ADMIN Network & Security

Domain-Driven Design

ShardingSphere

SPECIAL docker

ADMIN

Network & Security

ISSUE 78

Domain-Driven Design

Principles for programming a domain model

Event-Driven Ansible

Microsoft Power Apps

Low-code/no-code programming

3 Black Box Monitoring Solutions

Monitoror, Vigil, Statping-ng

Velociraptor Incident Response

Keeping Azure VMs Up to Date

LINUX NEW MEDIA
The Pulse of Open Source

Knative

Serverless workloads for Kubernetes

Visual Studio Code for the Web

Secure remote connectivity

MACsec

Layer 2 link encryption

Ripgrep

Accelerated terminal search

Dockerizing Legacy Applications

Makeover

Sooner or later, you'll want to convert your legacy application to a containerized environment. Docker offers the tools for a smooth and efficient transition. By Artur Skura

In the past, we ran applications on physical machines. We cared about every system on our network, and we even spent time discussing a proper naming scheme for our servers (RFC 1178 [1]). Then virtual machines came along, and the number of servers we needed to manage increased dramatically. We would spin them up and shut them down as necessary. Then containers took this idea even further: It typically took several seconds or longer to start a virtual machine, but you could start and stop a container in almost no time.

In essence, a container is a well-isolated process, sharing the same kernel as all other processes on the same machine. Although several container technologies exist, the most popular is Docker. Docker's genius was to create a product that is so smooth and easy to use that suddenly everybody started using it. Docker managed to hide the underlying complexity of spinning up a container and to make common operations as simple as possible.

Containerizing Legacy Apps

Although most modern apps are created with containerization in mind, many legacy applications based on older architectures are still in use. If your legacy application is running fine in a legacy context, you might be wondering why you would want to go to the trouble to containerize.

The first advantage of containers is the uniformity of environments: Containerization ensures that the application runs consistently across multiple environments by packaging the app and its dependencies together. This means that the development environment on the developer's laptop is fundamentally the same as the testing and production environments. This uniformity can lead to significant savings with testing and troubleshooting future releases. Another benefit is that containers can be horizontally scaled; in other words, you can scale the application by increasing (and decreasing) the number of containers. Adding a container orchestration tool like Kubernetes means you can optimize resource allocation and better use the machines you have – whether physical or virtual. The power of container orchestration makes it easy to scale the app with the load. Because containers start faster than virtual machines, you can scale much more efficiently, which is crucial for applications that have to deal with sudden load spikes. The fact that you can start and terminate containers quickly has several other consequences. You can deploy your applications much faster – and roll them back equally quickly if you experience problems.

Getting Started

To work with Docker, you need to set up a development environment. First, you'll need to install Docker itself. Installation steps vary, depending on your operating system [2]. Once Docker is installed, open a terminal and execute the following command to confirm Docker is correctly installed:

```
docker --version
```

Now that you have Docker installed, you'll also need Docker Compose, a tool for defining and running multi-container Docker applications [3]. If you have Docker Desktop installed, you won't need to install Docker Compose separately because the Compose plugin is already included. For a simple example to illustrate the fundamentals of Docker, consider a Python application running Flask, a web framework that operates on a specific version of Python and relies on a few third-party packages.

Listing 1 shows a snippet of a typical Python application using Flask.
To dockerize this application, you would write a Dockerfile – a script containing a sequence of instructions to build a Docker image. Each instruction in the Dockerfile generates a new layer in the resulting image, allowing for efficient caching and reusability. By constructing a Dockerfile, you essentially describe the environment your application needs to run optimally, irrespective of the host system.
Start by creating a file named `Dockerfile` (no file extension) in your project directory. The basic structure involves specifying a base image, setting environment variables, copying files, and defining the default command for the application. Listing 2 shows a simple Dockerfile for the application in Listing 1.
In this Dockerfile, I specify that I'm using Python 3.11, set the working directory in the container to `/app`, copy the required files, and install the necessary packages, as defined in a `requirements.txt` file. Finally, I specify that the application should start by running `app.py`.
To build this Docker image, you would navigate to the directory containing the Dockerfile and execute the following commands to build and run the app:

```
docker build -t my-legacy-app .
docker run -p 5000:5000 ➥
  my-legacy-app
```

With these steps, you have containerized the Flask application using Docker. The application now runs isolated from the host system, making it more portable and easier to deploy on any environment that supports Docker.

