writing the Tests

1. Change the file `ui/src/index.html` by adding a `<title>` element to the `<head>`:

```
<html>
<head>
  <title>Cat of the Day</title>
  ...
</head>
...
```

we need this element to write our sample test against.

2. Notice the folder `end-2-end/e2e`. It will contain everything for the test container.
3. Open the file `e2e/server.js`. It contains simple code to run an end-to-end test:

```
var webdriver = require('selenium-webdriver'),
    By = webdriver.By,
    until = webdriver.until;

console.log('Selenium driver initialized.');
```

```
function test(){
  console.log('Now running test');
  var driver = new webdriver.Builder()
    .forBrowser('chrome')
    .usingServer('http://selenium:4444/wd/hub')
    .build();
  driver.get('http://ui:3000/pet');
  driver.wait(until.titleIs('Cat of the Day'), 1000);
  driver.quit();
}
```

```
console.log('Now waiting for 3sec');
setTimeout(test, 3000);
```

**Note:** we're waiting for 3 seconds to give Selenium a chance to start up. This could be done in a more sophisticated way, e.g. by polling `http://selenium:4444/wd/hub` until we get a response.

Also note that we're using service discovery provided by Docker and thus can access the UI with the URL `http://ui:3000/pet`.

4. Note there is also a file `e2e/package.json` since our tests are written in Node JS.



5. Add a Dockerfile to the 'e2e' folder with this content:

```
FROM node:8.0.0-alpine
RUN npm install selenium-webdriver
WORKDIR /app
COPY . .
CMD npm start
```

6. Create a file end-2-end-tests/docker-compose-e2e.yml (next to the other docker-compose files) with this content:

```
version: '3.1'

services:
  selenium:
    image: selenium/standalone-chrome:3.4.0-dysprosium
    volumes:
      - /dev/shm:/dev/shm
    networks:
      - front-tier
    ports:
      - 4444:4444
  e2e:
    build: e2e
    networks:
      - front-tier
    command: npm start

networks:
  front-tier:
```

Gist: <http://bit.ly/2ARmSYP>

Why do we connect the services to the front-tier network?

7. Run the application:

```
$ docker-compose up --build
```

and wait until the app is ready.

8. In another terminal window run the end-to-end tests:

```
$ docker-compose -f docker-compose-e2e.yml up --build
```

you should see an output similar to this (shortened):

```
...
e2e_1      | npm info it worked if it ends with ok
e2e_1      | npm info using npm@5.0.0
e2e_1      | npm info using node@v8.0.0
e2e_1      | npm info lifecycle e2etests@1.0.0~prestart: e2etests@1.0.0
e2e_1      | npm info lifecycle e2etests@1.0.0~start: e2etests@1.0.0
e2e_1      |
e2e_1      | > e2etests@1.0.0 start /app
e2e_1      | > node server.js
e2e_1      |
selenium_1 | 20:39:23.973 INFO - Selenium build info: version: '3.4.0', revision: 'unknown'
selenium_1 | 20:39:23.973 INFO - Launching a standalone Selenium Server
selenium_1 | 2017-07-10 20:39:23.996:INFO::main: Logging initialized @241ms to org.seleniumhq.jet
selenium_1 | 20:39:24.058 INFO - Driver provider org.openqa.selenium.ie.InternetExplorerDriver reg
selenium_1 | registration capabilities Capabilities [{ensureCleanSession=true, browserName=intern
```

```

selenium_1 | 20:39:24.059 INFO - Driver provider org.openqa.selenium.edge.EdgeDriver registration
selenium_1 | registration capabilities Capabilities [{browserName=MicrosoftEdge, version=, platf
selenium_1 | 20:39:24.060 INFO - Driver class not found: com.opera.core.systems.OperaDriver
selenium_1 | 20:39:24.060 INFO - Driver provider com.opera.core.systems.OperaDriver registration i
selenium_1 | Unable to create new instances on this machine.
selenium_1 | 20:39:24.060 INFO - Driver class not found: com.opera.core.systems.OperaDriver
selenium_1 | 20:39:24.061 INFO - Driver provider com.opera.core.systems.OperaDriver is not regist
selenium_1 | 20:39:24.062 INFO - Driver provider org.openqa.selenium.safari.SafariDriver registrat
selenium_1 | registration capabilities Capabilities [{browserName=safari, version=, platform=MAC}
selenium_1 | 2017-07-10 20:39:24.099:INFO:osjs.Server:main: jetty-9.4.3.v20170317
selenium_1 | 2017-07-10 20:39:24.136:INFO:osjs.ContextHandler:main: Started o.s.j.s.ServletContext
selenium_1 | 2017-07-10 20:39:24.156:INFO:osjs.AbstractConnector:main: Started ServerConnector@754
selenium_1 | 2017-07-10 20:39:24.157:INFO:osjs.Server:main: Started @402ms
selenium_1 | 20:39:24.157 INFO - Selenium Server is up and running
e2e_1 | Selenium driver initialized.
e2e_1 | Now waiting for 3sec
e2e_1 | Now running test
selenium_1 | 20:39:27.338 INFO - SessionCleaner initialized with insideBrowserTimeout 0 and client
selenium_1 | 20:39:27.360 INFO - Executing: [new session: Capabilities [{browserName=chrome}]]
selenium_1 | 20:39:27.379 INFO - Creating a new session for Capabilities [{browserName=chrome}]
selenium_1 | Starting ChromeDriver 2.30.477691 (6ee44a7247c639c0703f291d320bdf05c1531b57) on port
selenium_1 | Only local connections are allowed.
selenium_1 | 20:39:28.046 INFO - Detected dialect: OSS
selenium_1 | 20:39:28.060 INFO - Done: [new session: Capabilities [{browserName=chrome}]]
selenium_1 | 20:39:28.089 INFO - Executing: [get: http://ui:3000/pet]
selenium_1 | 20:39:28.937 INFO - Done: [get: http://ui:3000/pet]
selenium_1 | 20:39:28.942 INFO - Executing: [get title]
selenium_1 | 20:39:28.952 INFO - Done: [get title]
selenium_1 | 20:39:28.956 INFO - Executing: [delete session: 4c208b23-0724-4268-9077-2e889d67559f]
selenium_1 | 20:39:29.026 INFO - Done: [delete session: 4c208b23-0724-4268-9077-2e889d67559f]
e2e_1 | npm info lifecycle e2etests@1.0.0~poststart: e2etests@1.0.0
e2e_1 | npm info ok
end2endtests_e2e_1 exited with code 0

```

9. Clean up when done; hit CTRL-c to stop the application and then:

```

$ docker-compose down
$ docker-compose -f docker-compose-e2e.yml down

```

## 7.2 Conclusion

In this exercise you have implemented and tested the mechanics of an End-to-End test scenario. We were using **Selenium Web Driver** for Node JS to automate the tests. The tests itself are run through the Selenium standalone server listening at 0.0.0.0:4000/wd/hub for incoming commands.

From the container orchestration standpoint, this is similar to the the API tests we did in the last exercise, in that we stand up completely independent testing infrastructure, described as its own services in its own docker-compose file, to test interactions with our full application. In this case however, our application's docker-compose file can be closer to (or exactly) how we would define it in production - no public API port exposures, no mounted volumes, and no special execution environments for our application components; all that has been validated previously, and now we can test our full application with a totally independent testing stack.

## 8 Service Discovery

For true application portability, connections between containers need to be as portable as the container themselves. Therefore, the Docker Engine provides an on-board DNS service that allows traffic to be routed between application components by container or service name, without requiring any explicit service discovery in our application logic. Let's see this in action by creating a simple service that uses Redis to store a value. The service will access Redis via its service name.

The project folder for this exercise is `~/ddev-labs/service-discovery`.

### 8.1 Creating an Application

1. Open the file `server.py` from the project folder. It should have this content:

```
from flask import Flask, request, jsonify
from redis import Redis

app = Flask(__name__)
redis = Redis("redis")

@app.route('/')
def info():
    return "Demo for Service Discovery"

@app.route('/scores', methods=['POST'])
def score():
    json = request.get_json()
    redis.set(json['name'], json['score'])
    return 'CREATED', 201

@app.route('/scores/<name>', methods=['GET'])
def get_score(name):
    value = redis.get(name);
    if not value:
        return 'NOT FOUND', 404
    ret = { 'name': name, 'score': value }
    return jsonify(ret)

if __name__ == '__main__':
    app.run(port=3000, host="0.0.0.0")
```

Note how on line 5 we're using the (short) service name of the Redis service to define the Redis host in the statement `redis = Redis("redis")`. There is no need to use IP addresses or FQDNs to access Redis. This is Docker's **service discovery** in action.

**Stop:** How is the service name `redis` defined? It is defined in the `docker-compose.yml` file introduced further down.

2. There is also a file `requirements.txt` containing the list of dependencies needed by the Python app:

```
flask
redis
```

3. Add the following instructions to the file `service-discovery/Dockerfile`:

```
FROM python:2.7-alpine
RUN mkdir -p /app
WORKDIR /app
COPY requirements.txt /app/
```

```

RUN pip install -r requirements.txt
COPY . /app
EXPOSE 3000
CMD python server.py

```

- Also add the following content to `service-discovery/docker-compose.yml`:

```

version: '3.1'
services:
  app:
    build: .
    ports:
      - 3000:3000
    networks:
      - back-tier
  redis:
    image: redis:latest
    networks:
      - back-tier
networks:
  back-tier:

```

Gist: <http://bit.ly/2h4uAdy>

Note that the services are both connected to the `back-tier` network; recall that containers not plugged into the same network are firewalled from each other by default, as part of Docker's container networking model.

- Run the application:

```
$ docker-compose up --build
```

- Use a `curl` to test the API. Open a second terminal window and execute:

```
$ curl -H "Content-Type: application/json" \
  -X POST -d '{"name": "Joey", "score": 6}' localhost:3000/scores
```

- Make sure the value was written:

```
$ curl localhost:3000/scores/Joey
```

which should result in an answer like this:

```
{'name': 'Joey', 'score': 6}
```

## 8.2 Using Aliases in Service Discovery

Sometimes we have to live with given FQDNs for services and are not free to choose a short name like in the above sample. In this case we can use aliases.

- Suppose our `redis` service had to be called `redis.example.com`. Rename the service as such in the `docker-compose.yml` file and add a `links:` section to the `app` service to alias this name to the name `redis` used in our python server:

```

...
app:
  ...
  links:
    - "redis.example.com:redis"
  ...
redis.example.com:

```

```
image: redis:latest
...
```

2. Run and test the application as in the example above. This allows us to decouple the name of the service from how we refer to it in our application logic.

### 8.3 Network Partitioning

Docker's container network model is centered around software defined networks; regardless of underlying implementation details, Docker will always firewall containers on different Docker networks from each other.

1. Ping your redis service from within your app service container:

```
$ docker container exec <app container id> ping redis.example.com
```

The ping should proceed as normal.

2. Stop your application, and modify the docker-compose.yml file to connect your redis service to a different network:

```
...
  redis.example.com:
    image: redis:latest
    networks:
      - extra-net

networks:
  back-tier:
  extra-net:
```

3. Restart your application and attempt the same ping as above; it fails, since these services can no longer resolve each other's names and are firewalled from each other by Docker's network partitioning.
4. Try the curl commands from above to write and read data to the database; they also fail, for the same reason.

### 8.4 Kubernetes and Service Discovery

Contrary to Docker Swarm, Kubernetes' DNS works across the whole cluster. There are no default network partitions like in Swarm with its software defined networks. In Kubernetes one can use network policies to partition applications from each other.

1. Open the file kube.yaml and analyze its content. Notice that an image has been provided for you for running the application:

```
...
spec:
  containers:
    - name: ui
      image: training/ddev-service-discovery-kube:1.0
      ports:
        - containerPort: 3000
  ...
```

2. Run the application in Minikube:

```
$ kubectl create -f kube.yaml
```

you should see this:

```
namespace "ddev" created
deployment "redis" created
service "redis" created
```

```
deployment "app" created
service "app" created
```

3. Get the IP of Minikube:

```
$ export KUBE_IP=$(minikube ip)
```

4. Get the public port on which the application is exposed:

```
$ export APP_PORT=$(kubectl get -n ddev svc/app \
  --template '{{range .spec.ports}}{{.nodePort}}{{end}}')
```

Note, we're using Go template syntax above to extract the `nodePort` value from the app service meta data.

5. Use `curl` to add some value to the application:

```
$ curl -H "Content-Type: application/json" \
  -X POST -d '{"name": "Joey", "score": 6}' ${KUBE_IP}:${APP_PORT}/scores
```

the response should be:

```
CREATED
```

6. Now double check that the value has been stored in Redis, again using `curl`:

```
$ curl ${KUBE_IP}:${APP_PORT}/scores/Joey
```

which should result in this answer:

```
{
  "name": "Joey",
  "score": "6"
}
```

7. Discuss with your peers how and why the Python application can access Redis when running in Kubernetes.
8. Clean up:

```
$ kubectl delete namespace/ddev
```

## 8.5 Conclusion

In this exercise we have seen how Docker provides out of the box service discovery for services running on the same network. In our case the Flask API could discover Redis simply by its service name `redis`. We also have shown how we can use aliases to use different, given (for example) FDQNs instead of the short service names.

Furthermore, we have illustrated the basic behavior of Docker network partitioning; containers on different networks are automatically firewalled from each other, and cannot resolve container or service names for objects not connected to the same network. As such, when designing a containerized application it is crucial to record in your `docker-compose.yml` which services need to be connected to which networks. Done conservatively, this is a simple pattern to enhance the security of your containerized applications.

Meanwhile, Kubernetes provides no such default network segmentation; Kube's networking model prioritizes flat, cluster-wide networks that resemble flat networks of VMs in order to make application networking when moving from VMs to containers as simple as possible. In order to impose network segmentation in Kubernetes, we need to choose a networking plugin that support the `NetworkPolicy` object.

## 9 Health Checks

In this exercise we're going to extend the Java API of our workshop sample application with a health check.

## 9.1 Health Checks in Swarm

The project folder for this exercise is `~/ddev-labs/health-checks`.

