Chapter 33: Deploying with Docker

By Jonas Schwartz

Docker是一种流行的容器化技术，它对应用程序的部署方式产生了巨大的影响。容器是一种隔离应用程序的方法，允许您在同一服务器上运行多个应用程序。

使用容器而不是完整的虚拟机，允许容器化应用程序共享更多主机资源。反过来，这为您的应用程序留下了更多资源，而不是消耗它们来支持虚拟机本身。

Docker几乎可以在任何地方运行，因此它提供了一种标准化应用程序运行方式的好方法，从本地测试到生产。

> 如果您需要进一步学习Docker术语 - 容器和镜像等概念 - 请访问 https://www.raywenderlich.com/9159- docker-on-macos-getting-started. 查看我们的Docker教程。

Docker Compose

本章还将向您展示如何使用Docker Compose。 Docker Compose是一种指定作为单个单元一起工作的不同容器列表的方法。这些容器共享相同的虚拟网络，使它们彼此合作变得简单。

例如，使用Docker Compose，只需一个命令即可启动Vapor应用程序和PostgreSQL数据库实例。它们可以相互通信，但与在同一主机上运行的其他实例隔离。

设置Vapor和PostgreSQL进行开发

首先，设置一个简单的开发配置，以在Linux环境中测试您的应用程序。为了便于调试出现的任何问题，这将比您在生产中使用的配置简单得多。

> 本章的示例项目与第21章“Validation”末尾的项目相同。您可以使用它，也可以继续使用现有项目。要学习本章，您需要重命名项目主目录中当前存在的任何**Dockerfile**或**docker-compose.yml**。您的主目录是项目的**Package.resolved**文件所在的位置。

在项目的主目录中，创建名为**Dockerfile**的文件并添加以下内容：

```
#1
FROM swift:4.2
#2
WORKDIR /app
#3
COPY . .
#4
RUN swift package clean
RUN swift build -c release
RUN mkdir /app/bin
RUN mv `swift build -c release --show-bin-path` /app/bin
EXPOSE 8080
#5
ENTRYPOINT ./bin/release/Run serve --env local \
  --hostname 0.0.0.0
```

Dockerfile provides the “recipe” for creating a Docker container for your app. Here’s what this one does:

1. Use version 4.2 of the “swift” image from the Docker Hub repository as the starting point.
2. Tell Docker to use **/app** as its working directory.
3. Copy your project to the Docker container.
4. Build your project and move the executable to **/app/bin** within the container.
5. Tell Docker how to start the Vapor app.

Next, also in your project's main directory, create a file named **docker-compose.yml** and add the following contents:

```
# 1
version: '3'
# 2
services:
  # 3
  til-app:
    # 4
    depends_on:
      - postgres
    # 5
    build: .
    # 6
    ports:
      - "8080:8080"
    environment:
      - DATABASE_HOSTNAME=postgres
      - DATABASE_PORT=5432
  # 7
  postgres:
    # 8
    image: "postgres"
    # 9
    environment:
      - POSTGRES_DB=vapor
      - POSTGRES_USER=vapor
      - POSTGRES_PASSWORD=password

  # 10
  start_dependencies:
    image: dadarek/wait-for-dependencies
    depends_on:
      - postgres
    command: postgres:5432
```

docker-compose.yml specifies the "recipe" for your entire app with all of its dependencies. Here's what this one does:

1. Specify the Docker Compose version.
2. Define the services for this application.
3. Define a service for the TIL application.
4. Set a dependency on the `postgres` service so Docker Compose starts the PostgreSQL container first.
5. Build the **Dockerfile** in the current directory. This is the **Dockerfile** you created earlier.

6. Make port 8080 accessible on the host system and inject the `DATABASE_HOSTNAME` environment variable. Docker Compose has an internal DNS resolver. This allows the `til-app` container to connect to the `postgres` container with the hostname `postgres`. Also set the port for the database. You can specify any other environment variable values your app needs here.
7. Define a service for the PostgreSQL database.
8. Use the standard `postgres` image.
9. Set the necessary environment variables.
10. Docker starts all containers at once and PostgreSQL takes several seconds to become ready to accept connections. If TILapp starts before PostgreSQL is ready, TILapp will crash. This service provides a way to ensure the database is running before starting your app.

To bring your app to life, enter the following commands in Terminal:

```
# 1
docker-compose build
# 2
docker-compose run --rm start_dependencies
# 3
docker-compose up til-app
```

Here's what this does:

1. Build the different Docker images.
2. Run the `start_dependencies` service to ensure that PostgreSQL is running and ready.
3. Start your app.

> If you receive an error stating the "**vapor**" database is not found, follow the clean up steps below and retry the commands above and the application should start successfully. This error might occur if you have previous Docker postgres images on your device.

