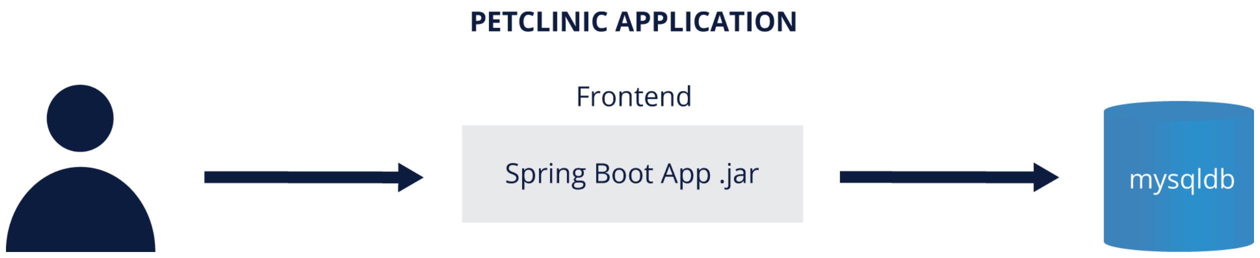
Lab 8. Automating Container Deployments with Compose

The objective of this lab exercise is to learn how to launch a collection of services for an application stack, and how to automate this process using Docker Compose. Here is what we will discuss:

- The process and the challenges of launching the application stack with more than one container using **`docker run`** command.
- How to automate that launch sequence using a simple, declarative interface which relies on YAML.
- How to create a composition of all services and connect them together using the inherent DNS-based service discovery that Docker offers along with Compose.
- What is the difference between Dockerfile and Docker Compose spec and how to tie them together.
- How to use Docker Compose to rapidly create and tear down disposable development environments.

Launching Application Stack

Let's say you have an application stack with more than one container to launch. Look at an example of the PetClinic application available on GitHub: devopsdemoapps/spring-petclinic: A Sample Spring-based Application. You may have already forked it and cloned it, so use your copy of the code for the exercise.



Running MySQL Database

Now, proceed to launch the application stack using `docker run` command. To launch a MySQL database, you will use a few environment variables. This will allow you to create the database, grant user permissions and have the frontend application connect to it later.

```
docker run -idt  -p 3306:3306 \
       -e MYSQL_ALLOW_EMPTY_PASSWORD=true \
       -e MYSQL_USER=petclinic \
       -e MYSQL_PASSWORD=petclinic \
       -e MYSQL_DATABASE=petclinic \
       mysql:5.7
```

Find out the container ID for MySQL:

```
docker ps
```

Sample output:

```
CONTAINER ID         IMAGE                   COMMAND                     CREATED
STATUS               PORTS                               NAMES
37c49c92bb1c         mysql:5.7                     "docker-entrypoint.s…"    6
minutes ago         Up 6 minutes         0.0.0.0:3306->3306/tcp, 33060/tcp
exciting_bose
```

where,

- `37c49c92bb1c`: is the container ID. Please write it down as you will be referring to it later.

Launching Frontend Application Manually

The frontend application PetClinic is a Spring Boot application built with Maven which connects to an in-memory H2 database by default. You can also have it connect to the MySQL database launched earlier as a backend, by setting up profile to `spring.profiles.active=mysql`.

Launch the PetClinic application with MySQL profile:

```
docker run -idt -p 8080:8080 -e SPRING_PROFILES_ACTIVE=mysql   <docker
hub id>/petclinic:v4
```

