Lab 4 : Automated container deployments with Compose

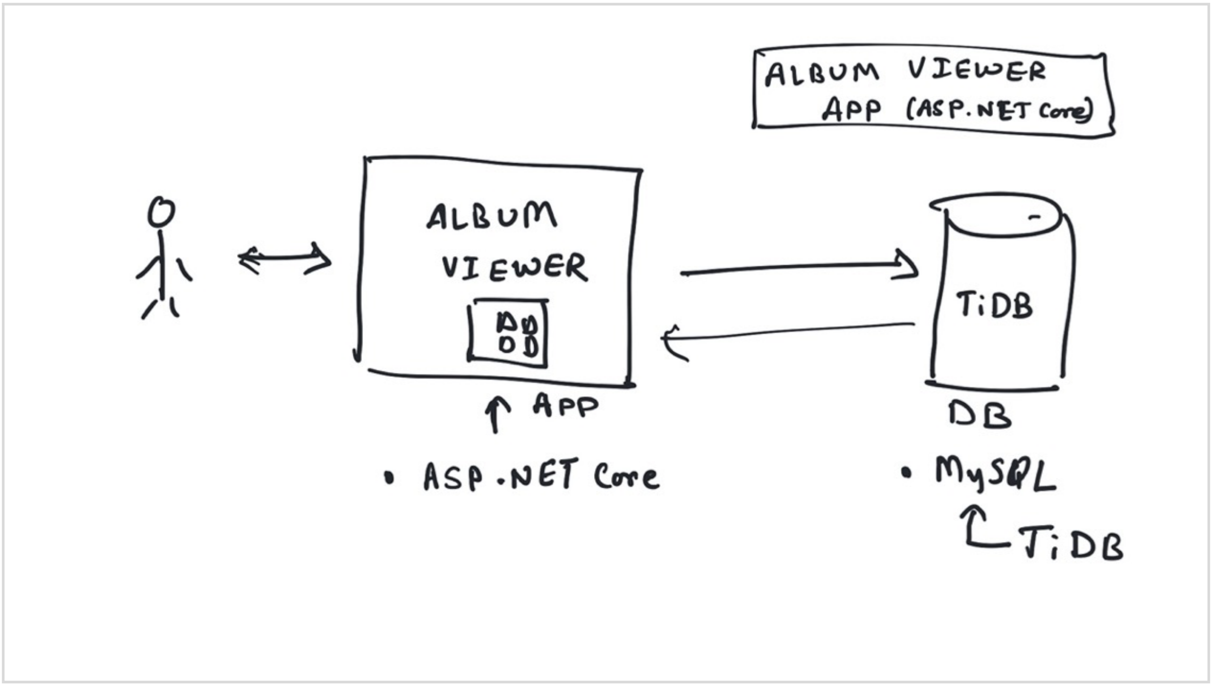
Author: Gourav Shah
Publisher: School of Devops
Version: v1.0.0

The objective of this module is to learn how to launch a collection of services together, and automate the process of doing so. And the name of the game to do so is called Docker Compose.   This is what you will learn in this chapter,

- The process and the pain points of launching the application stack with more than one containers using `docker run` commands.
- 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.
- How to use docker compose as a deployment tool for dev environments.
- How to rapidly create and tear down disposable development environments with docker compose.

Launching Containers with Compose

Lets say you have an application stack with more than one container to launch. Lets take an example of a Album Viewer Application available here GitHub - schoolofdevops/dotnet-album-viewer: West Wind Album Viewer ASP.NET Core and Angular Sample



This is a ASP .NET Core application which also uses Angular, and connects to MySQL database as a backend. For example of running this application, you would be using TiDB, which is compatible with MySQL. Try launching this application manually using the following sequence of commands,

```
docker build -t schoolofdevops/dotnet-album-viewer .


docker run --name tidb -idt --rm -p 3306:4000 dockersamples/tidb:nanoserver-
sac2016

docker inspect tidb --format="{{.NetworkSettings.Networks.nat.IPAddress}}"

docker run -itd --name app --rm  -p 80:80 -e "Data:Provider=MySQL" -e
"Data:ConnectionString=Server=172.17.187.15;Port=4000;Database=AlbumViewer;User
=root;SslMode=None" schoolofdevops/dotnet-album-viewer
```