In your browser, visit **http://localhost:8080** to verify the app is up and running. When you're ready to move ahead, press **Control-C** to stop everything. Then, clean up your development environment by entering the following in Terminal:

```
docker-compose down
docker volume prune
```

This shuts down any running containers and removes all containers and network definitions associated with your app. It then cleans up any old Docker storage you can no longer access.

Setting up Vapor and PostgreSQL for Production

There are several changes you can make to your Docker configuration to simplify managing your app in a production environment. In this section, you'll split your app into a "builder" container and a production image. You'll also configure the PostgreSQL container to save its database in your host's file system, making your data persist across changes to your app and its configuration.

In the main directory for your project, create a file named **production.Dockerfile** and add the following contents:

```
# 1
FROM swift:4.2 as builder

# 2
RUN apt-get -qq update && apt-get -q -y install \
  tzdata \
  && rm -r /var/lib/apt/lists/*

# 3
WORKDIR /app
# 4
COPY . .
# 5
RUN mkdir -p /build/lib && \
  cp -R /usr/lib/swift/linux/*.so /build/lib
RUN swift build -c release && \
  mv `swift build -c release --show-bin-path` /build/bin

# Production image
# 6
FROM ubuntu:16.04
# 7
RUN apt-get -qq update && apt-get install -y \
  libicu55 libxml2 libbsd0 libcurl3 libatomic1 \
  tzdata \
  && rm -r /var/lib/apt/lists/*
# 8
WORKDIR /app
# 9
COPY --from=builder /build/bin/Run .
COPY --from=builder /build/lib/* /usr/lib/
# You need the next line if your app serves static resources
# from the Public directory
COPY --from=builder /app/Public ./Public
# You need the next line if your app uses Leaf
COPY --from=builder /app/Resources ./Resources

# 10
```

```
ENTRYPOINT ./Run serve --env production --hostname 0.0.0.0 \
  --port 8080
```

1. Use version 4.2 of the "swift" image from the Docker Hub repository as the starting point. This container is only for building your app and may be deleted once the app is built.
2. Install the tzdata package, then clean up the working files. This cleanup is a standard operation when building Docker images based on Linux. It reduces the overall size of the image.
3. Tell Docker to use **/app** as its working directory.
4. Copy your project to the Docker container.
5. Make a build folder and copy the needed Swift run-time support to it. Build your project and move the executable into the build folder.
6. Base your production image on Ubuntu 16.04.
7. Install some needed dependencies, then clean up the working files.
8. Set up same work directory
9. Copy files from the builder container. If your project doesn't serve static content or use Leaf, you may drop the appropriate `COPY` steps.
10. Tell Docker how to start the Vapor app.

Next, also in your project's main directory, create a file named **docker-compose.production.yml** and add the following contents:

```
# 1
version: '3'
# 2
services:
  # 3
  til-app:
    # 4
    depends_on:
      - postgres
    # 5
    build:
      context: .
      dockerfile: production.Dockerfile
    # 6
    ports:
      - "8080:8080"

    environment:
      - DATABASE_HOSTNAME=postgres
```

```
      - DATABASE_PORT=5432
  # 7
  postgres:
    # 8
    image: "postgres"
    # 9
    volumes:
      - ~/postgres-data:/var/lib/postgresql/data
    # 10
    environment:
      - POSTGRES_DB=vapor
      - POSTGRES_USER=vapor
      - POSTGRES_PASSWORD=password

  # 11
  start_dependencies:
    image: dadarek/wait-for-dependencies
    depends_on:
      - postgres
    command: postgres:5432
```

Here's what this does:

1. Specify the Docker Compose version.
2. Define the services for this application.
3. Define a service for the TIL application.
4. Set a dependency on the `postgres` service so Docker Compose starts the PostgreSQL container first.
5. Build the project using **production.Dockerfile** as its recipe.
6. Inject the `DATABASE_HOSTNAME` environment variable. Docker Compose has an internal DNS resolver. This allows the `til-app` container to connect to the `postgres` container with the hostname `postgres`. Also set the port for the database. You can specify any other environment variable values your app needs here.
7. Define a service for the PostgreSQL database.
8. Use the standard `postgres` image.
9. Set up a persistent volume from **~/postgres-data** into the container. This causes the data to live in the host system's file system rather than inside a Docker container and allows it to persist across launches.
10. Set the necessary environment variables.
11. Use the same technique as before to ensure that PostgreSQL is ready to accept connections before starting your app.

To bring your app to life, enter the following commands in Terminal:

```
docker-compose -f docker-compose.production.yml build
docker-compose -f docker-compose.production.yml \
  run --rm start_dependencies
docker-compose -f docker-compose.production.yml up til-app
```

These commands mirror the ones used in the Development section but they use your new production configuration files.

Where to go from here?

You've seen some basic recipes for how to run your app in a Docker environment. Because Docker is so flexible, these recipes only scratch the surface of the possibilities available to you. For example, you might want to allow your app to save uploaded files in the host's file system. Or, you might want to configure the app to run behind an nginx proxy server to get secure HTTPS access.