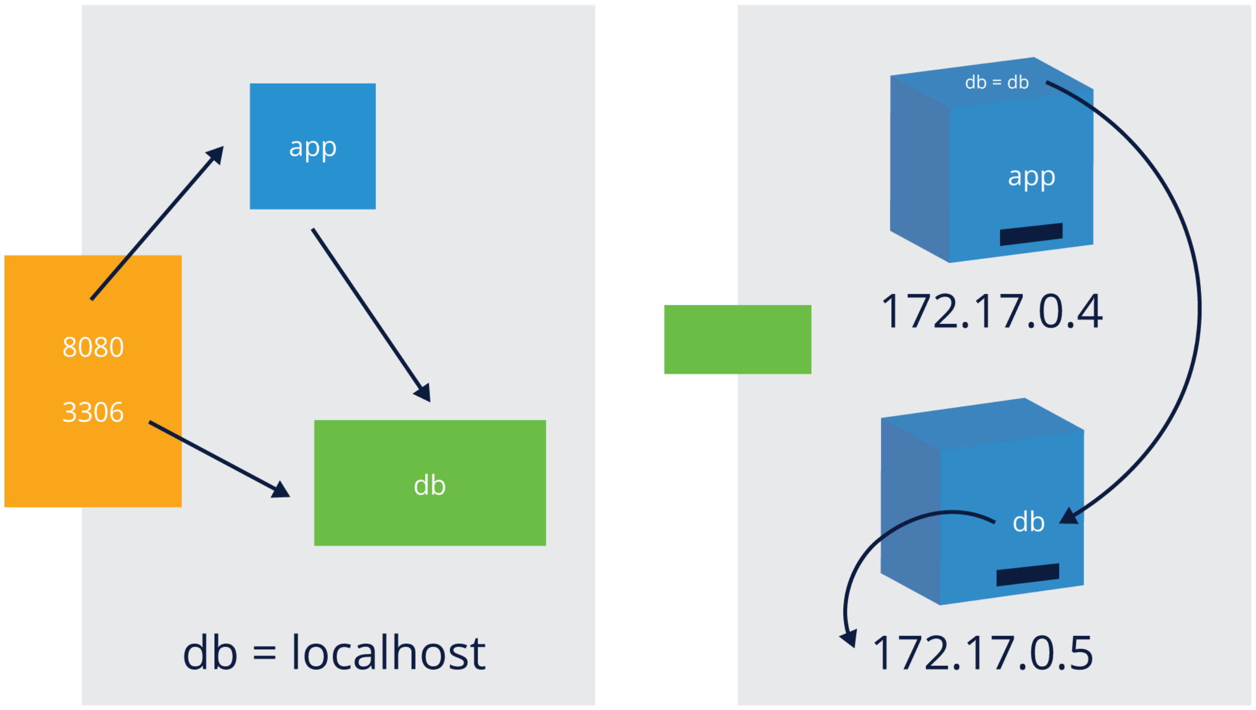
where,

- **`v4`**: the most up to date version of your image (change it to match the tag that you are using).

Find out the container ID and check the logs.

Did it work? Of course not! The reason for that is, the connection string configured in **`src/main/resources/application-mysql.properties`** is currently pointing to the **`localhost`**:

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://localhost/petclinic}
```



The above diagram illustrates two possibilities, with and without containers.

- If you run a db and an app on the same host (e.g. directly on your host), you can use **`localhost`** and connect to the database running on the same host.
- When you launch an app and a db as containers running on the same host, each of these containers will have a network stack. This means that when you point to the **`localhost`** in the connection string, it's actually pointing to the app container itself. Instead, you should be pointing to the hostname/IP address of the container which is running as a db.

One way to do this would be to use the `--link` option with `docker run` to provide a hostname alias (e.g. db) which would then point to the IP address of the db container, and use the same host alias in the connection string.

This is demonstrated below.

Update connection string inside `application-mysql.properties`.

First, edit the following file:

```
src/main/resources/application-mysql.properties
```

And update the connection string from:

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://localhost/petclinic}
```

to

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://db/petclinic}
```

where,

- `db`: hostname for the database.

This is a change inside the source code, which requires you to rebuild the image for this application. For example:

```
docker image build -f Dockerfile.multistage.v2 -t <docker hub
id>/petclinic:v5 .
```

Now, launch the PetClinic application manually and link it with the MySQL database launched previously. Use the container ID that you wrote down earlier:

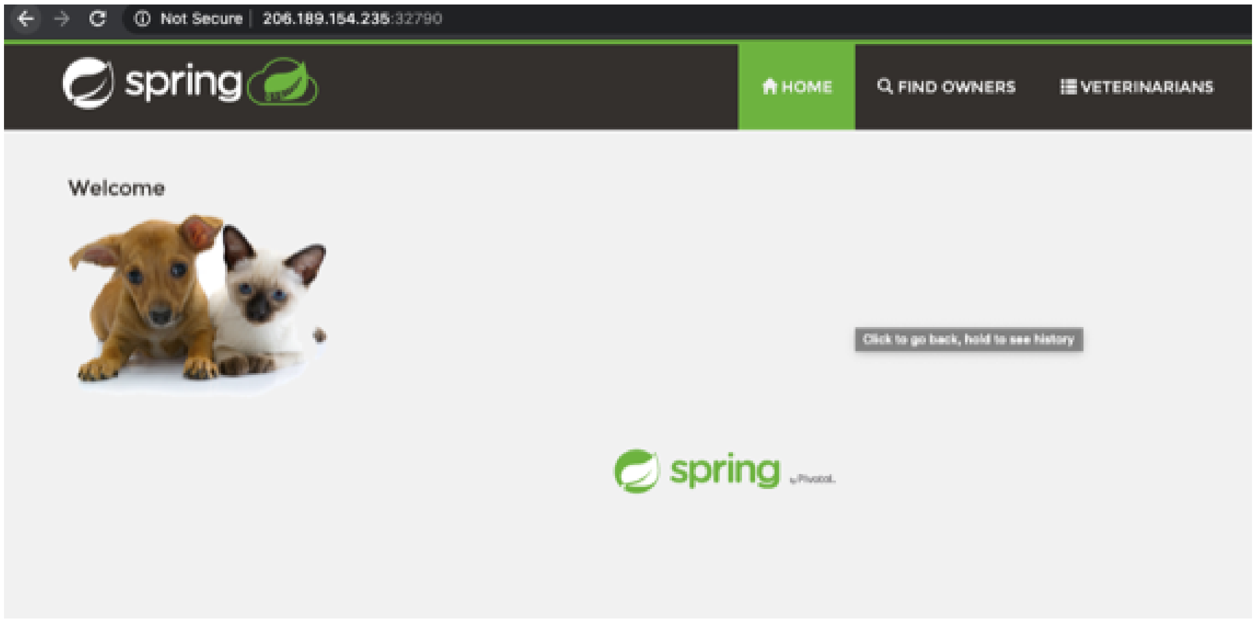
```
docker run -idt --name app -p 8080:8080 -e
SPRING_PROFILES_ACTIVE=mysql --link 37c49c92bb1c:db  <docker hub
id>/petclinic:v5
```

where,

- `37c49c92bb1c`: refers to the container which is running MySQL database.
- `db`: host alias added to `/etc/hosts` inside app container, pointing to the actual IP address of MySQL container.

Validate you are able to connect to the application using the 80 port on the host:

- http://localhost:8080 if you use Docker Desktop
- http://IPADDRESS:8080 if remote host


Not Secure | 206.189.154.235:32790
spring
HOME
FIND OWNERS
VETERINARIANS
Welcome
Click to go back, hold to see history
spring by Pivotal.

Once validated, you can delete the containers launched above using the following command:

```
docker rm -f 3d0e8ccbeeb6  37c49c92bb1c
```

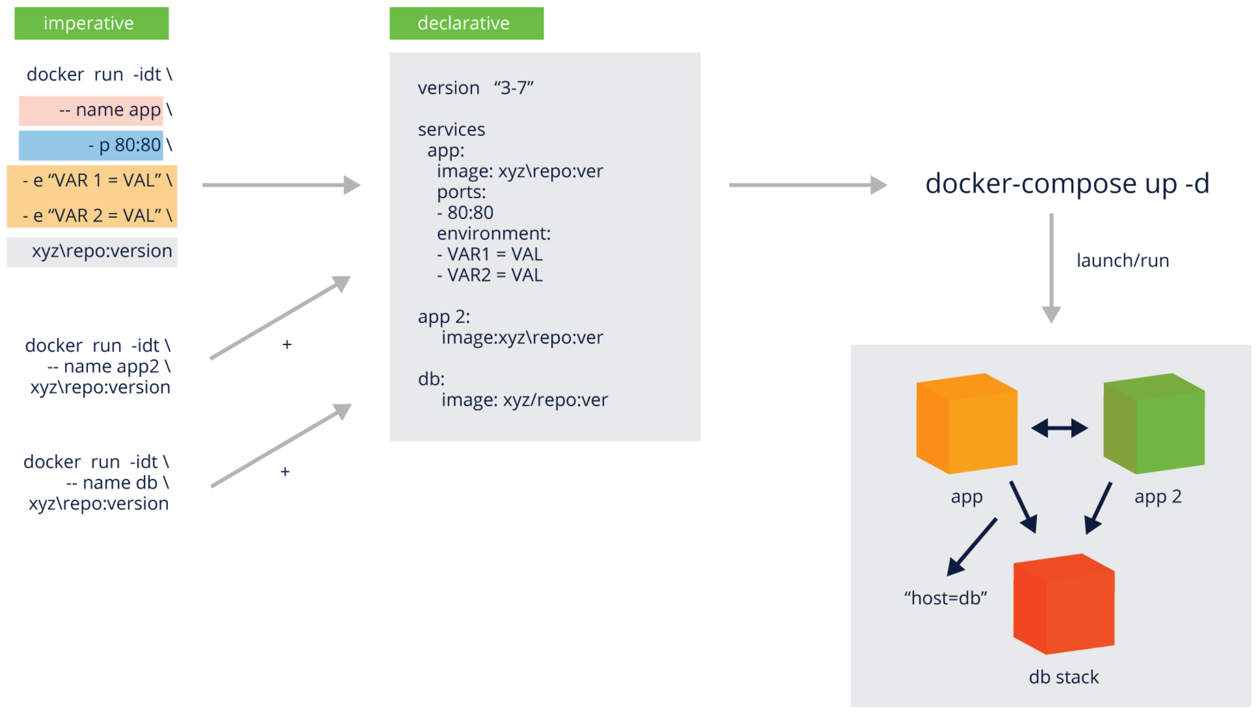
Make sure you replace `3d0e8ccbeeb6` and `37c49c92bb1c` with actual IDs of application and MySQL containers.

Composing Docker Services as a Code

You can manually launch the PetClinic application along with the database mentioned above, using a `docker run` command. Imagine you have a development team of 10 who is working on this app. Each of your teammates would have to set up this app locally to develop and test with. Here are some of the challenges that you may face:

1. Each one of you would have to remember how to launch multiple services, such as db and app. And as you start splitting this app into microservices, the list of applications you would have to launch is going to grow.
2. Also, you will either have to remember the `docker run` command with all its options, or rely on some documentation and keep track of it.
3. In addition, you may notice that during the launch of the application container above, you had to first write down the container ID of the database container, or even worse its IP address and then add it to the connection string/configurations for the app server. This creates an issue because every time you re-launch this stack, or one of your colleagues wants to replicate this setup, you will have to update this IP address. One way to solve it is to use an external service discovery, for example Consul, etcd, ZooKeeper, etc.

These reasons are good enough to consider Docker Compose. It solves all the listed issues and some more by allowing you to create a composition of services which need to work together. It also allows it to be defined as a code using a simple declarative syntax.



The same set of services that you launched earlier, e.g. app and db, can be converted into a compose file.

To better understand these concepts, watch video lessons first and try to create this file yourself. Use the following code only if you face issues and are absolutely unable to proceed.

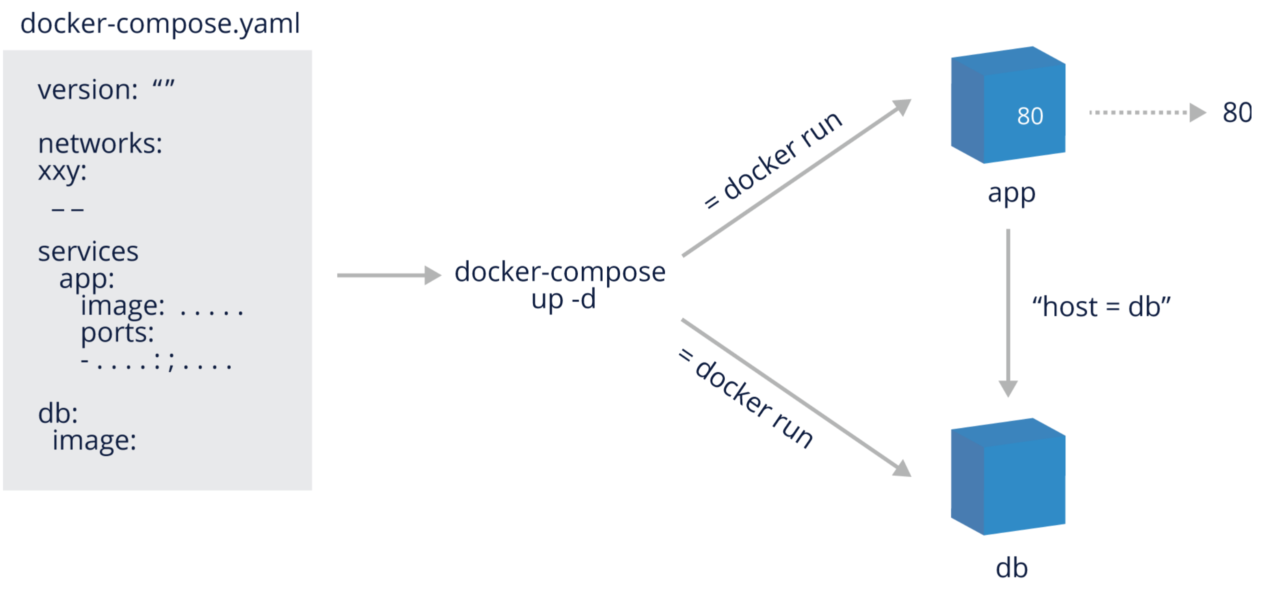
file: spring-petclinic/docker-compose.yaml