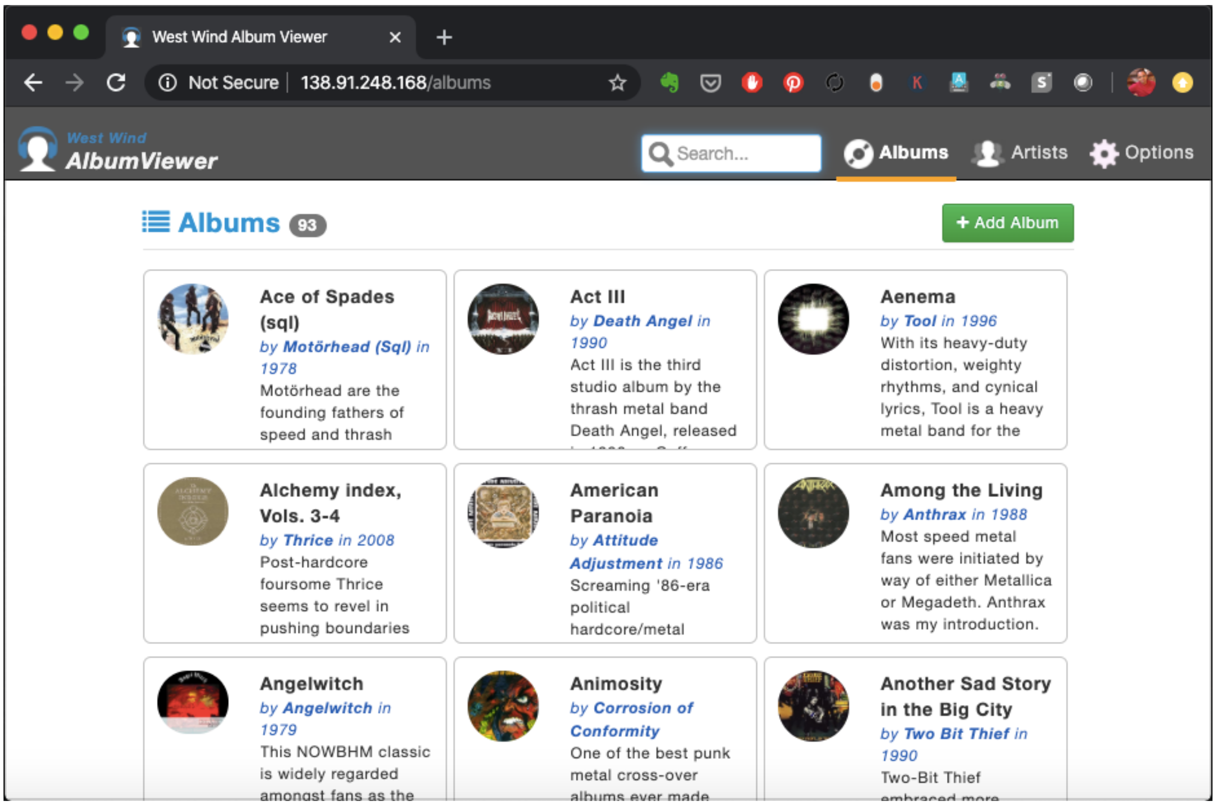
The last command should launch the Album Viewer Application. -e is used to provide connection string and provider configurations to the application as environment variables.

Replace 172.17.187.15 with the actual IP address of TiDB container and schoolofdevops/dotnet-album-viewer with the tag of your own image if built.

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

e.g.

- http://localhost:80 if using Docker Desktop
- http://IPADDRESS:80 if remote host



Albums 93

Ace of Spades (sql)
by Motörhead (Sql) in 1978
Motörhead are the founding fathers of speed and thrash

Act III
by Death Angel in 1990
Act III is the third studio album by the thrash metal band Death Angel, released

Aenema
by Tool in 1996
With its heavy-duty distortion, weighty rhythms, and cynical lyrics, Tool is a heavy metal band for the

Alchemy index, Vols. 3-4
by Thrice in 2008
Post-hardcore foursome Thrice seems to revel in pushing boundaries

American Paranoia
by Attitude Adjustment in 1986
Screaming '86-era political hardcore/metal

Among the Living
by Anthrax in 1988
Most speed metal fans were initiated by way of either Metallica or Megadeth. Anthrax was my introduction.

