Title: How to setup database.yml to connect to Postgres Docker container?

Post Body:

I have a Rails app. In the development and test environments, I want the Rails app to connect to a dockerized Postgres. The Rails app itself will not be in a container though - just Postgres.

What should my database.yml look like?

I have a docker `default` machine running. I created docker-compose.yml:

```
postgres:   image: postgres   ports:     - '5432:5432'   environment:     - POSTGRES_USER=timbuktu     - POSTGRES_PASSWORD=mys
```

I ran `docker-compose up` to get Postgres running.

Then I ran `docker-machine ip default` to get the IP address of the Docker virtual machine, and I updated database.yml accordingly:

```
... development:    adapter: postgresql   host: 192.168.99.100   port: 5432   database: timbuktu_development   username: timbu
```

So all is well and I can connect to Postgres in its container.

But, if someone else pulls the repo, they won't be able to connect to Postgres using my database.yml, because the IP address of their Docker default machine will be different from mine.

So how can I change my database.yml to account for this?

One idea I have is to ask them to get the IP address of their Docker default machine by running `docker-machine env default`, and pasting the env DOCKER_HOST line into their bash_rc. For example,

```
export DOCKER_HOST='tcp://192.168.99.100:2376'
```

Then my database.yml host can include the line

```
host: <%= ENV['DOCKER_HOST'].match(/tcp:\/\/(.+):\d{3,}/)[1] %>
```

But this feels ugly and hacky. Is there a better way?

Accepted Answer:

2015: You could set a correct environment variable first, and access it from your `database.yml`:

```
host: <%= ENV['POSTGRES_IP'] %>
```

With a bashrc like (using bash substring removal):

```
export DOCKER_HOST=$(docker-machine env default) export POSTGRES_IP=${DOCKER_HOST#tcp://}
```

2023: Reminder, the `docker machine` command, which was part of the Docker Machine tool, has been replaced by Docker Desktop's built-in features and Docker Compose.
Docker Machine was primarily used to provision and manage Docker hosts (or machines) on virtual environments.

The `docker-machine env default` command was used to set up the shell environment for interacting with a specific Docker machine, named `default` in this case. That command was particularly useful when managing multiple Docker hosts or when working with Docker running on a virtual machine.

In the modern Docker ecosystem, especially with Docker Desktop, the need for such a command is greatly reduced. Docker Desktop runs Docker directly on your host (either Windows or macOS), negating the need for a separate virtual machine to host Docker. So the Docker client is configured by default to communicate with Docker Desktop's daemon without additional environment setup.

However, if you are looking to replicate a similar setup where Docker runs in a virtual machine or a remote host, you would configure your Docker client to communicate with that Docker daemon. That is commonly done through environment variables, particularly `DOCKER_HOST`, `DOCKER_CERT_PATH`, and `DOCKER_TLS_VERIFY`.

You would need to:

set environment variables manually:
Determine the IP address and the port of your remote Docker daemon.
Set the `DOCKER_HOST` environment variable, and if necessary, `DOCKER_CERT_PATH` and `DOCKER_TLS_VERIFY`.

```
export DOCKER_HOST="tcp://[IP_ADDRESS]:[PORT]" export DOCKER_CERT_PATH="[path/to/cert]" export DOCKER_TLS_VERIFY=1
```

use SSH to connect to remote docker:
Docker also supports using SSH to connect to a remote Docker daemon.

```
export DOCKER_HOST="ssh://user@remote-host"
```

use Docker contexts:
Docker contexts can be used to manage different Docker endpoints.
Create a new context for your remote Docker daemon:

```
docker context create my-remote-docker --docker "host=tcp://[IP_ADDRESS]:[PORT]"
```

And use the context:

```
docker context use my-remote-docker
```

Again, in a modern setup, especially for local development, Docker Desktop eliminates the need for most of this configuration.
The steps above are more relevant if you are working with Docker in remote or specialized environments. However, with the evolution of Docker's ecosystem, especially the enhancements in Docker Desktop and Docker Compose, the functionalities of Docker Machine became redundant and less relevant.

Docker Desktop now includes features that cover most use cases of Docker Machine. It allows you to manage Docker containers, images, networks, and volumes, as well as build and run multi-container applications defined with Docker Compose. Docker Desktop's integration with cloud providers and its ability to handle local Kubernetes clusters further extend its capabilities beyond what Docker Machine offered.

Highest Rated Answer:

I found a simpler way:

```
host: <%= `docker-machine ip default` %>
```