```
app:
  image: xxxxxx/petclinic:v5
  ports:
    - 8080:8080
  links:
    - db:db
  environment:
    - SPRING_PROFILES_ACTIVE=mysql
db:
  image: mysql:5.7
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=
    - MYSQL_ALLOW_EMPTY_PASSWORD=true
    - MYSQL_USER=petclinic
    - MYSQL_PASSWORD=petclinic
    - MYSQL_DATABASE=petclinic
```

***NOTE**: Replace **`xxxxxx/petclinic:v5`** with the actual tag.*

This **`docker-compose.yaml`** file is then fed to the **`docker compose`** utility, which reads it and launches your application stack:

```
cd spring-petclinic
```



Validate the syntax:

```
docker compose config
```

Check if any services launched with this Compose spec (should return blank for the first run):

```
docker compose ps
```

Launch all services in Compose spec:

```
docker compose up -d
```

When you check again, it should now show your services:

```
docker compose ps
```

Check logs for a service named **`app`**:

```
docker compose logs app
```

***NOTE**: Whenever you run **`docker compose`**, ensure that you are in the same directory as **`docker-compose.yaml`**. If that is not the case, or if the name of your Compose file is other than **`docker-compose.yaml`**, for every command you run, you need to provide a relative/absolute path to the Docker Compose file using the **`-f`** option.*

Validate that your service is launched and you are able to access it on port **`8080`** of the host using your browser.

If you get an error related to port conflicts, you may need to delete the previous containers using that port, or update the port mapping (specifically host port) in the **`docker-compose.yaml`** file.

Let's analyze what's happening:

- When you run a **`docker compose config`**, it reads the **`docker-compose.yaml`** in the current directory and checks for any syntactical errors. If errors are found, it shows the line number to help you debug.
- **`docker compose up -d`** launches containers for all applications defined in the **`services`** section. It does that in the detached mode using option **`-d`**, which is similar to the **`docker run`** option. Try running **`docker compose up`** without this option to see what happens.
- **`docker compose ps`** lists only the container specified in the Compose file. The container is prefixed with the directory name, e.g. **`spring-petclinic-app_1`** which allows **`docker compose`** to uniquely identify the containers launched from a specific path. This also means that you can use the same code and launch another instance of this stack from a different path.
- **`docker compose logs app`** will show you the logs from the container created for **app** service. If you do not provide the name of the service, consolidated log for all services will be shown.

Find out a list of sub-commands that Docker Compose supports using the following command:

```
docker compose
```

Play around with these sub-commands and learn what they do.

Refactoring Compose Spec with Version 3

What you have created earlier is version 1 of the Docker Compose spec, which has undergone significant changes over the years. Nowadays, you can add networks and volumes along with many advanced configurations, and luckily it's possible to refactor Docker Compose with version 3 specs by doing:

```
version: "3.8"

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge

services:
  app:
    image: xxxxxx/petclinic:xx
    ports:
      - 8080:8080
    environment:
      - SPRING_PROFILES_ACTIVE=mysql
    networks:
      - frontend
      - backend
    depends_on:
      - db

  db:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_USER=petclinic
      - MYSQL_PASSWORD=petclinic
      - MYSQL_DATABASE=petclinic
    networks:
      - backend
```

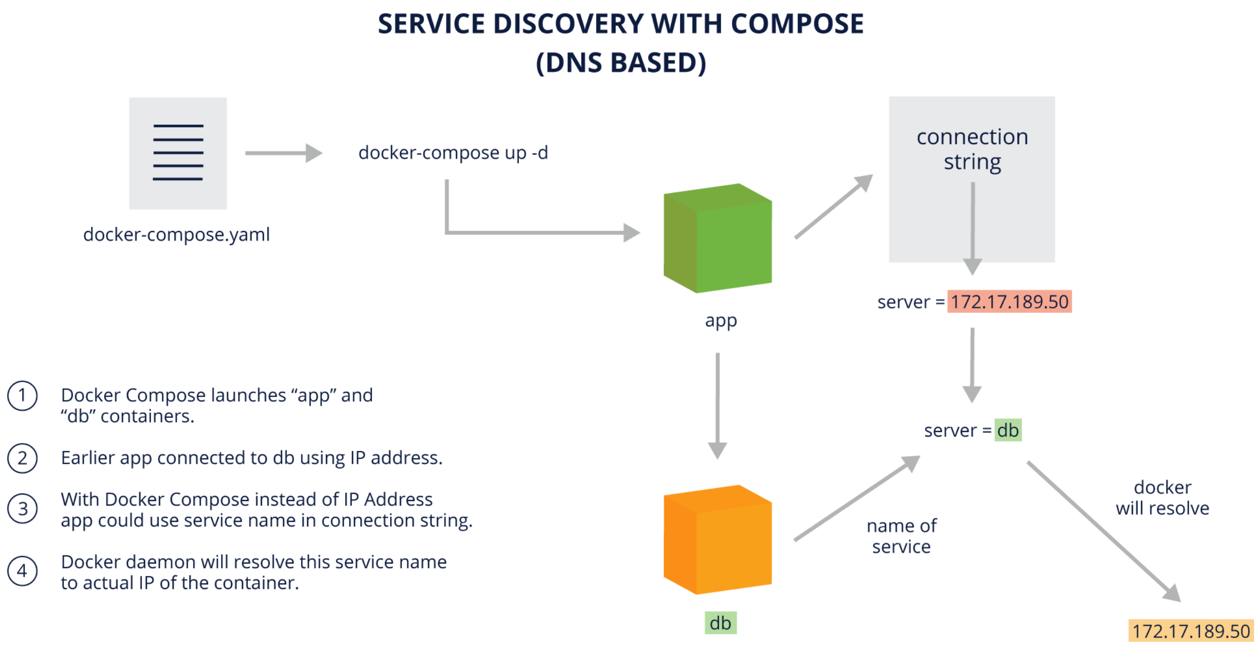
where,

- `version`: defines the Docker Compose spec version. Reference official Docker documentation to learn more about the syntax, *"Compose File"*.
- `services`: allow you to decide which applications/containers you should launch as well as define the list of services/microservices. Each service contains a code block which is nothing but a `docker run` command converted into a declarative code using YAML. It's like asking, what properties this container is created with.
- `networks`: define custom network configurations. This can then be referenced in the `service` configurations in the subsequent section.

NOTE*: You may notice port mapping removed for the db service. This is because you define the port mapping only for the services which need to be accessed from outside the Docker host. In* *this example, the only entity that needs to access db is the application server. Since the app container will be running on the same host and can access all ports of the db container, there is no need for port mapping to be defined here. This is also an advantage from the security point of view as you are reducing the attached surface of the host, by not exposing this db service.*