Angelwitch
by Angelwitch in 1979
This NOWBHM classic is widely regarded amongst fans as the

Animosity
by Corrosion of Conformity
One of the best punk metal cross-over albums ever made

Another Sad Story in the Big City
by Two Bit Thief in 1990
Two-Bit Thief embraced more

Once validated, you could delete the containers launched above using the commands below,

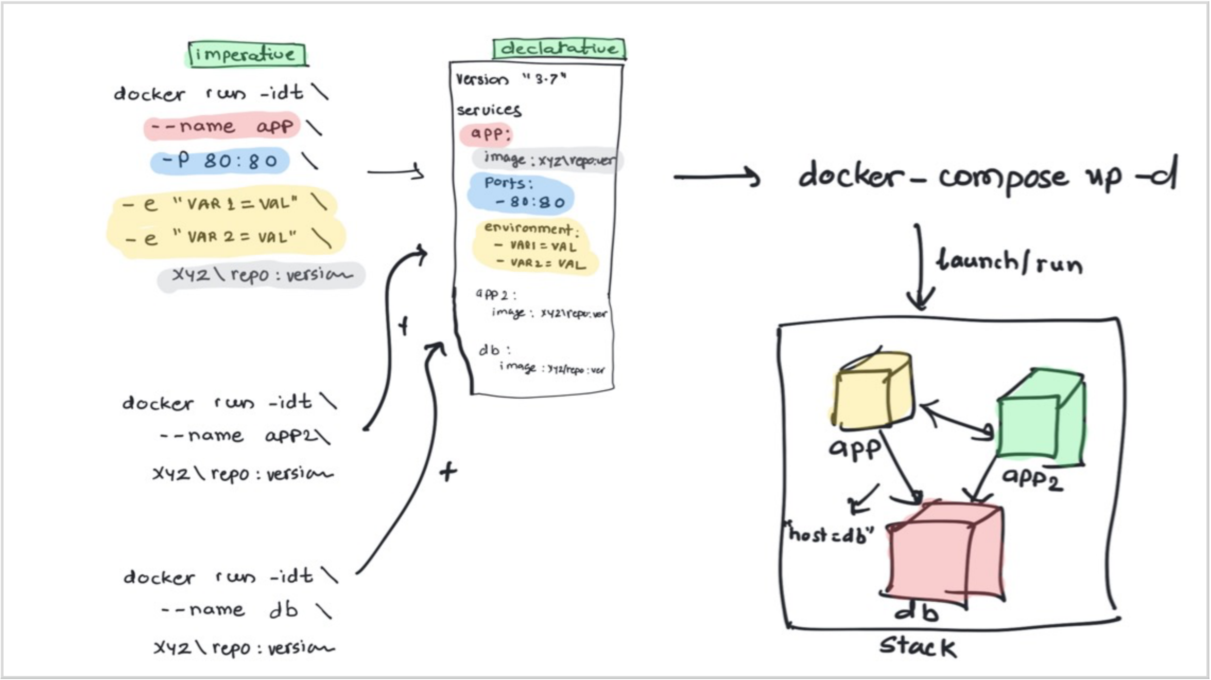
```
docker rm -f tidb app
```

Composing Docker Services as a Code

You could launch the Album Viewer application along with database above manually, using docker run commands. Imagine you have a development team of 10 who is working on this app. Each of your team mates would have to setup 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 e.g. db and app. And as you start splitting the app into micro services, the list of applications you would have to launch would grow.
2. Either you remember the `docker run` commands with all its options, or you would have to rely on a documentation such as above and keep track of it.
3. You may also notice that during the launch of the application container above, you had to first note down the IP address of the database container, and then add it to the connection string/configurations for app server. This is a problem because every time you re launch this stack, or if one of your colleagues wants to replicate this setup, you would have to update this IP address. One way to solve it is to use a external service discovery e.g. consul, etcd, zookeeper etc.

These reasons are good enough to consider **Docker Compose** which would solve all these issues and some more by allowing you to create a composition of these services which need to work together, and allow 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 **tidb**, could be converted into a compose file as below,

[To get the best learning out of this, do watch the video lessons and try to create this file yourself. Use the following code only if you face issues and are absolutely unable to proceed.]

