Week 12.3

Actionable Docker

If you're aiming to develop a thorough understanding of Docker and delve into its extensive features, I strongly suggest watching the detailed Docker series available on YouTube, which is split into part 1 and part 2. These tutorials are designed to provide you with an in-depth exploration of Docker, covering a wide range of topics from the basics to more advanced features.

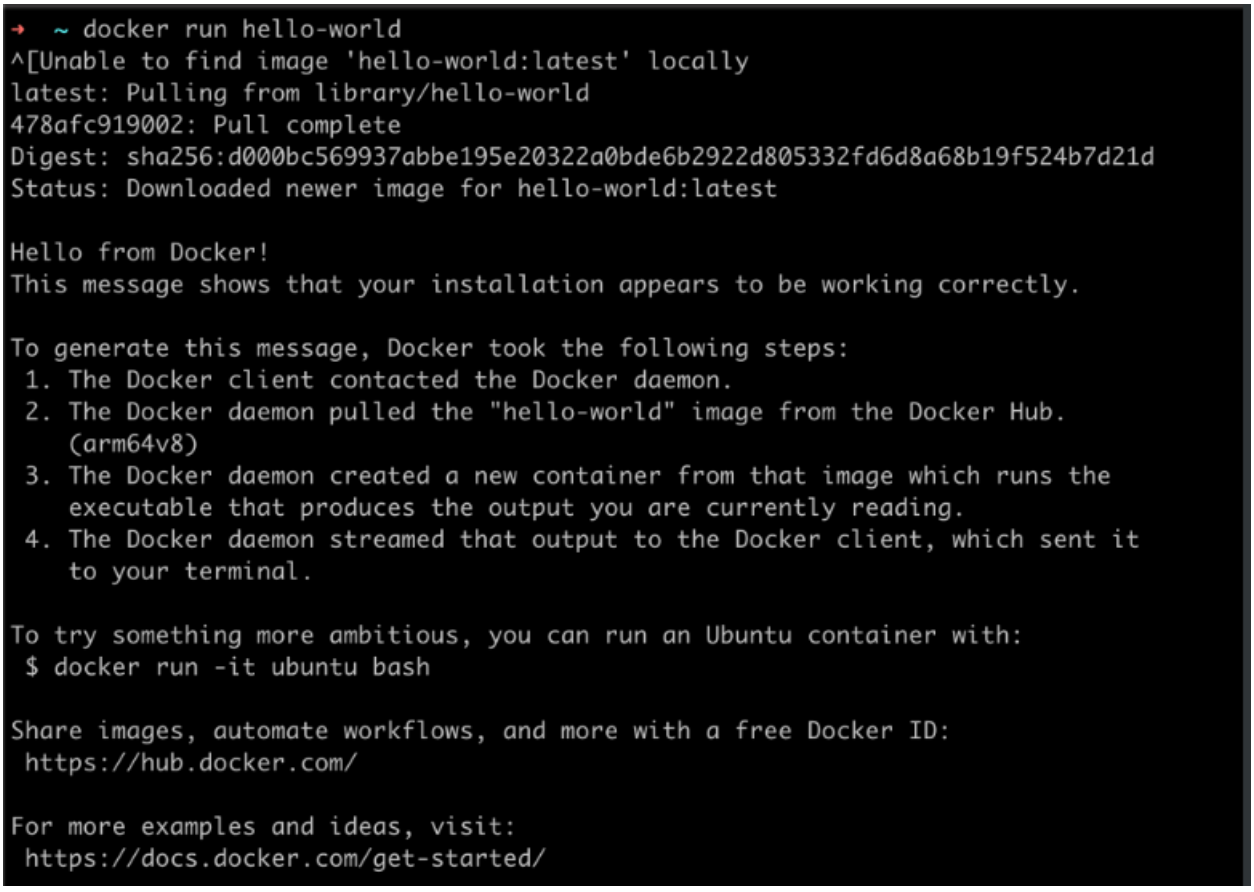
Today's session, however, is focused on providing a short, actionable introduction to Docker. This will help you get started with running packages locally and give you a taste of what Docker can do.