### 9.1.1 Implementing a Health Endpoint

1. Open the file `api/src/main/java/com/docker/ddev/controller/UtilityController.java`, and add the following healthcheck route, immediately under the `JdbcTemplate` declaration in the `UtilityController` class:

```
@SuppressWarnings("unchecked")
@RequestMapping(value="/healthcheck", method = RequestMethod.GET)
public ResponseEntity<?> healthCheck() {
    logger.info("Performing healthcheck");
    if(unhealthy) {
        return new ResponseEntity<Object>(new CustomErrorType("API unhealthy."),
            HttpStatus.INTERNAL_SERVER_ERROR);
    }
    JSONObject healthcheck = new JSONObject();
    try
    {
        String sql = "SELECT to_char(current_timestamp, 'YYYY-MM-DD HH24:MI')";
        String status = jdbcTemplate.queryForObject(sql, String.class);
        healthcheck.put("status", status);
    } catch (Exception e) {
        logger.warn("An exception occurred while checking the database: {}", e);
        return new ResponseEntity<Object>(new CustomErrorType("Database not responding."),
            HttpStatus.INTERNAL_SERVER_ERROR);
    }
    return new ResponseEntity<JSONObject>(healthcheck, HttpStatus.OK);
}
```

Our healthcheck checks a boolean that flags an unhealthy state, as well as performs its own check on database connectivity. If either fail, a server error is returned. Otherwise, the healthcheck responds with a 200 / OK.

### 9.1.2 Adjusting the Dockerfile and building the Images

1. Open the file `api/Dockerfile`, and note that after the second `FROM` we have added an instruction to install `curl` as follows:

```
...
FROM java:8-jdk-alpine
RUN apk update && apk add curl
...
```

We need `curl` inside our container to probe the healthcheck endpoint.

2. Build, tag, and push images for all three services of the application by running the provided `build-and-push.sh` script:

```
$ export MY_DOCKER_ID=<Docker ID>
$ bash build-and-push.sh
```

### 9.1.3 Running the Application with Health Checks in a Swarm

1. Make sure your Docker host is running in swarm mode. If not please init a swarm.
2. Create the necessary secret in the swarm:

```
$ echo 'gordonpass' | docker secret create postgres_password -
```

3. In the health-checks folder there is a file called `my-stack.yml` with the following content. We will be using it to run our application as a **Stack**, which is much the same as an application run using Docker Compose, but taking advantage of all the features of Docker Swarm Mode. Note that since we are using Docker Swarm Mode, we will need to modify the `my-stack.yml` below and replace `user` with our Docker Hub username. This will allow each host within our swarm to pull the image used by each service.

```
version: "3.1"

services:
  database:
    image: user/ddev-db:1.0
    environment:
      POSTGRES_USER: gordonuser
      POSTGRES_DB: ddev
    ports:
      - "5432:5432"
    networks:
      - back-tier
    secrets:
      - postgres_password

  api:
    image: user/ddev-api:1.0
    ports:
      - "8080:8080"
      - "5005:5005"
    networks:
      - front-tier
      - back-tier
    secrets:
      - postgres_password
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/utility/healthcheck"]
      interval: 30s
      timeout: 10s
      retries: 3

  ui:
    image: user/ddev-ui:1.0
    ports:
      - "3000:3000"
    networks:
      - front-tier

secrets:
  postgres_password:
    external: true

networks:
  front-tier:
  back-tier:
```

Please note the section `healthcheck` in the details of the service `api`. We're doing a health check every 30 seconds. On each check we timeout after 10 seconds. Finally, we assume that the service task is dead after three consecutive failures.



- Please discuss with your peers what exactly the above test command in the `healthcheck` section does. For more details see <https://docs.docker.com/engine/reference/builder/#healthcheck>.

- Create a stack:

```
$ docker stack deploy -c my-stack.yml app
```

Give it some time to start up. The following command:

```
$ watch docker stack ps app
```

should eventually show this:

```
Every 2.0s: docker stack ps app
```

ID	NAME	IMAGE	NODE	DESIRED STATE
n6uvhy7rxmux	app_ui.1	ddev-ui:1.0	moby	Running
woky0lgjcnj1	app_api.1	ddev-api:1.0	moby	Running
kwhwz3olk5j6	app_database.1	ddev-db:1.0	moby	Running

- List all containers of the app stack:

```
$ docker container ls | grep app_
```

which should show something similar to this:

6b4f28d713da	ddev-ui:1.0	"/bin/sh -c 'node ..."	8 minutes ago	Up 8 minutes
5cd074b02131	ddev-api:1.0	"java -jar /app/dd..."	8 minutes ago	Up 8 minutes
478f3f1295ba	ddev-db:1.0	"docker-entrypoint..."	8 minutes ago	Up 8 minutes

Please note the (healthy) in the STATUS column of the api container.

- Test whether the api healthcheck works:

```
$ curl localhost:8080/utility/healthcheck
```

it should return something like:

```
{"status": "2017-07-14 15:03"}%
```

- Test whether the app runs as expected:

```
$ curl localhost:3000/pet
```

we should get the HTML for the cat view. Everything seems healthy and functioning correctly at this point.

### 9.1.4 Responding to Health Failure

Now we're going to artificially trigger an unhealthy state in our application.

- Notice in the `UtilityController` class you modified above, there is an additional route `simulate-failure`. Of course you would never have such a route in a real application, but in this case we include it to conveniently induce an unhealthy application state.

- Monitor the containers:

```
$ watch docker container ls
```

- Open another terminal and watch the api service:

```
$ watch docker service ps app_api
```

- And in a third terminal window trigger the failure of the api:

```
$ curl localhost:8080/utility/simulate-failure
```

5. Observe what is happening (be patient). How long will you have to wait until failure is reported? Play with the healthcheck settings in the stack file to make your wait shorter.
6. Don't forget to clean up after yourself:

```
$ docker stack rm app
```

## 9.2 Health Checks in Kubernetes

The project folder for this part of the exercise is `~/ddev-labs/health-check-kube`.

### 9.2.1 Implementing a Health Endpoint

1. Open the file `main.py` and note the endpoint `/health` used for health checks:

```
...
@app.route("/health", methods=['GET'])
def get_health():
    if healthy == True:
        return 'OK', 200
    return 'NOT OK', 500
...
```

we return OK (200) if a global variable is True, otherwise we return NOT OK (500).

2. Open the file `kube.yaml` and note the `livenessProbe` part in the YAML:

```
...
livenessProbe:
  exec:
    command:
      - curl
      - -f
      - http://localhost:8000/health
  initialDelaySeconds: 5
  periodSeconds: 5
```

we're using a command to check the health endpoint: `curl -f http://localhost:8000/health`.

3. Create the pod:

```
$ kubectl create -f kube.yaml
```

4. Use the `describe` command to observe the pod:

```
$ kubectl describe pod/liveness-exec
```

which should show something like this (shortened):

```
...
Events:
  Type     Reason             Age          From              Message
  ----     -
  Normal   Scheduled          16m         default-scheduler Successfully assigned liveness-exec to minikube
  Normal   SuccessfulMountVolume 16m         kubelet, minikube MountVolume.SetUp succeeded for volume "training/liveness-exec"
  Normal   Pulling            16m         kubelet, minikube pulling image "training/liveness-exec"
  Normal   Pulled              15m         kubelet, minikube Successfully pulled image "training/liveness-exec"
  Normal   Created             15m         kubelet, minikube Created container liveness-exec
  Normal   Started             15m         kubelet, minikube Started container liveness-exec
```

5. Now after we have confirmed that the pod runs normally when the health endpoint reports OK, it is time to see what's happening when the container running inside the pod becomes unhealthy. For this we run a shell inside the container:

```
$ kubectl exec -it liveness-exec -- /bin/sh
```

6. Once inside the container we use curl to set the value of the global health variable in the app to False:

```
# curl -X POST localhost:8000/health/bad
```

7. We can double check that the health endpoint now reports NOT OK:

```
# curl -I localhost:8000/health
```

which gives us something like this:

```
HTTP/1.0 500 INTERNAL SERVER ERROR
Content-Type: text/html; charset=utf-8
Content-Length: 6
Server: Werkzeug/0.14.1 Python/2.7.14
Date: Tue, 13 Feb 2018 19:40:53 GMT
```

8. Leave the container by typing exit.  
9. And check the status of the pod after this:

```
$ kubectl describe pod/liveness-exec
```

which returns this (shortened):

```
...
Events:
  Type     Reason            Age          From          Message
  ----     -
  Normal   Scheduled         2h           default-scheduler Successfully assigned liveness-exec to node1
  Normal   SuccessfulMountVolume 2h           kubelet, minikube MountVolume.SetUp succeeded for volume "data"
  Warning  Unhealthy         1m (x3 over 1m) kubelet, minikube Liveness probe failed: HTTP probe failed with code 500
  Normal   Pulled            1m (x2 over 2h) kubelet, minikube Container image "training/liveness-exec" not found
  Normal   Created           1m (x2 over 2h) kubelet, minikube Created container
  Normal   Started           1m (x2 over 2h) kubelet, minikube Started container
  Normal   Killing           1m           kubelet, minikube Killing container with id ...
```

Evidently the pod is killed after 3 unsuccessful retries to probe the health endpoint and then restarted. Kubernetes will abort this process of killing and creating the pod after a few times.

10. Do the very same as above with a liveness probe base on an HTTP call. Use kube2.yaml for this; the pod will be called liveness-http this time.  
11. Cleanup:

```
$ kubectl delete pod/liveness-exec
$ kubectl delete pod/liveness-http
```

## 9.3 Optional Challenge: Health Check Endpoint in Python

### 9.3.1 Creating the application

1. In your editor open the project in the folder api-tests-flask.
2. Open the file main.py in the subfolder api.
3. Add a GET endpoint /health to the API:

```
healthy = True

@app.route("/health", methods=['GET'])
def get_health():
    if healthy == True:
```

```
    return 'OK', 200
    return 'NOT OK', 500
```

4. Add a POST endpoint `/health/<status>` to the API which will be used to simulate a healthy or unhealthy service:

```
@app.route("/health/<status>", methods=['POST'])
def set_health(status):
    global healthy
    if status.lower() == "good":
        healthy = True;
    else:
        healthy = False;
    return 'OK', 200
```

5. Build the image. Remember we called it `api-tests`.

6. Run a container from this image:

```
$ docker container run --rm -it -p 8000:8000 api-tests
```

7. Open another terminal and query the `health` endpoint:

```
$ curl localhost:8000/health
```

The result should be: `OK`

8. Switch the health to bad:

```
$ curl -X POST localhost:8000/health/bad
```

9. Probe the `/health` endpoint again. This time the result should be: `NOT OK`.

10. Set the status to good again and make sure the `/health` endpoint returns `OK`.

11. Kill the container.

### 9.3.2 Defining the Health Check Command

Now we need to define the command that Docker swarm will use to check the health of the service. We can do this in the `Dockerfile`.

1. Open the `Dockerfile` in the `api` sub-folder.
2. Add the following line to it just before the definition of the `ENTRYPOINT`:

```
HEALTHCHECK CMD curl --fail http://localhost:8000/health || exit 1
```

3. Please discuss with your neighbor what exactly the above command does. For more details see <https://docs.docker.com/engine/reference/builder/#healthcheck>
4. Build the image again.

### 9.3.3 Running a Service with Health Check

Now we want to run the API as a service.

1. Make sure your Docker host is running in swarm mode. If not please init a swarm.
2. Create a file called `api-tests-flask/my-stack.yml` and add the following content:

```
version: '3.1'
services:
  api:
    image: api-tests
```

```
ports:
  - 8000:8000
healthcheck:
  interval: 2s
  timeout: 2s
  retries: 3
```

Gist: <http://bit.ly/2v0UCEb>

3. Save the file.

4. Deploy the stack using the above stack file:

```
$ docker stack deploy -c my-stack.yml api
```

5. Now open a second terminal and run the following command to observe the `api-tests` service:

```
$ watch docker service ps api_api
```

you should see something like this:

```
Every 2.0s: docker service ps api_api
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
a1cqlkz8jfpq	api_api.1	api-tests	moby	Running	Running 20 seconds ago

As we can see, the service is up and running happily.

6. In another terminal window set the health of the service to bad:

```
$ curl -X POST localhost:8000/health/bad
```

7. Observe what is happening in the window running the watch command. Give it at least 16 sec. or so.

8. With your neighbor discuss what you saw and why this happened. Specifically discuss why it took approx. 16 seconds until something happened. The result should look something like this:

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
u2dfx17snj26	api_api.1	api-tests	moby	Running	Running 16 seconds ago	
t5agzq65t8ez	\_ api_api.1	api-tests	moby	Shutdown	Failed 25 seconds ago	"task: non-zero

9. Try to query the `/health` endpoint. What do you see?

10. Set the service to unhealthy again and observe.

11. For a third time set the service to unhealthy but after approx. 4 to 5 sec. set it back to healthy. What happens? What happens if you wait longer than 6 sec. but less than 16 sec.?

## 9.4 Conclusion

As a developer, you know best what it means for your application to be healthy. Bearing in mind that there could be an enormous number of instances of your application being scheduled across a datacenter (an order of magnitude more once containerized compared to VMs is not uncommon), it is a crucial responsibility for you to write healthcheck endpoints for all your containerized software. That way, your orchestrator (and not your ops team) can take responsibility for maintaining the health of your application.

Also notice the orchestrator's "shoot first" mentality; no actions are taken to try and fix an unhealthy container. It is simply killed and replaced with a fresh container. Make sure to plan for this: if your container enters an unhealthy state, it should try and log what's wrong, finish transactions, push valuable stateful information out to somewhere it can be recovered, and generally take action to compensate for the fact that it is about to be terminated.

## 10 Defensive Programming

In this exercise we will implement a mechanism to protect the API from a failure in case that the database on which it depends takes longer than expected to initialize.

The project folder for this exercise is `~/ddev-labs/defensive-programming`.

### 10.1 Defining an Entrypoint Script

We will implement a startup script that defines a `wait_for` function which waits for a TCP connection on a given `<HOST>:<PORT>` combination. The function ends if either the connection is possible or a certain timeout time is reached.

1. Open `api/entrypoint.sh`. It should have this content:

```
#!/bin/sh

echoerr() { if [[ $QUIET -ne 1 ]]; then echo "$@" 1>&2; fi }

wait_for()
{
    if [[ $TIMEOUT -gt 0 ]]; then
        echoerr "$cmdname: waiting $TIMEOUT seconds for $HOST:$PORT"
    else
        echoerr "$cmdname: waiting for $HOST:$PORT without a timeout"
    fi
    start_ts=$(date +%s)
    while :
    do
        echo "Probing $HOST:$PORT..."
        nc -z -w 1 $HOST $PORT </dev/null
        result=$?
        end_ts=$(date +%s)
        delta_ts=$((end_ts-start_ts))
        if [[ $result -eq 0 ]]; then
            echoerr "$cmdname: $HOST:$PORT is available after $((end_ts - start_ts)) seconds"
            break
        fi
        echo "Passed $delta_ts of $TIMEOUT seconds"
        if [[ $delta_ts -ge $TIMEOUT ]]; then
            echo "Timeout reached, aborting..."
            break
        fi
        sleep 1
    done
    return $result
}

TIMEOUT=${TIMEOUT:-15}
QUIET=${QUIET:-0}

if [[ "$HOST" == "" || "$PORT" == "" ]]; then
    echoerr "Error: you need to provide a HOST and PORT to test."
    exit 1
fi

cmdname="entrypoint.sh"
```

```
wait_for
RESULT=$?
if [[ $RESULT -ne 0 ]]; then
    exit $RESULT
fi

echo "Starting API..."
java -jar /app/ddev-0.0.1-SNAPSHOT.jar --spring.profiles.active=postgres
```

2. We need to make the file an executable:

```
$ chmod +x ./api/entrypoint.sh
```

3. Open the file `api/Dockerfile-def` and modify it until the second part looks like this:

```
...
FROM java:8-jdk-alpine
RUN adduser -Dh /home/gordon gordon
WORKDIR /app
COPY --from=appserver /usr/src/ddev/target/ddev-0.0.1-SNAPSHOT.jar .
COPY --from=appserver /usr/src/ddev/entrypoint.sh ./
ENTRYPOINT ["/entrypoint.sh"]
```

Note that the original `ENTRYPOINT` command is equivalent to the last line of our `entrypoint.sh`; we now run `entrypoint.sh` on container startup to wrap our defensive measures together with the previous startup command.

4. Open the file `docker-compose-def.yml` and focus on the definition of the `api` service:

```
version: "3.1"

services:
  database:
    ...

  api:
    build:
      context: api
      dockerfile: Dockerfile-def
    image: ddev_api
    ...