`file: dotnet-album-viewer/docker-compose.yaml`

```
version: "3.7"

services:
  app:
    image: schoolofdevops/dotnet-album-viewer
    ports:
      - 80:80
    depends_on:
      - db
    environment:
      - "Data:Provider=MySQL"
      -
"Data:ConnectionString=Server=db;Port=4000;Database=AlbumViewer;User=root;SslMo
de=None"

  db:
    image: dockersamples/tidb:nanoserver-sac2016
```

Where,

- **Version**: Defines the Docker Compose spec version. Reference official documentation on Compose file version 3 reference | Docker Documentation to know more about the syntax.
- **Services** : this section allows you to define which applications/containers to launch. Define the list of services/micros services here.
- Each service contains a code block which is nothing but `docker run` command converted into a declarative code using YAML. Its as if saying what properties you want to see that container created with.
- In addition to *services* you could add sections such as *networks*, *volumes* as needed.

> **Note**: you may notice port mapping not being defined for 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 the 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 attach surface of the host, by not exposing this db service.

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

```
cd dotnet-album-viewer

[validate the syntax]
docker-compose config

[list 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

[list again, should now show your services]
docker-compose ps

[check logs for a service by name 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 something else that `docker-compose.yaml` you need to provide a relative/absolute path to docker compose file using `-f` option, for every command you run.

Validate that your service is launched and you are able to access it on port 80 of the host using your browser.

What Happened ?

- When you run `docker-compose config`, it reads the docker-compose.yaml in the current directory and checks for any syntactical errors. If found, shows the error with the line number to help you debug.
- `docker-compose up -d` launched containers for all applications defined in the services section. It does that in the detached mode provided 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. Each of this container is prefixed with the directory name e.g. `dotnet-album-viewer-app_1` which is how docker-compose uniquely identifies the containers launched from a specific path. This also means, you could 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.

> **Error Alert**: If you see related to the MySQL database such as `unknown database AlbumViewer` its ok, you could ignore this. First time the application connects to the db, this database may not be present, and is then created automatically. So long as your application is loading, all is fine.

Exercise :

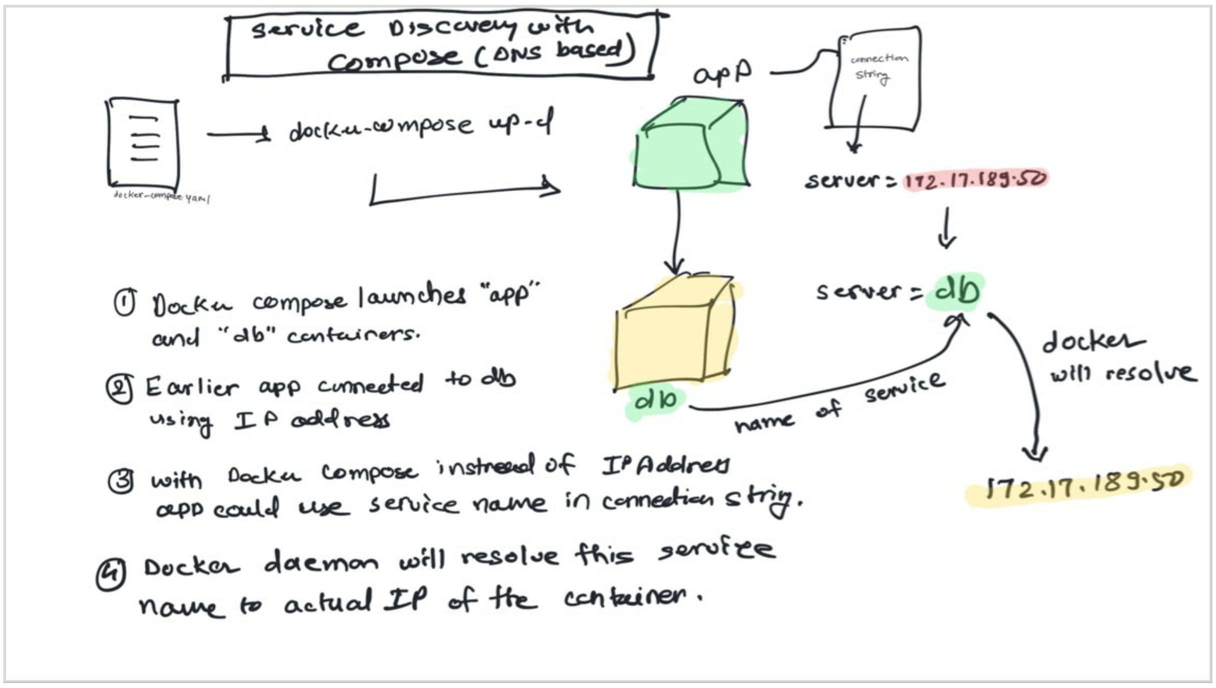
- Find out a list of sub commands that Docker Compose supports using the following