Installation

Docker can be installed using the Docker GUI, which simplifies the setup process. Detailed instructions for various operating systems can be found on the official Docker documentation website at https://docs.docker.com/engine/install/.

Verification

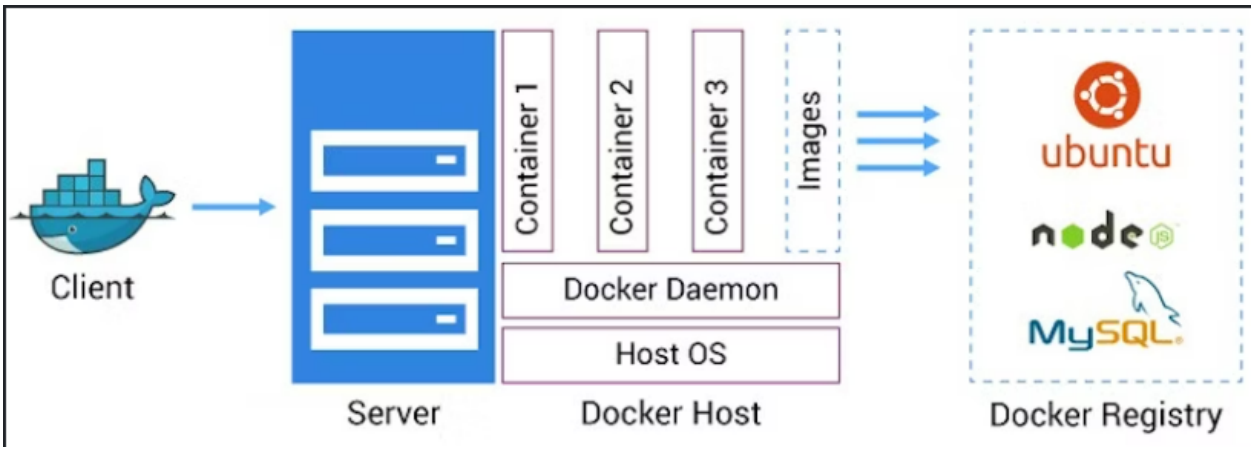
After installation, you should verify that Docker is installed correctly by running the `docker run hello-world` command. This command will pull the "hello-world" image from Docker Hub and run it in a new container, which should print a message to the terminal.



Why Docker?

Docker is a powerful platform that serves several purposes in the development, deployment, and running of applications. Below are the reasons why it is used:

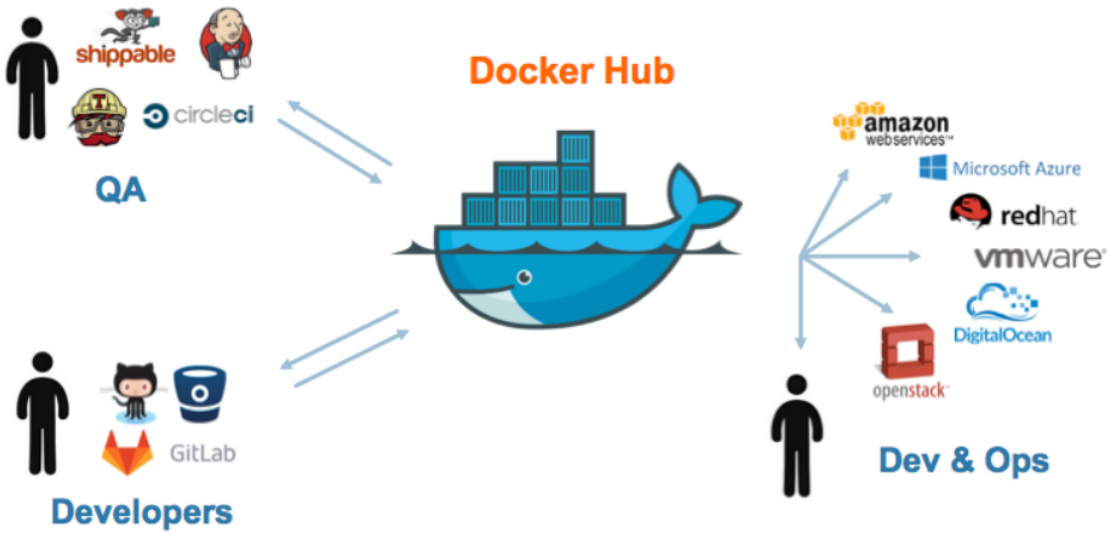
- **Containerization of Applications**: Docker allows you to package your application and its dependencies into a container, which is a lightweight, standalone, and executable software package. This containerization ensures that the application runs consistently across different computing environments, from development to staging to production.
- **Running Other People's Code and Packages**: With Docker, you can easily run software and applications built by others without worrying about setting up the required environment or dependencies. This is because all the necessary components are included within the container.
- **Running Common Software Packages**: Docker provides the ability to run common software packages, such as databases (MongoDB, PostgreSQL, etc.), within containers. This means you can quickly deploy and manage these services without the need to install and configure them directly on your host machine.