Service Discovery

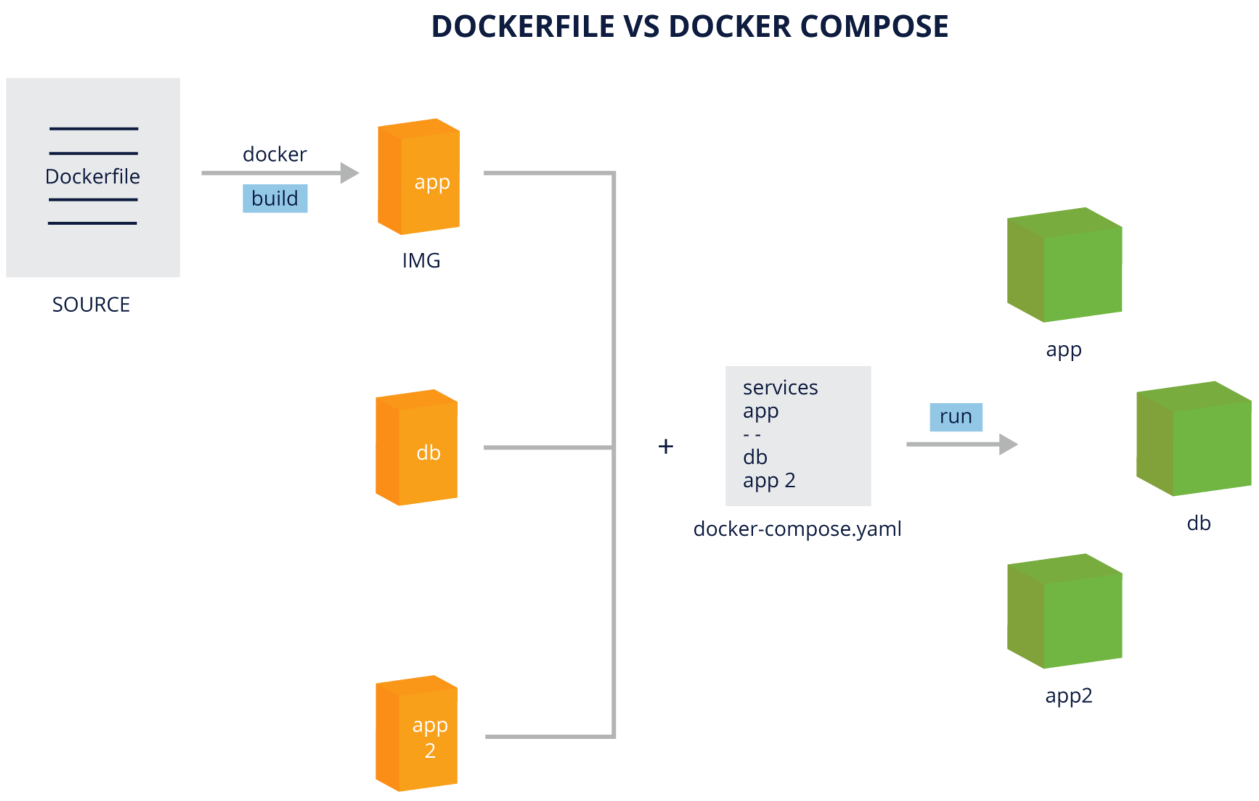
Did you notice a link was removed from the app service? You may see that it's also using db as the hostname for the database in the **`applications-mysql.properties`** file. Where does it come from? You may have already figured that out. It's the name of the service defined to launch the MySQL container. What does this mean? When you start using **`docker compose`** v3, you get service discovery built-in. That's because Docker daemon comes with a DNS service which is available while using Compose.


SERVICE DISCOVERY WITH COMPOSE
(DNS BASED)
docker-compose.yaml
docker-compose up -d
app
connection string
server = 172.17.189.50
server = db
db
name of service
docker will resolve
172.17.189.50
1 Docker Compose launches "app" and "db" containers.
2 Earlier app connected to db using IP address.
3 With Docker Compose instead of IP Address app could use service name in connection string.
4 Docker daemon will resolve this service name to actual IP of the container.

Dockerfile and Docker Compose

Both Dockerfile and **`docker-compose.yaml`** offer a declarative interface, and allow us to write code and automate processes with containerization. Now, you may wonder, what's the difference between the two, and why are they important?

The answer to these questions lies in the *when* part. Dockerfile is used during the image build process. And once you have the images built for all your applications, this is where Docker Compose comes in helping in the launch process.



You can also integrate Dockerfile with **`docker-compose.yaml`** and have Compose build the images for you as well (which is a common practice). Let's learn how to integrate these two files.

Integrating Dockerfile with Docker Compose

Let's add the image build part to the app service definition:

`file: docker-compose.yaml`