```
docker-compose
```

- Play around with these sub commands and find out what each of these do.

Service Discovery

Have you noticed environment variables in app section ? . You may see that its no more relying on IP address, but is using **db** as the hostname for database. Where does that come from ? You may have already figured its the name of the service defined to launch the TiDB container. What does this mean ? Well, once you start using docker-compose you get service discovery built in. Thats because docker daemon comes with a DNS service which is available while using compose.

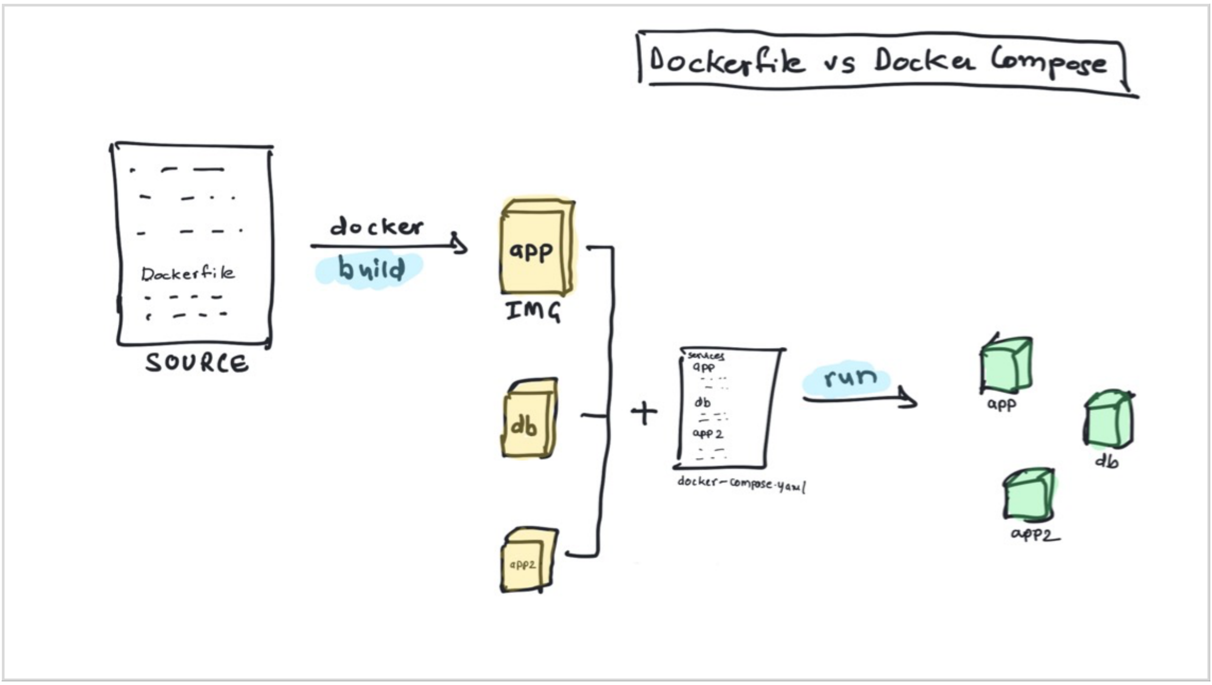


service Discovery with Compose (DNS based)
docker-compose.yaml
docker-compose up -d
app
connection string
server = 172.17.189.50
server = db
db
name of service
docker will resolve
172.17.189.50
① Docker compose launches "app" and "db" containers.
② Earlier app connected to db using IP address
③ with 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

Dockerfile as well as docker-compose.yaml, both offer a declarative interface, and allow us to write code, and help us automate with containerisation. A common question asked is what's the difference between the two, and why two different files that I need to bother about.

The answer to this lies in **when** part. The difference is Dockerfile is used during the image build process. Once you have the images built for all your applications, how to launch it is where Docker Compose come in.



You could indeed integrate Dockerfile with docker-compose.yaml and have compose build the images for you as well, which is a common practice. Lets look at how.

Integrating Dockerfile with Docker Compose

Lets add the image build part to **app** service definition as follows,