> The use of Docker streamlines the development process, simplifies deployment, and enhances the scalability and portability of applications. It isolates applications in containers, making it easier to manage dependencies and avoid conflicts between different software running on the same system.

Docker Hub

Docker registries are similar to version control repositories for code, such as GitHub or GitLab, but instead of code, they store Docker images. Docker images are the blueprints for creating Docker containers, which include the application and all of its dependencies.



Docker Hub is the default registry for Docker and is analogous to GitHub in the context of Docker images. It's a cloud-based repository where users can sign up for an account, push their custom images, pull images published by others, and

work with automated build workflows. Here's the relevance of Docker Hub in comparison to GitHub:

- **Version Control and Collaboration**: Just as GitHub allows developers to store, version, and collaborate on code, Docker Hub provides similar functionalities for Docker images. Users can keep track of different versions of their images, collaborate with team members, and integrate with continuous integration/continuous deployment (CI/CD) pipelines.
- **Public and Private Repositories**: Both platforms offer the ability to have public repositories, where anyone can access and use the resources, and private repositories, which are restricted to authorized users.
- **Community and Official Images**: Docker Hub hosts a vast collection of community-generated images, similar to how GitHub hosts open-source projects. It also provides official images maintained by software vendors or the Docker team, ensuring a trusted source of commonly used software packages.
- **Automated Builds**: Docker Hub can automatically build images from source code in a repository when changes are made, similar to how CI/CD systems work with GitHub to automate the testing and deployment of code.

> In summary, Docker Hub is a central repository for Docker images, where users can store, manage, and distribute their containerized applications. It plays a crucial role in the Docker ecosystem, facilitating the sharing and deployment of software in a manner that's consistent with how code is managed on platforms like GitHub.

Common Commands

The commands listed are part of the basic Docker CLI (Command Line Interface) operations that allow you to manage Docker containers. Here's an explanation of each command, along with an analogy to help understand their functions:

- `docker run`: This command is used to create and start a Docker container from a specified image. It's like saying "start this application" in the Docker world. For example, `docker run mongo` starts a MongoDB container using the official MongoDB image from Docker Hub.
- `docker ps`: This command lists all currently running containers, much like the `ps` command in Unix-based systems that shows running processes. It's like looking at a list of active applications on your computer.
- `docker kill <container_id>`: This command stops a running container immediately. It's similar to force-quitting an application on your computer.