```
version: "3.7"
networks:
  petclinic:
    driver: bridge

services:
  app:
    image: xxxxxx/petclinic:dev
    build:
      context: .
      dockerfile: Dockerfile.multistage.v2
    ports:
      - 8080:8080
    environment:
      - SPRING_PROFILES_ACTIVE=mysql
    networks:
      - petclinic
    depends_on:
      - db
  db:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_USER=petclinic
      - MYSQL_PASSWORD=petclinic
      - MYSQL_DATABASE=petclinic
    networks:
      - petclinic
```

Now, after every code change, to build an image you can run:

```
docker compose build
```

This will:

- Run **`docker build`** using options provided in the Compose spec.
- Tag the image with the image tag provided in the Compose files's **`service.app.image`** spec.

To use this newly built image to launch/recreate the application container, run:

```
docker compose up -d
```

Using Docker Compose to Deploy to Dev

Following are the two "I's" of a Docker Compose deployment:

1. *Idempotent*: meaning, you can run **`docker compose up -d`** multiple times. If there are no changes in the images/code, it will not redeploy. However, as soon as it detects the image has been updated, it will redeploy.
2. *Immutable*: each time **`docker compose`** updates the application, it throws away the old container, and replaces it with a new one created using the updated image. This also means that any changes made directly to the previous container are going to be lost. Immutable deployments have many advantages though, mainly related to maintaining the consistency across all instances of the application in an environment.

These, along with the inherent nature of it being an automation tool relying on declarative code, makes Docker Compose suitable as a deployment tool for development environments. Why do I say dev environments? It’s because Docker Compose is still limited to be run on one host. It cannot launch containers crossing the boundary of the host it's running on, and spanning across multiple hosts. If you need that, for example, to operate in higher environments such as staging and production, you have to consider a Container Orchestration Engine (COE). And yes, that’s where you will start using Kubernetes. In essence, Kubernetes is the platform which takes your containers beyond one host and helps you simplify deploying and managing those containers on multiple hosts (thousands in some cases).

In this chapter though, we will focus on how to use Docker Compose as a deployment tool to dev. You can also use it along with your Continuous Integration platform such as Jenkins to set up integrated development environments which more than one developer can use, and to do some validation testing.

To see how this works, launch a stack using **`docker compose`**:

```
docker compose build
docker compose up -d
```

If you have already launched the stack with Compose, you would notice it does not change anything and shows all services being *up-to-date*. In fact, even if you have launched it for the first time, try running the above command a few times. This will help you understand what *idempotence* is all about. Docker Compose has the intelligence to know when to take action and, more importantly, when not to.

Now, update the source code to:

`file: src/main/resources/messages/messages.properties`

Change the title from:

```
welcome=Welcome
```

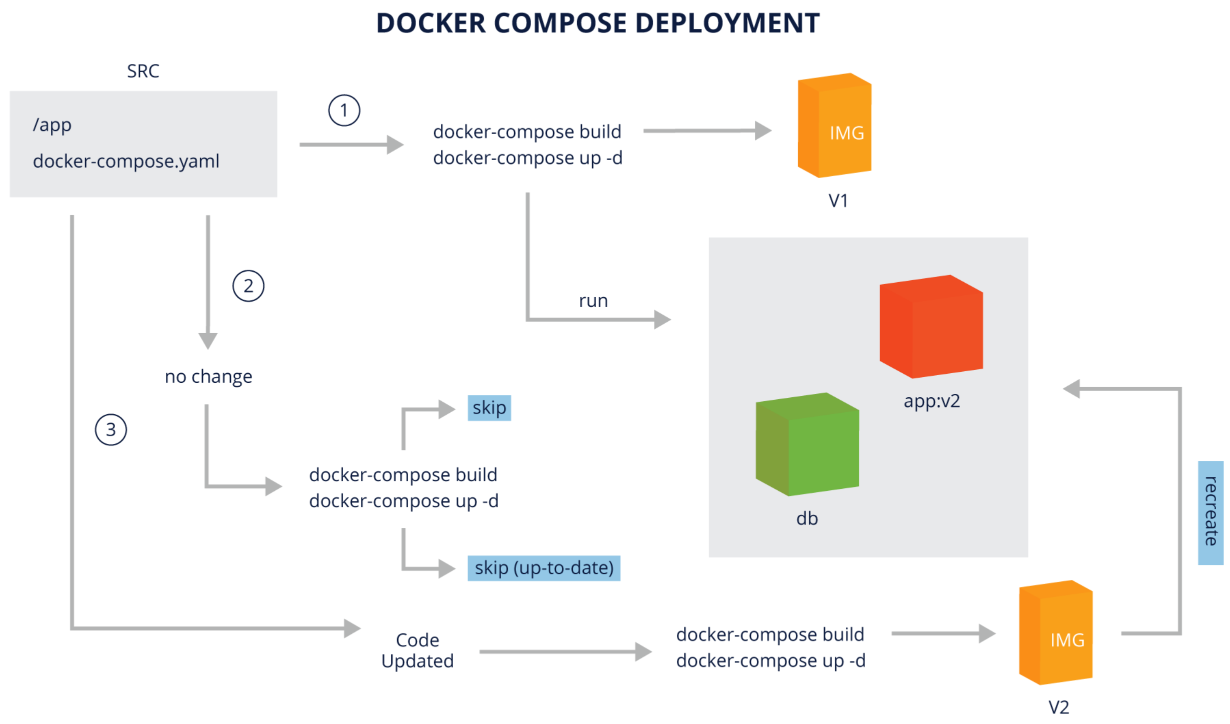
To:

```
welcome=Welcome to Petclinic
```

And then redeploy this application with:

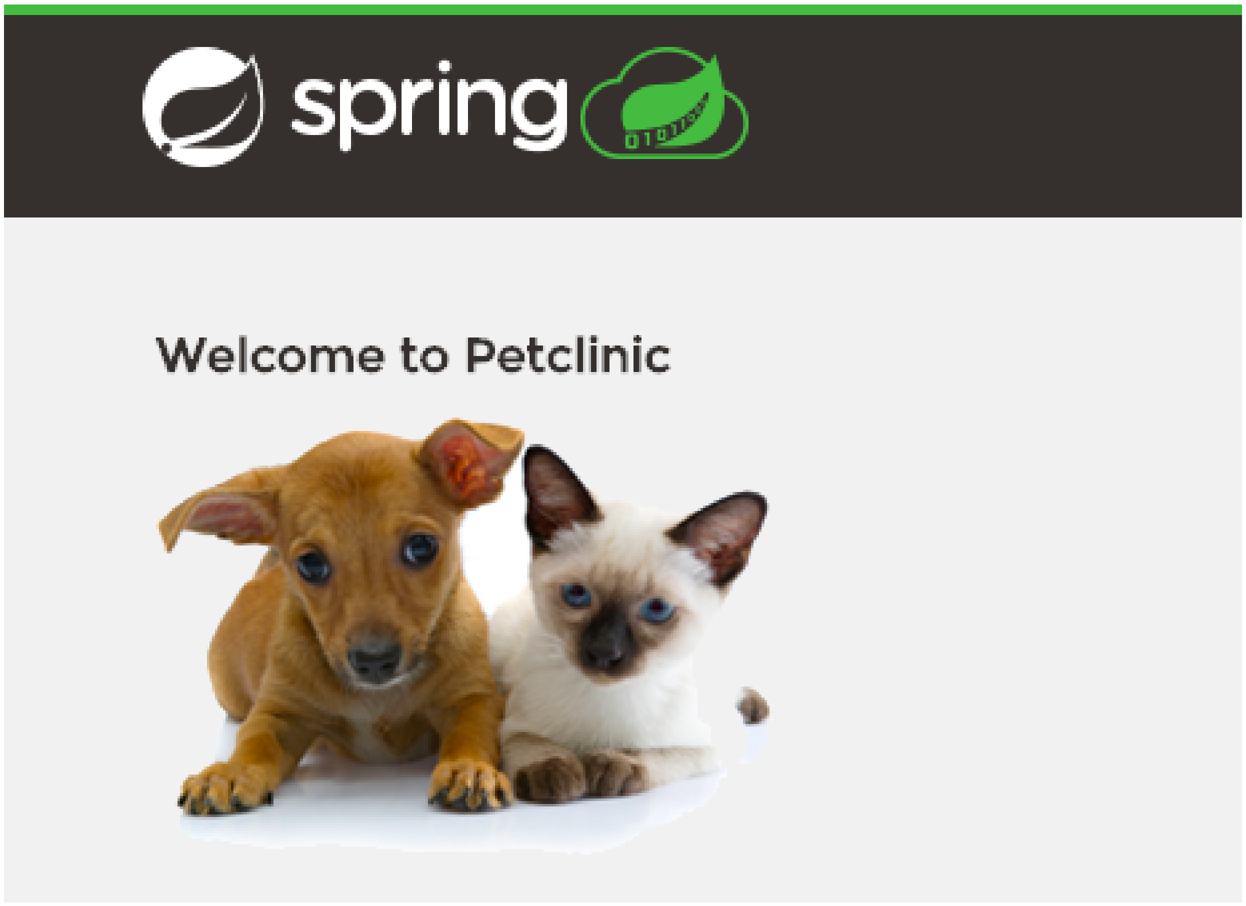
```
docker compose build
docker compose up -d
```

What happened?



- Database is not changed, so it says *up-to-date* for a db service.
- Application source was updated. Hence the Docker build process detects this change and this time launches a new image build.
- A new image is built and tagged with the same tag as earlier. This means the previous image is now untagged, orphaned and will be deleted when **`docker image prune`** is run.
- Also, the container for the app is recreated. It's an immutable deployment since the previous container is deleted and a new one launched using the newly built image.

As a result of the new deployment, if you refresh the browser tab, you should now see an updated welcome message on the home page.



Tear Down the Stack

Docker Compose is not only useful to quickly launch a dev environment, but also to tear it down and clean it up.

To stop all the services defined in the **`docker-compose.yaml`**, run:

```
docker compose stop
```

NOTE*: Whenever you run Docker Compose, ensure that you are in the same directory as* ***`docker-compose.yaml`****. If that is not the case, or if the name of your compose file is different than* ***`docker-compose.yaml`****, you need to provide a relative/absolute path to the Docker Compose file using the* ***`-f`*** *option, for every command you run. We emphasize this again as this is a very common mistake that people make.*

Services that are stopped can be started back again (with the same containers) either selectively by providing a service name (e.g. db) or all together by omitting the service names. Both commands are depicted below:

```
docker compose start db
docker compose start
```

FInally, to tear down this environment as rapidly as you started it, use either of the following commands.

For stopped services:

```
docker compose rm
```

For running services:

```
docker compose down
```

`docker compose` down is a combination of **`docker compose stop`** and **`docker compose rm`** commands. These commands remove the trace of services you had launched (except for the images pulled), which can be very useful if you want to reuse the same host to launch another application, work with it and tear it down the same way later.

Summary

In this lab, you learned how to compose more than one service to be launched together by using a simple declarative YAML interface. You also learned how to use this code to quickly set up and tear down an application stack, and use this to deploy to development environments.