...
```

Note how we're using the file `api/Dockerfile-def` to build the `api` service.

5. Run the application:

```
$ docker-compose -f docker-compose-def.yml up --build
```

Carefully observe the (log) output generated in the terminal, specifically for the `api_1` service. You should see something similar to this (shortened for brevity):

```
...
database_1 | The files belonging to this database system will be owned by user "postgres".
database_1 | This user must also own the server process.
database_1 |
api_1      | entrypoint.sh: waiting 30 seconds for database:5432
database_1 | The database cluster will be initialized with locale "en_US.utf8".
database_1 | The default database encoding has accordingly been set to "UTF8".
database_1 | The default text search configuration will be set to "english".
```

```

api_1      | Probing database:5432...
database_1 |
database_1 | Data page checksums are disabled.
database_1 |
database_1 | fixing permissions on existing directory /var/lib/postgresql/data ... ok
api_1      | Passed 0 of 30 seconds
database_1 | creating subdirectories ... ok
database_1 | selecting default max_connections ... 100
database_1 | selecting default shared_buffers ... 128MB
database_1 | selecting dynamic shared memory implementation ... posix
database_1 | creating configuration files ... ok
database_1 | running bootstrap script ... ok
api_1      | Probing database:5432...
api_1      | Passed 1 of 30 seconds
database_1 | performing post-bootstrap initialization ... ok
database_1 | syncing data to disk ...
database_1 | WARNING: enabling "trust" authentication for local connections
database_1 | You can change this by editing pg_hba.conf or using the option -A, or
database_1 | --auth-local and --auth-host, the next time you run initdb.
database_1 | ok
database_1 |
database_1 | Success. You can now start the database server using:
database_1 |
database_1 |         pg_ctl -D /var/lib/postgresql/data -l logfile start
database_1 |
database_1 | *****
database_1 | WARNING: No password has been set for the database.
database_1 |         This will allow anyone with access to the
database_1 |         Postgres port to access your database. In
database_1 |         Docker's default configuration, this is
database_1 |         effectively any other container on the same
database_1 |         system.
database_1 |
database_1 |         Use "-e POSTGRES_PASSWORD=password" to set
database_1 |         it in "docker run".
database_1 | *****
database_1 | waiting for server to start...LOG:  could not bind IPv6 socket: Cannot assign request
database_1 | HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds
database_1 | LOG:  database system was shut down at 2017-07-21 17:12:46 UTC
database_1 | LOG:  MultiXact member wraparound protections are now enabled
database_1 | LOG:  database system is ready to accept connections
database_1 | LOG:  autovacuum launcher started
api_1      | Probing database:5432...
api_1      | Passed 2 of 30 seconds
database_1 | done
database_1 | server started
database_1 | CREATE DATABASE
database_1 |
database_1 | CREATE ROLE
database_1 |
database_1 |
database_1 | /usr/local/bin/docker-entrypoint.sh: running /docker-entrypoint-initdb.d/init-db.sql
database_1 | CREATE TABLE
database_1 | ALTER TABLE
database_1 | ALTER ROLE

```





of the database for a successful TCP connection. Only once the script could connect did we continue and initialize the API. For the case of initialization defenses, we saw a typical Dockerized pattern of wrapping the nominal startup command in a script that performs the defense check first, and calling that script as our container's ENTRYPOINT.

## 11 Logging

In this lab we're going to apply some of the recommended best practices around logging and error handling when using containers.

Note: In the following <Docker ID> denotes your Docker Store user name. If you don't have an account on Docker Store please create one here (it's free): <https://store.docker.com/signup>

The project folder for this exercise is `~/ddev-labs/logging`.

### 11.1 Logging in Java

In this exercise you're going to create a Test controller in the Java API of our workshop sample application which logs events to STDIN such as that these events can be picked up by a logging aggregator. We use slf4j (Simple Logging Facade for Java) for this (<https://www.slf4j.org/>).

1. To the folder `api/src/main/java/com/docker/ddev/controller` we have added a file `TestController.java` with the following content:

```
package com.docker.ddev.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class TestController{
    @RequestMapping(value = "/factorial/{number}", method = RequestMethod.GET)
    public ResponseEntity<String> getFactorial(@PathVariable("number") long number) {
        Long fac = calcFactorial(number);
        return new ResponseEntity<String>(Long.toString(fac), HttpStatus.OK);
    }

    private Long calcFactorial(long number){
        if(number == 1) return number;
        return number * calcFactorial(number-1);
    }
}
```

2. Run the application with:

```
$ cd ~/ddev-labs/logging
$ docker-compose up --build
```

3. And test the new endpoint:

```
$ curl localhost:8080/api/factorial/12
```

you should see the value 479001600 returned.

4. Our function seems to work, but it isn't providing any logging information. Add the following 2 import statements to the class to allow us to define a logger:

```
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;
```

5. Add a static logger to the class:

```
...
public class TestController{
    static final Logger logger = LoggerFactory.getLogger(TestController.class);
    ...
}
```

6. Now we can introduce logging to give us some more information to help us analyze what's going on in the system. Often we add some debug only information. Modify the getFactorial function to look like this:

```
public ResponseEntity<String> getFactorial(@PathVariable("number") long number) {
    logger.debug("Entering getFactorial with number {}", number);

    logger.info("Calculating factorial for number {}", number);
    Long fac = calcFactorial(number);

    logger.debug("Exiting getFactorial with result {}", fac);
    return new ResponseEntity<String>(Long.toString(fac), HttpStatus.OK);
}
```

We have instrumented our function with debug messages at the entry- and exit point. We also generate a log message of type info every time we try to calculate the factorial.

7. Now we want to make sure the factorial is only calculated for allowed numbers and report an error otherwise. Thus modify the body of the function like this:

```
logger.debug("Entering getFactorial with number {}", number);

logger.info("Calculating factorial for number {}", number);

if(number < 0){
    String msg = "Cannot calculate factorial for negative numbers";
    logger.error(msg);
    return new ResponseEntity<String>(msg, HttpStatus.BAD_REQUEST);
}

if(number > 20){
    String msg = "Cannot calculate factorial for numbers > 20";
    logger.error(msg);
    return new ResponseEntity<String>(msg, HttpStatus.BAD_REQUEST);
}

Long fac = calcFactorial(number);
...
```

8. Now run the application:

```
$ docker-compose up --build
```

9. And test it for different numbers while observing the output in the console:

```
$ curl localhost:8080/api/factorial/10
$ curl localhost:8080/api/factorial/-2
$ curl localhost:8080/api/factorial/22
$ curl localhost:8080/api/factorial/0
```

In the case of 22 you should see something similar to this in the terminal:

```
api_1      | 2017-07-12 14:39:04.562 INFO 1 --- [nio-8080-exec-1] c.docker.ddev.controller.
            | TestController : Calculating factorial of 22.
api_1      | 2017-07-12 14:39:04.563 ERROR 1 --- [nio-8080-exec-1] c.docker.ddev.controller.
            | TestController : Cannot calculate factorial for numbers > 20
```

10. Notice that by default logging messages with loglevel debug are not output. Discuss with your peers what change would need to happen to see debug level messages.

## 11.2 Optional: Logging in .NET Core

We can do a similar exercise in .NET Core as we have done in Java. Here we'll use log4net, which is a port of log4j, as our logging library.

1. Add a project folder logging-net to the lab folder ~/ddev/logging.
2. In a terminal window navigate to this folder and run a microsoft/dotnet container:

```
$ docker container run --rm -it -v $(pwd):/app -w /app microsoft/dotnet
```

3. Inside this container create a new WebAPI project, and exit the container:

```
$ dotnet new webapi
$ exit
```

4. Add a file SampleController.cs file to the Controllers folder that the last step created with this content:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace app.Controllers
{
    [Route("")]
    public class SampleController : Controller
    {
        ILogger _log;

        public SampleController(ILogger<SampleController> logger){
            _log = logger;
        }

        [HttpGet]
        public string Get()
        {
            return "Sample controller";
        }

        [HttpGet("/info")]
        public string Info()
        {
            _log.LogInformation(1000, "This is an INFO message.");
            return "info";
        }

        [HttpGet("/warning")]
        public string Warning()
        {
            _log.LogWarning(2000, "This is an WARN message.");
            return "warning";
        }
    }
}
```

```

[HttpGet("/error")]
public string Error()
{
    _log.LogError(3000, "This is an ERR message.");
    return "error";
}

[HttpGet("/critical")]
public string Critical()
{
    _log.LogCritical(9000, "This is an CRITICAL message.");
    return "critical";
}
}
}

```

Gist: <http://bit.ly/2tEsLpG>

5. Now add a Dockerfile to the root logging-net directory:

```

FROM microsoft/dotnet:2.1-sdk
RUN mkdir -p /app
WORKDIR /app
COPY app.csproj /app/
RUN dotnet restore
COPY . /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
CMD ["dotnet", "run"]

```

6. Finally add a .dockerignore file to the project that contains patterns for elements to ignore when building the Docker image:

```

obj
bin
Dockerfile
.dockerignore

```

7. Build the image:

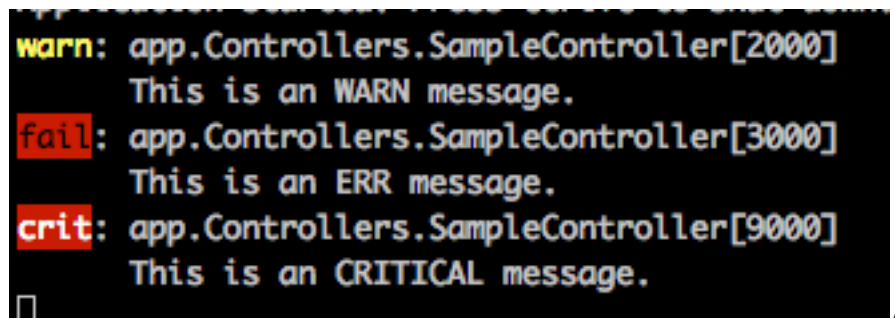
```
$ docker image build -t logging-net .
```

8. Run a container:

```
$ docker container run --rm -it -p 5000:80 logging-net
```

9. Test the API by opening a browser at localhost:5000/info. You should get a response info.

10. Navigate to /warning, /error and /critical. In the browser you should get the respective confirmation warning, error and critical. Notice in the terminal that you get something like this:



```

warn: app.Controllers.SampleController[2000]
      This is an WARN message.
fail: app.Controllers.SampleController[3000]
      This is an ERR message.
crit: app.Controllers.SampleController[9000]
      This is an CRITICAL message.

```

Note that the info message did not make it to the terminal. Why?

#### 11. Optional:

- Change the application such as that info messages also get reported.
- Add a /debug endpoint which generates a debug log message and make sure it is reported in the terminal

### 11.3 Optional: Logging in Python/Flask

To demonstrate the same principles in a different language and framework let's create an API similar to the one we did in .NET Core but this time using Python and Flask.