Networking in Docker

Networking is one of Docker's core features, enabling isolated containers to communicate amongst themselves and with external networks. The most straightforward networking scenario involves a single container that needs to be accessible from the host machine or the outside world. To support network connections, you'll need to expose ports. When running a container, the `-p` flag maps a host port to a container port:

```
docker run -d -p 8080:80 ➥
  --name web-server nginx
```

In this case, NGINX is running inside the container on port 80. The `-p 8080:80` maps port 8080 on the host to port 80 on the container. Now, accessing `http://localhost:8080` on the host machine directs traffic to the NGINX server running in the container.
For inter-container communication, Docker offers several options. The simplest approach involves using container names as DNS names, made possible by the default `bridge` network. First, run a database container:

```
docker run -d --name ➥
  my-database mongo
```

Now, if you want to link a web application to this database, you can reference the database container by its name:

```
docker run -d --link my-database:db ➥
  my-web-app
```

In this setup, `my-web-app` can connect to the MongoDB server by using `db` as the hostname.
Although useful, the `--link` flag is considered legacy and is deprecated. A more flexible approach is to create custom bridge networks. A custom network facilitates automatic DNS resolution for container names, and it also allows for network isolation.
For example, you can create a custom network as follows:

```
docker network create my-network
```

Now, run containers in this custom network with:

```
docker run -d --network=my-network ➥
  --name my-database mongo ➥
  --network-alias=db
docker run -d --network=my-network ➥
  my-web-app
```

Here, `my-web-app` can still reach `my-database` using its name or a DNS alias, but now both containers are isolated in a custom network, offering more control and security.
For applications requiring more complex networking setups, you can use Docker Compose and define multiple services, networks, and even volumes in a single `docker-compose.yml` file (Listing 3).
When you run `docker-compose up`, both services will be instantiated, linked, and isolated in a custom network, as defined.
As you can see, effective networking in Docker involves understanding and combining these elements: port mapping for external access, inter-container communication via custom bridge networks, and orchestration (managed here by Docker Compose).

Volumes and Persistent Data

Managing persistent data within Docker involves understanding and leveraging volumes. Unlike a container, a volume exists independently

Listing 1: Simple Flask App

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
  return 'Hello, World!'

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000)
```

Listing 2: Dockerfile for Flask App (Listing 1)

```
# Use an official Python runtime as a base image
FROM python:3.11-slim

# Set the working directory in the container
WORKDIR /app

# Copy the requirements.txt file into the container
COPY requirements.txt /app/

# Install the dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the current directory contents into the container
COPY . /app/

# Run app.py when the container launches
CMD ["python", "app.py"]
```

and retains data even when a container is terminated. This characteristic is crucial for stateful applications, like databases, that require data to persist across container life cycles.
For simple use cases, you can create anonymous volumes at container runtime. When you run a container with an anonymous volume, Docker generates a random name for the volume. The following command starts a MongoDB container and attaches an anonymous volume to the `/data/db` directory, where MongoDB stores its data:

```
docker run -d --name my-mongodb ➥
  -v /data/db mongo
```

Listing 3: Sample docker-compose.yml File

```
services:
  web:
    image: nginx
    networks:
      - my-network
  database:
    image: mongo
    networks:
      - my-network
networks:
  my-network:
    driver: bridge
```

Listing 4: Sample Named Volume

```
services:
  database:
    image: mongo
    volumes:
      - my-mongo-data:/data/db
volumes:
  my-mongo-data:
```

Listing 5: A sample Dockerfile for Apache web server