```
file: dotnet-album-viewer/docker-compose.yaml
```

```
version: "3.7"

services:
  app:
    image: schoolofdevops/dotnet-album-viewer
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 80:80
    depends_on:
      - db
    environment:
      - "Data:Provider=MySQL"
      -
"Data:ConnectionString=Server=db;Port=4000;Database=AlbumViewer;User=root;SslMo
de=None"

  db:
    image: dockersamples/tidb:nanoserver-sac2016
```

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

```
docker-compose build
```

This would,

- run docker build using options provided in the compose spec
- tag the image with 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 could run `docker-compose up -d` as many times. If there are no changes in the images/code, it will not redeploy. However, as soon as it detects the image is been updated, it would 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 if there were any changes made directly to the previous container, its 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 a automation tool relying on a declarative code, makes docker compose suitable as a deployment tool for development environments. Why do I say dev environments is because, docker compose is still limited to be run on one host. It can not launch containers crossing the boundary of the host its running on, and spanning across multiple hosts. If you need that, and you would in higher environments such as staging and production, you would have to consider a Container Orchestration Engine (COE). And yes, that's where you would start using Kubernetes. In essence, kubernetes is the platform which would take your containers beyond one host and help you simplify deploying and managing these containers on multiple hosts (thousands in some cases) .

For this chapter though, we will stick to how to use Docker Compose as a deployment tool to dev. You could also use it along with your Continuous Integration platform such as Jenkins to setup integrated development environments which more than one developers 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 already have 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 as,

```
file: dotnet-album-viewer/src/AlbumViewerNetCore/wwwroot/index.html
```

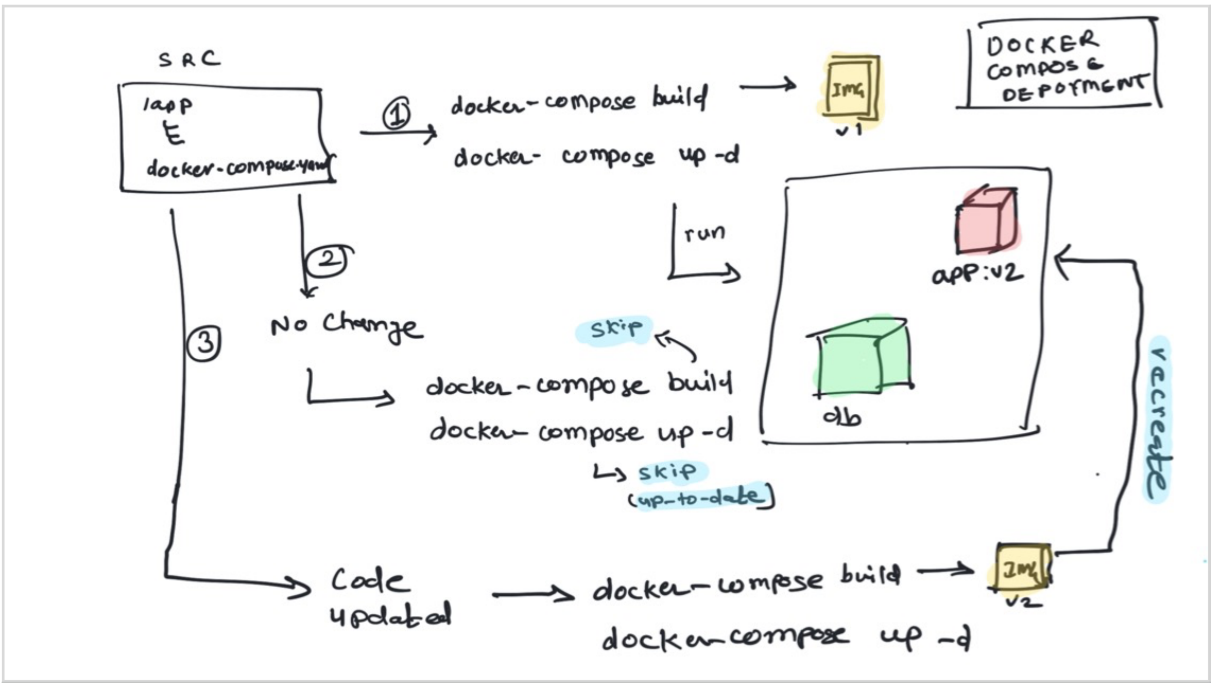
```
[change title from ]
<title>West Wind Album Viewer</title
```