1. Add a file `server.py` to the project folder with this content:

```
import logging
from logging.handlers import RotatingFileHandler

from flask import Flask

app = Flask(__name__)
app.logger.setLevel(logging.DEBUG)

# create a console logger
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
app.logger.addHandler(ch)

@app.route('/')
def index():
    app.logger.info('This is a Python/Flask sample API')
    return "This is a Python/Flask sample API"

@app.route('/info')
def info():
    app.logger.info('This is an info message')
    return "info"

@app.route('/warning')
def warning():
    app.logger.warning('This is a warning message')
    return "warning"

@app.route('/error')
def error():
    app.logger.error('This is an error message')
    return "error"

@app.route('/critical')
def critical():
    app.logger.critical('This is a critical message')
    return "critical"

if __name__ == '__main__':
    app.run(port=3000, host="0.0.0.0")
```

Gist: <http://bit.ly/2uz8Dbi>

2. Add a file `requirements.txt` to the project folder with this content:

```
flask
```

3. Now add a `Dockerfile` with these instructions:

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . /app
CMD python server.py
```

4. Build the Docker image:

```
$ docker image build -t logging-flask .
```

5. And run a container:

```
$ docker container run --rm -it -p 3000:3000 logging-flask
```

6. Open a browser and navigate to `localhost:3000/info` and observe the output in the terminal. You should see something similar to:

```
2017-04-05 17:23:06,690 - __main__ - INFO - This is an info message
172.17.0.1 - - [05/Apr/2017 17:23:06] "GET /info HTTP/1.1" 200 -
```

7. Navigate to `/warning`, `/error` and `/critical` and observe the output.

## 11.4 Conclusion

In this lab we have created some sample apps that generate log messages of different levels of severity. When thinking about logging in containerized applications, remember that your application may be replicated and distributed across many containers, services, and datacenters; will ops be able to put any given log message in context with the logs of the rest of your distributed application? Try to offer information that will help operations not only understand this individual process, but how it fits into the larger application as a whole, where relevant.

## 12 Error Handling

In this lab we're going to implement a simple, defensively programmed component.

The project folder for this exercise is `~/ddev-labs/error-handling`.

### 12.1 Implementing the Hasher

This is a component that will simulate malfunction.

1. Open the file `hasher/server.py`. It should contain the following simple Python code:

```
from flask import Flask, Response
import random
import uuid

app = Flask(__name__)

@app.route("/hash")
def get_hash():
    value = random.randint(0, 9)
```

```

    if value < 8:
        return Response(status=500)
    hash_value = uuid.uuid4().hex
    return Response(hash_value)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)

```

notice how the method `getHash` returns an error 80% of the time. Otherwise it returns the hash code of a random UUID value.

2. To the file `hasher/Dockerfile` add this content:

```

FROM python:2.7-alpine
RUN pip install flask
RUN mkdir /app
WORKDIR /app
COPY . /app
CMD python server.py

```

notice how we explicitly install `flask` as an external dependency of our Python app.

## 12.2 Implementing the Worker

This will be the consumer of the above hasher component. It has to be implemented in a defensive way to account for potential malbehavior of the hasher component.

1. Open the file `worker/server.py` which should have the following content:

```

import requests
import logging
from time import sleep

def get_hash(num_tries):

    n = 0
    while True:
        r = requests.get('http://hasher:8000/hash')
        if r.status_code == 200:
            logging.warning("Got hash=%s", r.text)
            n = 0
        else:
            if n == num_tries:
                logging.error("Call failed after %d retries", num_tries)
                n = 0
            else:
                logging.warn("request failed; re-trying %d", n)
                sleep((2**n)*1)
                n+=1

if __name__ == "__main__":
    logging.info(">>> Starting worker")
    get_hash(5)
    logging.info("<<< Ending worker")

```

The python worker tries to get hashes from the hasher service, but takes two precautions:

- The outer `if / else` handles bad responses from the hasher, and retries in the event of a bad response.



- The retry logic backs off exponentially, and gives up after a while. If the hasher service is unhealthy, we don't want to hammer it with requests as fast as we can loop! Instead, give it time to hopefully recover.

Also note that in all cases, plenty of logs are generated at various severity levels to create an auditable trail for when things go wrong.

2. To the file worker/Dockerfile add this content:

```
FROM python:2.7-alpine
RUN pip install flask requests
RUN mkdir /app
WORKDIR /app
COPY . /app
CMD python server.py
```

Once again we install flask as an external dependency. But this time we also install the requests library.

## 12.3 Running the application

1. To the file docker-compose.yml in the lab folder add this content:

```
version: '3.1'
services:
  hasher:
    image: hasher
  worker:
    image: worker
```

2. Build your images, and run the application as a stack:

```
$ docker image build -t worker worker
$ docker image build -t hasher hasher
$ docker stack deploy -c docker-compose.yml errordemo
```

3. Find the container ID of the worker container, and tail its logs (the first worker container might fail if it comes up before the hasher - just wait a second and the orchestrator will replace it):

```
$ docker container ls | grep worker
$ docker container logs --follow <worker container ID>
```

4. Watch the logs for a minute. Notice how retries back off exponentially.
5. When done, leave the tail with CTRL+c, and clean up your stack:

```
$ docker stack rm errordemo
```

## 12.4 Conclusion

In this lab we have implemented a component that uses defensive coding style to account for possible failures of an external component. Failed network connections are of special importance in containerized, orchestrated applications. The hasher was a toy built to fail, but even well engineered components may fail to respond to network requests. For example, the orchestrator might (re)schedule them across a cluster, and there is a lapse as connection information takes time to propagate through a cluster (as in the case of Swarm's eventually consistent gossip control plane). This behavior means that, though the service is ready and available, a given node might not find it for a period of time. Furthermore, in order to avoid swamping our networks with requests that services aren't ready to receive, always exponentially back off your retries like we did in the example above.

## 13 Builder

In this exercise we're going to demonstrate how we can avoid creating unnecessarily bloated images that can cause problems like slow container start times and unnecessary consumption of bandwidth and disk space.

The project folder for this exercise is `~/ddev-labs/builder`.

1. The file `python/server.py` in the project folder contains a simple **Hello World** app in Python:

```
if __name__ == "__main__":
    print "Hello World."
```

2. There is also a file `python/requirements.txt` containing a (random) list of dependencies for our Python "Hello World" application:

```
flask
django==1.9
# TODO: add more dependencies
```

3. Open the file `python/Dockerfile` and add the instructions necessary to build the **Hello World** app into a container image:

```
FROM python:2.7
RUN mkdir /app
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . /app
CMD python server.py
```

4. Build the image:

```
$ cd ~/ddev-labs/builder
$ docker image build -t python-app-large python
```

5. List the image just built and observe its size:

```
$ docker image ls | grep python-app-large
```

As you can see, the image is using 700 MB for 'hello world'. But we can do better.

6. Modify the `Dockerfile` and change the `FROM` statement such as that we use an Alpine based base image instead:

```
FROM python:2.7-alpine
...
```

7. Build the image again:

```
$ docker image build -t python-app-small python
```

8. And now compare the sizes of the two images:

```
$ docker image ls | grep 'python-app-'
```

We reduced the size by nearly 7 times just by selecting the "right" base image, which itself was based off of a minimal linux distro.

### 13.1 Optimize the Dockerfile

Various techniques exist on how to optimize a `Dockerfile` such as that smaller image sizes or faster builds result.

### 13.1.1 Collapse layers and clean up

In this exercise we're going to show the difference in size we can achieve by correctly installing libraries.

1. To the file `libraries/Dockerfile` add this content:

```
FROM centos:7

RUN yum -y update
RUN curl --silent --location https://rpm.nodesource.com/setup_7.x | bash -
RUN yum -y install nodejs
RUN yum -y install gcc-c++ make
```

2. Build the image:

```
$ docker image build -t libraries-large libraries
```

3. List the image and discover its size:

```
$ docker image ls | grep libraries-large
```

Once again, the image size is > 600 MB (for scale, compare to a base alpine image of about 4 MB).

4. Now we optimize the Dockerfile for image size by rearranging the installation of the libraries and by cleaning up after ourselves. Change the `libraries/Dockerfile` to look like this:

```
FROM centos:7

RUN yum -y update && \
    curl --silent --location https://rpm.nodesource.com/setup_7.x | bash - && \
    yum -y install nodejs && \
    yum -y install gcc-c++ make && \
    yum clean all
```

5. Build the image:

```
$ docker image build -t libraries-small libraries
```

6. List and compare the two image sizes:

```
$ docker image ls | grep 'libraries-'
```

The difference is significant; we have decreased the size by ~50%. Given that we installed the same things in both cases, why is the second image so much smaller?

### 13.1.2 Decrease Build Times

Here we are going to optimize the Dockerfile in such a manner that we achieve optimal build times.

1. Add the following to `ordering/Dockerfile`:

```
FROM python:2.7-alpine
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
RUN apk update
RUN apk add curl
RUN apk add net-tools
EXPOSE 3000
CMD python server.py
```

2. Build the image, and time the build process:

```
$ cd ~/ddev-labs/builder
$ time docker image build -t ordering-slow --no-cache ordering
```

It takes a while to build the image without the cache since some Python libraries and some other tools need to be installed.

- Now change the message in the `ordering/server.py` file to “Hello wonderful world!” and save.

- Build the image again:

```
$ time docker image build -t ordering-slow ordering
```

As you notice the build takes a long time **again**. Why?

- Modify the ordering of the commands in the above Dockerfile to look like this:

```
FROM python:2.7-alpine
RUN apk update && \
    apk add curl && \
    apk add net-tools
RUN mkdir /app
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . /app
EXPOSE 3000
CMD python server.py
```

- Build the image:

```
$ time docker image build -t ordering-fast ordering
```

As you notice, the first time you build it still takes a while...

- Change something in `server.py`, e.g. add a comment and build the image again. Notice that this time the build is much faster! Analyze the build output carefully. What do you see?
- Discuss with your neighbor what exactly we have changed in the Dockerfile and why this resulted in a faster re-build when only changing code in `server.py`.
- Now change something in `ordering/requirements.txt`, e.g. remove `django` and re-build the image. What is happening during the build now, and why?

## 13.2 Defining a multi-stage build

- In multi-stage/Dockerfile, add:

```
FROM alpine:3.5
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

- Note `hello.c` in the same folder contains a trivial implementation of hello world, in C.
- Build the image and observe its (massive) size:

```
$ docker image build -t my-app-large multi-stage
$ docker image ls | grep my-app-large
```

Hello world is currently almost 200 MB. We can use a multi-stage build to keep just the executable, and throw away the build tools we don't need in production.

4. Change the Dockerfile so we have multiple stages:

```
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
RUN gcc hello.c -o bin/hello

FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

5. Build the image again and compare the size with the previous version:

```
$ docker image build -t my-app-small multi-stage
$ docker image ls | grep 'my-app-'
```

And as expected, the size of the small build is much smaller (4 MB) than the large one since it does not contain all the Alpine SDK.

6. Discuss with your peers how we could even further slim down the image. For the moment do not look at point 8 and try to find your own solution
7. Here is one possible solution:

```
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
RUN gcc -static hello.c -o bin/hello

FROM busybox
COPY --from=build /app/bin/hello /app/hello
CMD ["/app/hello"]
```

Discuss with your peers what changed and why in relation to our previous Dockerfile. How big is the image size now? Did you find an even better solution?

## 13.3 Conclusion

In this exercise we saw some of the patterns that can dramatically affect image size and build times. In addition to conserving sheer size on disk, minimizing image size has security benefits (everything you install in an image presents a potential security vulnerability), and performance benefits (the more layers an image has, the more work the storage drivers have to do when assembling those layers into a filesystem at runtime). In general, you will often find it optimal to have more layers in your development images in order to leverage the build cache and get faster build times as you saw above, and less layers in your production images in order to minimize their size, attack surfaces, and spool-up times. Whenever creating build environments in an image, make sure to use a multi-stage build to copy final build artefacts into lightweight, production-ready images that do not include things like compilers or SDKs.

## 14 Creating a Docker Swarm

In this exercise we're going to create a Docker Swarm in the environment on AWS provided to you. Later, we'll use this as the foundation for setting up a build pipeline.

### 14.1 Creating a clean Slate

Before we start we want to make sure we are not having some dangling resources or wrong configurations on our VMs.

1. SSH into each of the VMs `ucp-manager-[0-2]` and `ucp-node-[0-1]`.

```
$ ssh centos@<public IP address of VM>
```

Where `<public IP address of VM>` is the respective public IP address of the respective VM provided to you in the email mentioned above.

2. When asked for the password enter the password provided to you in the email.
3. Once logged in execute:

```
$ docker version
```

Which version of Docker are we running? Docker EE Platform 2.0 requires at least 17.06.2-ee.

4. Now execute:

```
$ docker info
```

What OS and kernel is this VM running?

5. Run the following script on each of your five nodes:

```
$ docker swarm leave -f
$ docker container rm -f $(docker container ls -aq)
$ docker image prune -f -a
$ docker volume prune -f
$ docker network prune -f
```

Some of these will produce warnings if there are no swarms to leave or containers to delete - that's ok, as long as there's nothing left over afterwards. Discuss with your peers what the above script exactly does.

### 14.2 Creating a new Swarm

1. SSH into the VM called `ucp-manager-0`.
2. Init a new swarm:

```
$ docker swarm init
```

The output should look similar to this

```
Swarm initialized: current node (vul0zkipowds1jp69bouwg4qyk) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
  --token SWMTKN-1-4mu0xfuw1kvrvaqjic5z8n1ql1t1ozuxfmhbu7hvuks6rtvqq-738ceutv99raja1b4wjkii4i \
  10.0.7.126:2377
```

To add a manager to this swarm, run `'docker swarm join-token manager'` and follow the instructions.

- Copy the `docker swarm join...` command and SSH into each of the remaining 4 VMs and execute the join command. The response to this command should be:

```
This node joined a swarm as a worker.
```

- Back on `ucp-manager-0` run this command:

```
$ docker node ls
```

You should see something like this:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER S
71wwtm33kqb2zgtr1sbn0g8c	ucp-node-0	Ready	Active	
r37xbzd19q52g412j3ub535s4	ucp-manager-2	Ready	Active	
wqski57vivfklmn6v74gevp04	ucp-node-1	Ready	Active	
z6u0utum14mtbvstn2pg0pw2v *	ucp-manager-0	Ready	Active	Leader
zklms519tdk6q88325rwtxu5p	ucp-manager-1	Ready	Active	

This indicates that we have one manager node (`ucp-manager-0`) and 4 worker nodes in our swarm. At the moment, this swarm is not highly available; if `ucp-manager-0` goes down, we'll lose our swarm management capabilities.

- Make your swarm highly available by promoting `ucp-manager-[1-2]` into the management consensus. From `ucp-manager-0`:

```
$ docker node promote ucp-manager-1 ucp-manager-2
```

Both new managers should report as `Reachable` in the node list now, indicating that they are healthy members of the management consensus (but are not the manager leader).

## 14.3 Conclusion

In this lab you have created a swarm on AWS consisting of 3 manager nodes and 2 worker nodes. This Docker swarm is going to be used by subsequent labs as a build pipeline and deployment environment.

## 15 Routing Mesh

Docker Swarm makes its own scheduling decisions, at least by default. When we deploy a container that's meant to be reached from an external network, we don't want our load balancers to have to adapt to swarm scheduling decisions. Swarm's routing mesh frees us from this concern, by load balancing traffic received on a configurable port on *any* node in the swarm to the service to which that swarm port has been allocated.

### 15.1 Publishing a Service

- From one of your manager nodes, create a service:

```
$ docker service create --name whoami -p 8000:8000 training/whoami
```

- Identify on which node the service task is running:

```
$ docker service ps whoami
```

- On the node found above, try to access the service:

```
$ curl -4 localhost:8000
```

the service will respond with the hostname of the container it is running in, similar to:

```
I'm 396bb4c062b1
```

4. Try the same command on any other node on which the service is NOT running. The result is the same; the traffic is transparently routed to the service no matter which swarm node receives the request.

## 15.2 Scaling the Service

1. Now let's scale the service to 3 instances. On a manager node run:

```
$ docker service update --replicas=3 whoami
```

2. On any node run the curl command several times:

```
$ for i in {1..10}; do curl -4 localhost:8000; done;
```

in my case the output looks like this:

```
I'm 25e2ef82cc54
I'm 396bb4c062b1
I'm 87db25c0d4cb
I'm 25e2ef82cc54
I'm 396bb4c062b1
I'm 87db25c0d4cb
I'm 25e2ef82cc54
I'm 396bb4c062b1
I'm 87db25c0d4cb
I'm 25e2ef82cc54
```

3. Note how each request is load balanced across the 3 running instances in a round robin fashion.
4. Run the above command on any other node of the swarm and analyze the result.

## 15.3 Cleanup

1. On a manager node run the following command to remove the `whoami` service:

```
$ docker service rm whoami
```

## 15.4 Conclusion

In this lab we have verified that the swarm's routing mesh load balances requests from any node to any node that has a running instance of the published service (a running container). This is true even if the node that receives the request does not have a running instance on it itself.

Note the similarity of this behavior to the service discovery functionality you saw earlier. In both cases, traffic is routed to services by that service's VIP. In the service discovery exercise, it was Docker's internal DNS resolving service names to VIPs for routing traffic between containers running on a swarm; in this case, it was an IP virtual server integrated into the Docker engine on each host in the swarm which received traffic from the external network on a port allocated swarm-wide to the service, and routed that traffic to the service's VIP.

## 16 Secrets

In this exercise, we'll build an app that consumes GitHub's API. In order to make authenticated requests, we'll provision the app with an auth token protected by Docker's secrets functionality.

The project folder for this exercise is `~/ddev-labs/secrets`.



## 16.1 Setting up a demo

1. Inspect `secrets/app.py`, a minimal Flask app to send a request to GitHub's API:

```
from flask import Flask, render_template
import requests, json, os

app = Flask(__name__, template_folder='.')

@app.route('/')
def homepage():

    r = requests.get(
        'https://api.github.com/user/repos'
    )
    return render_template('repos.html', repos=json.loads(r.text))

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

2. `repos.html` dumps whatever it is handed to a webpage:

```
<html>
  <body>
    <div>{{repos}}</div>
  </body>
</html>
```

3. Dockerfile contains the minimal steps necessary to stand up your Flask app:

```
FROM python:2.7-alpine
RUN mkdir /app
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . /app
CMD python app.py
```

4. Build and run your new image (don't forget to expose port 5000), and visit it at `http://localhost:5000`. We get a 'requires authentication' complaint. We don't ever want sensitive information like login credentials written to an image, so we'll use Docker's secrets functionality to securely provision them.

## 16.2 Getting GitHub access credentials

1. Login to GitHub
2. Go to the following link: <https://github.com/settings/tokens/new>
3. Name your token `secret-demo`
4. Check the `public_repo` box. This token will only be authorized to interact with public repos.
5. Click "Generate Token". Your token is created and presented; make sure to save this before navigating away from the page, as there is no way to get it back afterwards.

## 16.3 Creating and using a Secret

1. Make sure your Docker engine is running in Swarm Mode; feel free to use the Swarm you've created in a prior exercise or initialize a new one.

2. Place your secret access token in a file called `mytoken.dat`.
3. Create a secret to be encrypted and managed by the swarm:

```
$ docker secret create access_token mytoken.dat
```

4. Modify `app.py` to set the Authorization header:

```
...
authtoken = open(os.environ['AUTHTOKEN'], 'r').readline().strip()

headers = {
    "Authorization": "token " + authtoken
}

r = requests.get(
    'https://api.github.com/user/repos',
    headers=headers
)
return render_template('repos.html', repos=json.loads(r.text))
...
```

`app.py` should look like this gist: <http://bit.ly/2wJEfg2>.

Note our application logic is looking for the auth token it needs in a file at the path indicated by the environment variable `AUTHTOKEN`, which we will have to define as part of our application configuration (see below).

5. Since you are starting the service in a swarm, the image has to be reachable from any node. Rebuild your image, and call it `<DockerID>/git_secret`. Push the image to Docker hub after you build it.
6. Launch your application as a service, mount your secret into it, and define the environment variable your application expects:

```
$ docker service create \
  --secret access_token \
  -e AUTHTOKEN='/run/secrets/access_token' \
  -p 5000:5000 <DockerID>/git_secret
```

7. Visit your application at `localhost:5000`. This time, the application successfully authenticates and a json description of all your repositories is presented.
8. Stop your service.

## 16.4 Mocking Secrets

As we saw above, secrets are mounted by the Swarm manager into a file inside your container at `/run/secrets/secret_name`. During development, we might not run Docker Swarm Mode on our development laptop, and therefore need to mock the presence of a secret in a container locally.

1. Turn off swarm mode on your local machine or remote desktop:

```
$ docker swarm leave --force
```

2. Write a `docker container run...` command to run your `git_secret` image as a single container with the secret it expects mocked. Think about what you know about mounting files inside a container's filesystem to achieve this. Once the container is running, make sure you get the behavior you expect.
3. Delete the container once it is working correctly.

## 16.5 Using Secrets in Kubernetes

1. Open the file `kube.yaml` and `kube-ns.yaml` and analyze their content. Specifically pay attention to how the secret is defined.

2. Build and push an image based on the Dockerfile in the secrets directory (make sure you're logged in to Docker Hub):

```
$ export DOCKER_ID=<your DOCKER ID>
$ docker image build -t ${DOCKER_ID}/ddev-secrets-kube:1.0 .
$ docker push ${DOCKER_ID}/ddev-secrets-kube:1.0
```

3. Use `kubectl` to create the namespace on Minikube:

```
$ kubectl create -f kube-ns.yaml
```

4. Use `kubectl` to create the secret:

```
$ kubectl create -n ddev secret generic github-token \
  --from-file=./mytoken.dat
```

5. Create the application using the secret on Minikube:

```
$ kubectl create -f kube.yaml
```

6. Test the application:

```
$ export KUBE_IP=$(minikube ip)
$ export APP_PORT=$(kubectl get -n ddev svc/app \
  --template '{{range .spec.ports}}{{.nodePort}}{{end}}')
$ curl ${KUBE_IP}:${APP_PORT}
```

you should see a list of your GitHub repositories formatted as JSON string.

7. Cleanup:

```
$ kubectl delete ns/ddev
```

## 16.6 Conclusion

In this exercise, you created a simple application that needs sensitive information in the form of an auth token. You provisioned this auth token without ever including it in your image via Docker's secrets functionality, and saw a typical pattern of using secrets in application logic:

- Application logic consumes secret from a file with path set in an environment variable `$FOO`
- `$FOO` is set to `/run/secrets/secret_name` in the service definition (at the command line as above, or in a `docker-compose.yml`), since this is where the Swarm manager will mount the secret called `secret_name`.

By following this pattern, you can use Docker to store encrypted secrets in the Swarm manager group, and pass them encrypted over the network to the containers that need them at runtime.

Next, you saw a typical pattern for mocking secrets in development. Simply place your secret in a file, mount it in the container at the same route the Swarm manager would, and your application logic can consume it in the same way.

All enterprise grade orchestrators provide similar secrets functionality; secrets can be provisioned in much the same way in Kubernetes as they were in a Swarm.

## 17 Configuration Management

In this exercise we are going to extend the Java API of the workshop sample such as that it uses a bunch of environment variables, and allow for different secrets configuration in development versus production environments.

The project folder for this exercise is `~/ddev-labs/configuration-management`.

## 17.1 Planning Environment-Specific API Config

1. Have a look in `api/Dockerfile`; the final image is based off of `java:8-jdk-alpine`. This java base image corresponds to the **application base image**, likely provided to us by our operations team. We expect any company-wide general config to have already been captured there.
2. Build the API image as is:

```
$ cd ~/ddev-labs/configuration-management/api
$ docker image build -t api:release .
```

This corresponds to our **release image** - it has our application on top of our enterprise base image, but no environment-specific config has been included yet.

3. Have a look in `api/src/main/java/com/docker/ddev/configuration/JpaConfiguration.java`. There should be a block that begins:

```
// Set password to connect to postgres using Docker secrets.
try(BufferedReader br = new BufferedReader(new FileReader("/run/secrets/postgres_password"))) {
...
}
```

Our API relies on a secret to be able to access our Postgres database. As discussed in the secrets section, it may be convenient to mock the presence of this secret in development, in case we aren't running Swarm mode on our dev machines.

4. Make a development environment image for your API that mocks the presence of a secret in its nominal location by copying the file with your postgres password into the `api` folder:

```
$ cp ../devsecrets/postgres_password .
```

And create a Dockerfile called `Dockerfile-dev` that places this file where it would if it was a Swarm-managed secret, in an image based on your release image:

```
FROM api:release
ADD postgres_password /run/secrets/postgres_password
```

Another option in this situation would be to use the same image for development and production, but mount a secret file into `/run/secrets/postgres_password` at runtime to mock the secret in the dev image. This has the advantage of only creating one image for both environments, but the disadvantage that that config (mounting the volume) has to be repeated each time the application is run in the dev environment. The approach followed in this step pushes that config as far 'upstream' as possible, to avoid repetition later, at the cost of image flexibility.

5. Build this development environment image:

```
$ docker image build -f Dockerfile-dev -t api:dev .
```

6. Open the class `TestController.java` in the folder `/api/src/main/java/com/docker/ddev/controller`. In this class we have a `showConfig` method that listens on the endpoint `/utility/showConfig` and returns the content of some environment variables:

```
...
public ResponseEntity<?> showConfig() {
    logger.info("Performing showConfig");

    JSONObject config = new JSONObject();
    config.put("APP_ENVIRONMENT", System.getenv("APP_ENVIRONMENT"));
    config.put("IMAGES_DIRECTORY", System.getenv("IMAGES_DIRECTORY"));
    config.put("PDF_DIRECTORY", System.getenv("PDF_DIRECTORY"));
    config.put("MIN_STOCK_COUNT", System.getenv("MIN_STOCK_COUNT"));
    return new ResponseEntity<JSONObject>(config, HttpStatus.OK);
}
...
```

We'd like to control these variables dynamically at startup.

7. To the file `configuration-management/env-dev.conf` add the following definitions:

```
APP_ENVIRONMENT=DEV
IMAGES_DIRECTORY=/some/folder/for/dev/images
PDF_DIRECTORY=/some/folder/for/dev/pdf
MIN_STOCK_COUNT=2
```

8. Copy the file `configuration-management/docker-compose.yml` to a new file `docker-compose-dev.yml`:

9. Open the file `docker-compose-dev.yml` and add the following at the end of the definition of the `api` service:

```
...
env_file:
  - env-dev.conf
...
```

10. Run the application with the configuration file:

```
$ docker-compose -f docker-compose-dev.yml up --build
```

11. Test the configuration:

```
$ curl localhost:8080/api/show-config
```

and you should see something like this (reformatted for readability):

```
{
  "PDF_DIRECTORY": "/some/folder/for/dev/images",
  "IMAGES_DIRECTORY": "/some/folder/for/dev/pdf",
  "APP_ENVIRONMENT": "DEV",
  "MIN_STOCK_COUNT": "2"
}
```

At this point, we have defined a hierarchy of images, from application base image, to release image, to development environment image that capture increasing levels of specificity at each tier. Also, we have successfully captured the environment variables for our developer environment in a configuration file referenced by a compose file dynamically at startup.

## 17.2 Preparing Deployment to other Environments

Next, let's take advantage of the hierarchy we've set up to quickly and easily prepare to deploy to a production environment.

1. In production, we want to use a real secret, not the mock in our `api:dev` image. In this case, there's no special config we need to insert into the image, since Docker's secrets functionality will do this for us. Simply retag our release image:

```
$ docker image tag api:release api:prod
```

(We'll see later how Docker Enterprise Edition can do such 'image promotion' for us as part of an automated CI/CD chain).

2. Start up swarm mode if you haven't already, to use as our 'production environment', and then create a secret for use by our production app:

```
$ docker swarm init
$ echo "gordonpass" | docker secret create postgres_password -
```

3. Define a new environment variable file `env-prod.conf`:

```
APP_ENVIRONMENT=PROD
IMAGES_DIRECTORY=/some/folder/for/production/images
```

```
PDF_DIRECTORY=/some/folder/for/production/pdf
MIN_STOCK_COUNT=5
```

4. Create a file `docker-compose-prod.yml` based on the existing `docker-compose.yml` that uses your new `api:prod` image, provisioning it with the secret and production environment-specific config file you just created.
5. Test the production version of your application to make sure it works as expected.

## 17.3 Configuration Management in Kubernetes

We have already seen how secrets are handled in Kubernetes. Let's now see how we can use ConfigMap objects to configure our application.