```
# Use an official Ubuntu as a parent image
FROM ubuntu:latest

# Install Apache HTTP Server
RUN apt-get update && apt-get install -y apache2

# Copy local configuration files into the container
COPY ./my-httpd.conf /etc/apache2/apache2.conf

# Expose port 80 for the web server
EXPOSE 80

# Start Apache when the container runs
CMD ["apachectl", "-D", "FOREGROUND"]
```

Whereas anonymous volumes are suitable for quick tasks, named volumes provide more control and are easier to manage. If you use `docker run` and specify a named volume, Docker will auto-create it if needed. You can also create a named volume explicitly with:

```
docker volume create my-mongo-data
```

Now you can start the MongoDB container and explicitly attach this named volume:

```
docker run -d --name my-mongodb ➥
  -v my-mongo-data:/data/db mongo
```

You can use named volumes to share data between containers. If you need to share data between the container and the host system, host volumes are the choice. This feature mounts a specific directory from the host into the container:

```
docker run -d --name my-mongodb ➥
  -v /path/on/host:/data/db mongo
```

Here, `/path/on/host` corresponds to the host system directory you want to mount.
With Docker Compose, volume specification becomes streamlined and readable, especially when dealing with multi-container, stateful legacy applications. Listing 4 shows how you could define a service in `docker-compose.yml` with a named volume. When you run `docker-compose up`, it will instantiate the service with the specified volume.
Data persistence isn't confined to just storing data; backups are equally vital. Use `docker cp` to copy files or directories between a container and the local filesystem. To back up data from a MongoDB container, enter:

```
docker cp my-mongodb:/data/db ➥
  /path/on/host
```

Here, data from `/data/db` inside the `my-mongodb` container is copied to `/path/on/host` on the host system.

Dockerizing a Legacy Web Server

Containerizing a legacy web server involves several phases: assessment, dependency analysis, containerization, and testing. For this example, I'll focus on how to containerize an Apache HTTP Server. The process generally involves creating a Dockerfile, bundling configuration files, and possibly incorporating existing databases.
The first step is to create a new directory to hold your Dockerfile and configuration files. This directory acts as the build context for the Docker image:

```
mkdir dockerized-apache
cd dockerized-apache
```

Start by creating a Dockerfile that specifies the base image and installation steps. Imagine you're using an Ubuntu-based image for compatibility with your legacy application (Listing 5).
In Listing 5, the `RUN` instruction installs Apache, and the `COPY` instruction transfers your existing Apache configuration file (`my-httpd.conf`) into the image. The `CMD` instruction specifies that Apache should run in the foreground when the container starts. Place your existing Apache configuration file in the same directory as the Dockerfile. This configuration should be a working setup for your legacy web server. Build the Docker image from within the `dockerized-apache` directory:

```
docker build -t dockerized-apache .
```

Run a container from this image, mapping port 80 inside the container to port 8080 on the host:

```
docker run -d -p 8080:80 --name ➥
  my-apache-container ➥
  dockerized-apache
```

The legacy Apache server should now be accessible via `http://localhost:8080`.

If your legacy web server interacts with a database, you'll likely need to

dockerize that component as well or ensure the web server can reach the existing database. For instance, if you have a MySQL database, you can run a MySQL container and link it to your Apache container. A tool like Docker Compose can simplify the orchestration of multi-container setups.
For debugging, you can view the logs using the following command:

```
docker logs my-apache-container
```

This example containerized a legacy Apache HTTP Server, but you can use this general framework with other web servers and applications as well. The key is to identify all dependencies, configurations, and runtime parameters to ensure a seamless transition from a traditional setup to a containerized environment.

What About a Database?

Containers are by nature stateless, whereas data is inherently stateful. Therefore databases require a more nuanced approach. In the past, running databases in containers was usually not recommended, but nowadays you can do it perfectly well – you just need to make sure the data is treated properly.
Or, you can decide not to containerize your databases at all. In this scenario, your containers connect to a dedicated database, such as an RDS instance managed by Amazon Web Services (AWS), which makes sense if your containers are running on AWS. Amazon then takes care of provisioning, replication, backup, and so on. This safe and clean solution lets you concentrate on other tasks while AWS is doing the chores. One common scenario is to use a containerized database in local development (so it's easy to spin up/tear down), but then swap out for a managed database service in production. At the end of the day, your app is using the database's communication protocol, regardless of where and how the database is running.
Dockerizing an existing database like MySQL or Oracle Database is a nontrivial task that demands meticulous planning and execution. The procedure involves containerizing the database, managing persistent storage, transferring existing data, and ensuring security measures are in place. One area where containerizing a traditional SQL database is extremely useful is in development and testing (see the "Testing" box).
If you choose to dockerize a database, the first step is to choose a base image. For MySQL, you could use the official Docker image available on Docker Hub. You will also find official images for Oracle Database.
The following is a basic example of how to launch a MySQL container:

```
docker run --name my-existing-mysql ➥
  -e MYSQL_ROOT_PASSWORD=➥
  my-secret-pw -d mysql:8.0
```

In this example, the environment variable `MYSQL_ROOT_PASSWORD` is set to your desired root password. The `-d` flag runs the container in detached mode, meaning it runs in the background.
This quick setup works for new databases, but keep in mind that existing databases require you to import existing data. You can use a Docker volume to import a MySQL dump file into the container and then import it into the MySQL instance within the Docker container.

Configurations and Environment Variables

Legacy applications often rely on complex configurations and environment variables. When dockerizing such applications, it's crucial to manage these configurations efficiently, without compromising security or functionality. Docker provides multiple ways to inject configurations and environment variables into containers: via Dockerfile instructions, command-line options, environment files, and Docker Compose. Each method serves a particular use case.
Dockerfile-based configurations are suitable for immutable settings that don't change across different environments. For instance, setting the `JAVA_HOME` variable for a Java application can be done in the Dockerfile:

```
FROM openjdk:11
ENV JAVA_HOME ➥
  /usr/lib/jvm/java-11-openjdk-amd64
```

However, hard-coding sensitive or environment-specific information in the Dockerfile is not recommended, because it compromises security and reduces flexibility.

Testing

You can quickly spin up a container with your database, usually containing test data, and then immediately verify if your app works properly with the database. And you can do it all without asking the Ops team to provision a database host for you.
Good examples of this usage are services in GitLab CI/CD pipelines, such as PostgreSQL [4] or MySQL [5]. In Listing 6, I use a Docker image containing Docker and Docker Compose and the usual variables defining the database, its user, and password. I also define the so-called service, based on the image `postgres:16` and aliased as `postgres`. In the `test` job, I install the PostgreSQL command-line client and execute a sample SQL query connecting to the `postgres` service defined earlier, having exported the password. The `postgres` service is simply a Docker container with the chosen version of PostgreSQL conveniently started in the same network as the main container so that you can connect to it directly from your pipeline.

Listing 6: PostgreSQL in a GitLab CI Pipeline

```
image: my-image-with-docker-and-docker-compose

variables:
  POSTGRES_DB: my-db
  POSTGRES_USER: ${USER_NAME}
  POSTGRES_PASSWORD: ${USER_PASSWORD}

services:
  - name: postgres:16
    alias: postgres

test:
  script:
    - apt-get update && apt-get install -y
      postgresql-client
    - PGPASSWORD=$POSTGRES_PASSWORD psql -h postgres
      -U $POSTGRES_USER -d $POSTGRES_DB -c 'SELECT 1;'
```

For more dynamic configurations, use the `-e` option with `docker run` to set environment variables:

```
docker run -e "DB_HOST=➥
  database.local" -e "DB_PORT=➥
  3306" my-application
```

While convenient for a few variables, this approach becomes unwieldy with a growing list. As a more scalable alternative, Docker allows you to specify an environment file:

```
# .env file
DB_HOST=database.local
DB_PORT=3306
```

Then, run the container as follows:

```
docker run --env-file .env ➥
  my-application
```

This method keeps configurations organized, is easy to manage with version control systems, and separates the configurations from application code. However, exercise caution; ensure the `.env` files, especially those containing sensitive information, are adequately secured and not accidentally committed to public repositories.
In multi-container setups orchestrated with Docker Compose, you can define environment variables in the `docker-compose.yml` file:

```
services:
  my-application:
    image: my-application:latest
    environment:
      DB_HOST: database.local
      DB_PORT: 3306
```

For variable data across different environments (development, staging, production), Docker Compose supports variable substitution:

```
services:
  my-application:
    image: my-application:➥
      ${TAG-latest}
    environment:
      DB_HOST: ${DB_HOST}
      DB_PORT: ${DB_PORT}
```

Run it with environment variables sourced from a `.env` file or directly from the shell:

```
DB_HOST=database.local ➥
  DB_PORT=3306 docker-compose up
```

Configuration files necessary for your application can be managed using Docker volumes. Place the configuration files on the host system and mount them into the container:

```
docker run -v ➥
  /path/to/config/on/host:➥
  /path/to/config/in/container ➥
  my-application
```

In Docker Compose, use:

```
services:
  my-application:
    image: my-application:latest
    volumes:
      - /path/to/config/on/host:➥
        /path/to/config/in/container
```

This approach provides a live link between host and container, enabling real-time configuration adjustments without requiring container restarts.

Docker and Security Concerns

Securing Docker containers requires checking every layer: the host system, the Docker daemon, images, containers, and networking. Mistakes in any of these layers can expose your application to a variety of threats, including unauthorized data access, denial of service, code execution attacks, and many others.
Start by securing the host system running the Docker daemon. Limit access to the Docker Unix socket, typically `/var/run/docker.sock`. This socket allows communication with the Docker daemon and, if compromised, grants full control over Docker. Use Unix permissions to restrict access to authorized users.
Always fetch Docker images from trusted sources. Scan images for vulnerabilities using a tool like Docker Scout [6] or Clair [7].

Implement least privilege principles for containers. For instance, don't run containers as the root user. Specify a non-root user in the Dockerfile:

```
FROM ubuntu:latest
RUN useradd -ms /bin/bash myuser
USER myuser
```

Containers also should not run with least privileges. The following example:

```
docker run --cap-drop=all -cap-add=➥
  net_bind_service my-application
```

starts a container with all capabilities dropped and then adds back only the `net_bind_service` capability required to bind to ports lower than 1024.
Use read-only mounts for sensitive files or directories to prevent tampering:

```
docker run -v /my-secure-data:➥
  /data:ro my-application
```

If the container needs to write to a filesystem, consider using Docker volumes and restricting read/write permissions appropriately.
It is also important to implement logging and monitoring to detect abnormal container behavior, such as unexpected outgoing traffic or resource utilization spikes.

Dockerizing a Legacy CRM System

To dockerize a legacy Customer Relationship Management (CRM) system effectively, you need to first understand its current architecture. The hypothetical legacy CRM I'll dockerize consists of an Apache web server, a PHP back end, and a MySQL database. The application currently runs on a single, aging physical server, handling functions from customer data storage to sales analytics.
The CRM's monolithic architecture means that the web server, PHP back end, and database are tightly integrated, all residing on the same machine. The web server listens on port 80 and communicates directly with

the PHP back end, which in turn talks to the MySQL database on port 3306. Clients interact with the CRM through a web interface served by the Apache server.

The reasons for migrating the CRM to a container environment are as follows:

- Scalability: The system's monolithic nature makes it hard to scale individual components.
- Maintainability: Patching or updating one part of the applications often requires taking the entire system offline.
- Deployment: New feature rollouts are time-consuming and prone to errors.
- Resource utilization: The aging hardware is underutilized but can't be decommissioned due to the monolithic architecture.

To containerize the CRM, you need to take the following steps.

Step 1: Initial Isolation of Components and Dependencies

Before you dive into dockerization, it is important to isolate the individual components of the legacy CRM system: the Apache web server, PHP back end, and MySQL database. This step will lay the groundwork for creating containerized versions of these components. However, the tightly integrated monolithic architecture presents challenges in isolation, specifically in ensuring that dependencies are correctly mapped and that no features break in the process.

Start by decoupling the Apache web server from the rest of the system. One approach is to create a reverse proxy that routes incoming HTTP requests to a separate machine or container where Apache is installed. You can achieve this using NGINX:

```
# nginx.conf
server {
  listen 80;
  location / {
    proxy_pass ➥
    http://web:80;
  }
}
```

Next, move the PHP back end to its own environment. Use PHP-FPM to manage PHP processes separately. Update Apache's `httpd.conf` to route PHP requests to the PHP-FPM service:

```
# httpd.conf
ProxyPassMatch ^/(.*\.php(/.*)?)$➥
  fcgi://php:9000/path/to/app/$1
```

For the MySQL database, configure a new MySQL instance on a separate machine. Update the PHP back end to connect to this new database by altering the database connection string in the configuration:

```
<?php
$db = new PDO('mysql:host=db;dbname=➥
  your_db', 'user', 'password');
?>
```

During this isolation, you might find that some components have shared libraries or dependencies that are stored locally, such as PHP extensions or Apache modules. These should be identified and installed in the respective isolated environments. Missing out on these dependencies can cause runtime errors or functional issues.

While moving the MySQL database, ensuring data consistency can be a challenge. Use tools like `mysqldump` [8] for data migration and validate the consistency (Listing 7).

If user sessions were previously managed by storing session data locally, you'll need to migrate this functionality to a distributed session management system like Redis.

Step 2: Creating Dockerfiles and Basic Containers

Once components and dependencies are isolated, the next step is crafting Dockerfiles for each element: the Apache web server, PHP back end, and MySQL database. For Apache, the Dockerfile starts from a base Apache image and copies the necessary HTML and configuration files. A simplified Dockerfile appears in Listing 8.

Build the Apache image with:

```
docker build -t my-apache-image .
```

Then, run the container:

```
docker run --name ➥
  my-apache-container ➥
  -d my-apache-image
```

For PHP, start with a base PHP image and then install needed extensions. Add your PHP code afterwards (Listing 9).

Build and run the PHP image similarly to Apache:

```
docker build -t my-php-image .
docker run --name my-php-container ➥
  -d my-php-image
```

MySQL Dockerfiles are less common because the official MySQL Docker images are configurable via environment variables. However, if you have SQL scripts to run at startup, you can include them (Listing 10).

Run the MySQL container with environment variables to set up the database name, user, and password:

```
docker run --name my-mysql-container ➥
  -e MYSQL_ROOT_PASSWORD=➥
  my-secret -d my-mysql-image
```

Listing 7: MySQL Data

```
# Data export from old MySQL
mysqldump -u username -p database_name > data-dump.sql

# Data import to new MySQL
mysql -u username -p new_database_name < data-dump.sql
```

Listing 8: Dockerfile for Apache

```
# Use an official Apache runtime as base image
FROM httpd:2.4

# Copy configuration and web files
COPY ./my-httpd.conf /usr/local/apache2/conf/httpd.conf
COPY ./html/ /usr/local/apache2/htdocs/
```

Listing 9: Dockerfile for PHP

```
# Use an official PHP runtime as base image
FROM php:8.2-fpm

# Install PHP extensions
RUN docker-php-ext-install pdo pdo_mysql

# Copy PHP files
COPY ./php/ /var/www/html/
```

For production, you'll need to optimize these Dockerfiles and runtime commands with critical settings, such as specifying non-root users to run services in containers, fine-tuning Apache and PHP settings for performance, and enabling secure connections to MySQL.

Step 3: Networking and Data Management

At this point, the decoupled components – Apache, PHP, and MySQL – each reside in a separate container.

Listing 10: Dockerfile for MySQL Startup Scripts