Now, let's delve into the specific scenarios mentioned:

- **Running a simple image**: When you run `docker run mongo`, you're starting a MongoDB container. However, without port mapping, you won't be able to access the MongoDB instance from your host machine.
- **Adding a port mapping**: By using `docker run -p 27017:27017 mongo`, you map the default MongoDB port (27017) from the container to the host. This is like setting up a direct phone line to a specific office in a large building; the port number is the extension number.
- **Starting in detached mode**: The `d` flag starts the container in the background (detached mode), allowing you to continue using the terminal. It's like putting a program to run in the background on your computer so you can do other tasks.
- **Inspecting a container with `docker ps`**: This shows you all the containers that are currently running, providing details such as container ID, image used, command executed, creation time, status, and ports.
- **Stopping a container with `docker kill`**: When you want to stop a container, you use `docker kill` followed by the container ID. This is like using the "End Task"

feature in a task manager to stop a program.

In summary, the flow of commands for running a MongoDB container with port mapping in detached mode and then inspecting and stopping it would be:

1. `docker run -d -p 27017:27017 mongo` (Run MongoDB in detached mode with port mapping)
2. `docker ps` (Inspect running containers)
3. `docker kill <container_id>` (Stop the specified container)

> These commands provide a basic workflow for managing Docker containers, from starting them to making them accessible and finally stopping them when they're no longer needed.

Common Packages

To better understand the use of Docker for running database services, let's consider the MongoDB and PostgreSQL packages. Docker allows you to run these databases in containers, which are isolated environments that contain everything the software needs to run.

- **MongoDB**:

  ```
  docker run -d -p 27017:27017 mongo
  ```

  This command runs a MongoDB container in detached mode (`-d`), which means it runs in the background. The `-p 27017:27017` option maps the default

MongoDB port inside the container (27017) to the same port on the host machine, allowing you to connect to MongoDB from your local machine as if it were running natively.

- **PostgreSQL**:

  ```
  docker run -e POSTGRES_PASSWORD=mysecretpassword -d -p 543
  2:5432 postgres
  ```

  This command runs a PostgreSQL container with a specified environment variable (`-e`) setting the default user's password to "mysecretpassword". It also runs in detached mode and maps the default PostgreSQL port (5432) from the container to the host. The connection string provided is used to connect to this PostgreSQL instance from your local machine.

  The connection string `postgresql://postgres:mysecretpassword@localhost:5432/postgres` is used to connect to the PostgreSQL server from your local machine. It includes the username, password, host, port, and database name.

Below is a simple Node.js script to test the connection to the PostgreSQL database running in the Docker container:

```
// Import the pg library
const { Client } = require('pg');

// Define your connection string (replace placeholders with y
our actual data)
const connectionString = 'postgresql://postgres:mysecretpassw
ord@localhost:5432/postgres';

// Create a new client instance with the connection string
const client = new Client({
  connectionString: connectionString
```

```
});

// Connect to the database
client.connect(err => {
  if (err) {
    console.error('connection error', err.stack);
  } else {
    console.log('connected to the database');
  }
});

// Run a simple query (Example: Fetching the current date and
time from PostgreSQL)
client.query('SELECT NOW()', (err, res) => {
  if (err) {
    console.error(err);
  } else {
    console.log(res.rows[0]);
  }

  // Close the connection
  client.end();
});
```

The Node.js code snippet provided is an example of how to use the `pg` library to connect to the PostgreSQL server running in the Docker container. It creates a new client with the connection string, connects to the database, runs a query to fetch the current date and time, and then closes the connection.

This script does the following:

1. Imports the `pg` library, which is a PostgreSQL client for Node.js.

2. Defines the connection string using the credentials and port specified in the `docker run` command for PostgreSQL.
3. Creates a new client instance with the connection string.
4. Connects to the PostgreSQL database.
5. Runs a query to fetch the current date and time from the database.
6. Logs the result of the query or an error if the connection or query fails.
7. Closes the database connection.

> The images provided in the search results show the output of running the `docker run hello-world` command, which is a test to ensure that Docker is installed and running correctly on your system. This command is not directly related to the MongoDB and PostgreSQL commands or the Node.js code snippet.

Using Docker to run these databases in containers simplifies the setup and ensures that the software runs the same way on any machine, regardless of the host environment. This is because the container includes the database software and all of its dependencies, configured exactly as needed to run.