1. In `kube.yaml`, notice the definition of a ConfigMap object:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: dev-config
  namespace: ddev
data:
  APP_ENVIRONMENT: DEV
  APP_IMAGES_DIRECTORY: "/some/folder/for/dev/images"
  APP_PDF_DIRECTORY: "/some/folder/for/dev/pdf"
  APP_MIN_STOCK_COUNT: "2"
```

We import this into a pod by name in the same file:

```
envFrom:
- configMapRef:
  name: dev-config
```

2. Create the objects in Minikube:

```
$ kubectl apply -f kube.yaml
```

Note that the container will run in the pod executing this command: `/bin/sh -c "env"` which will output the list of all environment variables as seen by the container.

3. Now get the (sorted) log from the the container we just ran:

```
$ kubectl -n ddev logs po/config-test-pod | sort
```

it should look similar to this:

```
APP_ENVIRONMENT=DEV
APP_IMAGES_DIRECTORY=/some/folder/for/dev/images
APP_MIN_STOCK_COUNT=2
APP_PDF_DIRECTORY=/some/folder/for/dev/pdf
HOME=/root
HOSTNAME=config-test-pod
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
...
```

as you can see, the `envFrom` key in our `kube.yaml` points to batches of environment variables to be defined in a running container, as defined by ConfigMap objects.

4. Clean up:

```
$ kubectl delete ns/ddev
```

## 17.4 Optional Challenge: Packaging and Shipping

Now that we've created two versions of our application for two different environments, let's package it up so we can deploy it easily on a remote datacenter.

1. Create a git repo with two branches `dev` and `prod`, put the appropriate versions of your app in each, and push to GitHub.
2. Tag all the images your datacenter will need to pull appropriately, and push them to Docker Hub.
3. Log into your first AWS node (`ucp-manager-0`), and pull your project from GitHub.
4. Stand up your application configured for the dev environment.
5. Shut down your dev deployment, and stand up your application for the prod environment.

## 17.5 Conclusion

In this exercise, we defined a simple hierarchy of images and config files that capture configuration at a few different steps. As a rule of thumb, configuration should live at the most general level of this hierarchy that makes sense. For example, it wouldn't make sense to mock your secret in your upstream release image, since the production image will mount a real secret instead. However, it's not necessary to put off mocking that secret to startup or runtime for your developer environment, so the correct place to mock that secret was in the developer environment image.

By the same token, we left environment variables out of our images and defined them at startup in `env.conf` files in Swarm or via `ConfigMap` objects in Kubernetes, to maintain flexibility in their definition as long as possible. We could go a step further and push config definition all the way to runtime, by providing our containers with whatever information they need to query outside services (Hashicorp vaults, Consul KV stores...) dynamically while running.

## 18 Installing UCP

### 18.1 Prerequisites:

A Docker swarm as described in lab **Creating a Docker Swarm** running on Docker 17.06.2-ee or later (EE is important).

### 18.2 Installing UCP

1. SSH into `ucp-manager-0` of your swarm.
2. Run the following script to install UCP (make sure to change `<admin-password>` to whatever password you want for your UCP deployment):

```
$ UCP_IP=<ucp-manager-0 IP>           #Public IP address of ucp-manager-0
$ UCP_FQDN=<ucp-manager-0 FQDN>       #FQDN of the ucp-manager-0
$ docker container run --rm -it --name ucp \
  -v /var/run/docker.sock:/var/run/docker.sock \
  docker/ucp:3.0.2 install \
  --admin-username admin \
  --admin-password adminadmin \
  --san ${UCP_IP} \
  --san ${UCP_FQDN}
```

Note, this will take a while since the UCP installer will recognize all nodes of your swarm and configure them accordingly. Sometimes the manager nodes will throw warnings or errors while they establish consensus. Wait a minute or two, and the manager consensus will converge.

3. Once installed, open a browser at `https://<UCP-IP>` and login to UCP using your admin username and password. You'll need to upload a license for UCP. If you don't yet have one, get a free trial license at [store.docker.com](https://store.docker.com).
4. Click on **Nodes** on the left hand side and make sure you can see the 5 nodes of your cluster and that they are all in status **Healthy UCP manager** or **Healthy UCP worker**.
5. Please also note that UCP has transparently installed Kubernetes on your swarm too. Each worker node can now be used to either host swarm or Kubernetes workload or both. The latter is not recommended in production though.

## 18.3 Uninstalling UCP

**STOP:** Only do this if your installation failed and you need to start over!

If for some reason we need to uninstall UCP we can use the following commands.

1. Remove UCP:

```
$ docker container run --rm -it --name ucp \  
-v /var/run/docker.sock:/var/run/docker.sock \  
docker/ucp:3.0.2 uninstall-ucp \  
--id <UCP ID>
```

**Note:** to find out the <UCP ID> we can first run the above command without the `--id` parameter. The command will fail and tell us which ID to use.

2. Clean up orphaned resources:

```
$ docker system prune -f
```

## 18.4 Conclusion

In this lab you have installed UCP on an existing Docker swarm; in fact, all of UCP's functionality is provided by containers running across your swarm. UCP automatically recognized and configured all nodes of the pre-existing swarm, but you can also add more worker and managers to UCP just like you did for a regular swarm, at any time. UCP will automatically recognize and configure the new nodes as you join them to the swarm.

# 19 Layer 7 Load Balancing

As developers we are introducing new services all the time. If those service have a public API then we must make sure they can be accessed from the outside world, which in practice will be separated from our swarm by a load balancer. The layer 4 mesh net we studied earlier successfully decouples that load balancer from our orchestrator's scheduling decisions, but reconfiguring our load balancer to route traffic to a new port every time we expose a new service is a hassle. Instead, we can use application level load balancing (also called layer 7 load balancing) to ask UCP to route traffic to our services based on header information rather than port in those requests. In that case, traffic to all such services can be recieved by our swarm on a single port.

## 19.1 Pre-requisites:

- A Docker swarm as described in lab **Creating a Docker Swarm** running on Docker 17.06.2-ee or later (EE is important).
- UCP as described in **Installing UCP**.



## 19.2 Enabling HTTP Routing Mesh

1. From a UCP admin account, navigate to **Admin** -> **Admin Settings** -> **Routing Mesh**.
2. Tick the checkbox which says **Enable Routing Mesh**.
3. Specify port 3080 in the **HTTP port** field and port 3443 in the **HTTPS port** field, and click **Save**.
4. Close the **Admin** dialog box.
5. Under the category **Swarm** click on **Networks** on the left navigation bar. Verify that a network called **ucp-interlock** has been created.

## 19.3 Deploy Services with Routing Labels

### 19.3.1 Creating the Service from UCP

1. Navigate **Services** -> **Create Service**, and then:
  1. On the **Details** tab enter webapp in the **Name** field and nginx:latest in the **Image** field.
  2. On the **Network** tab click **Publish Port** +. Fill in 80 in the **Internal Port** and 8000 in the **Public Port** field. Make sure that **Publish Mode** is Ingress.
  3. Click **Add Hostname based routes** + and select http:// for **External Schema** and enter nginx.example.org in the **Routing Mesh Host** field.
  4. Now click **Confirm**.
  5. Click **Attach Network** + and select the ucp-interlock network.
  6. Click **Create** on the bottom right of the page to create the service.

2. Test that traffic is routed to this service by the Host value of its http header. From any node in the swarm, do:

```
$ curl -v --header "Host: nginx.example.org" -4 localhost:3080
```

You should see the html for the nginx landing page. Note that we hit this page at the http port we configured for the routing mesh above.

### 19.3.2 Creating the Service from the CLI

1. On ucp-manager-0 create an overlay network demo just to keep the traffic to Interlock isolated and secure:

```
$ docker network create -d overlay demo
```

2. Still on ucp-manager-0 create a new service using this command:

```
$ docker service create --name demo \
  --network demo \
  --label com.docker.lb.hosts=web.example.com \
  --label com.docker.lb.port=8000 \
  training/whoami:latest
```

Please note the --label parameters where we define the external route and the container port.

3. Test that traffic sent to the same host and port as above gets routed to this service instead, when we set the http Host header accordingly:

```
$ curl -v --header "Host: web.example.com" -4 localhost:3080
```

The container responds with its container ID. Hit nginx.example.org one more time to show that both these services are receiving the correct traffic from the layer 7 routing mesh.

## 19.4 Conclusion

In this exercise, we set up the layer 7 load balancing, and saw it in action. Note that Interlock consists of a set of 3 swarm services (core, extension and proxy), and traffic is routed to the Interlock proxy exactly like traffic going to any exposed service: via the layer 4 swarm routing mesh. Only once traffic arrives at the Interlock proxy container is the header read for the Host: value, and the packet routed to the corresponding service. In this way, we only have to make one fixed configuration in our load balancer (to direct traffic to the Interlock service), and can then dynamically deploy apps with different load balancing labels and have traffic successfully routed to them.

## 20 Installing Docker Trusted Registry

In this exercise we're going to install Docker Trusted Registry (DTR) on our existing UCP swarm.

### 20.1 Prerequisites:

- A Docker swarm as described in lab **Creating a Docker Swarm**.
- UCP as described in **Installing UCP**.

### 20.2 Installing DTR

1. SSH into the UCP manager leader (ucp-manager-0). We'll install DTR on the same node as UCP, for a lightweight developer environment. In production, each would have a dedicated node, but there's no need for that in a testing environment.
2. Run the following script:

```
$ DTR_FQDN=<ucp-manager-0 FQDN>
$ UCP_IP=<ucp-manager-0 public IP>
$ docker container run -it --rm docker/dtr:2.5.3 install \
  --ucp-node ucp-manager-0 \
  --ucp-username admin \
  --ucp-password adminadmin \
  --ucp-url https://{UCP_IP} \
  --ucp-insecure-tls \
  --replica-https-port 4443 \
  --replica-http-port 81 \
  --dtr-external-url https://{DTR_FQDN}:4443
```

Those replica ports are necessary for installing DTR on the same nodes as UCP, to avoid port collisions. If DTR had dedicated nodes, we could leave the defaults of 80 and 443.

### 20.3 Trusting DTR

All communication from each swarm node to DTR happens via TLS. Thus each node must trust DTR by downloading the CA certificate of DTR and updating the local CA trust.

1. Still on ucp-manager-0, run the following script to make the node trust DTR:

```
# Specify the fully qualified domain name (FQDN) of DTR (e.g. dtr.example.com)
$ DTR_FQDN=<DTR FQDN>
# Download the DTR CA certificate
$ sudo curl -k https://{DTR_FQDN}:4443/ca -o /etc/pki/ca-trust/source/anchors/{DTR_FQDN}:4443.crt
# Refresh the list of certificates to trust
$ sudo update-ca-trust
```

```
# Restart the Docker daemon
$ sudo /bin/systemctl restart docker.service
```

2. Login into DTR:

```
docker login ${DTR_FQDN}:4443
```

And use your UCP admin credentials (admin / adminadmin by the defaults above) to login.

3. Repeat the above step for each node of the swarm.

## 20.4 Testing DTR

1. Open a browser tab at `https:<DTR_FQDN>:4443` and login with your UCP admin credentials.
2. Create a repository admin/alpine.
3. Back on the UCP controller node pull the Alpine image:

```
$ docker image pull alpine:latest
```

4. Re-tag the image so that it can be pushed to DTR:

```
$ docker image tag alpine:latest $DTR_FQDN:4443/admin/alpine:latest
```

5. Push the image to DTR:

```
$ docker image push $DTR_FQDN:4443/admin/alpine:latest
```

6. In the DTR UI double check that the image has been pushed by navigating to the **Images** tab of the repository admin/alpine.

## 20.5 Uninstalling DTR

**STOP:** Only do this if you have screwed up you installation and need to start over with DTR!!!

If we need to uninstall DTR then we can use this command:

```
$ docker container run -it --rm docker/dtr:2.5.3 destroy \
  --ucp-username admin \
  --ucp-password adminadmin \
  --ucp-url https://${UCP_IP} \
  --ucp-insecure-tls \
  --replica-id <replica ID>
```

**Note:** To find out the replica ID we can first run the above command without the `--replica-id` parameter. The command will fail and tell us the replica ID that was found and should be used.

## 20.6 Conclusion

In this exercise we have installed DTR. We have installed it on the same node as UCP. This is OK for non-production systems. In a production environment it is recommended to install DTR in high availability mode on 3 worker nodes of the swarm, and to configure independent, highly available storage backing for your images. What we have here is suitable for demo and testing purposes only.

We also have configured each node of our swarm to trust DTR and to test the integration we have pushed an image from a swarm node to DTR.

## 21 Content Trust

This exercise focuses on understanding and securing image distribution. We'll start with a simple `docker image pull` and build up to using Docker Content Trust.

### 21.1 Pulling Images by Tag

1. The most common and basic way to pull a Docker image is by a tag. For example to pull an image by the tag `edge`:

```
$ docker image pull alpine:edge
```

Run this command now, and it will pull the Alpine image tagged as `edge`. The corresponding image can be found in Docker Store (<https://store.docker.com/images/alpine?tab=description>). If no tag is specified, Docker will pull the image with the `latest` tag.

2. List the images on your node:

```
$ docker image ls
```

You should see `alpine:edge` as one of your local images.

### 21.2 Pulling Images by Digest

Pulling by tag is easy and convenient. However, tags are mutable, and the same tag can refer to different images over time. For example, you can add updates to an image and push the updated image using the same tag as a previous version of the image. This mutability means there is no *a priori* guarantee of what content a given tag will refer to, and that content can change from pull to pull.

This is why pulling by **digest** is such a powerful operation. Thanks to the content-addressable storage model used by Docker images, we can target pulls to specific image contents by pulling by digest.

1. Pull the Alpine image with the `sha256:b7233...` digest:

```
$ docker image pull \
  alpine@sha256:b7233dafbed64e3738630b69382a8b231726aa1014ccaabc1947c5308a8910a7
```

2. List your images as in the previous section, and confirm that you have an `alpine` image with the desired sha. Notice that there are now two Alpine images in the Docker host's local repository. One lists the `edge` tag. The other lists `<none>` as the tag along with the `b7233daf...` digest. The content-addressable storage model used by Docker images means that any changes made to an image results in a new digest for the updated image. This means it is not possible for two images with different contents to have the same digest.

### 21.3 Using Docker Content Trust

Digests are not a human-friendly way to label an image. We'd like to be able to encode things like version numbers and base layers into the names of our images, which is how image tags are most commonly used. But how do we ensure the integrity of an image when we aren't specifying it by digest?

Enter Docker Content Trust: a system currently in the Docker Engine that verifies the publisher of images without sacrificing usability. Docker Content Trust implements The Update Framework (TUF), an NSF-funded research project succeeding Thandy of the Tor project. TUF uses a key hierarchy to ensure recoverable key compromise and robust freshness guarantees.

Docker Content Trust handles name resolution from image tags to image digests by signing its own metadata. When Content Trust is enabled, Docker will verify the signatures and expiration dates in the metadata. If the verification is successful, the pull request will be replaced with a pull request using a digest instead.

1. Enable Docker Content Trust by setting the `DOCKER_CONTENT_TRUST` environment variable:

```
$ export DOCKER_CONTENT_TRUST=1
```

It is worth noting that although Docker Content Trust is now enabled, all Docker commands remain the same. Docker Content Trust will work silently in the background.

2. Pull the riyaz/dockercon:trust signed image:

```
$ docker image pull riyaz/dockercon:trust
```

Look closely at the output of the `docker image pull` command and take particular notice of the name translation — how the command is translated to the digest as shown below:

```
Pull (1 of 1): riyaz/dockercon:trust@sha256:88a7...
sha256:88a7...: Pulling from riyaz/dockercon
fae91920dcd4: Pull complete
Digest: sha256:88a7...
Status: Downloaded newer image for riyaz/dockercon@sha256:88a7...
Tagging riyaz/dockercon@sha256:88a7... as riyaz/dockercon:trust
```

3. Try pulling an unsigned image:

```
$ docker image pull riyaz/dockercon:untrust
```

You cannot pull unsigned images with Docker Content Trust enabled. Once Docker Content Trust is enabled you can only pull, run, or build with trusted images.

### 21.3.1 Understanding The Update Framework (TUF)

1. Make your own tag of the alpine image, and push it to DTR:

```
$ docker image tag alpine:latest ${DTR_FQDN}:4443/admin/alpine:trusted
$ docker image push ${DTR_FQDN}:4443/admin/alpine:trusted
```

You should see some dialog prompts like the following:

```
$ docker image push $DTR_FQDN:4443/admin/alpine:trusted
The push refers to a repository [ec2-54-161-32-69.compute-1.amazonaws.com...
cd7100a72410: Layer already exists
trusted: digest: sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e493cb39abb...
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 2c49bce:
Repeat passphrase for new root key with ID 2c49bce:
Enter passphrase for new repository key with ID 1051ad9 (ec2-54-161-32-69...
Repeat passphrase for new repository key with ID 1051ad9 (ec2-54-161-32-6...
Finished initializing "ec2-54-161-32-69.compute-1.amazonaws.com:4443/admi...
Successfully signed "ec2-54-161-32-69.compute-1.amazonaws.com:4443/admin/...
```

This command prompts you for passphrases. This is because Docker Content Trust is generating a hierarchy of keys with different signing roles. Each key is encrypted with a passphrase, and it is best practice to provide different passphrases for each key.

The **root key** is the most important key in TUF as it can rotate any other key in the system. The root key should be kept offline, ideally in a hardware crypto device. It is stored in `~/.docker/trust/private/root_keys` by default.

The **targets** key is the only local key required to push new tags to an existing repo. It is stored in `~/.docker/trust/private/tuf_keys` by default.

2. Explore the `~/.docker/trust` directory to view the internal metadata and key information that Docker Content Trust generates:

```
~/.docker$ tree
.
├── config.json
├── trust
│   ├── private
│   │   ├── root_keys
│   │   │   └── 4041d9098c09360ddfb1585f29ec52b72c6d9b6a1001287261d5c3842ced1e90.key
│   │   ├── tuf_keys
│   │   │   ├── DTR FQDN
│   │   │   ├── username
│   │   │   │   └── alpine
│   │   │   │       └── 688a5af45359e399e0ecc178f1b5c74866e8e929d516c0046e13f3ae71b0257f.key
│   │   └── tuf
│   │       ├── DTR FQDN
│   │       ├── username
│   │       │   └── alpine
│   │       │       ├── changelist
│   │       │       ├── metadata
│   │       │       │   ├── root.json
│   │       │       │   └── targets.json
```

3. Disable Docker Content Trust by unsetting the appropriate variable, and pull the image you just pushed in the previous step:

```
$ export DOCKER_CONTENT_TRUST=0
$ docker image pull ${DTR_FQDN}:4443/admin/alpine:trusted
```

4. Now do the same with Docker Content Trust re-enabled:

```
$ export DOCKER_CONTENT_TRUST=1
$ docker image pull ${DTR_FQDN}:4443/admin/alpine:trusted
```

Take note of the difference between the pull of a signed image with Docker Content Trust enabled and disabled. With Docker Content Trust enabled, the image pull is converted from a tagged image pull to a digest image pull.

## 21.4 Exploring Content Trust Metadata

1. List the metadata fetched along with your `alpine:trusted` image:

```
$ cd ~/.docker/trust/tuf/${DTR_FQDN}:4443/admin/alpine/metadata/
$ ls
root.json snapshot.json targets.json timestamp.json
```

Four pieces of signed metadata appear.

2. Examine `root.json`. This lists the public keys used to verify all the other pieces of metadata in the collection. The private root key to generate this is found at `~/.docker/trust/private/root_keys/`.
3. Examine `targets.json`. This is the signed metadata you produced when you pushed your `alpine:trusted` image, identifying the trusted content itself. Notice that the tag `trusted` appears under the `targets` key, indicating the latest signed content for the `trusted` tag of your `alpine` repo. Also notice that the `keyid` matches the `targets` key id you created when you first signed an image; the private key for this keypair is found at `~/.docker/trust/private/tuf_keys/${DTR_FQDN}:4443/admin/alpine/`.

4. Examine `snapshot.json`. Snapshot metadata is usually, as in this case, generated server side to capture a hash of the root and targets metadata to ensure they have been presented in valid combinations. While there is only one piece of targets metadata in this example, it is also possible to generate an arbitrary number of delegation keys, so that multiple signers can generate targets metadata for situations when multiple stakeholders must sign off on a given image. The snapshot key captures hashes of all of these.
5. Examine `timestamp.json`. The server-side timestamp key signs the latest valid snapshot for the target in question, with a short expiration date so that when a new snapshot is created (presumably when a patched version of the image is pushed), the old metadata expires and is prevented from being trusted.

## 21.5 Conclusion

In this exercise, you have seen how to enable and disable Docker Content Trust. You have also seen how to sign images that you push. By doing this, you enable image consumers to not only consume only images that have been appropriately signed, but they can also demand images signed only by keyholders they trust; in UCP, see **Admin Settings** -> **Content Trust** to enforce this. For more information about Docker Content Trust, see the documentation.

## 22 Image Scanning in DTR

In this exercise, you'll walk through enabling image scanning in DTR, and inspecting the results of a scan.

### 22.1 Enabling Image Scanning

1. Log into your DTR web ui as an admin, and navigate **System** -> **Security** and click the 'Enable Scanning' toggle. Click 'Enable' on the popup, without changing the default behavior.
2. Notice that a yellow warning appears saying that the 'Vulnerability Database is updating', the CVE database version number is 0, and the button below this is disabled, has a spinner, and says 'Database Updating'. All these things indicate that DTR is downloading its first CVE database. You can proceed with the exercise, but **do not push** until this database download is complete!

### 22.2 Creating a repo

1. Create a new user-owned repository called `admin/whale-test`. On the repo creation page, click **Show Advanced Settings**, and enable the 'SCAN ON PUSH' toggle. Click **Create** when done.
2. Pull the `ubuntu:16.04` image or any image that you would like to perform image scanning on:

```
$ docker image pull ubuntu:16.04
```

3. Re-tag the image with your DTR repository name and a tag of 1.0.

```
$ docker image tag ubuntu:16.04 \
  ${DTR_FQDN}:4443/admin/whale-test:1.0
```

4. **Make sure the CVE database download is complete**, and then log in and push the image to DTR:

```
$ docker login ${DTR_FQDN}:4443
$ docker image push ${DTR_FQDN}:4443/admin/whale-test:1.0
```

### 22.3 Investigating layers & components

1. Go to the repository on the DTR web UI and verify that you now have a 1.0 tag on the repository. At the moment, this tag probably reports 'Scanning' with a spinner under the Vulnerabilities column. Wait for this to finish (it can take several minutes).

2. Click 'View Details' link of your image, it will take you to 'Layers' view. Dockerfile entries are listed on the left; clicking on them will display the report for that layer.
3. Click on 'Components'; the same vulnerabilities are listed by component instead of layer, with links to the relevant CVE reports.
4. In the Components view, try clicking on **Hide** next to one of the CVE summaries; this CVE will now not be counted towards vulnerability totals for this *or any other* image in DTR. In this way, you can mark a vulnerability as not a concern.

## 22.4 Conclusion

At this point, layers pushed to the repo you created in this exercise will be scanned when they are received by DTR, and when the local CVE database is updated. These scans in turn can be set to trigger webhooks on scan completion or vulnerability detection, allowing users to integrate image scanning into CI/CD pipelines.

## 23 Image Promotion & Webhooks

1. For the following exercises disable content trust:

```
$ unset DOCKER_CONTENT_TRUST
```

2. Also define an environment variable with your Docker ID:

```
$ export USER_ID=<your Docker Store ID>
```

### 23.1 Pre-requisites:

- UCP as described in **Installing UCP**.
- DTR as described in **Installing DTR**.

### 23.2 Using Webhooks

In this exercise we will create a service that will be triggered by a Webhook defined in DTR.

#### 23.2.1 Creating the Service

First we will assemble a simple app that logs some information about POST requests received, before setting this up as a service and firing webhooks at it.

1. On your development machine, open the file `dtr-ci-cd/webhooks/src/server.js`. It is a Node JS/Express JS API with a POST endpoint defined as follows:

```
...
app.post('/',function(req,res){
  var hostname = os.hostname();
  console.log('Host: '+hostname+' got triggered with:');
  console.log('  type:      '+req.body.type);
  console.log('  createdAt: '+req.body.createdAt);
  console.log('  contents:  '+JSON.stringify(req.body.contents, null, 2));
  console.log('-----');
  console.log('Full body:  '+JSON.stringify(req.body, null, 2));
  res.send('OK');
});
...
```



2. Open the file `dtr-ci-cd/webhooks/package.json`. It should contain:

```
{
  "name": "WebhookSample",
  "version": "1.0.0",
  "description": "Demo app for Webhooks",
  "main": "server.js",
  "scripts": {
    "start": "node src/server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.14.1",
    "body-parser": "^1.9.2"
  }
}
```

3. To the Dockerfile in the `webhooks` folder add the following instructions to build the container image:

```
FROM node:8-alpine

# for edit-and-continue only
RUN npm install nodemon -g

RUN mkdir /app
WORKDIR /app
COPY package.json ./
RUN npm install
COPY ./src ./src
EXPOSE 3000
CMD ["npm", "start"]
```

4. To the file `docker-compose.yml` in the `webhooks` folder add:

```
version: '3.1'
services:
  sut:
    build: .
    volumes:
      - ./src:/app/src
    ports:
      - "3000:3000"
    command:
      nodemon ./src/server.js 0.0.0.0:3000
```

Gist: <http://bit.ly/2Gg61SR>

note the use of `nodemon` as we saw before, to create an **edit and continue** experience if we want to manipulate our app while it's running in a container. Windows users, don't forget the additional `-L` flag if necessary.

5. Now run the app to test it:

```
$ docker-compose up --build
```

6. Use a REST client such as Postman (for Chrome) to send a POST request to `localhost:3000` with the following body (see below for `curl` equivalent):

```
{
  "type": "PULL_IMAGE",
  "createdAt": "2017-06-09T09:30:43",
  "contents": {
```

```

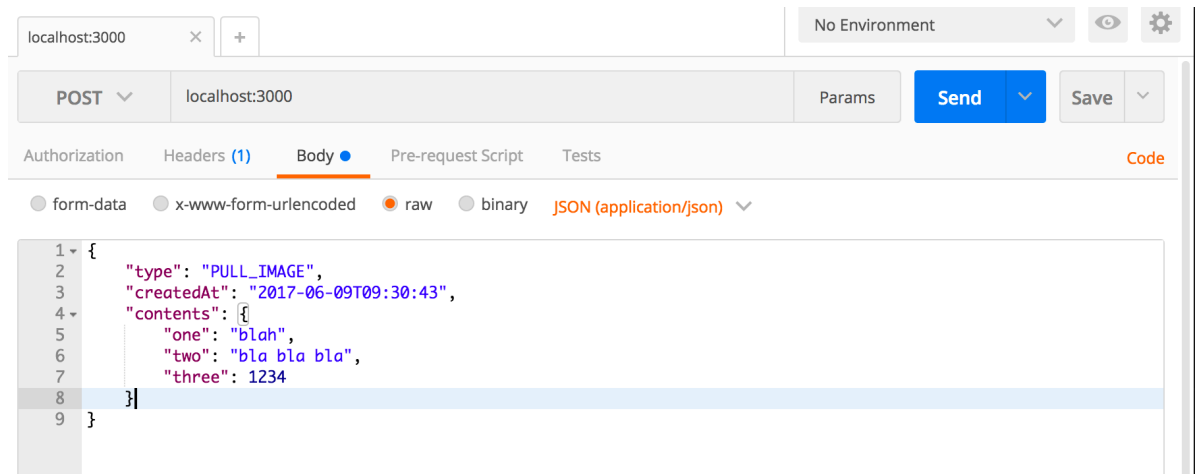
    "one": "blah",
    "two": "bla bla bla",
    "three": 1234
  }
}

```

Gist: <http://bit.ly/2uDWIHP>

and a header:

Content-Type: application/json



To do the same thing via curl, place the above json in a file payload.json and do:

```
$ curl -d @payload.json -H "Content-Type:application/json" localhost:3000
```

In the container log you should see something like this:

```

Attaching to webhooks_sut_1
sut_1 | [nodemon] 1.17.2
sut_1 | [nodemon] to restart at any time, enter `rs`
sut_1 | [nodemon] watching: *.*
sut_1 | [nodemon] starting `node ./src/server.js 0.0.0.0:3000`
sut_1 | Running at Port 3000
sut_1 | Host: c76ea851f5f7 got triggered with:
sut_1 |   type:      PULL_IMAGE
sut_1 |   createdAt: 2017-06-09T09:30:43
sut_1 |   contents: {
sut_1 |     "one": "blah",
sut_1 |     "two": "bla bla bla",
sut_1 |     "three": 1234
sut_1 |   }
sut_1 | -----
sut_1 | Full body:  {
sut_1 |   "type": "PULL_IMAGE",
sut_1 |   "createdAt": "2017-06-09T09:30:43",
sut_1 |   "contents": {
sut_1 |     "one": "blah",
sut_1 |     "two": "bla bla bla",
sut_1 |     "three": 1234
sut_1 |   }
sut_1 | }

```

7. Now we want to build and push an image of this app, so we can pull it into our UCP cluster:

```
$ docker image build -t ${USER_ID}/webhook-sample:1.0 .
$ docker image push ${USER_ID}/webhook-sample:1.0
```

- On one of your UCP managers, run a service using the above image:

```
$ export USER_ID=<your Docker ID>
$ docker service create --name webhook-sample \
  --detach=false --publish 3000:3000 \
  ${USER_ID}/webhook-sample:1.0
```

- Again use a REST client like **Postman** or **curl** to trigger the service with a test payload. This time we have to use the public IP address or DNS of any of our swarm nodes instead of localhost.
- Back on one of the UCP manager nodes, analyze the log generated by the service:

```
$ docker service logs webhook-sample
```

It should show the same sort of record as above; this service is now ready to receive webhooks from DTR.

### 23.2.2 Registering the Webhook

- Login to your DTR (UI).
- Create a repository `admin/webhook-sample`.
- Navigate to the repository details and select the tab **Webhooks**.
- Click the button **New Webhook**.
- For **NOTIFICATIONS TO RECEIVE** select the value **Tag pushed to repository** and for **WEBHOOK URL** enter `http://<UCP_FQDN>:3000`.
- Click the link **Test** and then check the service logs and find out whether or not the callback was successfully executed.
- Click **Save** back on the webhook dialog to create the webhook.

### 23.2.3 Triggering the Webhook

- Pull an image from Docker Store:

```
$ docker image pull centos:7
```

- Tag the image:

```
$ docker image tag centos:7 ${DTR_FQDN}:4443/admin/webhook-sample:1.0
```

- Push the image to DTR:

```
$ docker image push ${DTR_FQDN}:4443/admin/webhook-sample:1.0
```

- Now analyze the log of our `webhook-sample` service running in the swarm:

```
$ docker service logs webhook-sample
```

You should see log entries indicating that DTR posted a webhook to the service, for example:

```
webhook-sample.1.n7n35ci3ahun@node | Host: deb12fb57621 got triggered with:
webhook-sample.1.n7n35ci3ahun@node |   type:      TAG_PUSH
webhook-sample.1.n7n35ci3ahun@node |   createdAt: 2017-07-18T19:04:35.240532079Z
webhook-sample.1.n7n35ci3ahun@node |   contents: {
webhook-sample.1.n7n35ci3ahun@node |     "namespace": "admin",
webhook-sample.1.n7n35ci3ahun@node |     "repository": "webhook-sample",
webhook-sample.1.n7n35ci3ahun@node |     "tag": "1.0",
```

```

webhook-sample.1.n7n35ci3ahun@node | "digest": "sha256:89751557f508153f133a6f6750e87f871506c0a
webhook-sample.1.n7n35ci3ahun@node | "imageName": "admin/webhook-sample:1.0",
webhook-sample.1.n7n35ci3ahun@node | "os": "linux",
webhook-sample.1.n7n35ci3ahun@node | "architecture": "amd64",
webhook-sample.1.n7n35ci3ahun@node | "author": "admin",
webhook-sample.1.n7n35ci3ahun@node | "pushedAt": "2017-07-18T19:04:35.213817243Z"
webhook-sample.1.n7n35ci3ahun@node | }
webhook-sample.1.n7n35ci3ahun@node | -----
webhook-sample.1.n7n35ci3ahun@node | Full body: {
webhook-sample.1.n7n35ci3ahun@node |   "type": "TAG_PUSH",
webhook-sample.1.n7n35ci3ahun@node |   "createdAt": "2017-07-18T19:04:35.240532079Z",
webhook-sample.1.n7n35ci3ahun@node |   "contents": {
webhook-sample.1.n7n35ci3ahun@node |     "namespace": "engineering",
webhook-sample.1.n7n35ci3ahun@node |     "repository": "webhook-sample",
webhook-sample.1.n7n35ci3ahun@node |     "tag": "1.0",
webhook-sample.1.n7n35ci3ahun@node |     "digest": "sha256:89751557f508153f133a6f6750e87f871506c0a
webhook-sample.1.n7n35ci3ahun@node |     "imageName": "admin/webhook-sample:1.0",
webhook-sample.1.n7n35ci3ahun@node |     "os": "linux",
webhook-sample.1.n7n35ci3ahun@node |     "architecture": "amd64",
webhook-sample.1.n7n35ci3ahun@node |     "author": "admin",
webhook-sample.1.n7n35ci3ahun@node |     "pushedAt": "2017-07-18T19:04:35.213817243Z"
webhook-sample.1.n7n35ci3ahun@node |   },
webhook-sample.1.n7n35ci3ahun@node |   "location": "/repositories/admin/webhook-sample/tags/1.0"
webhook-sample.1.n7n35ci3ahun@node | }

```

Metadata about the image push is posted to the address we specified in the webhook definition.

### 23.3 Configuring Image Promotion

In addition to posting metadata triggered by events, DTR can also rename and / or re-tag an image based on similar events, in order to track its progress through your pipelines.

1. In DTR create a repository admin/webhook-sample-qa.
2. Navigate to the repository admin/webhook-sample and select the tab **Promotions**.
3. Make sure **Is Source** is selected and click **New promotion policy**.
4. From the section **PROMOTE TO TARGET IF...** select **Tag Name**. Then select **Tag name ends with** and enter `-good` as value and click **Add**.
5. Select admin/webhook-sample-qa as **TARGET REPOSITORY**.
6. Enter `%n-QA` in **TAG NAME IN TARGET**.
7. Click **Save and Apply**.

8. Tag an image such as that it will trigger promotion when pushed:

```
$ docker image tag centos:7 ${DTR_FQDN}:4443/admin/webhook-sample:1.0-good
```

9. Push the image:

```
$ docker image push ${DTR_FQDN}:4443/admin/webhook-sample:1.0-good
```

10. In DTR double check that the image has been stored in admin/webhook-sample and also been promoted to admin/webhook-sample-qa.
11. Optional: Configure a webhook for the repository admin/webhook-sample-qa and verify that it gets triggered when an image is promoted to this repo.

## 23.4 Conclusion

Taken together, webhooks and image promotion allow us to use Docker Trusted Registry as the backbone of a CI/CD pipeline. As images are pushed to DTR by a build server, DTR can fire webhooks to kick off testing pipelines, and as images pass stages in that pipeline, they can be automatically re-tagged or renamed to clearly indicate to users the stage to which they have progressed in testing. It's even possible to promote, or 'mirror', an image from one DTR to another, so that once testing is complete, the final, production ready image can be promoted into a totally separate DTR reserved only for production.

## 24 Build Server

In a typical Docker pipeline, a build agent will be responsible for pulling code and base images from their respective registries, building developers' images, and pushing them to DTR to trigger a CI/CD pipeline. In this exercise, you'll configure a containerized Jenkins to watch a GitHub repo, build images out of code it finds there, and push those images to DTR.

A lot of information has been distilled from here: <https://github.com/yongshin/leroy-jenkins>.

### 24.1 Prerequisites

1. Make sure all your UCP swarm nodes trust DTR. As a reminder, here is the commands that you would use to do so for CentOS:

```
# Specify the fully qualified domain name (FQDN) of DTR (e.g. dtr.example.com)
$ DTR_FQDN=<DTR FQDN>
# Download the DTR CA certificate
$ sudo curl -k https://$DTR_FQDN:4443/ca -o /etc/pki/ca-trust/source/anchors/$DTR_FQDN:4443.crt
# Refresh the list of certificates to trust
$ sudo update-ca-trust
# Restart the Docker daemon
$ sudo /bin/systemctl restart docker.service
```

## 24.2 Creating and Shipping the Jenkins Image

1. On one of your UCP nodes, create a folder build-server/jenkins.
2. Add a Dockerfile to the folder with this content:

```
FROM jenkins:2.60.3
USER root
RUN apt-get update \
    && apt-get install -y \
        apt-transport-https \
        ca-certificates \
        curl \
        software-properties-common \
    && curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add - \
    && add-apt-repository \
        "deb [arch=amd64] https://download.docker.com/linux/debian \
        $(lsb_release -cs) \
        stable" \
    && apt-get update && apt-get install -y docker-ce sudo \
    && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
COPY entrypoint.sh /
ENTRYPOINT /entrypoint.sh
```

we're using the base Jenkins image, and add Docker CE and a few other libraries to it.

3. Add a file entrypoint.sh to the same folder with the following content:

```
openssl s_client -connect ${DTR_IP}:443 -showcerts \
    </dev/null 2>/dev/null | openssl x509 -outform PEM | sudo tee \
    /usr/local/share/ca-certificates/${DTR_IP}.crt
sudo update-ca-certificates

/bin/tini -- /usr/local/bin/jenkins.sh
```

4. Make the above file executable:

```
$ chmod +x ./entrypoint.sh
```

5. Build the image, make the corresponding repo in DTR, and push:

```
$ docker image build -t ${DTR_FQDN}:4443/admin/my-jenkins:1.0 .
$ docker image push ${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

## 24.3 Preparing the Repository in DTR

1. Login to your DTR as admin.

2. Create a public repo called `admin/jenkins-demo`. This is where Jenkins will push built images to.

## 24.4 Preparing a Source Repo

1. Log in to GitHub, and create a public repo called `jenkins-demo`. This is where the code describing your image will be controlled.
2. On your local machine or remote desktop, make a fresh folder called `jenkins-demo`, and add a Dockerfile to it describing an image you're developing (feel free to just use the Dockerfile and `entrypoint.sh` defined above for the Jenkins image itself if you like).
3. Set this code up to push, but *don't* push yet:

```
$ cd jenkins-demo
$ git init
$ git remote add origin <your jenkins-demo GitHub repo url>
$ git add *
$ git commit -m 'code ready for pipeline testing'
```

## 24.5 Running Jenkins in the Swarm

### 24.5.1 Preparing the Swarm Node

1. SSH into the swarm node `ucp-node-1` where we'll run Jenkins.
2. Create a directory `jenkins` and a directory `ucp-bundle-admin`:

```
$ mkdir jenkins
$ mkdir ucp-bundle-admin
```

3. Create an auth token to use UCP's API from `ucp-node-1`:

```
$ UCP_IP=<UCP IP>
$ AUTHTOKEN=$(curl -sk -d '{"username":"admin","password":"adminadmin"}' \
  https://${UCP_IP}/auth/login | jq -r .auth_token)
```

4. Download and unzip a client bundle for your admin account:

```
$ cd ucp-bundle-admin
$ curl -k -H "Authorization: Bearer $AUTHTOKEN" \
  https://${UCP_IP}/api/clientbundle -o bundle.zip
$ unzip bundle.zip
```

5. Run the shell configuration script which is part of the bundle:

```
$ source env.sh
```

### 24.5.2 Creating the Jenkins Service

1. SSH into one of your UCP manager nodes.
2. Label the target node for Jenkins such that we can use constraints with the Jenkins service to place the master on this node:

```
$ docker node update --label-add jenkins=master ucp-node-1
```

3. Run a service for Jenkins:

```
$ export DTR_IP=<public IP of ucp-manager-0>
$ docker service create --name my-jenkins --publish 8080:8080 \
  --mount type=bind,source=/var/run/docker.sock,destination=/var/run/docker.sock \
```

```
--mount type=bind,source=/home/centos/jenkins,destination=/var/jenkins_home \
--mount \
type=bind,source=/home/centos/ucp-bundle-admin,destination=/home/jenkins/ucp-bundle-admin \
--constraint 'node.labels.jenkins == master' \
--detach=false \
-e DTR_IP=${DTR_IP} \
${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

4. Verify that the service is up and running using e.g.:

```
$ docker service ps my-jenkins
```

### 24.5.3 Finalizing the Jenkins Configuration

1. Open a browser window and navigate to Jenkins at `http://<UCP_FQDN>:8080`
2. Enter the initial password that you can get on the node where you had installed Jenkins by entering this command:

```
$ sudo cat ~/jenkins/secrets/initialAdminPassword
```

3. Install the suggested plugins from the initial popup window, create your admin account and click **Continue as Admin** -> **Start Using Jenkins**.

## 24.6 Configuring Jenkins Jobs

1. Login to Jenkins if not already done.
2. Create a Jenkins Job of type **Free Style** and call it **docker build and push**.
3. In **Source Code Management** -> **Git** - set repository to the url of the jenkins-demo GitHub repo you created above.
4. Set Build Triggers -> Poll SCM, and enter \* \* \* \* \* in the 'Schedule' field, to poll for code changes every minute. See <https://en.wikipedia.org/wiki/Cron> for an explanation of this scheduling syntax.
5. Add a Build Step of type **Execute Shell** and add the following script (remember to replace <DTR\_FQDN> with the FQDN of your DTR deployment):

```
#!/bin/bash
export DTR_FQDN=<DTR_FQDN>
docker image build -t ${DTR_FQDN}:4443/admin/jenkins-demo .
docker image tag ${DTR_FQDN}:4443/admin/jenkins-demo ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_
docker login -u admin -p adminadmin ${DTR_FQDN}:4443
docker image push ${DTR_FQDN}:4443/admin/jenkins-demo
docker image push ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_NUMBER}
```

6. Save the job.
7. Back on your development machine, push your jenkins-demo code to GitHub (git push origin master in the jenkins-demo folder). Wait a minute, and SCM polling in Jenkins will trigger a build and push your new image to DTR when complete.

## 24.7 Conclusion

In this exercise we set up a containerized Jenkins build server that ingests code from a version control repository, and pushes built images into Docker Trusted Registry. From this point, you can configure DTR to scan images for vulnerabilities, promote them to new repositories, and trigger webhooks to external services at each step.



## 25 Appendix: Connecting to the Remote Desktop

At times, we'd like to manipulate Universal Control Plane from our personal machines using things like the API or client bundles, or develop containerized software locally before pushing it the Docker Trusted Registry. If you are unable to install the necessary utilities on your machine for this workshop, you can use a pre-configured remote desktop.

1. Open or install a remote desktop client. Windows machines should have Microsoft Remote Desktop pre-installed; Mac users can install the same from the app store.

### 25.1 Windows Users

1. Enter the public IP of your `elk` node after opening Remote Desktop.
2. Click Connect. You'll be presented with a security warning; click 'Yes' to proceed.
3. A login form should be presented. Enter username **ubuntu**, and the SSH password your instructor provided to you for your AWS nodes. Click OK, and you'll be taken to your remote desktop.

### 25.2 Mac Users

1. Open Microsoft Remote Desktop, and click 'New'.
2. Fill in the following fields:
  - Connection Name: Docker
  - PC Name: Public IP of your `elk` node.
  - User name: `ubuntu`
  - Password: SSH password your instructor provided to you for your AWS nodes.
3. Click the red dismiss button in this form field. It is automatically saved, and you should see a connection called 'Docker' in the 'My Desktops' list. Double-click it.
4. An error message will pop up complaining about connection problems. Dismiss it by clicking 'ok', then fill in your AWS SSH password on the form that appears. Click 'ok', and you'll be taken to your remote desktop.