```
# Use the official MySQL image
FROM mysql:8.0

# Initialize database schema
COPY ./sql-scripts/ /docker-entrypoint-initdb.d/
```

Listing 11: Network Setup

```
docker run --network crm-network --name
  my-apache-container -d my-apache-image
docker run --network crm-network --name my-php-container
  -d my-php-image
docker run --network crm-network --name
  my-mysql-container -e MYSQL_ROOT_PASSWORD=my-secret -d
  my-mysql-image
```

Listing 12: docker-compose.yml Network Setup

```
services:
  web:
    image: my-apache-image
    networks:
      - crm-network

  php:
    image: my-php-image
    networks:
      - crm-network

  db:
    image: my-mysql-image
    environment:
      MYSQL_ROOT_PASSWORD: my-secret
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - crm-network

networks:
  crm-network:
    driver: bridge

volumes:
  mysql-data:
```

For these containers to function cohesively as your legacy CRM system, appropriate networking and data management strategies are vital. Containers should communicate over a user-defined bridge network rather than Docker's default bridge to enable hostname-based communication. Create a user-defined network:

```
docker network create crm-network
```

Then attach each container to this network (Listing 11).

Now, each container can reach another using an alias or the service name as the hostname. For instance, in your PHP database connection string, you can replace the hostname with `my-mysql-container`.

Data in Docker containers is ephemeral. For a database system, losing data upon container termination is unacceptable. You can use Docker volumes to make certain data persistent and manageable:

```
docker volume create mysql-data
```

Bind this volume to the MySQL container:

```
docker run --network crm-network ➥
  --name my-mysql-container ➥
  -e MYSQL_ROOT_PASSWORD=➥
  my-secret -v mysql-data:➥
  /var/lib/mysql -d my-mysql-image
```

For the Apache web server and PHP back end, you should map any writable directories (e.g., for logs or uploads) to Docker volumes.

Docker Compose facilitates running multi-container applications. Create a `docker-compose.yml` file as shown in Listing 12.

Execute `docker-compose up`, and all your services will start on the defined network with the appropriate volumes for data persistence. Note that user-defined bridge networks incur a small overhead. Although this overhead is negligible for most applications, high-throughput systems might require `host` or `macvlan` networks.

If you decide to run your app in Kubernetes, for example, you will not need to worry about Docker networking, because Kubernetes has its own networking plugins.

Step 4: Configuration Management and Environment Variables

Configuration management and environment variables form the backbone of a flexible, maintainable dockerized application. They allow you to parametrize your containers so that the same image can be used in multiple contexts, such as development, testing, and production, without alteration. These parameters might include database credentials, API keys, or feature flags.

You can pass environment variables to a container at runtime via the `-e` flag:

```
docker run --name my-php-container ➥
  -e API_KEY=my-api-key ➥
  -d my-php-image
```

In your PHP code, the `API_KEY` variable can be accessed as `$_ENV['API_KEY']` or `getenv('API_KEY')`. For a more comprehensive approach, Docker Compose allows you to specify environment variables for each service in the `docker-compose.yml` file:

```
services:
  db:
    image: my-mysql-image
    environment:
      MYSQL_ROOT_PASSWORD: my-secret
```

Alternatively, you can use a `.env` file in the same directory as your `docker-compose.yml`. Place your environment variables in the `.env` file:

```
API_KEY=my-api-key
MYSQL_ROOT_PASSWORD=my-secret
```

Reference these in `docker-compose.yml`:

```
services:
  db:
    image: my-mysql-image
    environment:
      MYSQL_ROOT_PASSWORD: ➥
        ${MYSQL_ROOT_PASSWORD}
```

Running `docker-compose up` will load these environment variables

automatically. Never commit sensitive information like passwords or API keys in your Dockerfiles or code. Configuration files for Apache, PHP, or MySQL should never be hard-coded into the image. Instead, mount them as volumes at runtime. If you're using Docker Compose, you can specify a volume using the `volumes` directive:

```
services:
  web:
    image: my-apache-image
    volumes:
- ./my-httpd.conf:/usr/local/➥
  apache2/conf/httpd.conf
```

Some configurations might differ between environments (e.g., development and production). Use templates for your configuration files where variables can be replaced at runtime by environment variables. Tools like `envsubst` can assist in this substitution before the service starts:

```
envsubst < my-httpd-template.conf > ➥
  /usr/local/apache2/conf/httpd.conf
```

Strive for immutable configurations and idempotent operations to ensure your system's consistency. Once a container is running, changing its configuration should not require manual intervention. If a change is needed, deploy a new container with the updated configuration.
While this approach is flexible, it introduces complexity into the system, requiring well-documented procedures for setting environment variables and mounting configurations. Remember that incorrect handling of secrets and environment variables can lead to security vulnerabilities.

Step 5: Testing and Validation

Testing and validation are nonnegotiables in the transition from a legacy system to a dockerized architecture. Ignoring or cutting corners in this phase jeopardizes the integrity of the system, often culminating in performance bottlenecks, functional inconsistencies, or security vulnerabilities. The CRM system, being business-critical, demands meticulous validation.

The most basic level of validation is functional testing to ensure feature parity with the legacy system. Automated tools like Selenium [9] for web UI testing or Postman [10] for API testing offer this capability. Running a test suite against both the legacy and dockerized environments verifies consistent behavior. For example, to run Selenium tests in a Docker container, you would type a command similar to the following:

```
docker run --net=host selenium/➥
  standalone-chrome python ➥
  my_test_script.py
```

Once functionality is confirmed, performance metrics such as latency, throughput, and resource utilization must be gauged using tools like Apache JMeter, Gatling, or custom scripts. You should also simulate extreme conditions to validate the system's reliability under strain.
Static application security testing (SAST) and dynamic application security testing (DAST) should also be employed. Tools like OWASP ZAP can be dockerized and incorporated into the testing pipeline for dynamic testing. While testing, activate monitoring solutions like Prometheus and Grafana or ELK stack for real-time metrics and logs. These tools will identify potential bottlenecks or security vulnerabilities dynamically.
Despite rigorous testing, unforeseen issues might surface post-deployment. Therefore, formulate a rollback strategy beforehand. Container orchestration systems, such as Kubernetes and Swarm, provide the ability to easily rollout changes and rollback when issues occur.

Step 6: Deployment

Deployment into a production environment is the final phase of dockerizing a legacy CRM system. The delivery method will depend on the application and your role as a developer. Many containerized applications reside today in application repositories, including Docker's own Docker Hub container image library. If you are deploying the application within your own infrastructure, you will likely opt for a container orchestration solution already in use, such as Kubernetes.

Conclusion

Containerization offers many technical benefits, including uniformity, security, and better scaling. In addition, containerizing your apps can save you money with more efficient testing and rollout, and a container strategy can minimize the need for continual customization to adapt to new hardware and software settings. Docker Compose and other tools in the Docker toolset provide a safe, efficient, and versatile approach for migrating your existing applications to a container environment. ■

*This article was made possible by support from Docker through Linux New Media's Topic Subsidy Program (**https://www.linuxnewmedia.com/Topic_Subsidy**).*

Info

[1] RFC 1178: Choosing a Name for your Computer: [https://datatracker.ietf.org/doc/html/rfc1178]
[2] Install Docker Engine: [https://docs.docker.com/engine/install/]
[3] Docker Compose: [https://docs.docker.com/compose/]
[4] GitLab: Using PostgreSQL: [https://docs.gitlab.com/ee/ci/services/postgres.html]
[5] GitLab: Using MySQL: [https://docs.gitlab.com/ee/ci/services/mysql.html]
[6] Docker Scout: [https://docs.docker.com/scout/]
[7] Clair: [https://github.com/quay/clair]
[8] mysqldump: [https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html]
[9] Selenium: [https://www.selenium.dev/]
[10] Postman: [https://www.postman.com/automated-testing/]

Author

Artur Skura is a senior DevOps engineer currently working for a leading pharmaceutical company based in Switzerland. Together with a team of experienced engineers, he builds and maintains cloud infrastructure for large data science and machine learning operations. In his free time, he composes synth folk music, combining the vibrant sound of the '80s with folk themes.