```
[to]
<title>West Wind Album Viewer v2.0</title
```

And then redeploy this application with

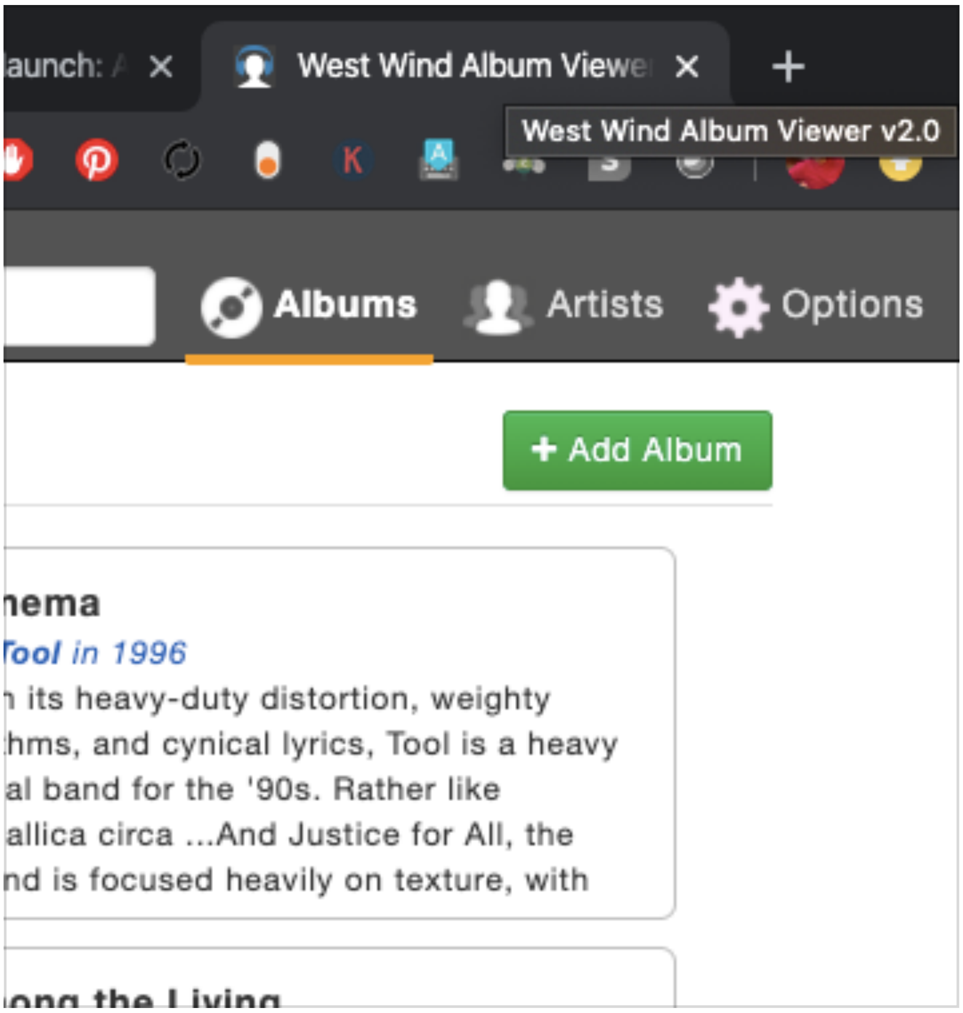
```
docker-compose build
docker-compose up -d
```

What happened ?



- Database is not changed, so it says *up-to-date* for 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 same tag as earlier. This means the previous image is now untagged, is orphan and will be deleted when `docker image prune` is run.
- also the container for app is recreated. Its a **immutable** deployment as 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 see the title updated.



nema
Tool in 1996
n its heavy-duty distortion, weighty
hms, and cynical lyrics, Tool is a heavy
al band for the '90s. Rather like
allica circa ...And Justice for All, the
nd is focused heavily on texture, with

ong the Living

Tear down the stack

Docker compose is not only useful to quickly launch a dev environment, but also tearing it down and cleaning up.

To simply 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 something else that `docker-compose.yaml` you need to provide a relative/absolute path to docker compose file using `-f` option, for every command you run.   I emphasise on 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 at 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 completely (except for the images pulled). This could be very useful as you could reuse the same host to launch another application, work with it and tear it down the same way later.

Summary

In this chapter, you not only learnt how to compose more than one services to be launched together by using a simple declarative YAML interface. You also learnt how to use this code to quickly setup and tear down a application stack , and use this to deploy to development environments.